Unit I

1

ABSTRACT DATA TYPES

Unit Structure

1.0 OBJECTIVE

In this chapter we are going to learn.

- ➔ the concept of abstract data types (ADTs) for both simple types, those containing individual data fields, and the more complex types, those containing data structures.
- ➔ ADTs definition, use, and implementation.
- ➔ the importance of abstraction,
- ➔ several ADTs and then how a well-defined ADT can be used without knowing how its actually implemented.
- ➔ the implementation of the ADTs with an emphasis placed on the importance of selecting an appropriate data structure.

The chapter includes an introduction to the Python iterator mechanism and provides an example of a user-defined iterator for use with a container type ADT.

1.1 INTRODUCTION

- Data items are represented within a computer as a sequence of binary digits.
- These sequences can appear very similar but have different meanings since computers can store and manipulate different types of data.
- For example, the binary sequence 01001100110010110101110011011100 could be a string of characters, an integer value, or a real value.
- To distinguish between the different types of data, the term **type** is often used to refer to a collection of values and the term **data type** to refer to a given type along with a collection of operations for manipulating values of the given type.
- Programming languages commonly provide data types as part of the language itself.
- **These data types, known as primitives, come in two categories:**

  Simple and complex.
- The **simple data types** consist of values that are in the most basic form and cannot be decomposed into smaller parts.

  Integer and real types, for example, consist of single numeric values.
- The **complex data types**, on the other hand, are constructed of multiple components consisting of simple types or other complex types.

  In Python, objects, strings, lists, and dictionaries, which can contain multiple values, are all examples of complex types.
- The **primitive types** provided by a language may not be sufficient for solving large complex problems.
- Thus, most languages allow for the construction of additional data types, known as **user-defined types** since they are defined by the programmer and not the language.
- Some of these data types can themselves be very complex.

1.1.1 Abstractions

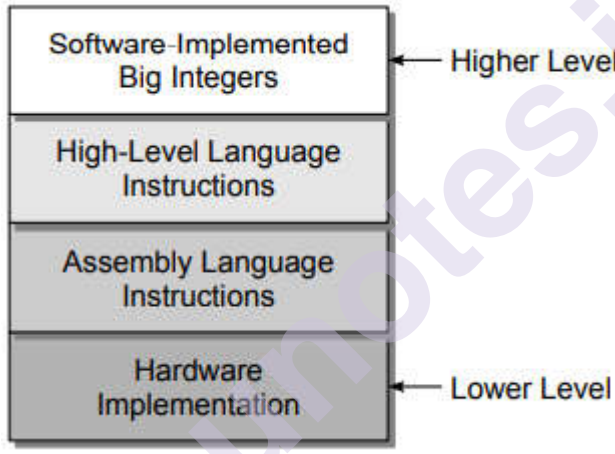
- An abstraction is a mechanism for separating the properties of an object and restricting the focus to those relevant in the current context.
- The user of the abstraction does not have to understand all of the details in order to utilize the object, but only those relevant to the current task or problem.

- Two common types of abstractions encountered in computer science are

  - **procedural, or functional, abstraction and**
  - **data abstraction.**

    - **Procedural abstraction** is the use of a function or method knowing what it does but ignoring how it's accomplished.
    - **Data abstraction** is the separation of the properties of a data type (its values and operations) from the implementation of that data type.

    - Consider the problem of representing integer values on computers and performing arithmetic operations on those values.
    - Following Figure illustrates the common levels of abstractions used with integer arithmetic.



Levels of abstraction used with integer arithmetic.

- At **the lowest level is the hardware** with little to no abstraction since it includes binary representations of the values and logic circuits for performing the arithmetic.
- Hardware designers would deal with integer arithmetic at this level and be concerned with its correct implementation.
- A higher level of abstraction for integer values and arithmetic is provided through assembly language, which involves working with binary values and individual instructions corresponding to the underlying hardware.
- Compiler writers and assembly language programmers would work with integer arithmetic at this level and must ensure the proper selection of assembly language instructions to compute a given mathematical expression.
- For example, suppose we wish to compute x = a + b − 5.

- At **the assembly language level**, this expression must be split into multiple instructions for loading the values from memory, storing them into registers, and then performing each arithmetic operation separately, as shown in the following psuedocode:

  loadFromMem(R1, 'a')

  loadFromMem(R2, 'b')

  add R0, R1, R2

  sub R0, R0, 5

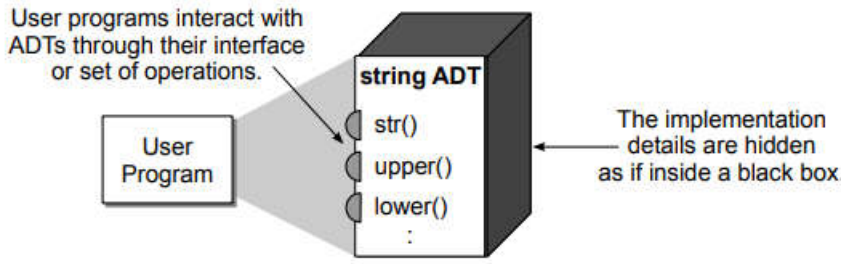
  storeToMem(R0, 'x')

- To avoid this level of complexity, **high-level programming languages** add another layer of abstraction above the assembly language level.
- This abstraction is provided through a primitive data type for storing integer values and a set of well-defined operations that can be performed on those values.
- Example: mathematical expressions like (x = a + b − 5) is possible with assembly language instructions.
- One problem with the integer arithmetic provided by most high-level languages and in computer hardware is that it works with values of a limited size.
- In this case, we can provide long or "big integers" implemented in **software** to allow values of unlimited size.
- This would involve storing the individual digits and implementing functions or methods for performing the various arithmetic operations.
- The implementation of the operations would use the primitive data types and instructions provided by the high-level language.
- Software libraries that provide big integer implementations are available for most common programming languages.

1.1.2 Abstract Data Types

- **An abstract data type (or ADT)** is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values.
- Abstract data types are defined independent of their implementation, allowing us to focus on the use of the new data type instead of how it's implemented.
- This separation is typically enforced by requiring interaction with the abstract data type through an interface or defined set of operations.This is known as information hiding.

- By hiding the implementation details and requiring ADTs to be accessed through an interface, we can work with an abstraction and focus on what functionality the ADT provides instead of how that functionality is implemented.
- Abstract data types can be viewed like black boxes as illustrated in following Figure:



Separating the ADT definition from its implementation.

- User programs interact with instances of the ADT by invoking one of the several operations defined by its interface.
- The set of operations can be grouped into four categories:

  Constructors: Create and initialize new instances of the ADT.

  Accessors: Return data contained in an instance without modifying it.

  Mutators: Modify the contents of an ADT instance.

  Iterators: Process individual data components sequentially.

1.1.3 Data Structures

- Abstract data types can be **simple or complex.**
- **A simple ADT** is composed of a single or several individually named data fields such as those used to represent a date or rational number.
- **The complex ADTs** are composed of a collection of data values such as the Python list or dictionary.
- Complex abstract data types are implemented using a particular **data structure**, which is the physical representation of how data is organized and manipulated.
- **Data structures** can be characterized by how they store and organize the individual data elements and what operations are available for accessing and manipulating the data.
- There are many common data structures, including arrays, linked lists, stacks, queues, and trees, to name a few.

- All data structures store a collection of values, but differ in how they organize the individual data items and by what operations can be applied to manage the collection.
- The choice of a particular data structure depends on the ADT and the problem at hand. Some data structures are better suited to particular problems.
- For example, the queue structure is perfect for implementing a printer queue, while the B-Tree is the better choice for a database index.

1.1.4 General Definitions:

We define some of the common terms we will be using throughout the text in the following table:

Term	Definition
Collection	A **collection** is a group of values with no implied organization or relationship between the individual values. Sometimes we may restrict the elements to a specific data type such as a collection of integers or floating-point values.
Container	A **container** is any data structure or abstract data type that stores and organizes a collection.
Elements	The individual values of the collection are known as **elements** of the container.
Empty	A container with no elements is said to be **empty**.
Sequence	A **sequence** is a container in which the elements are arranged in linear order from front to back, with each element accessible by position
Sorted Sequence	A **sorted sequence** is one in which the position of the elements is based on a prescribed relationship between each element and its successor. For example, we can create a sorted sequence of integers in which the elements are arranged in ascending or increasing order from smallest to largest value
List	The term **list** to refer to the data type provided by Python and
General List Or List Structure	The terms **general list or list structure** when referring to the more general list structure as defined earlier.

1.2 THE DATE ABSTRACT DATA TYPE

An abstract data type is defined by specifying the domain of the data elements that compose the ADT and the set of operations that can be performed on that domain.

Next, we provide the definition of a simple abstract data type for representing a date in the proleptic Gregorian calendar.

1.2.1 Defining the ADT

- The Gregorian calendar was introduced in the year 1582 by Pope Gregory XIII to replace the Julian calendar.
- The new calendar corrected for the miscalculation of the lunar year and introduced the leap year.
- The official first date of the Gregorian calendar is Friday, October 15, 1582.

Definition:

A date represents a single day in the proleptic Gregorian calendar in which the first day starts on November 24, 4713 BC.

Methos	Description
Date(month, day, year):	➢ Creates a new Date instance initialized to the given Gregorian date which must be valid. ➢ Year 1 BC and earlier are indicated by negative year components.
day():	➢ Returns the Gregorian day number of this date.
month():	➢ Returns the Gregorian month number of this date.
year():	➢ Returns the Gregorian year of this date.
monthName():	➢ Returns the Gregorian month name of this date.

dayOfWeek():	➢ Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.
numDays(otherDate):	➢ Returns the number of days as a positive integer between this date and the otherDate.
isLeapYear():	➢ Determines if this date falls in a leap year and returns the appropriate boolean value.
advanceBy(days):	➢ Advances the date by the given number of days. ➢ The date is incremented if days are positive and decremented if days are negative. ➢ The date is capped to November 24, 4714 BC, if necessary.
comparable (otherDate):	➢ Compare this date to the otherDate to determine their logical ordering. ➢ This comparison can be done using any of the logical operators <, <=, >, >=, ==, !=.
toString ():	➢ Returns a string representing the Gregorian date in the format mm/dd/yyyy. ➢ Implemented as the Python operator that is automatically called via the str() constructor

1.2.2 Preconditions and Postconditions

- **Preconditions:**

  A precondition indicates the condition or state of the ADT instance and inputs before the operation can be performed.

- **Postconditions:**

  A postcondition indicates the result or ending state of the ADT instance after the operation is performed.

- The precondition is assumed to be true while the postcondition is a guarantee as long as the preconditions are met.

- Attempting to perform an operation in which the precondition is not satisfied should be flagged as an error.
- **Example:**

  Consider the use of the **pop(i)** method for removing a value from a list.

  - When this method is called, the precondition states the supplied index must be within the legal range.
  - Upon successful completion of the operation, the postcondition guarantees the item has been removed from the list.
  - If an invalid index, one that is out of the legal range, is passed to the **pop()** method, an exception is raised.
- All operations have at least one precondition, which is that the ADT instance has to have been previously initialized.
- When implementing abstract data types, it's important that we ensure the proper execution of the various operations by verifying any stated preconditions.
- The appropriate mechanism when testing preconditions for abstract data types is to test the precondition and raise an exception when the precondition fails.

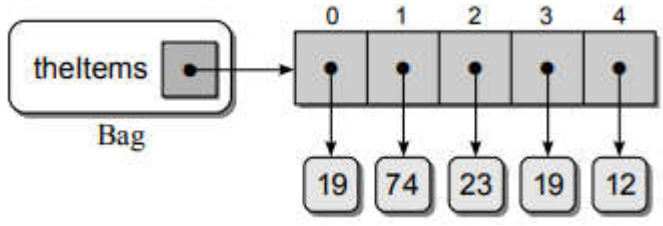
1.3 BAGS

- The Date ADT provided an example of a simple abstract data type.
- To illustrate the design and implementation of a complex abstract data type, we define the Bag ADT.
- **A bag** is a simple container like a shopping bag that can be used to store a collection of items.

> **Definition:**
>
> A bag is a container that stores a collection in which duplicate values are allowed.
>
> The items, each of which is individually stored, have no particular order but they must be comparable.

Method	Description
Bag():	➢ Creates a bag that is initially empty.
length ():	➢ Returns the number of items stored in the bag. Accessed using the len () function.
contains (item):	➢ Determines if the given target item is stored in the bag and returns the appropriate Boolean value. ➢ Accessed using the in operator.
add(item):	➢ Add the given item to the bag.
remove(item):	➢ Removes and returns an occurrence of an item from the bag. An exception is raised if the element is not in the bag.
iterator ():	➢ Creates and returns an iterator that can be used to iterate over the collection of items.

- Bags are containers – they hold things
- Fundamental operations all bag objects should provide:
  - Put something in
  - Take an item out
  - Take everything out
  - Count how many things are in it
  - See if it is empty
  - Check to see if a particular item is in it
  - Count the number of items in it
  - Look at all the contents
- A list stores references to objects and technically would be illustrated as shown in the figure to the right.
- To conserve space and reduce the clutter that can result in some figures, however, we illustrate objects in the text as boxes with rounded edges and show them stored directly within the list structure.
- Variables will be illustrated as square boxes with a bullet in the middle and the name of the variable printed nearby.



1.4 ITERATORS

- Traversals are very common operations, especially on containers.
- A traversal iterates over the entire collection, providing access to each individual element.
- Traversals can be used for a number of operations, including searching for a specific item or printing an entire collection.
- An iterator is an object that provides a mechanism for performing generic traversals through a container without having to expose the underlying implementation.
- Iterators are used with Python's for loop construct to provide a traversal mechanism for both built-in and user-defined containers.
- Example:

```
# Iterate over the bag and check the ages.
for date in bag :
  if date <= bornBefore :
    print( "Is at least 21 years of age: ", date )
```

- To use Python's traversal mechanism with our own abstract data types, we must define an iterator class, which is a class in Python containing two special methods,

  _ _ iter_ _ and _ _ next.

- Iterator classes are commonly defined in the same module as the corresponding container class.
- Example:

```
# An iterator for the Bag ADT implemented as a Python list.
class _BagIterator :
  def __init__( self, theList ):
    self._bagItems = theList
    self._curItem = 0

  def __iter__( self ):
    return self

  def __next__( self ):
    if self._curItem < len( self._bagItems ) :
      item = self._bagItems[ self._curItem ]
      self._curItem += 1
      return item
    else :
      raise StopIteration
```

1.5 APPLICATION: STUDENT RECORDS

- Most computer applications are written to process and manipulate data that is stored external to the program.
- Data is commonly extracted from files stored on disk, from databases, and even from remote sites through web services.
- For example, suppose we have a collection of records stored on disk that contain information related to students at Smalltown College.
- We have been assigned the task to extract this information and produce a report similar to the following in which the records are sorted by identification number.

LIST OF STUDENTS

ID	NAME	CLASS	GPA
10015	Smith, John	Sophomore	3.01
10167	Jones, Wendy	Junior	2.85
10175	Smith, Jane	Senior	3.92
10188	Wales, Sam	Senior	3.25
10200	Roberts, Sally	Freshman	4.00
10208	Green, Patrick	Freshman	3.95
10226	Nelson, Amy	Sophomore	2.95
10334	Roberts, Jane	Senior	3.81
10387	Taylor, Susan	Sophomore	2.15
10400	Logan, Mark	Junior	3.33
10485	Brown, Jessica	Sophomore	2.91

Number of students: 11

- Our contact in the Registrar's office, who assigned the task, has provided some information about the data.
- We know each record contains five pieces of information for an individual student:

  (1) the student's id number represented as an integer;

  (2) their first and last names, which are strings;

  (3) an integer classification code in the range [1 . . . 4] that indicates if the student is a freshman, sophomore, junior, or senior; and

  (4) their current grade point average represented as a floatingpoint value.

- The data could be stored in a plain text file, in a binary file, or even in a database.

- In addition, if the data is stored in a text or binary file, we will need to know how the data is formatted in the file, and if it's in a relational database, we will need to know the type and the structure of the database.

Definition Student File Reader ADT

A student file reader is used to extract student records from external storage.

The five data components of the individual records are extracted and stored in a storage object specific for this collection of student records.

Student File Reader (filename):	➢ Creates a student reader instance for extracting student records from the given file. ➢ The type and format of the file is dependent on the specific implementation.
open():	➢ Opens a connection to the input source and prepares it for extracting student records. ➢ If a connection cannot be opened, an exception is raised.
close():	➢ Closes the connection to the input source. ➢ If the connection is not currently open, an exception is raised.
fetchRecord():	➢ Extracts the next student record from the input source and returns a reference to a storage object containing the data. ➢ None is returned when there are no additional records to be extracted. ➢ An exception is raised if the connection to the input source was previously closed
fetchAll():	➢ The same as fetch Record(), but extracts all student records (or those remaining) from the input source and returns them in a Python list.

1.6 EXERCISE

Answer the following:

1. Define ADT (Abstract Data Type). Mention the features of ADT.What are the benefits of ADT?
2. Explain the various operations of the list ADT with examples

Reference Book:

Data Structure and Algorithm Using Python, Rance D. Necaise, 2016 Wiley India Edition

❖❖❖❖

2

ARRAYS

Unit Structure

2.0 OBJECTIVE

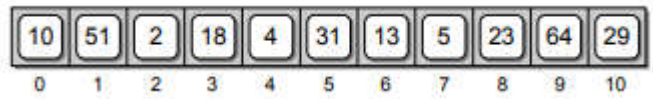
- Introduces the student to the array structure, which is important since Python only provides the list structure and students are unlikely to have seen the concept of the array as a fixed-sized structure in a first course using Python.
- We define an ADT for a one-dimensional array and implement it using a hardware array provided through a special mechanism of the C-implemented version of Python.
- The two-dimensional array is also introduced and implemented using a 1-D array of arrays.
- The array structures will be used throughout the text in place of the Python's list when it is the appropriate choice.
- The implementation of the list structure provided by Python is presented to show how the various operations are implemented using a 1-D array.
- The Matrix ADT is introduced and includes an implementation using a two-dimensional array that exposes the students to an example of an ADT that is best implemented using a structure other than the list or dictionary.
- The most basic structure for storing and accessing a collection of data is the array.
- Arrays can be used to solve a wide range of problems in computer science. Most programming languages provide this structured data type as a primitive and allow for the creation of arrays with multiple dimensions.

- In this chapter, we implement an array structure for a one-dimensional array and then use it to implement a two-dimensional array and the related matrix structure.

2.1 ARRAY STRUCTURE

At the hardware level, most computer architectures provide a mechanism for creating and using one-dimensional arrays.

A one-dimensional array, as illustrated in following Figure, is composed of multiple sequential elements stored in contiguous bytes of memory and allows for random access to the individual elements.



The entire contents of an array are identified by a single name.

Individual elements within the array can be accessed directly by specifying an integer subscript or index value, which indicates an offset from the start of the array.

This is similar to the mathematics notation (x_i), which allows for multiple variables of the same name.

The difference is that programming languages typically use square brackets following the array name to specify the subscript, x[i].

The array is best suited for problems requiring a sequence in which the maximum number of elements are known up front, whereas the list is the better choice when the size of the sequence needs to change after it has been created.

For example, suppose we need a sequence structure with 100, 000 elements.

We could create a list with the given number of elements using the replication operator: values = [None] * 100000

List	Array
The list can store the value of different types.	It can only consist of value of same type.
The list cannot handle the direct arithmetic operations.	It can directly handle arithmetic operations.
We need to import the array before work with the array.	The lists are the build-in data structure so we don't need to import it.

The lists are less compatible than the array to store the data.	An array are much compatible than the list.
It consumes a large memory.	It is a more compact in memory size comparatively list.
It is suitable for storing the longer sequence of the data item.	It is suitable for storing shorter sequence of data items.
We can print the entire list using explicit looping.	We can print the entire list without using explicit looping.
It can be nested to contain different types of elements.	It must contain either all nested elements of same size.

2.1.1 The Array Abstract Data Type

- The array structure is commonly found in most programming languages as a primitive type, but Python only provides the list structure for creating mutable sequences.
- We can define the Array ADT to represent a one-dimensional array for use in Python that works similarly to arrays found in other languages.
- It will be used throughout the text when an array structure is required.

Definition: Array ADT

A one-dimensional array is a collection of contiguous elements in which individual elements are identified by a unique integer subscript starting with zero.

Once an array is created, its size cannot be changed.

Method	Description
Array(size):	Creates a one-dimensional array consisting of size elements with each element initially set to None. size must be greater than zero.
length ():	Returns the length or number of elements in the array.
getitem (index):	Returns the value stored in the array at element position index. The index argument must be within the valid range. Accessed using the subscript operator.

setitem (index, value):	Modifies the contents of the array element at position index to contain value. The index must be within the valid range. Accessed using the subscript operator.
clearing(value):	Clears the array by setting every element to value.
iterator ():	Creates and returns an iterator that can be used to traverse the elements of the array.

2.1.2 Implementing the Array:

Python is a scripting language built using the C language, a high-level language that requires a program's source code be compiled into executable code before it can be used.

The ctypes Module

➔ While Python does not provide the array structure as part of the language itself, it now includes the ctypes module as part of the Python Standard Library.

➔ The ctypes module provides the capability to create hardware-supported arrays just like the ones used to implement Python's string, list, tuple, and dictionary collection types.

Creating a Hardware Array

➔ The ctypes module provides a technique for creating arrays that can store references to Python objects.

➔ The following code segment

```
import ctypes

ArrayType = ctypes.py_object * 5

slots = ArrayType()
```

➔ creates an array named slots that contains five elements each of which can store a reference to an object.

➔ After the array has been created, the elements can be accessed using the same integer subscript notation as used with Python's own sequence types.

➔ For the slots array, the legal range is [0 . . . 4].

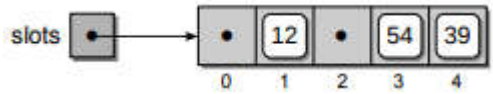
➔ The **range()** function to indicate the number of elements to be initialized.

➔ References to any type of Python object can be stored in any element of the array.

➔ For example, the following code segment stores three integers in various elements of the array:

```
slots[1] = 12
slots[3] = 54
slots[4] = 37
```

the result of which is illustrated here:

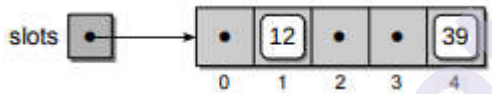


➔ To **remove** an item from the array, we simply set the corresponding element to **None**.

➔ For example, suppose we want to remove value 54 from the array

```
slots[3] = None
```

which results in the following change to the slots array:



➔ The size of the array can never change, so removing an item from an array has no effect on the size of the array or on the items stored in other elements.

➔ The array does not provide any of the list type operations such as appending or popping items, searching for a specific item, or sorting the items.

2.2 PYTHON LIST

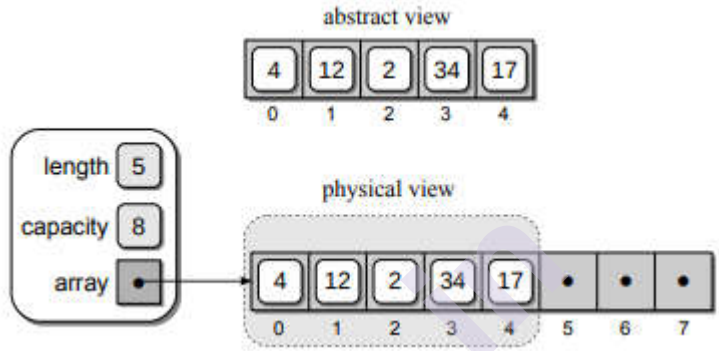
- Python's list structure is a mutable sequence container that can change size as items are added or removed.
- It is an abstract data type that is implemented using an array structure to store the items contained in the list.
- In this section, we examine the implementation of Python's list, which can be very beneficial not only for learning more about abstract data types and their implementations but also to illustrate the major differences between an array and Python's list structure.
- We explore some of the more common list operations and describe how they are implemented using an array structure.

2.2.1 Creating a Python List

- Suppose we create a list containing several values:

  pyList = [4, 12, 2, 34, 17]

- the list() constructor being called to create a list object and fill it with the given values.
- Following Figure illustrates the abstract and physical views of our sample list:



- In the physical view, the elements of the array structure used to store the actual contents of the list are enclosed inside the dashed gray box.
- The elements with null references shown outside the dashed gray box are the remaining elements of the underlying array structure that are still available for use.
- This notation will be used throughout the section to illustrate the contents of the list and the underlying array used to implement it.

2.2.2 Appending Items

- If there is room in the array, the item is stored in the next available slot of the array and the length field is incremented by one.
- The result of appending 50 to pyList is illustrated in Figure

  pyList.append (50)

- For example, consider the following list operations:

  pyList.append(18)

  pyList.append(64)

  pyList.append(6)

- After the second statement is executed, the array becomes full and there is no available space to add more values as illustrated in Figure:

- By definition, a list can contain any number of items and never becomes full.
- Thus, when the third statement is executed, the array will have to be expanded to make room for value 6.
- From the discussion in the previous section, we know an array cannot change size once it has been created.
- Hence To allow for the expansion of the list, the following steps have to be performed:

  (1) a new array is created with additional capacity,

  (2) the items from the original array are copied to the new array,

  (3) the new larger array is set as the data structure for the list, and

  (4) the original smaller array is destroyed.

  - The result of expanding the array and appending value 6 to the list is shown in Figure:

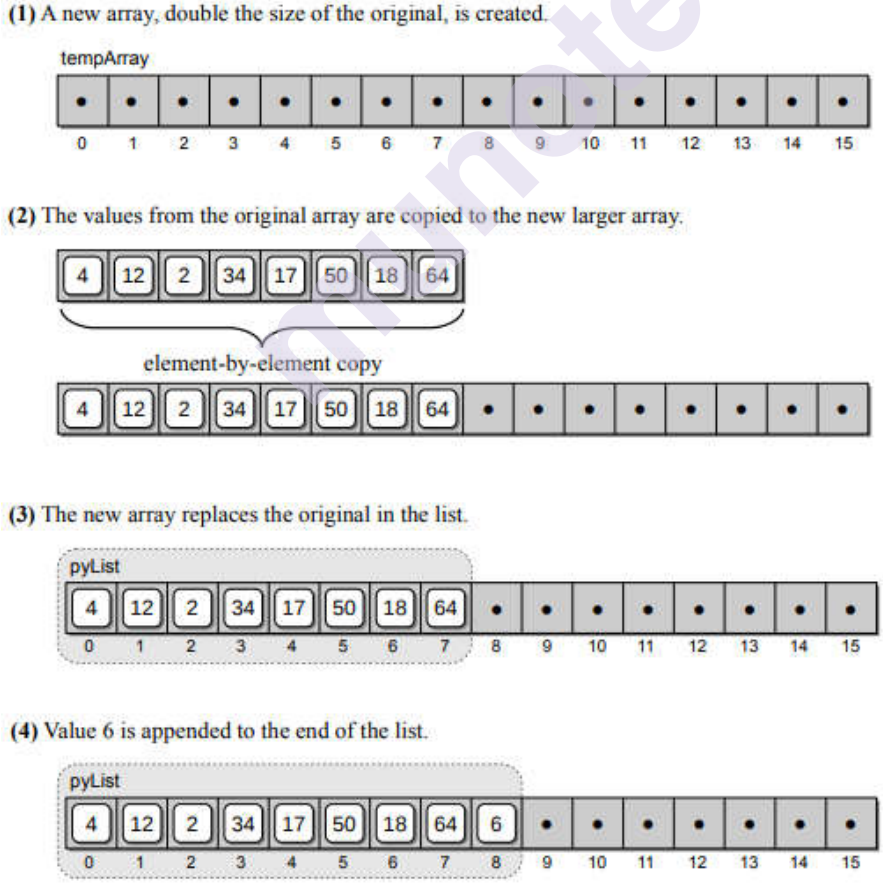


Figure: The steps required to expand the array to provide space for value 6

2.2.3 Extending A List

- A list can be appended to a second list using the extend() method as shown in the following example:

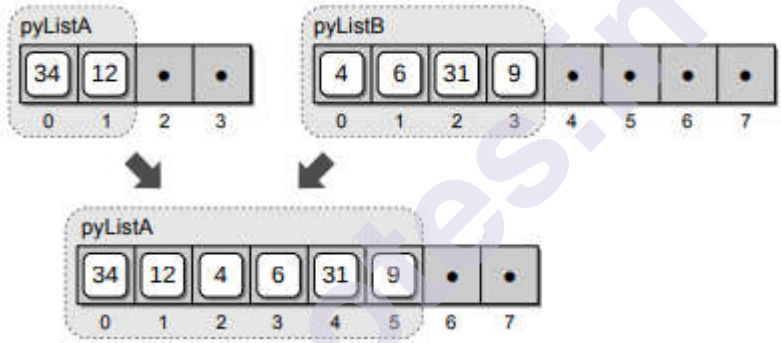
  pyListA = [34, 12]

  pyListB = [4, 6, 31, 9]

  pyListA.extend(pyListB)

- The new array will be created larger than needed to allow more items to be added to the list without first requiring an immediate expansion of the array.

- After the new array is created, elements from the destination list are copied to the new array followed by the elements from the source list, pyListB, as illustrated in Figure:



2.2.4 Inserting Items

- An item can be inserted anywhere within the list using the insert() method.

- In the following example

  pyList.insert(3, 79)

  we insert the value 79 at index position 3.

- Since there is already an item at that position, we must make room for the new item by shifting all of the items down one position starting with the item at index position 3.

- After shifting the items, the value 79 is then inserted at position 3 as illustrated in Figure:

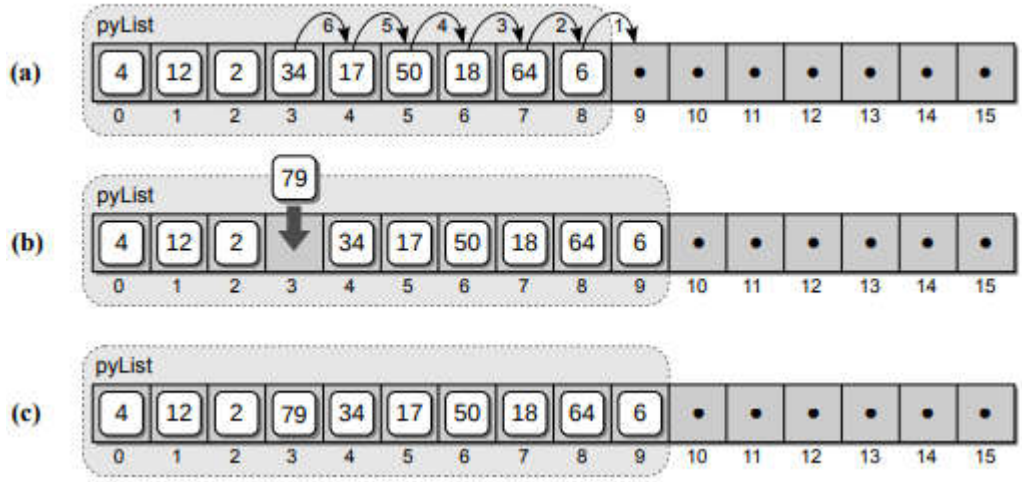


Figure: Inserting an item into a list: (a) the array elements are shifted to the right one at a time, traversing from right to left; (b) the new value is then inserted into the array at the given position; (c) the result after inserting the item.

2.2.5 Removing Items

- An item can be removed from any position within the list using the pop() method.
- Consider the following code segment, which removes both the first and last items from the sample list:

```
pyList.pop( 0 )     # remove the first item
pyList.pop()        # remove the last item
```

- The first statement removes the first item from the list.
- After the item is removed, typically by setting the reference variable to None, the items following it within the array are shifted down, from left to right, to close the gap.
- Finally, the length of the list is decremented to reflect the smaller size.
- Following Figure on the next page illustrates the process of removing the first item from the sample list.
- The second pop() operation in the example code removes the last item from the list

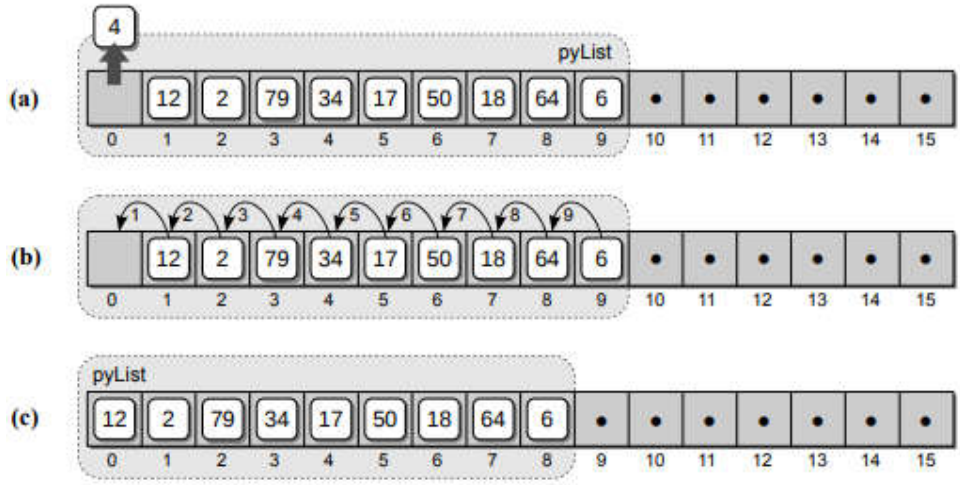


Figure: Removing an item from a list: (a) a copy of the item is saved; (b) the array elements are shifted to the left one at a time, traversing left to right; and (c) the size of the list is decremented by one.

- After removing an item from the list, the size of the array may be reduced using a technique similar to that for expansion.
- This reduction occurs when the number of available slots in the internal array falls below a certain threshold.
- For example, when more than half of the array elements are empty, the size of the array may be cut in half.

2.2.6 List Slice

- Slicing is an operation that creates a new list consisting of a contiguous subset of elements from the original list.
- The original list is not modified by this operation.
- Instead, references to the corresponding elements are copied and stored in the new list.
- In Python, slicing is performed on a list using the colon operator and specifying the beginning element index and the number of elements included in the subset.
- Consider the following example code segment, which creates a slice from our sample list:

```
aSlice = theVector[2:3]
```

- To slice a list, a new list is created with a capacity large enough to store the entire subset of elements plus additional space for future insertions.
- The elements within the specified range are then copied, element by element, to the new list. The result of creating the sample slice is illustrated in Figure:

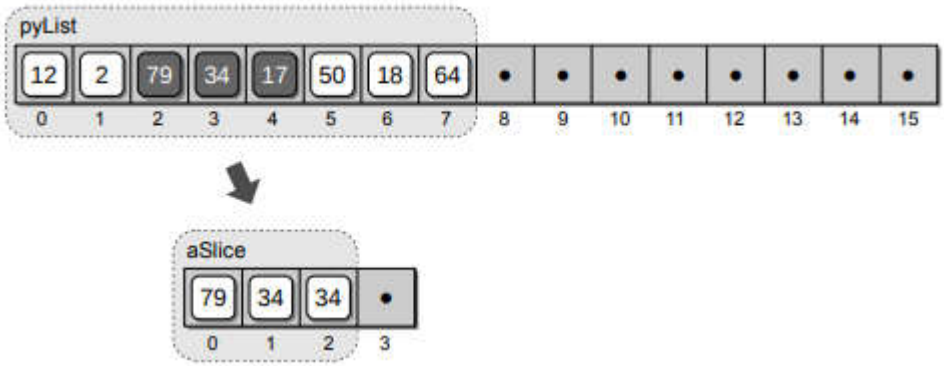


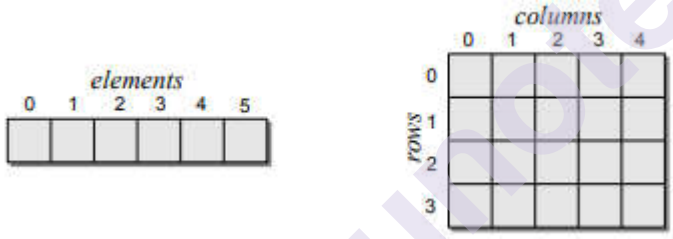
Figure: The result of creating a list slice.

2.3 TWO DIMENSIONAL ARRAYS

The use of a two-dimensional array is to organize data into rows and columns similar to a table or grid.

The individual elements are accessed by specifying two indices, one for the row and one for the column, [i,j].

Following: Figure shows an abstract view of both a one- and a two-dimensional array



2.3.1 The Array2D Abstract Data Type

Two-dimensional arrays are also very common in computer programming, where they are used to solve problems that require data to be organized into rows and columns.

Since 2-D arrays are not provided by Python, we define the Array2D abstract data type for creating 2-D arrays.

It consists of a limited set of operations similar to those provided by the one-dimensional Array ADT.

Definition Array2D ADT

A two-dimensional array consists of a collection of elements organized into rows and columns.

Individual elements are referenced by specifying the specific row and column indices (r, c), both of which start at 0.

Method	Description
Array2D(nrows, ncols):	Creates a two-dimensional array organized into rows and columns. The nrows and ncols arguments indicate the size of the table. The individual elements of the table are initialized to None.
numRows():	Returns the number of rows in the 2-D array.
numCols():	Returns the number of columns in the 2-D array.
clear(value):	Clears the array by setting each element to the given value.
getitem(i1, i2):	Returns the value stored in the 2-D array element at the position indicated by the 2-tuple (i1, i2), both of which must be within the valid range. Accessed using the subscript operator: y = x[1,2].
setitem(i1, i2, value):	Modifies the contents of the 2-D array element indicated by the 2-tuple (i1, i2) with the new value. Both indices must be within the valid range. Accessed using the subscript operator: x[0,3] = y.

2.3.2 Implementing the 2-D Array:

- There are several approaches that we can use to store and organize the data for a 2-D array.
- Two of the more common approaches include :
  - ○ the use of a single 1-D array to physically store the elements of the 2-D array by arranging them in order based on either row or column and
  - ○ the other uses an array of arrays.
- When using an array of arrays to store the elements of a 2-D array, we store each row of the 2-D array within its own 1-D array.
- Following Figure shows the abstract view of a 2-D array and the physical storage of that 2-D array using an array of arrays.:

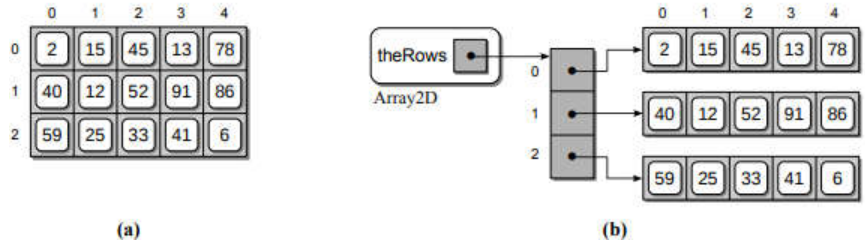


Figure: A sample 2-D array:

(a) the abstract view organized into rows and columns and

(b) the physical storage of the 2-D array using an array of arrays.

- **Basic Operations**

numRows()	The numRows() method can obtain the number of rows by checking the length of the main array, which contains an element for each row in the 2-D array.
numCols()	To determine the number of columns in the 2-D array, the numCols() method can simply check the length of any of the 1-D arrays used to store the individual rows.
clear()	The clear() method can set every element to the given value by calling the clear() method on each of the 1-D arrays used to store the individual rows.

2.4 MATRIX ABSTRACT DATA TYPE

In mathematics, a matrix is an m ×n rectangular grid or table of numerical values divided into m rows and n columns.

Matrices, which are an important tool in areas such as linear algebra and computer graphics, are used in a number of applications, including representing and solving systems of linear equations.

The Matrix ADT is defined next.

Definition Matrix ADT

A matrix is a collection of scalar values arranged in rows and columns as a rectangular grid of a fixed size.

The elements of the matrix can be accessed by specifying a given row and column index with indices starting at 0.

Method	Description
Matrix(rows, ncols):	Creates a new matrix containing nrows and ncols with each element initialized to 0.
numRows():	Returns the number of rows in the matrix.
numCols():	Returns the number of columns in the matrix.
getitem (row, col):	Returns the value stored in the given matrix element. Both row and col must be within the valid range.
setitem (row, col, scalar):	Sets the matrix element at the given row and col to scalar. The element indices must be within the valid range.
scaleBy(scalar):	Multiplies each element of the matrix by the given scalar value. The matrix is modified by this operation.
transpose():	Returns a new matrix that is the transpose of this matrix.
add (rhsMatrix):	Creates and returns a new matrix that is the result of adding this matrix to the given rhsMatrix. The size of the two matrices must be the same.
subtract (rhsMatrix):	The same as the add() operation but subtracts the two matrices.
multiply (rhsMatrix):	Creates and returns a new matrix that is the result of multiplying this matrix to the given rhsMatrix. The two matrices must be of appropriate sizes as defined for matrix multiplication.

2.4.1 Matrix Operations

- **Addition and Subtraction.**
  - Two m × n matrices can be added or subtracted to create a third m × n matrix.
  - When adding two m × n matrices, corresponding elements are summed as illustrated here.

- Subtraction is performed in a similar fashion but the corresponding elements are subtracted instead of summed.
- $$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0+6 & 1+7 \\ 2+8 & 3+9 \\ 4+1 & 5+0 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \\ 5 & 5 \end{bmatrix}$$

- **Scaling.**
  - A matrix can be uniformly scaled, which modifies each element of the matrix by the same scale factor.
  - A scale factor of less than 1 has the effect of reducing the value of each element whereas a scale factor greater than 1 increases the value of each element.
  - Scaling a matrix by a scale factor of 3 is illustrated here:

$$3\begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3*6 & 3*7 \\ 3*8 & 3*9 \\ 3*1 & 3*0 \end{bmatrix} = \begin{bmatrix} 18 & 21 \\ 24 & 27 \\ 3 & 0 \end{bmatrix}$$

- **Transpose**
  - Another useful operation that can be applied to a matrix is the matrix transpose.
  - Given a m × n matrix, a transpose swaps the rows and columns to create a new matrix of size n × m as illustrated here:
  - $$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

- **Multiplication.**
  - Matrix multiplication is only defined for matrices where the number of columns in the matrix on the lefthand side is equal to the number of rows in the matrix on the righthand side.
  - The result is a new matrix that contains the same number of rows as the matrix on the lefthand side and the same number of columns as the matrix on the righthand side.
  - In other words, given a matrix of size m × n multiplied by a matrix of size n × p, the resulting matrix is of size m × p.
  - In multiplying two matrices, each element of the new matrix is the result of summing the product of a row in the lefthand side matrix by a column in the righthand side matrix.

- In the example matrix multiplication illustrated here, the row and column used to compute entry (0, 0) of the new matrix is shaded in gray.

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} * \begin{bmatrix} 6 & 7 & 8 \\ 9 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} (0*6+1*9) & (0*7+1*1) & (0*8+1*0) \\ (2*6+3*9) & (2*7+3*1) & (2*8+3*0) \\ (4*6+5*9) & (4*7+5*1) & (4*8+5*0) \end{bmatrix}$$

$$= \begin{bmatrix} 9 & 1 & 0 \\ 39 & 17 & 16 \\ 69 & 33 & 32 \end{bmatrix}$$

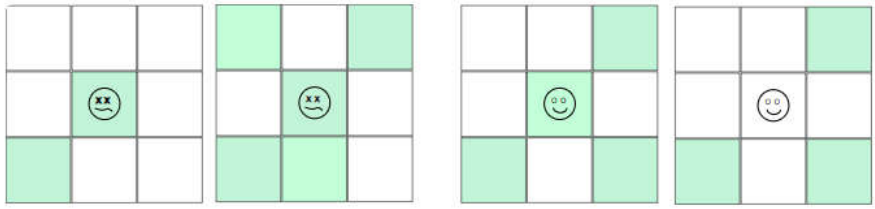
2.5 APPLICATION: THE GAME OF LIFE

The game of Life was an early example of a problem in the modern field of mathematics called cellular automata.

Rules of the Game

- The game uses an infinite-sized rectangular grid of cells in which each cell is either empty or occupied by an organism.
- The occupied cells are said to be alive, whereas the empty ones are dead.
- The status of a cell in the next generation is determined by applying the following four basic rules to each cell in the current configuration:

1. If a cell is alive and has either two or three live neighbors, the cell remains alive in the next generation. The neighbors are the eight cells immediately surrounding a cell: vertically, horizontally, and diagonally.
2. A living cell that has no live neighbors or a single live neighbor dies from isolation in the next generation.
3. A living cell that has four or more live neighbors dies from overpopulation in the next generation.
4. A dead cell with exactly three live neighbors results in a birth and becomes alive in the next generation. All other dead cells remain dead in the next generation.



Cell is about to die in the next time step

Cell will live on or be brought back to live in the next time step

Cell is alive Cell is dead

LIFE RULES

BIRTH RULE: *If at time t a cell is dead (empty), and the cell has 3 live (full) neighbors in any direction, then at time t+1 the cell becomes alive.*
DEATH RULE: *If at time t a live cell has 0 or 1 neighbors it dies of isolation, and if a live cell has 4 or more neighbors it dies of overcrowding.*
SURVIVAL RULE: *If at time t a live cell has 2 or 3 live neighbors, then at time t+1 the cell is still alive.*
With the "Lifeform" shown, the five larger cells comprise the "glider" as it moves along the grid—the dark larger cells are live and remains so in the next iteration, the light larger cells are live but die in the next iteration, and the smaller dots are dead cells that in the next iteration come alive.

- The game of Life requires the use of a grid for storing the organisms.
- A Life Grid ADT can be defined to add a layer of abstraction between the algorithm for "playing" the game and the underlying structure used to store and manipulate the data.

Definition Life Grid ADT

A life grid is used to represent and store the area in the game of Life that contains organisms.

The grid contains a rectangular grouping of cells of a finite size divided into rows and columns.

The individual cells, which can be alive or dead, are referenced by row and column indices, both of which start at zero.

Method	Description
LifeGrid(nrows, ncols):	Creates a new game grid consisting of nrows and ncols. All cells in the grid are set to dead.
numRows():	Returns the number rows in the grid.
numCols():	Returns the number of columns in the grid.
configure(coordList):	Configures the grid for evolving the next generation. The coordList argument is a sequence of 2-tuples with each tuple representing the coordinates (r, c) of the cells to be set as alive. All remaining cells are cleared or set to dead.
clearCell(row, col):	Clears the individual cell (row, col) and sets it to dead. The cell indices must be within the valid range of the grid.
setCell(row, col):	Sets the indicated cell (row, col) to be alive. The cell indices must be within the valid range of the grid.
isLiveCell(row,col):	Returns a boolean value indicating if the given cell (row, col) contains a live organism. The cell indices must be within the valid range of the grid.
numLiveNeighbors(row, col):	Returns the number of live neighbors for the given cell (row, col). The neighbors of a cell include all of the cells immediately surrounding it in all directions. For the cells along the border of the grid, the neighbors that fall outside the grid are assumed to be dead. The cell indices must be within the valid range of the grid

The gameoflife.py program.

```
1 # Program for playing the game of Life.
2 from life import LifeGrid
3
4 # Define the initial configuration of live cells.
5 INIT_CONFIG = [ (1,1), (1,2), (2,2), (3,2) ]
6
7 # Set the size of the grid.
8 GRID_WIDTH = 5
9 GRID_HEIGHT = 5
10
11 # Indicate the number of generations.
12 NUM_GENS = 8
13
14 def main():
15 # Construct the game grid and configure it.
16 grid = LifeGrid( GRID_WIDTH, GRID_HEIGHT )
17 grid.configure( INIT_CONFIG )
18
19 # Play the game.
20 draw( grid )
21 for i in range( NUM_GENS ):
22 evolve( grid )
23 draw( grid )
24
25 # Generates the next generation of organisms.
26 def evolve( grid ):
27 # List for storing the live cells of the next generation.
28 liveCells = list()
29
30 # Iterate over the elements of the grid.
31 for i in range( grid.numRows() ) :
32 for j in range( grid.numCols() ) :
33
```

```
34 # Determine the number of live neighbors for this cell.
35 neighbors = grid.numLiveNeighbors( i, j )
36
37 # Add the (i,j) tuple to liveCells if this cell contains
38 # a live organism in the next generation.
39 if (neighbors == 2 and grid.isLiveCell( i, j )) or \
40 (neighbors == 3 ) :
41 liveCells.append( (i, j) )
42
43 # Reconfigure the grid using the liveCells coord list.
44 grid.configure( liveCells )
45
46 # Prints a text-based representation of the game grid.
47 def draw( grid ):
48 ......
49
50 # Executes the main routine.
51 main()
```

2.6 EXERCISE

Answer the following:

1. Explain different operation performed on List.
2. Explain the application of Array in ADT

Reference:

- Data Structure and algorithm Using Python, Rance D. Necaise, 2016 Wiley India Edition

3

SETS AND MAPS

Unit Structure

3.0 OBJECTIVE

In this chapter we are going to learn about:

- ➢ Both the Set and Map (or dictionary) ADTs.
- ➢ Multi-dimensional arrays (those of two or more dimensions)
- ➢ Concept of physically storing these using a one-dimensional array in either row-major or column-major order.
- ➢ Benefit from the use of a three-dimensional array.

3.1 SETS

- The Set ADT is a common container used in computer science.
- Set is commonly used when you need to store a collection of unique values without regard to how they are stored or when you need to perform various mathematical set operations on collections.

3.1.1 The Set Abstract Data Type

The definition of the set abstract data type is provided here, followed by an implementation using a list.

Definition Set ADT

A set is a container that stores a collection of unique values over a given comparable domain in which the stored values have no particular ordering.

Method	Description
Set():	Creates a new set initialized to the empty set.
length ():	Returns the number of elements in the set, also known as the cardinality. Accessed using the len() function.
contains (element):	Determines if the given value is an element of the set and returns the appropriate boolean value. Accessed using the in operator.
add(element):	Modifies the set by adding the given value or element to the set if the element is not already a member. If the element is not unique, no action is taken and the operation is skipped.
remove(element):	Removes the given value from the set if the value is contained in the set and raises an exception otherwise.
equals (setB):	Determines if the set is equal to another set and returns a boolean value. For two sets, A and B, to be equal, both A and B must contain the same number of elements and all elements in A must also be elements in B. If both sets are empty, the sets are equal. Access with == or !=.
isSubsetOf(setB):	Determines if the set is a subset of another set and returns a boolean value. For set A to be a subset of B, all elements in A must also be elements in B.
union(setB):	Creates and returns a new set that is the union of this set and set B. The new set created from the union of two sets, A and B, contains all elements in A plus those elements in B that are not in A. Neither set A nor set B is modified by this operation.

intersect(setB):	Creates and returns a new set that is the intersection of this set and set B. The intersection of sets A and B contains only those elements that are in both A and B. Neither set A nor set B is modified by this operation.
difference(setB):	Creates and returns a new set that is the difference of this set and set B. The set difference, A−B, contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation.
iterator ():	Creates and returns an iterator that can be used to iterate over the collection of items.

Example:

In the following code segment, we create two sets and add elements to each. The results are illustrated in Figure:

```
smith = Set()
smith.add( "CSCI-112" )
smith.add( "MATH-121" )
smith.add( "HIST-340" )
smith.add( "ECON-101" )
roberts = Set()
roberts.add( "POL-101" )
roberts.add( "ANTH-230" )
roberts.add( "CSCI-112" )
roberts.add( "ECON-101" )
```

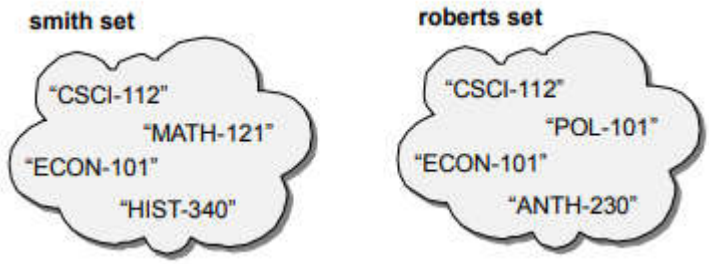


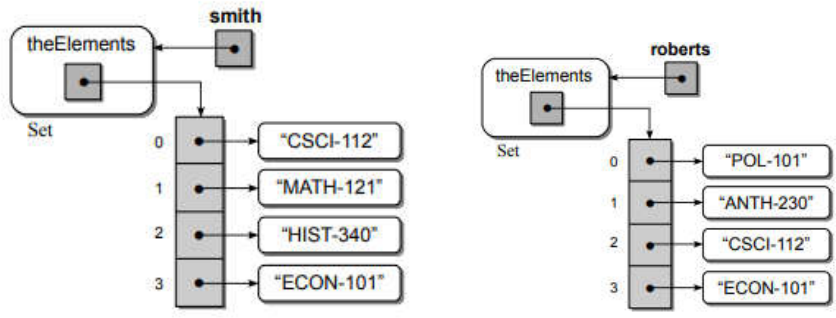
Figure: Abstract view of the two sample sets.

3.1.2 Selecting a Data Structure

- We are trying to replicate the functionality of the set structure provided by Python,which leaves the array, list, and dictionary containers for consideration in implementing the Set ADT.
- The storage requirements for the bag and set are very similar with the difference being that a set cannot contain duplicates.
- The dictionary would seem to be the ideal choice since it can store unique items, but it would waste space in this case.
- The dictionary stores key/value pairs, which requires two data fields per entry.
- An array could be used to implement the set, but a set can contain any number of elements and by definition an array has a fixed size.
- To use the array structure, we would have to manage the expansion of the array when necessary in the same fashion as it's done for the list.
- Since **the list** can grow as needed, it seems ideal for storing the elements of a set just as it was for the bag and it does provide for the complete functionality of the ADT.
- Since the list allows for duplicate values, however, we must make sure as part of the implementation that no duplicates are added to our set.

3.1.3 List-Based Implementation

- Having selected the list structure, we can now implement the Set ADT.
- Some of the operations of the set are very similar to those of the Bag ADT and are implemented in a similar fashion.
- Sample instances for the two sets from Figure (a) are illustrated in Figure (b)



(a) (b)

Figure (a) and (b): Two instances of the Set class implemented as a list.

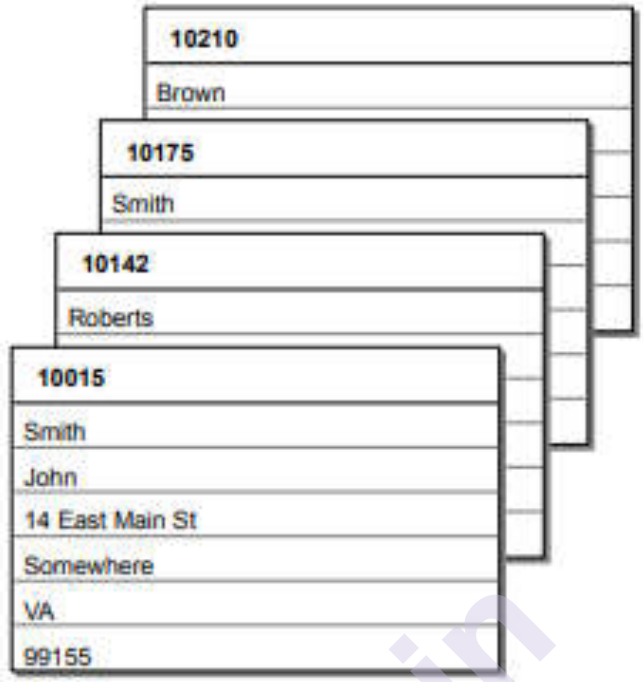
Adding Elements	➢ We must ensure that duplicate values are not added to the set since the list structure does not handle this for us. ➢ When implementing the add method we must first determine if the supplied element is already in the list or not. ○ If the element is not a duplicate, we can simply append the value to the end of the list; ○ If the element is a duplicate, we do nothing. ➢ The reason for this is that the definition of the **add() operation** indicates no action is taken when an attempt is made to add a duplicate value. ➢ This is known as a noop, which is short for no operation and indicates no action is taken. ➢ Noops are appropriate in some cases, which will be stated implicitly in the definition of an abstract data type by indicating no action is to be taken when the precondition fails as we did with the add() operation.
Comparing Two Sets	➢ For the operations that require a second set as an argument, we can use the operations of the Set ADT itself to access and manipulate the data of the second set. ➢ Consider the "equals" operation, which determines if both sets contain the exact same elements. ➢ We first check to make sure the two sets contain the same number of elements; otherwise, they cannot be equal. ➢ It would be inefficient to compare the individual elements since we already know the two sets cannot be equal.

	➢ After verifying the size of the lists, we can test to see if the self set is a subset of setB by calling **self.isSubsetOf(setB)**. ➢ This is a valid test since two equal sets are subsets of each other and we already know they are of the same size. ➢ To determine if one set is the subset of another, we can iterate over the list of elements in the self set and make sure each is contained in setB. ➢ If just one element in the self set is not in setB, then it is not a subset.
The Set Union	➢ Some of the operations create and return a new set based on the original, but the original is not modified. ➢ This is accomplished by creating a new set and populating it with the appropriate data from the other sets. ➢ Consider the union() method, which creates a new set from the self set and setB passed as an argument to the method. ➢ Creating a new set, populated with the unique elements of the other two sets, requires three steps: (1) create a new set; (2) fill the newSet with the elements from setB; and (3) iterate through the elements of the self set, during which each element is added to the newSet if that element is not in setB

The Set Union	➢ For the **first step**, we simply create a new instance of the Set class. ➢ The **second step** is accomplished with the use of the list extend() method. ➢ It directly copies the entire contents of the list used to store the elements of the self set to the list used to store the elements of the newSet. ➢ For the **final step**, we iterate through the elements of setB and add those elements to the the newSet that are not in the self set. ➢ The unique elements are added to the newSet by appending them to the list used to store the elements of the newSet. ➢ The remaining operations of the Set ADT can be implemented in a similar fashion and are left as exercises.

3.2 MAPS

- An abstract data type that provides this type of search capability is often referred to as a **map or dictionary** since it maps a key to a corresponding value.
- Consider the problem of a university registrar having to manage and process large volumes of data related to students.
- To keep track of the information or records of data, the registrar assigns a unique student identification number to each individual student as illustrated in following Figure.



- Later, when the registrar needs to search for a student's information, the identification number is used.
- Using this keyed approach allows access to a specific student record.
- The Python dictionary is implemented using a hash table, which requires the key objects to contain the hash method for generating a hash code.
- This can limit the type of problems with which a dictionary can be used.

3.2.1 The Map Abstract Data Type

- The Map ADT provides a great example of an ADT that can be implemented using one of many different data structures.
- Our definition of the Map ADT, which is provided next, includes the minimum set of operations necessary for using and managing a map.

Definition Map ADT

A map is a container for storing a collection of data records in which each record is associated with a unique key.

The key components must be comparable.

Method	Description
Map():	Creates a new empty map.
length ():	Returns the number of key/value pairs in the map.
contains (key):	Determines if the given key is in the map and returns True if the key is found and False otherwise.
add(key, value):	Adds a new key/value pair to the map if the key is not already in the map or replaces the data associated with the key if the key is in the map. Returns True if this is a new key and False if the data associated with the existing key is replaced.
remove(key):	Removes the key/value pair for the given key if it is in the map and raises an exception otherwise.
valueOf(key):	Returns the data record associated with the given key. The key must exist in the map or an exception is raised.
iterator ():	Creates and returns an iterator that can be used to iterate over the keys in the map.

3.2.2 List-Based Implementation

- We indicated earlier that many different data structures can be used to implement a map.
- As with the Set ADT, both the array and list structures can be used, but the list is a better choice since it does not have a fixed size like an array and it can expand automatically as needed.
- In the implementation of the Bag and Set ADTs, we used a single list to store the individual elements.
- For the Map ADT, however, we must store both a key component and the corresponding value component for each entry in the map.
- We cannot simply add the component pairs to the list without some means of maintaining their association.
- One approach is to use two lists, one for the keys and one for the corresponding values.
- Accessing and manipulating the components is very similar to that used with the Bag and Set ADTs.
- The difference, however, is that the association between the component pairs must always be maintained as new entries are added and existing ones removed.

- To accomplish this, each key/value must be stored in corresponding elements of the parallel lists and that association must be maintained.
- Instead of using two lists to store the key/value entries in the map, we can use a single list.
- The individual keys and corresponding values can both be saved in a single object, with that object then stored in the list.
- A sample instance illustrating the data organization required for this approach is shown in Figure:

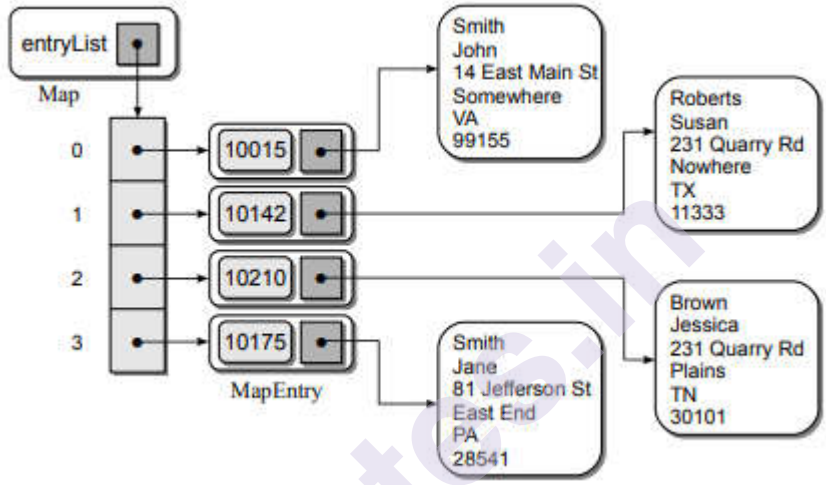
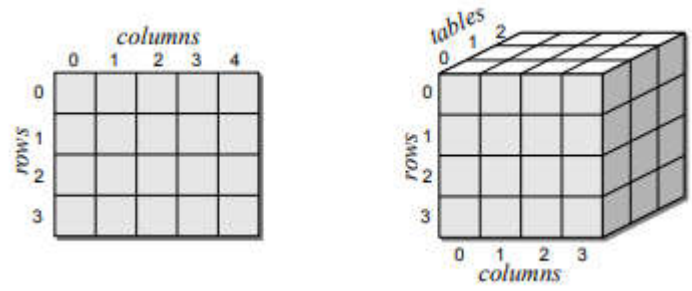


Figure: The Map ADT implemented using a single list.

3.3 MULTI-DIMENSIONAL ARRAYS

- A multi-dimensional array stores a collection of data in which the individual elements are accessed with multicomponent subscripts: $x_{i,j}$ or $y_{i,j,k}$.
- Following Figure illustrates the abstract view of a two- and three-dimensional array.



- Figure: Sample multi-dimensional arrays: (left) a 2-D array viewed as a rectangular table and (right) a 3-D array viewed as a box of tables.
- As we saw earlier, a two-dimensional array is typically viewed as a table or grid consisting of rows and columns.

- An individual element is accessed by specifying two indices, one for the row and one for the column.
- The three-dimensional array can be visualized as a box of tables where each table is divided into rows and columns.
- Individual elements are accessed by specifying the index of the table followed by the row and column indices.
- Larger dimensions are used in the solutions for some problems, but they are more difficult to visualize.

3.3.1 The MultiArray Abstract Data Type

To accommodate multi-dimensional arrays of two or more dimensions, we define the MultiArray ADT and as with the earlier array abstract data types, we limit the operations to those commonly provided by arrays in most programming languages that provide the array structure.

Definition MultiArray ADT

A multi-dimensional array consists of a collection of elements organized into multiple dimensions.

Individual elements are referenced by specifying an n-tuple or a subscript of multiple components, (i1, i2, . . . in), one for each dimension of the array.

All indices of the n-tuple start at zero.

Method	Description
MultiArray(d_1, d_2, . . . d_n):	Creates a multi-dimensional array of elements organized into n-dimensions with each element initially set to None. The number of dimensions, which is specified by the number of arguments, must be greater than 1. The individual arguments, all of which must be greater than zero, indicate the lengths of the corresponding array dimensions. The dimensions are specified from highest to lowest, where d_1 is the highest possible dimension and d_n is the lowest.
dims():	Returns the number of dimensions in the multi-dimensional array.

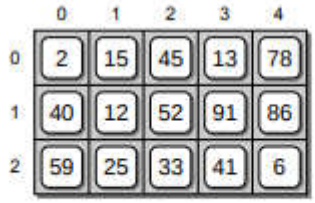
length(dim):	Returns the length of the given array dimension. The individual dimensions are numbered starting from 1, where 1 represents the first, or highest, dimension possible in the array. Thus, in an array with three dimensions, 1 indicates the number of tables in the box, 2 is the number of rows, and 3 is the number of columns.
clear(value):	Clears the array by setting each element to the given value.
getitem ($i_1, i_2, \ldots i_n$):	Returns the value stored in the array at the element position indicated by the n-tuple ($i_1, i_2, \ldots i_n$). All of the specified indices must be given and they must be within the valid range of the corresponding array dimensions. Accessed using the element operator: y = x[1, 2].
setitem ($i_1, i_2, \ldots i_n$, value):	Modifies the contents of the specified array element to contain the given value. The element is specified by the n-tuple ($i_1, i_2, \ldots i_n$). All of the subscript components must be given and they must be within the valid range of the corresponding array dimensions. Accessed using the element operator: x[1, 2] = y.

Data Organization

- Most computer architectures provide a mechanism at the hardware level for creating and using one-dimensional arrays.
- Programming languages need only provide appropriate syntax to make use of a 1-D array.
- Multi-dimensional arrays are not handled at the hardware level.
- Instead, the programming language typically provides its own mechanism for creating and managing multi-dimensional arrays.

Array Storage

- A one-dimensional array is commonly used to physically store arrays of higher dimensions.
- Consider a two-dimensional array divided into a table of rows and columns as illustrated in the following Figure.



The abstract view of a sample 3×5 two-dimensional array.

- There are two common approaches.
- The elements can be stored in row-major order or column-major order.
- Most high-level programming languages use row-major order, with FORTRAN being one of the few languages that uses column-major ordering to store and manage 2-D arrays.
- In row-major order, the individual rows are stored sequentially, one at a time, as illustrated in given Figure:

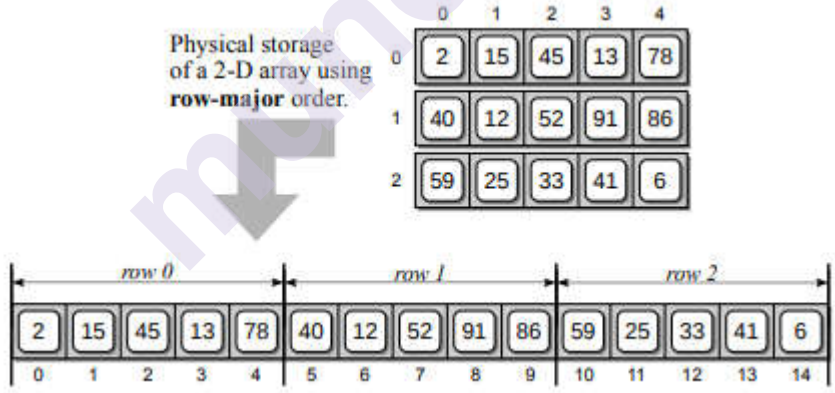


Figure: Physical storage of a sample 2-D array (top) in a 1-D array using row-major order (bottom).

- Figure: Physical storage of a sample 2-D array (top) in a 1-D array using row-major order (bottom).
- The first row of 5 elements are stored in the first 5 sequential elements of the 1-D array,
- the second row of 5 elements are stored in the next five sequential elements, and so forth.

- In column-major order, the 2-D array is stored sequentially, one entire column at a time, as illustrated in the given Figure.

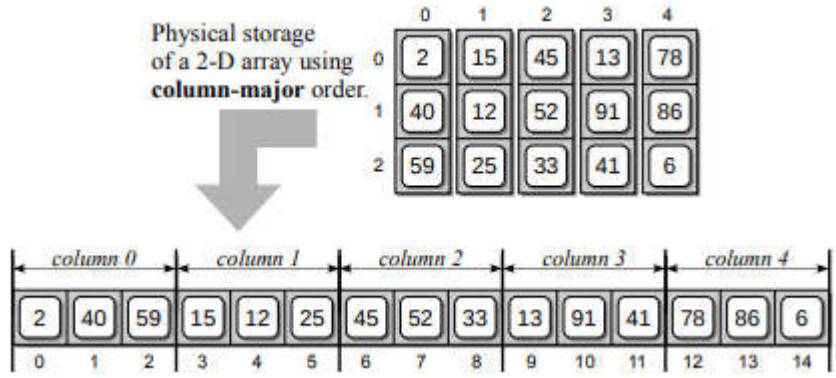


Figure: Physical storage of a sample 2-D array (top) in a 1-D array using column major order (bottom).

- The first column of 3 elements are stored in the first 3 sequential elements of the 1-D array, followed by the 3 elements of the second column, and so on.

3.3.2 Implementing the MultiArray

- To implement the MultiArray ADT, the elements of the multi-dimensional array can be stored in a single 1-D array in row-major order.
- Not only does this create a fast and compact array structure, but it's also the actual technique used by most programming languages.

Constructor	➢The constructor, defines three data fields: **_dims** stores the sizes of the individual dimensions; **_factors** stores the factor values used in the index equation; and ➢elements is used to store the 1-D array used as the physical storage for the multi-dimensional array. ➢The resulting tuple will contain the sizes of the individual dimensions and is assigned to the **_dims** field. ➢1-D array is created and assigned to the **_factors** field.

Dimensionality and Lengths	➢ In the multi-dimensional version of the array,each dimension of the array has an associated size. ➢ The size of the requested dimension is then returned using the appropriate value from the **_dims** tuple. ➢ The **_numDims()** method returns the dimensionality of the array, which can be obtained from the number of elements in the **_dims** tuple.
Element Access	➢ Access to individual elements within an n-D array requires an n-tuple or multicomponent subscript, one for each dimension. ➢ The contents of the **ndxTuple** are passed to the ➢ **_computeIndex()** helper method to compute the index offset within the 1-D storage array. ➢ The use of the helper method reduces the need for duplicate code that otherwise would be required in both element access methods. The **_ _setitem_ _** operator method can be implemented in a similar fashion. ➢ The major difference is that this method requires a second argument to receive the value to which an element is set and modifies the indicated element with the new value instead of returning a value.
Computing the Offset	➢ The **_computeIndex()** helper method implements Equation which computes the offset within the 1-D storage array. ➢ The method must also verify the subscript components are within the legal range of the dimension lengths. ➢ If they are valid, the offset is computed and returned; otherwise, None is returned to flag an invalid array index. ➢ By returning **None** from the helper method instead of raising an exception within the method, better information can be provided to the programmer as to the exact element access operation that caused the error.

3.4 APPLICATION: SALES REPORTS

Scenario:

- LazyMart, Inc. is a small regional chain department store with locations in several different cities and states.
- The company maintains a collection of sales records for the various items sold and would like to generate several different types of reports from this data.
- One such report, for example, is the yearly sales by store, as illustrated in Figure :

LazyMart Sales Report
Store #1

Item#	Jan	Feb	Mar	...	Nov	Dec
1	1237.56	1543.23	1011.00		2101.88	2532.99
2	829.85	974.18	776.54		802.50	643.21
3	3100.00	3218.25	3005.34		2870.50	3287.25
4	1099.45	1573.75	1289.21		1100.00	1498.25
:	:	:	:	...	:	:
99	704.00	821.30	798.00		532.00	699.50
100	881.25	401.00	375.00		732.00	500.00

Figure: A sample sales report

- The sales data of the current calendar year for all of LazyMart's stores is maintained as a collection of entries in a text file.
- **For example, the following illustrates the first several lines of a sample sales data text file:**

```
8
100
5  11  85   45.23
1   4  26   128.93
1   8  75   39.77
    :
```

  - where the first line indicates the number of stores;
  - the second line indicates the number of individual items (both of which are integers); and
  - the remaining lines contain the sales data.
- Each line of the sales data consists of four pieces of information: the store number, the month number, the item number, and the sales amount for the given item in the given store during the given month.

- For simplicity, the store and item numbers will consist of consecutive integer values in the range [1 . . . max], where max is the number of stores or items as extracted from the first two lines of the file.
- The month is indicated by an integer in the range [1 . . . 12] and the sales amount is a floating-point value.

Data Organization	➢ For this problem, where we may need to produce many different reports from the same collection of data , we first organize the data in some meaningful way in order to extract the information needed. ➢ The ideal structure for storing the sales data is a 3-D array, in which one dimension represents the stores, another represents the items sold in the stores, and the last dimension represents each of the 12 months in the calendar year. ➢ The 3-D array can be viewed as a collection of spreadsheets ➢ Each spreadsheet contains the sales for a specific store and is divided into rows and columns where each row contains the sales for one item and the columns contain the sales for each month. ➢ Example: the creation and initialization of the 3-D array as shown here: salesData = MultiArray(8, 100, 12)
Total Sales by Store	➢ With the data loaded from the file and stored in a 3-D array, we can produce many different types of reports or extract various information from the sales data. ➢ For example, suppose we want to determine the total sales for a given store, which includes the sales figures of all items sold in that store for all 12 months. ➢ Assuming our view of the data as a collection of spreadsheets, this requires traversing over every element in the spreadsheet containing the data for the given **store**.

	➢ If **store** equals 1, this is equivalent to processing every element in the spreadsheet. ➢ Two nested loops are required since we must sum the values from each row and column contained in the given store spreadsheet. ➢ The number of rows (dimension number 2) and columns (dimension number 3) can be obtained using the length() array method. ➢ **return total**
Total Sales by Month	➢ Next, suppose we want to compute the total sales for a given month that includes the sales figures of all items in all stores sold during that month. ➢ This time, the two nested loops have to iterate over every row of every spreadsheet for the single column representing the given month. ➢ **return total**

```
# Compute the total sales of all items for all months
def totalSalesByStore( salesData, store ):
   # Subtract 1 from the store # since the array indic
   # than the given store #.

   s = store-1
    # Accumulate the total sales for the given store.
   total = 0.0

    # Iterate over item.
   for i in range( salesData.length(2) ):
      # Iterate over each month of the i item.
     for m in range( salesData.length(3) ):
       total += salesData[s, i, m]
```

```
# Compute the total sales of all items in all stores for a given
def totalSalesByMonth( salesData, month ):
   # The month number must be offset by 1.
  m = month - 1
   # Accumulate the total sales for the given month.
  total = 0.0

   # Iterate over each store.
  for s in range( salesData.length(1) ):
     # Iterate over each item of the s store.
    for i in range( salesData.length(2) ):
      total += salesData[s, i, m]
```

Total Sales by Item

➢ Another value that we can compute from the sales data in the 3-D array is the total sales for a given item, which includes the sales figures for all 12 months and from all 8 stores.

```
# Compute the total sales of a single item in all stores over all m
def totalSalesByItem( salesData, item ):
   # The item number must be offset by 1.
  m = item - 1

   # Accumulate the total sales for the given month.
  total = 0.0

   # Iterate over each store.
  for s in range( salesData.length(1) ):
     # Iterate over each month of the s store.
    for m in range( salesData.length(3) ):
      total += salesData[s, i, m]

➢   return total
```

Monthly Sales by Store

➢ Finally, suppose we want to compute the total monthly sales for each of the 12 months at a given store.

➢ While the previous examples computed a single value, this task requires the computation of 12 different totals, one for each month.

➢ We can store the monthly totals in a 1-D array and return the structure, as is done in the following function:

```
# Compute the total sales per month for a given store. A 1-D array is
# returned that contains the totals for each month.

def totalSalesPerMonth( salesData, store ):
   # The store number must be offset by 1.
  s = store - 1

   # The totals will be returned in a 1-D array.
  totals = Array( 12 )

   # Iterate over the sales of each month.
  for m in range( salesData.length(3) ):
    sum = 0.0

     # Iterate over the sales of each item sold during the m month.
    for i in range( salesData.length(2) ):
      sum += salesData[s, i, m]

     # Store the result in the corresponding month of the totals array.
    totals[m] = sum

   # Return the 1-D array.
➢  return totals
```

3.5 EXERCISE

Answer the following:

1. Complete the Set ADT by implementing intersect() and difference().
2. Modify the Set() constructor to accept an optional variable argument to which a collection of initial values can be passed to initialize the set.

   The prototype for the new constructor should look as follows:

   def Set(self, *initElements = None)

   It can then be used as shown here to create a set initialized with the given values:

   s = Set(150, 75, 23, 86, 49)
3. Add a new operation to the Set ADT to test for a proper subset. Given two sets, A and B, A is a proper subset of B, if A is a subset of B and A does not equal B.

Reference:

Data Structure and algorithm Using Python, Rance D. Necaise, 2016 Wiley India Edition

4

ALGORITHM ANALYSIS

Unit Structure

4.0 OBJECTIVE

- In this section we are going to learn about Algorithm analysis.
- In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.
- By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.
- In this chapter, we will also discuss the analysis of the algorithm using Big – O asymptotic notation in complete detail.

4.1 ALGORITHM ANALYSIS

- Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.
- Most algorithms are designed to work with inputs of arbitrary length.
- Algorithms are designed to solve problems, but a given problem can have many different solutions.
- One approach is to measure the execution time to determine which solution is the most efficient for a given problem.

- We can implement the solution by constructing a computer program, using a given programming language.
- We then execute the program and time it using a wall clock or the computer's internal clock.
- The execution time is dependent on several factors.
- First, the amount of data that must be processed directly affects the execution time.
- As the data set size increases, so does the execution time.
- Second, the execution times can vary depending on the type of hardware and the time of day a computer is used.
- If we use a multi-process, multi-user system to execute the program, the execution of other programs on the same machine can directly affect the execution time of our program.
- Finally, the choice of programming language and compiler used to implement an algorithm can also influence the execution time.
- Some compilers are better optimizers than others and some languages produce better optimized code than others.
- Thus, we need a method to analyze an algorithm's efficiency independent of the implementation details.

4.1.1 The Need for Analysis

- Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation).
- However, the main concern of analysis of algorithms is the required time or performance.
- Following are the types of analysis :
    - Worst-case − The worst case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' Therefore, if various algorithms for sorting are taken into account and say 'n,' input data items are supplied in reverse order for a sorting algorithm, then the algorithm will require n^2 operations to perform the sort which will correspond to the worst case time complexity of the algorithm.
    - **Best-case** − The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' The running time of many algorithms varies not only for the inputs of different sizes but also for the different inputs of the same size.

- **Average case** − An average number of steps taken on any instance of size a.
- **Amortized** − A sequence of operations applied to the input of size a averaged over time.

- To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.
- Reasons for analyzing algorithms:
  1. To predict the resources that the algorithm require
     - Computational Time(CPU consumption).
     - Memory Space(RAM consumption).
     - Communication bandwidth consumption.
  2. To predict the running time of an algorithm
     - Total number of primitive operations executed.

4.2 COMPLEXITY ANALYSIS

- To determine the efficiency of an algorithm, we can examine the solution itself and measure those aspects of the algorithm that most critically affect its execution time.
- For example, we can count the number of logical comparisons, data interchanges, or arithmetic operations.
- Consider the following algorithm for computing the sum of each row of an n × n matrix and an overall sum of the entire matrix:

```
totalSum = 0                          # Version 1
for i in range( n ) :
  rowSum[i] = 0
  for j in range( n ) :
    rowSum[i] = rowSum[i] + matrix[i,j]
    totalSum = totalSum + matrix[i,j]
```

- Suppose we want to analyze the algorithm based on the number of additions performed.
- In this example, there are only two addition operations, making this a simple task.
- The algorithm contains two loops, one nested inside the other.
- The inner loop is executed n times and since it contains the two addition operations, there are a total of 2n additions performed by the inner loop for each iteration of the outer loop.

- The outer loop is also performed n times, for a total of $2n^2$ additions.
- We improve upon this algorithm to reduce the total number of addition operations performed.
- Consider a new version of the algorithm in which the second addition is moved out of the inner loop and modified to sum the entries in the rowSum array instead of individual elements of the matrix.

```
totalSum = 0                              # Version 2
for i in range( n ) :
  rowSum[i] = 0
  for j in range( n ) :
    rowSum[i] = rowSum[i] + matrix[i,j]
  totalSum = totalSum + rowSum[i]
```

- In this version, the inner loop is again executed n times, but this time, it only contains one addition operation.
- That gives a total of n additions for each iteration of the outer loop, but the outer loop now contains an addition operator of its own.
- To calculate the total number of additions for this version, we take the n additions performed by the inner loop and add one for the addition performed at the bottom of the outer loop.
- This gives n + 1 additions for each iteration of the outer loop, which is performed n times for a total of $n^2 + n$ additions.

Compare the two results:

➔ The number of additions in the second version is less than the first for any n greater than 1.

➔ Thus, the second version will execute faster than the first, but the difference in execution times will not be significant.

➔ The reason is that both algorithms execute on the same order of magnitude, namely n^2.

➔ Thus, as the size of n increases, both algorithms increase at approximately the same rate (though one is slightly better), as illustrated numerically in following Table :

n	$2n^2$	$n^2 + n$
10	200	110
100	20,000	10,100
1000	2,000,000	1,001,000
10000	200,000,000	100,010,000
100000	20,000,000,000	10,000,100,000

Table: Growth rate comparisons for different input sizes.

➔ and graphically in below Figure:

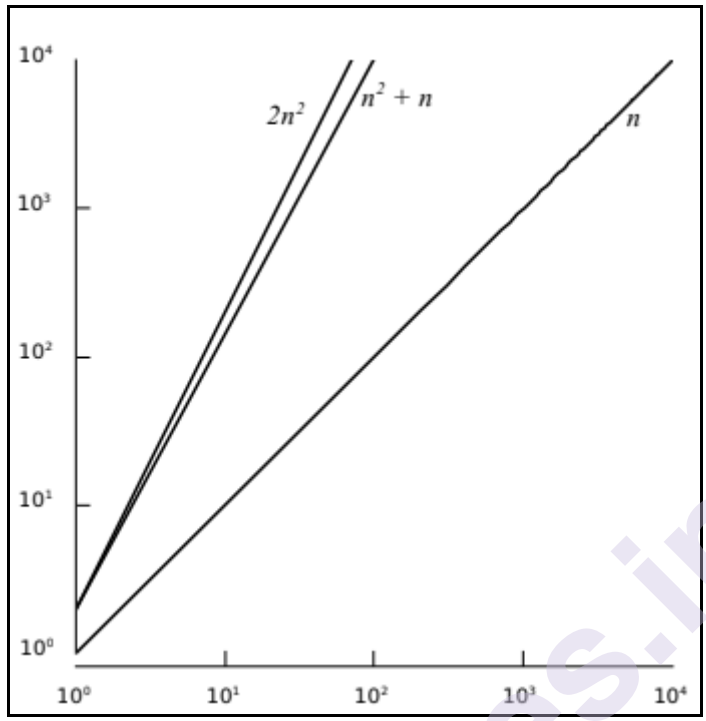


Figure: Graphical comparison of the growth rates from above Table.

4.2.1 Big-O Notation:

- Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- Big O is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann–Landau notation or asymptotic notation.
- Instead of counting the precise number of operations or steps, computer scientists are more interested in classifying an algorithm based on the order of magnitude as applied to execution time or space requirements.
- This classification approximates the actual number of required steps for execution or the actual storage requirements in terms of variable-sized data sets.
- The term big-O, which is derived from the expression "on the order of," is used to specify an algorithm's classification.
- We can express algorithmic complexity using the big-O notation. For a problem of size N:

  ➔ A constant-time function/method is "order 1" : O(1)

➔ A linear-time function/method is "order N" : O(N)

➔ A quadratic-time function/method is "order N squared" : O(N^2)

Definition:

Let g and f be functions from the set of natural numbers to itself.

The function f is said to be O(g) (read big-oh of g), if there is a constant c > 0 and a natural number n0 such that f (n) ≤ cg(n) for all n >= n0 .

- The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

f(n) = O(g(n))

if there exists a positive integer n_0 and a positive constant c, such that

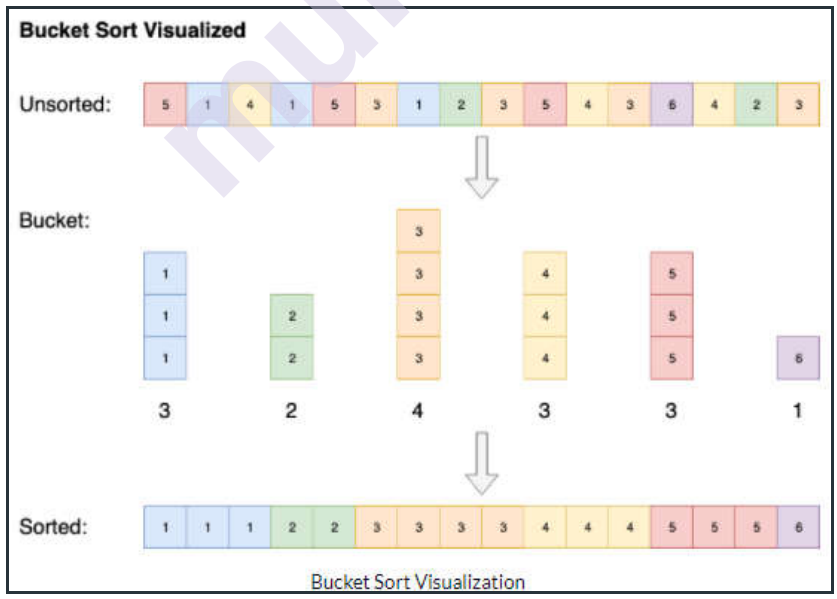
$$f(n) \leq c.g(n) \ \forall \ n \geq n_0$$

- The general step wise procedure for Big-O runtime analysis is as follows:
  - Figure out what the input is and what n represents.
  - Express the maximum number of operations the algorithm performs in terms of n.
  - Eliminate all excluding the highest order terms.
  - Remove all the constant factors.
- Big - O helps to determine the time as well as space complexity of the algorithm.
- Using Big - O notation, the time taken by the algorithm and the space required to run the algorithm can be ascertained.
- Some of the lists of common computing times of algorithms in order of performance are as follows:
  - O (1)
  - O (log n)
  - O (n)
  - O (nlog n)
  - O (n^2)
  - O (n^3)
  - O (2^n)

- Thus algorithm with their computational complexity can be rated as per the mentioned order of performance.

Time & Space Complexity

- So far, we have only been discussing the time complexity of the algorithms.
- That is, we only care about how much time it takes for the program to complete the task.
- What also matters is the space the program takes to complete the task.
- The space complexity is related to how much memory the program will use, and therefore is also an important factor to analyze.
- The space complexity works similarly to time complexity.
- For example, selection sort has a space complexity of O(1), because it only stores one minimum value and its index for comparison, the maximum space used does not increase with the input size.
- Some algorithms, such as bucket sort, have a space complexity of O(n), but are able to chop down the time complexity to O(1).
- Bucket sort sorts the array by creating a sorted list of all the possible elements in the array, then increments the count whenever the element is encountered.
- In the end the sorted array will be the sorted list elements repeated by their counts.



Bucket Sort Visualization

4.3 EVALUATING PYTHON CODE

- As indicated earlier, when evaluating the time complexity of an algorithm or code segment, we assume that basic operations only require constant time.
- The **basic operations** include statements and function calls whose execution time does not depend on the specific values of the data that is used or manipulated by the given instruction.
- For example, the assignment statement

  x = 5

  is a basic instruction since the time required to assign a reference to the given variable is independent of the value or type of object specified on the right hand side of the = sign.
- The evaluation of arithmetic and logical expressions

  y = x

  z = x + y * 6

  done = x > 0 **and** x < 100
- are basic instructions, again since they require the same number of steps to perform the given operations regardless of the values of their operands.
- The subscript operator, when used with Python's sequence types (strings, tuples, and lists) is also a basic instruction.
- **Linear Time Examples:**
  - Now, consider the following assignment statement:

    y = ex1(n)
  - An assignment statement only requires constant time, but that is the time required to perform the actual assignment and does not include the time required to execute any function calls used on the righthand side of the assignment statement.
  - To determine the run time of the previous statement, we must know the cost of the function call ex1(n).
  - The time required by a function call is the time it takes to execute the given function. For example, consider the ex1() function, which computes the sum of the integer values in the range [0 . . . n):

```
def ex1( n ):
  total = 0
  for i in range( n ) :
    total += i
  return total
```

- The time required to execute a loop depends on the number of iterations performed and the time needed to execute the loop body during each iteration.
- In this case, the loop will be executed n times and the loop body only requires constant time since it contains a single basic instruction.
- (Note that the underlying mechanism of the for loop and the range() function are both O(1).)
- We can compute the time required by the loop as $T(n) = n * 1$ for a result of O(n).
- The first line of the function and the return statement only require constant time.
- Since the loop is the only non-constant step, the function ex1() has a run time of O(n).
- That means the statement y = ex1(n) from earlier requires linear time.
- Next, consider the following function, which includes two for loops:

```
def ex2( n ):
  count = 0
  for i in range( n ) :
    count += 1
  for j in range( n ) :
    count += 1
  return count
```

- To evaluate the function, we have to determine the time required by each loop.
- The two loops each require O(n) time as they are just like the loop in function ex1() earlier.
- If we combine the times, it yields T(n) = n + n for a result of O(n).

4.4 EVALUATING PYTHON LIST

- We defined several abstract data types for storing and using collections of data in the previous chapters.

- The next logical step is to analyze the operations of the various ADTs to determine their efficiency.
- The result of this analysis depends on the efficiency of the Python list since it was the primary data structure used to implement many of the earlier abstract data types.
- In this section, we use those details and evaluate the efficiency of some of the more common operations.
- A summary of the worst case run times are shown in Table 4.4.

List Operation	Worst Case
`v = list()`	$O(1)$
`v = [ 0 ] * n`	$O(n)$
`v[i] = x`	$O(1)$
`v.append(x)`	$O(n)$
`v.extend(w)`	$O(n)$
`v.insert(x)`	$O(n)$
`v.pop()`	$O(n)$
traversal	$O(n)$

Table 4.4: Worst case time-complexities for the more common list operations.

List Traversal

- A sequence traversal accesses the individual items, one after the other, in order to perform some operation on every item.
- Python provides the built-in iteration for the list structure, which accesses the items in sequential order starting with the first item.
- Consider the following code segment, which iterates over and computes the sum of the integer values in a list:

```
sum = 0
for value in valueList :
  sum = sum + value
```

- To determine the order of complexity for this simple algorithm, we must first look at the internal implementation of the traversal.
- Iteration over the contiguous elements of a 1-D array, which is used to store the elements of a list, requires a count-controlled loop with an index variable whose value ranges over the indices of the subarray.
- The list iteration above is equivalent to the following:

```
sum = 0
for i in range( len(valueList) ) :
  sum = sum + valueList[i]
```

- Assuming the sequence contains n items, it's obvious the loop performs n iterations.
- Since all of the operations within the loop only require constant time, including the element access operation, a complete list traversal requires O(n) time.
- Note, this time establishes a minimum required for a complete list traversal.
- It can actually be higher if any operations performed during each iteration are worse than constant time, unlike this example.

List Allocation

- Creating a list, like the creation of any object, is considered an operation whose time-complexity can be analyzed.
- There are two techniques commonly used to create a list:

```
temp = list()
valueList = [ 0 ] * n
```

- The first example creates an empty list, which can be accomplished in constant time.
- The second creates a list containing n elements, with each element initialized to 0.
- The actual allocation of the n elements can be done in constant time, but the initialization of the individual elements requires a list traversal.
- Since there are n elements and a traversal requires linear time, the allocation of a vector with n elements requires O(n) time.

Appending to a List

- The append() operation adds a new item to the end of the sequence.
- If the underlying array used to implement the list has available capacity to add the new item, the operation has a best case time of O(1) since it only requires a single element access.
- Creating the new larger array and destroying the old array can each be done in O(1) time.

- To copy the contents of the old array to the new larger array, the items have to be copied element by element, which requires O(n) time.
- Combining the times from the three steps yields a time of

  T(n) = 1 + 1 + n and

  a worst case time of O(n).

Extending a List

- The extend() operation adds the entire contents of a source list to the end of the destination list.
- This operation involves two lists, each of which have their own collection of items that may be of different lengths.
- To simplify the analysis, however, we can assume both lists contain n items.
- When the destination list has sufficient capacity to store the new items, the entire contents of the source list can be copied in O(n) time.
- But if there is not sufficient capacity, the underlying array of the destination list has to be expanded to make room for the new items.
- This expansion requires O(n) time since there are currently n items in the destination list.
- After the expansion, the n items in the source list are copied to the expanded array, which also requires O(n) time.
- Thus, in the worst case the extend operation requires T(n) = n + n = 2n or O(n) time.

Inserting and Removing Items

- Inserting a new item into a list is very similar to appending an item except the new item can be placed anywhere within the list, possibly requiring a shift in elements.
- An item can be removed from any element within a list, which may also involve shifting elements.
- Both of these operations require linear time in the worst case, the proof of which is left as an exercise.

The list data type has some more methods. Here are all of the methods of list objects:

Method	Description
list.append(x)	➔ Add an item to the end of the list. Equivalent to a[len(a):] = [x].
list.extend(iterable)	➔ Extend the list by appending all the items from the iterable. ➔ Equivalent to a[len(a):] = iterable.
list.insert(i, x)	➔ Insert an item at a given position. ➔ The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
list.remove(x)	➔ Remove the first item from the list whose value is equal to x. ➔ It raises a ValueError if there is no such item.
list.pop([i])	➔ Remove the item at the given position in the list, and return it. ➔ If no index is specified, a.pop() removes and returns the last item in the list. ➔ (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
list.clear()	➔ Remove all items from the list. Equivalent to del a[:].
list.index(x[, start[, end]])	➔ Return zero-based index in the list of the first item whose value is equal to x. ➔ Raises a ValueError if there is no such item. ➔ The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. ➔ The returned index is computed relative to the beginning of the full sequence rather than the start argument.

list.count(x)	➔ Return the number of times x appears in the list.
list.sort(*, key=None, reverse=False)	➔ Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).
list.reverse()	➔ Reverse the elements of the list in place.
list.copy()	➔ Return a shallow copy of the list. Equivalent to a[:].

4.5 AMORTIZED COST

Amortize Analysis

- This analysis is used when the occasional operation is very slow, but most of the operations which are executing very frequently are faster.
- Data structures we need amortized analysis for Hash Tables, Disjoint Sets etc.
- In the Hash-table, the most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations.
- When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.
- Amortized analysis is the process of computing the time-complexity for a sequence of operations by computing the average cost over the entire sequence.
- For this technique to be applied, the cost per operation must be known and it must vary in which many of the operations in the sequence contribute little cost and only a few operations contribute a high cost to the overall time.
- This is exactly the case with the append() method.
- In a long sequence of append operations, only a few instances require O(n), while many of them are O(1).
- The amortized cost can only be used for a long sequence of append operations.
- If an algorithm used a single append operation, the cost for that one operation is still O(n) in the worst case since we do not know if that's the instance that causes the underlying array to be expanded.

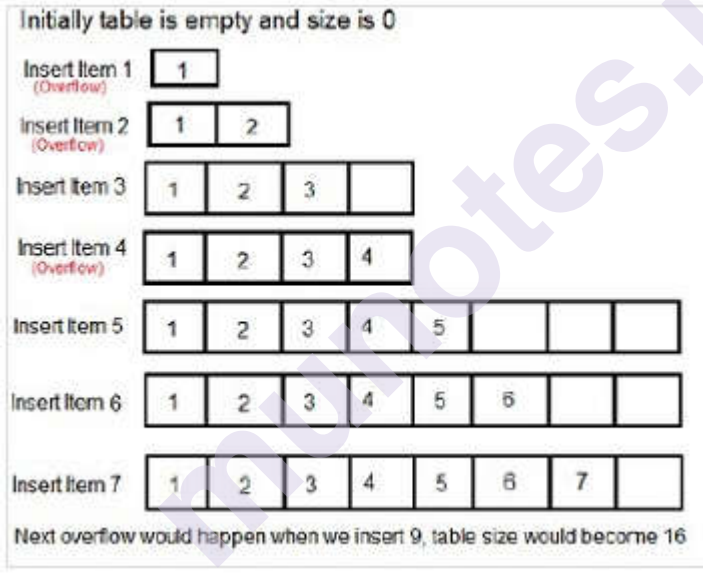
Aggregate Method

- The aggregate method is used to find the total cost.
- If we want to add a bunch of data, then we need to find the amortized cost by this formula.
- For a sequence of n operations, the cost is −

$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

Example on Amortized Analysis

- For a dynamic array, items can be inserted at a given index in O(1) time.
- But if that index is not present in the array, it fails to perform the task in constant time.
- For that case, it initially doubles the size of the array then inserts the element if the index is present.



For the dynamic array, let = cost of *ith* insertion.

$$So\ ci = 1 + \begin{cases} i-1, & if\ i-1\ is\ power\ of\ 2 \\ 0, & Otherwise \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} \leq \frac{n + \sum_{j=1}^{\lfloor log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

Following Table illustrates the aggregate method when applied to a sequence of 16 append operations. s_i represents the time required to physically store the i^{th} value

when there is an available slot in the array or immediately after the array has been expanded.

Storing an item into an array element is a constant time operation.

e_i represents the time required to expand the array when it does not contain available capacity to store the item.

Based on our assumptions related to the size of the array, an expansion only occurs when $i - 1$ is a power of 2 and the time incurred is based on the current size of the array $(i - 1)$.

While every append operation entails a storage cost, relatively few require an expansion cost.

Note that as the size of n increases, the distance between append operations requiring an expansion also increases.

Based on the tabulated results in following Table , the total time required to perform a sequence of 16 append operations on an initially empty list is 31, or just under 2n.

This results from a total storage cost (s_i) of 16 and a total expansion cost (e_i) of 15.

It can be shown that for any n, the sum of the storage and expansion costs, $s_i + e_i$, will never be more than $T(n) = 2n$.

Since there are relatively few expansion operations, the expansion cost can be distributed across the sequence of operations, resulting in an amortized cost of $T(n) = 2n/n$ or $O(1)$ for the append operation.

i	s_i	e_i	Size	List Contents
1	1	-	1	1
2	1	1	2	1 2
3	1	2	4	1 2 3
4	1	-	4	1 2 3 4
5	1	4	8	1 2 3 4 5
6	1	-	8	1 2 3 4 5 6
7	1	-	8	1 2 3 4 5 6 7
8	1	-	8	1 2 3 4 5 6 7 8
9	1	8	16	1 2 3 4 5 6 7 8 9
10	1	-	16	1 2 3 4 5 6 7 8 9 10
11	1	-	16	1 2 3 4 5 6 7 8 9 10 11
12	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12
13	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13
14	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14
15	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Table : Using the aggregate method to compute the total run time for a sequence of 16 append operations.

4.6 EVALUATING SET ADT

We can use complexity analysis to determine the efficiency of the Set ADT operations .

For convenience, the relevant portions of that implementation are shown again in below figur.

Listing 4.1 A partial listing of the `linearset.py` module from Listing 3.1.

```
1  class Set :
2    def __init__( self ):
3      self._theElements = list()
4
5    def __len__( self ):
6      return len( self._theElements )
7
8    def __contains__( self, element ):
9      return element in self._theElements
10
11   def add( self, element ):
12     if element not in self :
13       self._theElements.append( element )
14
15   def remove( self, element ):
16     assert element in self, "The element must be in the set."
17     self._theElements.remove( item )
18
19   def __eq__( self, setB ):
20     if len( self ) != len( setB ) :
21       return False
22     else :
23       return self.isSubsetOf( setB )
24
25   def isSubsetOf( self, setB ):
26     for element in self :
27       if element not in setB :
28         return False
29     return True
30
31   def union( self, setB ):
32     newSet = Set()
33     newSet._theElements.extend( self._theElements )
34     for element in setB :
35       if element not in self :
36         newSet._theElements.append( element )
37     return newSet
```

- The evaluation is quite simple since the ADT was implemented using the list and we just evaluated the methods for that structure.
- Following Table provides a summary of the worst case time-complexities for those operations implemented earlier in the text.

Operation	Worst Case
`s = Set()`	$O(1)$
`len(s)`	$O(1)$
`x in s`	$O(n)$
`s.add(x)`	$O(n)$
`s.isSubsetOf(t)`	$O(n^2)$
`s == t`	$O(n^2)$
`s.union(t)`	$O(n^2)$
traversal	$O(n)$

Table: Time-complexities for the Set ADT implementation using an unsorted list.

Simple Operations

- Evaluating the constructor and length operation is straightforward as they simply call the corresponding list operation.
- The contains method, which determines if an element is contained in the set, uses the in operator to perform a linear search over the elements stored in the underlying list.
- The search operation, which requires O(n) time, will be presented in the next section and we postpone its analysis until that time.
- The add() method also requires O(n) time in the worst case since it uses the in operator to determine if the element is unique and the append() method to add the unique item to the underlying list, both of which require linear time in the worst case.

Operations of Two Sets

- The remaining methods of the Set class involve the use of two sets, which we label A and B, where A is the self set and B is the argument passed to the given method.
- To simplify the analysis, we assume each set contains n elements.
- A more complete analysis would involve the use of two variables, one for the size of each set. But the analysis of this more specific case is sufficient for our purposes.
- The isSubsetOf() method determines if A is a subset of B.
- It iterates over the n elements of set A, during which the in operator is used to determine if the given element is a member of set B.
- Since there are n repetitions of the loop and each use of the in operator requires O(n) time, the isSubsetOf() method has a quadratic run time of $O(n^2)$.

- The set equality operation is also $O(n^2)$ since it calls isSubsetOf() after determining the two sets are of equal size.

Set Union Operation

- The set union() operation creates a new set, C, that contains all of the unique elements from both sets A and B. It requires three steps.
- The first step creates the new set C, which can be done in constant time.
- The second step fills set C with the elements from set A, which requires O(n) time since the extend() list method is used to add the elements to C.
- The last step iterates over the elements of set B during which the in operator is used to determine if the given element is a member of set A.
- If the element is not a member of set A, it's added to set C by applying the append() list method.
- We know from earlier the linear search performed by the in operator requires O(n) time and we can use the O(1) amortized cost of the append() method since it is applied in sequence.
- Given that the loop is performed n times and each iteration requires n + 1 time, this step requires $O(n^2)$ time.
- Combining the times for the three steps yields a worst case time of $O(n^2)$.

4.7 EXERCISE

Answer the following:

1. Arrange the following expressions from slowest to fastest growth rate.

$n \log_2 n$	4^n	$k \log_2 n$	$5n^2$	$40 \log_2 n$	$\log_4 n$	$12n^6$

2. Determine the O(·) for each of the following functions, which represent the number of steps required for some algorithm.

(a) $T(n) = n^2 + 400n + 5$	(e) $T(n) = 3(2^n) + n^8 + 1024$
(b) $T(n) = 67n + 3n$	(f) $T(n, k) = kn + \log k$
(c) $T(n) = 2n + 5n \log n + 100$	(g) $T(n, k) = 9n + k \log n + 1000$
(d) $T(n) = \log n + 2n^2 + 55$	

3. Prove or show why the worst case time-complexity for the insert() and remove() list operations is O(n).

4. Evaluate each of the following code segments and determine the O(·) for the best and worst cases. Assume an input size of n.

(a)
```
sum = 0
for i in range( n ) :
  if i % 2 == 0 :
    sum += i
```

(b)
```
sum = 0
i = n
while i > 0 :
  sum += i
  i = i / 2
```

(c)
```
for i in range( n ) :
  if i % 3 == 0 :
    for j in range( n / 2 ) :
      sum += j
  elif i % 2 == 0 :
    for j in range( 5 ) :
      sum += j
  else :
    for j in range( n ) :
      sum += j
```

❖❖❖❖

5

APPLICATION OF SEARCHING

Unit Structure

5.0 OBJECTIVE

In this chapter, we explore these important topics and study some of the basic algorithms for use with sequence structures.

The searching problem will be discussed many times throughout the text as it can be applied to collections stored using many different data structures, not just sequences.

In this chapter we are going to be able to explain and implement sequential search and binary search.

5.1 APPLICATION OF SEARCHING

- Searching Algorithms are designed to retrieve an element from any data structure where it is used.
- A Sorting Algorithm is used to arranging the data of list or array into some specific order.
- These algorithms are generally classified into two categories i.e. Sequential Search and Interval Search.

5.2 SEARCHING

- Searching is the process of selecting particular information from a collection of data based on specific criteria.
- In this text, we restrict the term searching to refer to the process of finding a specific item in a collection of data items.

- The search operation can be performed on many different data structures.
- The sequence search, which is the focus in this chapter, involves finding an item within a sequence using a search key to identify the specific item.
- A key is a unique value used to identify the data elements of a collection.
- In collections containing simple types such as integers or reals, the values themselves are the keys.
- For collections of complex types, a specific data component has to be identified as the key.
- In some instances, a key may consist of multiple components, which is also known as a compound key.

5.2.1 Linear Search Algorithm

- In this article, we will discuss the Linear Search Algorithm.
- Searching is the process of finding some particular element in the list.
- If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.
- Two popular search methods are Linear Search and Binary Search.
- So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.
- Linear search is also called a sequential search algorithm.
- It is the simplest searching algorithm.
- In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted.
- The worst-case time complexity of linear search is O(n).

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -

- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.

- If there is no match or the search element is not present in the given array, return **-1.**

The algorithm of linear search.

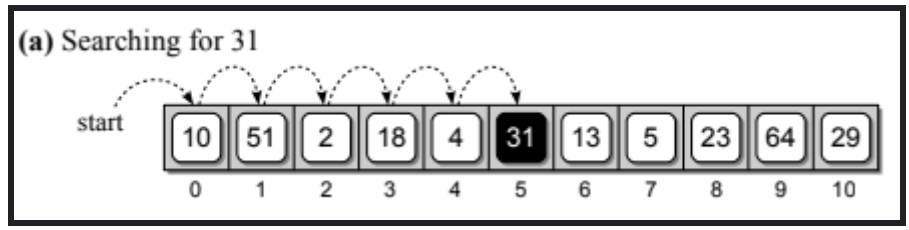
Algorithm

```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
set pos = i
print pos
go to step 6
[end of if]
set ii = i + 1
[end of loop]
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

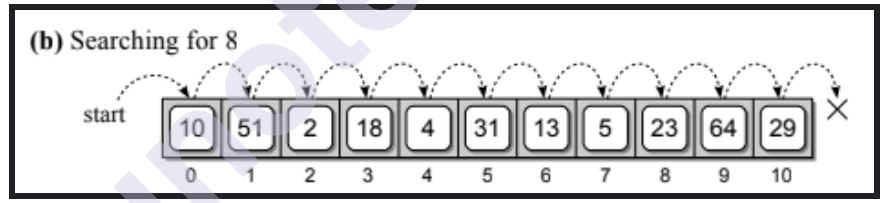

- The simplest solution to the sequence search problem is the sequential or linear search algorithm.
- This technique iterates over the sequence, one item at a time, until the specific item is found or all items have been examined.
- In Python, a target item can be found in a sequence using the in operator:

```
if key in theArray :
   print( "The key is in the array." )
else :
   print( "The key is not in the array." )
```

- The use of the in operator makes our code simple and easy to read but it hides the inner workings.
- Underneath, the in operator is implemented as a linear search.
- Consider the unsorted 1-D array of integer values shown in Figure (a).



- To determine if value 31 is in the array, the search begins with the value in the first element.
- Since the first element does not contain the target value, the next element in sequential order is compared to value 31.
- This process is repeated until the item is found in the sixth position.
- What if the item is not in the array?
- For example, suppose we want to search for value 8 in the sample array.
- The search begins at the first entry as before, but this time every item in the array is compared to the target value.
- It cannot be determined that the value is not in the sequence until the entire array has been traversed, as illustrated in Figure (b).

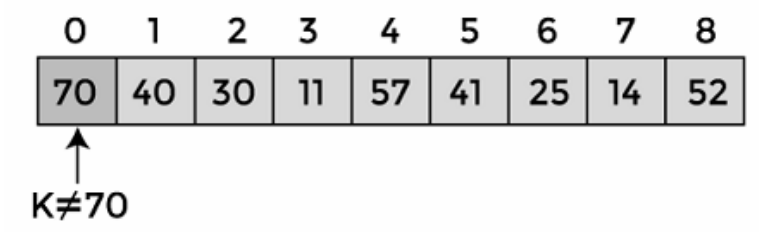


Working of Linear search

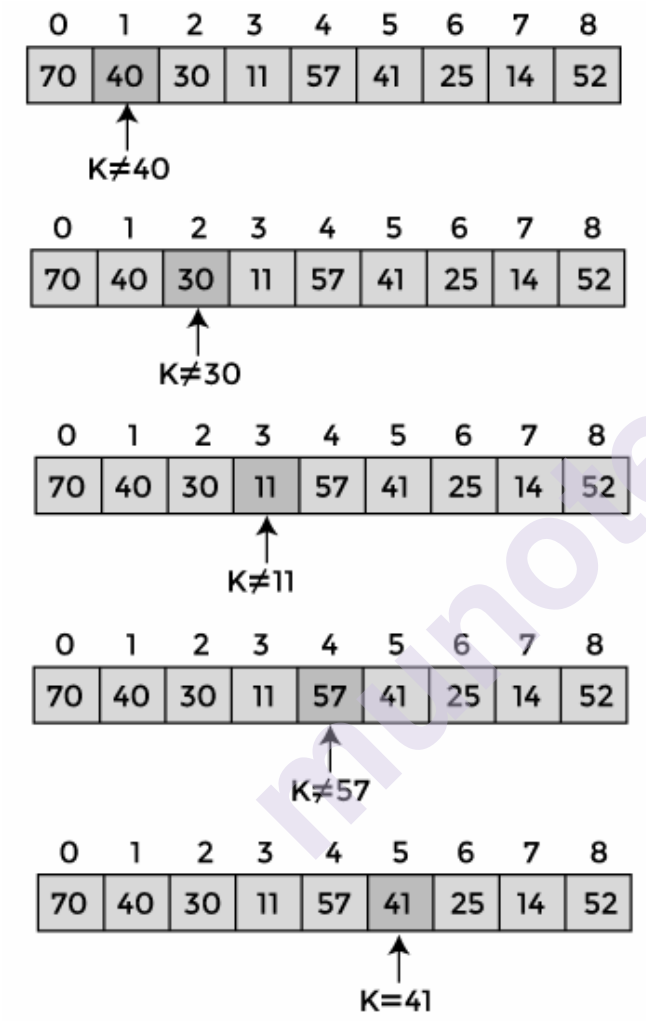
- Now, let's see the working of the linear search Algorithm.
- To understand the working of linear search algorithms, let's take an unsorted array. It will be easy to understand the working of linear search with an example.
- Let the elements of array are -

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

- Let the element to be searched is **K = 41**
- Now, start from the first element and compare **K** with each element of the array.



- The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



- Now, the element to be searched is found.
- So algorithm will return the index of the element matched.

Finding a Specific Item

- The function in Listing implements the sequential search algorithm, which results in a Boolean value indicating success or failure of the search.

```
1  def linearSearch( theValues, target ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is in the ith element, return True
5      if theValues[i] == target
6        return True
7
8    return False   # If not found, return False.
```

Implementation of the linear search on an unsorted sequence.

- This is the same operation performed by the Python operator.
- A count-controlled loop is used to traverse through the sequence during which each element is compared against the target value.
- If the item is in the sequence, the loop is terminated and True is returned.
- Otherwise, a full traversal is performed and False is returned after the loop terminates.
- To analyze the sequential search algorithm for the worst case, we must first determine what conditions constitute the worst case.
- Remember, the worst case occurs when the algorithm performs the maximum number of steps.
- For a sequential search, that occurs when the target item is not in the sequence and the loop iterates over the entire sequence.
- Assuming the sequence contains n items, the linear search has a worst case time of O(n).

Finding the Smallest Value

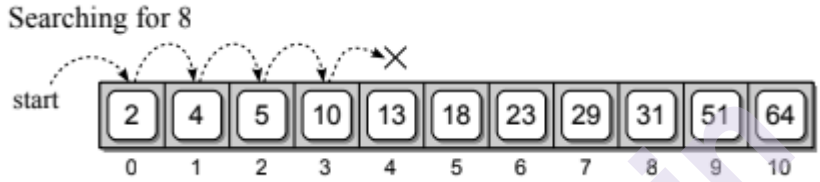
- Instead of searching for a specific value in an unsorted sequence, suppose we wanted to search for the smallest value, which is equivalent to applying Python's min() function to the sequence.
- A linear search is performed as before, but this time we must keep track of the smallest value found for each iteration through the loop, as illustrated in following figure:

```
1  def findSmallest( theValues ):
2    n = len( theValues )
3     # Assume the first item is the smallest value.
4    smallest = theValues[0]
5     # Determine if any other item in the sequence is smaller.
6    for i in range( 1, n ) :
7      if theList[i] < smallest :
8        smallest = theValues[i]
9
10   return smallest       # Return the smallest found.
```

- To prime the loop, we assume the first value in the sequence is the smallest and start the comparisons at the second item.
- Since the smallest value can occur anywhere in the sequence, we must always perform a complete traversal, resulting in a worst case time of O (n).

Searching a Sorted Sequence

- A linear search can also be performed on a sorted sequence, which is a sequence containing values in a specific order.
- For example, the values in the array illustrated in Fig. are in ascending or increasing numerical order.



- **The linear search on a sorted array.**
  - ○ That is, each value in the array is larger than its predecessor.
  - ○ A linear search on a sorted sequence works in the same fashion as that for the unsorted sequence, with one exception.
  - ○ It's possible to terminate the search early when the value is not in the sequence instead of always having to perform a complete traversal.
  - ○ For example, suppose we want to search for 8 in the array from Fig.
  - ○ When the fourth item, which is value 10, is examined, we know value 8 cannot be in the sorted sequence or it would come before 10.
  - ○ The implementation of a linear search on a sorted sequence is shown in Fig. on the next page.

```
1  def sortedLinearSearch( theValues, item ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is found in the ith element, return True
5      if theValues[i] == item :
6        return True
7      # If target is larger than the ith element, it's not in the sequence.
8      elif theValues[i] > item :
9        return False
10
11   return False   # The item is not in the sequence.
```

  - ○ The only modification to the earlier version is the inclusion of a test to determine if the current item within the sequence is larger than the target value.

- If a larger value is encountered, the loop terminates and False is returned.
- With the modification to the linear search algorithm, we have produced a better version, but the time-complexity remains the same.

- The reason is that the worst case occurs when the value is not in the sequence and is larger than the last element.
- In this case, we must still traverse the entire sequence of n items.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	O(1)

- The space complexity of linear search is O(1).

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.

2. It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

3. It has a very simple implementation.

5.2.2 The Binary Search:

Binary Search Algorithm

- In this section, we will discuss the Binary Search Algorithm.
- Searching is the process of finding some particular element in the list.
- If the element is present in the list, then the process is called successful, and the process returns the location of that element.
- Otherwise, the search is called unsuccessful.
- Linear Search and Binary Search are the two popular searching techniques.
- Here we will discuss the Binary Search Algorithm.
- Binary search is the search technique that works efficiently on sorted lists.
- Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.
- If the match is found then, the location of the middle element is returned.
- Otherwise, we search into either of the halves depending upon the result produced through the match.
- NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.
- The linear search algorithm for a sorted sequence produced a slight improvement over the linear search with an unsorted sequence, but both have a linear time complexity in the worst case.
- To improve the search time for a sorted sequence, we can modify the search technique itself.
- Consider an example where you are given a stack of exams, which are in alphabetical order, and are asked to find the exam for “Jessica Roberts.”

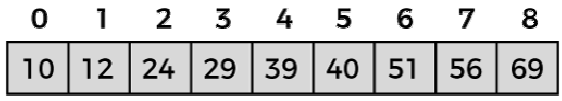
- In performing this task, most people would not begin with the first exam and flip through one at a time until the requested exam is found, as would be done with a linear search.
- Instead, you would probably flip to the middle and determine if the requested exam comes alphabetically before or after that one.
- Assuming Jessica's paper follows alphabetically after the middle one, you know it cannot possibly be in the top half of the stack.
- Instead, you would probably continue searching in a similar fashion by splitting the remaining stack of exams in half to determine which portion contains Jessica's exam.
- This is an example of a divide and conquer strategy, which entails dividing a larger problem into smaller parts and conquering the smaller part.

The Algorithm of Binary Search:

```
1. Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given
   array, 'lower_bound' is the index of the first array element,
   'upper_bound' is the index of the last array element, 'val' is the
   value to search
2. Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
3. Step 2: repeat steps 3 and 4 while beg <=end
4. Step 3: set mid = (beg + end)/2
5. Step 4: if a[mid] = val
6. set pos = mid
7. print pos
8. o to step 6
9. else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
Step 6: exit
```

Working of Binary search

- Now, let's see the working of the Binary Search Algorithm.
- To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.
- There are two methods to implement the binary search algorithm -
    - Iterative method
    - Recursive method
- The recursive method of binary search follows the divide and conquer approach.
- Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

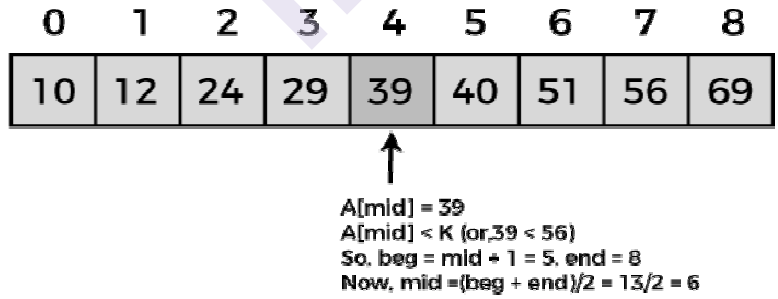
1.mid = (beg + end)/2

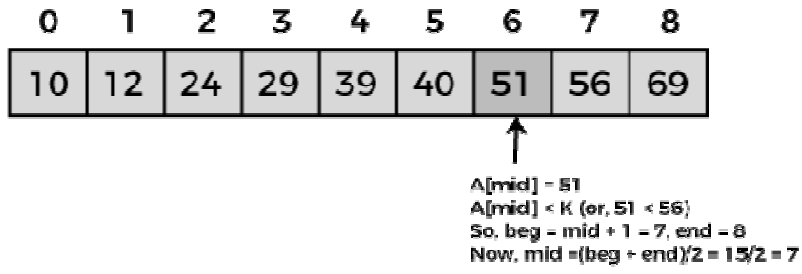
So, in the given array -

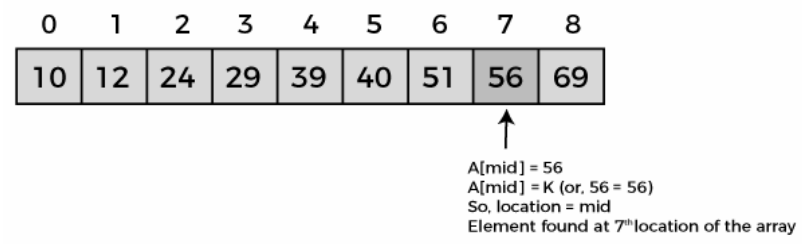
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found.

So algorithm will return the index of the element matched.

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

2. Space Complexity

Space Complexity	O(1)

- The space complexity of binary search is O(1).

5.3 IMPLEMENTATION USING PYTHON

5.3.1 Linear Search Using Python Programming:

```
# Linear Search in Python

def linearSearch(array, n, x):

    # Going through array sequentially

    for i in range(0, n):

        if (array[i] = = x):

            return i

    return -1

array = [2, 4, 0, 1, 9]

x = 1

n = len(array)

result = linearSearch(array, n, x)

if(result == -1):

    print("Element not found")

else:

    print("Element found at index: ", result)
```

Output

```
Element found at index 3
```

5.3.2 Binary Search Iterative Method:

```
# Binary Search in python
def binarySearch(array, x, low, high):
    # Repeat until the pointers low and high meet each other
    while low <= high:
        mid = low + (high - low)//2
        if array[mid] == x:
            return mid
        elif array[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
array = [3, 4, 5, 6, 7, 8, 9]
x = 4
result = binarySearch(array, x, 0, len(array)-1)
if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```

Output

```
Element is present at index 1
```

5.3.3 Binary Search (Recursive Method)

```
# Binary Search in python
def binarySearch(array, x, low, high):
    if high >= low:
        mid = low + (high - low)//2
        # If found at mid, then return it
        if array[mid] == x:
            return mid
        # Search the left half
        elif array[mid] > x:
            return binarySearch(array, x, low, mid-1)
        # Search the right half
        else:
            return binarySearch(array, x, mid + 1, high)
    else:
        return -1
array = [3, 4, 5, 6, 7, 8, 9]
x = 4
result = binarySearch(array, x, 0, len(array)-1)
if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```

Output

Element is present at index 1

5.4 EXERCISE

1. What do you mean by Searching? Explain Sequential search and Binary search with help of example.
2. What is searching?
3. What is Linear search?
4. Define Space Complexity
5. Define Time Complexity
6. What are asymptotic notations?

❖❖❖❖

6

APPLICATION OF SORTING AND WORKING WITH SORTED LISTS

Unit Structure

6.0 OBJECTIVE

- In this chapter we are going to be able to explain and understand the difference between searching and sorting.
- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order.
- Most common orders are in numerical or lexicographical order.
- In this chapter, we will discuss the Bubble sort Algorithm.
- A sorting algorithm is an algorithm that puts elements of a list in a certain order.
- The most used orders are numerical order and lexicographical order.
- Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human - readable input.

6.1 SORTING

- Sorting is the process of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed relationship.
- The items can be simple values, such as integers and reals, or more complex types, such as student records or dictionary entries.
- In either case, the ordering of the items is based on the value of a sort key.
- The key is the value itself when sorting simple types or it can be a specific component or a combination of components when sorting complex types.
- We encounter many examples of sorting in everyday life.
- Consider the listings of a phone book, the definitions in a dictionary, or the terms in an index, all of which are organized in alphabetical order to make finding an entry much easier.
- As we saw earlier in the chapter, the efficiency of some applications can be improved when working with sorted lists.
- Another common use of sorting is for the presentation of data in some organized fashion.
- For example, we may want to sort a class roster by student name, sort a list of cities by zip code or population, rank order SAT scores, or list entries on a bank statement by date.
- Sorting is one of the most studied problems in computer science and extensive research has been done in this area, resulting in many different algorithms.
- While Python provides a sort() method for sorting a list, it cannot be used with an array or other data structures.
- In addition, exploring the techniques used by some of the sorting algorithms for improving the efficiency of the sort problem may provide ideas that can be used with other types of problems.
- In this section, we present three basic sorting algorithms, all of which can be applied to data stored in a mutable sequence such as an array or list.

Sorting algorithms are often classified by :

- Computational complexity (worst, average and best case) in terms of the size of the list (N).

  For typical sorting algorithms good behaviour is O(NlogN) and worst case behaviour is O(N2) and the average case behaviour is O(N).

- Memory Utilization
- Stability - Maintaining relative order of records with equal keys.

- No. of comparisons.
- Methods applied like Insertion, exchange, selection, merging etc.
- Sorting is a process of linear ordering of lists of objects.
- Sorting techniques are categorized into
  - Internal Sorting
  - External Sorting
- Internal Sorting takes place in the main memory of a computer.
  - Example: - Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.
- External Sorting, takes place in the secondary memory of a computer, Since the number of objects to be sorted is too large to fit in main memory.
  - Example: - Merge Sort, Multiway Merge, Polyphase merge.

6.1.1 Difference between Searching and Sorting Algorithms

S.No.	Searching Algorithm	Sorting Algorithm
1.	Searching Algorithms are designed to retrieve an element from any data structure where it is used.	A Sorting Algorithm is used to arranging the data of list or array into some specific order.
2.	These algorithms are generally classified into two categories i.e. Sequential Search and Interval Search.	There are two different categories in sorting. These are Internal and External Sorting.
3.	The worst-case time complexity of searching algorithm is O(N).	The worst-case time complexity of many sorting algorithms like Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort is O(N2).

4.	There is no stable and unstable searching algorithms.	Bubble Sort, Insertion Sort, Merge Sort etc are the stable sorting algorithms whereas Quick Sort, Heap Sort etc are the unstable sorting algorithms.
5.	The Linear Search and the Binary Search are the examples of Searching Algorithms.	The Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort etc are the examples of Sorting Algorithms.

6.1.2 Bubble Sort

- The working procedure of bubble sort is simplest.
- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.
- It is not suitable for large data sets.
- The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.
- Bubble short is majorly used where -
    - complexity does not matter
    - simple and shortcode is preferred

Bubble Sort Algorithm:

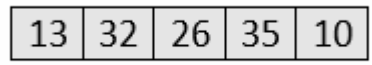
In the algorithm given below, suppose arr is an array of n elements.

The assumed swap function in the algorithm will swap the values of given array elements.

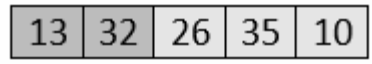
```
1. begin BubbleSort(arr)
2. for all array elements
3. if arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end if
6. end for
7. return arr
8. end BubbleSort
```

Working of Bubble sort Algorithm

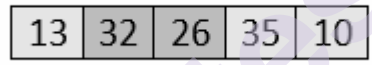
- To understand the working of bubble sort algorithm, let's take an unsorted array.
- We are taking a short and accurate array, as we know the complexity of bubble sort is **$O(n^2)$.**
- Let the elements of array are -



- **First Pass**
  - Sorting will start from the initial two elements. Let compare them to check which is greater.

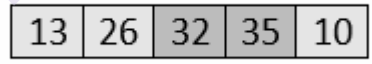


  - Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

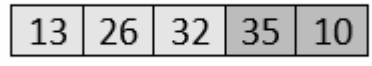


  - Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

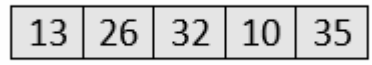
  - Now, compare 32 and 35.



  - Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
  - Now, the comparison will be in between 35 and 10.

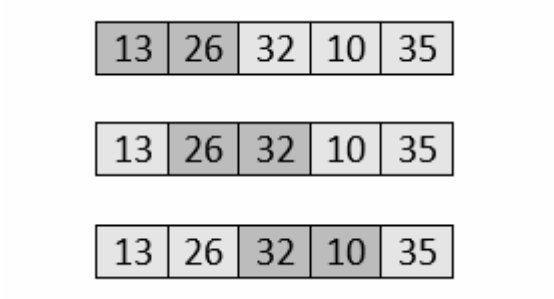


  - Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

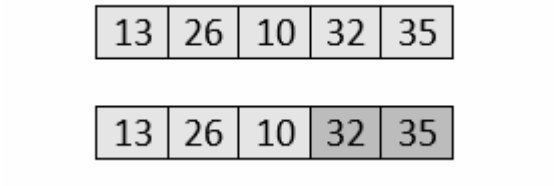


- Now, move to the second iteration.

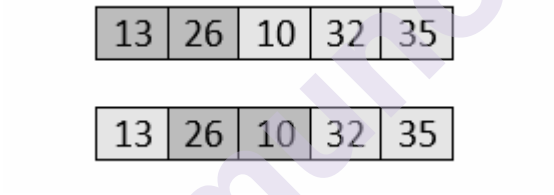
- **Second Pass**
  - The same process will be followed for second iteration.
  - The same process will be followed for second iteration.



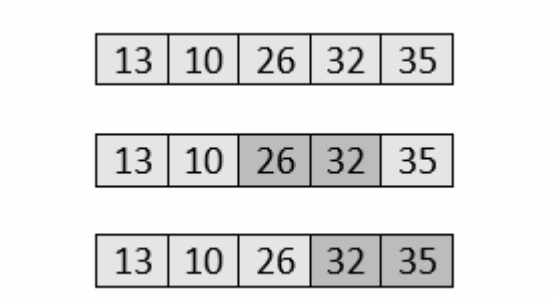
  - Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -



- Now, move to the third iteration.
- **Third Pass**
  - The same process will be followed for third iteration.

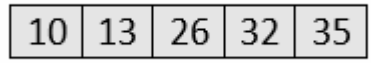


  - Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



  - Now, move to the fourth iteration.

- **Fourth pass**
  - Similarly, after the fourth iteration, the array will be -



  - Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

Optimized Bubble sort Algorithm

- In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.
- To solve it, we can use an extra variable ***swapped.*** It is set to **true** if swapping requires; otherwise, it is set to **false.**
- It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false.**
- It means that the elements are already sorted, and no further iterations are required.
- This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

```
1. bubbleSort(array)
2. n = length(array)
3. repeat
4. swapped = false
5. for i = 1 to n - 1
6. if array[i - 1] > array[i], then
7. swap(array[i - 1], array[i])
8.  swapped = true
9.  end if
10. end for
11. n = n - 1
12. until not swapped
13. end bubbleSort
```


6.1.3 Selection Sort

- In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.
- It is also the simplest algorithm. It is an in-place comparison sorting algorithm.
- In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part.
- Initially, the sorted part of the array is empty, and unsorted part is the given array.

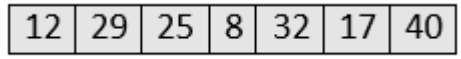
- Sorted part is placed at the left, while the unsorted part is placed at the right.
- In selection sort, the first smallest element is selected from the unsorted array and placed at the first position.
- After that second smallest element is selected and placed in the second position.
- The process continues until the array is entirely sorted.
- The average and worst-case complexity of selection sort is O(n2), where n is the number of items.
- Due to this, it is not suitable for large data sets.
- Selection sort is generally used when -
    - A small array is to be sorted
    - Swapping cost doesn't matter
    - It is compulsory to check all elements

The algorithm of selection sort:

```
1. SELECTION SORT(arr, n)
2.
3. Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
4. Step 2: CALL SMALLEST(arr, i, n, pos)
5. Step 3: SWAP arr[i] with arr[pos]
6. [END OF LOOP]
7. Step 4: EXIT
8.
9. SMALLEST (arr, i, n, pos)
10. Step 1: [INITIALIZE] SET SMALL = arr[i]
11. Step 2: [INITIALIZE] SET pos = i
12. Step 3: Repeat for j = i+1 to n
13. if (SMALL > arr[j])
14.     SET SMALL = arr[j]
15. SET pos = j
16. [END OF if]
17. [END OF LOOP]
18. Step 4: RETURN pos
```

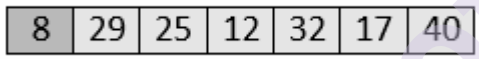
Working of Selection sort Algorithm

- Now, let's see the working of the Selection sort Algorithm.
- To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.
- Let the elements of array are -



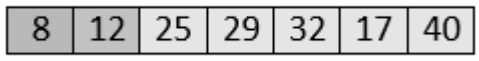
  - Now, for the first position in the sorted array, the entire array is to be scanned sequentially.
  - At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

  - So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

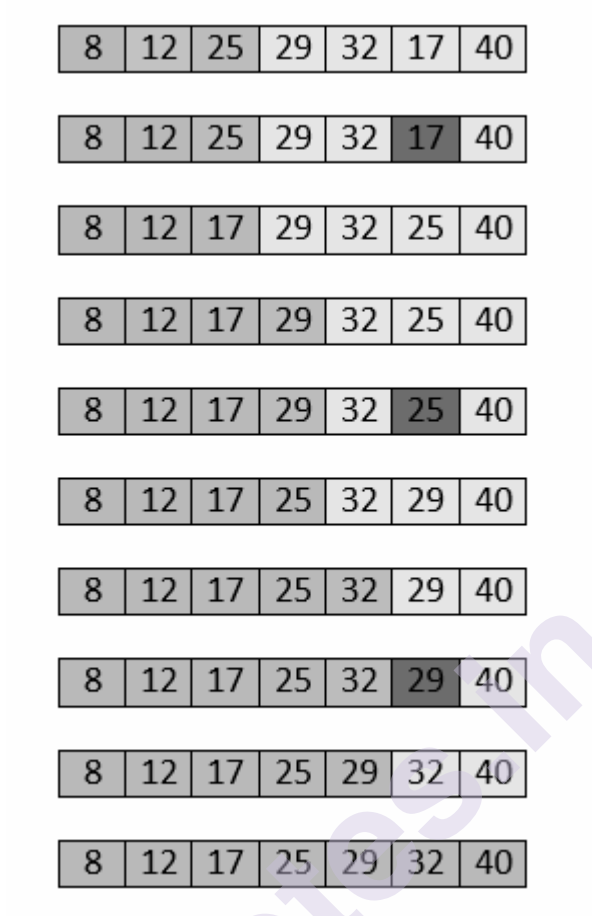


- For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array.
  - After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

- Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



- The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



- Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **$O(n^2)$**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **$O(n^2)$**.

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

6.1.4 Insertion Sort

- Insertion sort works similar to the sorting of playing cards in hands.
- It is assumed that the first card is already sorted in the card game, and then we select an unsorted card.
- If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side.
- Similarly, all unsorted cards are taken and put in their exact place.
- The same approach is applied in insertion sort.
- The idea behind the insertion sort is that first take one element, iterate it through the sorted array.
- Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items.
- Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.
- Insertion sort has various advantages such as -
  - Simple implementation
  - Efficient for small data sets
  - Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.
- Now, let's see the algorithm of insertion sort.

Algorithm:

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm:

- Now, let's see the working of the insertion sort Algorithm.
- To understand the working of the insertion sort algorithm, let's take an unsorted array.

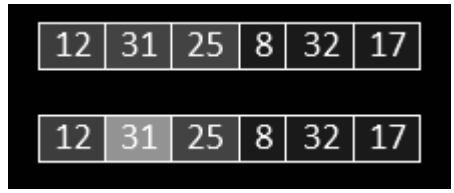
It will be easier to understand the insertion sort via an example.

- Let the elements of array are -

- Initially, the first two elements are compared in insertion sort.

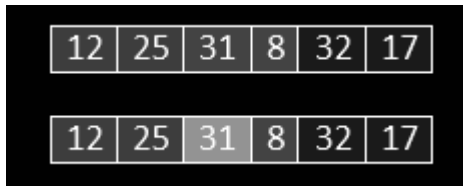
- Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

- Now, move to the next two elements and compare them.

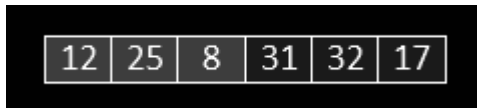


- Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.
- For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

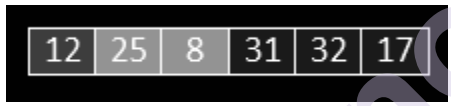
- Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



- Both 31 and 8 are not sorted. So, swap them.



- After swapping, elements 25 and 8 are unsorted.



- So, swap them.

- Now, elements 12 and 8 are unsorted.

- So, swap them too.

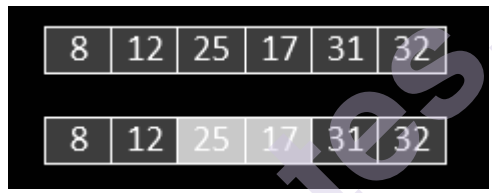
- Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

- Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

- Move to the next elements that are 32 and 17.

- 17 is smaller than 32. So, swap them.

- Swapping makes 31 and 17 unsorted. So, swap them too.



- Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

- Now, the array is completely sorted.

Insertion sort complexity'

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

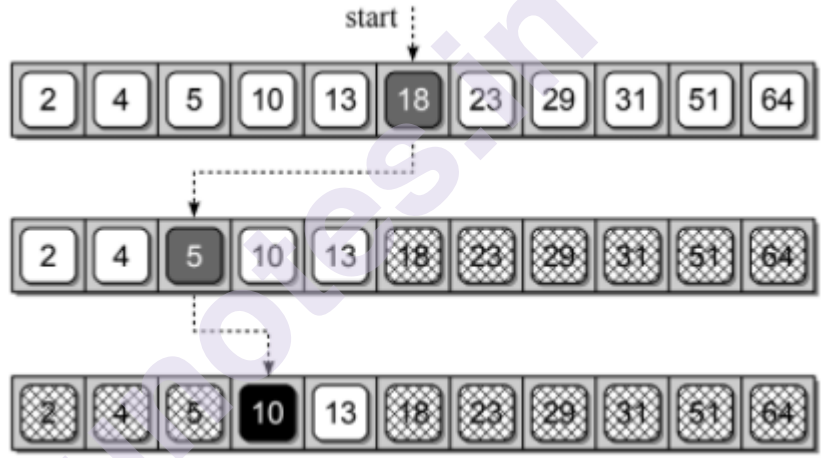
6.2 WORKING WITH SORTED LISTS

- The efficiency of some algorithms can be improved when working with sequences containing sorted values.
- We saw this earlier when performing a search using the binary search algorithm on a sorted sequence.
- Sorting algorithms can be used to create a sorted sequence, but they are typically applied to an unsorted sequence in which all of the values are known and the collection remains static.
- In other words, no new items will be added to the sequence nor will any be removed.
- In some problems, like the set abstract data type, the collection does not remain static but changes as new items are added and existing ones are removed.
- If a sorting algorithm were applied to the underlying list each time a new value is added to the set, the result would be highly inefficient since even the best sorting algorithm requires O(n log n) time.
- Instead, the sorted list can be maintained as the collection changes by inserting the new item into its proper position without having to re-sort the entire list.

- Note that while the sorting algorithms from the previous section all require $O(n^2)$ time in the worst case, there are more efficient sorting algorithms that only require O(n log n) time.

6.2.1 Maintaining a Sorted List

- To maintain a sorted list in real time, new items must be inserted into their proper position.
- The new items cannot simply be appended at the end of the list as they may be out of order.
- Instead, we must locate the proper position within the list and use the **insert() method** to insert it into the indicated position.
- Consider the sorted list from following Figure:

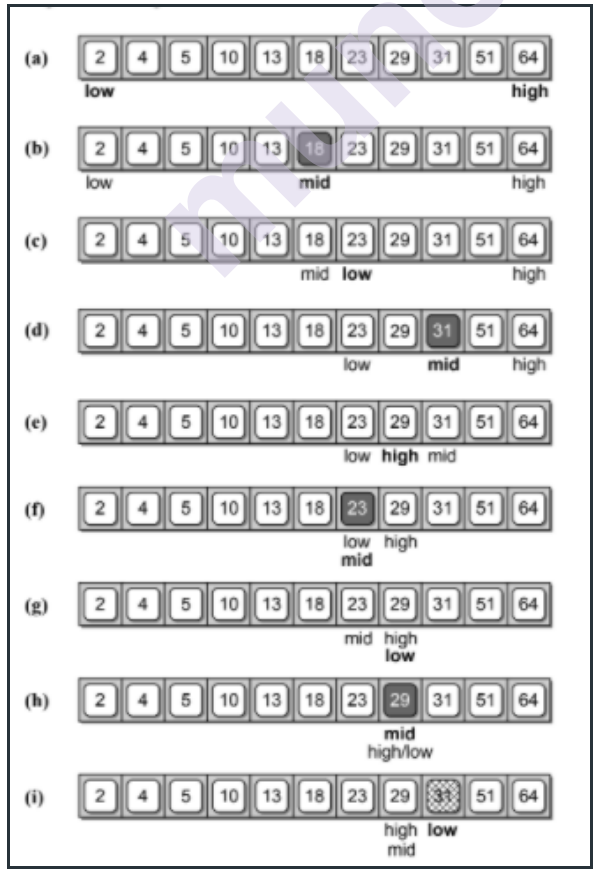


- If we want to add 25 to that list, then it must be inserted at position 7 following value 23.
- To find the position of a new item within a sorted list, a modified version of the binary search algorithm can be used.
- The binary search uses a divide and conquer strategy to reduce the number of items that must be examined to find a target item or to determine the target is not in the list.
- Instead of returning True or False indicating the existence of a value, we can modify the algorithm to return the index position of the target if it's actually in the list or where the value should be placed if it were inserted into the list.
- The modified version of the binary search algorithm is shown following:

Finding the location of a target value using the binary search.

```
1   # Modified version of the binary search that returns the index within
2   # a sorted sequence indicating where the target should be located.
3  def findSortedPosition( theList, target ):
4    low = 0
5    high = len(theList) - 1
6    while low <= high :
7      mid = (high + low) // 2
8      if theList[mid] == target :
9        return mid                    # Index of the target.
10     elif target < theList[mid] :
11       high = mid - 1
12     else :
13       low = mid + 1
14
15   return low           # Index where the target value should be.
```

- Note the change to the two return statements.
- If the target value is contained in the list, it will be found in the same fashion as was done in the original version of the algorithm.
- Instead of returning True, however, the new version returns its index position.
- When the target is not in the list, we need the algorithm to identify the position where it should be inserted.
- Consider the illustration in the following Figure,



- It shows the changes to the three variables low, mid, and high as the binary search algorithm progresses in searching for value 25.
- The while loop terminates when either the low or high range variable crosses the other, resulting in the condition low > high.
- Upon termination of the loop, the low variable will contain the position where the new value should be placed.
- This index can then be supplied to the insert() method to insert the new value into the list.
- The findOrderedPosition() function can also be used with lists containing duplicate values, but there is no guarantee where the new value will be placed in relation to the other duplicate values beyond the proper ordering requirement that they be adjacent.

6.2.2 Merging Sorted Lists:

- Sometimes it may be necessary to take two sorted lists and merge them to create a new sorted list.
- Consider the following code segment:

```
ListA = [ 2, 8, 15, 23, 37 ]

ListB = [ 4, 6, 15, 20 ]

newList = mergeSortedLists( listA, listB )

print( newList )
```

- which creates two lists with the items ordered in ascending order and then calls a user-defined function to create and return a new list created by merging the other two.
- Printing the new merged list produces

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```

Problem Solution

- This problem can be solved by simulating the action a person might take to merge two stacks of exam papers, each of which are in alphabetical order.
- Start by choosing the exam from the two stacks with the name that comes first in alphabetical order.
- Flip it over on the table to start a new stack.
- Again, choose the exam from the top of the two stacks that comes next in alphabetical order and flip it over and place it on top of first one.
- Repeat this process until one of the two original stacks is exhausted.

- The exams in the remaining stack can be flipped over on top of the new stack as they are already in alphabetical order and alphabetically follow the last exam flipped onto the new stack.
- A similar approach can be used to merge two sorted lists.
- Consider the illustration in the following Figure:

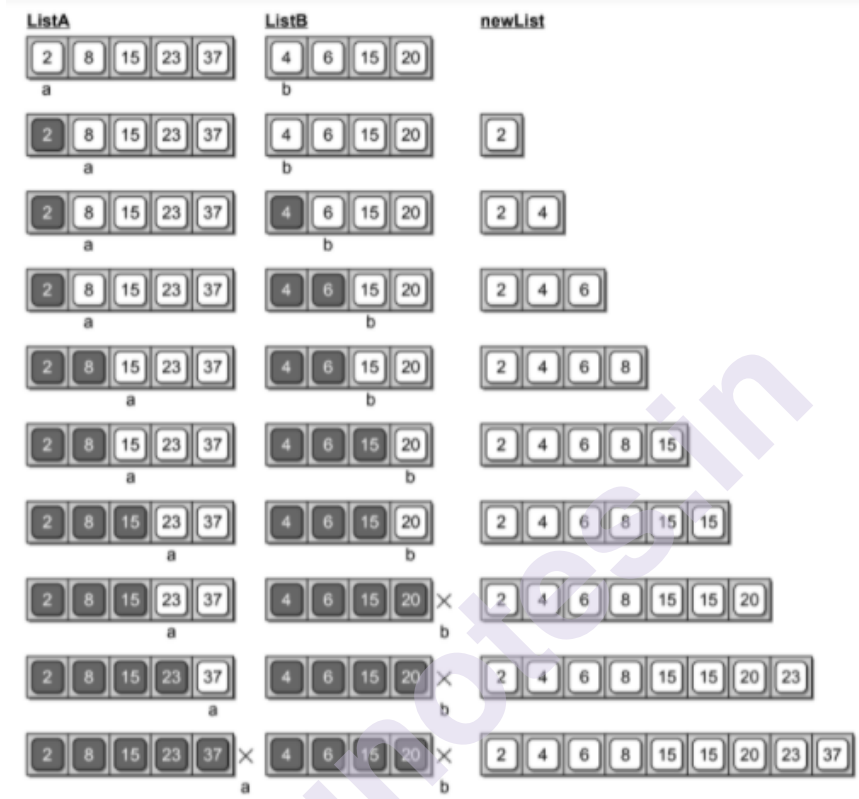


Figure 5.10: The iterative steps for merging two sorted lists into a new sorted list. a and b are index variables indicating the next value to be merged from the respective list.

- It demonstrates this process on the sample lists created in the example code segment from earlier.
- The items in the original list are not removed, but instead copied to the new list.
- Thus, there is no "top" item from which to select the smallest value as was the case in the example of merging two stacks of exams.
- Instead, index variables are used to indicate the "top" or next value within each list.
- The implementation of the mergeSortedLists() function is provided as following:

Listing 5.9 Merging two sorted lists.

```
1   # Merges two sorted lists to create and return a new sorted list.
2  def mergeSortedLists( listA, listB ) :
3     # Create the new list and initialize the list markers.
4    newList = list()
5    a = 0
6    b = 0
7
8     # Merge the two lists together until one is empty.
9    while a < len( listA ) and b < len( listB ) :
10     if listA[a] < listB[b] :
11       newList.append( listA[a] )
12       a += 1
13     else :
14       newList.append( listB[b] )
15       b += 1
16
17    # If listA contains more items, append them to newList.
18   while a < len( listA ) :
19     newList.append( listA[a] )
20     a += 1
21
22    # Or if listB contains more items, append them to newList.
23   while b < len( listB ) :
24     newList.append( listB[b] )
25     b += 1
26
27   return newList
```

- The process of merging the two lists begins by creating a new empty list and initializing the two index variables to zero.
- A loop is used to repeat the process of selecting the next largest value to be added to the new merged list.
- During the iteration of the loop, the value at listA[a] is compared to the value listB[b].
- The largest of these two values is added or appended to the new list.
- If the two values are equal, the value from listB is chosen.
- As values are copied from the two original lists to the new merged list, one of the two index variables a or b is incremented to indicate the next largest value in the corresponding list.
- This process is repeated until all of the values have been copied from one of the two lists, which occurs when a equals the length of listA or b equals the length of listB.
- Note that we could have created and initialized the new list with a sufficient number of elements to store all of the items from both listA and listB.
- While that works for this specific problem, we want to create a more general solution that we can easily modify for similar problems where the new list may not contain all of the items from the other two lists.
- After the first loop terminates, one of the two lists will be empty and one will contain at least one additional value.

- All of the values remaining in that list must be copied to the new merged list.
- This is done by the next two while loops, but only one will be executed depending on which list contains additional values.
- The position containing the next value to be copied is denoted by the respective index variable a or b.

Run Time Analysis

- To evaluate the solution for merging two sorted list, assume listA and listB each contain n items.
- The analysis depends on the number of iterations performed by each of the three loops, all of which perform the same action of copying a value from one of the two original lists to the new merged list.
- The first loop iterates until all of the values in one of the two original lists have been copied to the third.
- After the first loop terminates, only one of the next two loops will be executed, depending on which list still contains values.
- The first loop performs the maximum number of iterations when the selection of the next value to be copied alternates between the two lists.
- This results in all values from either listA or listB being copied to the newList and all but one value from the other for a total of $2n - 1$ iterations.
- Then, one of the next two loops will execute a single iteration in order to copy the last value to the newList.
- The minimum number of iterations performed by the first loop occurs when all values from one list are copied to the newList and none from the other.
- If the first loop copies the entire contents of listA to the newList, it will require n iterations followed by n iterations of the third loop to copy the values from listB.
- If the first loop copies the entire contents of listB to the newList, it will require n iterations followed by n iterations of the second loop to copy the values from listA.
- In both cases, the three loops are executed for a combined total of 2n iterations.
- Since the statements performed by each of the three loops all require constant time, merging two lists can be done in O(n) time.

6.3 IMPLEMENTATION USING PYTHON

6.3.1 Bubble Sort

Program: Write a program to implement bubble sort in python.

```
a = [35, 10, 31, 11, 26]
print("Before sorting array elements are - ")
for i in a:
    print(i, end = " ")
for i in range(0,len(a)):
    for j in range(i+1,len(a)):
        if a[j]<a[i]:
            temp = a[j]
            a[j]=a[i]
            a[i]=temp
print("\nAfter sorting array elements are - ")
for i in a:
    print(i, end = " ")
```

Output

```
Before sorting array elements are -
35 10 31 11 26
After sorting array elements are -
10 11 26 31 35
```

6.3.2 Selection Sort

Program: Write a program to implement selection sort in python.

```
def selection(a): # Function to implement selection sort
    for i in range(len(a)): # Traverse through all array elements
        small = i # minimum element in unsorted array
        for j in range(i+1, len(a)):
            if a[small] > a[j]:
                small = j
    # Swap the found minimum element with
    # the first element
        a[i], a[small] = a[small], a[i]
         def printArr(a): # function to print the array
      for i in range(len(a)):
        print (a[i], end = " ")
    a = [69, 14, 1, 50, 59]
print("Before sorting array elements are - ")
printArr(a)
selection(a)
print("\nAfter sorting array elements are - ")
selection(a)
printArr(a)
```

Output:

```
Before sorting array elements are -
69 14 1 50 59
After sorting array elements are -
1 14 50 59 69
```

6.3.3 Insertion Sort:

Program: Write a program to implement insertion sort in python.

```
def insertionSort(a): # Function to implement insertion sort
    for i in range(1, len(a)):
        temp = a[i]
         # Move the elements greater than temp to one position
        #ahead from their current position
        j = i-1
        while j >= 0 and temp < a[j] :
                a[j + 1] = a[j]
                j = j-1
        a[j + 1] = temp
  def printArr(a): # function to print the array
      for i in range(len(a)):
        print (a[i], end = " ")
  a = [70, 15, 2, 51, 60]
print("Before sorting array elements are - ")
printArr(a)
insertionSort(a)
print("\nAfter sorting array elements are - ")
printArr(a)
```

Output:

```
Before sorting array elements are -
70 15 2 51 60
After sorting array elements are -
2 15 51 60 70
```

6.4 EXERCISE:

Answer the following:

1. In this chapter, we used a modified version of the merge Sorted Lists() function to develop a linear time union() operation for our Set ADT implemented using a sorted list. Use a similar approach to implement new linear time versions of the is Subset Of (), intersect(), and difference() methods.

2. Given the following list of keys (80, 7, 24, 16, 43, 91, 35, 2, 19, 72), show the contents of the array after each iteration of the outer loop for the indicated algorithm when sorting in ascending order.

   (a) bubble sort (b) selection sort (c) insertion sort

3. Given the following list of keys (3, 18, 29, 32, 39, 44, 67, 75), show the contents of the array after each iteration of the outer loop for the

   (a) bubble sort (b) selection sort (c) insertion sort

4. Evaluate the insertion sort algorithm to determine the best case and the worst case time complexities.

Unit II

7

LINKED STRUCTURES

Unit Structure

7.0 OBJECTIVE

1. To understand the concept of linked list in Data Structures.
2. To understand the implementation of linked lists using python.
3. To learn different operations under the linked list.
4. To study the Bag ADT implementation.
5. To understand the application of linked lists.

7.1 INTRODUCTION

- Linked List can be defined as a collection of objects called nodes that are randomly stored in the memory.
- A data structure known as a linked list, which provides an alternative to an array-based sequence (also called a Python list).

- Both array-based sequences and linked lists keep elements in a certain type of order, but using a very different style.
- An array provides a more centric representation, with one large chunk of memory capable of accommodating references to many elements.
- A linked list, in contrast, relies on a more distributed representation in which a lightweight object, known as a node, is allocated for each element.
- Each node maintains a reference to its element and one or more references to next to nodes in order to collectively represent the linear order of the sequence.
- The Python list, which is also a sequence container, is an abstract sequence type implemented using an array structure. It gives more the functionality of an array by providing a larger set of operations than the array, and it can automatically adjust in size as items are added or removed.
- The linked list arrangement, which may be a general purpose structure which will be wont to store a set in linear order. The linked list improves on the development and management of an array and Python list by requiring smaller memory allocations and no element shifts for insertions and deletions.
- There are several sorts of linked lists. The singly linked list may be a linear structure during which traversals start at the front and progress, one element at a time, to the top. Other variations include the circularly linked, the doubly linked, and therefore the circularly doubly linked lists.

```
classListNode :
def __init__( self, data ) :
self.data = data
```

- In Linked List we create several instances of this class which is called ListNode, each storing data of our choosing. within the following example, we create three instances, each storing an integer value:

```
a = ListNode( 11 )
b = ListNode( 52 )
c = ListNode( 18 )
```

- This three node create three variables and three objects :

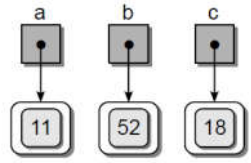


Fig. 1

- Add a second node or data field to the ListNode class:

```
classListNode :
def __init__( self, data ) :
self.data = data
self.next = None
```

- The three objects from the previous figure would now have a second node or data field initialized with a NULL reference, as shown in the following fig. 2:

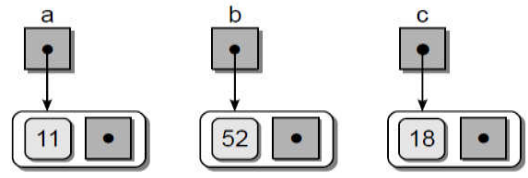


Fig. 2

- Since subsequent fields can contain a regard to any sort of object, we will assign thereto a regard to one among the opposite ListNode objects. For example, suppose we assign b to the next field of object a:

a.next = b

- which output in object a being linked to object b, as shown below.

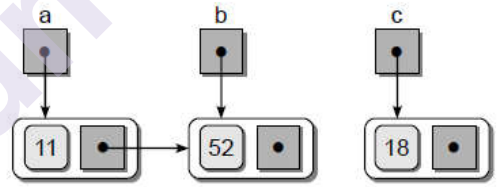


Fig.3

- And at the end, link object b to object c:

b.next = c

- resulting in a series of objects, as shown here.

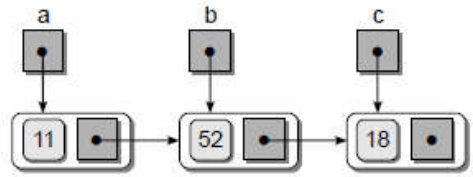


Fig. 4

- In Linked List, remove the two external references b and c by assigning none to each, as shown below

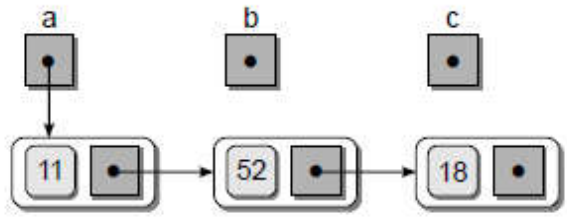


Fig. 5

- The final result is a linked list structure. The two objects previously pointed to by b and c are still accessible via a. For example, suppose we wanted to print the values of the three objects. We can access the other two objects through the next field of the first object:

```
print(a.data )

print(a.next.data )

print(a.next.next.data )
```

- A linked structure contains a collection of objects called nodes, each of which contains data and at least one reference or pointer or link to another node. A linked list is a linked structure in which the nodes are connected in order to form a linear list.

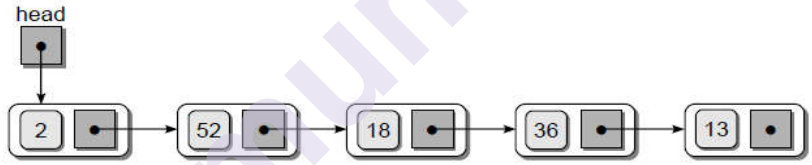


Fig. 6

- The above Linked list provides an example of a linked list consisting of 5 nodes. The last node within the list, commonly called the tail node, is indicated by a NULL link reference. Most nodes within the list haven't any name and are simply referenced via the link field of the preceding node.

- The first node within the list, however, must be named or referenced by an external variable because it provides an entry point into the linked list.

- This variable is usually referred to as the top pointer, or head reference. A linked list also can be empty, which is indicated when the top reference is NULL.

7.2 SINGLY LINKED LIST:

- A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence.
- Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.+
- A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

7.2.1 Traversing.

- Traversing a linked list requires the initialization and adjustment of a temporary external reference variable.
- Step by Step Linked List Traversing:

1. After initializing the temporary external reference.

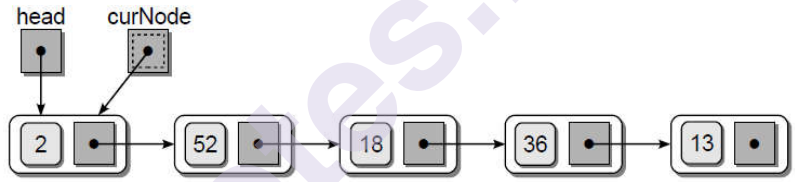


Fig. 7

2. Advancing the external reference after printing value 2.

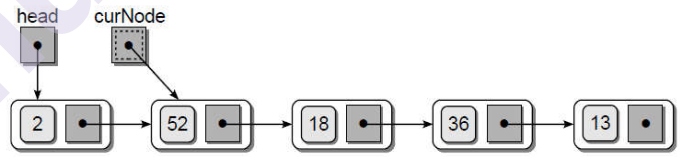


Fig. 8

3. Advancing the external reference after printing value 52.

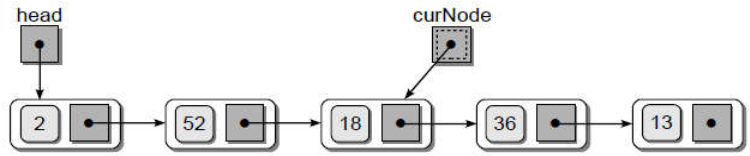


Fig. 9

4. Advancing the external reference after printing value 18.

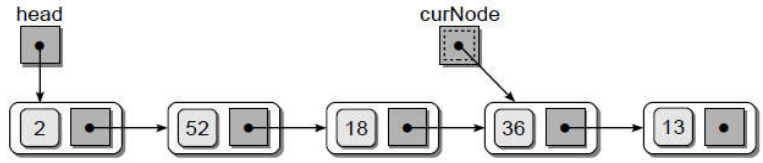


Fig. 10

5. Advancing the external reference after printing value 36.

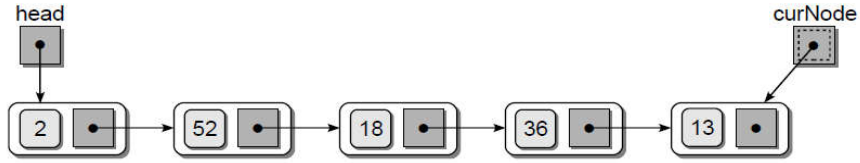


Fig. 11

6. The external reference is set to None after printing value 13.

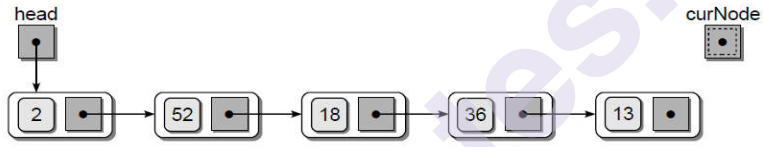


Fig. 12

Implementation of the linked list:

```
1  def traversal( head ):
2  currentNode = head
3  while currentNode is not None :
4  print currentNode.data
5 currentNode = currentNode.next
```

Example:

```
Code:
class Node:
def __init__(self, datavalue1=None):
    self.datavalue1 = datavalue1
    self.nextvalue1 = None
class SLinkedList:
def __init__(self):
    self.headvalue1 = None
deflistprint(self):
    printvalue1 = self.headvalue1
    while printvalue1 is not None:
      print (printvalue1.datavalue1)
      printvalue1 = printvalue1.nextvalue1
list = SLinkedList()
list.headvalue1 = Node("One")
e2 = Node("Two")
e3 = Node("Three")
# Link 1st Node to 2nd node
list.headvalue1.nextvalue1=e2
# Link 2nd Node to 3rd node
e2.nextvalue1=e3
list.listprint()
Output:
One
Two
Three
```

7.2.2 Searching

- A linear or sequential search operation can be carried out on a linked list. It is very closely the same as the traversal operation. The only difference is that the loop can stop early if we end the target value within the list.

Implementation of the linear search:

```
1  defunorderedSearch( head, target ):
2  currentNode = head
3  while currentNode is not None and currentNode.data != target :
4  currentNode= currentNode.next
5  return currentNode is not None
```

- About two logic in the while loop. It is important that we test for a NULL currentNode reference before trying to examine the contents of the node.
- If the item is not found in the list, currentNode will be NULL when the end of the list is reached.
- If we try to evaluate the data field of the NULL reference, an exception will be raised, resulting in a run-time error.
- A NULL reference does not point to an object and thus there are no fields or methods to be referenced.
- When we use the search operation for the linked list, we must make sure that it works with both empty and non-empty lists.
- In this fact, we do not need a separate test to identify if the list is empty. This is done automatically by checking the traversal variable as the loop condition. If the list were vacant, currentNode would be set to None initially and the loop would never be entered.
- In the linked list search operation needs O(n) in the worst case, which occurs when the target item is not in the list.

7.2.3 Prepending Nodes

- When operating with an unordered list, new values can be inserted at any point within the list. We only maintain the head reference as part of the list structure, we can easily prepend new items with little effort.
- The implementation is shown below. Prepending a node can be done in constant time hence no traversal operation is required.

```
1 def traversal( head ):
2 currentNode = head
3 while currentNode is not None :
4 print currentNode.data
5 currentNode = currentNode.next
```

Prepending a node to the linked list.

```
1 # Shown in the head pointer, prepend an item to an unsorted linked list.
2 newNode = ListNode( item )
3 newNode.next = head
4 head = newNode
```

- If we insert the value 96 to our example shown in Figure 13a.
- Adding an item to the front of the list requires many steps.
  - First, create a new node to store the new value and then set its next field to point to the node present at the front of the list.
  - We then modify head to point to the new node; it is now the first node in the list. These steps are represented as dashed lines which is shown in figure 13b.
  - Then we first link the new node into the list before modifying the head reference.
  - Or else, we lose our external reference to the list and in turn, we lose the list itself. Then linking the new node into the list, are shown in figure 13c
  - When altering or changing links in a linked list, we consider the case when the list is empty.For our implementation, the code works perfectly since the head reference will be NULL when the list is empty and the rst node inserted needs the next eld set to None.

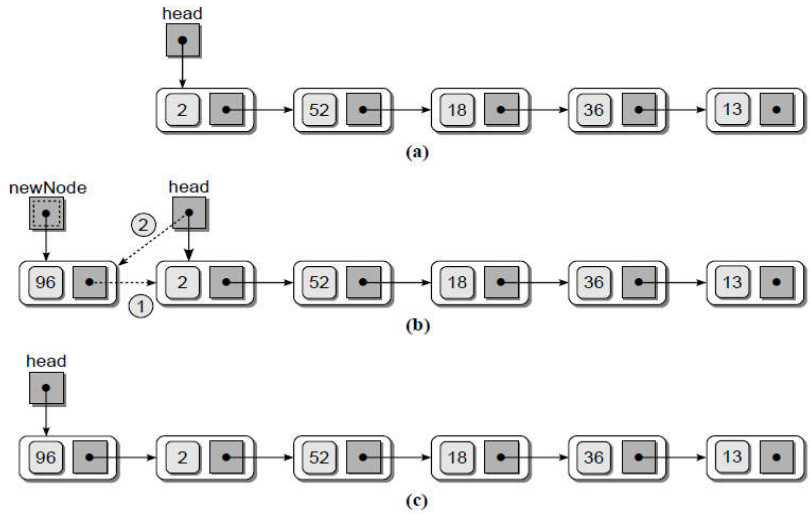


Fig. 13

Prepending a node to the linked list:

(a) The original list

(b) Link modifications required to prepend the node; and

(c) The result after prepending 96.

7.2.4 Removing Nodes

- An item or data delete from a linked list by removing or disjoining the node containing that item or data.
- The linked list as shown in the figure 14c and take it that we remove the node containing 18. First, we must find the node containing the target value and place an external reference variable pointing to it, as shown in figure 14a. After finding the node, it has to be unlinked from the list, which necessitates adjusting the link field of the node's predecessor to point to its successor as shown in figure 14b. The node's link field is also freed by setting it to none.

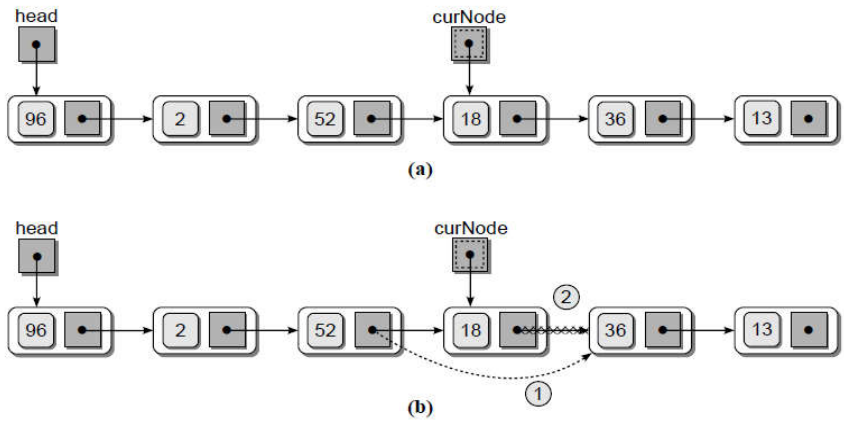


Fig. 14

Removing a node from a linked list:

1. Finding the node that need to be removed and assigning an external reference variable
2. The link alteration required to unlink and remove a node.

7.3 BAG ADT REVISITED

- A bag is a simple container like a shopping bag that can be used to store a collection of items.
- The bag container restricts access to the individual items by only dening operations for adding and removing individual items, for determining if an item is in the bag, and for traversing over the collection of items.
- The Date ADT provided an example of a simple abstract data type. To explain the design and implementation of a complex abstract data type, we define the Bag ADT.

The Bag Abstract Data Type

- There are many variations of the Bag ADT with the one illustrated here being a simple bag.
- A grab bag is the same as the simple bag but the items are removed from the bag at random.
- Additional Common variation is the counting bag, which includes an operation that returns the number of circumstances in the bag of a given item.
- A bag is a holder that stores a collection in which duplicate values are allowed. The items, each of which is differently stored, have no particular order but they must be comparable.

  ➢ Bag(): Creates a bag that is initially empty.

  ➢ length (): Returns the number of items stored in the bag. Accessed using the len() function.

  ➢ contains (item): Finding if the given target item is stored in the bag and returns the applicable boolean value. Acquired using the in operator.

  ➢ Add (item): Given item added to the bag.

  ➢ Remove (item): Erase and return an occurrence of an item from the bag.An exception is raised if the element is not in the bag.

  ➢ iterator (): Creates and returns an iterator that can be used to iterate over the collection of items.

Linked List Implementation

- The linked list implementation of the Bag ADT can be done with the constructor. Initially, the head field will store the head pointer of the linked list. The reference pointer is initiated to None to represent an empty bag.
- The size field is used to keep track of the number of items stored in the bag that is required by the len() method. This field is not needed. But it does avoid us from having to traverse the list to count the number of nodes each time the length is required. Define only a head pointer as a data field in the object. Short live references such as the currentNode reference used to traverse the list are not defined as attributes, but instead as local variables within the individual methods as needed.
- A sample instance of the new Bag class is shown in Figure 15

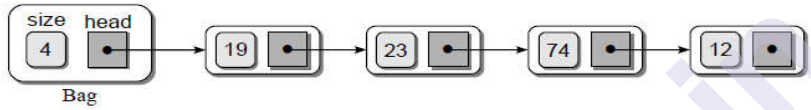


Fig. 15

Sample instance of the Bag class

- The contains () method is a simple search of the linked list, The add() method simply implements the prepend operation, though we must also increment the item counter (size) as new items are added.
- The Bag List Node class, used to represent the individual nodes, is also denied within the same module.

7.4 COMPARING IMPLEMENTATIONS

- The Python list and the linked list can both be used to handle the elements stored in a bag.
- Both Python list and linked list implementations provide the same time complexities for the various operations with the exception of the add() method.
- When inserting an item to a bag executed using a Python list, the item is appended to the list, which requires O(n) time in the worst case since the underlying array may have to be expanded.
- In the linked list version of the Bag ADT, a new bag item is stored in a new node that is prepended to the linked structure, which only requires O(1). Fig. 16 shows the time-complexities for two implementations of the Bag ADT.

Operation	Python List	Linked List
`b = Bag()`	$O(1)$	$O(1)$
`n = len(b)`	$O(1)$	$O(1)$
`x in b`	$O(n)$	$O(n)$
`b.add(x)`	$O(n)$	$O(1)$
`b.remove(x)`	$O(n)$	$O(n)$
traversal	$O(n)$	$O(n)$

Fig. 16

7.5 LINKED LIST ITERATORS

- An iterator for Bag ADT executes using a linked list as we did for the one implemented using a Python list.
- The process is the same, but our iterator class would have to keep track of the current node in the linked list instead of the current element in the Python list.
- By implementing a bag iterator class as listed below, which is inserted within the llistbag.py module which will be wont to iterate over the linked list.

An iterator for the Bag class implemented using a linked list.

```
1 # Defines a linked list iterator for the Bag ADT.
2  class _BagIterator :
3  def __init__( self, listHead ):
4  self._currentNode = listHead
5
6  def __iter__( self ):
7  return self
8
9   def next( self ):
10 if self._currentNode is None :
11 raise StopIteration
12 else :
13 item = self._currentNode.item
14 self._currentNode = self._currentNode.next
15 return item
```

- When repeated over a linked list, we need only keep track of the current node being processed and thus we use a single data held currentNode in the iterator.
- The linked list as the for loop iterates over the nodes.
- Figure 17 shows the Bag and BagIterator objects at the beginning of the for loop.
- The currentNode pointer in the BagIterator object is used just like the currentNode pointer we used when directly performing a linked list traversal.
- The difference is that we do not include a while loop since Python manages the iteration for us as part of the for loop.
- The iterator objects can be used with singly linked list configuration to traverse the nodes and return the data consist in each one.

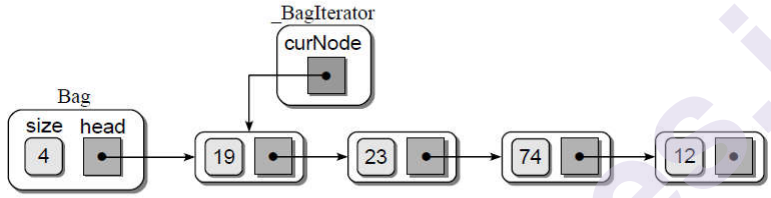


Fig. 17

7.6 MORE WAYS TO BUILD LINKED LISTS

- New nodes can be easily added to a linked list by prepending them to the linked structure.
- This is sufficient when the linked list is used to implement a basic container in which a linear order is not needed, such as with the Bag ADT. But a linked list can also be used to implement a container abstract data type that requires a specific linear ordering of its elements, such as with a Vector ADT.
- In the case of the Set ADT, it can be improved if we have access to the end of the list or if the nodes are sorted by element value.

  ○ **Use of Tail Reference**

  1. The use of a single external reference to point to the head of a linked list is enough for many applications.
  2. In some types, this needs to append items to the end of the list.
  3. Including a new node to the list using only a head reference requires linear time since a complete traversal is required to reach the end of the list.

4. Instead of a single external head reference, we have to use two external references, one for the head and one for the tail. Figure 18 shows a sample linked list with both a head and a tail reference.

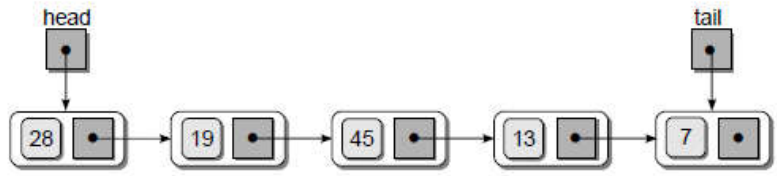


Fig. 18 Sample linked list using both head and tail external references

- **The Sorted Linked List**

  1. The items in a linked list can be sorted in ascending or descending order as was done with a sequence. Consider the sorted linked list illustrated in Figure 19

  2. The sorted list has to be created and maintained as items are added and removed.

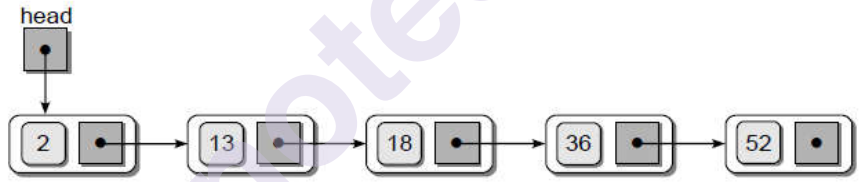


Fig. 19 A sorted linked list (ascending order.)

7.7 APPLICATION-POLYNOMIALS

- Arithmetic expressions specified in terms of variables and constants. A polynomial in one variable can be indicatedin expanded form:

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x^1 + a_0$$

- Where each $a_i x^i$ a component is called a term.
- The a_i part of the term, which is a scalar that can be zero, is called the coefficient of the term.
- The exponent of the x^ipart is called the degree of that variable and is limited to whole numbers. For Example,

- $12x^2 - 3x + 7$

- Consists of three terms.
    - The first term, $12x^2$, is of degree 2 and has a coefficient of 12
    - the second term, -3x, is of degree 1 and has a coefficient of -3
    - The last term, while constant, is of degree 0 with a coefficient of 7.
    - Polynomials can be characterized by degree (i.e., all second-degree polynomials).
    - The degree of a polynomial is the largest single degree of its terms.
    - The example polynomial above has a degree of 2 since the degree of the first term$12x^2$ has the largest degree.
- Design and implement an abstract data type to represent polynomials in one variable expressed in expanded form.

Polynomial Operations

A number of operations can be performed on polynomials. Let start with addition operation.

- **Addition**

  Two polynomials can be added the coefficients of corresponding terms of equal degree. The result is a third polynomial.

  $$5x^2 + 3x - 10$$
  $$2x^3 + 4x^2 + 3$$

  Which we can add to formed a new polynomial:

$$(5x^2 + 3x - 10) + (2x^3 + 4x^2 + 3) = 2x^3 + 9x^2 + 3x - 7$$

  Subtraction of polynomial done with same method but the coefficients is subtracted. To view polynomial addition is to align terms by degree and add the corresponding coefficients:

$$\begin{array}{rrrrr} & & 5x^2 & 3x & -10 \\ + & 2x^3 & 4x^2 & & 3 \\ \hline & 2x^3 & 9x^2 & 3x & -7 \end{array}$$

- **Multiplication**

  The product of two polynomials is also a third polynomial. The new polynomial is finding by summing the result from multiplying each term of the first polynomial by each term of the second.

$$(5x^2 + 3x - 10)(2x^3 + 4x^2 + 3)$$

The second polynomial has to be multiplied by each term of the first polynomial:

$$5x^2(2x^3 + 4x^2 + 3) + 3x(2x^3 + 4x^2 + 3) + -10(2x^3 + 4x^2 + 3)$$

We then distribute the terms of the first polynomial to yield three intermediate polynomials:

$$(10x^5 + 20x^4 + 15x^2) + (6x^4 + 12x^3 + 9x) + (-20x^3 - 40x^2 - 30)$$

Finally, the three polynomials are summed, resulting in

$$10x^5 + 26x^4 - 8x^3 - 25x^2 + 9x - 30$$

- **Evaluation**

  The evaluation is an easiest operation of a polynomial. Polynomials can be evaluated by assigning a value to the variable, commonly called the unknown. By making the variable known in specifying a value, the expression can be calculated, resulting in a real value. If we assign value 3 to the variable x in the equation

$$10x^5 + 26x^4 - 8x^3 - 25x^2 + 9x - 30$$

the result will be

$$10(3)^5 + 26(3)^4 - 8(3)^3 - 25(3)^2 + 9(3) - 30 = 4092$$

7.8 SUMMARY

1. The linked list improves on the construction and management of an array and Python list by requiring smaller memory allocations and no element required to shift for insertions and deletions.
2. A singly linked list, in its easiest form, is a collection of nodes that combine to form a linear sequence.
3. A bag is a simple container like a shopping bag that needs to be used to store a collection of items.
4. The Python list and the linked list can be used to handle the elements stored in a bag. Both implementations give the same time-complexities for the various operations with the exception of the add () method.

7.9 REFERENCE FOR FURTHER READING

1. Data Structure and algorithm Using Python, Rance D. Necaise, 2016 Wiley India Edition
2. Data Structure and Algorithms in Python, Michael T. Goodrich, Robertom Tamassia, M. H. Goldwasser, 2016 Wiley India Edition

7.10 UNIT END EXERCISES

1. What is a Linked List? Explain Singly Linked List with different operations.
2. What is the meaning of Bag ADT Revisited?
3. Write a short note on"

   a. Linked List Iterators

   b. Application-Polynomials

8

STACKS

Unit Structure

8.0 OBJECTIVE

1. To understand the concept of Stacks in Data Structures.
2. To understand the implementation of stack using python.
3. To learn different operations under the stack.
4. To study the Stack ADT implementation.
5. To understand the applications of stacks.

8.1 INTRODUCTION

- Python list and linked list structures to implement a different container ADT.
- The stack, which may be a sort of container with restricted access that stores a linear collection.
- Stacks are very common in computer science and utilized in many types of problems.
- Stacks also occur in our everyday lives.
- Consider a stack of trays. When a tray is taken away from the top, the others shift up. If trays are kept onto the stack, the others are pushed down.

8.2 STACK ADT

- A stack is used to store data like the last item inserted is the first item removed.
- It is used to implement a last-in first-out (LIFO) type of data structure.
- The stack is a linear data structure in which new items are added at the end, or existing items are removed from the same end, commonly called at the top of the stack.
- The opposite end is known as the base of the stack.
- Example shown in figure 7.1, which illustrates new values being added to the top of the stack and one value being removed from the top.

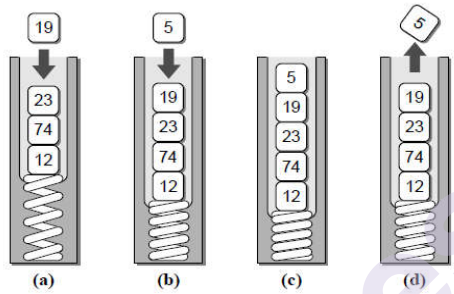


Fig. 1 Abstract view of a stack:

Above figure shows

(a) Pushing value 19;

(b) Pushing value 5;

(c) Resulting stack after 19 and 5 are added

(d) Popping top value

A stack is a data structure that stores a linear collection of objects with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the top of the stack. An empty stack is one containing no items.

- Stack(): Creates a new (empty) stack.
- isEmpty(): Returns a Boolean value if the stack is empty.
- length (): Returns the number of elements in the stack.
- pop(): Removes and returns the top element of the stack, if the stack is not empty. Items cannot be removed from an empty stack. The next item on the stack becomes the new top item.
- peek (): Use as a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.
- push (element): Adds the given element to the top of the stack.

Stack Example:

```
PROMPT = "Enter an integer value (<0 to end):"
myStack1 = Stack()
value = int(input( PROMPT ))
while value >= 0 :
myStack1.push( value )
value = int(input( PROMPT ))
while not myStack1.isEmpty() :
value = myStack1.pop()
print(value)
```

8.3 IMPLEMENTING STACKS-

- The two most common methods to implement the stack in Python include the use of a Python list and a linked list. The choice depends on the type of application we are going to use.

8.3.1 Using Python List

- The Python list-based implementation of the Stack ADT is very simple to implement.
- The first decision we have to make when using the list for the Stack ADT is which end of the list to use as the top and which as the base.
- For the most efficient ordering, we let the end of the list represent the top of the stack and the front defines the base.
- As the stack increases, items are appended to the end of the list and when items are popped, they are removed from the same end. Below Listing provides the complete implementation of the Stack ADT using a Python list.

```
# Implementation of the Stack ADT using a Python list.
class Stack :
# Creates an empty stack.
def __init__( self ):
self._theItems = list()
```

```
# Returns True if the stack is empty or False otherwise.
defisEmpty( self ):
return len( self ) == 0
# Returns the number of items in the stack.
def __len__ ( self ):
return len( self._theItems )
# Returns the top item on the stack without removing it.
def peek( self ):
assert not self.isEmpty(), "Cannot peek at an empty stack"
return self._theItems[-1]
# Removes and returns the top item on the stack.
def pop( self ):
assert not self.isEmpty(), "Cannot pop from an empty stack"
return self._theItems.pop()
# Push an item onto the top of the stack.
def push( self, item ):
self._theItems.append( item )
```

- The peek() and pop() operations can only be used with a non-empty stack.
- To accomplish this requirement, until we first assert the stack is not empty before performing the given operation.
- The peek() method directly returns a reference to the last item in the list.
- To implement the pop() method, call the pop() method of the list structure, which actually performs the same operation that we are striving to implement. This will save a copy of the last item in the list, delete the item from the list, and then return the saved item.
- The push() method simply inserts new items to the end of the list since that represents the top of our stack.
- The Separate stack operations are simple to evaluate for the Python list-based implementation. isEmpty(), len , and peek() only require O(1) time.
- The pop() and push() methods both require O(n) time in the worst case, When used in sequence, both operations have an restitution cost of O(1).

8.3.2 Using Linked List

- The Python list based implementation may not be the excellent choice for stacks with a huge number of push and pop operations.
- Keep in mind that, each append() and pop() list operation may require a reallocation of the underlying array used to implement the list.
- A singly linked list can be used to carry out the Stack ADT, helping the concern over array reallocations.
- The Stack ADT implemented using a linked list is shown below:

```
def __init__( self ):
self._top = None
self._size = 0
# Returns True if the stack is empty or False otherwise.
defisEmpty( self ):
return self._top is None
# Returns the number of items in the stack.
def __len__( self ):
return self._size
# Returns the top item on the stack without removing it.
def peek( self ):
assert not self.isEmpty(), "Cannot peek at an empty stack"
return self._top.item
# Removes and returns the top item on the stack.
def pop( self ):
assert not self.isEmpty(), "Cannot pop from an empty stack"
node = self._top
self.top = self._top.next
self._size -= 1
return node.item
# Pushes an item onto the top of the stack.
def push( self, item ) :
self._top = _StackNode( item, self._top )
self._size += 1
# The private storage class for creating stack nodes.
class _StackNode :
def __init__( self, item, link ) :
self.item = item
self.next = link
```

- The class constructor produces two instance variables for each Stack.
- The top field is the head reference for preserving the linked list while size is an integer value for keeping track of the number of items on the stack.
- The concluding has to be adjusted when items are pushed onto or popped off the stack.
- Figure 2 shows a sample Stack object for the stack from Figure 1b.
- The StackNode class is employed to make the linked list nodes.
- Note the inclusion of the link argument within the constructor, which is employed to initialize subsequent fields of the new node.
- By including this argument, we will simplify the prepend operation of the push () method.
- The 2 steps required to prepend a node to a linked list are combined by passing the top regard to p because of the second argument of the StackNode() constructor.

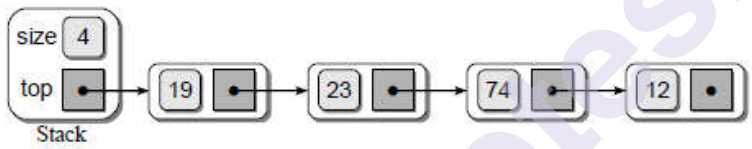


Fig. 2 Stack ADT implemented as a linked list.

- The peek () method simply returns a reference to the data item in the first node after checking the stack is not empty.
- If the method were used on the stack represented by the linked list in Figure 2, a reference to 19 would be returned.
- The peek operation only needs to identify the item on top of the stack. It should not be used to alter the top item as this would invade the definition of the Stack ADT.
- The pop () method perpetually removes the first node in the list. This operation is shown in figure 3a.
- This is easy to implement and does not require a search to end the node containing a specific item.
- The result of the linked list after removing the top item from the stack is shown in Figure 3b.
- The linked list implementation of the Stack ADT is more systematic than the Python-list based implementation. All the operations above are O(1) in the worst case, the proof of which is left as an exercise.

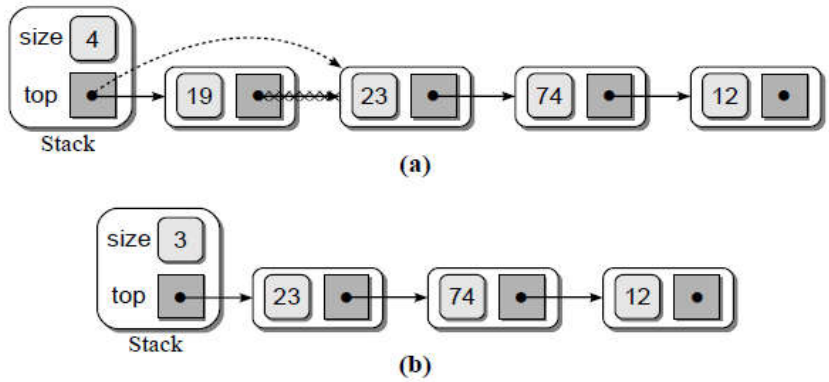


Fig. 3 popping an item from the stack:

8.4 STACK APPLICATIONS

The Stack ADT is essential for a number of applications in CS.

8.4.1 Balanced Delimiters

- A number of applications use delimiters to group strings of text or simple data into subparts by marking the beginning and end of the group. Some common examples include mathematical expressions, programming languages, and the HTML markup language used by web browsers. There are typically strict rules as to how the delimiters can be used, which includes the requirement of the delimiters being paired and balanced. Parentheses can be used in mathematical expressions to override the order of precedence for various operations.
- To assist in the reading of complicated expressions, we can use different types of symbol pairs, as shown here.

  {A + (B * C) - (D / [E + F])}
- The delimiters should be used in pairs of match types: {}, [], and ().
- They must also be positioned such that an opening delimiter within an outer pair must be closed within the same outer pair.
- For example, the following expression would be invalid since the pair of braces [] begin inside the pair of parentheses () but end outside.

  (A + [B * C)] - {D / E}

Following code shows the implementing a function to compute and return the sum of integer values contained in an array:

Example:

```
intsumList( inttheList[], int size )
{
int sum = 0;
int i = 0;
        while( i < size )
        {
                sum += theList[ i ];
                i += 1;
        }
return sum;
}
```

Table 1 shows the steps performed by our algorithm and the contents of the stack after each delimiter is encountered in given sample code

Operation	Stack	Current scan line
push (	(	int sumList(
push [	([	int sumList(int values[
pop & match]	(	int sumList(int values[]
pop & match)		int sumList(int values[], int size)
push {	{	{
	{	int sum = 0;
	{	int i = 0;
push (	{ (	while(
pop & match)	{	while(i < size)
push {	{ {	while(i < size) {
push [	{ { [	sum += theList[
pop & match]	{ {	sum += theList[i]
	{ {	i += 1;
pop & match }	{	}
	{	return sum;
pop & match }	empty	}

Table 1. Content of the Stack

- The sequence of steps scanning a valid set of delimiters:
  - The operation performed (left column)
  - The contents of the stack (middle column) as each delimiter are encountered (right column) in the code.

- The delimiters are balanced with an equal number of opening and closing delimiters
- If the delimiters are not balanced properly then we encounter more opening or closing delimiters.
- For example, if the programmer initiate a typographical error in the function header:

  intsum List (intthe List)], int size)

- In Table 2 the stack is empty since the left parenthesis was popped and matched with the preceding right parenthesis.
- Like so unbalanced delimiters in which there are more closing delimiters than opening ones can be detected when trying to pop from the stack and we determine the stack is empty.

Operation	Stack	Current point of scan
push (	(	int sumList(
pop & match)	empty	int sumList(int values)
pop & match]	error	int sumList(int values)]

Scanning: int sumList(int values)], int size)

Operation	Stack	Current point of scan
push (	(	int sumList(
push (	((	int sumList(int (
push [	(([	int sumList(int (values[
pop & match]	((	int sumList(int values[]
pop & match)	(	int sumList(int values[], int size)

Scanning: int sumList(int (values[], int size)

Table 2 Sequence of steps

- A stack is used to store the opening delimiters and either implementation can be used, we have selected to use the linked list.

Function for validating a C++ source file:

```
# Implementation of the algorithm for validating balanced brackets in
# a C++ source file.
from lliststack import Stack
defisValidSource( srcfile ):
s = Stack()
 for line in srcfile :
```

```
for token in line :
if token in "{[(" :
s.push( token )
elif token in "}])" :
if s.isEmpty() :
return False
else :
left = s.pop()
if (token == "}" and left != "{") or \
(token == "]" and left != "[") or \
(token == ")" and left != "(") :
return False
return s.isEmpty()
```

8.4.2 Evaluating Postfix Expressions

- Given the expression

  A * B + C / D

- A * B will be performed first, followed by the division and concluding with addition.
- When evaluating this expression stored as a string and scanning one character at a time from left to right,
- Consider the order of the precedence for the operators.
- Evaluating a string containing nine non-blank characters and have scanned the first three:

  A + B

- Moving to the the next character

  A + B /

- Scan more of the string to determine which operation is the first to be performed.

  A + B / (C * D)

- After determining the first operation to be performed, we must then decide how to return to those previously skipped. This can become a tedious process if we have to continuously scan forward and backward through the string in order to properly evaluate the expression.

- To simplify the estimate of a mathematical expression, we need an alternative representation for the expression.
- A representation in which the order the operators are performed is the order they are specified would allow for a single left-to-right scan of the expression string.

Converting from Infix to Postfix

- Infix expressions can be easily converted to postfix notation.
- The expression A + B - C would be written as AB+C- in postx form.
- The evaluation of this expression would involve first adding A and B and then subtracting C from that result.
- Consider the expression A*(B+C), which would be written in postx as ABC+*.
- To help in this conversion we can use a simple algorithm:

1. Place parentheses around every group of operators in the correct order of evaluation. There should be one set of parentheses for every operator in the infix expression.

((A * B) + (C / D))

2. For each set of parentheses, move the operator from the middle to the end preceding the corresponding closing parenthesis.

((A B *) (C D /) +)

3. Remove all of the parentheses, resulting in the equivalent postfix expression.

A B * C D / +

4. Compare this result to a modified version of the expression in which parentheses are used to place the addition as the rst operation:

A * (B + C) / D

5. Using the simple algorithm, we parenthesize the expression: ((A * (B + C)) / D) and move the operators to the end of each parentheses pair: ((A (B C +) *) D /)

6. Finally, removing the parentheses yields the postfix expression:

A B C + * D /

A same algorithm can be used for converting from infix to prefix notation.

Postfix Evaluation Algorithm

- Evaluating a postfix expression requires the use of a stack to store the operands or variables at the beginning of the expression until they are needed.

- Assume we are given a valid postfix expression stored in a string consisting of operators and single-letter variables. We can evaluate the expression by scanning the string, one character or token at a time. For each token, we perform the following steps:

1. If the current item is an operand, push its value onto the stack.
2. If the current item is an operator:

(a) Pop the top two operands of the stack.

(b) Perform the operation.

(Note the top value is the right operand while the next to the top value is the left operand.)

(c) Push the result of this operation back onto the stack.

- To illustrate the use of this algorithm, let's evaluate the postfix expression A B C + * D / from our earlier example. Assume the existence of an empty stack and the following variable assignments have been made:

A = 8 C = 3

B = 2 D = 4

- The complete sequence of algorithm steps and the contents of the stack after Each operation are shown in Table 3

Token	Alg. Step	Stack	Description
ABC+*D/	1	8	push value of A
ABC+*D/	1	8 2	push value of B
ABC+*D/	1	8 2 3	push value of C
ABC+*D/	2(a)	8	pop top two values: y = 3, x = 2
	2(b)	8	compute z = x + y or z = 2 + 3
	2(c)	8 5	push result (5) of the addition
ABC+*D/	2(a)		pop top two values: y = 5, x = 8
	2(b)		compute z = x * y or z = 8 * 5
	2(c)		push result (40) of the multiplication
ABC+*D/	1	40 4	push value of D
ABC+*D/	2(a)		pop top two values: y = 4, x = 40
	2(b)		compute z = x / y or z = 40 / 4
	2(c)	10	push result (10) of division

Table 3 the stack contents and sequence of algorithm steps required to evaluate the valid postfix expression A B C + * D.

- The following invalid expression in which there are more operands than available operators:

  A B * C D +

- if there are too many operators for the given number of operands. Consider such an invalid expression:

  A B * + C /

- In this case, there are too few operands on the stack when we encounter the addition operator, as shown in Table 4

Token	Alg. Step	Stack	Description
AB*CD+	1	8	push value of A
AB*CD+	1	8 2	push value of B
AB*CD+	2(a)		pop top two values: y = 2, x = 8
	2(b)		compute z = x * y or z = 8 * 2
	2(c)	16	push result (16) of the multiplication
AB*CD+	1	16 3	push value of C
AB*CD+	1	16 3 4	push value of D
AB*CD+	2(a)	16	pop top two values: y = 4, x = 3
	2(b)	16	compute z = x + y or z = 3 + 4
	2(c)	16 7	push result (7) of the addition
Error	*xxxxx*	*xxxxx*	*Too many values left on stack.*

Table 4 the sequence of algorithm steps when evaluating the invalid postfix expression

A B * C D +.

8.4 SUMMARY

1. A stack is used to store data such that the last item inserted is the rst item removed. It is used to implement a last-in rst-out (LIFO) type protocol.

2. A stack is a data structure that stores a linear collection of items with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the top of the stack. An empty stack is one containing no items.

3. The two most common approaches in Python include the use of a Python list and a linked list.

4. The function isValid Source() accepts a object, which we assume was pre- viously opened and contains C++ source code.

5. Evaluating a postfix expression requires the use of a stack to store the operands or variables at the beginning of the expression until they are needed.

8.5 REFERENCE FOR FURTHER READING

1. Data Structure and algorithm Using Python, Rance D. Necaise, 2016 Wiley India Edition
2. Data Structure and Algorithms in Python, Michael T. Goodrich, RobertomTamassia, M. H. Goldwasser, 2016 Wiley India Edition

8.6 UNIT END EXERCISES

1. What is a Stack? Explain Stack with different operations.
2. Explain the implementation of Stack using Linked List.
3. Write a short note on”

a. Balanced Delimiters.

b. Evaluating Postfix Expression.

9

LINKED LIST

Unit Structure

9.0 OBJECTIVES

This chapter will make the readers understand the following concepts:

- Doubly Linked List
- Operations on doubly linked list
- Circular Linked List
- Operations on circular linked list
- Multilinked list
- Examples

9.1 ADVANCED LINKED LISTS

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown below

Figure 1 - Linked List

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

9.1.1 Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.

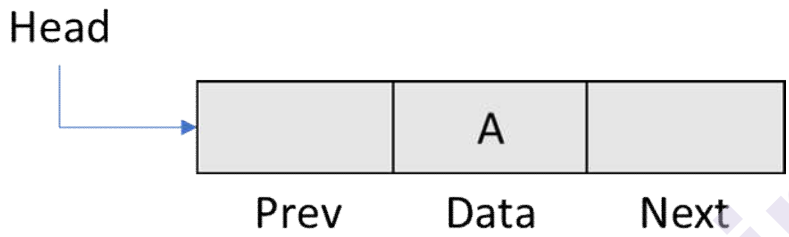


Figure 2- Doubly Linked List - Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

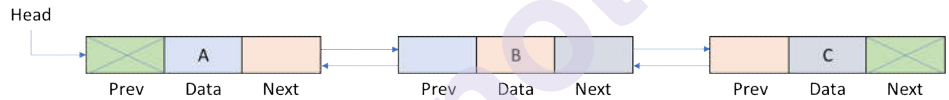


Figure 3 - Doubly Linked List

In C, structure of a node in doubly linked list can be given as:

```
1.  struct node
2.  {
3.  struct node *prev;
4.  int data;
5.  struct node *next;
6.  }
```

The **prev**part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous

node as well by using the previous address stored inside the previous part of each node.

9.1.1.1 Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.

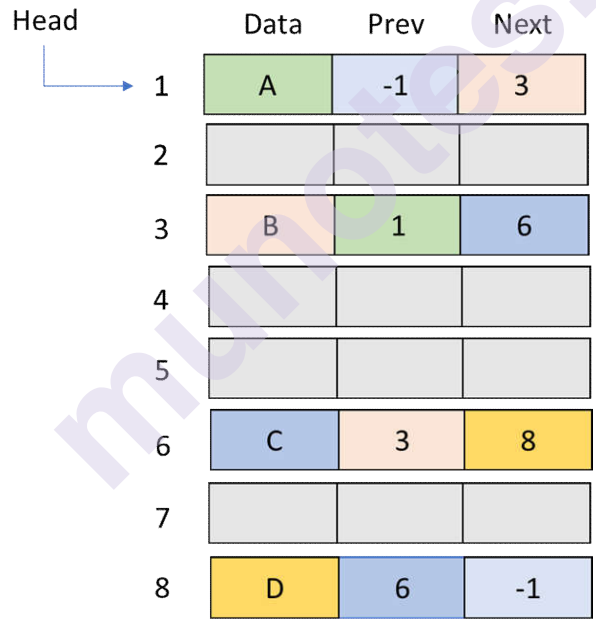


Figure 4 - Memory Representation of a Doubly Linked List

9.1.1.2 Operations on doubly linked list

Node Creation

```
1. struct node
2. {
3. struct node *prev;
4. int data;
5. struct node *next;
```

```
6.  };
7.  struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

SNo	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

9.1.2 Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.

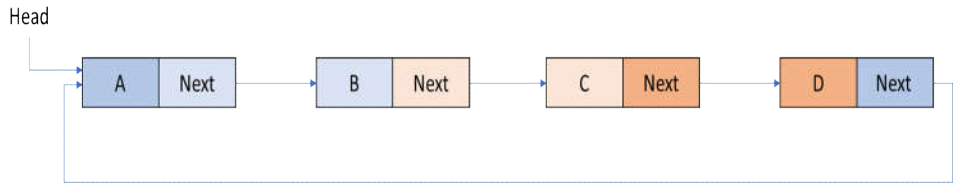


Figure 5 - Circular Singly Linked List

9.1.2.1 Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

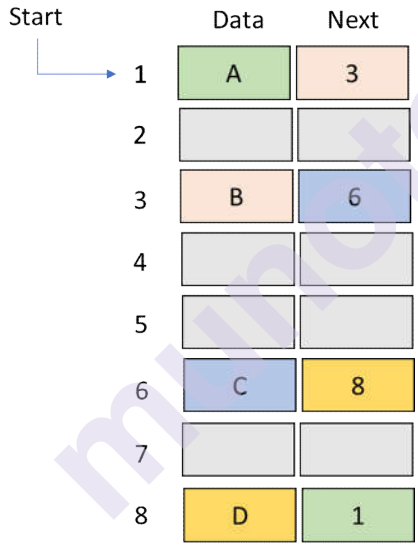


Figure 6 - Memory Representation of a Circular Linked List

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

9.1.2.2 Operations on Circular Singly linked list

Insertion

SNo	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SNo	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

9.1.3 Multi-Linked Lists

A multilinked list is a more general linked list with multiple links from nodes.

- In a general multi-linked list, each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.
- Multi-lists are essentially the technique of embedding multiple lists into a single data structure.
- A multi-list has more than one next pointer, like a doubly linked list, but the pointers create separate lists Linked Structures

  A doubly-linked list or multi-list is a data structure with multiple pointers in each node.
- In a doubly-linked list the two pointers create bi-directional links

- In a multi-list the pointers used to make multiple link routes through the data

  Doubly-linked lists are a special case of multi-linked lists; it is special in two ways:

- Each node has just 2 pointers
- The pointers are exact inverses of each other

In a general multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.

9.1.4.1 Example 1: Multiple Orders Of One Set Of Elements

The standard use of multi-linked lists is to organize a collection of elements in two different ways. For example, suppose my elements include the name of a person and his/her age. e.g.

(FRED,19) (MARY,16) (JACK,21) (JILL,18)

I might want to order these elements alphabetically and also order them by age. I would have two pointers - *NEXT-alphabetically*, *NEXT-age* - and the list header would have two pointers, one based on name, the other on age.

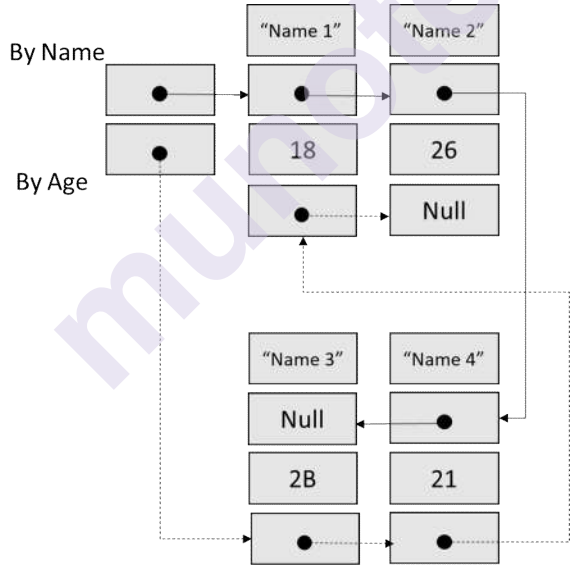


Figure 7 - Multiple Orders of One Set of Elements

Inserting into this structure is very much like inserting the same node into two separate lists. In multi-linked lists it is quite common to have back-pointers, i.e. inverses of each of the forward links; in our example this would mean that each node had four pointers.

9.1.3.2 Example 2: Sparse Matrices

A second very common use of multi-linked lists is sparse matrices. A sparse matrix is a matrix of numbers, as in mathematics, in which almost

all the entries are zero. These arise frequently in engineering applications. the use of a normal Pascal array to store a sparse matrix is extremely wasteful of space - in an NxN sparse matrix typically only about N elements are non-zero. For example:

X = 1 2 3

Y=1 | 0 88 0

Y=2 | 0 0 0

Y=3 | 27 0 0

Y=4 | 19 0 66

We can represent this by having linked lists for each row and each column. Because each node is in exactly one row and one column it will appear in exactly two lists - one row list and one column. So, it needs two pointers: *Next-in-this-row* and *Next-in-this-column*. In addition to storing the data in each node, it is normal to store the co-ordinates (i.e. the row and column the data is in in the matrix). Operations that set a value to zero cause a node to be deleted, and vice versa. As with any linked list in practice is common for every pointer to have a corresponding back pointer.

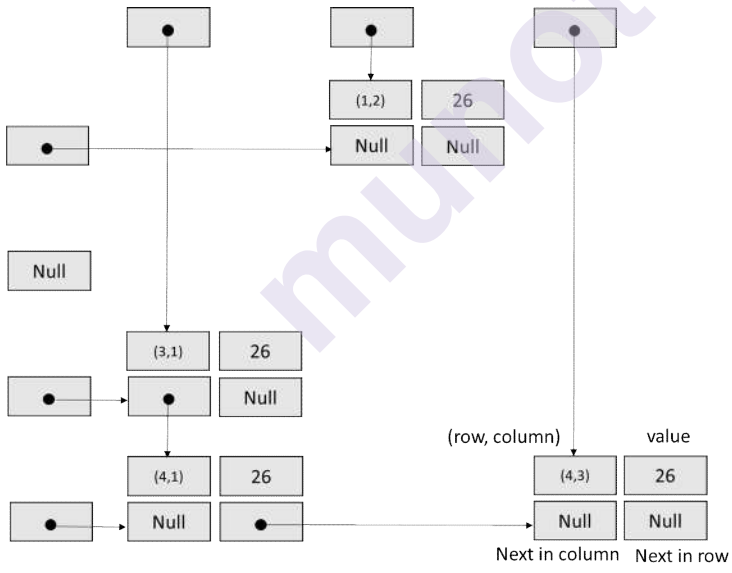


Figure 8 - Sparse Matrices

9.2 POINTS TO REMEMBER

- Priority Queue is an extension of queue_with following properties.
- We traverse a circular singly linked list until we reach the same node where we started.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- A doubly-linked list or multi-list is a data structure with multiple pointers in each node.
- In a doubly-linked list the two pointers create bi-directional links
- In a multi-list the pointers used to make multiple link routes through the data

9.3 REFERENCES

- Data structures and Algorithms Narasimha karum.
- Data structures and Algorithms using C ,C++ learnbay.com
- Greeksforgreeks.com
- Data Structures by Schaum Series
- Introduction to Algorithms by Thomas H Cormen
- Introduction to Algorithm: A Creative Approach

10

QUEUES

Unit Structure

10.0 OBJECTIVES

This chapter will make the readers understand the following concepts:

- Introduction to Queue
- Basic operations on queues

- Queues using Python List
- Concept of circular Queues
- Concept of Linked Queues
- Priority Queues
- Doubly Linked List
- Circular Linked List
- Multilinked list

10.1 THE QUEUE ABSTRACT DATA TYPE INTRODUCTION

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

Queue is referred to be as First In First Out list.For example, people waiting in line for a rail ticket form a queue.

10.1.1 Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Figure 1- Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.

10.2 BASIC OPERATIONS

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Some of the basic operations associated with a queue are:

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

  Few more functions are required to make the above-mentioned queue operation efficient. These are:

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

10.2.1 Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue:

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

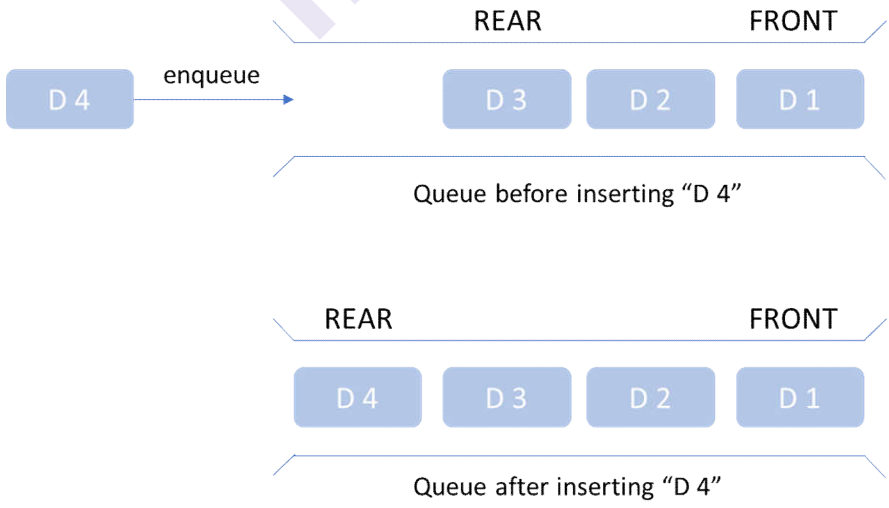


Figure 2 - Enqueue Operation

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
if queue is full
return overflow
  endif
  rear ← rear +1
  queue[rear]← data
returntrue
end procedure
```

Implementation of enqueue() in C programming language −

Example

```
intenqueue(int data)
if(isfull())
return0;
  rear = rear +1;
  queue[rear]= data;
return1;
end procedure
```


10.2.2 Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

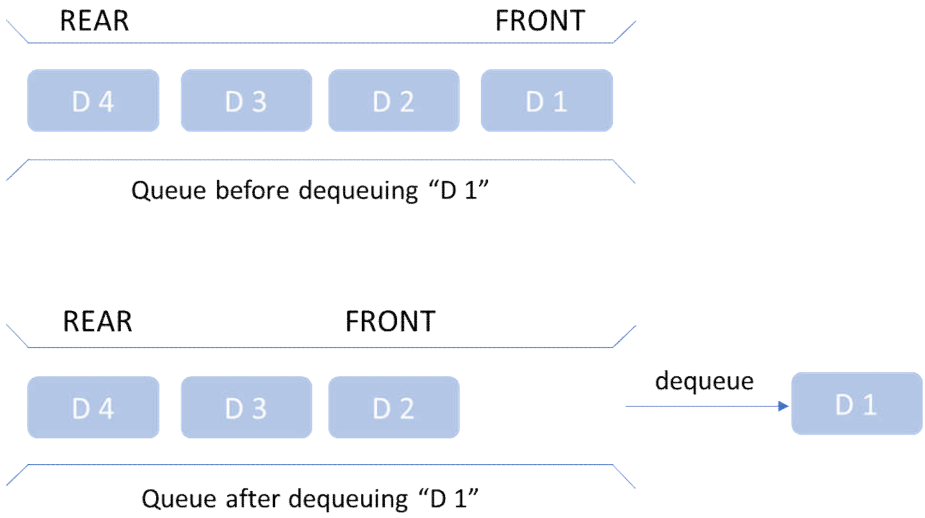


Figure 3 - Dequeue Operation

Algorithm for dequeue operation

```
procedure dequeue
if queue is empty
return underflow
endif
   data = queue[front]
   front ← front +1
returntrue
end procedure
```

Implementation of dequeue() in C programming language −

Example

```
intdequeue(){
if(isempty())
return0;
int data = queue[front];
   front = front +1;
return data;
}
```

10.3 IMPLEMENTING QUEUE-USING PYTHON LIST

There are various ways to implement a queue in Python. This can be done in the following ways:

- list
- collections.deque
- queue.Queue

10.3.1 Implementation using list

List is a Python's built-in data structure that can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used. However, lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

Python program to demonstrate queue implementation using list

```
# Initializing a queue

queue = []

# Adding elements to the queue

queue.append('a')

queue.append('b')

queue.append('c')

print("Initial queue")

print(queue)

# Removing elements from the queue

print("\nElements dequeued from queue")

print(queue.pop(0))

print(queue.pop(0))

print(queue.pop(0))

print("\nQueue after removing elements")

print(queue)

# Uncommenting print(queue.pop(0))

# will raise and IndexError

# as the queue is now empty
```

Output:

Initial queue

['a', 'b', 'c']

Elements dequeued from queue

a

b

c

Queue after removing elements

[]

10.4 CIRCULAR QUEUE

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.

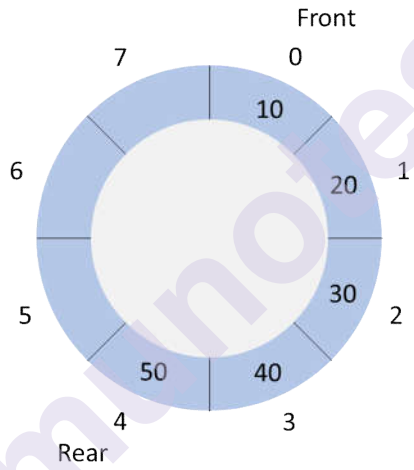


Figure 4 - Circular Queue

In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.

10.4.1 Operations on Circular Queue**:**

- Front: Get the front item from queue.
- Rear: Get the last item from queue.
- enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).

2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 &&front != 0) if it is true then set rear=0 and insert element.

- deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

1. Check whether queue is Empty means check (front==-1).
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

// C or C++ program for insertion and// deletion in Circular Queue

```
#include<bits/stdc++.h>
using namespace std;
class Queue
{
 // Initialize front and rear
   int rear, front;
   // Circular Queue
   int size;
   int *arr;
   Queue(int s)
   { front = rear = -1;
     size = s;
      arr = new int[s];
   }
   void enQueue(int value);
   int deQueue();
   void displayQueue();}
 /* Function to create Circular queue */
void Queue::enQueue(int value)
{if ((front == 0 && rear == size-1) ||
          (rear == (front-1)%(size-1)))
   { printf("\nQueue is Full");
```

```
        return;
    }
    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
        arr[rear] = value;
    }
    else if (rear == size-1 &&front != 0)
    {   rear = 0;
        arr[rear] = value;
    }
    else
    {   rear++;
        arr[rear] = value;
    }
// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front == -1)
    {printf("\nQueue is Empty");
        return INT_MIN;
    }
    int data = arr[front];
    arr[front] = -1;
    if (front == rear)
    {   front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
    else
        front++;
    return data;
}
```

```
// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
   if (front == -1)
   {
      printf("\nQueue is Empty");
      return;
   }
   printf("\nElements in Circular Queue are: ");
   if (rear >= front)
   {
      for (int i = front; i<= rear; i++)
         printf("%d ",arr[i]);
   } else
   { for (int i = front; i< size; i++)
         printf("%d ", arr[i]);
      for (int i = 0; i<= rear; i++)
         printf("%d ", arr[i]);
   }
}
/* Driver of the program */
int main()
{
   Queue q(5);
   // Inserting elements in Circular Queue
   q.enQueue(14);
   q.enQueue(22);
   q.enQueue(13);
   q.enQueue(-6);
   // Display elements present in Circular Queue
   q.displayQueue();
   // Deleting elements from Circular Queue
   printf("\nDeleted value = %d", q.deQueue());
```

```
    printf("\nDeleted value = %d", q.deQueue());
  q.displayQueue();
    q.enQueue(9);
    q.enQueue(20);
    q.enQueue(5);
  q.displayQueue();
    q.enQueue(20);
    return 0;
}
```

Output:
Elements in Circular Queue are: 14 22 13 -6

Deleted value = 14

Deleted value = 22

Elements in Circular Queue are: 13 -6

Elements in Circular Queue are: 13 -6 9 20 5

Queue is Full

Time Complexity: Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

10.4.2 Applications:

- Memory Management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- Traffic system: In computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
- CPU Scheduling: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

10.5 LINKED QUEUE

The array implementation cannot be used for the large-scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

In a linked queue, each node of the queue consists of two parts i.e., data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.

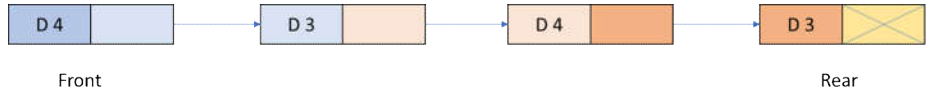


Figure 5 - Linked Queue

10.5.1 Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

- Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition front = NULL becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
    if(front == NULL)
    {
       front = ptr;
       rear = ptr;
       front -> next = NULL;
       rear -> next = NULL;
    }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node ptr. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
        rear = ptr;
        rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- Step 1: Allocate the space for the new node PTR
- Step 2: SET PTR -> DATA = VAL
- Step3: IFFRONT=NULL

SETFRONT=REAR=PTR
SETFRONT->NEXT=REAR->NEXT=NULL
ELSE
SETREAR->NEXT=PTR
SETREAR=PTR
SETREAR->NEXT=NULL
[END OF IF]

- Step 4: END

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
    front = front -> next;
    free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

- Step1: IFFRONT=NULL

Write“Underflow"
GotoStep5
[END OF IF]

- Step 2: SET PTR = FRONT
- Step 3: SET FRONT = FRONT -> NEXT
- Step 4: FREE PTR
- Step 5: END

10.6 PRIORITY QUEUE — ABSTRACT DATA TYPE

Priority Queue is an *Abstract Data Type (ADT)* that holds a collection of elements, it is similar to a normal Queue, the difference is that the elements will be dequeued following a priority order.

Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

A real-life example of a priority queue would be a hospital queue where the patient with the most critical situation would be the first in the queue. In this case, the priority order is the situation of each patient.

Atypical priority queue supports following operations.

- **insert (item, priority):** Inserts an item with given priority.
- **getHighestPriority ():** Returns the highest priority item.
- **getHighestPriority ():** Removes the highest priority item.

10.6.1 ADT — Interface

The Priority Queue interface can be implemented in different ways, is important to have operations to add an element to the queue and to remove an element from the queue.

Main operations

- enqueue(value, priority) -> Enqueue an element
- dequeue() -> Dequeue an element

- peek() -> Check the element on the top
- isEmpty() -> Check if the queue is empty

10.6.2 Implementation of priority queue

- *Using Array:* A simple implementation is to use array of following structure.

  - struct item {
  - int item;
  - int priority;
  - }

- insert() operation can be implemented by adding an item at end of array in O(1) time.
- get Highest Priority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
- delete Highest Priority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is delete Highest Priority() can be more efficient as we don't have to move items.

10.6.3 Bounded Priority Queue

Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation

A Bounded Priority Queue implements a priority queue with an upper bound on the number of elements. If the queue is not full, added elements are always added. If the queue is full and the added element is greater than the smallest element in the queue, the smallest element is removed and the new element is added. If the queue is full and the added element is not greater than the smallest element in the queue, the new element is not added.

Bounded priority queues are the ideal data structure with which to implement n-best accumulators. A priority queue of bound n can find the n-best elements of a collection of m elements using O(n) space and O(m n log n) time.

Bounded priority queues may also be used as the basis of a search implementation with the bound implementing heuristic n-best pruning.

Because bounded priority queues require a comparator and a maximum size constraint, they do not comply with the recommendation in the Collection interface in that they neither implement a nullary constructor nor a constructor taking a single collection. Instead, they are constructed with a comparator and a maximum size bound.

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in Class Cast Exception).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the optional methodsof the Collection and Iterator interfaces.

The Iterator provided in method iterator() is not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

10.6.4 Unbounded Priority Queues

Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue. For example, LinkedList

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at

least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator () is not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays. sort (pq.to Array()).

Note that this implementation is not synchronized. Multiple threads should not access a Priority Queue instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe Priority Blocking Queue class.

Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

10.6.5 Applications of Priority Queue:

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. All queue applications where priority is involved

10.7 POINTS TO REMEMBER

- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
- A queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.
- Some of the basic operations associated with a queue are:

  enqueue() − add (store) an item to the queue.
  dequeue() − remove (access) an item from the queue.
- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- Priority Queue is an extension of queue with following properties.
- We traverse a circular singly linked list until we reach the same node where we started.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

10.8 REFERENCES

- Data structures and Algorithms Narasimha karum.
- Data structures and Algorithms using C ,C++ learnbay.com
- Greeksforgreeks.com
- Data Structures by Schaum Series
- Introduction to Algorithms by Thomas H Cormen
- Introduction to Algorithm: A Creative Approach

10.9 UNIT END EXERCISES

1. What is Queue ? Explain Bounded queue with different operations.
2. What is the meaning of ADT?
3. Write a short note on"
   a. linked Queue
   b. Circular Queue

Unit III

11

RECURSION

Unit Structure:

11.0 OBJECTIVE

- To study of Recursion.
- To know importance of Recursion.
- To study types of recursion.

11.1 INTRODUCTION

In simple words, recursion is a problem solving, and in some cases, a programming technique that has a very special and exclusive property. In recursion, a function or method has the ability to call itself to solve the problem. The process of recursion involves solving a problem by turning it into smaller varieties of itself.

The process in which a function calls itself could happen directly as well as indirectly. This difference in call gives rise to different types of recursion, which we will talk about a little later.

The concept of recursion is established on the idea that a problem can be solved much easily and in lesser time if it is represented in one or smaller versions. Adding base conditions to stop recursion is another important part of using this algorithm to solve a problem.

11.2 TYPES OF RECURSION

There are only two types of recursion as has already been mentioned. Let us see how they are different from one another. Direct recursion is the simpler way as it only involves a single step of calling the original function or method or subroutine. On the other hand, indirect recursion involves several steps.

The first call is made by the original method to a second method, which in turn calls the original method. This chain of calls can feature a number of methods or functions. In simple words, we can say that there is always a variation in the depth of indirect recursion, and this variation in depth depends on the number of methods involved in the process.

Direct recursion can be used to call just a single function by itself. On the other hand, indirect recursion can be used to call more than one method or function with the help of other functions, and that too, a number of times. Indirect recursion doesn't make overhead while its direct counterpart does.

There are many ways to categorize a recursive function. Listed below are some of the most common.

1. Linear Recursive

A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution). The factorial function is a good example of linear recursion. Another example of a linear recursive function would be one to compute the square root of a number using Newton's method (assume EPSILON to be a very small number close to 0):

```
double my_sqrt(double x, double a)
{
    double difference = a*x-x;
    if (difference < 0.0) difference = -difference;
    if (difference < EPSILON) return(a);
    else return(my_sqrt(x,(a+x/a)/2.0));
    }
```

2. Tail recursive

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

```
intgcd(int m, int n)
{
int r;
if (m < n) return gcd(n,m);
r = m%n;
if (r == 0) return(n);
else return(gcd(n,r));
}
```

3. Binary Recursive

Some recursive functions don't just have one call to themself, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

The mathematical combinations operation is a good example of a function that can quickly be implemented as a binary recursive function. The number of combinations, often represented as nCk where we are choosing n elements out of a set of k elements, can be implemented as follows:

```
int choose(int n, int k)
{
   if (k == 0 || n == k) return(1);
   else return(choose(n-1,k) + choose(n-1,k-1));
}
```

4. Exponential recursion

An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set (exponential meaning if there were n elements, there would be $O(a^n)$ function calls where a is a positive number).

A good example an exponentially recursive function is a function to compute all the permutations of a data set. Let's write a function to take an array of n integers and print out every permutation of it.

```
void print_array(intarr[], int n)
{
inti;
  for(i=0; i<n; i) printf("%d ", arr[i]);
printf("\n");
}
void print_permutations(intarr[], int n, inti)
{
int j, swap;
print_array(arr, n);
  for(j=i+1; j<n; j)
  {
    swap = arr[i];
arr[i] = arr[j];
arr[j] = swap;
print_permutations(arr, n, i+1);
    swap = arr[i];
arr[i] = arr[j];
arr[j] = swap;
  }
}
```

To run this function on an array arr of length n, we'd do print_permutations (arr, n, 0) where the 0 tells it to start at the beginning of the array.

5. Nested Recursion

In nested recursion, one of the arguments to the recursive function is the recursive function itself. These functions tend to grow extremely fast. A good example is the classic mathematical function, "Ackerman's function. It grows very quickly (even for small values of x and y, Ackermann(x, y) is extremely large) and it cannot be computed with only definite iteration (a completely defined for() loop for example); it requires indefinite iteration (recursion, for example).

Ackerman's function

```
intackerman(int m, int n)
 {
 if (m == 0) return(n+1);
else if (n == 0)
return(ackerman(m-1,1));
else
return(ackerman(m-1,ackerman(m,n-1)));
 }
```

Try computing ackerman (4, 2) by hand... have fun!

6. Mutual Recursion

A recursive function doesn't necessarily need to call itself. Some recursive functions work in pairs or even larger groups. For example, function A calls function B which calls function C which in turn calls function A.

A simple example of mutual recursion is a set of function to determine whether an integer is even or odd. How do we know if a number is even? Well, we know 0 is even. And we also know that if a number n is even, then n - 1 must be odd. How do we know if a number is odd? It's not even!

```
intis_even(unsigned int n)
{
if (n==0) return 1;
else return(is_odd(n-1));
}
intis_odd(unsigned int n)
{
return (!iseven(n));
 }
```

I told you recursion was powerful! Of course, this is just an illustration. The above situation isn't the best example of when we'd want to use recursion instead of iteration or a closed form solution. A more efficient set of function to determine whether an integer is even or odd would be the following:

```
intis_even(unsigned int n)
 {
if (n % 2 == 0) return 1;
else return 0;
}
intis_odd(unsigned int n)
{
if (n % 2 != 0) return 1;
else return 0;
 }
```

Problem: Your boss asks you to write a function to sum up all of the numbers between some high and low value. You decide to write two different versions of the function, one recursive and one iterative. 1) Write them. The next morning you come into work and your boss calls you into his office, unhappy at how slow both of your functions work, compared to how the problem could be solved. 2) How else could you solve this problem?

1a) Iteratively:

```
intsum_nums(int low, int high)
{ inti, total=0;
for(i=low; i<=high; i++)
total+=i;
return total;
}
```

1b) Recursively:

```
intsum_nums(int low, int high)
{
if (low == high) return high;
else return low + sum_nums(low+1, high);
 }
```

2) Certain mathematical functions have closed form expressions; this means that there is actually a mathematical expression that can be used to explicitly evaluate the answer, thereby solving the problem in constant time, as opposed to the linear time it takes for the recursive and iterative versions.

```
intsum_nums(int low, int high)
{
return (((high*(high+1))/2) - (((low-1)*low)/2);
}
```

Problem: What is wrong with the following function?

```
int factorial(int n)
{ if (n<=1) return 1;
 else if (n<0) return 0;
else return factorial(n-1) * n;
 }
```

The first two if statements should be switched. This function works fine if the function is called on valid input (n > = 0). But if it is called on invalid input (n < 0), the function will incorrectly return 1.

Problem: Your research assistant has come to you with the following two functions:

```
intfactorial_iter(int n)
{ int fact=1;
if (n<0) return 0;
for( ; n>0; n--) fact *= n;
return(fact);
}
and
intfactorial_recur(int n)
{ if (n<0) return 0;
else if (n<=1)
return 1;
else
return n * factorial_recur(n-1);
}
```

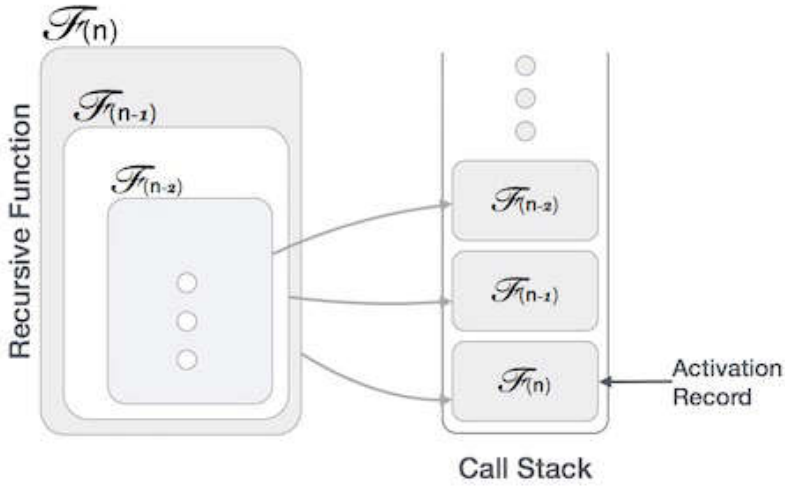
He claims that the factorial_recur() function is more efficient because it has fewer local variables and thus uses less space. What do you tell him?

Every time the recursive function is called, it takes up stack and space for its local variables are set aside. So actually, the recursive version takes up much more space overall than does the iterative version.

11.2.1 Implementation

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

11.2.2 Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

11.2.3 Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

11.2.4 Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

11.3 PROPERTIES OF RECURSIVE FUNCTIONS

All recursive algorithms must implement 3 properties:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself.

The base case is the condition that allows the algorithm to stop the recursion and begin the process of returning to the original calling function. This process is sometimes called unwinding the stack. A base case is typically a problem that is small enough to solve directly. In the accumulate algorithm the base case is an empty vector.

The second property requires modifying something in the recursive function that on subsequent calls moves the state of the program closer to the base case. A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way. In the accumulate algorithm our primary data structure is a vector, so we must focus our state-changing efforts on the vector. Since the base case is the empty vector, a natural progression toward the base case is to shorten the vector.

Lastly, a recursive algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into a smaller and easier problems.

In the remainder of this chapter we will look at more examples of recursion. In each case we will focus on designing a solution to a problem by using the three properties of recursive functions.

11.4 HOW DOES RECURSION WORK?

The concept of recursion is established on the idea that a problem can be solved much easily and in lesser time if it is represented in one or smaller versions. Adding base conditions to stop recursion is another important part of using this algorithm to solve a problem.

People often believe that it is not possible to define an entity in terms of itself. Recursion proves that theory wrong. And if this technique is carried out in the right way, it could yield very powerful results. Let us see how recursion works with a few examples. What is a sentence? It can be defined as two or more sentences joined together with the help of conjunction. Similarly, a folder could be a storage device that is used to store files and folders. An ancestor could be a parent of one and an ancestor of another family member in the family tree.

Recursion helps in defining complex situations using a few very simple words. How would you usually define an ancestor? A parent, a grandparent, or a great grandparent. This could go on. Similarly, defining a folder could be a tough task. It could be anything that holds some files and folders that could be files and folders in their own right, and this could

again go on. This is why recursion makes defining situations a lot easier than usual.

Recursion is also a good enough programming technique. A recursive subroutine is defined as one that directly or indirectly calls itself. Calling a subroutine directly signifies that the definition of the subroutine already has the call statement of calling the subroutine that has been defined.

On the other hand, the indirect calling of a subroutine happens when a subroutine calls another subroutine, which then calls the original subroutine. Recursion can use a few lines of code to describe a very complex task. Let us now turn our attention to the different types of recursion that we have already touched upon.

11.5 WHEN IS RECURSION USED?

There are situations in which you can use recursion or iteration. However, you should always choose a solution that appears to be the more natural fit for a problem. A recursion is always a suitable option when it comes to data abstraction. People often use recursive definitions to define data and related operations.

And it won't be wrong to say that recursion is mostly the natural solution for problems associate with the implementation of different operations on data. However, there are certain things related to recursion that may not make it the best solution for every problem. In these situations, an alternative like the iterative method is the best fit.

The implementation of recursion uses a lot of stack space, which can often result in redundancy. Every time we use recursion, we call a method that results in the creation of a new instance of that method. This new instance carries different parameters and variables, which are stored on the stack, and are taken on the return. So while recursion is the more simple solution than others, it isn't usually the most practical.

Also, we don't have a set of pre-defined rules that can help choose iteration or recursion for different problems. The biggest benefit of using recursion is that it is a concise method. This makes reading and maintaining it easier tasks than usual. But recursive methods aren't the most efficient methods available to us as they take a lot of storage space and consume a lot of time during implementation.

Keeping in mind a few things can help you decide whether choosing a recursion for a problem is the right way to go or not. You should choose recursion if the problem that you are going to solve is mentioned in recursive terms and the recursive solution seems less complex.

You should know that recursion, in most cases, simplifies the implementation of the algorithms that you want to use. Now if the complexities associated with using iteration and recursion are the same for a given problem, you should go with iteration as the chances of it being more efficient are higher.

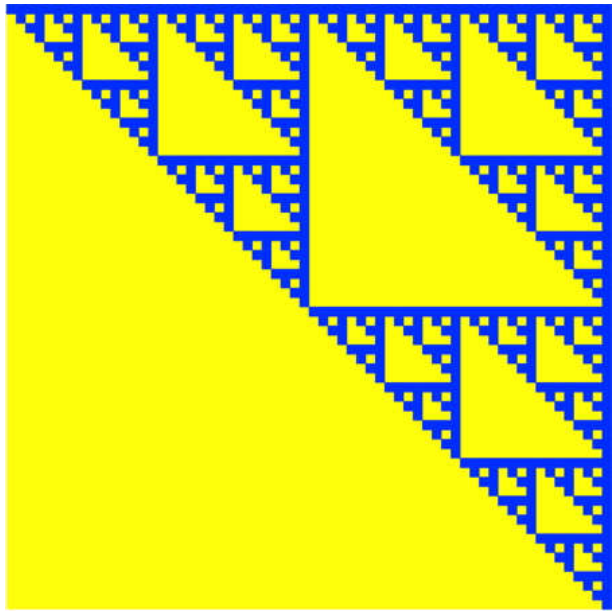
11.6 **PRACTICAL APPLICATIONS:**

The practical applications of recursion are near endless. Many math functions cannot be expressed without its use. The more famous ones are the Fibonacci sequence and the Ackermann function. It's through math functions that many software applications are built. Take for example Candy Crush which uses them to generate combinations of tiles.

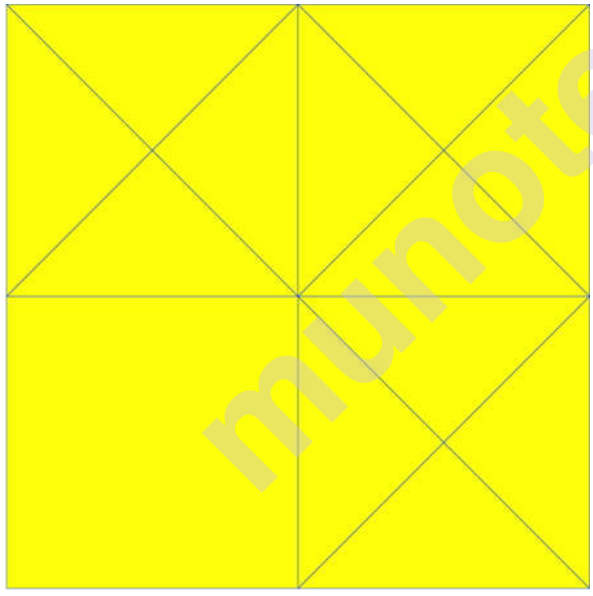
If you're not familiar with Candy Crush (You should be) then chess is another example of recursion in action. Almost all searching algorithms today use a form of recursion as well. In this day and age where information is key, recursion becomes one of the most important methods in programming.

Multiple recursion with the Sierpinski gasket

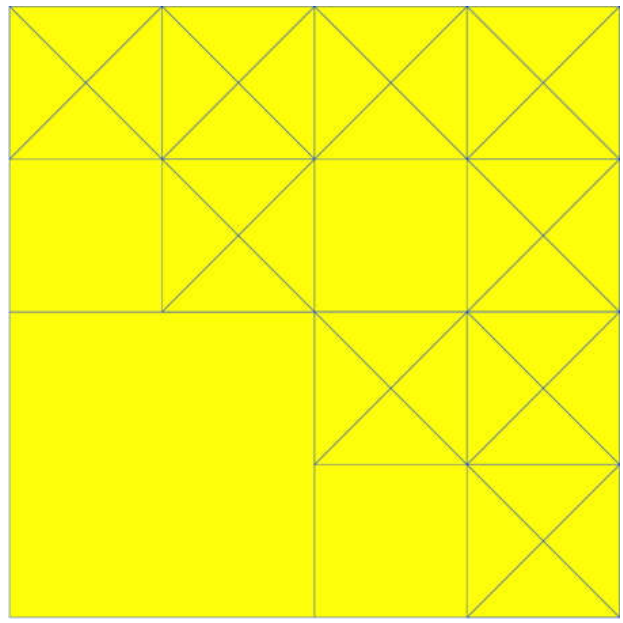
So far, the examples of recursion that we've seen require you to make one recursive call each time. But sometimes you need to make multiple recursive calls. Here's a good example, a mathematical construct that is a fractal known as a **Sierpinski gasket**:



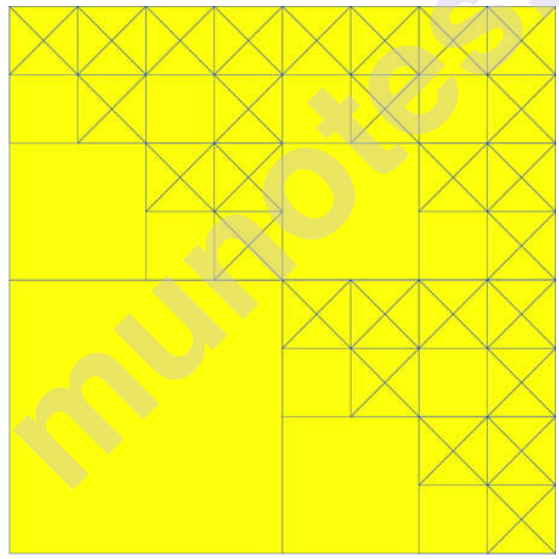
As you can see, it's a collection of little squares drawn in a particular pattern within a square region. Here's how to draw it. Start with the full square region, and divide it into four sections like so:

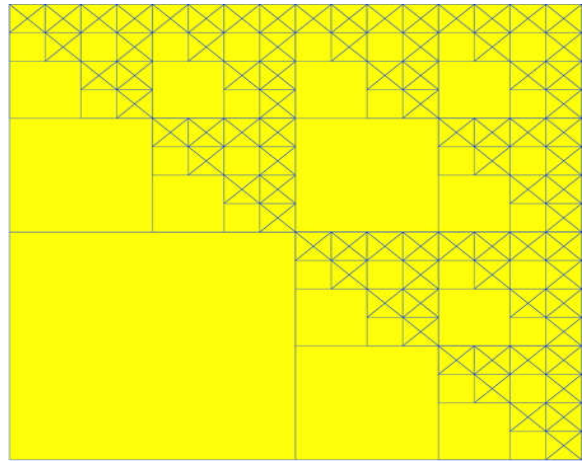


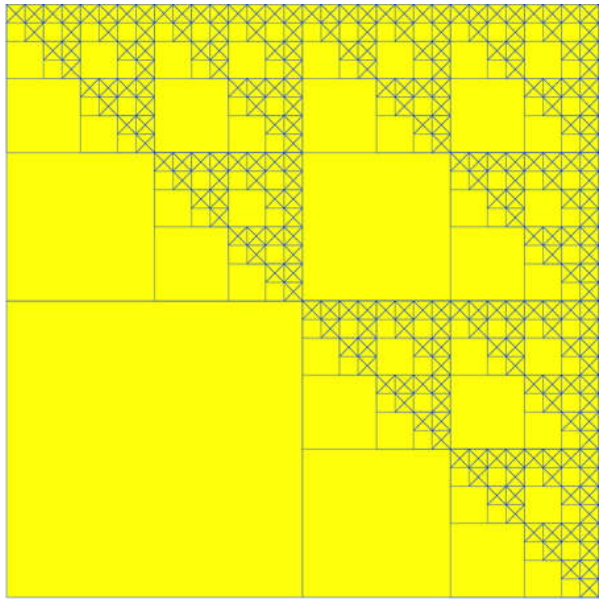
Take the three squares with an × through them—the top left, top right, and bottom right—and divide them into four sections in the same way:

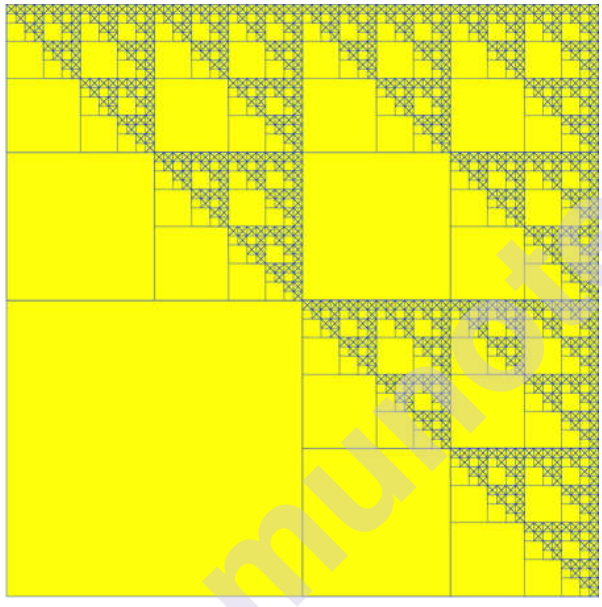


Keep going. Divide every square with an × into four sections, and place an × in the top left, top right, and bottom right squares, but never the bottom left.

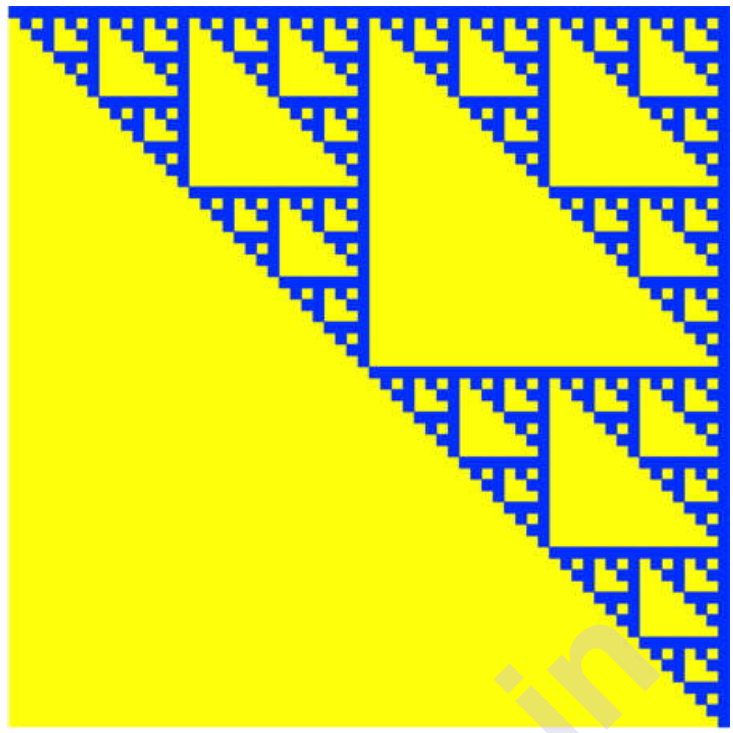








Once the squares get small enough, stop dividing. If you fill in each square with an × and forget about all the other squares, you get the Sierpinski gasket. Here it is once again:



To summarize, here is how to draw a Sierpinski gasket in a square:

- Determine how small the square is. If it's small enough to be a base case, then just fill in the square. You get to pick how small "small enough" is.
- Otherwise, divide the square into upper left, upper right, lower right, and lower left squares. Recursively "solve" three subproblems:
- Draw a Sierpinski gasket in the upper left square.
- Draw a Sierpinski gasket in the upper right square.
- Draw a Sierpinski gasket in the lower right square.

Notice that you need to make not just one but **three** recursive calls. That is why we consider drawing a Sierpinski gasket to exhibit multiple recursion.

You can choose any three of the four squares in which you recursively draw Sierpinski gaskets. The result will just come out rotated by some multiple of 90 degrees from the drawing above. (If you recursively draw Sierpinski gaskets in any other number of the squares, you don't get an interesting result.)

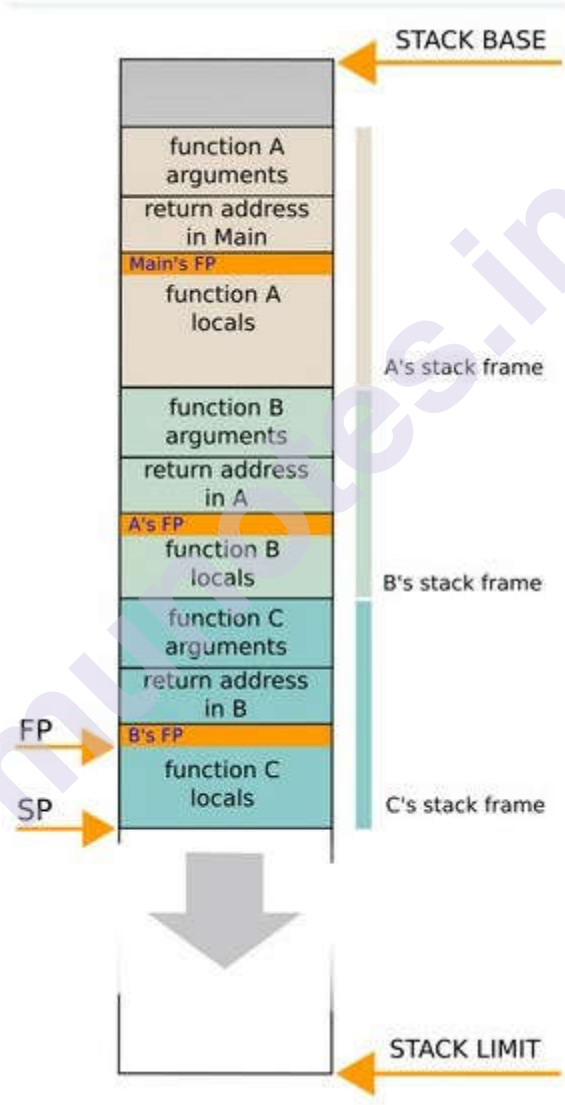
The program below draws a Sierpinski gasket. Try commenting and uncommenting some of the recursive calls to get rotated gaskets:

```
var dim = 240;
varminSize = 8;

vardrawGasket = function(x, y, dim) {
  if (dim <= minSize) {
 rect(x, y, dim, dim);
  }
  else {
 varnewDim = dim / 2;
 drawGasket(x, y, newDim);
 drawGasket(x + newDim, y, newDim);
    // drawGasket(x, y + newDim, newDim);
 drawGasket(x + newDim, y + newDim, newDim);
  }
};

draw = function() {
  background(255, 255, 255);
  fill(255, 255, 0);
rect(0, 0, dim, dim);
  stroke(0, 0, 255);
  fill(0, 0, 255);
drawGasket(0, 0, dim);
};
```


Example.

1. "Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number
2. Thus in recursion last function called needs to be completed first.
3. Now Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.
4. The High level Programming languages, such as Pascal , C etc. that provides support for recursion use stack for book keeping.

5. In each recursive call, there is need to save the
    1. current values of parameters,
    2. local variables and
    3. the return address (the address where the control has to return from the call).
6. Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.



7. Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.
8. The example of recursion as an application of stack is keeping books inside the drawer and the removing each book recursively.

Let's take a classic example where recursion is the best solution: the Fibonacci sequence. If we want to generate the nth Fibonacci number using recursion, we can do it like this:

```
public static int fibonacciRecursion(int nthNumber) {

        if (nthNumber == 0) { //base case

            return 0;

        } else if (nthNumber == 1) { //base case

            return 1;
        }   //recursive call
     return fibonacciRecursion(nthNumber - 1) + fibonacciRecursion(nthNumber - 2);

    }
```

Much cleaner than when compared to the iterative solution:

```
public int fibonacci(int n) {
        if(n <= 1) {
            return n;
        }
        int number = 1;
        int prevNumber = 1;

        for(int i=2; i<n; i++) {
            int temp = number;
            number+= prevNumber;
            prevNumber = temp;
        }
        return number;
    }
```


Let's take another example. In this case, we have a number of bunnies and each bunny has two big floppy ears. We want to compute the total number of ears across all the bunnies recursively. We could do it like this:

```
public static int bunnies(int numberBunnies, int ears) {
        ears = 2;
        if (numberBunnies <= 0) {
            return 0;
        } else {
            numberBunnies--;
            return bunnies(numberBunnies, ears) + ears;
        }
    }
```

11.7 SUMMARY

- In simple words, recursion is a problem solving, and in some cases, a programming technique that has a very special and exclusive property. In recursion, a function or method has the ability to call itself to solve the problem.
- A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution).
- Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does.
- Some recursive functions don't just have one call to themself, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.
- An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set (exponential meaning if there were n elements, there would be $O(a^n)$ function calls where a is a positive number).
- In nested recursion, one of the arguments to the recursive function is the recursive function itself! These functions tend to grow extremely fast.
- A recursive function doesn't necessarily need to call itself. Some recursive functions work in pairs or even larger groups.

11.8 QUESTIONS

1. Write a short note on Recursion.
2. What are the types of Recursion?
3. Explain Linear Recursive with example.
4. Explain Tail recursive with example.
5. Explain Binary Recursive with example.
6. Explain Exponential recursion with example.
7. Explain Nested Recursion with example.
8. Explain Mutual Recursion with example.
9. Properties of recursive functions
10. How does recursion work?
11. When is recursion used?

11.9 REFERENCE FOR FURTHER READING

- https://www.upgrad.com/blog/recursion-in-data-structure/
- https://www.sparknotes.com/cs/recursion/whatisrecursion/section2/
- https://www.upgrad.com/blog/recursion-in-data-structure/
- https://daveparillo.github.io/cisc187-reader/recursion/properties.html
- https://medium.com/@frankzou4000/recursion-and-its-applications-4dc00ee94130

❖❖❖❖

12

HASH TABLE

Unit Structure:

12.0 OBJECTIVE

- To study of Hashing techniques.
- To study application of Hashing Algorithm.
- To find best Hashing Algorithm for particular situation.

12.1 INTRODUCTION

Hash functions are used in conjunction with hash tables to store and retrieve data items or data records. The hash function translates the key associated with each datum or record into a hash code, which is used to index the hash table. When an item is to be added to the table, the hash code may index an empty slot (also called a bucket), in which case the item is added to the table there. If the hash code indexes a full slot, some kind of collision resolution is required: the new item may be omitted (not added to the table), or replace the old item, or it can be added to the table in some other location by a specified procedure.

12.2 HASHING

In data structures,

- There are several searching techniques like linear search, binary search, search trees etc.

- In these techniques, time taken to search any particular element depends on the total number of elements.

Example-

- **Linear Search** takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- **Binary Search** takes O(logn) time to perform the search in sorted arrays consisting of n elements.
- It takes O(logn) time to perform the search in **Binary Search Tree** consisting of n elements.

Drawback-

The main drawback of these techniques is-

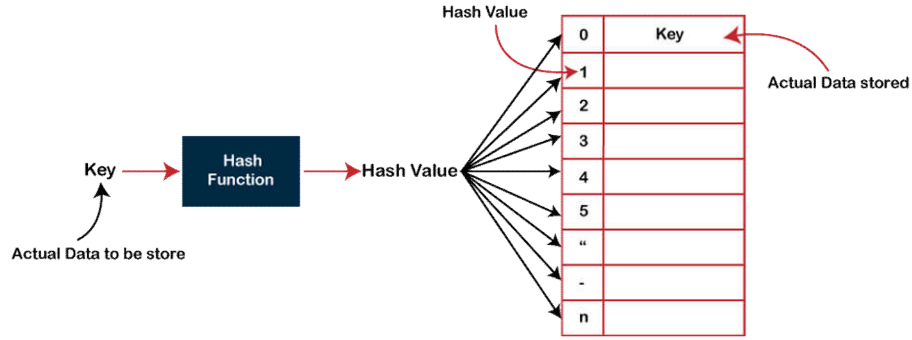
- As the number of elements increases, time taken to perform the search also increases.
- This becomes problematic when total number of elements become too large.

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key)



Advantage-

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

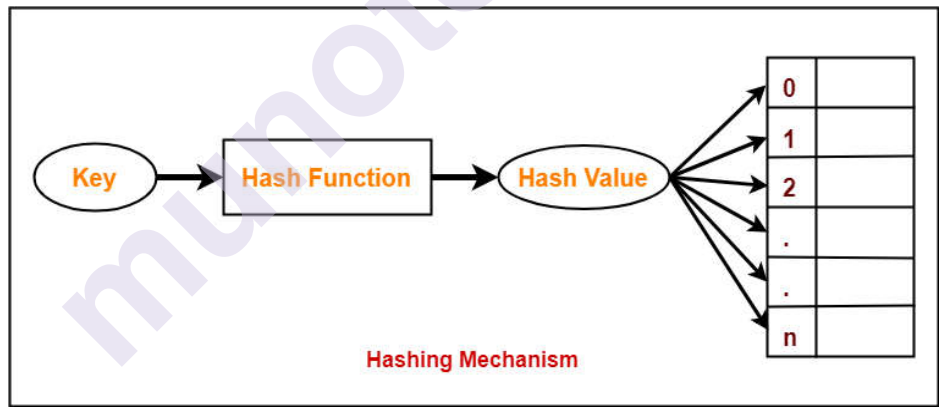
Hashing Mechanism-

In hashing,

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key Value-

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be be stored in the hash table.
- Hash key value is generated using a hash function.



Hashing Mechanism

12.3 HASH FUNCTION

Hash function is a function that maps any big number or string to a small integer value.

- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

13.3.1 Types of Hash Functions-

There are various types of hash functions available such as-

1. Mid Square Hash Function

- A good hash function for numerical values is the mid-square method. The mid-square method squares the key value, and then takes the middle r bits of the result, giving a value in the range 0 to 2^r-1.
- This works well because most or all bits of the key value contribute to the result. For example, consider records whose keys are 4-digit numbers in base 10.
- The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10.
- That is, r = 2. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57.
- All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

2. Division Hash Function

This is the easiest method to create a hash function. The hash function can be described as −

h(k) = k mod n

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder. It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

An example of the Division Method is as follows −

k=1276

n=10

h(1276) = 1276 mod 10

= 6

The hash value obtained is 6

A disadvantage of the division method id that consecutive keys map to consecutive hash values in the hash table. This leads to a poor performance.

3. Folding Hash Function

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10.

12.3.2 Properties of Hash Function-

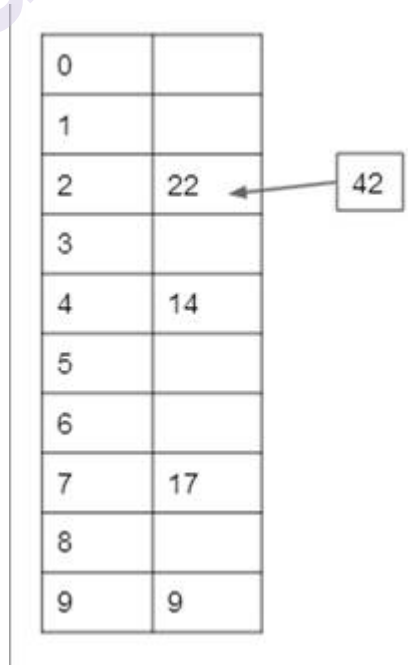
The properties of a good hash function are-

- It is efficiently computable.
- It minimizes the number of collisions.
- It distributes the keys uniformly over the table.

12.4 COLLISION

A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.

Example Hash Table With Collisions:



Let's take the exact same hash function from before: take the value to be hashed mod 10, and place it in that slot in the hash table.

Numbers to hash: 22, 9, 14, 17, 42

As before, the hash table is shown to the right.

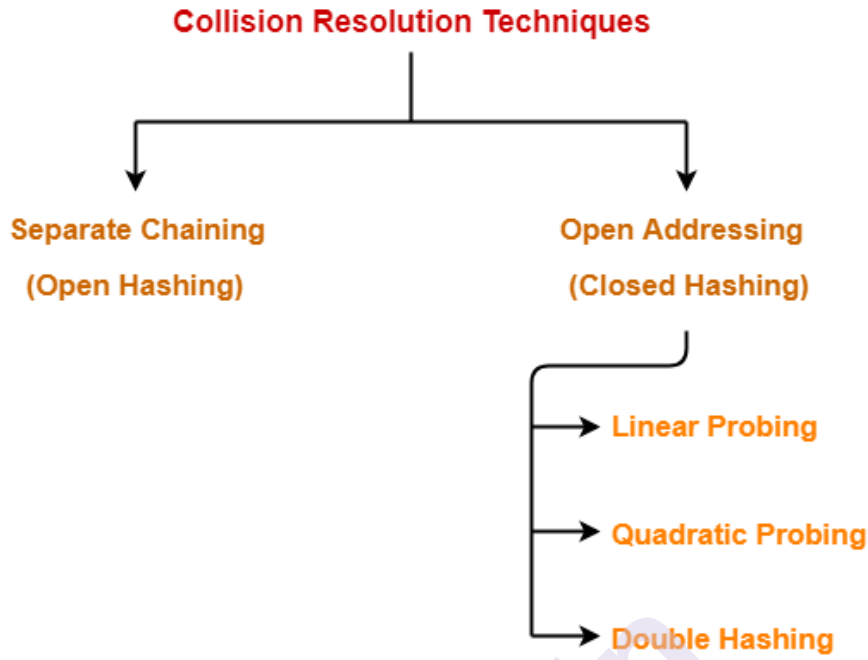
As before, we hash each value as it appears in the string of values to hash, starting with the first value. The first four values can be entered into the hash table without any issues. It is the last value, 42, however, that causes a problem. 42 mod 10 = 2, but there is already a value in slot 2 of the hash table, namely 22. This is a collision.

The value 42 must end up in one of the hash table's slots, but arbitrarly assigning it a slot at random would make accessing data in a hash table much more time consuming, as we obviously want to retain the constant time growth of accessing our hash table. There are two common ways to deal with collisions: chaining, and open addressing.

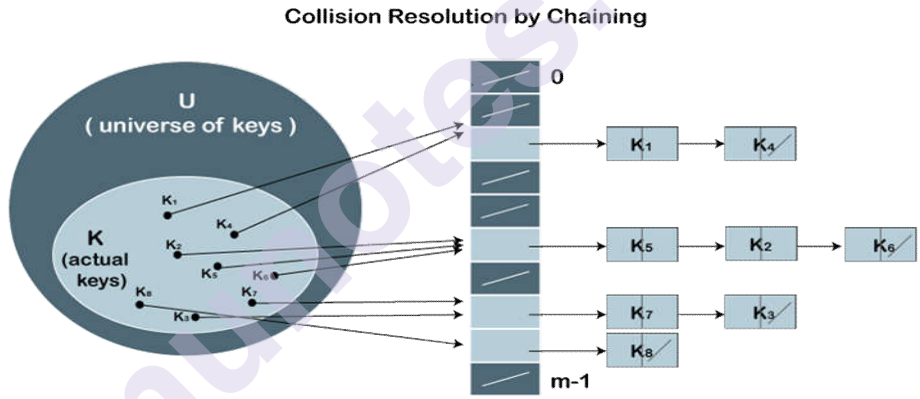
12.5 COLLISION RESOLUTION

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as **open addressing** in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**.

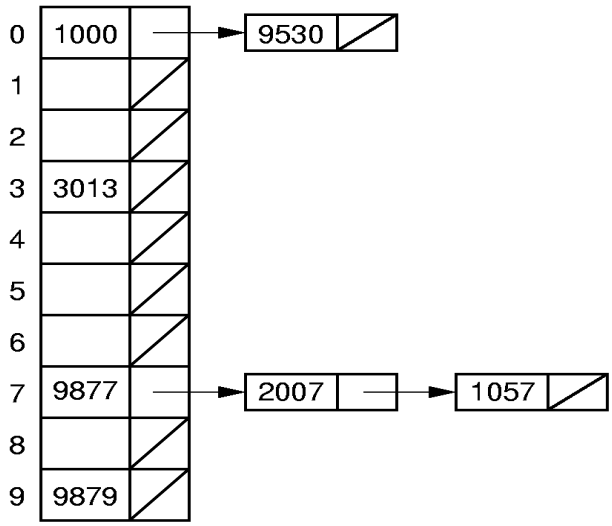


12.5.1 Separate chaining (Open Hashing)



While the goal of a hash function is to minimize collisions, collisions are normally unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. Collision resolution techniques can be broken into two classes: open hashing (also called separate chaining) and closed hashing (also called open addressing). (Yes, it is confusing when ``open hashing" means the opposite of ``open addressing," but unfortunately, that is the way it is.) The difference between the two has to do with whether collisions are stored outside the table (open hashing), or whether collisions result in storing one of the records at another slot in the table (closed hashing).

The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. The figure illustrates a hash table where each slot stores one record and a link pointer to the rest of the list.



Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because we know to stop searching the list once we encounter a key that is greater than the one being searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. Assuming that the table has more slots than there are records to be stored, we can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $\Theta(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

There are similarities between open hashing and Binsort. One way to view open hashing is that each record is simply placed in a bin. While multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation. In a similar fashion, a simple Binsort reduces the number of records in each bin to a small number that can be sorted in some other way.

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

- Insertion Operation
- Deletion Operation
- Searching Operation

NOTE-

- Deletion is easier in separate chaining.
- This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

PRACTICE PROBLEM BASED ON SEPARATE CHAINING-

Problem-

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

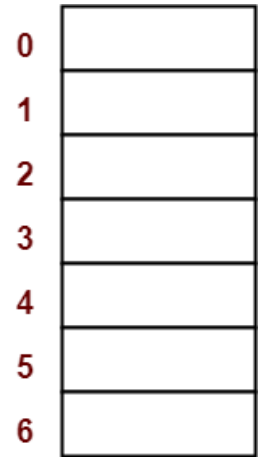
Use separate chaining technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

Step-01:

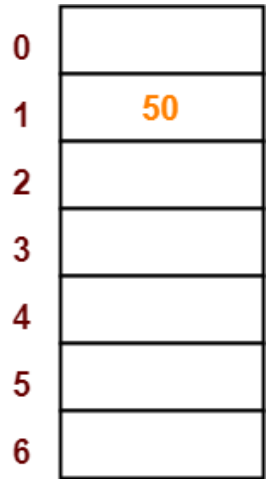
- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



Step-02:

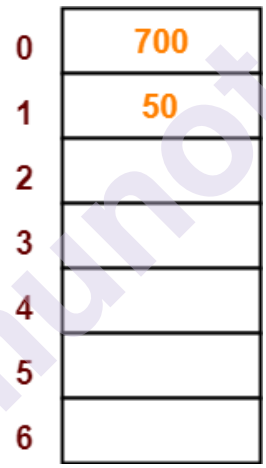
- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.

- So, key 50 will be inserted in bucket-1 of the hash table as-



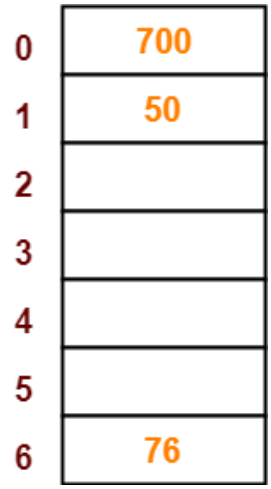
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



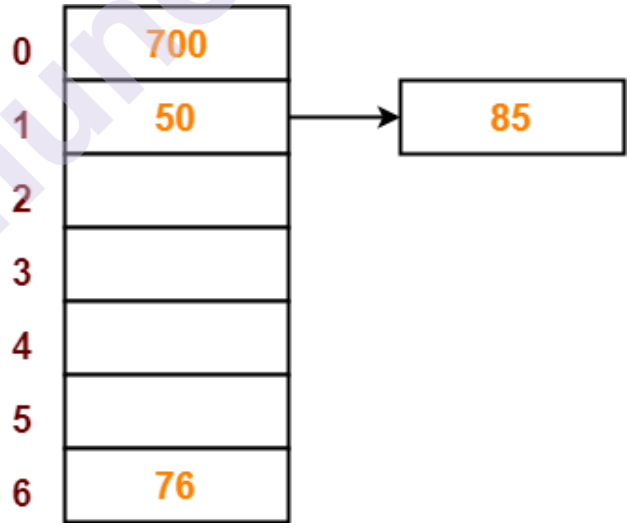
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



Step-05:

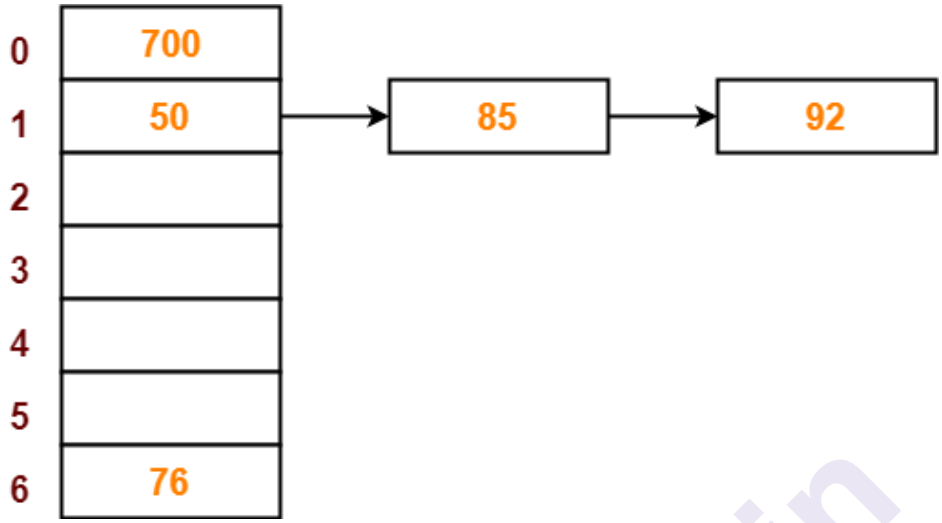
- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



Step-06:

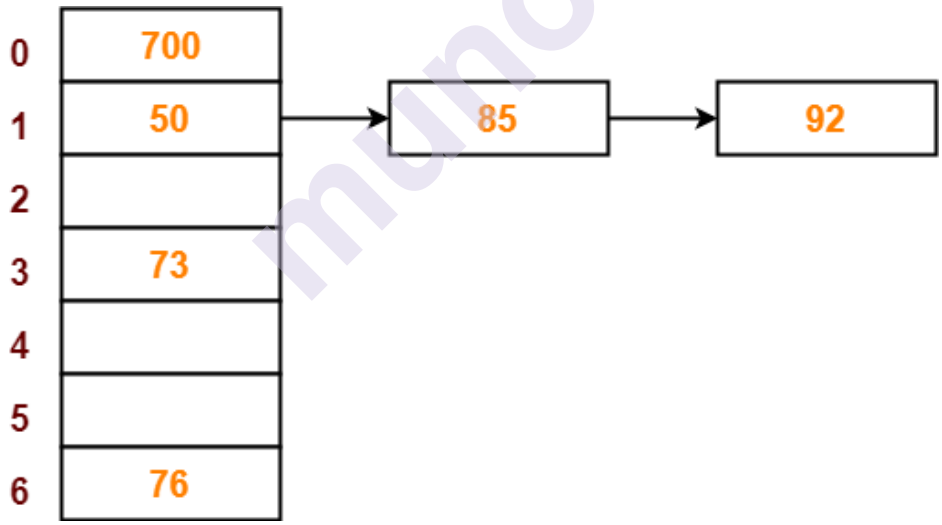
- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.

- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



Step-07:

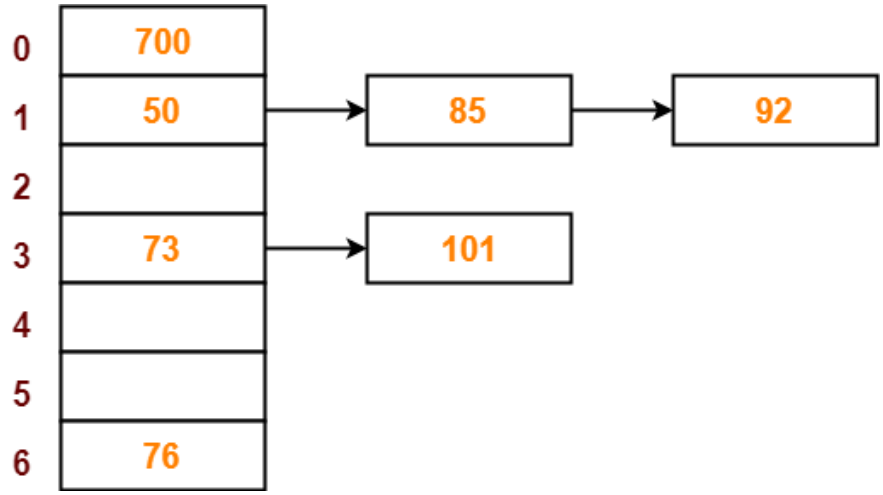
- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.

- So, key 101 will be inserted in bucket-3 of the hash table as-



12.5.2 In open addressing,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Techniques used for open addressing are-

- Linear Probing
- Quadratic Probing
- Double Hashing

Operations in Open Addressing-

Let us discuss how operations are performed in open addressing-

Insert Operation-

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

In case of collision,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

Search Operation-

To search any particular key,

- Its hash value is obtained using the hash function used.

- Using the hash value, that bucket of the hash table is checked.
- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

Delete Operation-

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as "deleted".

NOTE-

- During insertion, the buckets marked as "deleted" are treated like any other empty bucket.
- During searching, the search is not terminated on encountering the bucket marked as "deleted".
- The search terminates only after the required key or an empty bucket is found.

1. Linear probing technique

In this section we will see what is linear probing technique in open addressing scheme. There is an ordinary hash function h´(x) : U → {0, 1, . . ., m – 1}. In open addressing scheme, the actual hash function h(x) is taking the ordinary hash function h'(x) and attach some another part with it to make one linear equation.

The value of i| = 0, 1, . . ., m – 1. So we start from i = 0, and increase this until we get one freespace. So initially when i = 0, then the h(x, i) is same as h´(x).

Example

Suppose we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}

x	h(x, i) = (h'(x) + i) mod 20
96	i = 0, h(x, 0) = 16
48	i = 0, h(x, 0) = 8
63	i = 0, h(x, 0) = 3
29	i = 0, h(x, 0) = 9
87	i = 0, h(x, 0) = 7
77	i = 0, h(x, 0) = 17
48	i = 0, h(x, 0) = 8 i = 1, h(x, 1) = 9 i = 2, h(x, 2) = 10
65	i = 0, h(x, 0) = 5
69	i = 0, h(x, 0) = 9 i = 1, h(x, 1) = 10 i = 2, h(x, 2) = 11
94	i = 0, h(x, 0) = 14
61	i = 0, h(x, 0) = 1

Hash Table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	61		63		65		87	48	29	48	69			94		96	77		

Let's understand the linear probing through another example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

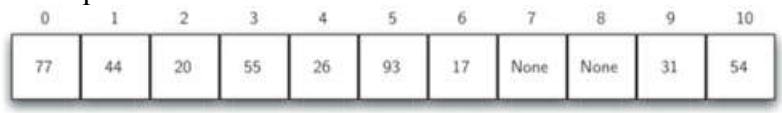
The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

2. Quadratic probing

- A variation of the linear probing idea is called quadratic probing. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on.

- This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on.
- In general, the i will be i^2rehash(pos)=(h+ i^2). In other words, quadratic probing uses a skip consisting of successive perfect squares.
- Figure shows our example values after they are placed using this technique.



Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

Index= $(5+0^2)\%10 = 5$

When i=1

Index = $(5+1^2)\%10 = 6$

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(9+0^2)\%10 = 9$

When i=1

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When i=0

Index= $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

When i=2

Index = $(7+2^2)\%10 = 1$

When i=3

Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

The final hash table would be:

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

3. Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from **(u+v*i)%m** where i=(0 to m-1)". In this case, u is the location computed using the hash function and v is equal to ($h_2(k)$%m).

Consider the same example that we use in quadratic probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and

$h_1(k) = 2k+3$

$h_2(k) = 3k+1$

key	Location (u)	v	probe
3	**((2*3)+3)%10 = 9**	-	1
2	**((2*2)+3)%10 = 7**	-	1
9	**((2*9)+3)%10 = 1**	-	1
6	**((2*6)+3)%10 = 5**	-	1
11	**((2*11)+3)%10 = 5**	(3(11)+1)%10 =4	3
13	**((2*13)+3)%10 = 9**	(3(13)+1)%10 = 0	
7	**((2*7)+3)%10 = 7**	(3(7)+1)%10 = 2	
12	**((2*12)+3)%10 = 7**	(3(12)+1)%10 = 7	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.

On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (5+4*0)%10 =5

When i=1

Index = (5+4*1)%10 = 9

When i=2

Index = (5+4*2)%10 = 3

Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.

The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (9+0*0)%10 = 9

We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.

The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (7 + 2*0)%10 = 7

When i=1

Index = (7+2*1)%10 = 9

When i=2

Index = (7+2*2)%10 = 1

When i=3

Index = (7+2*3)%10 = 3

When i=4

Index = (7+2*4)%10 = 5

When i=5

Index = (7+2*5)%10 = 7

When i=6

Index = (7+2*6)%10 = 9

When i=7

Index = (7+2*7)%10 = 1

When i=8

Index = (7+2*8)%10 = 3

When i=9

Index = (7+2*9)%10 = 5

Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert 7. Therefore, key 7 cannot be inserted in a hash table.

The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

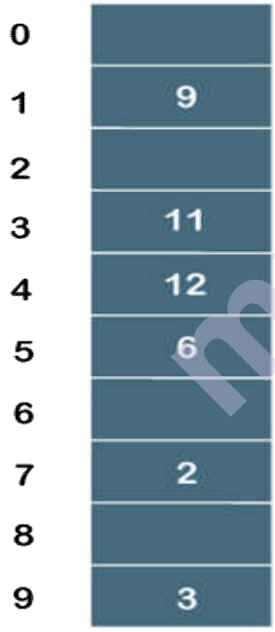
Index = (7+7*0)%10 = 7

When i=1

Index = (7+7*1)%10 = 4

Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.

The final hash table would be:



The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

Separate Chaining Vs Open Addressing-

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Comparison of Open Addressing Techniques-

	Linear Probing	**Quadratic Probing**	**Double Hashing**
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Conclusions-

- Linear Probing has the best cache performance but suffers from clustering.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double caching has poor cache performance but no clustering.

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

In open addressing, the value of load factor always lie between 0 and 1.

This is because-

- In open addressing, all the keys are stored inside the hash table.
- So, size of the table is always greater or at least equal to the number of keys stored in the table.

PRACTICE PROBLEM BASED ON OPEN ADDRESSING-

Problem-

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

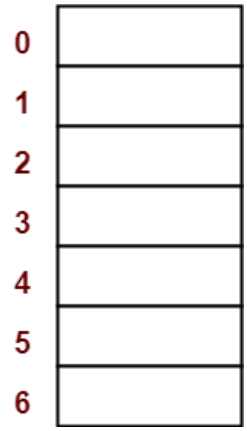
Use linear probing technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

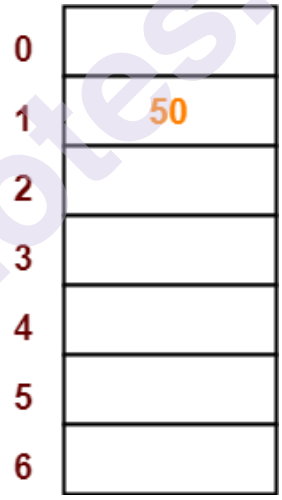
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



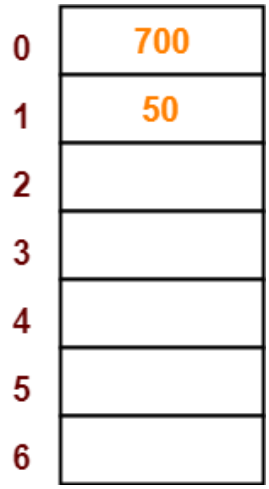
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



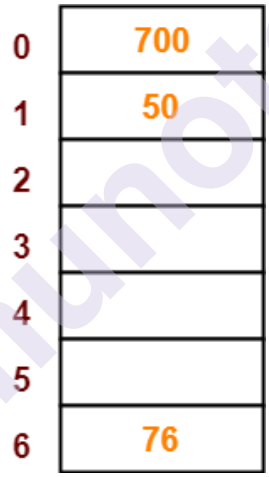
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-

0	700
1	50
2	85
3	
4	
5	
6	76

Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-

0	700
1	50
2	85
3	92
4	
5	
6	76

Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.

- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	
6	76

Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	101
6	76

12.6 SUMMARY

- **Linear Search** takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- **Binary Search** takes O(logn) time to perform the search in sorted arrays consisting of n elements.
- Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1).
- There are various types of hash functions available such as Mid Square Hash Function, Division Hash Function, Folding Hash Function
- A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.
- When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**.
- Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table- Insertion Operation, Deletion Operation, Searching Operation
- Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

12.7 QUESTIONS

1. Write a short note on Hashing.
2. Differentiate between Linear Search and Binary Search.
3. What are advantages and disadvantages of Hashing?
4. What are types of Hash Function?
5. Explain Collision I detail.
6. What techniques are used to avoid collision.
7. Explain quadratic probing.
8. Explain double hashing.
9. Differentiate between Separate Chaining and Open Addressing

12.8 REFERENCE FOR FURTHER READING

- http://www.cs.cmu.edu/~ab/15-111N09/Lectures/Lecture%2017%20-%20%20Hashing.pdf
- https://www.gatevidyalay.com/hashing/
- https://www.gatevidyalay.com/tag/hashing-in-data-structure-notes/
- http://web.mit.edu/16.070/www/lecture/hashing.pdf
- https://www.upgrad.com/blog/hashing-in-data-structure/
- https://research.cs.vt.edu/AVresearch/openalgoviz/VT/Hashing/tags/20090324-release/midsquare.php
- http://www.umsl.edu/~siegelj/information_theory/projects/HashingFunctionsInCryptography.html
- https://runestone.academy/runestone/books/published/pythonds/SortSearch/Hashing.html

❖❖❖❖

13

ADVANCED SORTING

Unit Structure:

13.0 OBJECTIVE

To study and analyze time complexity of various sorting algorithms

- To design, implement, and analyze insertion sort
- To design, implement, and analyze Merge sort
- To design, implement, and analyze Quick sort
- To design, implement, and analyze Radix sort (§23.4)

13.1 INTRODUCTION

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

13.2 MERGE SORT

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

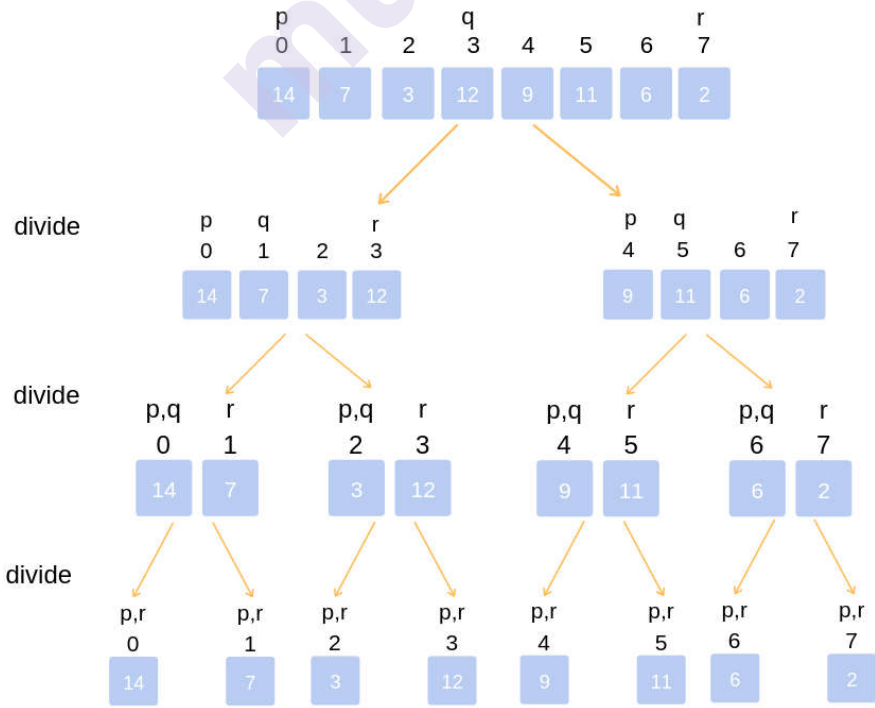
A merge sort works as follows:

Top-down Merge Sort Implementation:

The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as "What is required to be done to sort the array?" and having the answer as, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

Example: Let us consider an example to understand the approach better.

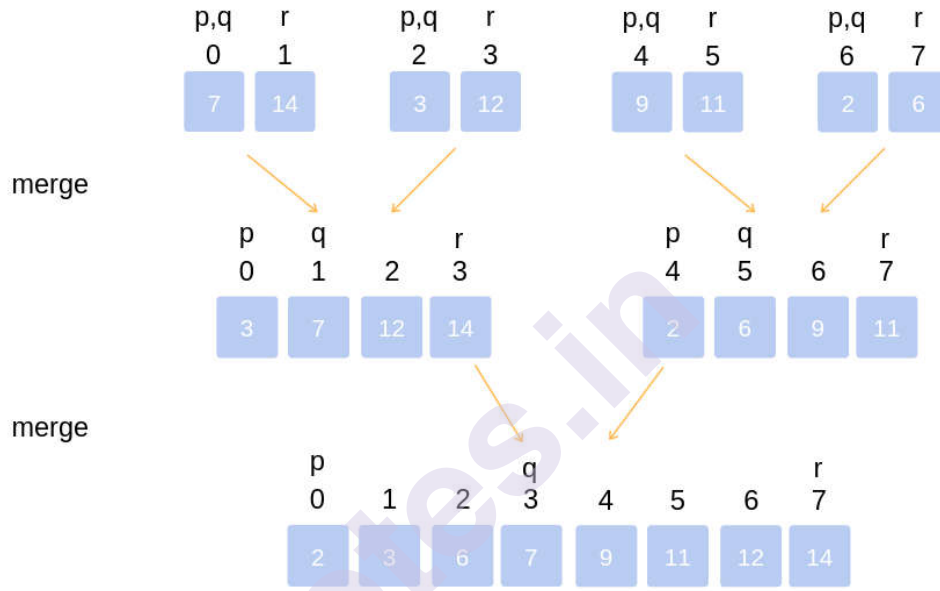
Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).



Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Merging of two lists done as follows:

The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.



Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
proceduremergesort( var a as array )
if ( n == 1 ) return a
var l1 as array = a[0] ... a[n/2]
var l2 as array = a[n/2+1] ... a[n]
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )
return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
```

```
var c as array
while ( a and b have elements )
if ( a[0] > b[0] )
add b[0] to the end of c
remove b[0] from b
else
add a[0] to the end of c
remove a[0] from a
end if
end while
while ( a has elements )
add a[0] to the end of c
remove a[0] from a
end while
while ( b has elements )
add b[0] to the end of c
remove b[0] from b
end while
return c
            end procedure
```


Implementation Of Merge Sort

```
defmerge_sort(alist, start, end):
'''Sorts the list from indexes start to end - 1 inclusive.'''
if end - start >1:
mid=(start + end)//2
merge_sort(alist, start, mid)
merge_sort(alist, mid, end)
merge_list(alist, start, mid, end)
 defmerge_list(alist, start, mid, end):
```

```
left=alist[start:mid]
right=alist[mid:end]
   k = start
i=0
   j =0
while(start + i< mid and mid + j < end):
if(left[i]<= right[j]):
alist[k]= left[i]
i=i + 1
else:
alist[k]= right[j]
          j = j + 1
      k = k + 1
if start + i< mid:
while k < end:
alist[k]= left[i]
i=i + 1
          k = k + 1
else:
while k < end:
alist[k]= right[j]
          j = j + 1
          k = k + 1
 alist=input('Enter the list of numbers: ').split()
alist=[int(x)for x inalist]
merge_sort(alist,0,len(alist))
print('Sorted list: ', end='')
print(alist)
```

Program Explanation

1. The user is prompted to enter a list of numbers.
2. The list is passed to the merge_sort function.
3. The sorted list is displayed.

Runtime Test Cases

Case 1:

Enter the list of numbers: 3 1 5 8 2 5 1 3

Sorted list: [1, 1, 2, 3, 3, 5, 5, 8]

Case 2:

Enter the list of numbers: 5 3 2 1 0

Sorted list: [0, 1, 2, 3, 5]

Case 3:

Enter the list of numbers: 1

Sorted list: [1]

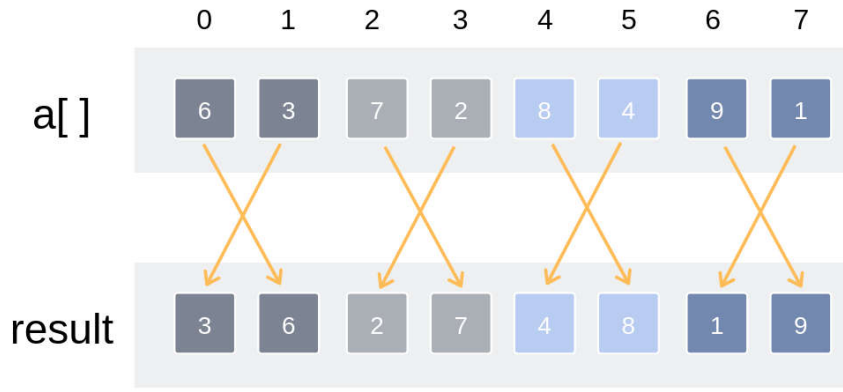
Bottom-Up Merge Sort Implementation:

The Bottom-Up merge sort approach uses iterative methodology. It starts with the "single-element" array, and combines two adjacent elements and also sorting the two at the same time. The combined-sorted arrays are again combined and sorted with each other until one single unit of sorted array is achieved.

Example: Let us understand the concept with the following example.

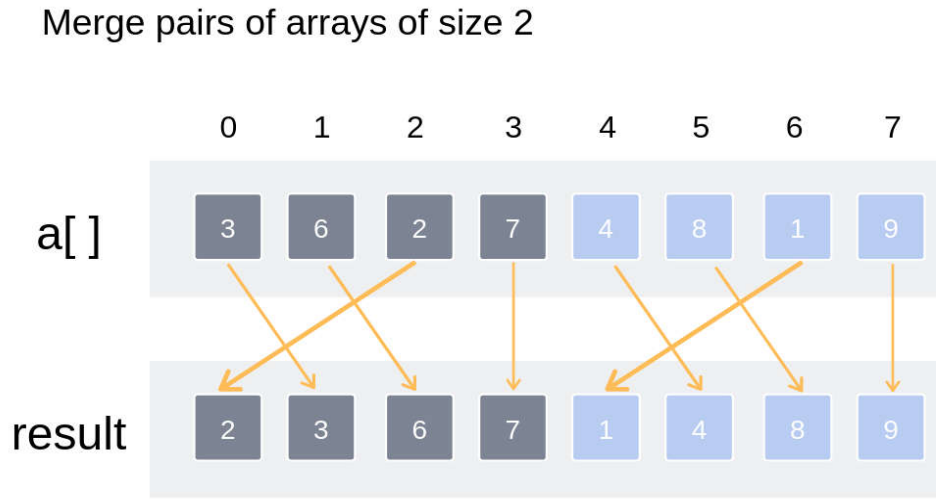
Iteration (1)

Merge pairs of arrays of size 1



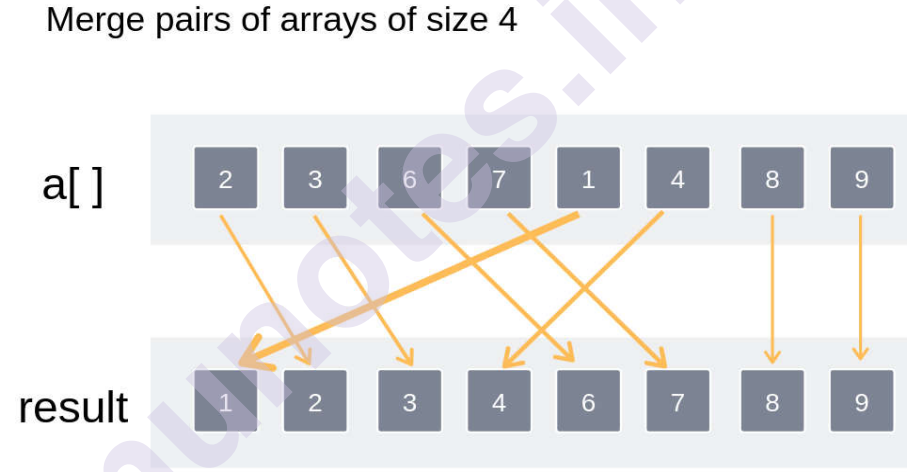
Merge pairs of arrays of size 1

Iteration (2)



Merge pairs of arrays of size 2

Iteration (3)



Merge pairs of arrays of size 4

Thus the entire array has been sorted and merged.

Complexity

Complexity	Best case	Average Case	Worst Case
Time Complexity	O(n log n)	O(n log n)	O(n log n)
Space Complexity			O(n)

13.3 QUICK SORT

The algorithm was developed by a British computer scientist Tony Hoare in 1959. The name "Quick Sort" comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster (twice or thrice faster) than any of the common sorting algorithms. It is one of the most efficient sorting algorithms and is based on the splitting of an array (partition) into smaller ones and swapping (exchange) based on the comparison with 'pivot' element selected. Due to this, quick sort is also called as "Partition Exchange" sort. Like Merge sort, Quick sort also falls into the category of divide and conquer approach of problem-solving methodology.

Application

Quicksort works in the following way

Before diving into any algorithm, its very much necessary for us to understand what are the real world applications of it. Quick sort provides a fast and methodical approach to sort any lists of things. Following are some of the applications where quick sort is used.

Commercial computing: Used in various government and private organizations for the purpose of sorting various data like sorting of accounts/profiles by name or any given ID, sorting transactions by time or locations, sorting files by name or date of creation etc.

Numerical computations: Most of the efficiently developed algorithms use priority queues and inturn sorting to achieve accuracy in all the calculations.

Information search: Sorting algorithms aid in better search of information and what faster way exists than to achieve sorting using quick sort.

Basically, quick sort is used everywhere for faster results and in the cases where there are space constraints.

Explanation

Taking the analogical view in perspective, consider a situation where one had to sort the papers bearing the names of the students, by name from A-Z. One might use the approach as follows:

Select any splitting value, say L. The splitting value is also known as **Pivot**.

Divide the stack of papers into two. A-L and M-Z. It is not necessary that the piles should be equal.

Repeat the above two steps with the A-L pile, splitting it into its significant two halves. And M-Z pile, split into its halves. The process is repeated until the piles are small enough to be sorted easily.

Ultimately, the smaller piles can be placed one on top of the other to produce a fully sorted and ordered set of papers.

The approach used here is **reduction** at each split to get to the single-element array.

At every split, the pile was divided and then the same approach was used for the smaller piles by using the method of recursion.

Technically, quick sort follows the below steps:
Step 1 − Make any element as pivot
Step 2 − Partition the array on the basis of pivot
Step 3 − Apply quick sort on left partition recursively
Step 4 − Apply quick sort on right partition recursively

Quick Sort Example:

Problem Statement

Consider the following array: `50, 23, 9, 18, 61, 32`. We need to sort this array in the most efficient manner without using extra place (inplace sorting).

Solution

Step 1:

- **Make any element as pivot:** Decide any value to be the pivot from the list. For convenience of code, we often select the rightmost index as pivot or select any at random and swap with rightmost. Suppose for two values "Low" and "High" corresponding to the first index and last index respectively.
  - In our case low is 0 and high is 5.
  - Values at low and high are 50 and 32 and value at pivot is 32.
- **Partition the array on the basis of pivot:** Call for partitioning which rearranges the array in such a way that pivot (32) comes to its actual position (of the sorted array). And to the left of the pivot, the array has all the elements less than it, and to the right greater than it.
  - In the partition function, we start from the first element and compare it with the pivot. Since 50 is greater than 32, we don't make any change and move on to the next element 23.
  - Compare again with the pivot. Since 23 is less than 32, we swap 50 and 23. The array becomes `23, 50, 9, 18, 61, 32`
  - We move on to the next element 9 which is again less than pivot (32) thus swapping it with 50 makes our array as `23, 9, 50, 18, 61, 32`.
  - Similarly, for next element 18 which is less than 32, the array becomes `23, 9, 18, 50, 61, 32`. Now 61 is greater than pivot (32), hence no changes.
  - Lastly, we swap our pivot with 50 so that it comes to the correct position.

Thus the pivot (32) comes at its actual position and all elements to its left are lesser, and all elements to the right are greater than itself.

Step 2: The main array after the first step becomes

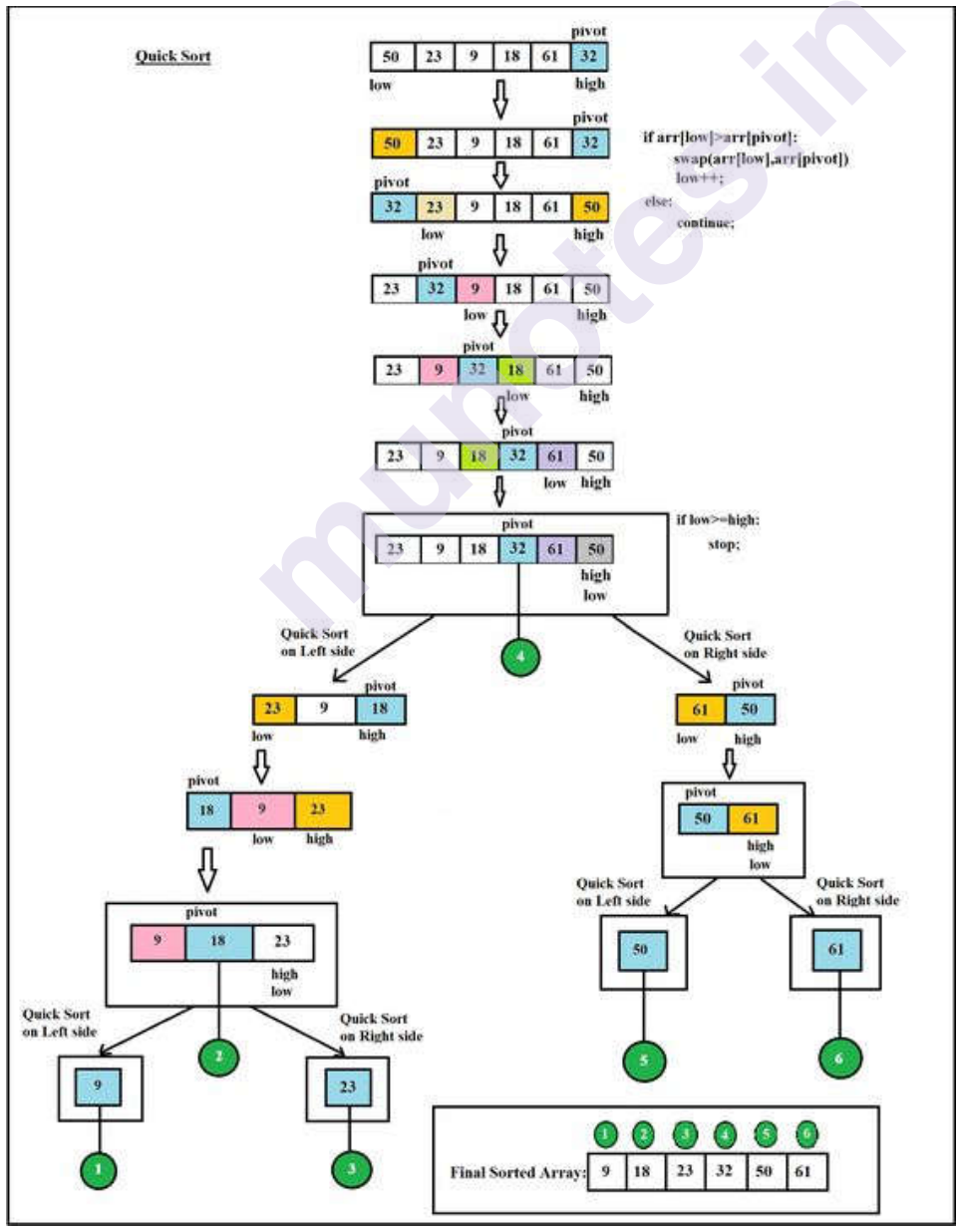
23, 9, 18, 32, 61, 50

Step 3: Now the list is divided into two parts:

1. Sublist before pivot element
2. Sublist after pivot element

Step 4: Repeat the steps for the left and right sublists recursively.The final array thus becomes `9, 18, 23, 32, 50, 61`.

The following diagram depicts the workflow of the Quick Sort algorithm which was described above.



Pseudocode of Quick Sort Algorithm:

```
/**
* The main function that implements quick sort.
* @Parameters: array, starting index and ending index
*/
quickSort(arr[], low, high)
{
if (low < high)
    {
        // pivot_index is partitioning index, arr[pivot_index] is now at correct place in sorted array
pivot_index = partition(arr, low, high);


quickSort(arr, low, pivot_index - 1);  // Before pivot_index
quickSort(arr, pivot_index + 1, high); // After pivot_index
    }
}
```

Implementation of Quick Sort

```
def quicksort(alist, start, end):
'''Sorts the list from indexes start to end - 1 inclusive.'''
if end - start >1:
        p =partition(alist, start, end)
quicksort(alist, start, p)
quicksort(alist, p + 1, end)
 def partition(alist, start, end):
pivot=alist[start]
i= start + 1
    j = end - 1
 whileTrue:
```

```
while(i<= j andalist[i]<= pivot):
i=i + 1
while(i<= j andalist[j]>= pivot):
        j = j - 1
 ifi<= j:
alist[i],alist[j]=alist[j],alist[i]
else:
alist[start],alist[j]=alist[j],alist[start]
return j
 alist=input('Enter the list of numbers: ').split()
alist=[int(x)for x inalist]
quicksort(alist,0,len(alist))
print('Sorted list: ', end='')
print(alist)
```

Program Explanation

1. The user is prompted to enter a list of numbers.
2. The list is passed to the quicksort function.
3. The sorted list is displayed.

Runtime Test Cases

Case 1:

Enter the list of numbers: 5 2 8 10 3 0 4

Sorted list: [0, 2, 3, 4, 5, 8, 10]

Case 2:

Enter the list of numbers: 7 4 3 2 1

Sorted list: [1, 2, 3, 4, 7]

Case 3:

Enter the list of numbers: 2

Sorted list: [2]

Complexity Analysis

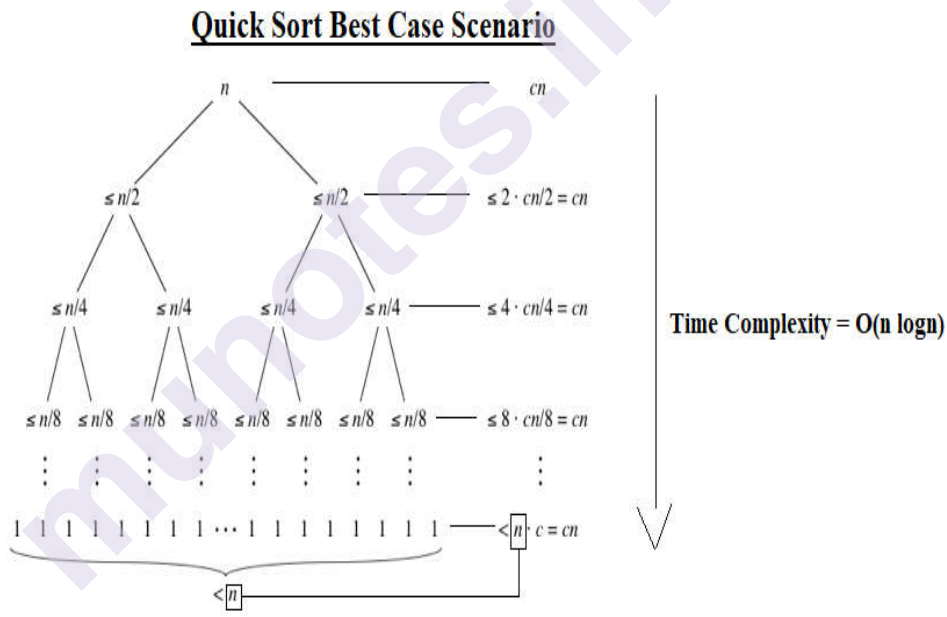
Time Complexity of Quick sort

- **Best case scenario:** The best case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either are equal or are have size difference of 1 of each other.

  - Case 1: The case when sizes of sublist on either side of pivot becomes equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have `(n-1)/2` elements.

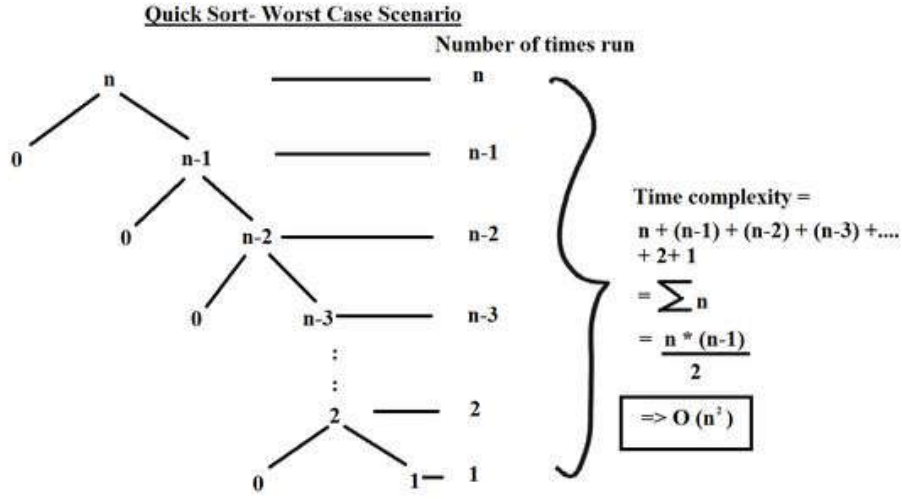
  - Case 2: The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, `n`, of elements. One partition will have `n/2` elements with the other having `(n/2)-1`.

In either of these cases, each partition will have at most `n/2` elements, and the tree representation of the subproblem sizes will be as below:



The best-case complexity of the quick sort algorithm is O(n logn)

- **Worst case scenario:** This happens when we encounter the most unbalanced partitions possible, then the original call takes `n` time, the recursive call on `n-1` elements will take `(n-1)` time, the recursive call on `(n-2)` elements will take `(n-2)` time, and so on. The worst case time complexity of Quick Sort would be **O(n2)**.



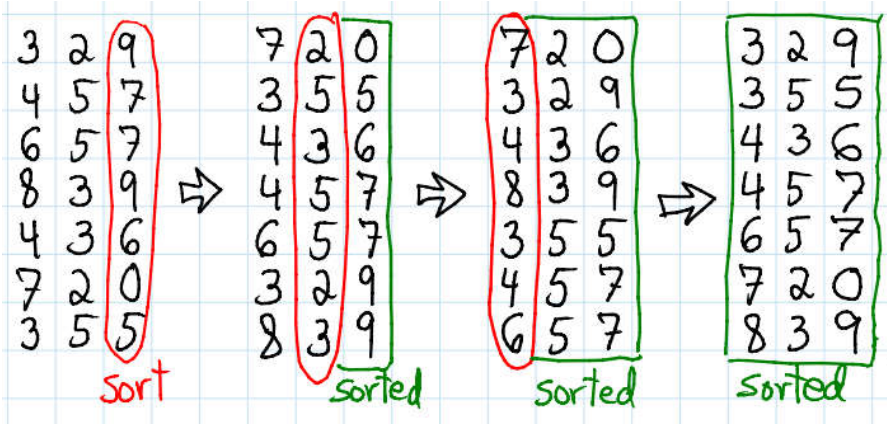
Space Complexity of Quick sort

The space complexity is calculated based on the space used in the recursion stack. The worst case space used will be `O(n)` . The average case space used will be of the order `O(log n)`. The worst case space complexity becomes `O(n)`, when the algorithm encounters its worst case where for getting a sorted list, we need to make `n` recursive calls.

13.4 RADIX SORT

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers. Because radix sort is not comparison based, it is not bounded by $\Omega(n \log n)$ for running time — in fact, radix sort can perform in linear time.

Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers without having to potentially decrease the efficiency by increasing the range of keys the algorithm must sort over (since this might cause a lot of wasted time).



Radix sort takes in a list of n*n* integers which are in base b*b* (the radix) and so each number has at most d*d* digits where $d = \lfloor (\log_b(k) + 1) \rfloor$ and *k* is the largest number in the list. For example, three digits are needed to represent decimal 104(in base 10). It is important that radix sort can work with any base since the running time of the algorithm, $O(d(n+b))$, depends on the base it uses. The algorithm runs in linear time when *b* and *n* are of the same size magnitude, so knowing *n*, *b* can be manipulated to optimize the running time of the algorithm.

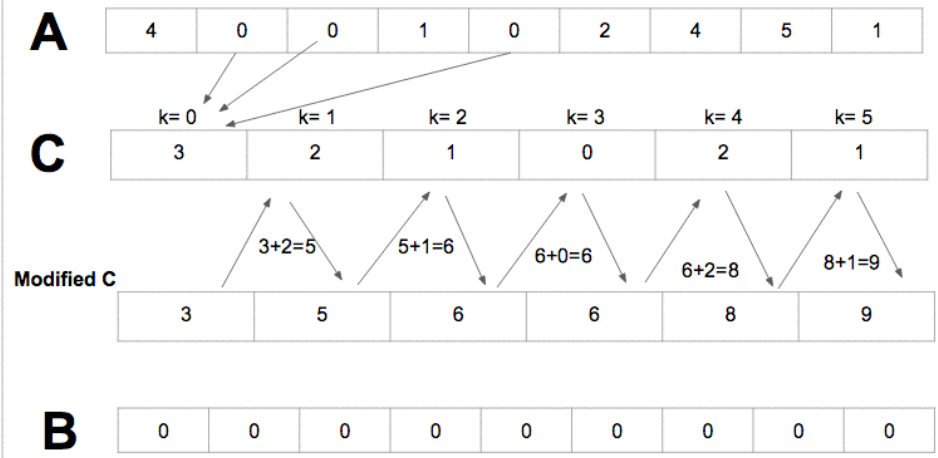
Radix sort works by sorting each digit from least significant digit to most significant digit. So in base 10 (the decimal system), radix sort would sort by the digits in the 1's place, then the 10's place, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value. This means that for a three-digit number in base 10, counting sort will be called to sort the 1's place, then it will be called to sort the 10's place, and finally, it will be called to sort the 100's place, resulting in a completely sorted list. Here is a quick refresher on the counting sort algorithm.

Counting Sort Subroutine

Counting sort uses three lists: the input list, $A[0,1,\ldots,n]$, the output list, $B[0,1,\ldots,n]$, and a list that serves as temporary memory, $C[0,1,\ldots,k]$. Note that *A* and *B* have *n* slots (a slot for each element), while *C* contains *k* slots (a slot for each key value).

Counting sort starts by going through *A*, and for each element $A[i]$, it goes to the index of *C* that has the same value as $A[i]$ (so it goes to $C[A[i]]$) and increments the value of $C[A[i]]$ by one. This means that if *A* has seven 0's in its list, after counting sort has gone through all *n* elements of *A*, the value at $C[0]$ will be 7. Similarly, if *A* has two 4's, after counting sort has gone through all of the elements of *A*, $C[4]$ (using 0 indexing) will be equal to 2.

In this step, *C* keeps track of how many elements in *A* there are that have the same value of a particular index in *C*. In other words, the indices of *C* correspond to the *values* of elements in *A*, and the *values* in *C* correspond to the total number of times that a value in *A* appears in *A*.



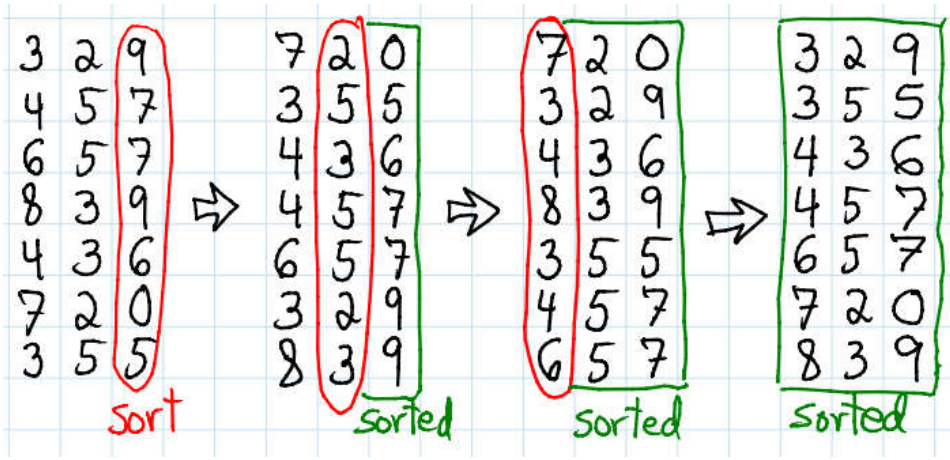
Radix sort is a **stable sort**, which means it preserves the relative order of elements that have the same key value. This is very important. For example, since the list of numbers [56,43,51,58] will be sorted as [51,43,56,58] when the 1's place is sorted (since $1 < 3 < 6 < 8$) and on the second pass, when the 10's place is being sorted, the sort will see that three of the four values are 5.

To preserve the sorting that the algorithm determined while sorting the 1's place, it is important to maintain relative order (namely $1 < 6 < 8$) between the numbers with the same value in the 10's place (or whatever place value is currently being sorted).

The second pass of the radix sort will produce [43,51,56,58].

Counting sort can only sort one place value of a given base. For example, a counting sort for base-10 numbers can only sort digits zero through nine. To sort two-digit numbers, counting sort would need to operate in base-100. Radix sort is more powerful because it can sort multi-digit numbers without having to search over a wider range of keys (which would happen if the base was larger).

In the image showing radix sort below, notice that each column of numbers (each place value) is sorted by the digit in question before the algorithm moves on to the next place value. This shows how radix sort preserves the relative order of digits with the same value at a given place value — remember, 66 and 68 will both appear as 66's in the 10's column, but $68 > 66$, so the order determined in the 1's column, that $8 > 6$ must be preserved for the sort to work properly and produce the correct answer.



Implementation of Radix Sort

```
defradix_sort(alist, base=10):
ifalist==[]:
return
 defkey_factory(digit, base):
def key(alist, index):
return((alist[index]//(base**digit)) % base)
return key
largest=max(alist)
exp=0
while base**exp<= largest:
alist=counting_sort(alist, base - 1,key_factory(exp, base))
exp=exp + 1
returnalist
 defcounting_sort(alist, largest, key):
   c =[0]*(largest + 1)
foriinrange(len(alist)):
c[key(alist,i)]= c[key(alist,i)] + 1
 # Find the last index for each element
c[0]= c[0] - 1# to decrement each element for zero-based indexing
foriinrange(1, largest + 1):
c[i]= c[i] + c[i - 1]
```

```
 result=[None]*len(alist)
for i in range(len(alist) - 1, -1, -1):
result[c[key(alist,i)]]=alist[i]
c[key(alist,i)]= c[key(alist,i)] - 1
 return result
 alist=input('Enter the list of (nonnegative) numbers: ').split()
alist=[int(x) for x in alist]
sorted_list=radix_sort(alist)
print('Sorted list: ', end='')
print(sorted_list)
```

Program Explanation

1. The user is prompted to enter a list of numbers.
2. The list is passed to the radix_sort function and the returned list is the sorted list.
3. The sorted list is displayed.

Runtime Test Cases

Case 1:

Enter the list of (nonnegative) numbers: 38 20 1 3 4 0 2 5 1 3 8 2 9 10

Sorted list: [0, 1, 1, 2, 2, 3, 3, 4, 5, 8, 9, 10, 20, 38]

Case 2:

Enter the list of (nonnegative) numbers: 7 5 3 2 1

Sorted list: [1, 2, 3, 5, 7]

Case 3:

Enter the list of (nonnegative) numbers: 3

Sorted list: [3]

Complexity of Radix Sort

Radix sort will operate on n d-digit numbers where each digit can be one of at most b different values (since bb is the base being used). For example, in base 10, a digit can be 0,1,2,3,4,5,6,7,8,or 9.

Radix sort uses counting sort on each digit. Each pass over n d-digit numbers will take $O(n+b)$ time, and there are d passes total. Therefore, the total running time of radix sort is $O(d(n+b))$. When d is a constant

and b isn't much larger than n (in other words, $b=O(n)$), then radix sort takes linear time.

13.5 SORTING LINKED LIST

1. Merge sort algorithm for a singly linked list

The Following solution uses the frontBackSplit(source) and sortedMerge() method to solve this problem efficiently.

```
# A Linked List Node
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next
  # Function to print a given linked list
defprintList(head):
     ptr = head
    while ptr:
        print(ptr.data, end=" —> ")
        ptr = ptr.next
    print("None")
 # Takes two lists sorted in increasing order and merge their nodes
# to make one big sorted list, which is returned
defsortedMerge(a, b):
     # base cases
    if a is None:
        return b
    elif b is None:
        return a
     # pick either `a` or `b`, and recur
    if a.data<= b.data:
        result = a
        result.next = sortedMerge(a.next, b)
    else:
        result = b
        result.next = sortedMerge(a, b.next)
     return result
```

```
 '''
   Split the given list's nodes into front and back halves,
   If the length is odd, the extra node should go in the front list.
   It uses the fast/slow pointer strategy
'''
 deffrontBackSplit(source):
    # if the length is less than 2, handle it separately
   if source is None or source.next is None:
      return source, None
    (slow, fast) = (source, source.next)
    # advance `fast` two nodes, and advance `slow` one node
   while fast:
       fast = fast.next
      if fast:
         slow = slow.next
         fast = fast.next
    # `slow` is before the midpoint of the list, so split it in two
   # at that point.
   ret = (source, slow.next)
   slow.next = None
    return ret
 # Sort a given linked list using the merge sort algorithm
defmergesort(head):
    # base case — length 0 or 1
   if head is None or head.next is None:
      return head
    # split `head` into `a` and `b` sublists
   front, back = frontBackSplit(head)
    # recursively sort the sublists
   front = mergesort(front)
   back = mergesort(back)
    # answer = merge the two sorted lists
   return sortedMerge(front, back)
 if __name__ == '__main__':
    # input keys
```

```
keys = [8, 6, 4, 9, 3, 1]
head = None
for key in keys:
    head = Node(key, head)
# sort the list
head = mergesort(head)
# print the sorted list
printList(head)
```

```
Output:


1 -> 3 -> 4 -> 6 -> 8 -> 9 -> None
```

2. Use Quick Sort to Sort a Linear Linked List

Quicksort algorithm is based on the concept of divide and conquer, where we do all the main work of sorting while dividing the given data structure(can be an array or in this case a Linked List) and during merging the data back, absolutely no processing is done, data is simply combined back together.

Quick Sort is also known as Partition-Exchange Sort.

In the quick sort algorithm, the given dataset is divided into three sections,

1. Data elements less than the pivot
2. The data element which is the pivot
3. And the data elements greater than the pivot.

Also in this case, when we want to use quicksort with a linked list, the main idea is to **swap pointers(in the node) rather than swapping data**.

Steps of the Algorithm:

The whole algorithm can be divided into two parts,

Partition:

1. Take the rightmost element as the **pivot**.
2. Traverse through the list:

1. If the current **node** has a value greater than the **pivot**, we will move it after the **tail**
2. else, keep it in the same position.

Quick Sort:

1. Call the `partition()` method, which will place the **pivot** at the right position and will return the **pivot**
2. Find the **tail** node of the left sublist i.e., left side of the **pivot** and recur the whole algorithm for the left list.
3. Now, recur the algorithm for the right list.

```
'''
sort a linked list using quick sort
'''
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None
class QuickSortLinkedList:
    def __init__(self):
        self.head=None

    defaddNode(self,data):
        if (self.head == None):
            self.head = Node(data)
            return
        curr = self.head
        while (curr.next != None):
            curr = curr.next
        newNode = Node(data)
        curr.next = newNode
    defprintList(self,n):
        while (n != None):
            print(n.data, end=" ")
            n = n.next
    ''' takes first and last node,but do not
    break any links in the whole linked list'''
    defparitionLast(self,start, end):
        if (start == end or start == None or end == None):
```

```
                return start
            pivot_prev = start
            curr = start
            pivot = end.data
            '''iterate till one before the end,
            no need to iterate till the end because end is pivot'''
            while (start != end):
                if (start.data< pivot):
                        # keep tracks of last modified item
                    pivot_prev = curr
                    temp = curr.data
                    curr.data = start.data
                    start.data = temp
                    curr = curr.next
start = start.next
            '''swap the position of curr i.e.
            next suitable index and pivot'''
            temp = curr.data
            curr.data = pivot
            end.data = temp
            ''' return one previous to current because
            current is now pointing to pivot '''
            return pivot_prev
        def sort(self, start, end):
            if(start == None or start == end or start == end.next):
                return
            # split list and partition recurse
            pivot_prev = self.paritionLast(start, end)
            self.sort(start, pivot_prev)
            '''
            if pivot is picked and moved to the start,
            that means start and pivot is same
            so pick from next of pivot
            '''
            if(pivot_prev != None and pivot_prev == start):
```

```
                self.sort(pivot_prev.next, end)
            # if pivot is in between of the list,start from next of pivot,
            # since we have pivot_prev, so we move two nodes
            elif (pivot_prev != None and pivot_prev.next != None):
                self.sort(pivot_prev.next.next, end)
if __name__ == "__main__":
    ll = QuickSortLinkedList()
    ll.addNode(30)
    ll.addNode(3)
    ll.addNode(4)
    ll.addNode(20)
    ll.addNode(5)
    n = ll.head
    while (n.next != None):
        n = n.next
    print("\nLinked List before sorting")
    ll.printList(ll.head)
    ll.sort(ll.head, n)
    print("\nLinked List after sorting");
    ll.printList(ll.head)
    # This code is contributed by humpheykibet.
```


3.Linked-list radix sort

Here's a small snippet to show how you can sort a linked-list using radix sort.

Radix sort isn't a comparison-based sort, which means it can attain O(n) performance. However if you're sorting a flat array, it has to do a lot of data shuffling. This is one of the few times where a linked-list actually has a performance advantage, as it can shuffle the lists around with only a single re-link operation.

```
# Python program for implementation of Radix Sort

# A function to do counting sort of arr[] according to

# the digit represented by exp.

defcountingSort(arr, exp1):
```

```
    n = len(arr)
    # The output array elements that will have sorted arr
    output = [0] * (n)
    # initialize count array as 0
    count = [0] * (10)
    # Store count of occurrences in count[]
    for i in range(0, n):
        index = arr[i] // exp1
        count[index % 10] += 1
    # Change count[i] so that count[i] now contains actual
    # position of this digit in output array
    for i in range(1, 10):
        count[i] += count[i - 1]
    # Build the output array
    i = n - 1
    while i>= 0:
        index = arr[i] // exp1
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1
    # Copying the output array to arr[],
    # so that arr now contains sorted numbers
    i = 0
    for i in range(0, len(arr)):
        arr[i] = output[i]
# Method to do Radix Sort
defradixSort(arr):
    # Find the maximum number to know number of digits
    max1 = max(arr)
```

```
    # Do counting sort for every digit. Note that instead
    # of passing digit number, exp is passed. exp is 10^i
    # where i is current digit number
    exp = 1
    while max1 / exp> 0:
        countingSort(arr, exp)
        exp *= 10
# Driver code
arr = [170, 45, 75, 90, 802, 24, 2, 66]
# Function Call
radixSort(arr)
for i in range(len(arr)):
    print(arr[i])
```

13.6 SUMMARY

- Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer.
- The top-down merge sort approach is the methodology which uses recursion mechanism.
- The Bottom-Up merge sort approach uses iterative methodology. It starts with the “single-element” array, and combines two adjacent elements and also sorting the two at the same time.
- The name "Quick Sort" comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster (twice or thrice faster) than any of the common sorting algorithms.
- **Radix sort** is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value).

13.7 QUESTIONS

1. Explain Merge Sort with example.
2. Write python code for Merge Sort.
3. Explain Quick Sort with example.

4. Write python code for Quick Sort.
5. Explain Radix Sort with example.
6. Write python code for Radix Sort.

13.8 REFERENCE FOR FURTHER READING

- https://www.interviewbit.com/tutorial/merge-sort-algorithm/
- https://brilliant.org/wiki/radix-sort/
- https://www.techiedelight.com/merge-sort-singly-linked-list/
- https://www.studytonight.com/post/use-quick-sort-to-sort-a-linear-linked-list
- https://daveparillo.github.io/cisc187-reader/recursion/properties.html
- https://medium.com/@frankzou4000/recursion-and-its-applications-4dc00ee94130
- https://www.sanfoundry.com/python-program-implement-radix-sort/

14

BINARY TREES

Unit Structure:

14.0 OBJECTIVE

Trees reflect structural relationships in the data. Trees are used to represent hierarchies. Trees provide an efficient insertion and searching. Trees are very flexible data, allowing to move subtrees around with minumum effort.

14.1 INTRODUCTION

We have all watched trees from our childhood. It has roots, stems, branches and leaves. It was observed long back that each leaf of a tree can be traced to root via a unique path. Hence tree structure was used to explain hierarchical relationships, e.g. family tree, animal kingdom classification, etc.

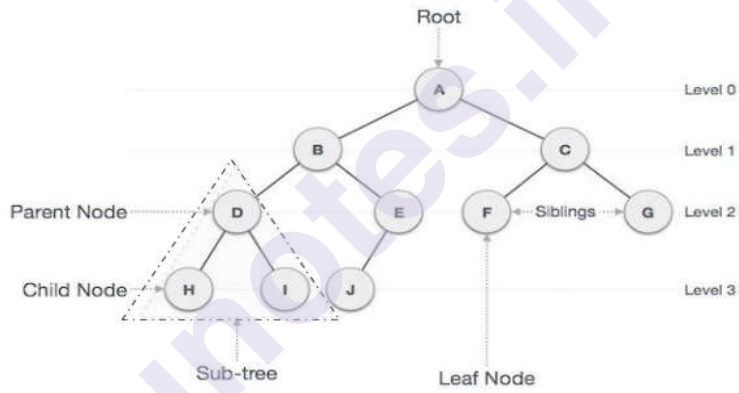
This hierarchical structure of trees is used in Computer science as an abstract data typefor various applications like data storage, search and sort algorithms. Let us explore this data type in detail.

14.2 TREE TERMINOLOGY

A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.
2. Each node has one parent only but can have multiple children.
3. Each node is connected to its children via edge.

Following diagram explains various terminologies used in a tree structure.



Terminology	Description	Example From Diagram
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.	Node A
Parent Node	Parent node is an immediate predecessor of a node.	B is parent of D & E
Child Node	All immediate successors of a node are its children.	D & E are children of B
Leaf	Node which does not have any child is called as leaf	H, I, J, F and G are leaf nodes

Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.	Line between A & B is edge
Siblings	Nodes with the same parent are called Siblings.	D & E are siblings
Path / Traversing	Path is a number of successive edges from source node to destination node.	A – B – E – J is path from node A to E
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.	A, B, C, D & E can have height. Height of A is no. of edges between A and H, as that is the longest path, which is 3. Height of C is 1
Levels of node	Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on	Level of H, I & J is 3. Level of D, E, F & G is 2
Degree of Node	Degree of a node represents the number of children of a node.	Degree of D is 2 and of E is 1
Sub tree	Descendants of a node represent subtree.	Nodes D, H, I represent one subtree.

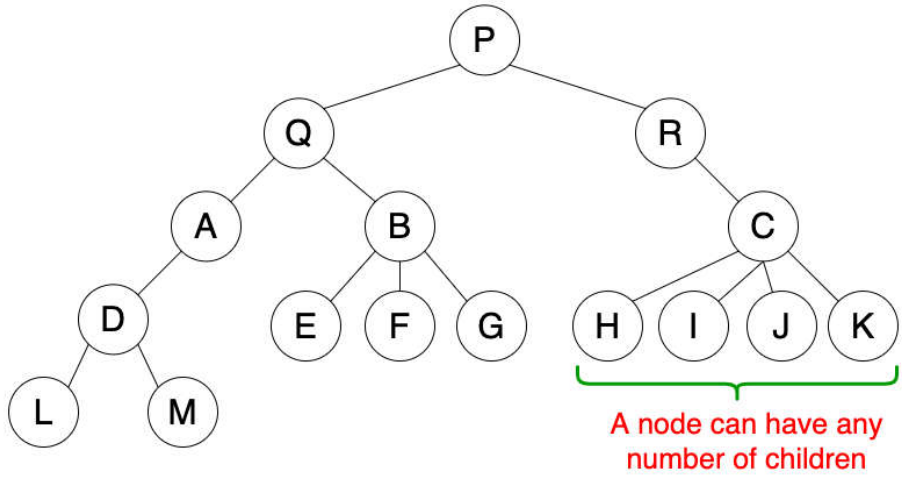
14.3 TYPES OF TREE

1. General Tree

A **general tree** is a tree data structure where there are no constraints on the hierarchical structure.

Properties

1. Follow properties of a tree.
2. A node can have any number of children.



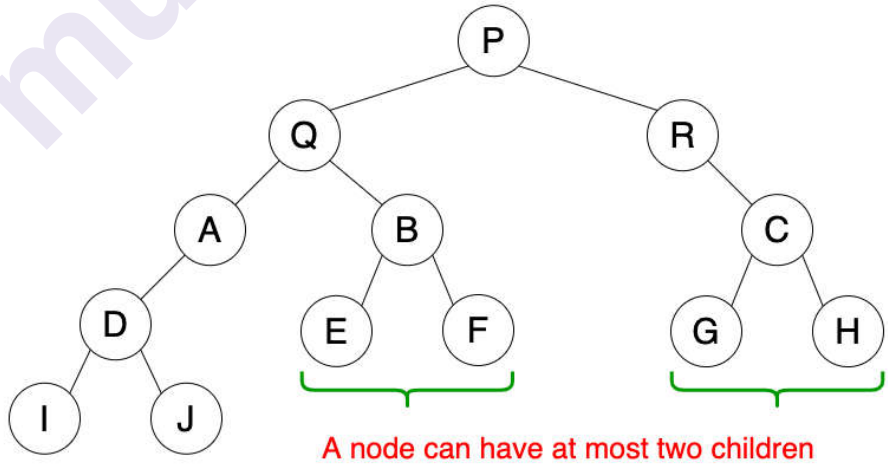
Usage

1. Used to store hierarchical data such as folder structures.
2. Binary Tree

A **binary tree** is a tree data structure where the following properties can be found.

Properties

1. Follow properties of a tree.
2. A node can have at most two child nodes (children).
3. These two child nodes are known as the **left child** and **right child**.



Usage

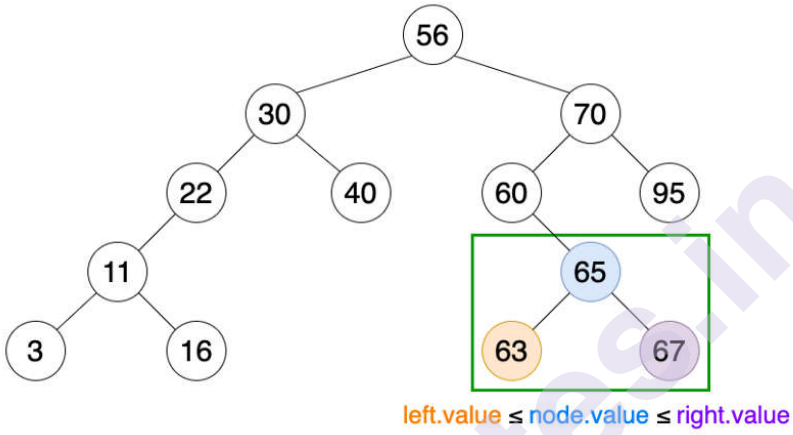
1. Used by compilers to build syntax trees.
2. Used to implement expression parsers and expression solvers.
3. Used to store router-tables in routers.

3. Binary Search Tree

A **binary search tree** is a more constricted extension of a binary tree.

Properties

1. Follow properties of a binary tree.
2. Has a unique property known as the **binary-search-tree property**. This property states that the value (or key) of the left child of a given node should be less than or equal to the parent value and the value of the right child should be greater than or equal to the parent value.



Usage

1. Used to implement simple sorting algorithms.
2. Can be used as priority queues.
3. Used in many search applications where data are constantly entering and leaving.

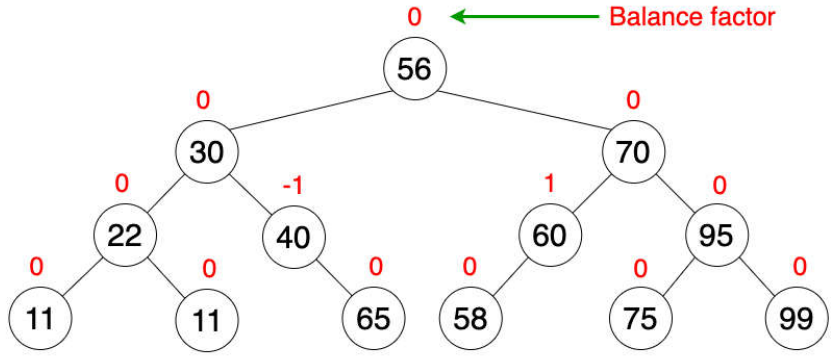
4. AVL tree

An **AVL tree** is a self-balancing binary search tree. This is the first tree introduced which automatically balances its height.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Each node stores a value called a **balance factor** which is the difference in height between its left subtree and right subtree.
4. All the nodes must have a balance factor of -1, 0 or 1.

After performing insertions or deletions, if there is at least one node that does not have a balance factor of -1, 0 or 1 then rotations should be performed to balance the tree (self-balancing).



Usage

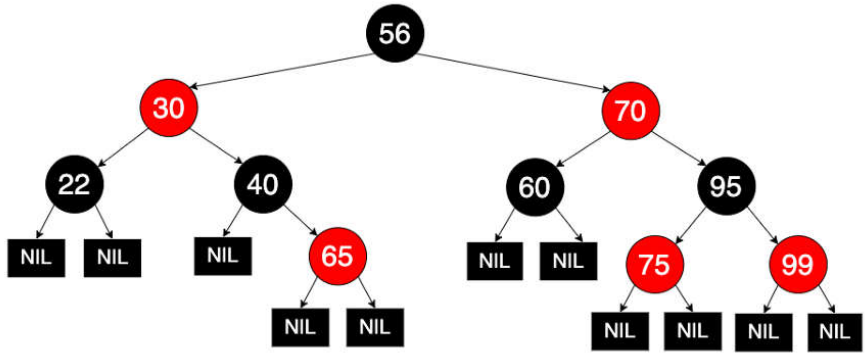
1. Used in situations where frequent insertions are involved.
2. Used in Memory management subsystem of the Linux kernel to search memory regions of processes during preemption.

5. Red-black tree

A red-black tree is a self-balancing binary search tree, where each node has a colour; red or black. The colours of the nodes are used to make sure that the tree remains approximately balanced during insertions and deletions.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Each node is either red or black.
4. The root is black (sometimes omitted).
5. All leaves (denoted as NIL) are black.
6. If a node is red, then both its children are black.
7. Every path from a given node to any of its leaf nodes must go through the same number of black nodes.



Usage

1. As a base for data structures used in computational geometry.
2. Used in the Completely Fair Scheduler used in current Linux kernels.
3. Used in the epoll system call implementation of Linux kernel.

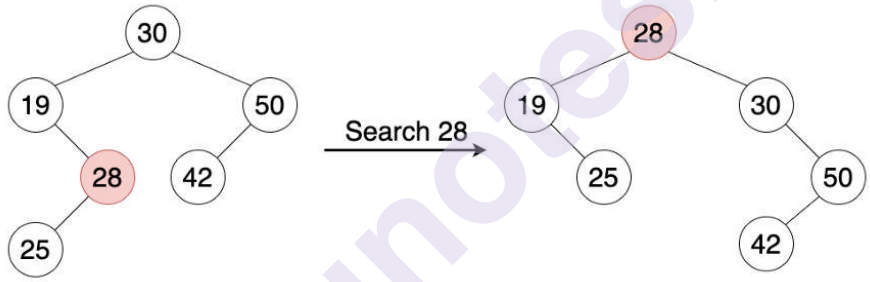
6. Splay tree

A **splay tree** is a self-balancing binary search tree.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Recently accessed elements are quick to access again.

After performing a search, insertion or deletion, splay trees perform an action called **splaying** where the tree is rearranged (using rotations) so that the particular element is placed at the root of the tree.



Usage

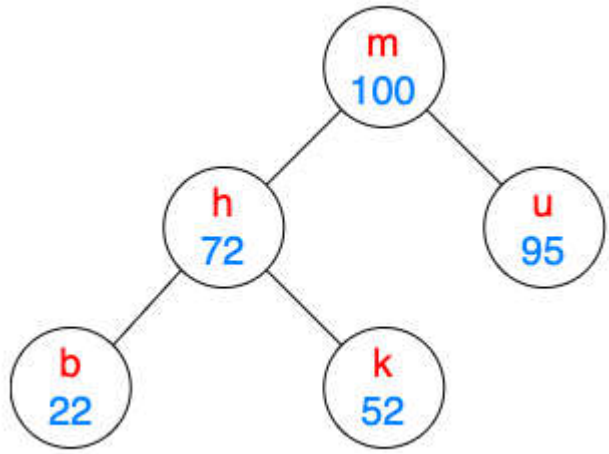
1. Used to implement caches
2. Used in garbage collectors.
3. Used in data compression

7. Treap

A **treap** (the name derived from **tree + heap**) is a binary search tree.

Properties

1. Each node has two values; a **key** and a **priority**.
2. The keys follow the binary-search-tree property.
3. The priorities (which are random values) follow the heap property.



Usage

1. Used to maintain authorization certificates in public-key cryptosystems.
2. Can be used to perform fast set operations.

8. B-tree

B tree is a self-balancing search tree and contains multiple nodes which keep data in sorted order. Each node has 2 or more children and consists of multiple keys.

Properties

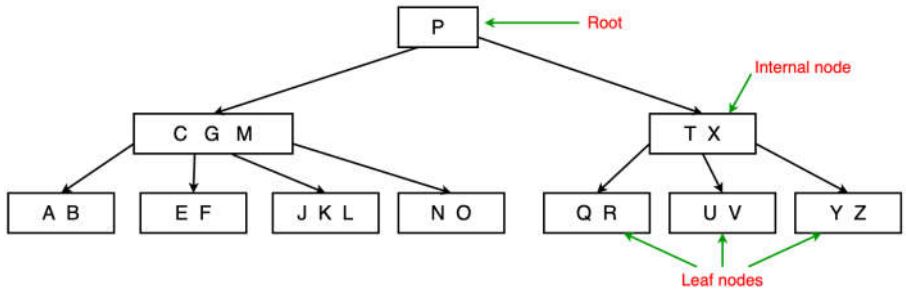
1. Every node x has the following:
   - x.n (the number of keys)
   - x.key(the keys stored in ascending order)
   - x.leaf (whether x is a leaf or not)

2. Every node x has (x.n + 1) children.

3. The keys x.key separate the ranges of keys stored in each sub-tree.

4. All the leaves have the same depth, which is the tree height.

5. Nodes have lower and upper bounds on the number of keys that can be stored. Here we consider a value $t \geq 2$, called **minimum degree** (or **branching factor**) of the B tree.
   - The root must have at least one key.
   - Every other node must have at least (t-1) keys and at most (2t-1) keys. Hence, every node will have at least t children and at most 2t children. We say the node is **full** if it has (2t-1) keys.



-

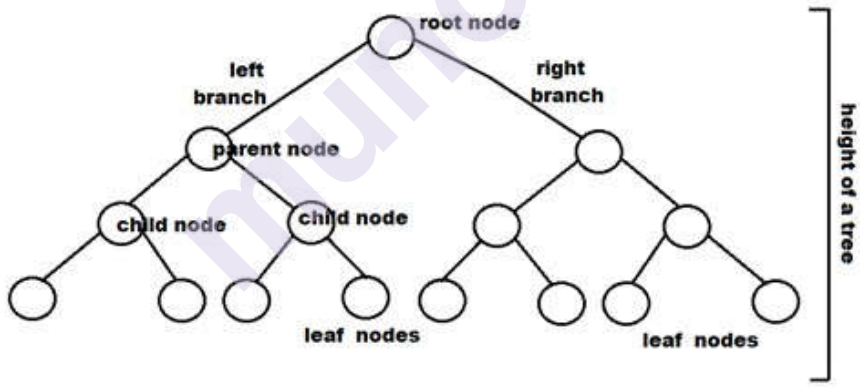
Usage

1. Used in database indexing to speed up the search.
2. Used in file systems to implement directories.

14.4 BINARY TREE

A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

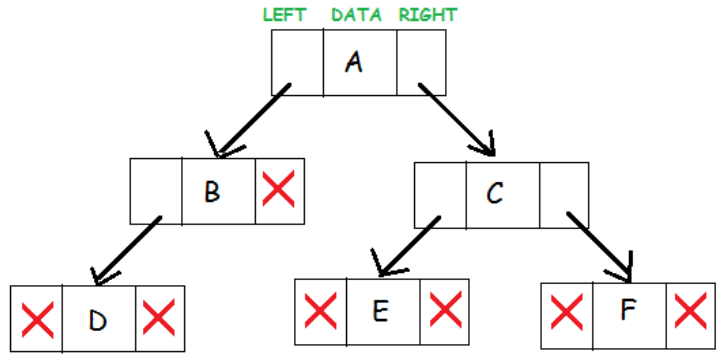
A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.



Binary Tree Components

There are three **binary tree components**. Every **binary tree** node has these three components associated with it. It becomes an essential concept for programmers to understand these three **binary tree components:**

1. Data element
2. Pointer to left subtree
3. Pointer to right subtree



Types of Binary Trees

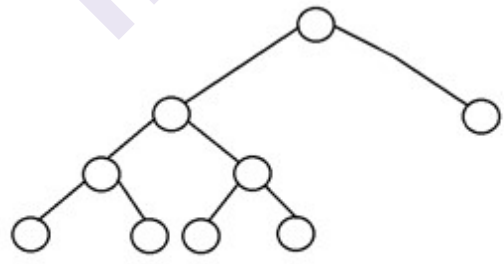
There are various **types of binary trees**, and each of these **binary tree types** has unique characteristics. Here are each of the **binary tree types** in detail:

1. Full Binary Tree

It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

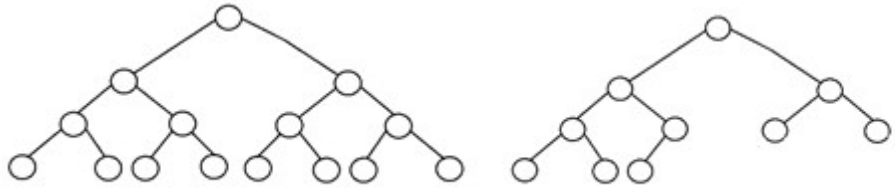
In other words, a full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree. Here, the quantity of leaf nodes is equal to the number of internal nodes plus one. The equation is like L=I+1, where L is the number of leaf nodes, and I is the number of internal nodes.

Here is the structure of a full binary tree:



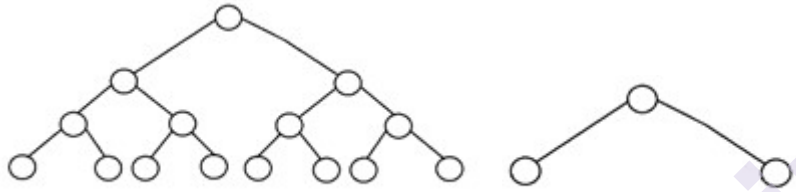
2. Complete Binary Tree

A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of a complete binary tree:



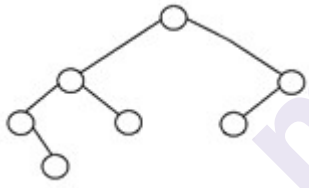
3. Perfect Binary Tree

A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height 'h' has 2h – 1 node. Here is the structure of a perfect binary tree:



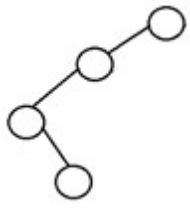
4. Balanced Binary Tree

A binary tree is said to be 'balanced' if the tree height is O(logN), where 'N' is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:



5. Degenerate Binary Tree

A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:



Benefits of a Binary Tree

- The search operation in a binary tree is faster as compared to other trees
- Only two traversals are enough to provide the elements in sorted order
- It is easy to pick up the maximum and minimum elements
- Graph traversal also uses binary trees
- Converting different postfix and prefix expressions are possible using binary trees

14.5 IMPLEMENTATION OF BINARY TREE

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties −

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have an arbiatry number of chid node.

We create a tree data structure in python by using the concept os node discussed earlier. We designate one node as root node and then add more nodes as child nodes. Below is program to create the root node.

Create Root

We just create a Node class and add assign a value to the node. This becomes tree with only a root node.

Example

```
class Node:
def __init__(self, data):
self.left = None
self.right = None
self.data = data
defPrintTree(self):
    print(self.data)
root = Node(10)
root.PrintTree()
```

Output

When the above code is executed, it produces the following result −

```
10
```

Inserting into a Tree

To insert into a tree we use the same node class created above and add a insert class to it. The insert class compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally the PrintTree class is used to print the tree.

Example

```
class Node:
def __init__(self, data):
self.left = None
self.right = None
self.data = data
def insert(self, data):
# Compare the new value with the parent node
    if self.data:
      if data <self.data:
        if self.left is None:
self.left = Node(data)
        else:
self.left.insert(data)
elif data >self.data:
          if self.right is None:
self.right = Node(data)
          else:
self.right.insert(data)
    else:
self.data = data
# Print the tree
```

```
defPrintTree(self):
    if self.left:
self.left.PrintTree()
    print( self.data),
    if self.right:
self.right.PrintTree()
# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

Output

When the above code is executed, it produces the following result −

```
3 6 12 14
```

14.6 TRAVERSING A TREE

Here, tree traversal means **traversing** or **visiting** each node of a tree. Linear data structures like Stack, Queue, and linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:

- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**

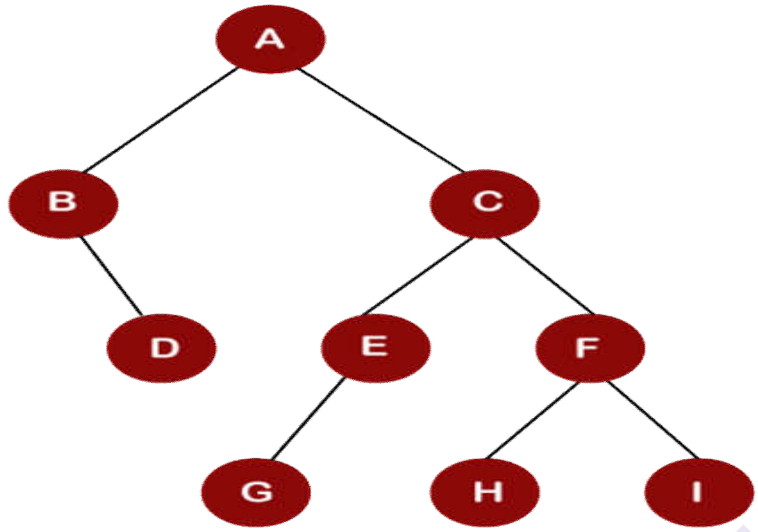
Let's look at each traversal one by one.

1. Inorder Traversal

An inorder traversal is a traversal technique that follows the policy, i.e., **Left Root Right**. Here, Left Root Right means that the left subtree of the root node is traversed first, then the root node, and then the right subtree of the root node is traversed. Here, inorder name itself suggests that the root node comes in between the left and the right subtrees.

Let's understand the inorder traversal through an example.

Consider the below tree for the inorder traversal.



First, we will visit the left part, then root, and then the right part of performing the inorder traversal. In the above tree, A is a root node, so we move to the left of the A, i.e., B. As node B does not have any left child so B will be printed as shown below:

After visiting node B, we move to the right child of node B, i.e., D. Since node D is a leaf node, so node D gets printed as shown below:

The left part of node A is traversed. Now, we will visit the root node, i.e., A, and it gets printed as shown below:

Once the traversing of left part and root node is completed, we move to the right part of the root node. We move to the right child of node A, i.e., C. The node C has also left child, i.e., E and E has also left child, i.e., G. Since G is a leaf node, so G gets printed as shown below:

The root node of G is E, so it gets printed as shown below:

Since E does not have any right child, so we move to the root of the E node, i.e., C. C gets printed as shown below:

Once the left part of node C and the root node, i.e., C, are traversed, we move to the right part of Node C. We move to the node F and node F has a left child, i.e., H. Since H is a leaf node, so it gets printed as shown below:

Now we move to the root node of H, i.e., F and it gets printed as shown below:

After visiting the F node, we move to the right child of node F, i.e., I, and it gets printed as shown below:

Therefore, the inorder traversal of the above tree is B, D, A, G, E, C, H, F, I.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then we create a insert function to add data to the tree. Finally the Inorder traversal logic is implemented by creating an empty list and adding the left node first followed by the root or parent node. At last the left node is added to

complete the Inorder traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

```
classNode:
def __init__(self, data):
self.left=None
self.right=None
self.data= data
# Insert Node
def insert(self, data):
ifself.data:
if data <self.data:
ifself.leftisNone:
self.left=Node(data)
else:
self.left.insert(data)
elif data >self.data:
ifself.rightisNone:
self.right=Node(data)
else:
self.right.insert(data)
else:
self.data= data
# Print the Tree
defPrintTree(self):
ifself.left:
self.left.PrintTree()
print(self.data),
ifself.right:
self.right.PrintTree()
# Inorder traversal
# Left -> Root -> Right
definorderTraversal(self, root):
res=[]
if root:
res=self.inorderTraversal(root.left)
```

```
res.append(root.data)
res= res +self.inorderTraversal(root.right)
return res
root=Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.inorderTraversal(root))
```

When the above code is executed, it produces the following result −

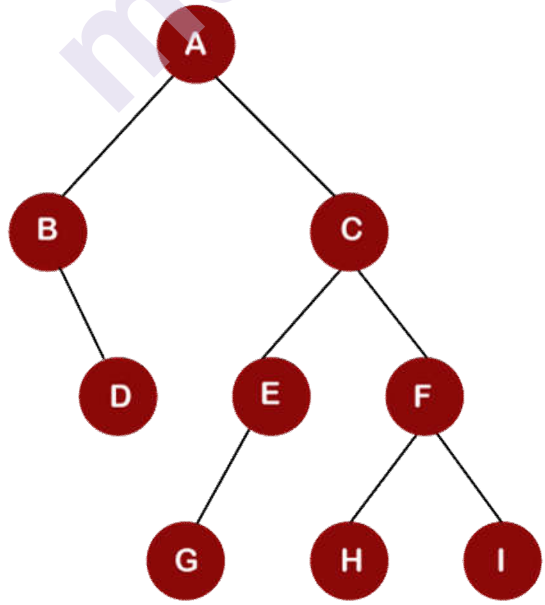
```
[10,14,19,27,31,35,42]
```

2. Preorder Traversal

A preorder traversal is a traversal technique that follows the policy, i.e., **Root Left Right**. Here, Root Left Right means root node of the tree is traversed first, then the left subtree and finally the right subtree is traversed. Here, the Preorder name itself suggests that the root node would be traversed first.

Let's understand the Preorder traversal through an example.

Consider the below tree for the Preorder traversal.



To perform the preorder traversal, we first visit the root node, then the left part, and then we traverse the right part of the root node. As node A is the root node in the above tree, so it gets printed as shown below:

Once the root node is traversed, we move to the left subtree. In the left subtree, B is the root node for its right child, i.e., D. Therefore, B gets printed as shown below:

Since node B does not have a left child, and it has only a right child; therefore, D gets printed as shown below:

Once the left part of the root node A is traversed, we move to the right part of node A. The right child of node A is C. Since C is a root node for all the other nodes; therefore, C gets printed as shown below:

Now we move to the left child, i.e., E of node C. Since node E is a root node for node G; therefore, E gets printed as shown below:

The node E has a left child, i.e., G, and it gets printed as shown below:

Since the left part of the node C is completed, so we move to the right part of the node C. The right child of node C is node F. The node F is a root node for the nodes H and I; therefore, the node F gets printed as shown below:

Once the node F is visited, we will traverse the left child, i.e., H of node F as shown below:

Now we will traverse the right child, i.e., I of node F, as shown below:

Therefore, the preorder traversal of the above tree is A, B, D, C, E, G, F, H, I.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then we create a insert function to add data to the tree. Finally the Pre-order traversal logic is implemented by creating an empty list and adding the root node first followed by the left node. At last the right node is added to complete the Pre-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

```
classNode:


def __init__(self, data):


self.left=None

self.right=None

self.data= data

# Insert Node

def insert(self, data):


ifself.data:

if data <self.data:

ifself.leftisNone:
```

```
self.left=Node(data)
else:
self.left.insert(data)
elif data >self.data:
ifself.rightisNone:
self.right=Node(data)
else:
self.right.insert(data)
else:
self.data= data

# Print the Tree
defPrintTree(self):
ifself.left:
self.left.PrintTree()
print(self.data),
ifself.right:
self.right.PrintTree()

# Preorder traversal
# Root -> Left ->Right
defPreorderTraversal(self, root):
res=[]
if root:
res.append(root.data)
res= res +self.PreorderTraversal(root.left)
res= res +self.PreorderTraversal(root.right)
return res
```

```
root=Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.PreorderTraversal(root))
```

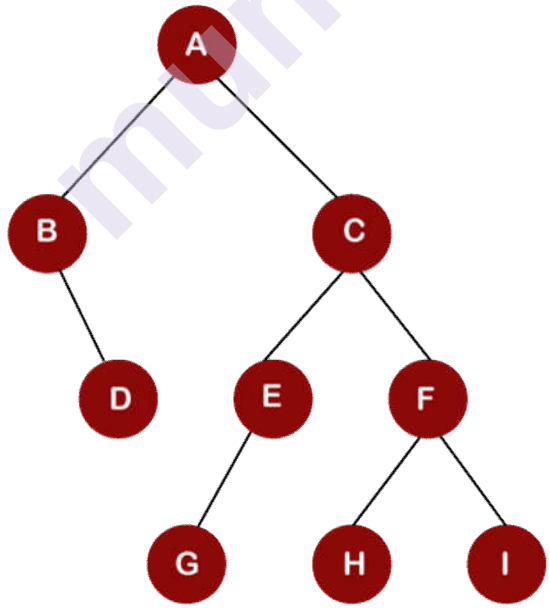
When the above code is executed, it produces the following result −

```
[27,14,10,19,35,31,42]
```

3. Postorder Traversal

A Postorder traversal is a traversal technique that follows the policy, i.e., **Left Right Root**. Here, Left Right Root means the left subtree of the root node is traversed first, then the right subtree, and finally, the root node is traversed. Here, the Postorder name itself suggests that the root node of the tree would be traversed at the last.

Let's understand the Postorder traversal through an example.
Consider the below tree for the Postorder traversal.



To perform the postorder traversal, we first visit the left part, then the right part, and then we traverse the root node. In the above tree, we move to the left child, i.e., B of node A. Since B is a root node for the node D; therefore, the right child, i.e., D of node B, would be traversed first and then B as shown below:

Once the traversing of the left subtree of node A is completed, then the right part of node A would be traversed. We move to the right child of node A, i.e., C. Since node C is a root node for the other nodes, so we move to the left child of node C, i.e., node E. The node E is a root node, and node G is a left child of node E; therefore, the node G is printed first and then E as shown below:

Once the traversal of the left part of the node C is traversed, then we move to the right part of the node C. The right child of node C is node F. Since F is also a root node for the nodes H and I; therefore, the left child 'H' is traversed first and then the right child 'I' of node F as shown below:

After traversing H and I, node F is traversed as shown below:

Once the left part and the right part of node C are traversed, then the node C is traversed as shown below:

In the above tree, the left subtree and the right subtree of root node A have been traversed, the node A would be traversed.

Therefore, the Postorder traversal of the above tree is D, B, G, E, H, I, F, C, A.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then we create a insert function to add data to the tree. Finally the Post-order traversal logic is implemented by creating an empty list and adding the left node first followed by the right node. At last the root or parent node

is added to complete the Post-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed

```
classNode:


def __init__(self, data):


self.left=None
self.right=None
self.data= data
# Insert Node
def insert(self, data):


ifself.data:
if data <self.data:
ifself.leftisNone:
self.left=Node(data)
else:
self.left.insert(data)
elif data >self.data:
ifself.rightisNone:
self.right=Node(data)
else:
self.right.insert(data)
else:
self.data= data


# Print the Tree
defPrintTree(self):
ifself.left:
```

```
self.left.PrintTree()
print(self.data),
ifself.right:
self.right.PrintTree()
# Postorder traversal
# Left ->Right -> Root
defPostorderTraversal(self, root):
res=[]
if root:
res=self.PostorderTraversal(root.left)
res= res +self.PostorderTraversal(root.right)
res.append(root.data)
return res
root=Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.PostorderTraversal(root))
```


When the above code is executed, it produces the following result −

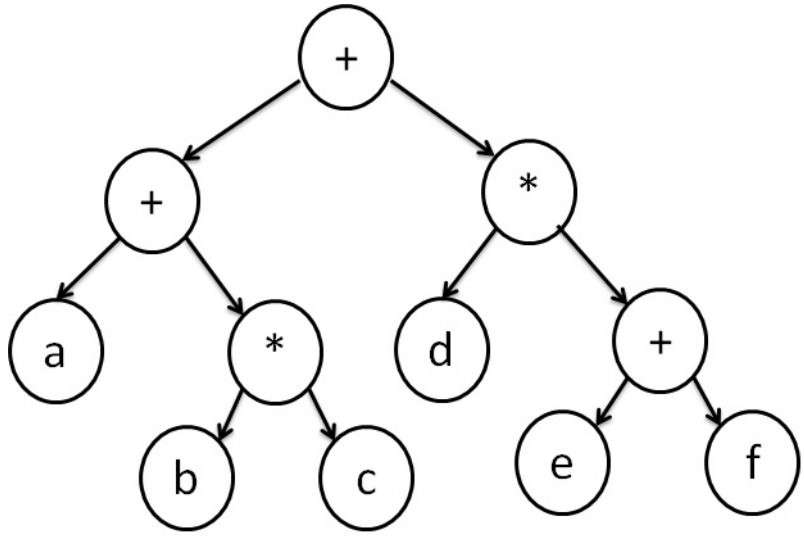
```
[10,19,14,31,42,35,27]
```

14.7 EXPRESSION TREES

Expression Tree is used to represent expressions.

An expression and expression tree shown below

```
a + (b * c) + d * (e + f)
```



Expression Tree

All the below are also expressions. Expressions may includes constants value as well as variables

```
a * 6

 16

(a^2)+(b^2)+(2 * a * b)

 (a/b) + (c)

m * (c ^ 2)
```

It is quite common to use parenthesis in order to ensure correct evaluation of expression as shown above

Construction of Expression Tree

Let us consider a **postfix expression** is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and push a pointer onto a stack
4. If the symbol is an operator, pop two pointers from the stack namely T_1 & T_2 and form a new tree with root as the operator, T_1 & T_2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

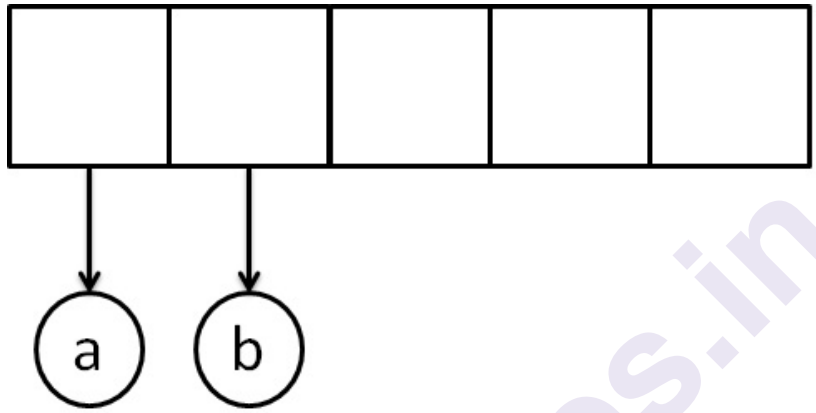
Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

Example - Postfix Expression Construction

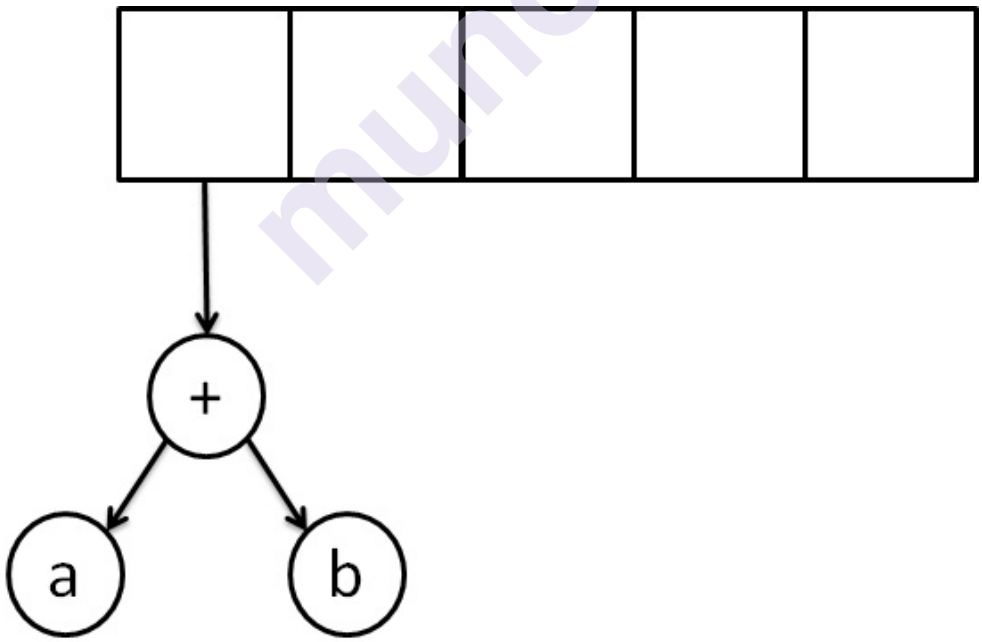
The input is:

a b + c *

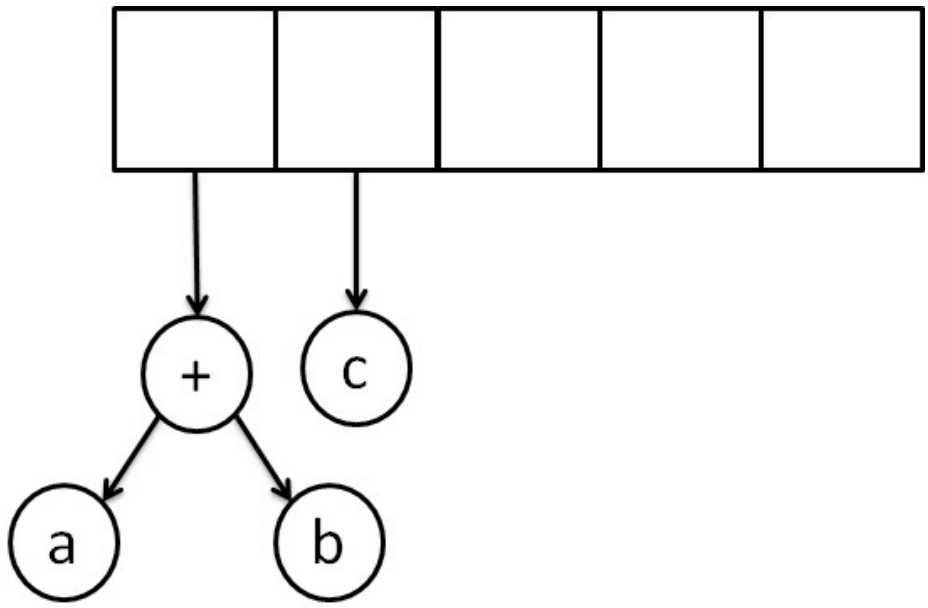
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



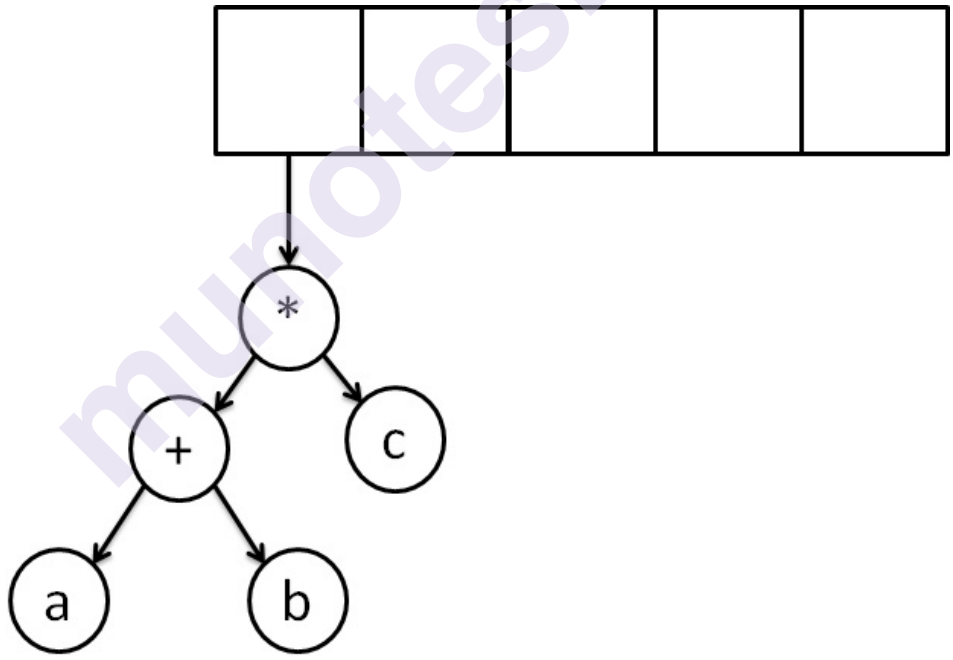
Next, read a'+' symbol, so two pointers to tree are popped,a new tree is formed and push a pointer to it onto the stack.



Next, **'c'** is read, we create one node tree and push a pointer to it onto the stack.



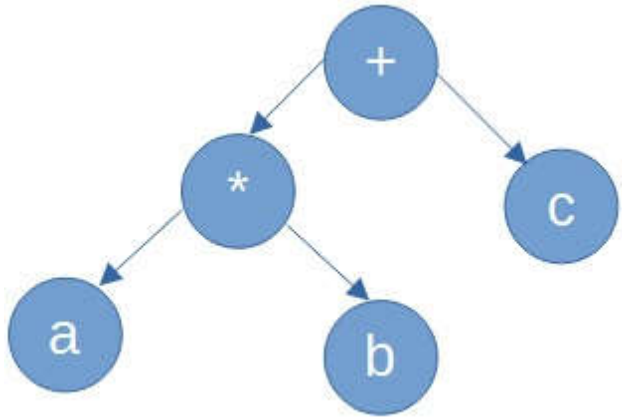
Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.



Examples

Expression Tree is used to represent expressions. Let us look at some examples of prefix, infix and postfix expressions from expression tree for 3 of the expresssions:

- a*b+c
- a+b*c+d
- a+b-c*d+e*f



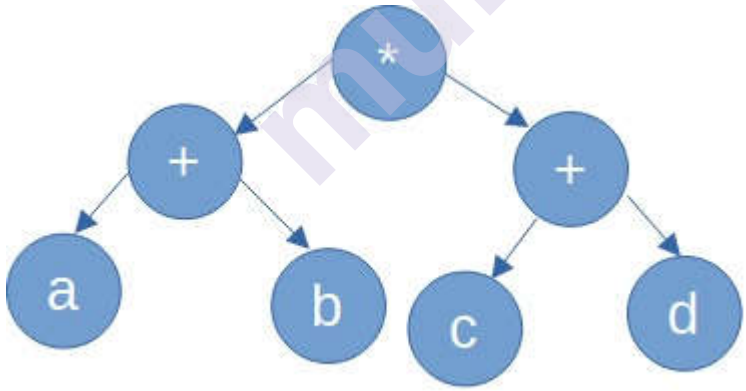
Expression Tree for a*b+c

Expressions from Expression Tree

Infix expression	a * b + c
Prefix expression	+ * a b c
Postfix expression	a b * c +

Infix, Prefix and Postfix Expressions from Expression Tree for a+b*c+d

Expression Tree for a + b * c + d can be represented as:



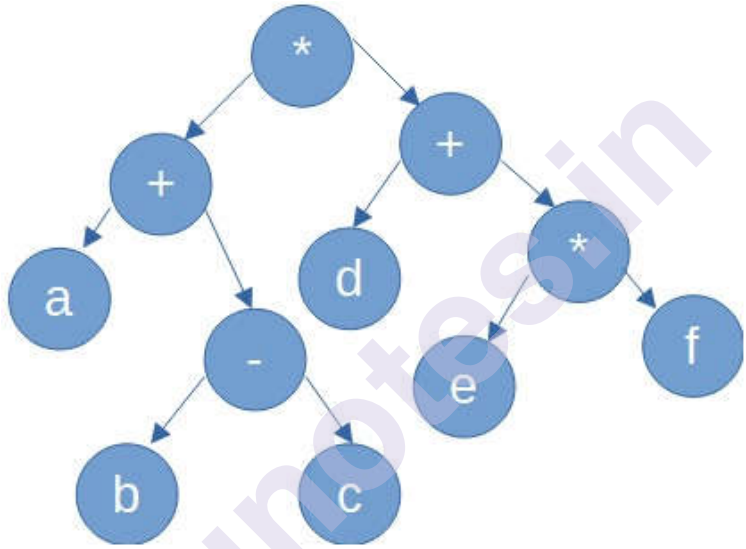
Expression Tree for a + b * c + d

Expressions for the binary expression tree above can be written as

Infix expression	a + b * c + d
Prefix expression	* + a b + c d
Postfix expression	a b + c d + *

Infix, Prefix and Postfix Expressions from Expression Tree for a+b-c*d+e*f

Expression Tree for a + b - c * d + e * f can be represented as:



Expression Tree for a+b-c*d+e*f

Expressions for the binary expression tree above can be written as

Infix expression	a + b - c *d + e * f
Prefix expression	* + a - b c + d * e f
Postfix expression	a b c - + d e f * + *

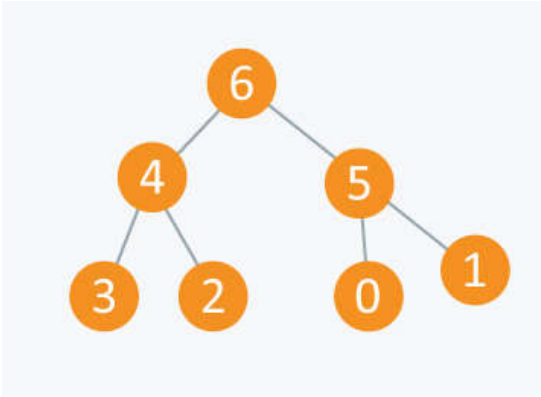
14.8 HEAPS AND HEAPSORT

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, if X is the parent node of Y, then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.

The maximum number of children of a node in a heap depends on the type of heap. However, in the more commonly-used heap type, there are at most 2 children of a node and it's known as a Binary heap.

In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is $\log_2 N$.



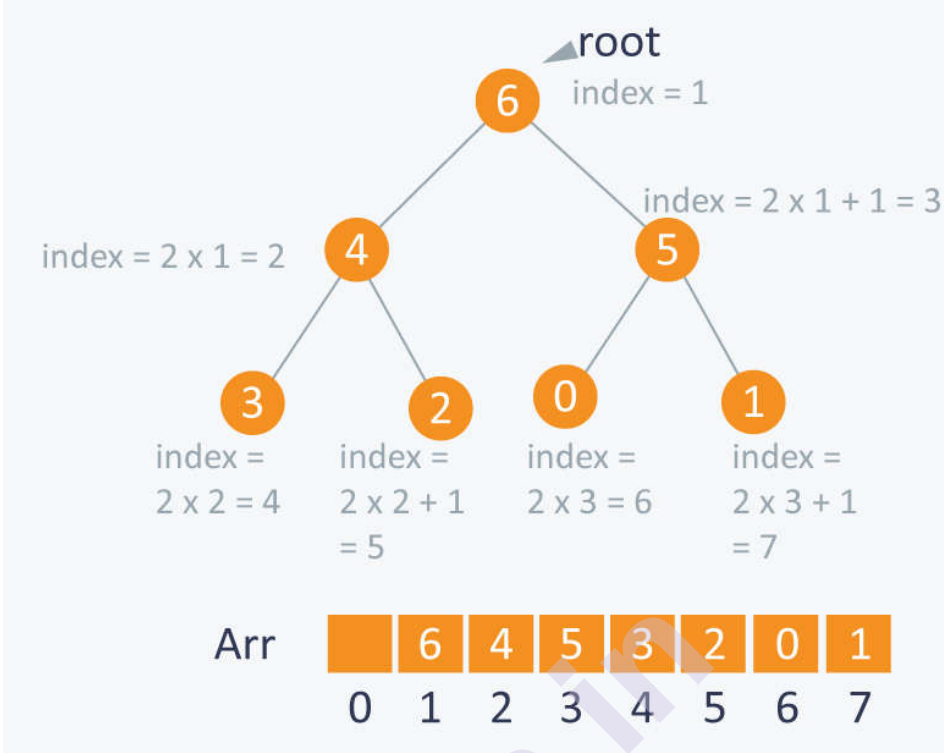
In the diagram above, you can observe a particular sequence, i.e each node has greater value than any of its children.

Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant, we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority.

So at each instant we have to check for the job with maximum priority to complete it and also insert if there is a new job. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.

As you can see in the diagram below, we can use an array to store the nodes of the tree. Let's say we have 7 elements with values {6, 4, 5, 3, 2, 0, 1}.

Note: An array can be used to simulate a tree in the following way. If we are storing one element at index i in array Arr, then its parent will be stored at index i/2 (unless its a root, as root has no parent) and can be accessed by Arr[i/2], and its left child can be accessed by Arr[2□i] and its right child can be accessed by Arr[2□i+1]. Index of root will be 1 in an array.



There can be two types of heap:

1. **Max Heap:** In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Implementation:

Let's assume that we have a heap having some elements which are stored in array Arr. The way to convert this array into a heap structure is the following. We pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps, in themselves and the node itself is a max heap (it's value should be greater than all the child nodes)

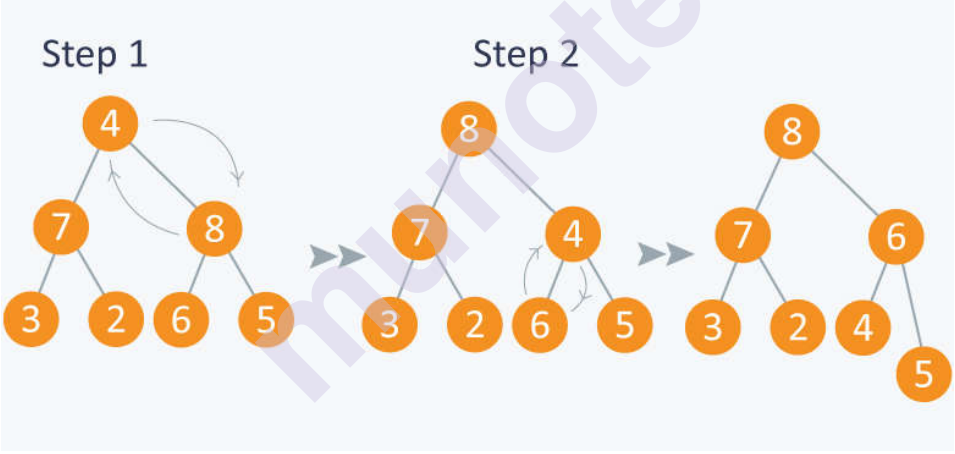
To do this we will implement a function that can maintain the property of max heap (i.e each element value should be greater than or equal to any of its child and smaller than or equal to its parent)

```
voidmax_heapify(intArr[],inti,int N)
{
int left =2*i//left child
int right =2*i+1//right child
if(left<= N andArr[left]>Arr[i])
largest= left;
else
```

```
largest=i;
if(right <= N andArr[right]>Arr[largest])
largest= right;
if(largest !=i)
{
swap(Arr[i],Arr[largest]);
max_heapify(Arr,largest,N);
}
}
```

Complexity: O(logN)

Example:
In the diagram below,initially 1st node (root node) is violating property of max-heap as it has smaller value than its children, so we are performing max_heapify function on this node having value 4.



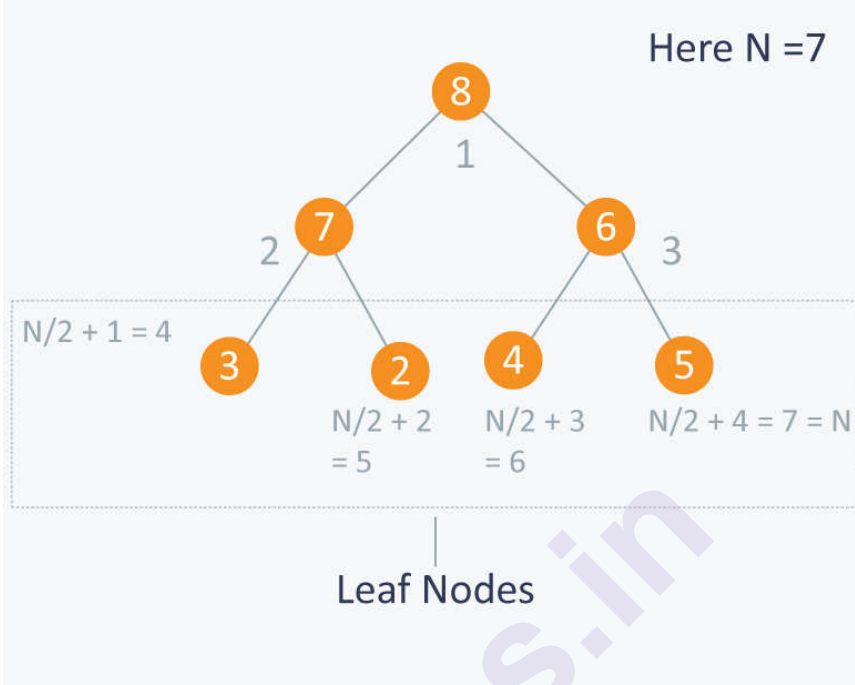
As 8 is greater than 4, so 8 is swapped with 4 and max_heapify is performed again on 4, but on different position. Now in step 2, 6 is greater than 4, so 4 is swapped with 6 and we will get a max heap, as now 4 is a leaf node, so further call to max_heapify will not create any effect on heap.

Now as we can see that we can maintain max- heap by using **max_heapify** function.

Before moving ahead, lets observe a property which states: A N element heap stored in an array has leaves indexed by N/2+1, N/2+2 , N/2+3 …. upto N.

Let's observe this with an example:

Lets take above example of 7 elements having values {8, 7, 6, 3, 2, 4, 5}.



So you can see that elements 3, 2, 4, 5 are indexed by N/2+1 (i.e 4), N/2+2 (i.e5) and N/2+3 (i.e 6) and N/2+4 (i.e 7) respectively.

Building MAX HEAP:

Now let's say we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So we can use max-heapify function to make a max heap out of the array.

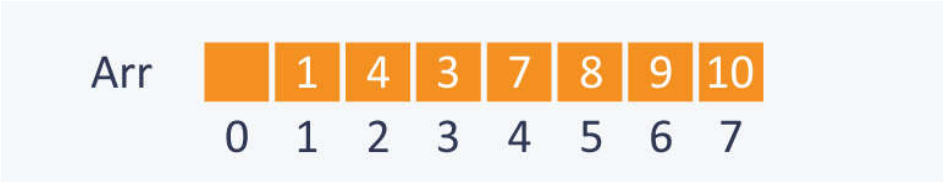
How?

From the above property we observed that elements from Arr [N/2+1] to Arr[N] are leaf nodes, and each node is a 1 element heap. We can use max_heapify function in a bottom up manner on remaining nodes, so that we can cover each node of tree.

```
voidbuild_maxheap(intArr[])
{
for(inti= N/2;i>=1;i--)
{
max_heapify(Arr,i);
}
}
```

Complexity: O(N). **max_heapify** function has complexity logN and the **build_maxheap** functions runs only N/2 times, but the amortized complexity for this function is actually linear.

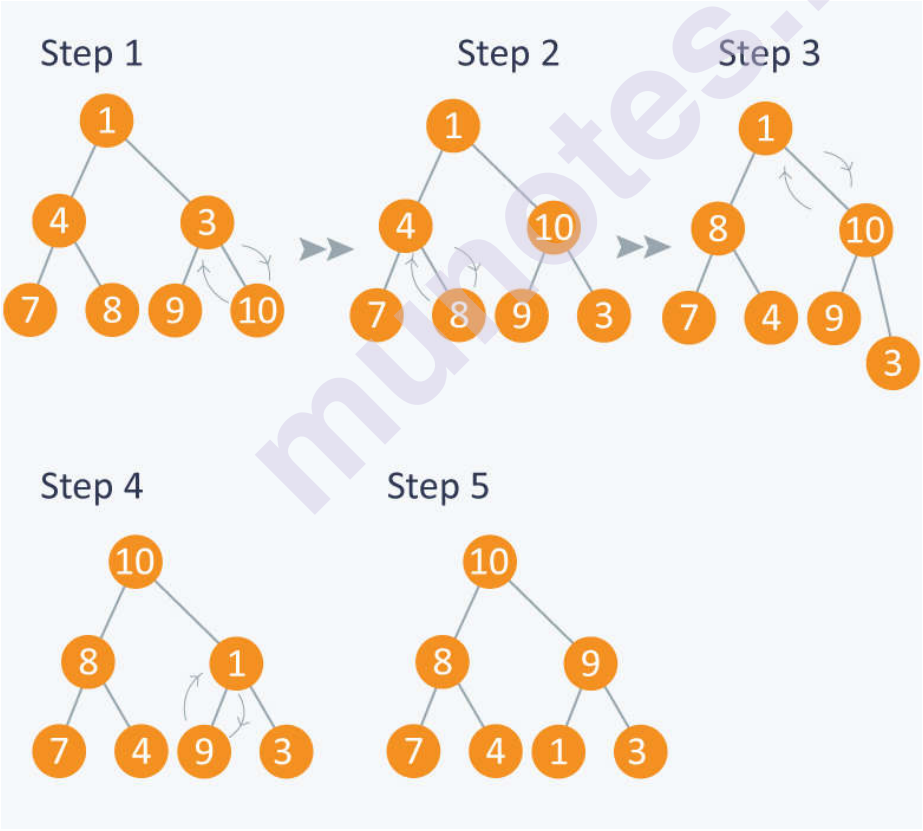
Example:
Suppose you have 7 elements stored in array Arr.



Here N=7, so starting from node having index N/2=3, (also having value 3 in the above diagram), we will call max_heapify from index N/2 to 1.

In the diagram below:

In step 1, in max_heapify(Arr, 3), as 10 is greater than 3, 3 and 10 are swapped and further call to max_heap(Arr, 7) will have no effect as 3 is a leaf node now.

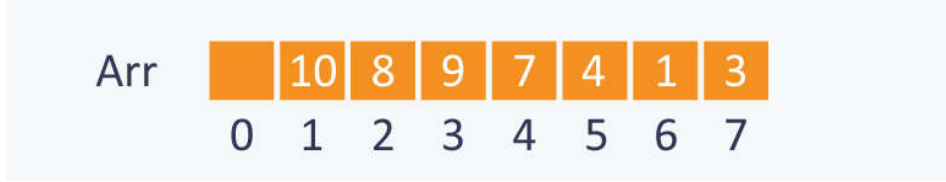


In step 2, calling max_heapify(Arr, 2) , (node indexed with 2 has value 4) , 4 is swapped with 8 and further call to max_heap(Arr, 5) will have no effect, as 4 is a leaf node now.

In step 3, calling max_heapify(Arr, 1) , (node indexed with 1 has value 1), 1 is swapped with 10 .

Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max_heapify(Arr, 3) will be performed , and 1 will be swapped with 9. Now further call to max_heapify(Arr, 7) will have no effect, as 1 is a leaf node now.

In step 5, we finally get a max- heap and the elements in the array Arr will be :



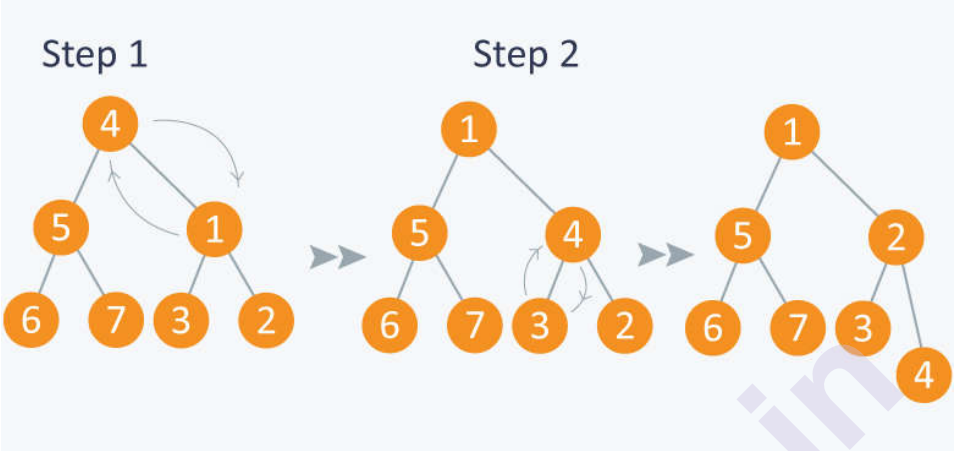
2. **Min Heap:** In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.

As you can see in the above diagram, each node has a value smaller than the value of their children.
We can perform same operations as performed in building max_heap. First we will make function which can maintain the min heap property, if some element is violating it.

```
voidmin_heapify(intArr[],inti,int N)
{
int left  =2*i;
int right =2*i+1;
int smallest;
if(left <= N andArr[left]<Arr[i])
smallest= left;
else
smallest=i;
if(right <= N andArr[right]<Arr[smallest])
smallest= right;
if(smallest !=i)
{
swap(Arr[i],Arr[ smallest ]);
min_heapify(Arr,smallest,N);
}
}
```

Complexity: O(logN) .

Example:
Suppose you have elements stored in array Arr {4, 5, 1, 6, 7, 3, 2}. As you can see in the diagram below, the element at index 1 is violating the property of min -heap, so performing min_heapify(Arr, 1) will maintain the min-heap.

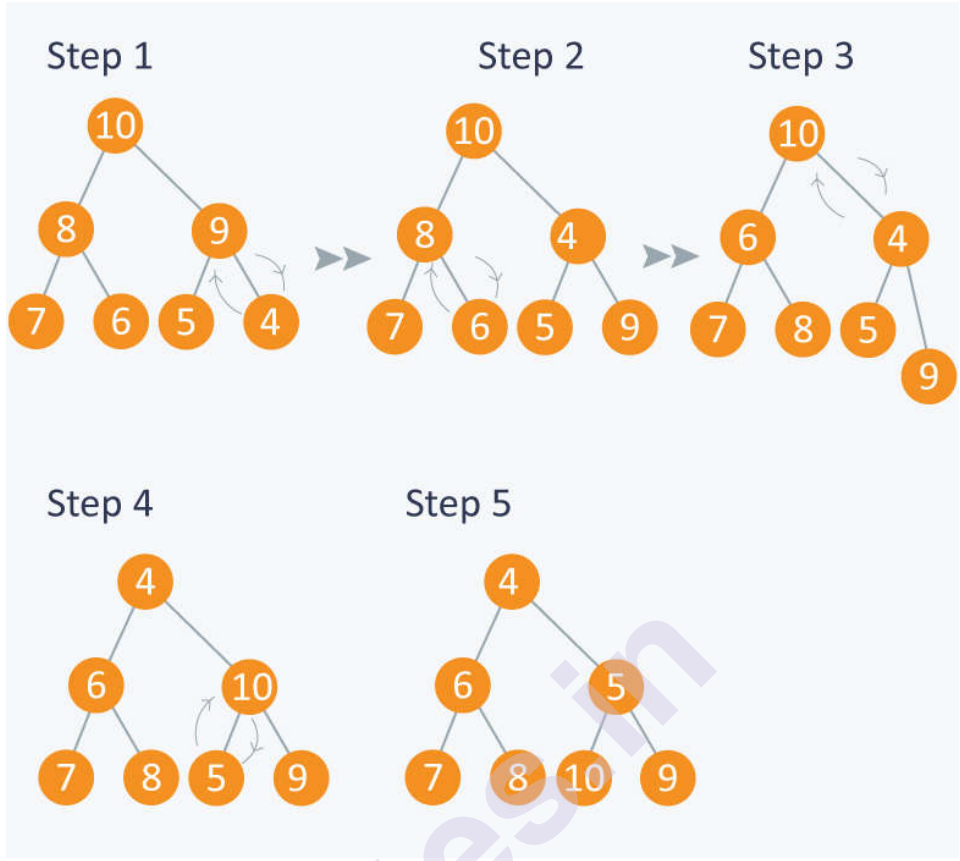


Now let's use above function in building min-heap. We will run the above function on remaining nodes other than leaves as leaf nodes are 1 element heap.

```
voidbuild_minheap(intArr[])

{

for(inti= N/2;i>=1;i--)

min_heapify(Arr,i);

}
```

Complexity: O(N). The complexity calculation is similar to that of building max heap.

Example:
Consider elements in array {10, 8, 9, 7, 6, 5, 4} . We will run min_heapify on nodes indexed from N/2 to 1. Here node indexed at N/2 has value 9. And at last, we will get a min_heap.



Heaps can be considered as partially ordered tree, as you can see in the above examples that the nodes of tree do not follow any order with their siblings(nodes on the same level). They can be mainly used when we give more priority to smallest or the largest node in the tree as we can extract these node very efficiently using heaps.

Heap Sort:

We can use heaps in sorting the elements in a specific order in efficient time.
Let's say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.

Idea: We build the max heap of elements stored in Arr, and the maximum element of Arr will always be at the root of the heap.

Leveraging this idea we can sort an array in the following manner.

Processing:

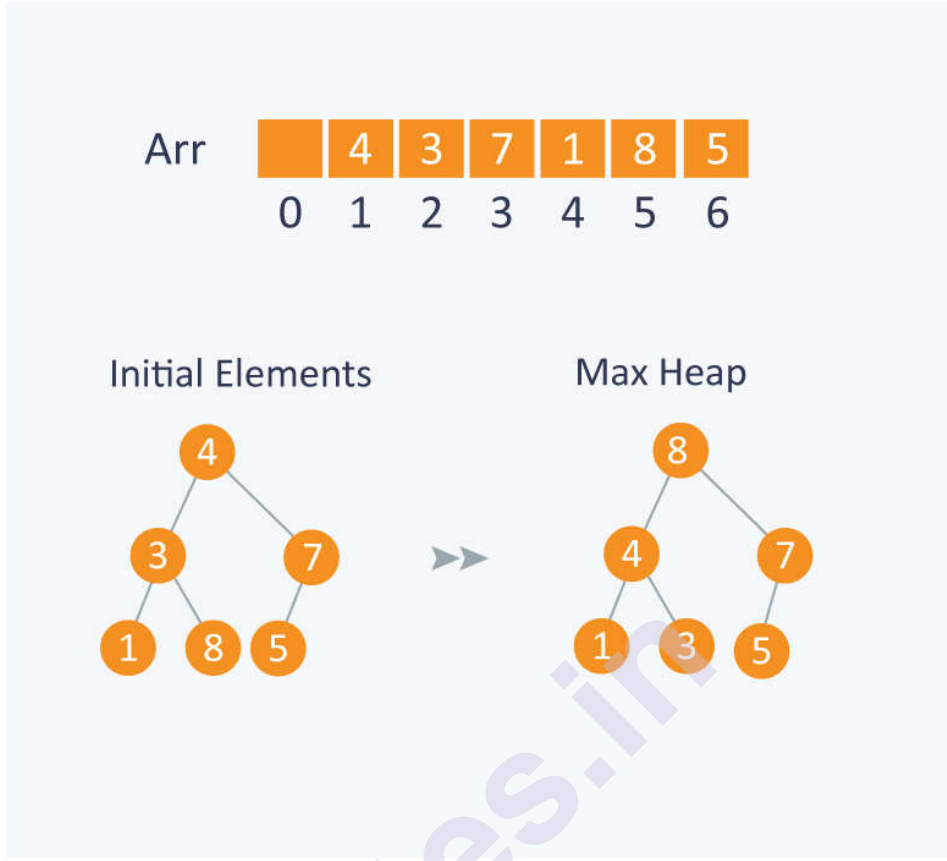
- Initially we will build a max heap of elements in Arr.
- Now the root element that is Arr[1] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will decrease the length of heap by one.

- We will repeat the step 2, until we get all the elements in their correct position.
- We will get a sorted array.
- **Implementation:**
- Suppose there are N elements stored in array Arr.

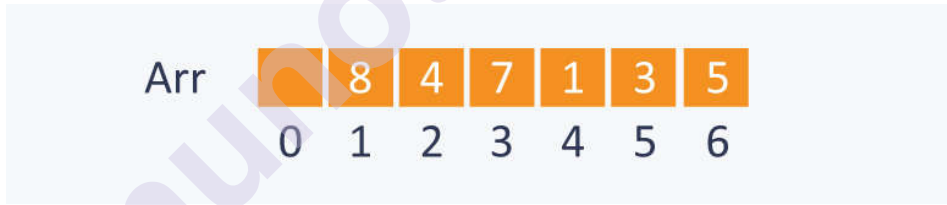
```
voidheap_sort(intArr[])
{
intheap_size= N;
build_maxheap(Arr);
for(inti= N;i>=2;i--)
{
        swap(Arr[1],Arr[i]);
heap_size= heap_size-1;
max_heapify(Arr,1,heap_size);
}
}
```

Complexity: As we know max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify N−1 times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).

Example:
In the diagram below,initially there is an unsorted array Arr having 6 elements. We begin by building max-heap.

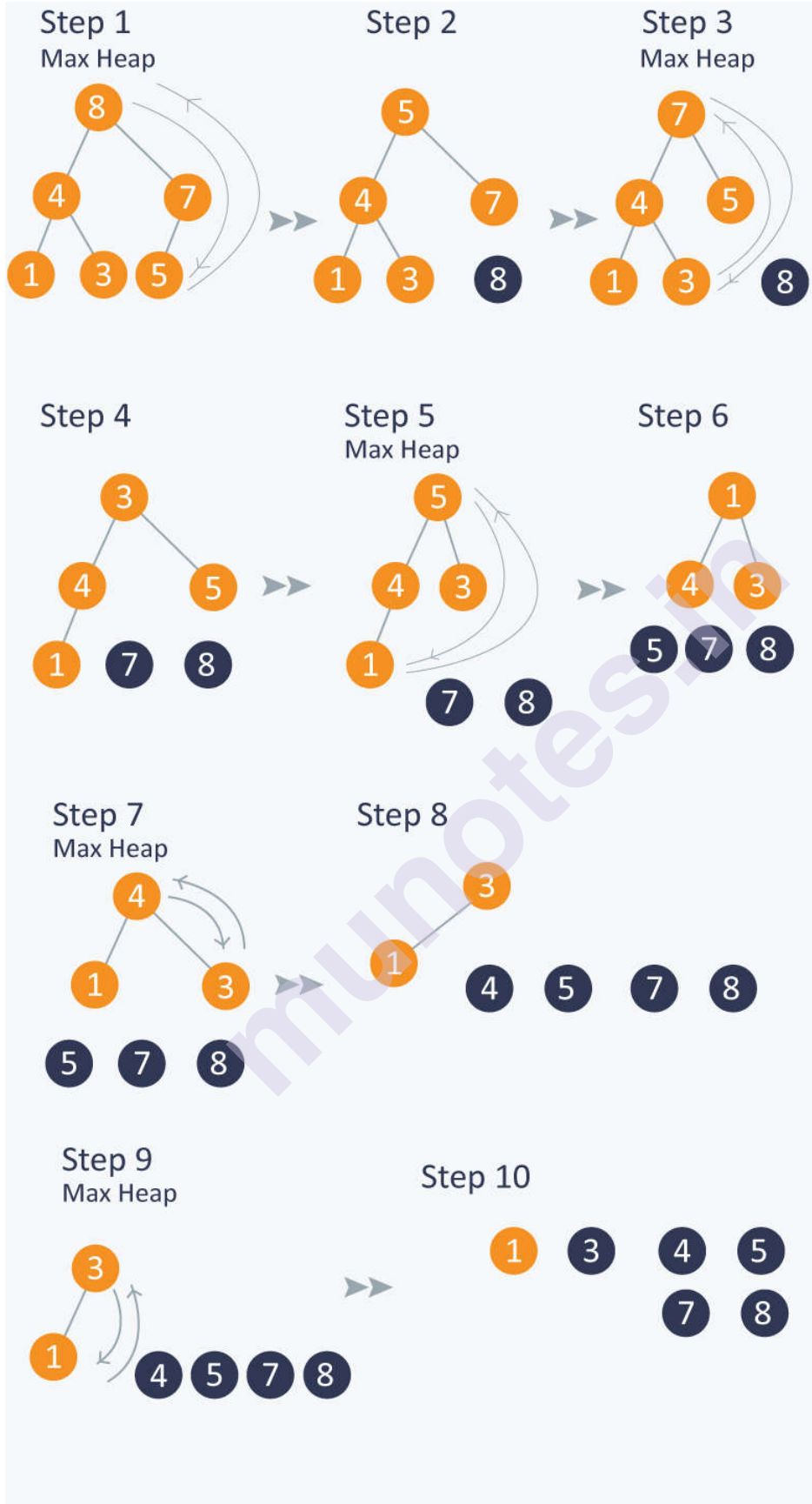


After building max-heap, the elements in the array Arr will be:

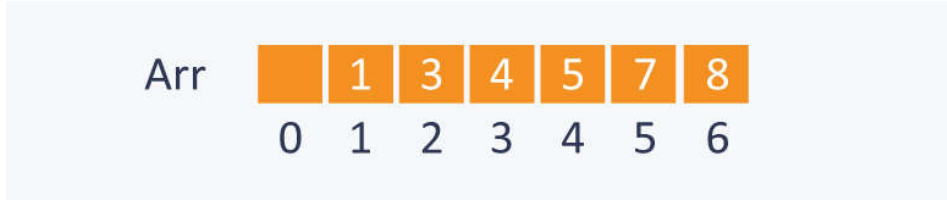


Processing:

Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.


Step 1
Max Heap
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

After all the steps, we will get a sorted array.



14.9 SEARCH TREES

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

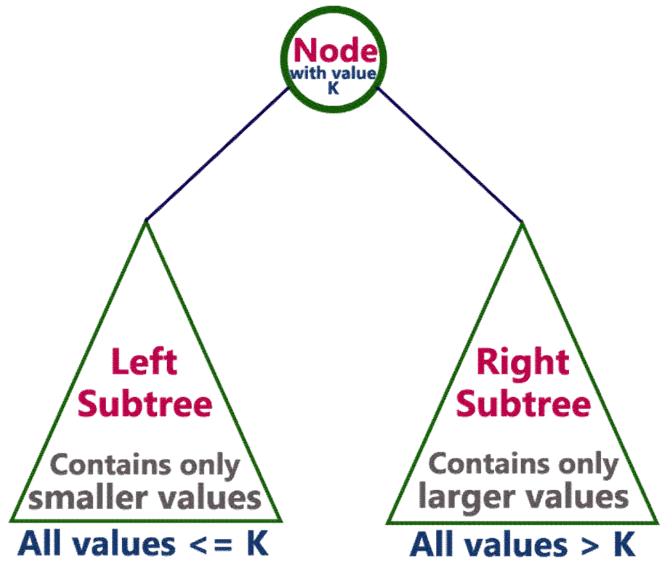
A binary tree has the following time complexities...

1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**
3. **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

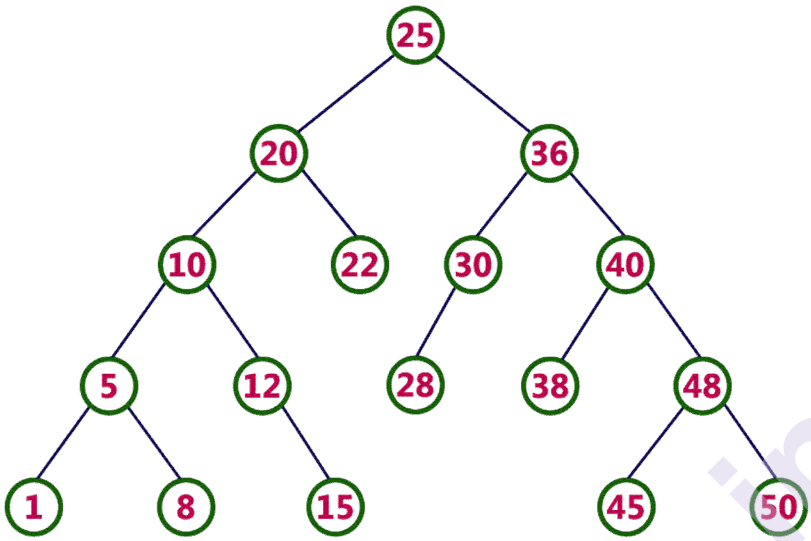
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.

- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6-** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node
- **Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1 -** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2 -** Check whether tree is Empty.
- **Step 3 -** If the tree is **Empty**, then set **root** to **newNode**.
- **Step 4 -** If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5 -** If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.
- **Step 6-** Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).
- **Step 7 -** After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: **Deleting a leaf node**

We use the following steps to delete a leaf node from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: **Deleting a node with one child**

We use the following steps to delete a node with one child from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

Case 3: **Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

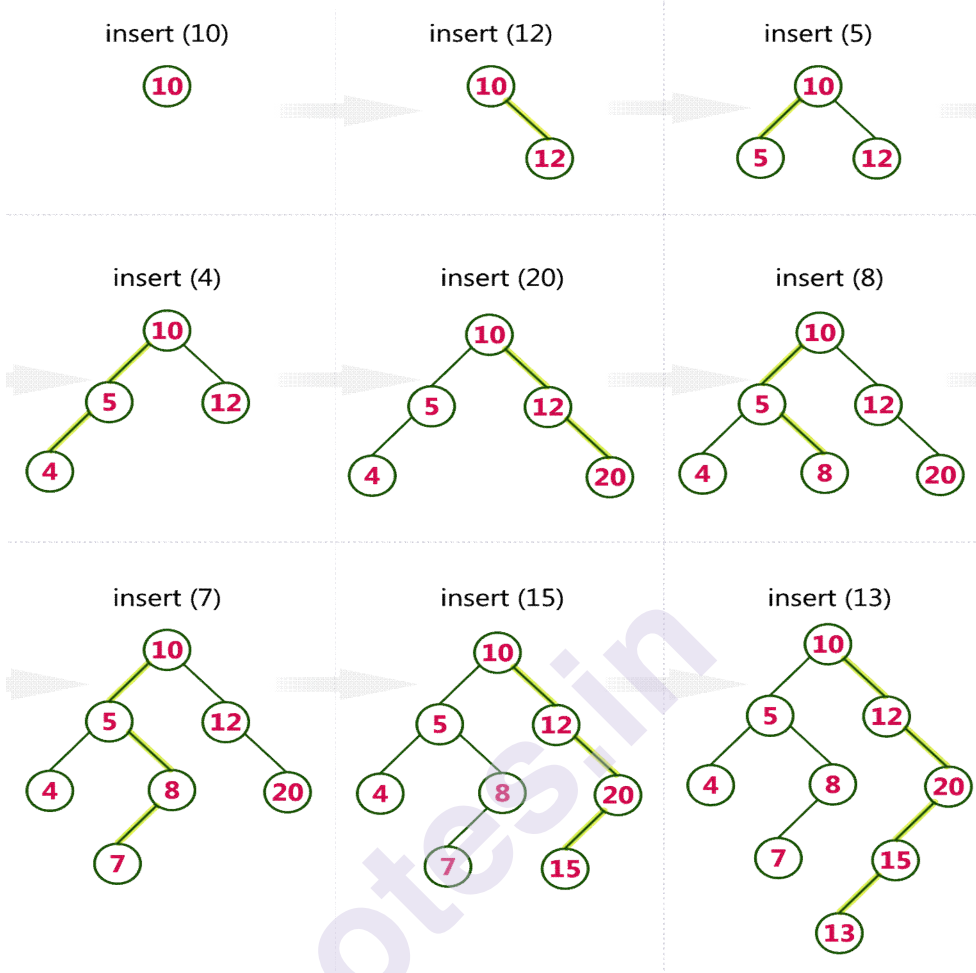
- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...




```
classNode:
def __init__(self, data):
self.left=None
self.right=None
self.data= data
# Insert method to create nodes
def insert(self, data):
ifself.data:
if data <self.data:
ifself.leftisNone:
self.left=Node(data)
else:
self.left.insert(data)
```

```
else data >self.data:
ifself.rightisNone:
self.right=Node(data)
else:
self.right.insert(data)
else:
self.data= data
# findval method to compare the value with nodes
deffindval(self,lkpval):
iflkpval<self.data:
ifself.leftisNone:
returnstr(lkpval)+" Not Found"
returnself.left.findval(lkpval)
elseiflkpval>self.data:
ifself.rightisNone:
returnstr(lkpval)+" Not Found"
returnself.right.findval(lkpval)
else:
print(str(self.data)+' is found')
# Print the tree
defPrintTree(self):
ifself.left:
self.left.PrintTree()
print(self.data),
ifself.right:
self.right.PrintTree()
root=Node(12)
root.insert(6)
root.insert(14)
```

```
root.insert(3)
print(root.findval(7))
print(root.findval(14))
```

Output

When the above code is executed, it produces the following result –

```
7 Not Found
14 is found
```

14.10 SUMMARY

- A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges.
- A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.
- **Full Binary Tree** is a special kind of a binary tree that has either zero children or two children.
- A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree.
- A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree.
- A binary tree is said to be 'balanced' if the tree height is O(logN), where 'N' is the number of nodes
- A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child.
- Tree traversal means **traversing** or **visiting** each node of a tree.
- An inorder traversal is a traversal technique that follows the policy, i.e., **Left Root Right**.
- A preorder traversal is a traversal technique that follows the policy, i.e., **Root Left Right**.
- A Postorder traversal is a traversal technique that follows the policy, i.e., **Left Right Root**.

- Expression Tree is used to represent expressions.
- A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.
- In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

14.11 QUESTIONS

1. Explain all tree terminology.
2. What are the types of tree?
3. Write a short note on AVL tree.
4. What is Binary Tree? And How it is different than general tree?
5. What are the types of Binary Tree?
6. Write a short note on Tree Traversal.
7. What do you mean by Expression Tree?
8. Write a short note on Heap and its types.
9. How Heap Sort Works?
10. Write a short note on Binary Search Tree.

14.12 REFERENCE FOR FURTHER READING

- https://www.mygreatlearning.com/blog/understanding-trees-in-data-structures/
- https://towardsdatascience.com/8-useful-tree-data-structures-worth-knowing-8532c7231e8c
- https://www.upgrad.com/blog/5-types-of-binary-tree/
- https://www.krivalar.com/data-structures-expression-tree
- https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/
- http://www.btechsmartclass.com/data_structures/binary-search-tree.html