1

INTRODUCTION TO ALGORITHM

Unit Structure:

1.1 OBJECTIVE

After going through this unit, you will be able to:

Understand the fundamental concepts of algorithm design and why algorithm analysis is necessary. How to Compare Algorithms, Rates of Growth and Their Varieties, Analysis Types, and Asymptotic Notation

1.2 INTRODUCTION TO ALGORITHM

A well-defined computing technique that accepts a value or set of values as input and outputs a value or set of values is referred to informally as an algorithm. As a result, an algorithm is a series of computations that converts an input into an output. A tool for resolving a well-defined computational problem is another way to look at an algorithm. The required input/output connection is briefly described in the issue

statement. In order to achieve that input/output connection, the algorithm specifies a particular computing method. For instance, it can be necessary to organise a list of integers in non-decreasing order. Numerous common design strategies and analytical tools may be included into this challenge because it regularly occurs in real-world settings.

What Makes a Good Algorithm?

- It is important to specify input and output clearly.
- The algorithm's steps should all be distinct and understandable.
- When compared to other possible solutions, algorithms should be the most efficient.
- There should be no computer code in an algorithm. It is preferable to write the method in a way that it may be applied to several programming languages.
- Algorithms operate on the basis of input and output. They start with an input of some value, follow the precise instructions, and end up with an output.

Characteristics of an algorithm:

1. Input: External input should be given in amounts of zero or more.

2. Output: Must generate at least one outcome.

3. Clearness: The actions to complete the task should be explicit and clear.

4. Finiteness: Must come to an end after a set number of steps.

5. Effectiveness: Must yield the intended outcome.

Example 1

Algorithm for adding two integers the user providedentered by the user.

Step 1: Start

Step 2: Declare three variables n1, n2 and res.

Step 3: Read the value of variables n1 and n2.

Step 4: Add the value of variablesn1 and n2 and assign the result to res.

res=n1+n2

Step 5: Display res

Step 6: Stop

Example 2

Algorithm for subtraction two integers the user provided.

Step 1: Start

Step 2: Declare three variables n1, n2 and res.

Step 3: Read the value of variables n1 and n2.

Step 4: Subtract the value of variables n1 and n2 and assign the result to res.

res=n1-n2

Step 5: Display res

Step 6: Stop

Example 3

Algorithm for calculating factorial of an integer the user provided.

Step 1: Start

Step 2: Declare two variables n andfact.

Step 3: Read the value of variable n.

Step 4: Variable fact is set as 1

Step 5: *fact <= fact * n*

Step 6: Decrease n

Step 7: Verify if n is equal to 0

Step 8: If n is equal to zero, goto step 10 (break out of loop)

Step 9: Else goto step 5

Step 10: Display fact

Example 4

Algorithm for calculating square of a number the user provided.

Step 1: Start

Step 2: Declare two variables n and res.

Step 3: Read the value of variable n.

Step 4*: res=n*n*

Step 5: Display res

Step 6: Stop

1.3 WHY TO ANALYSIS ALGORITHM

An essential component of computational complexity theory is algorithm analysis, which offers a theoretical estimate of the resources needed by an algorithm to solve a particular computing issue. Calculating how much time and space are needed to execute an algorithm is called algorithm analysis. As a result, the analysis can only be considered approximate. In addition, by examining many algorithms, we may contrast them and choose the most effective one for our needs. The technique of determining an algorithm's computational complexity is known as algorithm analysis. The time, space, and any other resources required to execute (run) the algorithm are all referred to as the method's computational complexity.

Comparing several algorithms that are applied to the same issue is the aim of algorithm analysis. This is performed to measure which approach uses less memory and resolves a certain problem more rapidly.

Algorithm Analysis Types:

1. Best case
2. Worst case
3. Average case

Best case: Best case: Specify the input for the method that requires the minimum amount of time. Calculate an algorithm's lower bound in the best case scenario. The lower bound on the algorithm's running time for every given input is provided by the best case analysis of algorithms. Simply stated, it says that every software will require at least (more than or equal to) that amount of time to do its task.

Example-The perfect scenario occurs in a linear search when the search data is present at the initial place of large data.

Worst Case: Specify the input for the algorithm to use if you want it to run quickly. Calculate an algorithm's upper bound as worst case scenario. The upper bound on the algorithm's running time for any given input is provided by the worst-case analysis of algorithms. In other words, it says that any programme will run in a maximum amount of time (less than or equal to).

Example-The worst case scenario happens in a linear search when there is no search data at all.

Average case: Consider all random inputs and figure out how long it will take to process each one on average. The total inputs are then divided by this number. The average case analysis of algorithms, as the name implies, adds up the running times on all potential inputs before taking the average. Therefore, in this instance, the algorithm's execution time serves as both the lower and upper bounds. In other words, we can determine the algorithm's typical running time through it.

Example:

Find the ace of spades from a deck of 52 cards.

One card at a time is flipped until an ace of spades is revealed.

Best case scenario: Everything goes as smoothly as it possibly can, we complete the task speedily, and we call it a day.

You turned over an ace of spades as your opening card.

Worst case scenario: Everything is getting completely out of control, so you must put in the most effort to complete your mission.

The final card you flipped was an ace of spades.

Average case scenario: Neither very poorly nor really well, things go.

The ace of spades is not the first or last card that you see.

1.4 RUNNING TIME ANALYSIS

When algorithms are analysed, their computational complexity—the amount of time, storage, and/or other resources required to run them—is determined. To put it simply, to gauge how effective a particular algorithm is.

In order to conduct performance study, we typically applied to calculate and evaluate the worst-case theoretical running times complexity of algorithms.

O(1), also known as Constant Running Time, is the quickest possible running time for any algorithm. In this example, the algorithm runs at the same speed regardless of the size of the input. An algorithm should have this runtime, although it's rarely possible.

In fact, the quantity of the input or amount of operations needed for every entry element, or n, determines an algorithm's runtime (performance).

According to running time complexity, the algorithms can be categorised as follows in order of best to worst performance where c is a positive constant and n is the size of the input:

Algorithm category	Explanation
logarithmic *O(logn)*	Running time increases in logarithmic proportion to n.
linear *O(n)*	Running time increases directly in proportion to n.
superlinear *O(nlogn)*	Running time increases in proportion to n.

Polynomial $O(n^c)$	Running time increases faster than previous iterations based on n.
exponential $O(c^n)$	Running time becomes even faster than using the polynomial approach based on n.
factorial $O(n!)$	Running time increases the fastest and quickly becomes unsuitable for even tiny values of n.

1.5 HOW TO COMPARE ALGORITHMS

In order to analyse the algorithm, we need to know two fundamental parameters:

- Space Complexity: The amount of space needed for an algorithm to operate completely might be thought of as the space complexity.

An algorithm's space complexity is a measure of how much memory it uses over its existence.

An algorithm's space requirements are equal to the sum of the two factors listed below.

1. A fixed component is a place needed to hold constants, simple variables, programme sizes, and other variables that are independent of the size of the problem.
2. A variable component is the area needed by variables, whose size is completely based on the scope of the issue. Stack space for recursions, dynamic memory allocation, etc.

Any algorithm's S(p) space complexity is defined as S(p) = A + Sp (I) Where S(I), which depends on instance characteristic I, is viewed as the variable component of the algorithm while A is treated as the fixed part of the algorithm.

- Time Complexity: Time complexity, which measures how long an algorithm takes to run completely, is a function of input size n.

The measure of how long an algorithm will take to complete execution is called the time complexity of the algorithm. Given that each step requires a fixed amount of time, time requirements can be expressed or defined as a numerical function t(N), where t(N) can be expressed as the number of steps.

For instance, N steps are required to add two n-bit integers. As a result, the total calculation time is given by t(N) = c*n, where c is the amount of time needed to add two bits. Here, we observe that as input size increases, t(N) climbs linearly..

1.6 RATE OF GROWTH

An algorithm's growth rate is the rate at which the cost of the algorithm increases as the size of its input increases. The term "Rate of Growth" describes how quickly running time grows in response to input.

Let's say you visited a store to purchase a car and a bicycle. If a friend sees you there and inquires about your purchase, we typically reply that you are purchasing a car. Because the price of a car is excessively high compared to the price of a bicycle (approximating the cost of cycle to the cost of a car).

Overall Cost = Car Cost + Bicycle Cost

Overall Cost ≈ Car Cost (approximation)

For the abovementioned example, we can describe the cost of the car and the cost of the bike in terms of function, ignoring the low order variables that are generally irrelevant (for large values of input size, n).

As an example in the below case, n^3, $2n^2$ and 10n are the individual costs of some function and approximate it to n^3. Since, n^3 is the highest rate of growth.

$n^3 + 2n^2 + 10n \approx n^3$

Understanding growth rates is the key to algorithm analysis. How much more resources will my algorithm need as the volume of data increases, in other words? Usually, we use a function to express the rate of resource expansion of a piece of code. This section will look at graphs for various growth rates, ranging from the most efficient to the least efficient, to assist comprehend the implications.

- **Constant Growth Rate**

Where the requirement for resources does not increase, it is said to be constant. To put it another way, processing 1 piece of data uses the same resources as processing 1 million pieces of data. A horizontal line appears on the graph with such a growth rate.

- **Logarithmic Growth Rate**

When the data is doubled, the resource requirements increase by one unit, which is known as a logarithmic growth rate. This basically indicates that the growth rate's curve flattens as big data increases (closer to horizontal but never reaching it). What a curve of this kind would look like is depicted in the following graph.

- **Linear Growth Rate**

A growth rate that is linear is one in which the demand for resources and the volume of data are precisely proportionate to one another. That is, the growth rate can be represented by a straight, non-horizontal line.

- **Log Linear**

A line with a small curve represents a log linear growth rate. Lower numbers have a more prominent curve than higher ones.

- **Quadratic Growth Rate**

One who can describe a parabola as having a quadratic growth rate.

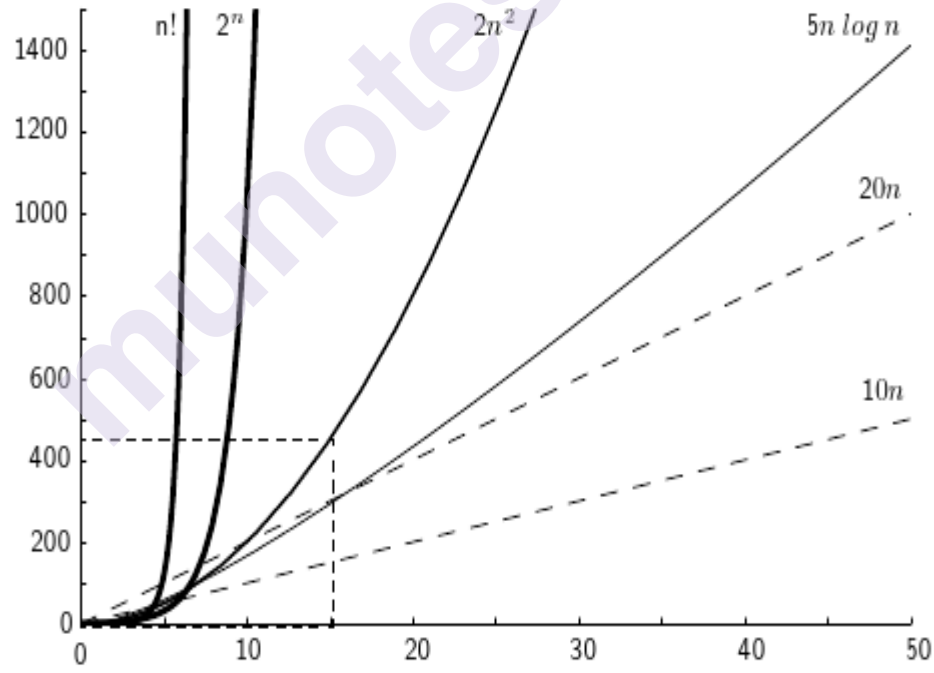
- **Cubic Growth Rate**

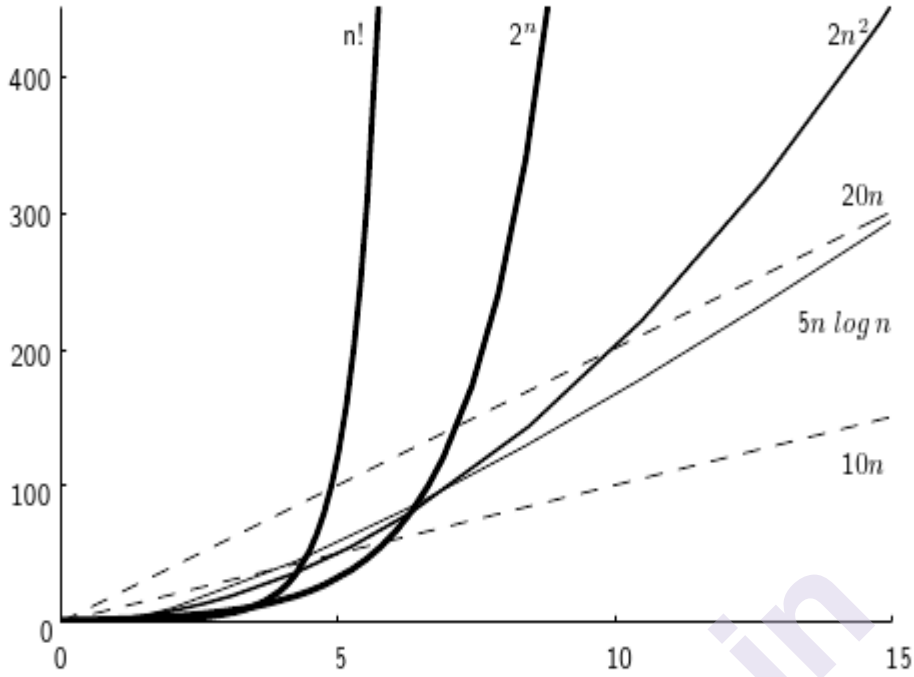
Although this may resemble the quadratic curve very much, it develops much more quickly.

- **Exponential Growth Rate**

Each additional unit of data demands a doubling of resources, which is known as an exponential growth rate. As you can see, the growth rate initially appears to be flat before rapidly rising to almost vertical (keep in mind that it cannot actually be vertical).

Following graph showing the rates of growth for six equations with n input values





Source: https://opendsa-server.cs.vt.edu/OpenDSA/Books/ CS4104/html/_ images/plot.png

1.7 COMMONLY USED RATES OF GROWTH

Time Complexity	Name	Example
1	Constant	A linked list's first element being added.
log n	Logarithmic	In a sorted array, locating a specific element.
n	Linear	Locating a specific element in an unsorted array.
n log n	Linear Logarithmic	Separating n things according to "Divide and Conquer."
n^2	Quadratic	The graph's shortest route connecting any two nodes
n^3	Cubic	Multiplication of a matrix.
2^n	Exponential	The challenge with the Towers of Hanoi.

Examples

Sequential Statements

If our sentences contain fundamental operations like comparisons, assignments, and variable reads. We can assume they take the same amount of time O(1).

Example where we can calculate the square sum of two values.

```
n1 = a * a;

n2 = b * b;

res = n1 + n2;
```

Constant time is spent on each line O(1).

1.8 TYPES OF ANALYSIS

It is a method for illustrating limiting behaviour. The approach is applicable to all fields of science. It can be used to evaluate how well an algorithm performs on a significant amount of data.

When analysing algorithms in computer science, it is important to take into account how well they function when used with very big input datasets.

The fastest and slowest potential execution times for an algorithm are expressed using asymptotic notation. These are also known as "best case" and "worst case," respectively.

We calculate the complexity based on the input size in asymptotic notations. These notations are essential because they allow us to assess the complexity of the algorithms without increasing the cost of running them. (Example in terms of n) " "

The simplest illustration is a function $f(n) = n^2+3n$, where the term 3n becomes unimportant in comparison to n^2 when n is very high. The function " $f(n)$ is stated to be asymptotically equivalent to n^2 as $n \rightarrow \infty$ " is expressed symbolically as " $f(n) \sim n^2$" here.

The significance of asymptotic notation

1. They provide straightforward indicators of an algorithm's effectiveness.

2. They make it possible to compare the results of different algorithms.

1.9 ASYMPTOTIC NOTATION

1. Omega(Ω) Notation :

The formal notation for expressing the lowest bound of an algorithm's execution time is Ω (n). It assesses the worst-case time complexity or the

fastest possible runtime for an algorithm. The best-case situation is depicted in this asymptotic notation, which is the opposite of large o notation. The lower bound of an algorithm's execution time is presented formally in this manner. This indicates that this is the shortest amount of time needed to execute an algorithm. It represents the speed at which an algorithm can operate.

Assume that the most significant term in the function f(n), g(n), represents the time complexity of an algorithm.

If f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1.

Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Example

The least amount of time needed to sort 1000 numbers is "10 seconds," if we are sorting 1000 numbers. These 1000 numbers cannot be sorted in less than 10 seconds using omega notation. Not 9 or 8 seconds, but 11 or 12 seconds is OK.

2. Big-o(O) Notation :

The formal notation for expressing an algorithm's running time upper bound is the big-o O. The worst-case time complexity or the longest time an algorithm might possibly take to finish is measured. By indicating the function's order of increase, this asymptotic notation measures an algorithm's efficiency. It gives a function an upper bound, assuring that it will never grow faster than that of the upper bound.

It measures the algorithm's worst-case complexity. Calculates the execution time that took the most time. As a result, it provides a formal means of expressing the maximum allowable execution time for an algorithm.

Assume that the most significant term in the function f(n), g(n), represents the time complexity of an algorithm.

If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0>= 1.

Then we can represent f(n) as O(g(n)).

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

Example:

The maximum amount of time needed to sort 1000 numbers is "50 secs," if we are sorting 1000 numbers. These 1000 numbers can be sorted in less than 50 seconds using big-o notation. 48 or 46 seconds are acceptable but 51 or 52 seconds are not.

3. Theta(Θ) Notation :

In order to express precise asymptotic behaviour, the theta notation bounds a functions from above and below. The execution time of a given algorithm will, "on average," be the same for any given input. The typical case scenario is depicted in this asymptotic notation. When solving a real-world problem, an algorithm cannot perform worst or best. Theta notation (), which denotes the average scenario, shows how the temporal complexity varies between the best case and worst situation. When the value of the worst-case and best-case scenarios are equal, theta notation is commonly utilized. It represents both the top and lower bounds of an algorithm's running time.

Assume that the most significant term in the function f(n), g(n), represents the time complexity of an algorithm.

If C1 g(n) <= f(n) <= C2 g(n) for all n >= n0, C1 > 0, C2 > 0 and n0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Example:

Let's use the example of sorting 1000 numbers once more. The algorithm takes 10 seconds the first time, 11 seconds the second time, and 7 seconds the third time.

1.10 PROPERTIES OF NOTATIONS

Let f(n) and g(n) be asymptotically positive functions. Properties of asymptotic notations:

- Transitivity:
  f(n) = Θ(g(n)) and g(n) = Θ(h(n)) ⇒ f(n) = Θ(h(n)). Valid for O and Ω as well.
- Reflexivity:
  f(n) = Θ(f(n)). Valid for O and Ω.
- Symmetry:
  f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n)).
- Transpose symmetry:
  f(n) = O(g(n)) if and only if g(n) = Ω(f(n)).
- If f(n) is in O(kg(n)) for any constant k > 0,
  then f(n) is in O(g(n)).
- If f1(n) is in O(g1(n)) and f2(n) is in O(g2(n)),
  then (f1 + f2)(n) is in O(max(g1(n)), (g1(n))).
- If f1(n) is in O(g1(n)) and f2(n) is in O(g2(n))
  then f1(n) f2(n) is in O(g1(n) g1(n)).

1.11 SUMMARY

- An algorithm is a set of clear, step-by-step directions for solving a particular issue.
- We may choose the algorithm that uses the least amount of time and space by performing an algorithm analysis.
- The objective of an algorithm analysis is to evaluate different algorithms, mostly in terms of running time but also in terms of other elements.
- The Best Case, Worst Case, and Average Case are used to evaluate an algorithm's complexity.
- Upper Bounding Function in Big-O Notation.
- Lower Bounding Function in the Omega-Ω Notation.
- Order Function in Theta- Θ Notation.

1.12 QUESTIONS

1. What are the essential properties of an algorithms? Explain.
2. Define algorithm. State its essential characteristics.
3. Write an algorithm for adding two integers the user provided.
4. Write an algorithm for calculating factorial of an integer the user provided.
5. Explain why analysis of algorithm is important.
6. Explain the running time of an algorithm.
7. Explain how to compare algorithms. Give example.
8. Explain the terms in Rate of growth.
9. Write a note on Upper and Lower bound of an algorithm.
10. Brief describe Big-O and Omega Ω in algorithm analysis.

1.13 REFERENCE FOR FURTHER READING

- https://www.geeksforgeeks.org/method-of-guessing-and-confirming/
- Analysis of algorithms. (2022, November 15). In Wikipedia. https://en.wikipedia.org/wiki/Analysis_of_algorithms
- https://bournetocode.com/projects/AQA_A_Theory/pages/4-3-Algorithms.html
- Data Structure and Algorithmic Thinking with Python, Narasimha Karumanchi, Career Monk Publications, 2016.
- Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, 2016, Wiley.
- Algorithms by Robert Sedgewick & Kevin Wayne. Addison-Wesley Professional.
- Python Algorithms: Mastering Basic Algorithms in the Python Language, Magnus Lie Hetland.

2

MASTER THEOREM

Unit Structure :

2.1 OBJECTIVE

After going through this unit, you will be able to understand:

Basic algorithm properties , Performance characteristics of algorithms, Master Theorem for Divide and Conquer Problems & Solutions, Master Theorem for Subtract andConquer Recurrences, Method of Guessing and Confirming.

2.2 COMMONLY USED LOGARITHMS AND SUMMATIONS

Because of a number of advantageous characteristics that made complex, time-consuming calculations simpler, logarithms were quickly adopted by scientists.

There are numerous uses for the common logarithm in science and engineering, which is a logarithm to base 10 (b = 10).

A base e logarithm is known as a natural logarithm. Its simpler derivative makes it useful in both mathematics and science.

The base-2 logarithm known as the binary logarithm is frequently employed in computer science.

2.2.1 Guidelines for Asymptotic Analysis

There are some rules to help us determine the running time of an algorithm. I have described with an example using python code

1. Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

2. Nested loops: Analyse from the inside out. Total running time is the product of the sizes of all the loops.

3. Consecutive statements: Add the time complexities of each statement.

4. If-then-else statements: Worst-case running time: the test, plus either the then part `the else part (whichever is the larger).

5. Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by 1/2).

For any positive values of m, n, and r as well as any positive integers a and b, logarithms have the following arithmetic properties.

1. $\log_b(xy) = \log_b x + \log_b y$	$b^{x+y} = b^x b^y$
2. $\log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y$	$b^{x-y} = \frac{b^x}{b^y}$
3. $\log_b(x^y) = y \cdot \log_b x$	$(b^x)^y = b^{xy}$
4. $\log_b 1 = 0$	$b^0 = 1$
5. $\log_b(b) = 1$	$b^1 = b$
6. $\log_b(b^n) = n$	$b^n = b^n$

Source:
https://d2vlcm61l7u1fs.cloudfront.net/media%2F53d%2F53d266b3-56c1-4cea-99a5-4eaf6ecae629%2FphpjB5pBJ.png

2.3 PERFORMANCE CHARACTERISTICS OF ALGORITHMS

There are numerous routes we can take to get from city "A" to city "B." We have the option of travelling by bicycle, bus, train, and aeroplane. We select the one that best suits us based on accessibility and convenience. Similar to this, there are other algorithms available in computer science to address an issue. When there are multiple algorithms available to address a problem, the best solution must be chosen. We can choose the best

algorithm to solve a problem from a variety of algorithms with the use of performance analysis.

When there are several potential solutions to a problem, we evaluate them and choose the one that best meets our needs. The official explanation is as follows:

Making an evaluation of an algorithm is the process of measuring its performance.

The following definitions are also possible:

An algorithm's performance is measured by how well it can predict the resources needed to complete its goal.

This indicates that when there are several algorithms available to solve a problem, we must choose the best algorithm to do so.

To select the best algorithm, we compare algorithms that are solving the same problem. Algorithms are compared using a set of parameters or elements, such as the amount of memory needed, how quickly the algorithm runs, how simple it is to understand and apply, etc.

In general, the following factors affect how well an algorithm performs:

1. Does that algorithm offer a precise solution to the issue?

2. Is it simple to understand?

3. How simple is it to put into practise?

4. How much memory (space) is needed to fix the issue?

5. How long does it take to resolve the issue? Etc.,

When analysing an algorithm, we solely take into account the time and space needed for that specific method, ignoring all other factors.

These details can also be used to define performance analysis of an algorithm, which is calculating the amount of time and space an algorithm needs to run is the process of performing performance analysis on the algorithm.

2.4 MASTER THEOREM FOR DIVIDE AND CONQUER

One of the various algorithm models is the Divide and Conquer strategy. Essentially, it consists of three steps. −

Divide : In this stage, the problem is divided into a number of smaller, related problems.

Conquer : Use recursive solutions to solve the sub problems.

Combine : the solutions to the individual problems to arrive at the solution.

$T(n) = aT(n/b) + f(n)$

where n is the size of the problem

a = the number of sub problems in the recursion, where a >= 1

n/b = the size of each sub problem

f(n) = the price of work performed in addition to recursive calls, such as splitting a problem into smaller ones and the cost of combining those smaller issues to obtain the answer.

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$

$\epsilon > 0$ is a constant.

2.5 MASTER THEOREM: PROBLEMS & SOLUTIONS

Example 1: Use Master's theorem to solve the recurrence relation shown below-

$T(n) = 3T(n/2) + n^2$

Solution:

$T(n) = aT(n/b) + f(n)$

Here

a = 3

b = 2

$f(n) = n^2$

$\log_b a = \log_2 3 = 1.58$

means $f(n) < n^{\log_b a+\square}$, where, □ is a constant.

Here, Case 3 suggests-

$T(n) = f(n) = \Theta(n^2)$

Example 2: Use Master's theorem to solve the recurrence relation shown below-

$T(n) = 8T(n/4) – n^2 \log n$

Solution:

$T(n) = aT(n/b) + f(n)$

Here

The provided recurrence relation does not match Master's theorem's general form.

Therefore, the Master's theorem cannot be used to solve it.

Example 3: Use Master's theorem to solve the recurrence relation shown below-

$T(n)=2T(n/2)+n$

Solution:

$T(n) = aT(n/b) + f(n)$

Here

$a = 2$

$b = 2$

$f(n) = n^2$

$\log_b{}^a = \log_2{}^2 = 1$

means $f(n) = n^{\log_b{}^{a+\square}}$, where, □ is a constant.

Here, Case 2 suggests-

$T(n) = f(n) = \Theta(n^{\log_b{}^a} * \log n) = \Theta(n \log n)$

Example 4: Use Master's theorem to solve the recurrence relation shown below-

$T(n) = 3T (n/2)+n^3$

Solution: $T(n) = 3T (n/2) + n^3 \Rightarrow T (n) =(n^3)$ (Master Theorem Case 3.a)

Solution:

$T(n) = aT(n/b) + f(n)$

Here

$a = 3$

$b = 2$

$f(n) = n^2$

$\log_b{}^a = \log_2{}^3 = 1.58$

means $f(n) = n^{\log_b{}^{a+\square}}$, where, □ is a constant.

Here, Case 3 suggests-

$T(n) = f(n) = \Theta(n^3)$

Example 5: Use Master's theorem to solve the recurrence relation shown below-

$T(n) = 4T(n/2)+n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = O(n\log n)$ (Master Theorem Case 2.a)

Solution:

$T(n) = aT(n/b) + f(n)$

Here

$a = 4$

$b = 2$

$f(n) = n^2$

$\log_b{}^a = \log_2{}^4 = 2$

means $f(n) = n^{\log_b{}^{a+\square}}$, where, □ is a constant.

Here, Case 2 suggests-

$T(n) = f(n) = \Theta(n^{\log_b{}^a} * \log n) = \Theta(n \log n)$

Example 6: Use Master's theorem to solve the recurrence relation shown below-

$T(n)=2^nT(n/2)+n^n$

Solution:

$T(n) = aT(n/b) + f(n)$

Here

The provided recurrence relation does not match Master's theorem's general form.

Therefore, the Master's theorem cannot be used to solve it.

Problem-7 $T(n) = 2T(n/2) + n/logn$
Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(nloglogn)$ (Master Theorem Case 2. b)

Problem-8 $T(n) = 2T(n/4) + n^{051}$
Solution: $T(n) = 2T(n/4) + n^{051} \Rightarrow T(n) = \Theta(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$
Solution: $T(n) = 0.5T(n/2) + 1/n$ => Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$
Solution: $T(n) = 6T(n/3) + n^2logn \Rightarrow T(n) = \Theta(n^2logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2logn$
Solution: $T(n) = 64T(n/8) - n^2logn$ => Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$
Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$
Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$
Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Source: https://dotnettutorials.net/lesson/master-theorem/

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$
Solution: $T(n) = \sqrt{2}T(n/2) + logn \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$
Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$
Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$
Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$
Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$
Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

Source: https://dotnettutorials.net/lesson/master-theorem/

2.6 MASTER THEOREM FOR SUBTRACT AND CONQUER RECURRENCES

Master theorem is used to determine the Big – O upper bound on functions which possess recurrence, i.e which can be broken into sub problems.

Let T(n) be a function defined on positive n as shown below:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases} ,$$

for some constants c, a>0, b>0, k>=0 and function f(n). If f(n) is O(nk), then

1. If a<1 then $T(n) = O(n^k)$

2. If a=1 then $T(n) = O(n^{k+1})$

3. if a>1 then $T(n) = O(n^k a^{n/b})$

1. If a<1 then $T(n) = O(n^k)$
2. If a=1 then $T(n) = O(n^{k+1})$
3. if a>1 then $T(n) = O(n^k a^{n/b})$

Example 1: Use Master's theorem for Subtract and Conquer Recurrences to solve the below-

T(n) = 3T(n-1) when a>0

=1 otherwise

Solution:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases} ,$$

Here

a=3

b=1

d=0

Hence T(n) = O(n03n/1)

Here, Case 2 suggests- (a>1)

Therefore T(n)=O(3n).

Example 2: Use Master's theorem for Subtract and Conquer Recurrences to solve the below-

T(n) = 5T(n-3)+O(n2) when n>0

=1 otherwise

Solution:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b)+f(n), & n > 1, \end{cases}$$

Here

a=5

b=3

d=2

Hence T(n) = O(n25n/3)

Here, Case 2 suggests- (a>1)

Therefore T(n)= O(n25n/3)

2.7 METHOD OF GUESSING AND CONFIRMING

This approach's fundamental premise is to make a well-informed prediction and then use induction to confirm that it is true. Any recurrence can be resolved using this technique. In general, either the proof will work (in which case we are finished) or the proof will fail if a solution is guessed and then we attempt to check our guess inductively (in that case the failure will help us refine our guess).

Consider the recurrence T(N) = N*T(N) + N, for instance. The format demanded by the Master Theorems is not met by this. A careful examination of the recurrence gives us the idea that it is equivalent to the divide and conquer strategy, which involves breaking the problem into N subproblems, each of size N. As can be seen, the first level of recursion's subproblems are N bytes in size. Therefore, let's assume that T(N) = O(N*log N) and then attempt to support our assumption. Let’s start by trying to prove an upper bound:

T(N) \le cN*logN:

T(N) = √N*T(√N) + N

T(N) \le √N* c√N*log√N + N

T(N) = N* clog√N + N

T(N) = N*1/2*c*log N + N

T(N) \le cN*log N

Only that 1 le 1/2*clog N is assumed in the final inequality. If N is large enough and c is constant, regardless of how little it is, this is true.

We can see from the previous proof that we were right about the upper bound. Let's now establish the bottom bound for its occurrence:

T(N) = √N*T(√N) + N

T(N) \ge √N* k√N*log√N + N

T(N) = N* klog√N + N

T(N) = N*1/2*k*log N + N

T(N) \ge N*k*log N

The final inequality just makes the assumption that 1 ge 1/2*k*log N. If N is large enough and for any constant k, this is false.

We can see from the previous proof that our assumption regarding the lower bound is unreliable.

It is clear from the reasoning above that (N*log N) is an excessively large number. How about Θ (N) instead? The direct proof of the lower bound is simple:

T(N) = √N*T(√N) + N \ge N

Now, let us prove the upper bound for this Θ(N):

T(N) = √N*T(√N) + N\\ \le √N*c√N + N\\ = N c+ N\\ = N (c + 1)\\ \nleqcN

It is clear from the previous induction that Θ(N) is too small and Θ(N*log N)(N*log N) is too large. So what do we need that is greater than N and less than N*log N? Consider N*√log N.

Proving upper bound for N*√log N:

T(N) = √N*T(√N) + N

T(N) \le √n*c√N*√(log √N) + N

T(N) = N* 1/√2 *c*log √N + N

T(N) \le N*c*log √N

Proving lower bound for N*√log N:

T(N) = √N*T(√N) + N

T(N) \ge √N*k√N*√(log√N) + N

T(N) = N*1/√2*k*log √N + N

T(N) \ngeq N*k*log √N

The last step doesn't work. So, Θ(N*√log N) doesn't work. What else is between N and N*log N? How about N*log(log N)?

Proving upper bound for N*log(log N):

T(N) = √N*T(√N) + N

T(N) \le √N*c√N*log(log √N) + N

T(N) = N*clog(log N) - cN + N

T(N) \le N*clog(log N), if \ c\ge 1

Proving lower bound for N*log(log N):

T(N) = √N*T(√N) + N

T(N) \ge √N*k√N*log(log √N) + N

T(N) = N*k*log(log N) - kN + N

T(N) \ge N*klog(log N), if \ k \le 1

From the above proofs, it can see that T(N) \le N*clog(log N), if \ c \ge 1 and T(N) \ge N*klog(log N), if \ k \le 1.

2.8 SUMMARY

- There are numerous uses for the common logarithm in science and engineering
- An algorithm's performance is measured by how well it can predict the resources needed to complete its goal.
- When analysing an algorithm, we solely take into account the time and space needed for that specific method, ignoring all other factors.
- The master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.
- Master theorem is used to determine the Big – O upper bound on functions which possess recurrence, i.e which can be broken into sub problems.
- The basic idea behind this method is to guess the answer, and then prove it correct by induction. This method can be used to solve any recurrence.

2.9 QUESTIONS

1. Use Master's theorem to solve the below:

a. $T(n) = 4T(n/2) + n$

b. $T(n) = 2T(n/2) + \log n$

c. $T(n) = 4T(n/2) + n \log n (\log_2^4 = 2)$

d. $T(n) = 3T(n/3) + n \log n$

e. $T(n) = 4T(n/4) + n$

f. $T(n) = T(n/2) + 1$

g. $T(n) = 2T(n/2) + n^2$

2. What is commonly used Logarithms and Summations.
3. Write a note on Master Theorem. Give an example.
4. Write a note on Master Theorem for Subtract and Conquer Recurrences
5. Write a note on method of guessing and conforming.
6. Write a note on Master Theorem for Divide and Conquer.
7. Explain Guidelines for Asymptotic Analysis.
8. Which factors affect how well an algorithm performs?
9. Write a note on Performance characteristics of algorithms.

2.10 REFERENCE FOR FURTHER READING

- https://www.geeksforgeeks.org/method-of-guessing-and-confirming/
- Analysis of algorithms. (2022, November 15). In *Wikipedia*. https://en.wikipedia.org/wiki/Analysis_of_algorithms
- https://bournetocode.com/projects/AQA_A_Theory/pages/4-3-Algorithms.html
- Data Structure and Algorithmic Thinking with Python, NarasimhaKarumanchi ,CareerMonk Publications, 2016.
- Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, 2016, Wiley.
- Algorithms by Robert Sedgewick & Kevin Wayne. Addison-Wesley Professional.
- Python Algorithms: Mastering Basic Algorithms in the Python Language, Magnus Lie Hetland.

3

TREE ALGORITHMS

Unit Structure

3.0 OBJECTIVES

What is the objective of algorithms?

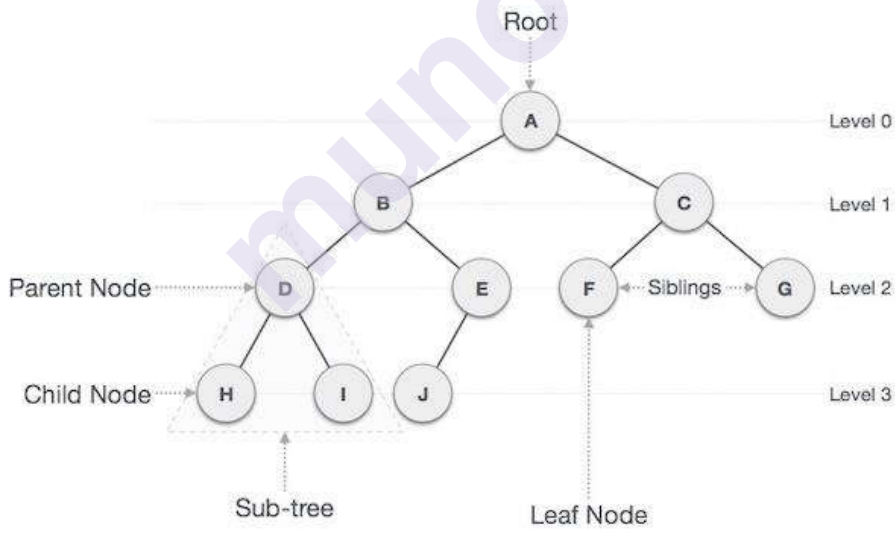
1. Algorithms **define how a particular task is to be performed**. They provide computers with instructions. They tell machines on assembly lines how specific parts should be put together. Or they evaluate application forms and recommend the best candidates.

2. Provides a hierarchical way of storing data. Reflects structural relationship in a data set. Allows insertion, deletion and searching operations that yield results faster than an array or linked list. Provides a flexible way to hold and move data.

3. Hierarchical Structure: Trees are used to model hierarchical structures, such as the file system in a computer or the organizational chart in a company. The tree structure **allows for a natural representation of parent-child relationships, making it easy to understand and visualize the data**.

3.1 INTRODUCTION

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path frocm the root node to any node.

- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − 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.
- **Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Types of Trees

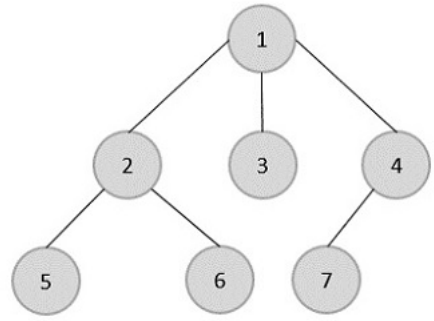
There are three types of trees −

- General Trees
- Binary Trees
- Binary Search Trees

General Trees

General trees are unordered tree data structures where the root node has minimum 0 or maximum 'n' subtrees.

The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.



General Tree Data Structure

Binary Trees

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.

Full Binary Tree

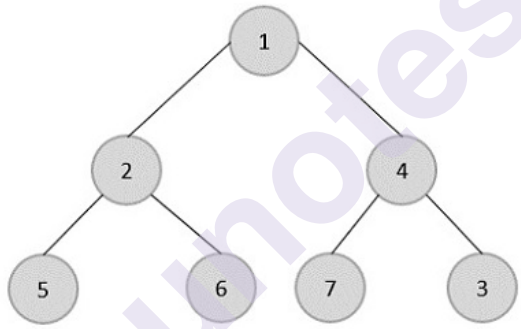
- A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree

- A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

Perfect Binary Tree

- A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

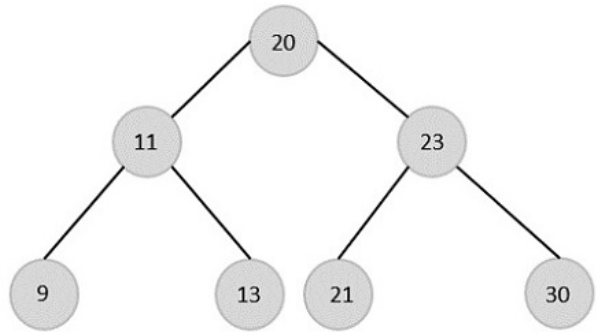


Binary Tree Data Structure

Binary Search Trees

Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.

The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.



Binary Search Tree Data Structure

Advantages of BST

- Binary Search Trees are more efficient than Binary Trees since time complexity for performing various operations reduces.
- Since the order of keys is based on just the parent node, searching operation becomes simpler.
- The alignment of BST also favors Range Queries, which are executed to find values existing between two keys. This helps in the Database Management System.

Disadvantages of BST

The main disadvantage of Binary Search Trees is that if all elements in nodes are either greater than or lesser than the root node, **the tree becomes skewed**. Simply put, the tree becomes slanted to one side completely.

This **skewness** will make the tree a linked list rather than a BST, since the worst case time complexity for searching operation becomes O(n).

To overcome this issue of skewness in the Binary Search Trees, the concept of **Balanced Binary Search Trees** was introduced.

Balanced Binary Search Trees

Consider a Binary Search Tree with ‘m’ as the height of the left subtree and ‘n’ as the height of the right subtree. If the value of (m-n) is equal to 0,1 or -1, the tree is said to be a **Balanced Binary Search Tree**.

The trees are designed in a way that they self-balance once the height difference exceeds 3. Binary Search Trees use rotations as self-balancing algorithms. There are four different types of rotations: Left Left, Right Right, Left Right, Right Left.

There are various types of self-balancing binary search trees −

- AVL Trees
- Red Black Trees
- B Trees
- B+ Trees
- Splay Trees
- Priority Search Trees

WHAT IS TREE?

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous

linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.3.1 shows a tree and a non-tree.

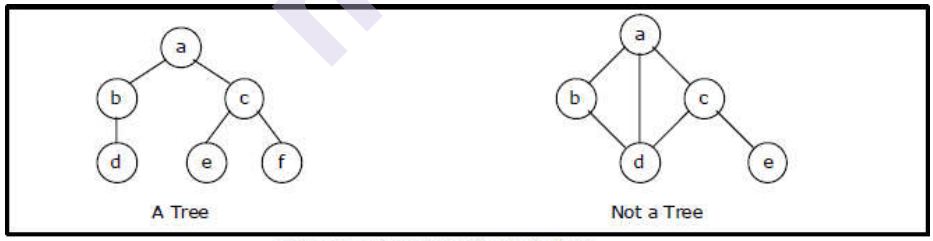


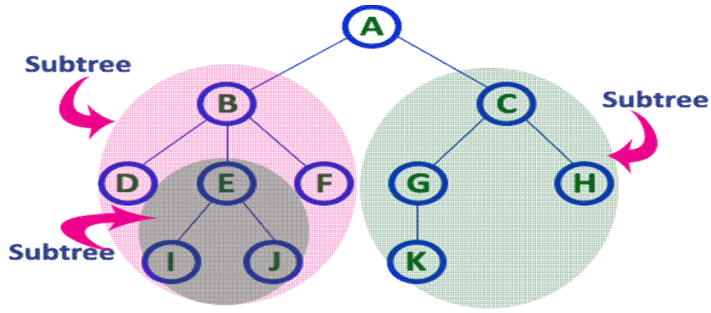
Figure 5.1.1 A Tree and a not a tree

Tree is a popular data structure used in wide range of applications. A tree data structure can bedefined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is arecursive definition.

A tree data structure can also be defined as follows...A tree is a finite set of one or more nodes such that:There is a specially designated node called the root. The remaining nodes are partitioned into n>=0disjoint sets T1, ...,

Tn, where each of these sets is a tree. We call T1, ..., Tn are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root ofa subtree. At the bottom of the tree are leaf nodes, which have no children.

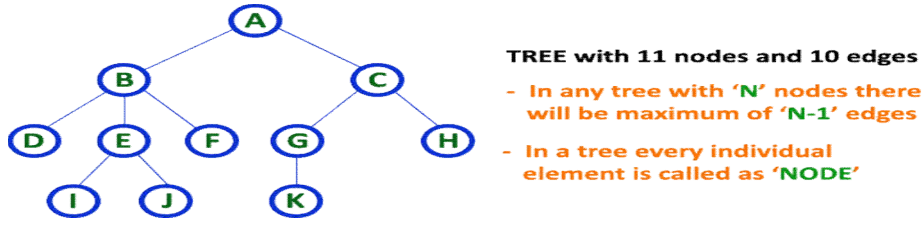
Advantages of trees

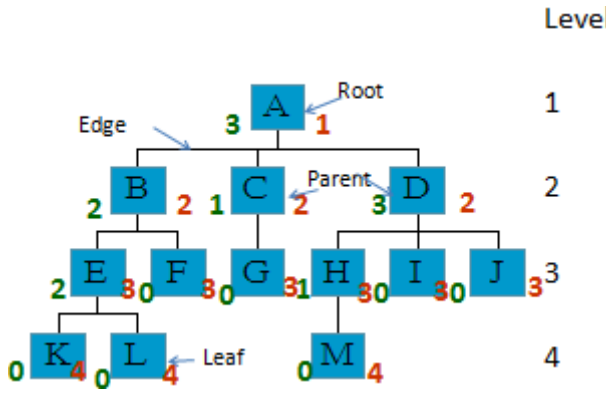
Trees are so useful and frequently used, because they have some very serious advantages:

- 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 sub trees around with minimum effort

3.2 AN OVERVIEW

In a Tree, Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. Example





1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. Wecan say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, **A** is a **Root** node

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. Insimple words, the node which has branch from it to any other node is called as parent node. Parentnode can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes. e.g., Children of D are (H, I,J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

6. Leaf

In a tree data structure, the node which does not have a child (or) node with degree zero is calledas LEAF Node. In simple words, a leaf is a node with no child.In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a nodewith no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

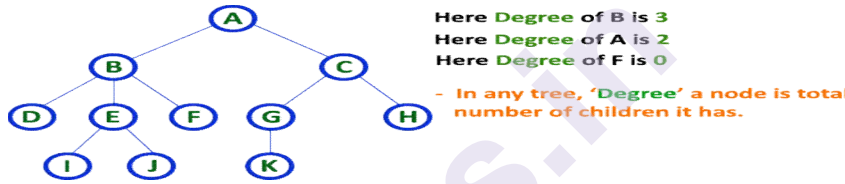
7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simplewords, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node isalso said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes. Ex:B,C,D,E,H

8. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children ithas. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simplewords, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 3.

10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longestpath is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of thetree. In a tree, height of all leaf nodes is '0'.

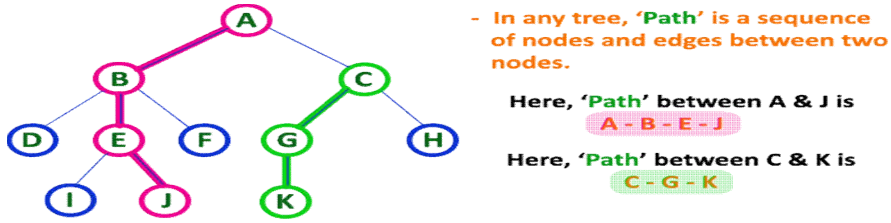
11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the

longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in atree is said to be depth of that tree. In a tree, depth of the root node is '0'.

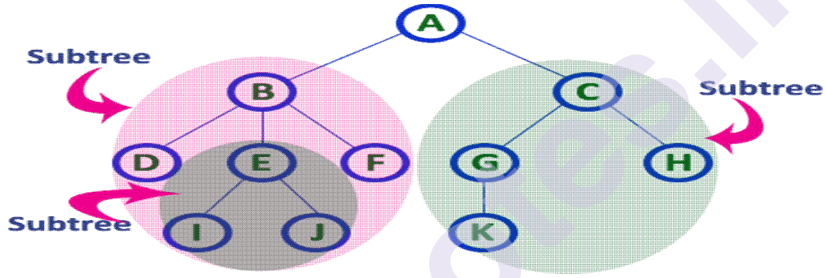
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In belowexample the path A - B - E - J has length 4.



13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node willform a subtree on its parent node.

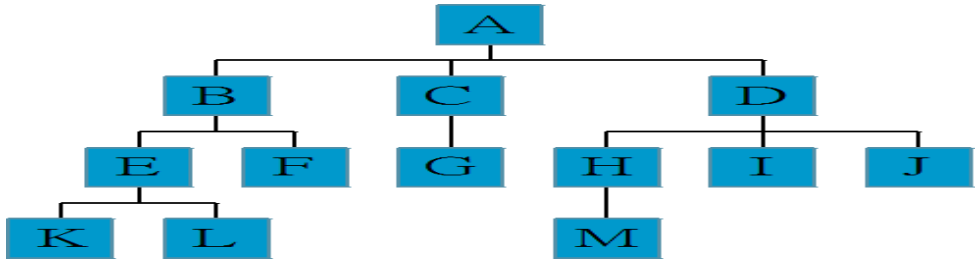


Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1.List Representation

2. Left Child - Right Sibling Representation Consider the following tree...



1. List Representation

In this representation, we use two types of nodes one for representing the node with data and anotherfor representing only references. We start with a node with data from root node in the tree. Then it islinked to an internal

node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

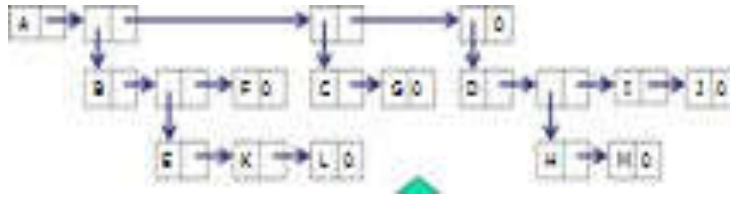


Fig: List representation of above Tree

List Representation

– (A (B (E (K, L), F), C (G), D (H (M), I, J)))

– The root comes first, followed by a list of sub-trees

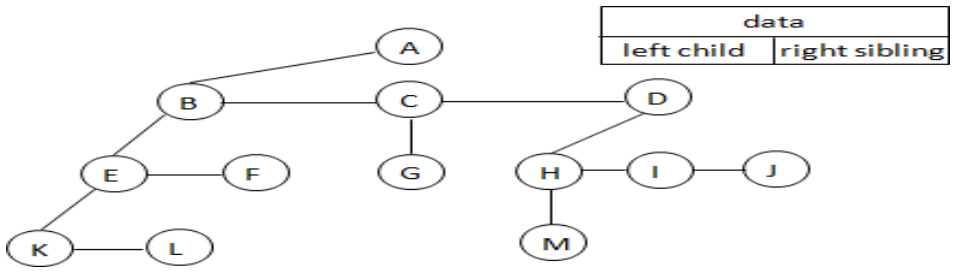
Fig: Possible node structure for a tree of degree k

2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value ofa node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

In this representation, every node's data field stores the actual value of that node. If that node has leftchild, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling nodeotherwise that field stores NULL.

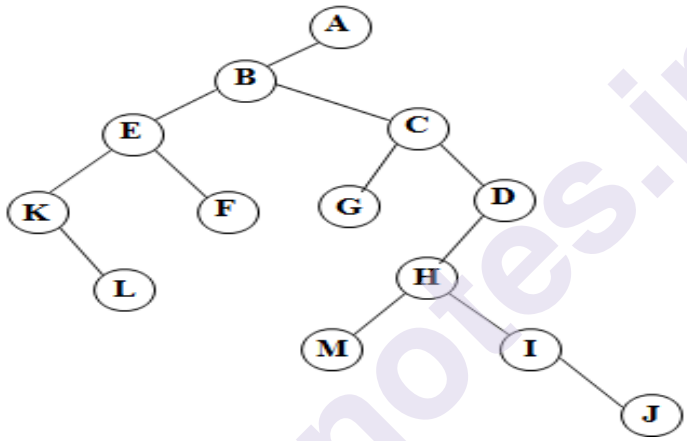
The above tree example can be represented using Left Child - Right Sibling representation asfollows...



Representation as a Degree –Two Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child- right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)*



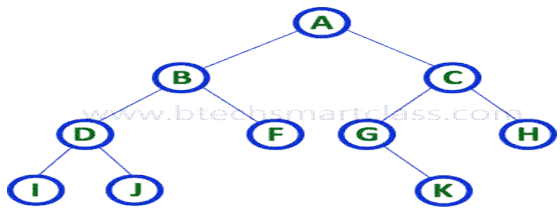
Binary Trees

Introduction

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left childand the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2children. Example



There are different types of binary trees and they are...Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, everynode should have exactly two children or none. That means every internal node must have exactlytwo children. A strictly Binary Tree can be defined as follows...

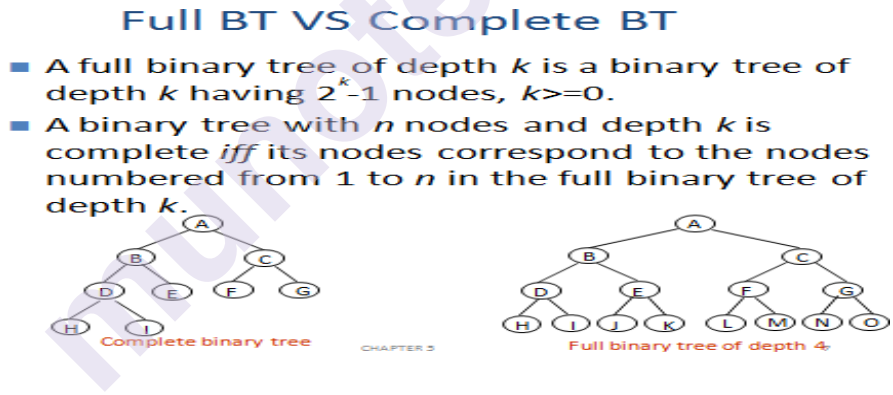
A binary tree in which every node has either two or zero number of children is called Strictly BinaryTree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

1. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, everynode should have exactly two children or none and in complete binary tree all the nodes must haveexactly two children and at every level of complete binary tree there must be 2 level number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at samelevel is called Complete Binary Tree.

Complete binary tree is also called as Perfect Binary Tree



2. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodeswherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended BinaryTree.

Abstract Data Type Definition: **A binary tree is a finite set of nodes that is either empty or consists of a root and twodisjoint binary trees called left subtree and right subtree.**

ADT contains specification for the binary tree ADT.

Structure Binary_Tree (abbreviated BinTree) is objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree. Functions:

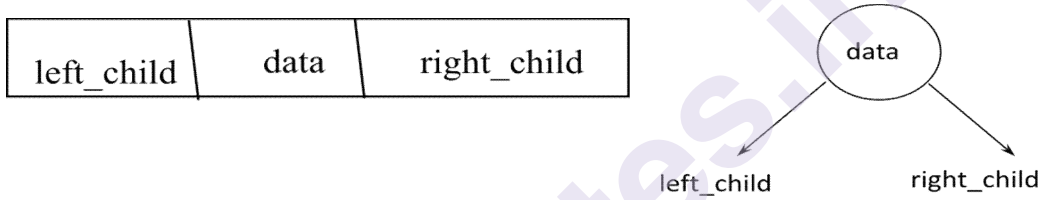
For all bt, bt1, bt2 ∈BinTree, item ∈ element Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(bt)::= if (bt==empty binary tree) return TRUE else return FALSE

BinTree MakeBT(bt1, item, bt2)::= return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item

Bintree Lchild(bt)::= if (IsEmpty(bt)) return error else return the left subtree of bt element Data(bt)::= if (IsEmpty(bt)) return error else return the data in the root node of btBintree Rchild(bt)::= if (IsEmpty(bt)) return error else return the right subtree of bt

We use linked list to represent a binary tree. In a linked list, every node consists of three fields. Firstfield, for storing left child address, second for storing actual data and third for storing right child address. In this linked

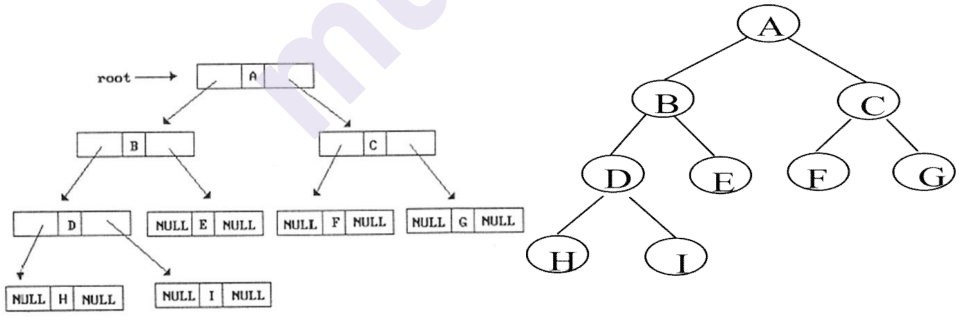


list representation, a node has the following structure...

Type def struct node *tree_pointer;typedef struct node {

int data;

tree_pointer left_child, right_child;};



Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes ofthat binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1) In - Order Traversal

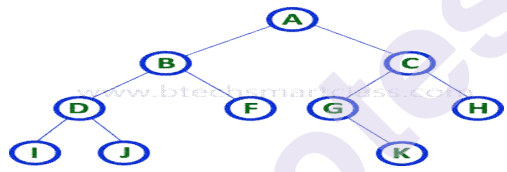
2) Pre - Order Traversal

3) Post - Order Traversal

Binary Tree Traversals

- 3. In – Order Traversal (leftChild – root – rightChild)
- I – D – J – B – F – A – G – K – C – H
- 2. Pre – Order Traversal (root – leftChild – rightChild)
- A – B – D – I – J – F – C – G – K – H
- 3. Post – Order Traversal (leftChild – rightChild – root)
- I – J – D – F – B – K – G – H – C – A

CHAPTER 5



In - Order Traversal (left Child - root – right Child)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, theleft child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child isa root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first wevisit 'I'then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part sowe visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node'C' and next visit C's right child 'H' which is the right most child in the tree so westop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-OrderTraversal.

In-Order Traversal for above example of binary tree isI - D - J - B - F - A - G - K - C – H

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

void inorder(tree_pointer ptr)**/* inorder tree traversal */ Recursive**

{

if (ptr) {

inorder(ptr->left_child); printf("%d", ptr->data); indorder(ptr->right_child);

}

}

1. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child'F'. With this we have completed root and left parts of node A. So we go for A's rightchild 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Algorithm

Until all nodes are traversed −Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree. Step 3 − Recursively traverse right subtree.

void preorder(tree_pointer ptr) **/* preorder tree traversal */ Recursive**

{

```
if (ptr) {

printf("%d", ptr->data); preorder(ptr->left_child); preorder(ptr->right_child);

}

}
```

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, leftchild node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree. Step 2 − Recursively traverse right subtree.Step 3 − Visit root node.

```
void postorder(tree_pointer ptr)      /* postorder tree traversal */
      Recursive

{

if (ptr) {

postorder(ptr->left_child); postorder(ptr->right_child); printf("%d", ptr->data);

}

}
```

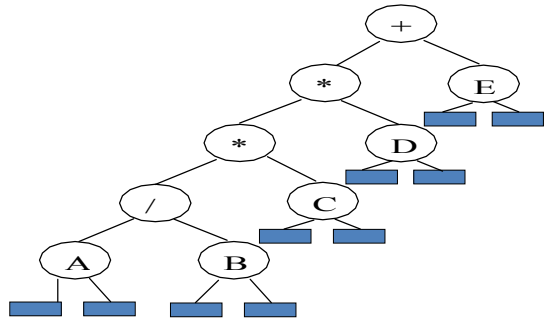
3.3 ARITHMATIC TREE

Arithmetic Expression Using BT

inorder traversal A / B * C * D + E infix expression preorder traversal

+ * * / A B C D E prefix expression postorder traversal A B / C * D * E + postfix expression level order traversal

+ * E * D / C A B



15

Trace Operations of Inorder Traversal

Iterative Inorder Traversal (using stack)

```
void iter_inorder (tree_pointer node)

{

int top= -1;    /* initialize stack */ tree_pointer stack[MAX_STACK_SIZE];for (;;) {

for (; node; node=node->left_child) add(&top, node);  /* add to stack */

node= delete(&top);/* delete from stack */if (!node) break;     /* empty stack */
printf("%D", node->data);

node = node->right_child;

}

}
```

In iterative in order traversal, we must create our own stack to add and remove nodes as in recursion.

Analysis: Let n be number of nodes in tree, every node of tree is placed on and removed from the stack exactly once. So time complexity is O(n). The space requirement is equal to the depth of tree which is O(n).

Level Order Traversal (Using Queue) -- Traversal without Stack

```
void level_order(tree_pointer ptr)  /* level order tree traversal */

{

int front = rear = 0;

tree_pointer queue[MAX_QUEUE_SIZE]; if (!ptr) return; /* empty queue */
addq(front, &rear, ptr);

for (;;) {

ptr = deleteq(&front, rear);if (ptr) {

printf("%d", ptr->data);if (ptr->left_child)
```

```
addq(front, &rear, ptr->left_child);if (ptr->right_child)
addq(front, &rear, ptr->right_child);
}
else break;
}                }
```

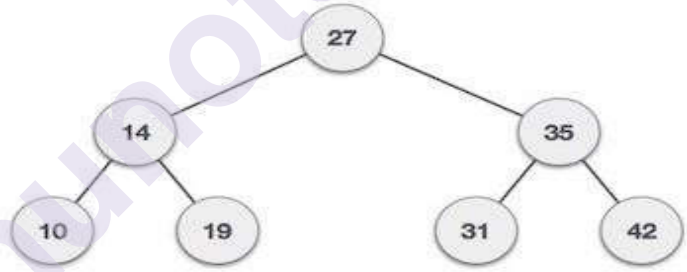
Level order Traversal is implemented with circular queue. In this order, we visit the root first, then root's left child followed by root's right child. We continue in this manner, visiting the nodes at each new level from left most to rightmost nodes.

We begin by adding root to the queue. The function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. The level order traversal for abovearithmetic expression is + * E * D / C A B.

Binary Search Trees

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty,then all nodes follows the below mentioned properties −

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in theroot of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.left sub-tree and right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

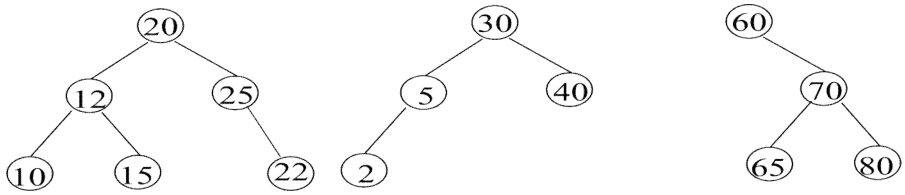


Fig: Example Binary Search Trees

ADT for Dictionary:

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following −

- **Search** − search an element in a binary search tree.
- **Insert** − insert an element into a binary search tree / create a tree.
- **Delete** − Delete an element from a binary search tree.
- **Height** -- Height of a binary search tree.

Searching a Binary Search Tree

Let an element k is to search in binary search tree. Start search from root node of the search tree. If root is NULL, search tree contains no nodes and search unsuccessful. Otherwise, compare k withthe key in the root. If k equals the root's key, terminate search, if k is less than key value, searchelement k in left subtree otherwise search element k in right subtree. The function search recursivelysearches the subtrees.

Algorithm:Recursive search of a Binary Search Tree

```
tree_pointer search(tree_pointer root,  int key)

{

/* return  a pointerto the node that contains  key. If there is no  such  node,
return NULL */

if (!root) return NULL;

if (key == root->data) return root;if (key < root->data)

return search(root->left_child,  key);return search(root->right_child,key);

}
```

Algorithm: Iteraive search of a Binary Search Tree

```
tree_pointer search2(tree_pointer tree, int key)
```

```
{
while (tree) {
if (key == tree->data) return tree; if (key < tree->data)
tree = tree->left_child; else tree = tree->right_child;
}
return NULL;
}
```

Analysis of Recursive search and Iterative Search Algorithms:

If h is the height of the binary search tree, both algorithms perform search in O(h) time. Recursive search requires additional stack space which is O(h).

3.4 BALANCED TREE

Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binarysearch 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.

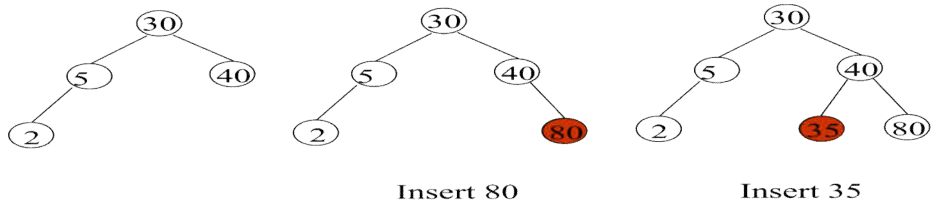
Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than thenode (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNodeis larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach a node (e.i., reach to NULL) where search terminates.

Step 7: After reaching a last node, then insert the newNode as left child if newNode is smaller orequal to that node else insert it as right child.



Algorithm

Create newnodeIf root is NULL

then create root nodereturn

If root exists then

compare the data with node.data

while until insertion position is locatedIf data is greater than node.data

goto right subtreeelse

goto left subtreeendwhile

insert newnodeend If

Implementation

The implementation of insert function should look like this –

```
void insert(int data) {
  struct node *tempNode = (struct node*)
  malloc(sizeof(struct node));struct node *current;
  struct
  node
  *parent;
  tempNo
  de-
  >data =
  data;
  tempNo
  de-
  >leftChi
  ld =
  NULL;
  tempNode->rightChild = NULL;
  //if tree is
  empty,
  create root
```

```
        retur        }
    }

    //go to right of
    the treeelse {
      current = current-
      >rightChild;
      //insert to the
      right if(current
      == NULL ) {
  }
}
```

Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

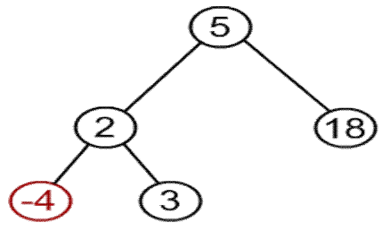
- search for a node to remove
- if the node is found, run remove algorithm.

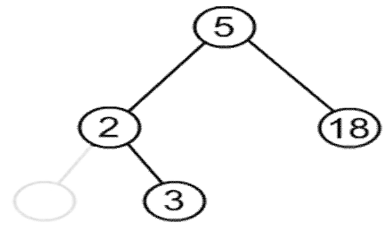
Remove algorithm in detail

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. Thereare three cases, which are described below**.**

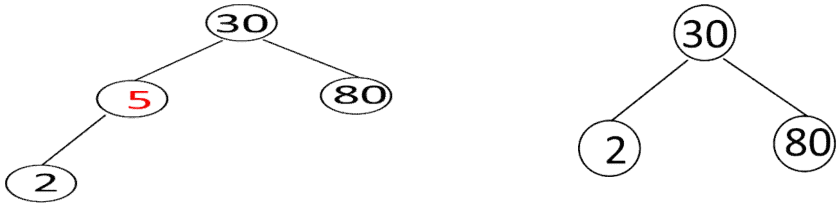
1. Node to be removed has no children. --This case is quite simple. Algorithm sets correspondinglink of the parent to NULL and disposes the node.

Example. Remove -4 from a BST.

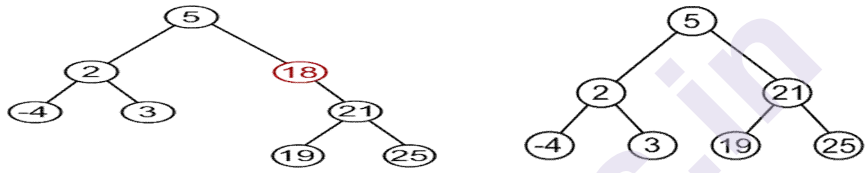




2. Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.



3. Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one 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 has 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 childrenCase 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 and child nodes.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 inits right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2Step 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 node is deleted from the tree.

```
/* deletion in binary search tree */

void deletion(struct treeNode **node, struct treeNode **parent, int data) {
struct treeNode *tmpNode, *tmpParent;

if (*node == NULL)return;

if ((*node)->data == data) {

/* deleting the leaf node */

if (!(*node)->left && !(*node)->right) {if (parent) {

/* delete leaf node */

if ((*parent)->left == *node) (*parent)->left = NULL;

else

(*parent)->right = NULL;

free(*node);

} else {

/* delete root node with no children */free(*node);

}

/* deleting node with one child */

} else if (!(*node)->right && (*node)->left) {

/* deleting node with left child alone */tmpNode = *node;

(*parent)->right = (*node)->left;free(tmpNode);

*node = (*parent)->right;

} else if ((*node)->right && !(*node)->left) {

/* deleting node with right child alone */tmpNode = *node;

(*parent)->left = (*node)->right;free(tmpNode);

(*node) = (*parent)->left;
```

```
} else if (!(*node)->right->left) {
/*
*       deleting a node whose right child
*       is the smallest node in the right
*       subtree for the node to be deleted.
*/
tmpNode = *node;
(*node)->right->left = (*node)->left;
(*parent)->left = (*node)->right;free(tmpNode);
*node = (*parent)->left;
} else {
/*
*       Deleting a node with two children.
*       First, find the smallest node in
*       the right subtree.  Replace the
*       smallest node with the node to be
*       deleted. Then, do proper connections
*       for the children of replaced node.
*/
tmpNode = (*node)->right;while (tmpNode->left) {tmpParent = tmpNode;
tmpNode = tmpNode->left;
}
tmpParent->left = tmpNode->right; tmpNode->left = (*node)->left;
tmpNode->right =(*node)->right; free(*node);
*node = tmpNode;
}
} else if (data < (*node)->data) {
/* traverse towards left subtree */ deletion(&(*node)->left, node, data);
} else if (data > (*node)->data) {
/* traversing towards right subtree */ deletion(&(*node)->right, node,
```

data);

}

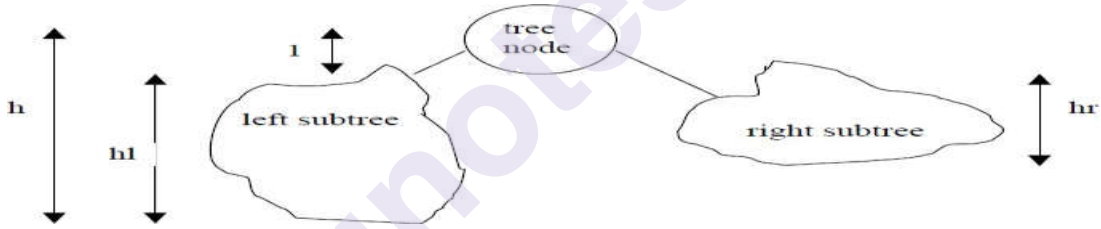
}

Height of a Binary Search Tree:

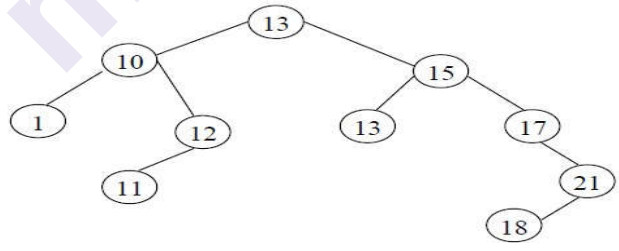
Height of a Binary Tree For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of –3.

The following height function in pseudocode is defined recursively

```
int height( BinaryTree Node t) {
    if t is a null tree
        return  -1;
    hl = height( left subtree of t);
    hr = height( right subtree of t);
    h = 1 + maximum of hl and hr;

    return h;
}
```



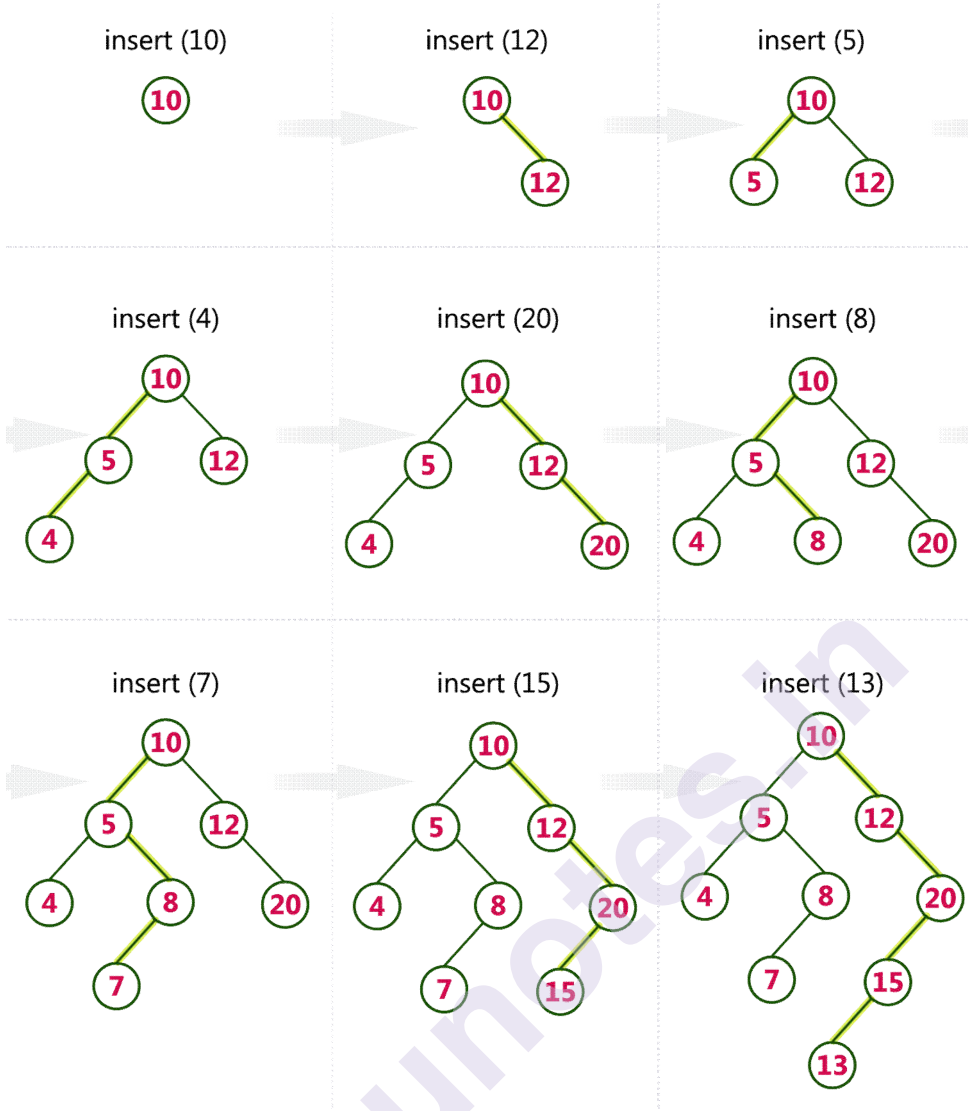
For example, the following tree has a height of 4. Its left subtree has height 2 and its right subtree 3.



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...



Generic Trees (N-ary Trees)

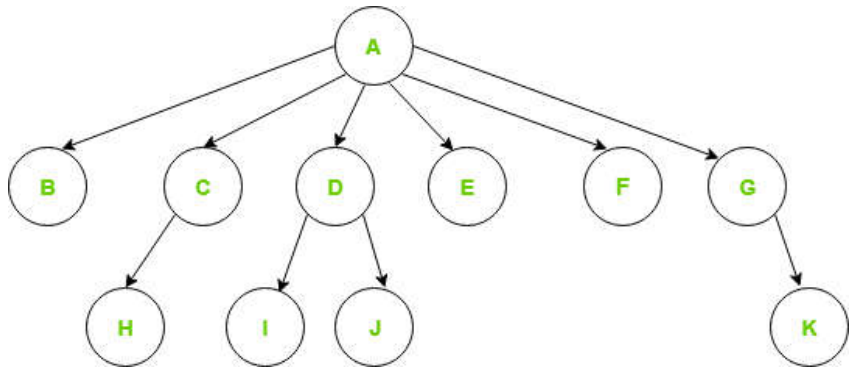
Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes. Every node stores address of its children and the very first node's address will be stored in a separate pointer called root.

The Generic trees are the N-ary trees which have the following properties:

1. Many children at every node.

2. The number of nodes for each node is not known in advance.

Example:



Generic Tree

To represent the above tree, we have to consider the worst case, that is the node with maximum children (in above example, 6 children) and allocate that many pointers for each node.

The node representation based on this method can be written as:

```
struct Node{
  int data;
  Node *firstchild;
  Node *secondchild;
  Node *thirdchild;
  Node *fourthchild;
  Node *fifthchild;
  Node *sixthchild;
};
```


Disadvantages of the above representation are:

1. **Memory Wastage** – All the pointers are not required in all the cases. Hence, there is lot of memory wastage.
2. **Unknown number of children** – The number of children for each node is not known in advance.

Simple Approach:

For storing the address of children in a node we can use an array or linked list. But we will face some issues with both of them.

1. In **Linked list**, we can not randomly access any child's address. So it will be expensive.

2. In **array**, we can randomly access the address of any child, but we can store only fixed number of children's addresses in it.

Better Approach:

We can use Dynamic Arrays for storing the address of children's address. We can randomly access any child's address and the size of the vector is also not fixed.

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

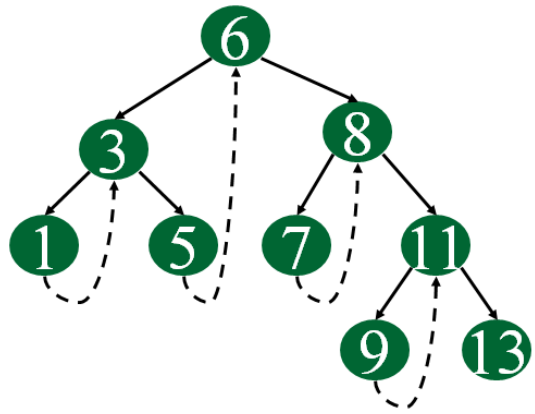
Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node. Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C and C++ representation of a Threaded Node
Following is C and C++representation of a single-threaded node.

```
struct Node
{
   int data;
   struct Node *left, *right;
   bool rightThread;
}
```

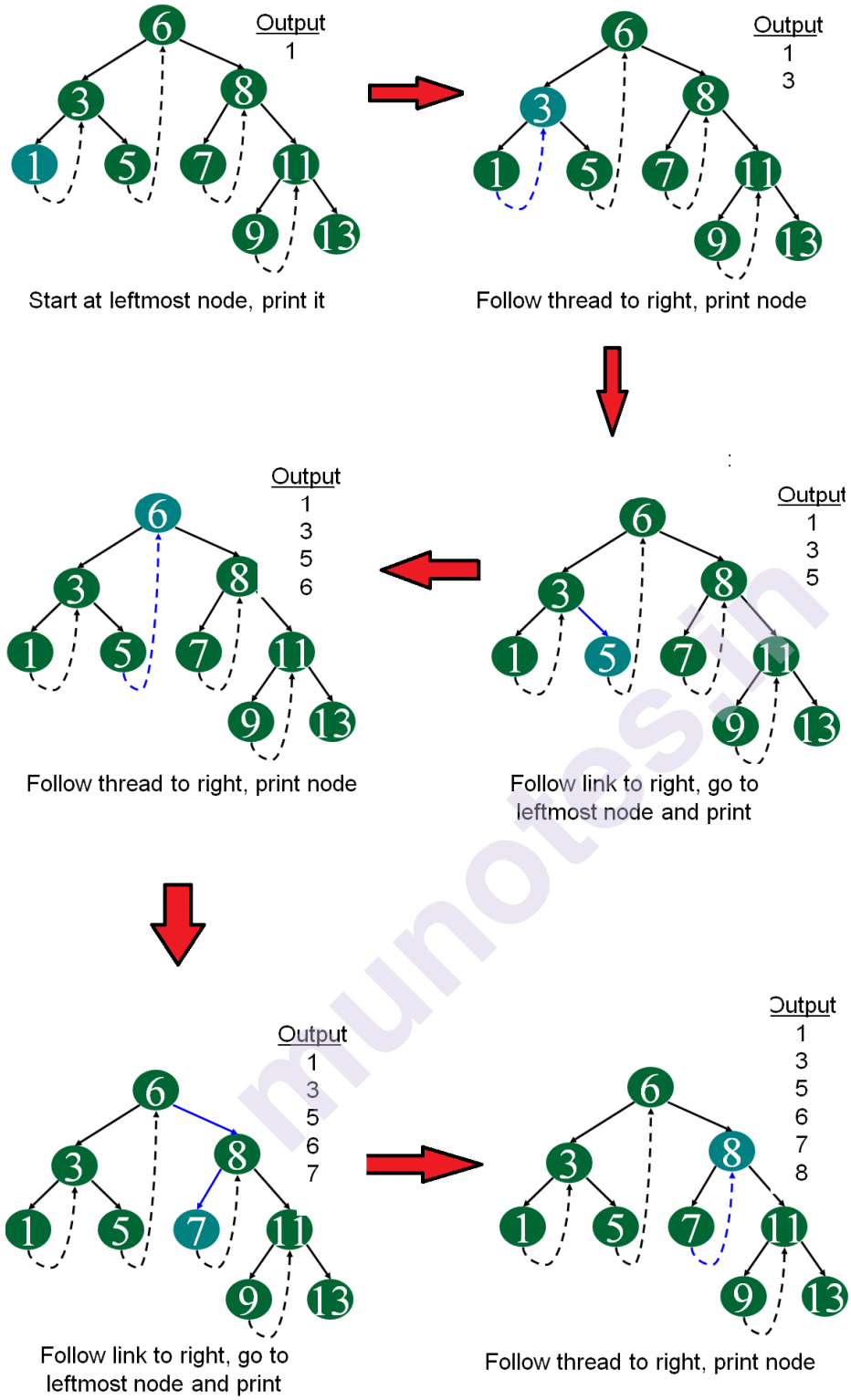
Java representation of a Threaded Node

Following is Java representation of a single-threaded node.

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Traversal using Threads Following is code for inorder traversal in a threaded binary tree.

Following diagram demonstrates inorder order traversal using threads.



continue same way for remaining node.....

Advantages of Threaded Binary Tree

- In this Tree it enables linear traversal of elements.
- It eliminates the use of stack as it perform linear traversal, so save memory.

- Enables to find parent node without explicit use of parent pointer
- Threaded tree give forward and backward traversal of nodes by in-order fashion
- Nodes contain pointers to in-order predecessor and successor
- For a given node, we can easily find inorder predecessor and successor. So, searching is much more easier.
- In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.
- The threads are pointing to successor and predecessor nodes. This makes us to obtain predecessor and successor node of any node quickly.
- There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

Disadvantages of Threaded Binary Tree

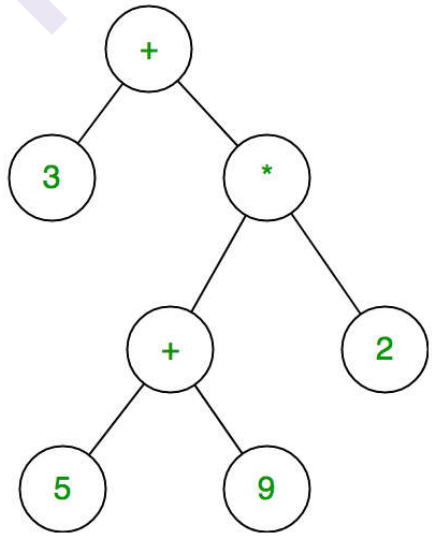
- Every node in threaded binary tree need extra information(extra memory) to indicate whether its left or right node indicated its child nodes or its inorder predecessor or successor. So, the node consumes extra memory to implement.
- Insertion and deletion are way more complex and time consuming than the normal one since both threads and ordinary links need to be maintained.
- Implementing threads for every possible node is complicated.
- Increased complexity: Implementing a threaded binary tree requires more complex algorithms and data structures than a regular binary tree. This can make the code harder to read and debug.
- Extra memory usage: In some cases, the additional pointers used to thread the tree can use up more memory than a regular binary tree. This is especially true if the tree is not fully balanced, as threading a skewed tree can result in a large number of additional pointers.
- Limited flexibility: Threaded binary trees are specialized data structures that are optimized for specific types of traversal. While they can be more efficient than regular binary trees for these types of operations, they may not be as useful in other scenarios. For example, they cannot be easily modified (e.g. inserting or deleting nodes) without breaking the threading.
- Difficulty in parallelizing: It can be challenging to parallelize operations on a threaded binary tree, as the threading can introduce data dependencies that make it difficult to process nodes independently. This can limit the performance gains that can be achieved through parallelism.

Applications of threaded binary tree –

- Expression evaluation: Threaded binary trees can be used to evaluate arithmetic expressions in a way that avoids recursion or a stack. The tree can be constructed from the input expression, and then traversed in-order or pre-order to perform the evaluation.
- Database indexing: In a database, threaded binary trees can be used to index data based on a specific field (e.g. last name). The tree can be constructed with the indexed values as keys, and then traversed in-order to retrieve the data in sorted order.
- Symbol table management: In a compiler or interpreter, threaded binary trees can be used to store and manage symbol tables for variables and functions. The tree can be constructed with the symbols as keys, and then traversed in-order or pre-order to perform various operations on the symbol table.
- Disk-based data structures: Threaded binary trees can be used in disk-based data structures (e.g. B-trees) to improve performance. By threading the tree, it can be traversed in a way that minimizes disk seeks and improves locality of reference.
- Navigation of hierarchical data: In certain applications, threaded binary trees can be used to navigate hierarchical data structures, such as file systems or web site directories. The tree can be constructed from the hierarchical data, and then traversed in-order or pre-order to efficiently access the data in a specific order.

Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)

Evaluating the expression represented by an expression tree:

Let t be the expression tree

If t is not null then

```
If t.value is operand then
        Return  t.value
A = solve(t.left)
B = solve(t.right)
 // calculate applies operator 't.value'
// on A and B, and returns value
Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

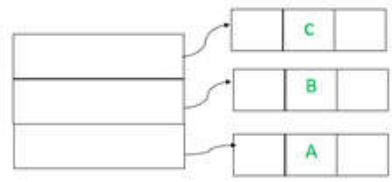
In the end, the only element of the stack will be the root of an expression tree.

Examples:

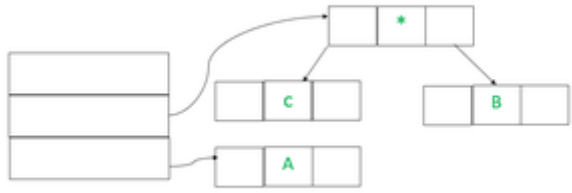
Input: A B C*+ D/

Output: A + B * C / D

The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



In the Next step, an operator '*' will going read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack



In the Next step, an operator '+' will read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.

Below is the implementation of the above approach:

```
import java.util.Stack;


class Node{
   char data;
   Node left,right;
   public Node(char data){
      this.data = data;
      left = right = null;
   }
}


public class Main {
   public static boolean isOperator(char ch){
      if(ch=='+' || ch=='-'|| ch=='*' || ch=='/' || ch=='^'){
         return true;
      }
      return false;
   }
```

```
public static Node expressionTree(String postfix){
   Stack<Node> st = new Stack<Node>();
   Node t1,t2,temp;

   for(int i=0;i<postfix.length();i++){
      if(!isOperator(postfix.charAt(i))){
         temp = new Node(postfix.charAt(i));
         st.push(temp);
      }
      else{
         temp = new Node(postfix.charAt(i));
          t1 = st.pop();
         t2 = st.pop();
          temp.left = t2;
         temp.right = t1;
          st.push(temp);
      }

   }
   temp = st.pop();
   return temp;
}
public static void inorder(Node root){
   if(root==null) return;

   inorder(root.left);
   System.out.print(root.data);
   inorder(root.right);
```

```
    }
    public static void main(String[] args) {
        String postfix = "ABC*+D/";

        Node r = expressionTree(postfix);
        inorder(r);
    }
}
```

Output

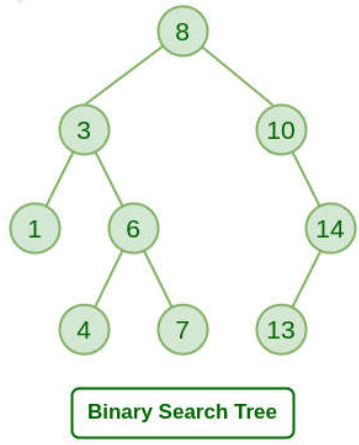
The Inorder Traversal of Expression Tree: A + B * C / D

Time complexity: O(n)
Auxiliary space: O(n)

3.5 WHAT IS BINARY SEARCH TREE?

Binary Search Tree is a node-based binary tree data structure which has the following properties:

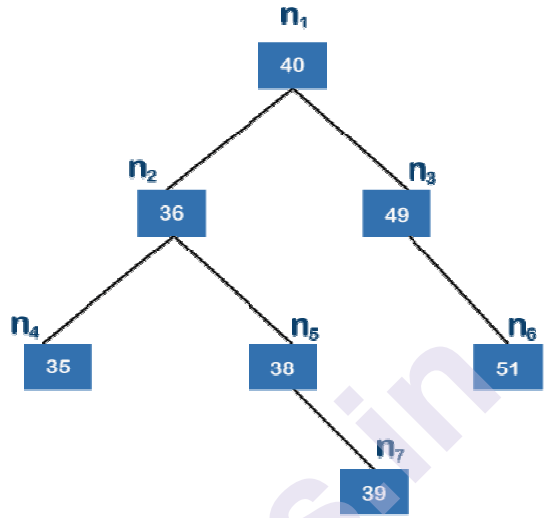
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

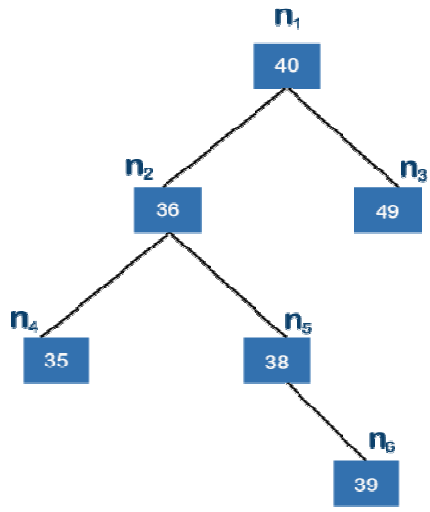
Balanced Binary Search Tree

A balanced binary tree is also known as height balanced tree. It is defined as binary tree in when the difference between the height of the left subtree and right subtree is not more than m, where m is usually equal to 3. The height of a tree is the number of edges on the longest path between the root node and the leaf node.

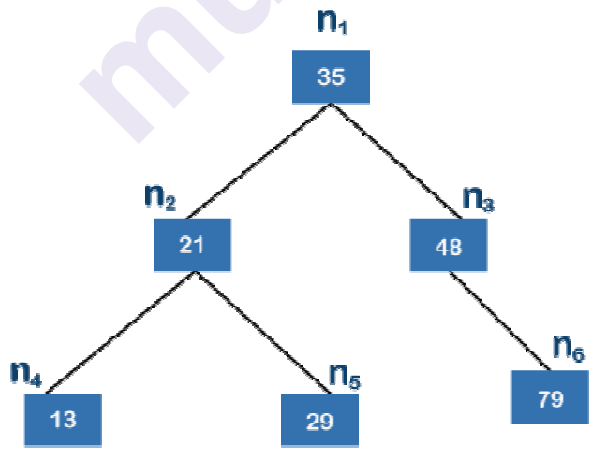


The above tree is a binary search tree. A binary search tree is a tree in which each node on the left side has a lower value than its parent node, and the node on the right side has a higher value than its parent node. In the above tree, n1 is a root node, and n4, n6, n7 are the leaf nodes. The n7 node is the farthest node from the root node. The n4 and n6 contain 2 edges and there exist three edges between the root node and n7 node. Since n7 is the farthest from the root node; therefore, the height of the above tree is 3.

Now we will see whether the above tree is balanced or not. The left subtree contains the nodes n2, n4, n5, and n7, while the right subtree contains the nodes n3 and n6. The left subtree has two leaf nodes, i.e., n4 and n7. There is only one edge between the node n2 and n4 and two edges between the nodes n7 and n2; therefore, node n7 is the farthest from the root node. The height of the left subtree is 2. The right subtree contains only one leaf node, i.e., n6, and has only one edge; therefore, the height of the right subtree is 3. The difference between the heights of the left subtree and right subtree is 3. Since we got the value 1 so we can say that the above tree is a height-balanced tree. This process of calculating the difference between the heights should be performed for each node like n2, n3, n4, n5, n6 and n7. When we process each node, then we will find that the value of k is not more than 1, so we can say that the above tree is a balanced binary tree.

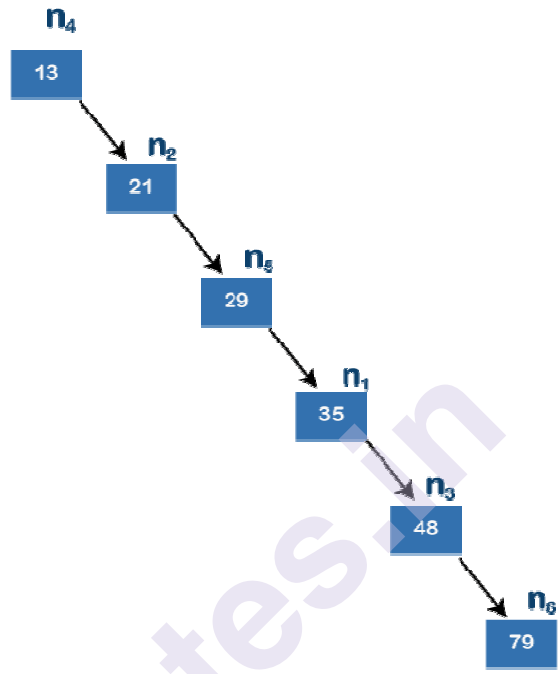


In the above tree, n6, n4, and n3 are the leaf nodes, where n6 is the farthest node from the root node. Three edges exist between the root node and the leaf node; therefore, the height of the above tree is 3. When we consider n1 as the root node, then the left subtree contains the nodes n2, n4, n5, and n6, while subtree contains the node n3. In the left subtree, n2 is a root node, and n4 and n6 are leaf nodes. Among n4 and n6 nodes, n6 is the farthest node from its root node, and n6 has two edges; therefore, the height of the left subtree is 2. The right subtree does have any child on its left and right; therefore, the height of the right subtree is 0. Since the height of the left subtree is 2 and the right subtree is 0, so the difference between the height of the left subtree and right subtree is 2. According to the definition, the difference between the height of left sub tree and the right subtree must not be greater than 3. In this case, the difference comes to be 2, which is greater than 1; therefore, the above binary tree is an unbalanced binary search tree.



The above tree is a binary search tree because all the left subtree nodes are smaller than its parent node and all the right subtree nodes are greater than its parent node. Suppose we want to want to find the value 79 in the above tree. First, we compare the value of node n1 with 79; since the value of 79 is not equal to 35 and it is greater than 35 so we move to the node n3, i.e.,

48. Since the value 79 is not equal to 48 and 79 is greater than 48, so we move to the right child of 48. The value of the right child of node 48 is 79 which is equal to the value to be searched. The number of hops required to search an element 79 is 2 and the maximum number of hops required to search any element is 2. The average case to search an element is O(logn).



The above tree is also a binary search tree because all the left subtree nodes are smaller than its parent node and all the right subtree nodes are greater than its parent node. Suppose we want to find the find the value 79 in the above tree. First, we compare the value 79 with a node n4, i.e., 13. Since the value 79 is greater than 13 so we move to the right child of node 13, i.e., n2 (21). The value of the node n2 is 21 which is smaller than 79, so we again move to the right of node 23. The value of right child of node 21 is 29. Since the value 79 is greater than 29 so we move to the right child of node 29. The value of right child of node 29 is 35 which is smaller than 79 so we move to the right child of node 35, i.e., 48. The value 79 is greater than 48, so we move to the right child of node 48. The value of right child node of 48 is 79 which is equal to the value to be searched. In this case, the number of hops required to search an element is 5. In this case, the worst case is O(n).

If the number of nodes increases, the formula used in the tree diagram1 is more efficient than the formula used in the tree diagram2. Suppose the number of nodes available in both above trees is 100,000. To search any element in a tree diagram2, the time taken is 100,000µs whereas the time taken to search an element in tree diagram is log(100,000) which is equal 16.6 µs. We can observe the enormous difference in time between above two trees. Therefore, we conclude that the balance binary tree provides searching more faster than linear tree data structure.

What is AVL Trees?

An AVL tree is a type of binary search tree. Named after it's inventors Adelson, Velskii, and Landis, AVL trees have the property of dynamic self-balancing in addition to all the other properties exhibited by binary search trees.

A BST is a data structure composed of nodes. It has the following guarantees:

1. Each tree has a root node (at the top)
2. The root node has zero, one, or two child nodes
3. Each child node has zero, one, or two child nodes
4. Each node has up to two children
5. For each node, its left descendants are less than the current node, which is less than the right descendants

AVL trees have an additional guarantee:

1. The difference between the depth of right and left sub-trees cannot be more than one. This difference is called the balance factor.

   In order to maintain this guarantee, an implementation of an AVL will include an algorithm to rebalance the tree when adding an additional element would upset this guarantee

AVL trees have a worst case lookup, insert, and delete time of O(log n), where n is the number of nodes in the tree. The worst case space complexity is O(n).

AVL Insertion Process

Insertion in an AVL tree is similar to insertion in a binary search tree. But after inserting and element, you need to fix the AVL properties using left or right rotations:

- If there is an imbalance in the left child's right sub-tree, perform a left-right rotation
- If there is an imbalance in the left child's left sub-tree, perform a right rotation
- If there is an imbalance in the right child's right sub-tree, perform a left rotation
- If there is an imbalance in the right child's left sub-tree, perform a right-left rotation

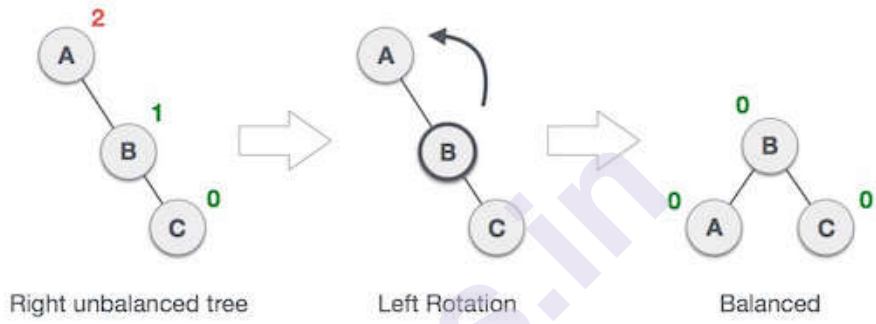
AVL Tree Rotations

In AVL trees, after each operation like insertion and deletion, the balance factor of every node needs to be checked. If every node satisfies the balance factor condition, then the operation can be concluded. Otherwise, the tree needs to be rebalanced using rotation operations.

There are four rotations and they are classified into two types:

Left Rotation (LL Rotation)

In left rotations, every node moves one position to left from the current position.



Right Rotation (RR Rotation)

In right rotations, every node moves one position to right from the current position.

Left-Right Rotation (LR Rotation)

Left-right rotations are a combination of a single left rotation followed by a single right rotation.

First, every node moves one position to the left, then one position to the right from the current position.

Right-Left Rotation (RL Rotation)

Right-left rotations are a combination of a single right rotation followed by a single left rotation.

First, every node moves one position to the right then, then one position to the left from the current position.

Application of AVL Trees

AVL trees are beneficial in cases like a database where insertions and deletions are not that frequent, but you frequently check for entries.

Python Functions

Define the class and initialize the nodes

```
class avl_Node(object):
   def __init__(self, value):
      self.value = value
      self.leaf = None
      self.root = None
      self.height = 1
```

Define a function to calculate height and Balance Factor.

```
def avl_Height(self, root):
      if not root:
         return 0
      return root.height


def avl_BalanceFactor(self, root):
     //base case for leaf nodes
      if not root:
         return 0
      //implementing the above mentioned formula
      return self.avl_Height(root.l) - self.avl_Height(root.r)
```

Define a function to find an empty node

```
def avl_MinValue(self, root):
      if root is None or root.left is None:
         return root
      return self.avl_MinValue(root.left)
```

Define a function to traverse the tree in a preorder way.

```
def preOrder(self, root):
      if not root:
         return
      print("{0} ".format(root.value), end=" ")
```

```
        self.preOrder(root.left)
        self.preOrder(root.right)
```

Define functions for rotations

```
def leftRotate(self, b):
        a = b.right
        T2 = a.left
        a.left = b
        b.right = T2
        b.height = 1 + max(self.avl_Height(b.left),
                           self.avl_Height(b.right))
        a.height = 1 + max(self.avl_Height(a.left),
                           self.avl_Height(a.right))
        return a
       def rightRotate(self, b):
        a = b.left
        T3 = a.right
        a.right = z
        b.left = T3
        b.height = 1 + max(self.avl_Height(b.left),
                           self.avl_Height(b.right))
        a.height = 1 + max(self.avl_Height(a.left),
                           self.avl_Height(a.right))
        return a
```

Define a function for inserting a node in AVL tree in Python

```
def insert_node(self, root, value):
         if not root:
            return avl_Node(value)
        elif value < root.value:
            root.left = self.insert_node(root.left, value)
```

```
    else:
        root.right = self.insert_node(root.right, value)

    root.height = 1 + max(self.avl_Height(root.left),
                          self.avl_Height(root.right))
     # Update the balance factor and balance the tree
    balanceFactor = self.avl_BalanceFactor(root)
    if balanceFactor > 1:
        if value < root.left.value:
            return self.rightRotate(root)
        else:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
     if balanceFactor < -1:
        if value > root.right.value:
            return self.leftRotate(root)
        else:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)
     return root
```

Define a function for deleting a node from the AVL tree in Python

```
def delete_node(self, root, value):
     # Find the node to be deleted and remove it
    if not root:
        return root
    elif value < root.value:
        root.left = self.delete_node(root.left, value)
    elif value > root.value:
        root.right = self.delete_node(root.right, value)
```

```
        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            temp = self.avl_MinValue(root.right)
            root.value = temp.key
            root.right = self.delete_node(root.right, temp.value)
        if root is None:
            return root
         # Update the balance factor of nodes
        root.height = 1 + max(self.avl_Height(root.left),
self.avl_Height(root.right))
        balanceFactor = self.avl_BalanceFactor(root)
         # Balance the tree
        if balanceFactor > 1:
            if self.avl_BalanceFactor(root.left) >= 0:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)
        if balanceFactor < -1:
            if self.avl_BalanceFactor(root.right) <= 0:
                return self.leftRotate(root)
```

```
else:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)
      return root
```

Complete Code for AVL Tree in Python

```
class avl_Node(object):
  def __init__(self, value):
      self.value = value
      self.left = None
      self.right = None
      self.height = 1
       class AVLTree(object):
          def insert_node(self, root, value):
       if not root:
          return avl_Node(value)
      elif value < root.value:
          root.left = self.insert_node(root.left, value)
      else:
          root.right = self.insert_node(root.right, value)
       root.height = 1 + max(self.avl_Height(root.left),
                             self.avl_Height(root.right))
       # Update the balance factor and balance the tree
      balanceFactor = self.avl_BalanceFactor(root)
      if balanceFactor > 1:
          if value < root.left.value:
              return self.rightRotate(root)
          else:
              root.left = self.leftRotate(root.left)
              return self.rightRotate(root)
```

```
        if balanceFactor < -1:
            if value > root.right.value:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)
        return root
    def avl_Height(self, root):
        if not root:
            return 0
        return root.height
    # Get balance factore of the node
    def avl_BalanceFactor(self, root):
        if not root:
            return 0
        return self.avl_Height(root.left) - self.avl_Height(root.right)
        def avl_MinValue(self, root):
        if root is None or root.left is None:
            return root
        return self.avl_MinValue(root.left)
    def preOrder(self, root):
        if not root:
            return
        print("{0} ".format(root.value), end=" ")
        self.preOrder(root.left)
        self.preOrder(root.right)
def leftRotate(self, b):
        a = b.right
        T2 = a.left
```

```
    a.left = b
    b.right = T2
    b.height = 1 + max(self.avl_Height(b.left),
                       self.avl_Height(b.right))
    a.height = 1 + max(self.avl_Height(a.left),
                       self.avl_Height(a.right))
    return a
  def rightRotate(self, b):
    a = b.left
    T3 = a.right
    a.right = b
    b.left = T3
    b.height = 1 + max(self.avl_Height(b.left),
                       self.avl_Height(b.right))
    a.height = 1 + max(self.avl_Height(a.left),
                       self.avl_Height(a.right))
    return a
        def delete_node(self, root, value):
     # Find the node to be deleted and remove it
    if not root:
        return root
    elif value < root.value:
        root.left = self.delete_node(root.left, value)
    elif value > root.value:
        root.right = self.delete_node(root.right, value)
    else:
        if root.left is None:
            temp = root.right
            root = None
```

```
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = self.avl_MinValue(root.right)
        root.value = temp.key
        root.right = self.delete_node(root.right, temp.value)
    if root is None:
        return root
     # Update the balance factor of nodes
    root.height = 1 + max(self.avl_Height(root.left),
self.avl_Height(root.right))
    balanceFactor = self.avl_BalanceFactor(root)
     # Balance the tree
    if balanceFactor > 1:
        if self.avl_BalanceFactor(root.left) >= 0:
            return self.rightRotate(root)
        else:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
    if balanceFactor < -1:
        if self.avl_BalanceFactor(root.right) <= 0:
            return self.leftRotate(root)
        else:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)
    return root
```

```
Tree = AVLTree()
root = None
root = Tree.insert_node(root,40)
root = Tree.insert_node(root,60)
root = Tree.insert_node(root,50)
root = Tree.insert_node(root,70)
 print("PREORDER")
Tree.preOrder(root)
```

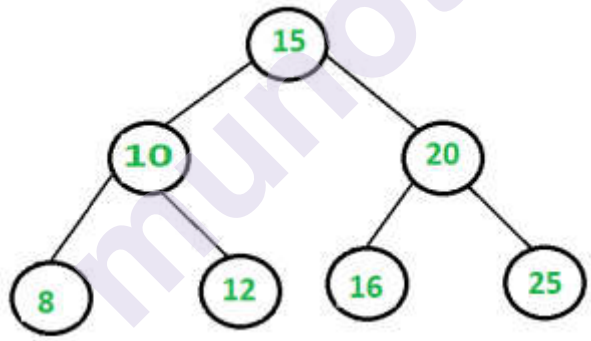
Output:

PREORDER

50 40 60 70

3.6 LET US SUM UP

Give an algorithm for finding the sum of all elements in a binary tree.



In the above binary tree sum = 106.

```
// Java Program to print sum of
// all the elements of a binary tree
class GFG
{
static class Node
{
        int key;
        Node left, right;
}

/* utility that allocates a new
```

```
Node with the given key */
static Node newNode(int key)
{
        Node node = new Node();
        node.key = key;
        node.left = node.right = null;
        return (node);
}

/* Function to find sum
of all the elements*/
static int addBT(Node root)
{
        if (root == null)
            return 0;
        return (root.key + addBT(root.left) +
                        addBT(root.right));
}

// Driver Code
public static void main(String args[])
{
        Node root = newNode(1);
        root.left = newNode(2);
        root.right = newNode(3);
        root.left.left = newNode(4);
        root.left.right = newNode(5);
        root.right.left = newNode(6);
        root.right.right = newNode(7);
        root.right.left.right = newNode(8);

        int sum = addBT(root);
        System.out.println("Sum of all the elements is: " + sum);
}
}
```

Output

```
Sum of all the elements is: 36
```

Time Complexity: O(N)
Auxiliary Space: O(1), but if we consider space due to the recursion call stack then it would be O(h), where h is the height of the Tree.

Method 2 – Another way to solve this problem is by using **Level Order Traversal.** Every time when a Node is deleted from the queue, add it to the sum variable.

Implementation:

- C++
- Java
- Python3
- C#
- Javascript

```
# Python3 Program to print sum of all
# the elements of a binary tree

# Binary Tree Node
class newNode:

    # Utility function to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Function to find sum of all the element
def sumBT(root):
    # sum variable to track the sum of
    # all variables.
    sum = 0

    q = []

    # Pushing the first level.
    q.append(root)

    # Pushing elements at each level from
    # the tree.
    while len(q) > 0:
        temp = q.pop(0)

        # After popping each element from queue
        # add its data to the sum variable.
        sum += temp.key

        if (temp.left != None):
            q.append(temp.left)
        if temp.right != None:
            q.append(temp.right)
```

```
    return sum


# Driver Code
if __name__ == '__main__':
    root = newNode(1)
    root.left = newNode(2)
    root.right = newNode(3)
    root.left.left = newNode(4)
    root.left.right = newNode(5)
    root.right.left = newNode(6)
    root.right.right = newNode(7)
    root.right.left.right = newNode(8)

    print("Sum of all elements in the binary tree is: ",
sumBT(root))

# This code is contributed by
# Abhijeet Kumar(abhijeet19403)
```

Output

```
Sum of all elements in the binary tree is: 36
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Method 3: Using Morris traversal:

The Morris Traversal algorithm is an in-order tree traversal algorithm that does not require the use of a stack or recursion, and uses only constant extra memory. The basic idea behind the Morris Traversal algorithm is to use the right child pointers of the nodes to create a temporary link back to the node's parent, so that we can easily traverse the tree without using any extra memory.

Follow the steps below to implement the above idea:

1. Initialize a variable sum to 0 to keep track of the sum of all nodes in the binary tree.
2. Initialize a pointer root to the root of the binary tree.
3. While root is not null, perform the following steps:
   If the left child of root is null, add the value of root to sum, and move to the right child of root.
   If the left child of root is not null, find the rightmost node of the left subtree of root and create a temporary link back to root.
   Move to the left child of root.

4. After the traversal is complete, return sum.

Below is the implementation of the above approach:

- C++

```
// C++ code to implement the morris traversal approach
#include <iostream>
using namespace std;

// Definition of a binary tree node
struct Node {
    int val;
    Node* left;
    Node* right;
    Node(int v)
        : val(v)
        , left(nullptr)
        , right(nullptr)
    {
    }
};

// Morris Traversal function to find the sum of all nodes in
// a binary tree
long int sumBT(Node* root)
{
    long int sum = 0;
    while (root != nullptr) {
        if (root->left
            == nullptr) { // If there is no left child, add
                          // the value of the current node
                          // to the sum and move to the
                          // right child
            sum += root->val;
            root = root->right;
        }
        else { // If there is a left child
            Node* prev = root->left;
            while (
                prev->right != nullptr
                && prev->right
                       != root) // Find the rightmost node
                                // in the left subtree of
                                // the current node
                prev = prev->right;
            if (prev->right
                == nullptr) { // If the right child of the
```

```
                                // rightmost node is null, set
                                // it to the current node and
                                // move to the left child
                    prev->right = root;
                    root = root->left;
                }
                else { // If the right child of the rightmost
                       // node is the current node, set it to
                       // null, add the value of the current
                       // node to the sum and move to the right
                       // child
                    prev->right = nullptr;
                    sum += root->val;
                    root = root->right;
                }
            }
        }
        return sum;
}

// Driver code
int main()
{
        // Example binary tree:    1
        //                        / \
        //                       2   3
        //                      / \
        //                     4   5
        Node* root = new Node(1);
        root->left = new Node(2);
        root->right = new Node(3);
        root->left->left = new Node(4);
        root->left->right = new Node(5);

        // Find the sum of all nodes in the binary tree
        long int sum = sumBT(root);
        cout << "Sum of all nodes in the binary tree is " << sum
             << endl;

        return 0;
}
// This code is contributed by Veerendra_Singh_Rajpoot
```

Output

```
Sum of all nodes in the binary tree is 15
```

Time Complexity: O(n) , Because of all the nodes are traversing only once.
Auxiliary Space: O(1)

3.7 LIST OF REFERENCES

http://www.cs.bu.edu/teaching/c/tree/binary/
http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses

3.8 UNIT END EXERCISE

1. What is Inorder Tree Traversal?
2. Write a shorte note on Preorder Tree Traversal?
3. What is Postorder Tree Traversal?
4. Check if two binary trees are identical or not??
5. Print bottom view of a binary tree?
6. Print top view of a binary tree?
7. In-place convert a binary tree to its sum tree?
8. Determine whether the given binary tree nodes are cousins of each other?
9. Print cousins of a given node in a binary tree?
10. Check if a binary tree is a sum tree or not?
11. Combinations of words formed by replacing given numbers with corresponding alphabets?
12. Determine whether a binary tree is a subtree of another binary tree?
13. Find the diameter of a binary tree?
14. Check if a binary tree is symmetric or not?
15. Convert a binary tree to its mirror?
16. Determine if a binary tree can be converted to another by doing any number of swaps of children?
17. Find the Lowest Common Ancestor (LCA) of two nodes in a binary tree?
18. Print all paths from the root to leaf nodes of a binary tree?
19. Find ancestors of a given node in a binary tree?
20. Find distance between given pairs of nodes in a binary tree?

21. Find the diagonal sum of a binary tree?
22. Sink nodes containing zero to the bottom of a binary tree?
23. Convert a binary tree to a full tree by removing half nodes?
24. Truncate a binary tree to remove nodes that lie on a path having a sum less than `k`?
25. Find maximum sum root to leaf path in a binary tree?
26. Check if a binary tree is height-balanced or not?
27. Convert binary tree to Left-child right-sibling binary tree?
28. Print all paths from leaf to root node of a binary tree?
29. Iteratively print the leaf to root path for every leaf node in a binary tree?
30. Build a binary tree from a parent array?
31. Find all nodes at a given distance from leaf nodes in a binary tree?
32. Count all subtrees having the same value of nodes in a binary tree?
33. Find the maximum difference between a node and its descendants in a binary tree?
34. Find the maximum sum path between two leaves in a binary tree?
35. Construct a binary tree from inorder and preorder traversal?
36. Construct a binary tree from inorder and postorder traversals?
37. Construct a binary tree from inorder and level order sequence?
38. Construct a full binary tree from the preorder sequence with leaf node information?
39. Construct a full binary tree from a preorder and postorder sequence?
40. Find postorder traversal of a binary tree from its inorder and preorder sequence?
41. Set next pointer to the inorder successor of all nodes in a binary tree?
42. Find preorder traversal of a binary tree from its inorder and postorder sequence?
43. Find the difference between the sum of all nodes present at odd and even levels in a binary tree?
44. Clone a binary tree with random pointers?
45. Threaded Binary Tree — Overview and Implementation?

46. Determine if a binary tree satisfies the height-balanced property of a red–black tree?
47. Construct an ancestor matrix from a binary tree?
48. Find all possible binary trees having the same inorder traversal?
49. Perform boundary traversal on a binary tree ?
50. Check if each node of a binary tree has exactly one child?
51. Evaluate a Binary Expression Tree?
52. Construction of an expression tree?
53. Fix children-sum property in a binary tree?
54. Maximum path sum in a binary tree?
55. Create a mirror of an m–ary tree?
56. Print a two-dimensional view of a binary tree?
57. Construct a binary tree from an ancestor matrix?
58. Determine whether a given binary tree is a BST or not?
59. Find inorder successor for the given key in a BST?
60. Fix a binary tree that is only one swap away from becoming a BST?
61. Find the size of the largest BST in a binary tree?
62. Print binary tree structure with its contents in C++?
63. Maximum Independent Set Problem?Huffman Coding Compression Algorithm
64. Construct a Cartesian tree from an inorder traversal?
65. Calculate the height of a binary tree with leaf nodes forming a circular doubly linked list?
66. Link nodes present in each level of a binary tree in the form of a linked list?
67. Convert a ternary tree to a doubly-linked list?
68. Extract leaves of a binary tree into a doubly-linked list?
69. Find the vertical sum of a binary tree?
70. In-place convert a binary tree to a doubly-linked list?
71. Check whether the leaf traversal of given binary trees is the same or not?

72. Efficiently print all nodes between two given levels in a binary tree?
73. Calculate the height of a binary tree?
74. Delete a binary tree?
75. Level order traversal of a binary tree?
76. Spiral order traversal of a binary tree
77. Reverse level order traversal of a binary tree?
78. Print all nodes of a perfect binary tree in a specific order?
79. Print left view of a binary tree?
80. Find the next node at the same level as the given node in a binary tree?
81. Check if a binary tree is a complete binary tree or not?
82. Print diagonal traversal of a binary tree?
83. Print corner nodes of every level in a binary tree?
84. Invert Binary Tree?
85. Convert a binary tree into a doubly-linked list in spiral order?
86. Check if a binary tree is a min-heap or not?
87. Invert alternate levels of a perfect binary tree?
88. Perform vertical traversal of a binary tree?
89. Compute the maximum number of nodes at any level in a binary tree?
90. Print right view of a binary tree?
91. Find the minimum depth of a binary tree?
92. Depth-First Search (DFS) vs Breadth-First Search (BFS)?
93. Print nodes of a binary tree in vertical order?

4

GRAPH ALGORITHMS

Unit Structure :

4.0 OBJECTIVES

- To learn what a graph is and how it is used.
- To implement the **graph** abstract data type using multiple internal representations.
- To see how graphs can be used to solve a wide variety of problems

In this chapter we will study graphs. Graphs are a more general structure than the trees we studied in the last chapter; in fact you can think of a tree as a special kind of graph. Graphs can be used to represent many interesting things about our world, including systems of roads, airline flights from city to city, how the Internet is connected, or even the

sequence of classes you must take to complete a major in computer science. We will see in this chapter that once we have a good representation for a problem, we can use some standard graph algorithms to solve what otherwise might seem to be a very difficult problem.

While it is relatively easy for humans to look at a road map and understand the relationships between different places, a computer has no such knowledge. However, we can also think of a road map as a graph. When we do so we can have our computer do interesting things for us. If you have ever used one of the Internet map sites, you know that a computer can find the shortest, quickest, or easiest path from one place to another.

As a student of computer science you may wonder about the courses you must take in order to get a major. A graph is good way to represent the prerequisites and other interdependencies among courses. Figure 1. shows another graph. This one represents the courses and the order in which they must be taken to complete a major in computer science at Luther College.

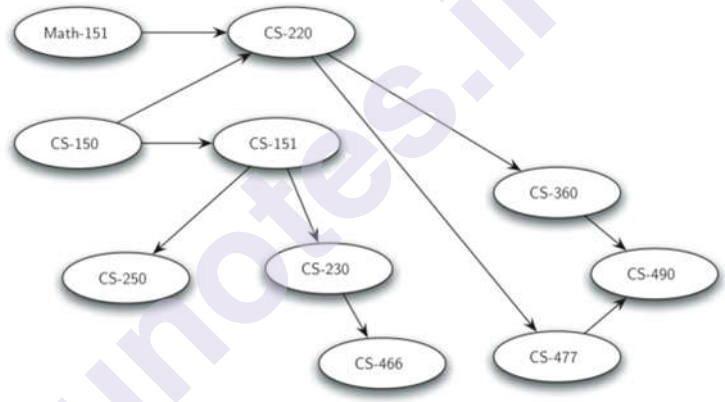


Figure 1: Prerequisites for a Computer Science Major

What is the purpose of graph algorithms?

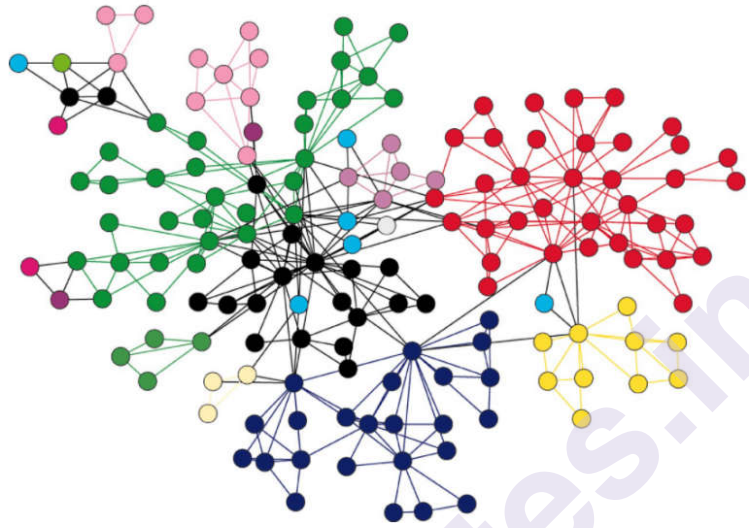
Graph algorithms are used to solve the problems of representing graphs as networks like airline flights, how the Internet is connected, or social network connectivity on Facebook. They are also popular in NLP and machine learning to form networks.

The course is designed to develop skills to design and analyze simple linear and non linear data structures.

4.1 WHAT IS GRAPHS?

Graph Algorithms: Introduction, Glossary, Applications of Graphs, Graph

Graph Algorithms



In this article, you would be learning a brief explanation of some of the most used graph algorithms, which have massive applications in today's words. Graphs cover most high-level data structure techniques that one experiences while implementing them and to know which graph algorithm is best for the moment effectively is what you would be learning here. First, let's get a clear idea from the very basics about graphs.

4.2 WHAT IS A GRAPH?

A graph is a **unique data structure** in programming that consists of finite sets of nodes or vertices and a set of edges that connect these vertices to them. At this moment, adjacent vertices can be called those vertices that are connected to the same edge with each other. In simple terms, a graph is a visual representation of vertices and edges sharing some connection or relationship. Although there are plenty of graph algorithms that you might have been familiar with, only some of them are put to use. The reason for this is simple as the standard graph algorithms are designed in such a way to solve millions of problems with just a few lines of logically coded technique. To some extent, one perfect algorithm is solely optimized to achieve such efficient results.

4.2.1 Types of Graphs

There are various types of graph algorithms that you would be looking at in this article but before that, let's look at some types of terms to imply the fundamental variations between them.

Order: Order defines the total number of vertices present in the graph.

Size: Size defines the number of edges present in the graph.

Self-loop: It is the edges that are connected from a vertex to itself.

Isolated vertex: It is the vertex that is not connected to any other vertices in the graph.

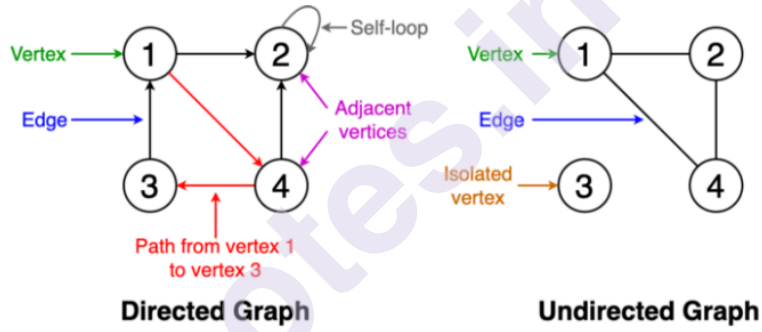
Vertex degree: It is defined as the number of edges incident to a vertex in a graph.

Weighted graph: A graph having value or weight of vertices.

Unweighted graph: A graph having no value or weight of vertices.

Directed graph: A graph having a direction indicator.

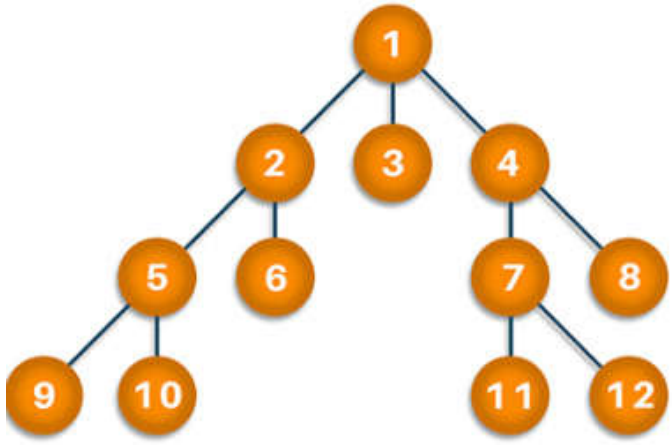
Undirected graph: A graph where no directions are defined.



Let's now carry forward the main discussion and learn about different types of graph algorithms.

4.2.2 Breadth-First Search

Traversing or searching is one of the most used operations that are undertaken while working on graphs. Therefore, in **breadth-first-search** (BFS), you start at a particular vertex, and the algorithm tries to visit all the neighbors at the given depth before moving on to the next level of traversal of vertices. Unlike trees, graphs may contain cyclic paths where the first and last vertices are remarkably the same always. Thus, in BFS, you need to keep note of all the track of the vertices you are visiting. To implement such an order, you use a queue data structure which First-in, First-out approach. To understand this, see the image given below.



BREADTH FIRST SEARCH

Algorithm

1. Start putting anyone vertices from the graph at the back of the queue.
2. First, move the front queue item and add it to the list of the visited node.
3. Next, create nodes of the adjacent vertex of that list and add them which have not been visited yet.
4. Keep repeating steps two and three until the queue is found to be empty.

Pseudocode

1. Set all nodes to "not visited";
2. q = new Queue();
3. q.enqueue(initial node);
4. while (q ? empty) do
5. {
6. x = q.dequeue();
7. if (x has not been visited)
8. {
9. visited[x] = true; // Visit node x !
10.
11. for (every edge (x, y) /* we are using all edges ! */)
12. if (y has not been visited)
13. q.enqueue(y); // Use the edge (x,y) !!!

14. }

15. }

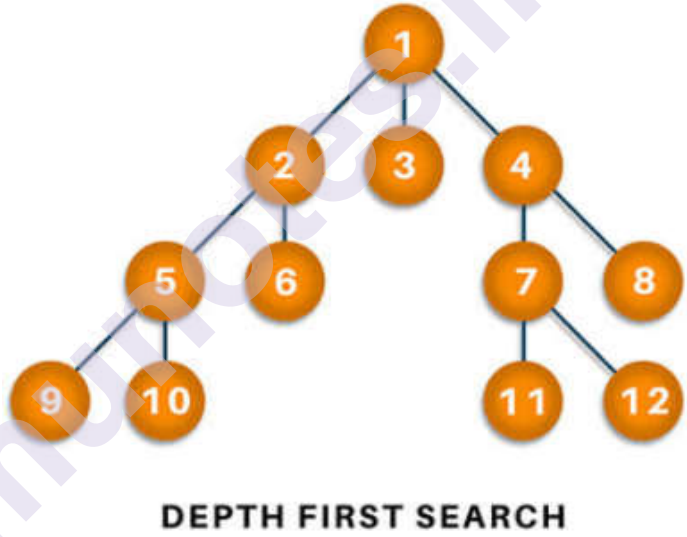
Complexity: 0(V+E) where V is vertices and E is edges.

Applications

BFS algorithm has various applications. For example, it is used to determine the **shortest path** and **minimum spanning tree.** It is also used in web crawlers to creates web page indexes. It is also used as powering search engines on social media networks and helps to find out peer-to-peer networks in BitTorrent.

4.2.4 Depth-first search

In depth-first-search (DFS), you start by particularly from the vertex and explore as much as you along all the branches before backtracking. In DFS, it is essential to keep note of the tracks of visited nodes, and for this, you use stack data structure.



DEPTH FIRST SEARCH

Algorithm

1. Start by putting one of the vertexes of the graph on the stack's top.
2. Put the top item of the stack and add it to the visited vertex list.
3. Create a list of all the adjacent nodes of the vertex and then add those nodes to the unvisited at the top of the stack.
4. Keep repeating steps 2 and 3, and the stack becomes empty.

Pseudocode

1. DFS(G,v) (v is the vertex where the search starts)
2. Stack S := {}; (start with an empty stack)
3. for each vertex u, set visited[u] := false;

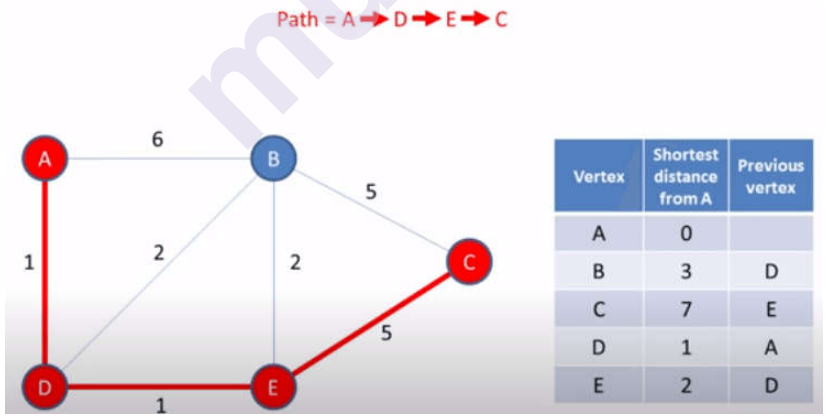
4. push S, v;
5. while (S is not empty) do
6. u := pop S;
7. if (not visited[u]) then
8. visited[u] := true;
9. for each unvisited neighbour w of uu
10. push S, w;
11. end if
12. end while
13. END DFS()

Applications

DFS finds its application when it comes to finding paths between two vertices and detecting cycles. Also, topological sorting can be done using the DFS algorithm easily. DFS is also used for one-solution puzzles.

Dijkstra's shortest path algorithm

Dijkstra's shortest path algorithm works to find the minor path from one vertex to another. The sum of the vertex should be such that their sum of weights that have been traveled should output minimum. The shortest path algorithm is a highly curated algorithm that works on the concept of receiving efficiency as much as possible. Consider the below diagram.



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Algorithm

1. Set all the vertices to infinity, excluding the source vertex.
2. Push the source in the form (distance, vertex) and put it in the min-priority queue.
3. From the priority, queue pop out the minimum distant vertex from the source vertex.

4. Update the distance after popping out the minimum distant vertex and calculate the vertex distance using (vertex distance + weight < following vertex distance).
5. If you find that the visited vertex is popped, move ahead without using it.
6. Apply the steps until the priority queue is found to be empty.

Pseudocode

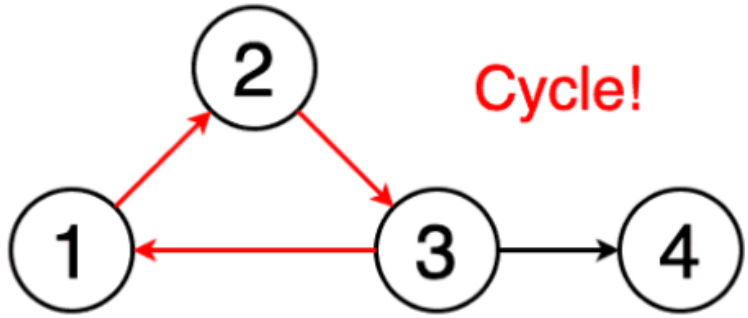
```
1.  function dijkstra(G, S)
2.  for each vertex V in G
3.  distance[V] <- infinite
4.  previous[V] <- NULL
5.  If V != S, add V to Priority Queue Q
6.  distance[S] <- 0
7.  while Q IS NOT EMPTY
8.  U <- Extract MIN from Q
9.  for each unvisited neighbour V of U
10. tempDistance <- distance[U] + edge_weight(U, V)
11. if tempDistance < distance[V]
12. distance[V] <- tempDistance
13. previous[V] <- U
14. return distance[], previous[]
```

Applications

Dijkstra's shortest path algorithm is used in finding the distance of travel from one location to another, like Google Maps or Apple Maps. In addition, it is highly used in networking to outlay min-delay path problems and abstract machines to identify choices to reach specific goals like the number game or move to win a match.

Cycle detection

A cycle is defined as a path in graph algorithms where the first and last vertices are usually considered. For example, if you start from a vertex and travel along a random path, you might reach the exact point where you eventually started. Hence, this forms a chain or cyclic algorithm to cover along with all the nodes present on traversing. Therefore, cycle detection is based on detecting this kind of cycle. Consider the below image.



Pseudocode

```
1. Brent's Cycle Algorithm Example
2. def brent(f, x0):
3. # main phase: search successive powers of two
4. power = lam = 1
5. tortoise = x0
6. hare = f(x0)  # f(x0) is the element/node next to x0.
7. while tortoise != hare:
8. if power == lam:  # time to start a new power of two?
9. tortoise = hare
10. power *= 2
11. lam = 0
12. hare = f(hare)
13. lam += 1
14. # Find the position of the first repetition of length ?
15. tortoise = hare = x0
16. for i in range(lam):
17. # range(lam) produces a list with the values 0, 1, ... , lam-1
18. hare = f(hare)
19. # The distance between the hare and tortoise is now ?.
20. # Next, the hare and tortoise move at same speed until they agree
21. mu = 0
22. while tortoise != hare:
```

23. tortoise = f(tortoise)

24. hare = f(hare)

25. mu += 1

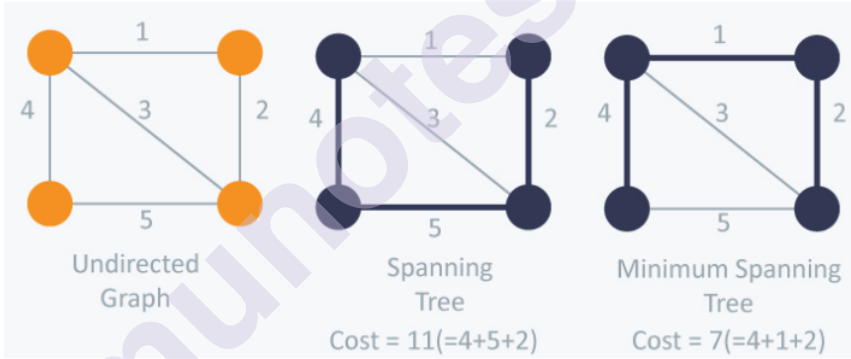
26. return lam, mu

Applications

Cyclic algorithms are used in message-based distributed systems and large-scale cluster processing systems. It is also mainly used to detect deadlocks in the concurrent system and various cryptographic applications where the keys are used to manage the messages with encrypted values.

4.2.4 Minimum Spaning Trees

A minimum spanning is defined as a subset of edges of a graph having no cycles and is well connected with all the vertices so that the minimum sum is availed through the edge weights. It solely depends on the cost of the spanning tree and the minimum span or least distance the vertex covers. There can be many minimum spanning trees depending on the edge weight and various other factors.



Pseudocode

1. Prim's Algorithm Example
2. ReachSet = {0};
3. UnReachSet = {1, 2, ..., N-1};
4. SpanningTree = {};
5. while (UnReachSet ? empty)
6. {
7. Find edge e = (x, y) such that:
8. 1. x ? ReachSet
9. 2. y ? UnReachSet
10. 3. e has smallest cost

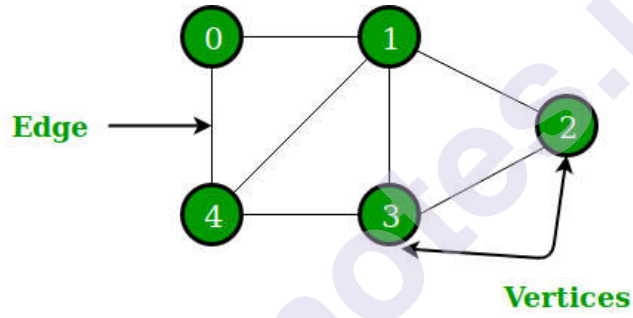
```
11.  SpanningTreeSpanningTree = SpanningTree ? {e};
12.  ReachSetReachSet   = ReachSet ? {y};
13.  UnReachSetUnReachSet = UnReachSet - {y};
14.  }
```

Applications

Minimum spanning tree finds its application in the network design and is popularly used in **traveling salesman** problems in a data structure. It can also be used to find the minimum-cost weighted perfect matching and multi-terminal minimum cut problems. MST also finds its application in the field of image and handwriting recognition and cluster analysis.

Applications of Graph Data Structure

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.



•

Thus the development of algorithms to handle graphs is of major interest in the field of computer science.

Three Applications of Graphs in the area of computer engineering:

1. The applications of graph split broadly into three categories:

a) First, analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist.

b) Second, analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it.

c) Third, analysis of dynamic properties of network. Map of a country can be represented using a graph. Road network, Air network or rail network can be represented using a graph. Connection among routers in a communication network can be represented using a graph. Routing of a packet between two communicating nodes can be done through the shortest path.

2. Graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist and the edges represent migration paths, or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease.

3. Different activities of a project can be represented using a graph. This graph can be useful in project scheduling.

Applications of Graphs to Solve Real-World Problems

Graphs are widely used in many fields. Let us take some real-life examples and solve them through graphs.

Ten Applications of Graphs

Since graphs are powerful abstractions, they can be essential in modelling data. Given below are some instances for the applications of graphs.

1. **Social network graphs:** Graphs show who knows who, how they communicate with one other, and how they impact each other, as well as other social structure relationships. The Facebook graph showing who follows whom or who sends friend invitations to whom is an example. These can be used to figure out how information spreads, how topics get popular, how communities grow, and even who could be a good fit for that person.

2. **Graphs in epidemiology:** Nowadays, the spread of disease is widespread worldwide. Analysing such graphs has become an essential component in understanding and controlling the spread of diseases.

3. **Utility graphs:** The power grid, the Internet, and the water network are graphs with vertices representing connection points and edges representing the cables or pipes that connect them. Understanding the reliability of such utilities under failure or assault and lowering the costs of building infrastructure that meets desired needs requires analysing the features of these graphs.

4. **Transportation networks:** In transportation network graphs, vertices are intersections in road networks, and edges are the road segments that connect them. Vertices are stops in public transit networks, and edges are the links that connect them. Many map systems, such as Google Maps, Bing Maps, and Apple iOS 6 maps, employ such networks to determine the optimal paths between sites. They're also utilised to figure out how traffic flows and how traffic lights work.

5. **Constraint graphs:** Graphs are often used to represent constraints among items. For example, the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can

be modelled as a graph where the cells are vertices and edges are placed between overlapping cells.

6. **Dependence graphs:** Graphs can be used to represent dependencies or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimise the total time or cost to completion while abiding by the dependences.

7. **Document link graphs:** The most well-known example is the web's link graph, in which each web page is a vertex, and a directed edge connects each hyperlink. Link graphs are used to assess the relevancy of online pages, the most acceptable sources of information, and good link sites, among other things.

8. **Finite element meshes:** Many models of physical systems in engineering entail partitioning space into discrete pieces, such as the flow of air over a vehicle or plane wing, the spread of earthquakes through the ground, or the structural vibrations of a building. The elements and the connections between them make up a graph known as a finite element mesh.

9. **Robot planning:** The edges represent possible transitions between states, whereas the vertices represent various states for the robot. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used to plan paths for autonomous vehicles, for example.

10. **Graphs in compilers:** Graphs are used extensively in compilers. They can be used for type inference, so-called data flow analysis, register allocation, and many other purposes. They are also used in specialised compilers, such as query optimisation in database languages.

Graph Representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

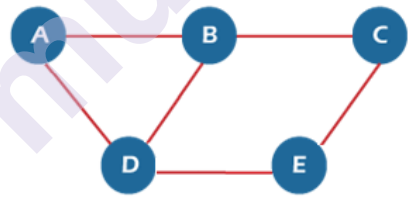
$a_{ij} = 1$ {if there is a path exists from V_i to V_j}

$a_{ij} = 0$ {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

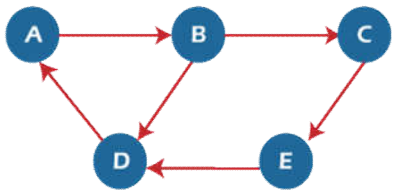
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

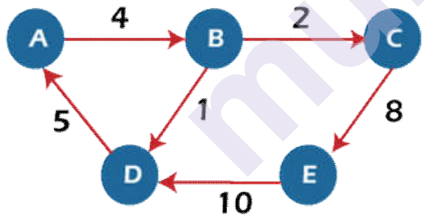
Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.

ADVERTISING



weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

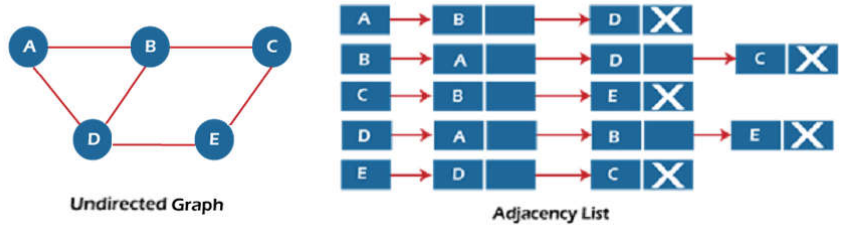
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

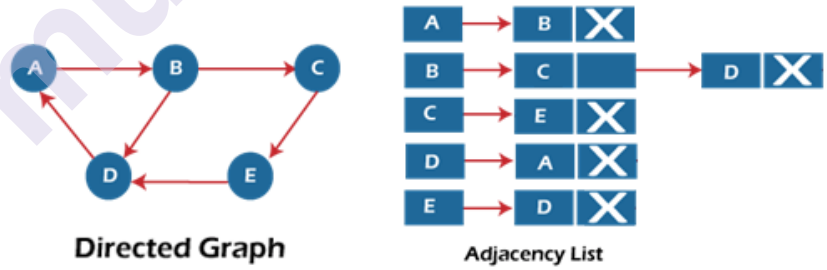


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

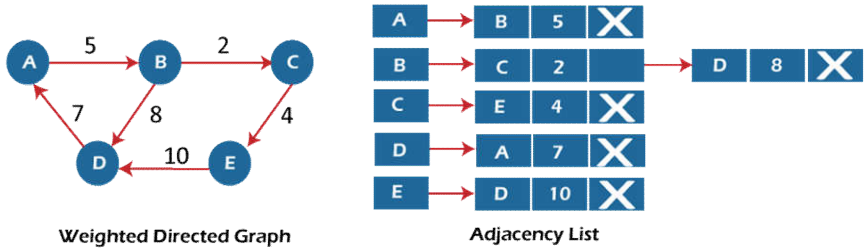
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

```
1.  /* Adjacency Matrix representation of an undirected graph in C */
2.
3.  #include <stdio.h>
4.  #define V 4 /* number of vertices in the graph */
5.
6.  /* function to initialize the matrix to zero */
7.  void init(int arr[][V]) {
8.  int i, j;
9.  for (i = 0; i < V; i++)
10. for (j = 0; j < V; j++)
11. arr[i][j] = 0;
12. }
13.
14. /* function to add edges to the graph */
15. void insertEdge(int arr[][V], int i, int j) {
```

```
16.  arr[i][j] = 1;
17.  arr[j][i] = 1;
18.  }
19.
20.  /* function to print the matrix elements */
21.  void printAdjMatrix(int arr[][V]) {
22.  int i, j;
23.  for (i = 0; i < V; i++) {
24.  printf("%d: ", i);
25.  for (j = 0; j < V; j++) {
26.  printf("%d ", arr[i][j]);
27.  }
28.  printf("\n");
29.  }
30.  }
31.
32.  int main() {
33.  int adjMatrix[V][V];
34.
35.  init(adjMatrix);
36.  insertEdge(adjMatrix, 0, 1);
37.  insertEdge(adjMatrix, 0, 2);
38.  insertEdge(adjMatrix, 1, 2);
39.  insertEdge(adjMatrix, 2, 0);
40.  insertEdge(adjMatrix, 2, 3);
41.
42.  printAdjMatrix(adjMatrix);
43.
44.  return 0;
```

```
45.  }
```

Output:

After the execution of the above code, the output will be -

```
0: 0 1 1 0
1: 1 0 1 0
2: 1 1 0 1
3: 0 0 1 0
```

Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

```
1.  /* Adjacency list representation of a graph in C */
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.
5.  /* structure to represent a node of adjacency list */
6.  struct AdjNode {
7.  int dest;
8.  struct AdjNode* next;
9.  };
10.
11.  /* structure to represent an adjacency list */
12.  struct AdjList {
13.  struct AdjNode* head;
14.  };
15.
16.  /* structure to represent the graph */
17.  struct Graph {
18.   int V; /*number of vertices in the graph*/
19.  struct AdjList* array;
```

```
20.  };
21.
22.
23.  struct AdjNode* newAdjNode(int dest)
24.  {
25.  struct AdjNode* newNode = (struct AdjNode*)malloc(sizeof(struct
AdjNode));
26.  newNode->dest = dest;
27.  newNode->next = NULL;
28.  return newNode;
29.  }
30.
31.  struct Graph* createGraph(int V)
32.  {
33.  struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
34.  graph->V = V;
35.      graph-
>array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
36.
37.      /* Initialize each adjacency list as empty by making head as NUL
L */
38.      int i;
39.      for (i = 0; i < V; ++i)
40.          graph->array[i].head = NULL;
41.      return graph;
42.  }
43.
44.  /* function to add an edge to an undirected graph */
45.  void addEdge(struct Graph* graph, int src, int dest)
46.  {
```

```
47.     /* Add an edge from src to dest. The node is added at the beginnin
g */
48.     struct AdjNode* check = NULL;
49.     struct AdjNode* newNode = newAdjNode(dest);
50.
51.     if (graph->array[src].head == NULL) {
52.        newNode->next = graph->array[src].head;
53.        graph->array[src].head = newNode;
54.     }
55.     else {
56.
57.        check = graph->array[src].head;
58.        while (check->next != NULL) {
59.           check = check->next;
60.        }
61.        // graph->array[src].head = newNode;
62.        check->next = newNode;
63.     }
64.
65.     /* Since graph is undirected, add an edge from dest to src also */
66.     newNode = newAdjNode(src);
67.     if (graph->array[dest].head == NULL) {
68.        newNode->next = graph->array[dest].head;
69.        graph->array[dest].head = newNode;
70.     }
71.     else {
72.        check = graph->array[dest].head;
73.        while (check->next != NULL) {
74.           check = check->next;
75.        }
```

```
76.         check->next = newNode;
77.     }
78. }
79. /* function to print the adjacency list representation of graph*/
80. void print(struct Graph* graph)
81. {
82.     int v;
83.     for (v = 0; v < graph->V; ++v) {
84.         struct AdjNode* pCrawl = graph->array[v].head;
85.         printf("\n The Adjacency list of vertex %d is: \n head ", v);
86.         while (pCrawl) {
87.             printf("-> %d", pCrawl->dest);
88.             pCrawl = pCrawl->next;
89.         }
90.         printf("\n");
91.     }
92. }
93.
94. int main()
95. {
96.
97.     int V = 4;
98.     struct Graph* g = createGraph(V);
99.     addEdge(g, 0, 1);
100.    addEdge(g, 0, 3);
101.    addEdge(g, 1, 2);
102.    addEdge(g, 1, 3);
103.    addEdge(g, 2, 4);
104.    addEdge(g, 2, 3);
```

```
105.    addEdge(g, 3, 4);
106.    print(g);
107.    return 0;
108. }
```

Output:

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -

```
The Adjacency list of vertex 0 is:
head -> 1-> 3

The Adjacency list of vertex 1 is:
head -> 0-> 2-> 3

The Adjacency list of vertex 2 is:
head -> 1-> 4-> 3

The Adjacency list of vertex 3 is:
head -> 0-> 1-> 2-> 4
```


Conclusion

Here, we have seen the description of graph representation using the adjacency matrix and adjacency list. We have also seen their implementations in C programming language.

So, that's all about the article. Hope, it will be helpful and informative to you.

Graph traversal

In this section we will see what is a graph data structure, and the traversal algorithms of it.

The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})

The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

Algorithm

bfs(vertices, start)

Input: The list of vertices, and the start vertex.

Output: Traverse all of the nodes, if the graph is connected.

```
Begin
  define an empty queue que
  at first mark all nodes status as unvisited
  add the start vertex into the que
  while que is not empty, do
    delete item from que and set to u
    display the vertex u
    for all vertices 1 adjacent with u, do
      if vertices[i] is unvisited, then
        mark vertices[i] as temporarily visited
        add v into the queue
      mark
    done
    mark u as completely visited
  done
End
```

Depth First Search (DFS)

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Algorithm

dfs(vertices, start)

Input: The list of all vertices, and the start node.

Output: Traverse all nodes in the graph.

```
Begin
  initially make the state to unvisited for all nodes
  push start into the stack
  while stack is not empty, do
    pop element from stack and set to u
    display the node u
    if u is not visited, then
      mark u as visited
      for all nodes i connected to u, do
        if ith vertex is unvisited, then
          push ith vertex into the stack
          mark ith vertex as visited
      done
  done
End
```

4.6 TOPOLOGICAL SORT

The **topological sort** algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to.

Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

> Can a graph have more than one valid topological ordering? Yep! In the example above, [1, 3, 2, 4, 5] works too.

Cyclic Graphs

Look at this directed graph with a cycle:

The cycle creates an impossible set of constraints—B has to be before and after D in the ordering.

As a rule, **cyclic graphs don't have valid topological orderings.**

The Algorithm

How can we produce a topological ordering for this directed graph?

Well, let's focus on the first node in the topological ordering. That node can't have any incoming directed edges; it must have an indegree ↴ of zero.

Why?

Because if it had incoming directed edges, then the nodes pointing to it would have to come first.

So, we'll find a node with an indegree of zero and add it to the topological ordering.

That covers the first node in our topological ordering. What about the next one?

Once a node is added to the topological ordering, we can take the node, and its outgoing edges, out of the graph.

Then, we can repeat our earlier approach: look for any node with an indegree of zero and add it to the ordering.

This is a common algorithm design pattern:

1. Figure out how to get the first thing.
2. Remove the first thing from the problem.
3. Repeat.

Note: this isn't the only way to produce a topological ordering.

Implementation

We'll use the strategy we outlined above:

1. Identify a node with no incoming edges.
2. Add that node to the ordering.
3. Remove it from the graph.
4. Repeat.

We'll keep looping until there aren't any more nodes with indegree zero. This could happen for two reasons:

- There are no nodes left. We've taken all of them out of the graph and added them to the topological ordering.
- There are some nodes left, but they all have incoming edges. This means the graph has a cycle, and no topological ordering exists.

One small tweak. Instead of actually removing the nodes from the graph (and destroying our input!), we'll use a hash map to track each node's indegree. When we add a node to the topological ordering, we'll decrement the indegree of that node's neighbors, representing that those nodes have one fewer incoming edges.

Let's code it up!

```
deftopological_sort(digraph):
# digraph is a dictionary:
#   key: a node
# value: a set of adjacent neighboring nodes
# construct a dictionary mapping nodes to their
# indegrees
   indegrees ={node :0for node in digraph}
for node in digraph:
forneighborin digraph[node]:
        indegrees[neighbor]+=1
# track nodes with no incoming edges
nodes_with_no_incoming_edges=[]
for node in digraph:
if indegrees[node]==0:
nodes_with_no_incoming_edges.append(node)
# initially, no nodes in our ordering
topological_ordering=[]
# as long as there are nodes with no incoming edges
# that can be added to the ordering
```

```
whilelen(nodes_with_no_incoming_edges)>0:
# add one of those nodes to the ordering
        node =nodes_with_no_incoming_edges.pop()
topological_ordering.append(node)
# decrement the indegree of that node's neighbors
forneighborin digraph[node]:
            indegrees[neighbor]-=1
if indegrees[neighbor]==0:
nodes_with_no_incoming_edges.append(neighbor)
# we've run out of nodes with no incoming edges
# did we add all the nodes or find a cycle?
iflen(topological_ordering)==len(digraph):
returntopological_ordering# got them all
else:
raiseException("Graph has a cycle! No topological ordering exists.")
```

Time and Space Complexity

Breaking the algorithm into chunks, we:

- Determine the indegree for each node. This is O(M) time (where M is the number of edges), since this involves looking at each directed edge in the graph once.
- Find nodes with no incoming edges. This is a simple loop through all the nodes with some number of constant-time appends. O(N) time (where N is the number of nodes).
- Add nodes until we run out of nodes with no incoming edges. This loop could run once for every node—O(N) times. In the body, we:
  - Do two constant-time operations on an array to add a node to the topological ordering.
  - Decrement the indegree for each neighbor of the node we added. Over the entire algorithm, we'll end up doing exactly one decrement for each edge, making this step O(M) time overall.
- Check if we included all nodes or found a cycle. This is a fast (1)O(1) comparison.

All together, the time complexity is O(M+N).

That's the fastest time we can expect, since we'll have to look at all the nodes and edges at least once.

What about space complexity? Here are the data structures we created:

- indegrees—this has one entry for each node in the graph, so it's O(N) space.
- nodesWithNoIncomingEdges—in a graph with no edges, this would start out containing every node, so it's O(N) space in the worst case.
- Topological Ordering—in a graph with no cycles, this will eventually have every node. O(N) space.

All in all, we have three structures and they're all O(N) space. **Overall space complexity: O(N)**.

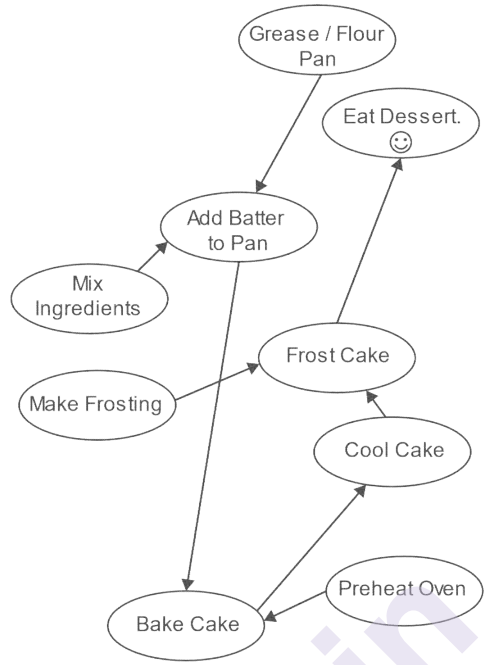
This is the best space complexity we can expect, since we must allocate a return array which costs O(N) space itself.

Uses

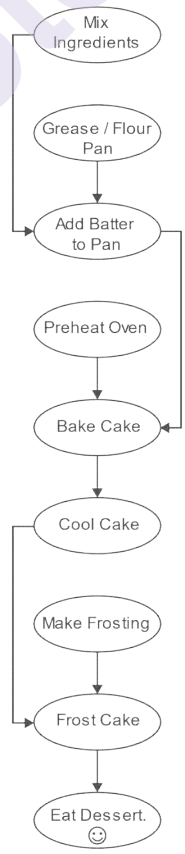
The most common use for topological sort is ordering steps of a process where some the steps depend on each other.

As an example, when making chocolate bundt cake,

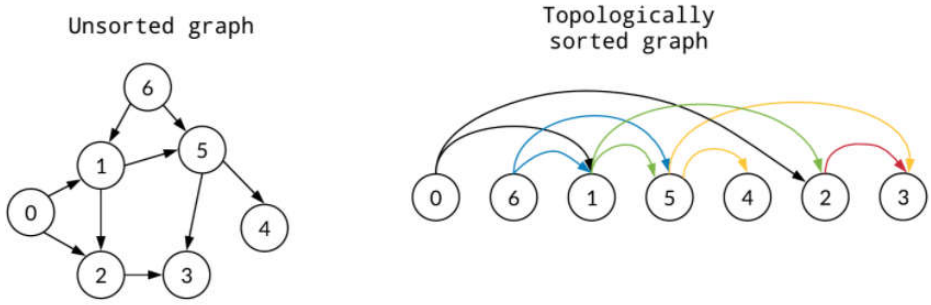
- The ingredients have to be mixed before going in the bundt pan.
- The bundt pan has to be greased and floured before the batter can be poured in.
- The oven has to be preheated before the cake can bake.
- The cake has to be baked before it cools.
- The cake has to cool before it can be iced.
- This process can be represented as a directed graph. Each step is a node, and we'll add directed edges between nodes to represent that one step has to be done before another.



- Once we have our dependencies represented using a directed graph, we can use topological sort to provide a valid ordering to tackle all the steps.



Topological sorting of a graph follows the algorithm of ordering the vertices linearly so that each directed graph having vertex ordering ensures that the vertex comes before it. Users can understand it more accurately by looking at the sample image given below.



In the above example, you can visualize the ordering of the unsorted graph and topologically sorted graph. The topologically sorted graph ensures to sort vertex that comes in the pathway.

Pseudocode

1. topological_sort(N, adj[N][N])
2. T = []
3. visited = []
4. in_degree = []
5. for i = 0 to N
6. in_degree[i] = visited[i] = 0
7. for i = 0 to N
8. for j = 0 to N
9. if adj[i][j] is TRUE
10. in_degree[j] = in_degree[j] + 1
11. for i = 0 to N
12. if in_degree[i] is 0
13. enqueue(Queue, i)
14. visited[i] = TRUE
15. while Queue is not Empty
16. vertex = get_front(Queue)
17. dequeue(Queue)
18. T.append(vertex)

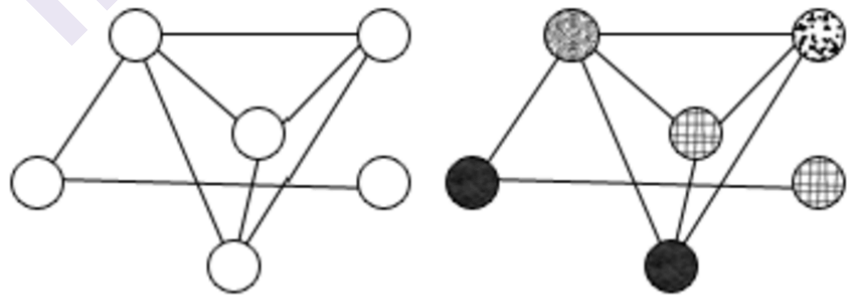
```
19.  for j = 0 to N
20.   if adj[vertex][j] is TRUE and visited[j] is FALSE
21.  in_degree[j] = in_degree[j] - 1
22.                          if in_degree[j] is 0
23.                               enqueue(Queue, j)
24.                               visited[j] = TRUE
25.          return T
```

Application

Topological sorting covers the room for application in Kahn's and DFS algorithms. In real-life applications, topological sorting is used in scheduling instructions and serialization of data. It is also popularly used to determine the tasks that are to be compiled and used to resolve dependencies in linkers.

Graph coloring

Graph coloring algorithms follow the approach of assigning colors to the elements present in the graph under certain conditions. The conditions are based on the techniques or algorithms. Hence, vertex coloring is a commonly used coloring technique followed here. First, in this method, you try to color the vertex using k color, ensuring that two adjacent vertexes should not have the same color. Other method includes face coloring and edge coloring. Both of these methods should also ensure that no edge or face should be inconsequent color. The coloring of the graph is determined by knowing the chromatic number, which is also the smaller number of colors needed. Consider the below image to understand how it works.



Pseudocode

```
1.  #include <iostream>
2.  #include <list>
3.  using namespace std;
```

```
4.  // A class that represents an undirected graph
5.  class Graph
6.  {
7.      int V;    // No. of vertices
8.      list<int> *adj;    // A dynamic array of adjacency lists
9.  public:
10.     // Constructor and destructor
11.     Graph(int V)   { this->VV = V; adj = new list<int>[V]; }
12.     ~Graph()       { delete [] adj; }
13.     // function to add an edge to graph
14.     void addEdge(int v, int w);
15.     // Prints greedy coloring of the vertices
16.     void greedyColoring();
17. };
18. void Graph::addEdge(int v, int w)
19. {
20.     adj[v].push_back(w);
21.     adj[w].push_back(v);  // Note: the graph is undirected
22. }
23. // Assigns colors (starting from 0) to all vertices and prints
24. // the assignment of colors
25. void Graph::greedyColoring()
26. {
27.     int result[V];
28.     // Assign the first color to first vertex
29.     result[0]  = 0;
30.     // Initialize remaining V-1 vertices as unassigned
31.     for (int u = 1; u < V; u++)
32.         result[u] = -1;  // no color is assigned to u
```

```
33.  // A temporary array to store the available colors. True
34.  // value of available[cr] would mean that the color cr is
35.  // assigned to one of its adjacent vertices
36.  bool available[V];
37.  for (int cr = 0; cr < V; cr++)
38.     available[cr] = false;
39.  // Assign colors to remaining V-1 vertices
40.  for (int u = 1; u < V; u++)
41.  {
42.     // Process all adjacent vertices and flag their colors
43.     // as unavailable
44.     list<int>::iterator i;
45.     for (i = adj[u].begin(); i != adj[u].end(); ++i)
46.        if (result[*i] != -1)
47.           available[result[*i]] = true;
48.     // Find the first available color
49.     int cr;
50.     for (cr = 0; cr < V; cr++)
51.        if (available[cr] == false)
52.           break;
53.     result[u] = cr; // Assign the found color
54.     // Reset the values back to false for the next iteration
55.     for (i = adj[u].begin(); i != adj[u].end(); ++i)
56.        if (result[*i] != -1)
57.           available[result[*i]] = false;
58.  }
59.  // print the result
60.  for (int u = 0; u < V; u++)
61.     cout << "Vertex " << u << " --->  Color "
```

```
62.             << result[u] << endl;
63.  }
64.  // Driver program to test above function
65.  int main()
66.  {
67.      Graph g1(5);
68.      g1.addEdge(0, 1);
69.      g1.addEdge(0, 2);
70.      g1.addEdge(1, 2);
71.      g1.addEdge(1, 3);
72.      g1.addEdge(2, 3);
73.      g1.addEdge(3, 4);
74.      cout << "Coloring of graph 1 \n";
75.      g1.greedyColoring();
76.      Graph g2(5);
77.      g2.addEdge(0, 1);
78.      g2.addEdge(0, 2);
79.      g2.addEdge(1, 2);
80.      g2.addEdge(1, 4);
81.      g2.addEdge(2, 4);
82.      g2.addEdge(4, 3);
83.      cout << "\nColoring of graph 2 \n";
84.      g2.greedyColoring();
85.      return 0;
86.  }
```

Application

Graph coloring has vast applications in data structures as well as in solving real-life problems. For example, it is used in timetable scheduling and assigning radio frequencies for mobile. It is also used in Sudoko and to check if the given graph is bipartite. Graph coloring can also be used in geographical maps to mark countries and states in different colors.

Maximum flow

The maximum flow algorithm is usually treated as a problem-solving algorithm where the graph is modeled like a network flow infrastructure. Hence, the maximum flow is determined by finding the path of the flow that has the **maximum flow rate.** The maximum flow rate is determined by augmenting paths which is the total flow-based out of source node equal to the flow in the sink node. Below is the illustration for the same.

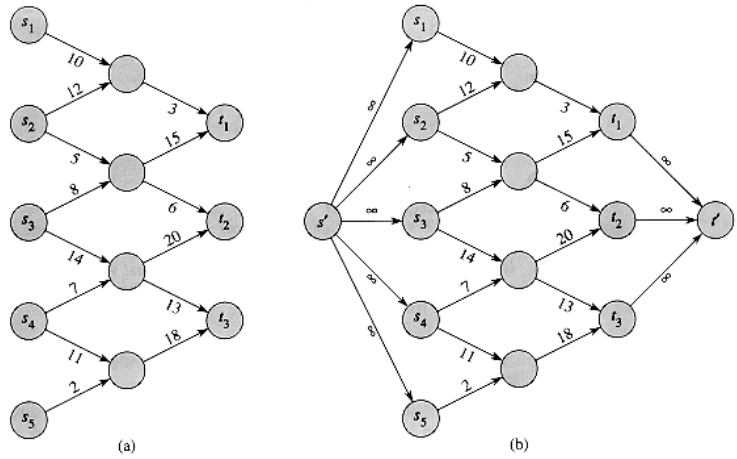
```
1.    function: DinicMaxFlow(Graph G,Node S,Node T):
2.      Initialize flow in all edges to 0, F = 0
3.      Construct level graph
4.      while (there exists an augmenting path in level graph):
5.        find blocking flow f in level graph
6.        FF = F + f
7.        Update level graph
8.      return F
```

Applications

Like you, the maximum flow problem covers applications of popular algorithms like the Ford-Fulkerson algorithm, Edmonds-Karp algorithm, and Dinic's algorithm, like you saw in the pseudocode given above. In real life, it finds its applications in scheduling crews in flights and image segmentation for foreground and background. It is also used in games like basketball, where the score is set to a maximum estimated value having the current division leader.

Matching

A matching algorithm or technique in the graph is defined as the edges that no common vertices at all. Matching can be termed maximum matching if the most significant number of edges possibly matches with as many vertices as possible. It follows a specific approach for determining full matches, as shown in the below image.



Applications

Matching is used in an algorithm like the Hopcroft-Karp algorithm and Blossom algorithm. It can also be used to solve problems using a Hungarian algorithm that covers concepts of matching. In real-life examples, matching can be used resource allocation and travel optimization and some problems like stable marriage and vertex cover problem.

Conclusion

In this article, you came across plenty of graph coloring algorithms and techniques that find their day-to-day applications in all instances of real life. You learned how to implement them according to situations, and hence the pseudo code helped you process the information strategically and efficiently. Graph algorithms are considered an essential aspect in the field confined not only to solve problems using data structures but also in general tasks like Google Maps and Apple Maps. However, a beginner might find it hard to implement Graph algorithms because of their complex nature. Hence, it is highly recommended to go through this article since it covers everything from scratch.

Shortest Path Algorithms

Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph. This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

What is Dijkstra's Algorithm?

Dijkstra's algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist **Edsger W. Dijkstra** in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited

vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

- **What is Dijkstra's Algorithm?**
- **Need of Dijkstra's Algorithm (Purpose and Use-Cases)**
- **Basics of Dijkstra's Algorithm**
- **Basics requirements for Implementation of Dijkstra's Algorithm**
- **Can Dijkstra's algorithm work on both directed and undirected graphs?**
- **Algorithm:**
- **How Does Dijkstra's Algorithm Works?**
- **Pseudo Code for Dijkstra's Algorithm**
- **Ways to Implement Dijkstra's Algorithm**
- **Implementation of Dijkstra's Algorithm:**
- **1. Dijkstra's Shortest Path Algorithm using priority_queue (Heap)**
- **2. Dijkstra shortest path algorithm using Prim's Algorithm**
- **3. Dijkstra's shortest path with minimum edges**
- **4. Dijkstra's shortest path algorithm using Set**
- **Complexity Analysis of Dijkstra's Algorithm**
- **Dijkstra's Algorithm vs Other Algorithms**
- **Problems Related to Dijkstra's Algorithm**

Need for Dijkstra's Algorithm (Purpose and Use-Cases)

The need for Dijkstra's algorithm arises in many applications where finding the shortest path between two points is crucial.

For example, It can be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting point and the Destination (as explained in How does Google Maps work?)

Basic Characterstics of Dijkstra's Algorithm

- Below are the basic steps of how Dijkstra's algorithm works:

  So Basically, Dijkstra's algorithm starts at the node source node we choose and then it analyzes the graph condition and its paths to find the optimal shortest distance between the given node and all other nodes in the graph.

- Dijkstra's algorithm keeps track of the currently known shortest distance from each node to the source node and updates the value after it finds the optimal path once the algorithm finds the shortest path between the source node and destination node then the specific node is marked as visited.

Basics requirements for Implementation of Dijkstra's Algorithm

1. **Graph:** Dijkstra's Algorithm can be implemented on any graph but it works best with a weighted Directed Graph with non-negative edge weights and the graph should be represented as a set of vertices and edges.

2. **Source Vertex:** Dijkstra's Algorithm requires a source node which is starting point for the search.

3. **Destination vertex:** Dijkstra's algorithm may be modified to terminate the search once a specific destination vertex is reached.

4. **Non-Negative Edges:** Dijkstra's algorithm works only on graphs that have positive weights this is because during the process the weights of the edge have to be added to find the shortest path. If there is a negative weight in the graph then the algorithm will not work correctly. Once a node has been marked as visited the current path to that node is marked as the shortest path to reach that node.

Can Dijkstra's algorithm work on both Directed and Undirected graphs?

Yes, Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

- **In a directed graph,** each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.

- **In an undirected graph,** the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

Algorithm for Dijkstra's Algorithm:

- Mark the source node with a current distance of 0 and the rest with infinity.

- Set the non-visited node with the smallest current distance as the current node.

- For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.

- Mark the current node 1 as visited.

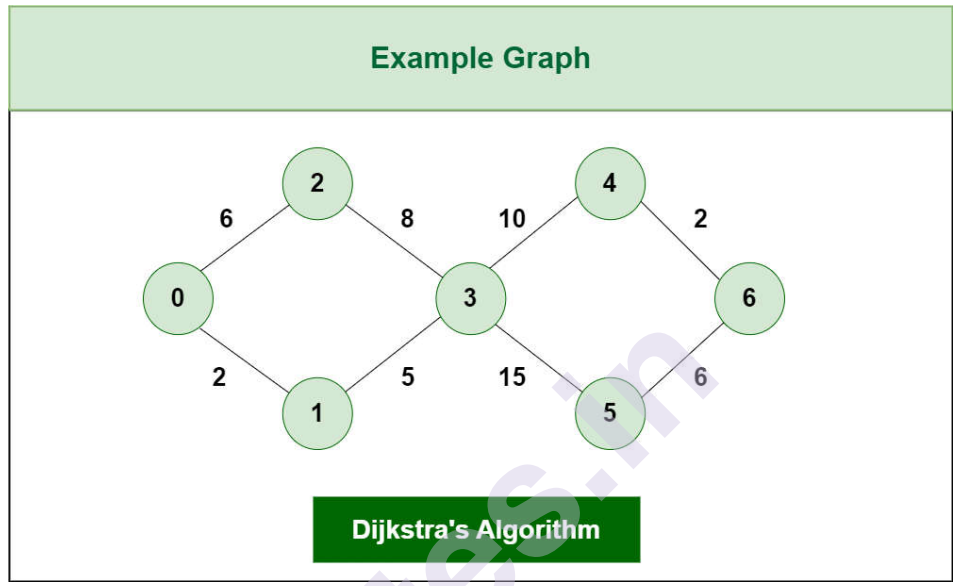
- Go to step 2 if there are any nodes are unvisited.

How does Dijkstra's Algorithm works?

Let's see how Dijkstra's Algorithm works with an example given below:

Dijkstra's Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.

Consider the below graph:



Dijkstra's Algorithm

The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

For this graph, we will assume that the weight of the edges represents the distance between two nodes.

As, we can see we have the shortest path from,Node 0 to Node 1, from
Node 0 to Node 2, from
Node 0 to Node 3, from
Node 0 to Node 4, from
Node 0 to Node 6.

Initially we have a set of resources given below :

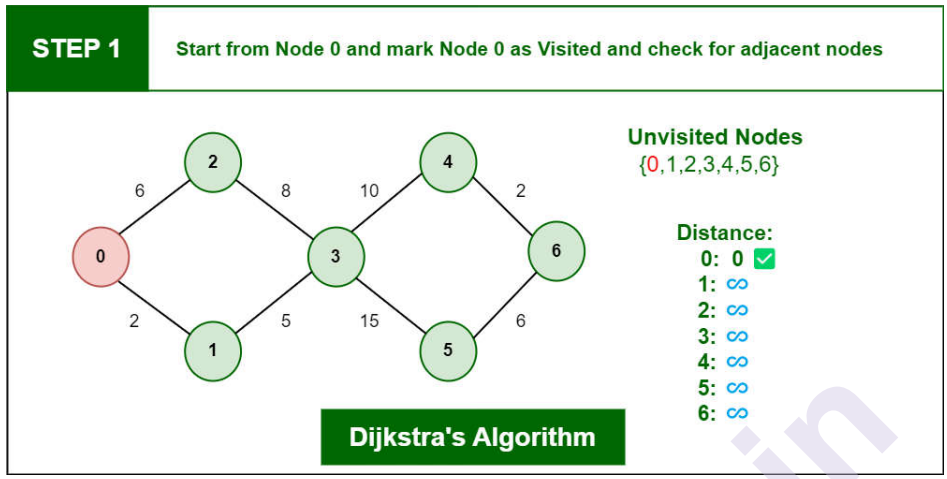
- The Distance from the source node to itself is 0. In this example the source node is 0.
- The distance from the source node to all other node is unknown so we mark all of them as infinity.

Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.

- we'll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.

- Algorithm will complete when all the nodes marked as visited and the distance between them added to the path. **Unvisited Nodes:- 0 1 2 3 4 5 6.**

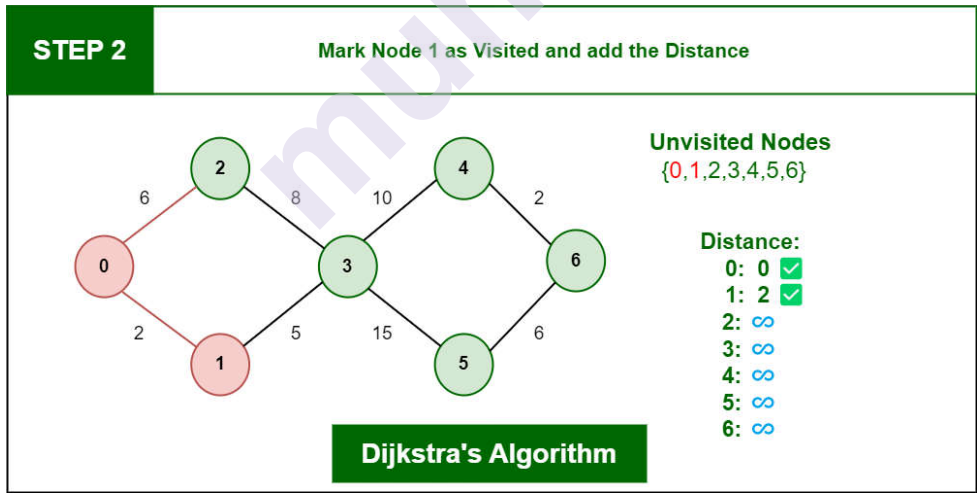
Step 1: Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.



Dijkstra's Algorithm

Step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step **Node 1** is Minimum distance adjacent Node, so marked it as visited and add up the distance.

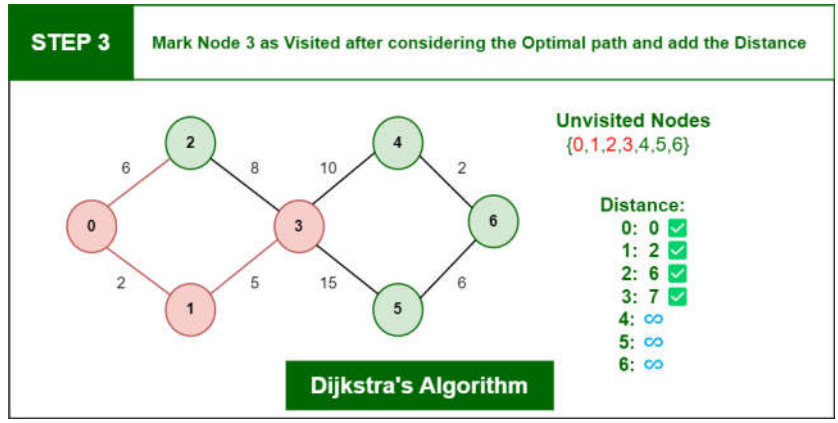
Distance: Node 0 -> Node 1 = 2



Dijkstra's Algorithm

Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:

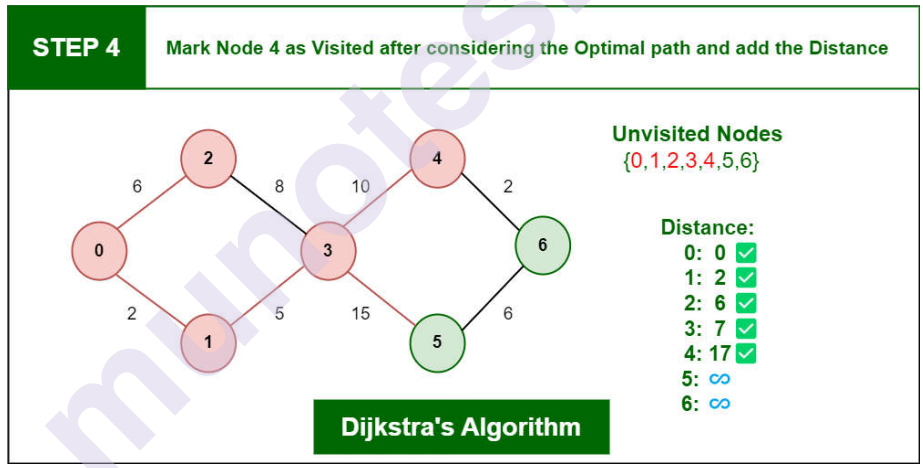
Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7



Dijkstra's Algorithm

Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step **Node 4** is Minimum distance adjacent Node, so marked it as visited and add up the distance.

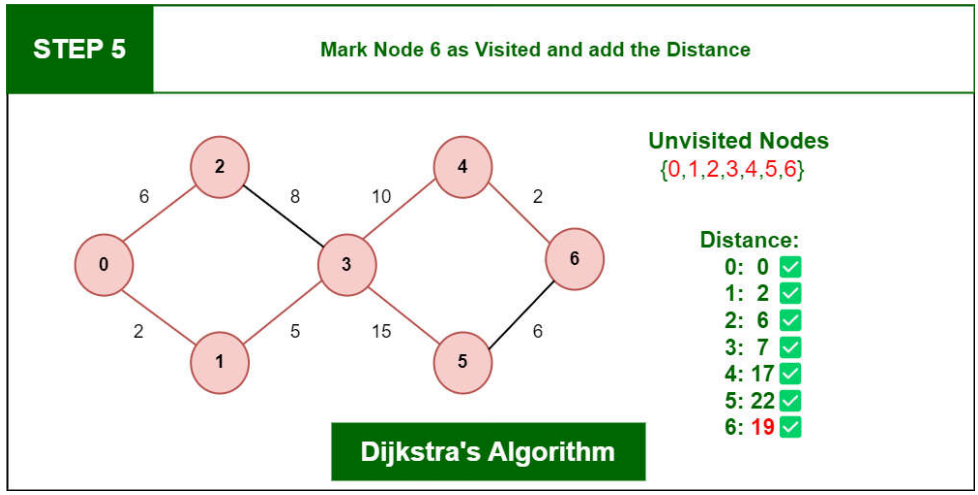
Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



Dijkstra's Algorithm

Step 5: Again, Move Forward and check for adjacent Node which is **Node 6**, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



Dijkstra's Algorithm

So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Pseudo Code for Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
  // Initialize distances to all nodes as infinity, except for the source node.
  distances = map infinity to all nodes
  distances = 0

  // Initialize an empty set of visited nodes and a priority queue to keep track of the nodes to visit.
  visited = empty set
  queue = new PriorityQueue()
  queue.enqueue(source, 0)

  // Loop until all nodes have been visited.
  while queue is not empty:
    // Dequeue the node with the smallest distance from the priority queue.
    current = queue.dequeue()

    // If the node has already been visited, skip it.
    if current in visited:
      continue

    // Mark the node as visited.
    visited.add(current)

    // Check all neighboring nodes to see if their distances need to be updated.
    for neighbor in Graph.neighbors(current):
      // Calculate the tentative distance to the neighbor through the current node.
      tentative_distance = distances[current] + Graph.distance(current, neighbor)
```

```
        // If the tentative distance is smaller than the current distance to the neighbor, update the distance.
        if tentative_distance< distances[neighbor]:
            distances[neighbor] = tentative_distance

            // Enqueue the neighbor with its new distance to be considered for visitation in the future.
            queue.enqueue(neighbor, distances[neighbor])

    // Return the calculated distances from the source to all other nodes in the graph.
    return distances
```

Ways to Implement Dijkstra's Algorithm:

There are several ways to Implement Dijkstra's algorithm, but the most common ones are:

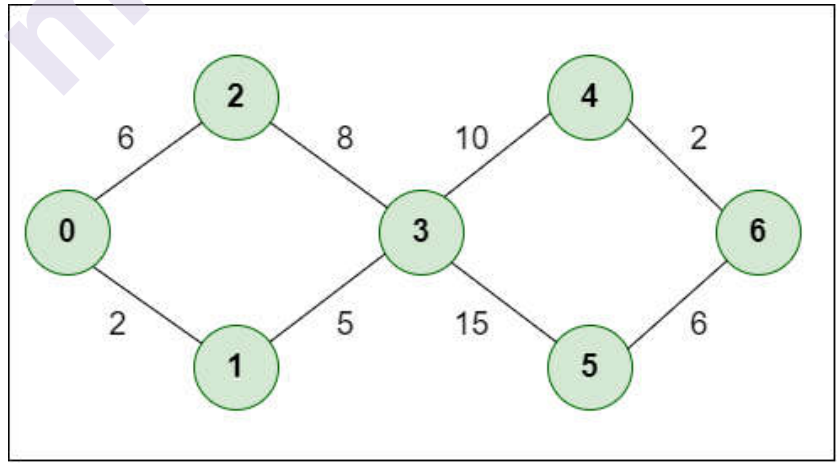
1. Using **priority queue** to keep track of all vertices.
2. Using an **array** to keep track of Distances.
3. Using a **set** to keep track of the visited vertices.

1. Dijkstra's Shortest Path Algorithm using priority_queue (Heap)

In this Implementation, we are given a graph and a source vertex in the graph, finding the shortest paths from the source to all vertices in the given graph.

Example:

Input: Source = 0



Example

Output: Vertex

Vertex Distance from Source

0 -> 0

1 -> 2
2 -> 6
3 -> 7
4 -> 17
5 -> 22
6 -> 19

Below is the algorithm based on the above idea:

1. Initialize distances of all vertices as infinite.
2. Create an empty **priority_queuepq**. Every item of pq is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs
3. Insert source vertex into **pq** and make its distance as 0.
4. While either **pq** doesn't become empty

1. Extract minimum distance vertex from pq. Let the extracted vertex be u.
2. Loop through all adjacent of u and do the following for every vertex v.
3. If there is a shorter path to v through u.

1. **Update distance of v, i.e., do dist[v] = dist[u] + weight(u, v)**
2. **Insert v into the pq (Even if v is already there)**

5. Print distance array **dist[]** to print all shortest paths.

Below is the C++ Implementation of the above approach:

```
// Program to find Dijkstra's shortest path using
// priority_queue in STL
#include <bits/stdc++.h>
usingnamespacestd;
#define INF 0x3f3f3f3f
 // iPair ==> Integer Pair
typedefpair<int, int>iPair;
 // This class represents a directed graph using
// adjacency list representation
```

```
class Graph {
    int V; // No. of vertices
     // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list<pair<int, int>>* adj;
 public:
    Graph(int V); // Constructor
     // function to add an edge to graph
    void addEdge(int u, int v, int w);
     // prints shortest path from s
    void shortestPath(int s);
};
 // Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair>[V];
}
 void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}
 // Prints shortest paths from src to all other vertices
void Graph::shortestPath(int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
```

```
// Refer below link for details of this syntax
// https://www.geeksforgeeks.org/implement-min-heap-using-stl/
priority_queue<iPair, vector<iPair>, greater<iPair>>
    pq;
 // Create a vector for distances and initialize all
// distances as infinite (INF)
vector<int>dist(V, INF);
 // Insert source itself in priority queue and initialize
// its distance as 0.
pq.push(make_pair(0, src));
dist[src] = 0;
 /* Looping till priority queue becomes empty (or all
distances are not finalized) */
while(!pq.empty()) {
    // The first vertex in pair is the minimum distance
    // vertex, extract it from priority queue.
    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted distance (distance must be first item
    // in pair)
    intu = pq.top().second;
    pq.pop();
     // 'i' is used to get all adjacent vertices of a
    // vertex
    list<pair<int, int>>::iterator i;
    for(i = adj[u].begin(); i != adj[u].end(); ++i) {
        // Get vertex label and weight of current
        // adjacent of u.
```

```
            intv = (*i).first;
            intweight = (*i).second;
             // If there is shorted path to v through u.
            if(dist[v] >dist[u] + weight) {
               // Updating distance of v
               dist[v] = dist[u] + weight;
               pq.push(make_pair(dist[v], v));
            }
         }
      }
      // Print shortest distances stored in dist[]
      printf("Vertex Distance from Source\n");
      for(inti = 0; i< V; ++i)
         printf("%d \t\t %d\n", i, dist[i]);
}
 // Driver program to test methods of graph class
intmain()
{
   // create the graph given in above figure
   intV = 7;
   Graph g(V);
    // making above shown graph
   g.addEdge(0, 1, 2);
   g.addEdge(0, 2, 6);
   g.addEdge(1, 3, 5);
   g.addEdge(2, 3, 8);
   g.addEdge(3, 4, 10);
   g.addEdge(3, 5, 15);
```

```
    g.addEdge(4, 6, 2);
    g.addEdge(5, 6, 6);
    g.shortestPath(0);
     return0;
}
```

Output

Vertex Distance from Source

0 0

1 2

2 6

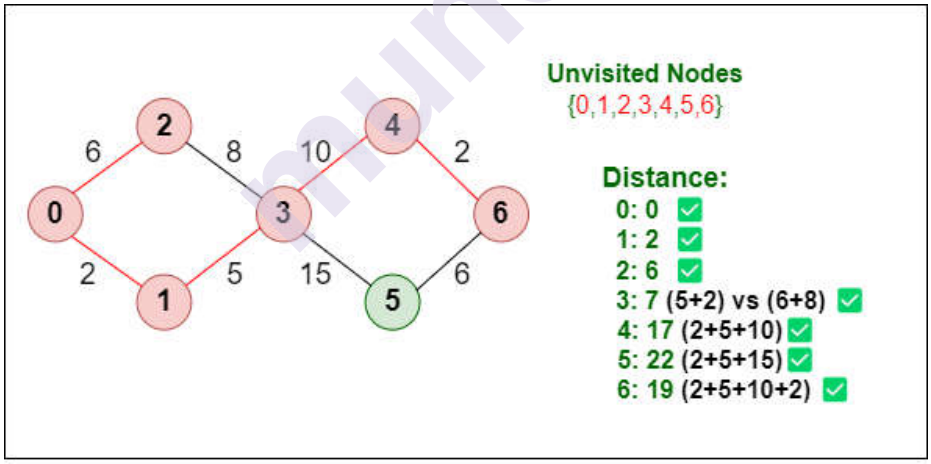
3 7

4 17

5 22

6 19

Final Answer:



Output

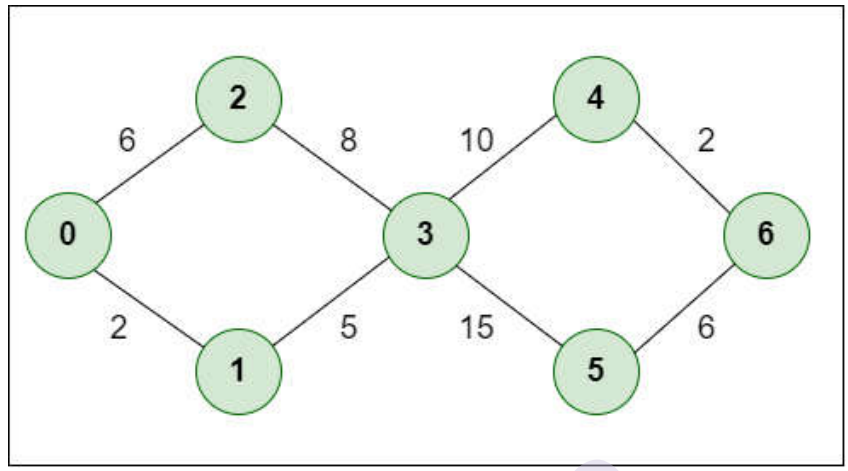
Complexity Analysis of Dijkstra's Algorithm using Priority Queue:

Time complexity : O(E log V)
Space Complexity: O(V^2), here V is the number of Vertices.

2. Dijkstra shortest path algorithm using Prim's Algorithm

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

Example:



Example

Output: 0 2 6 7 17 22 19

Follow the below steps to implement Dijkstra's Algorithm using Prim's Algorithm:

- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign the distance value as **0** for the source vertex so that it is picked first.
- While **sptSet** doesn't include all vertices
- Pick a vertex u which is not there in **sptSet** and has a minimum distance value.
- Include u to **sptSet**.
- Then update distance value of all adjacent vertices of **u.**
- To update the distance values, iterate through all adjacent vertices.
- For every adjacent vertex **v**, if the sum of the distance value of **u** (from source) and weight of edge **u-v,** is less than the distance value of **v**, then update the distance value of **v.**

Complexity Analysis of Dijkstra's Algorithm using Prims Algorithm:

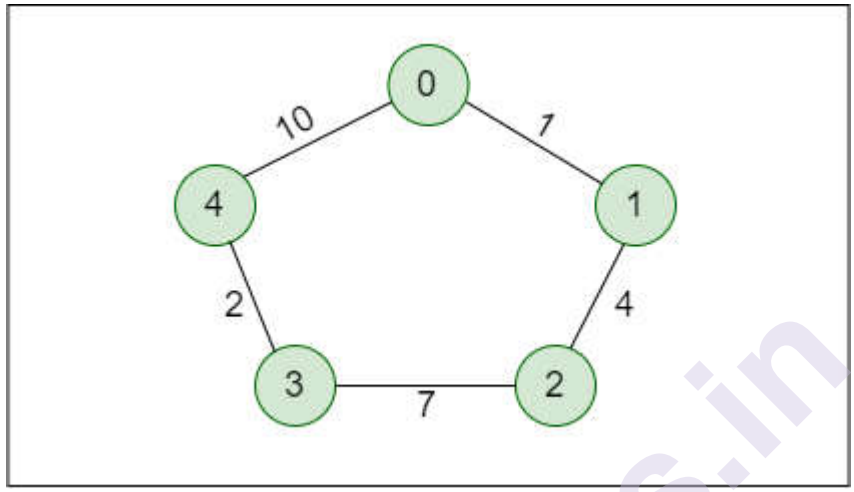
Time Complexity: O(V2)

Auxiliary Space: O(V)

3. Dijkstra's shortest path with minimum edges

In this Implementation we are given an adjacency atrix **graph** representing paths between the nodes in the given graph. The task is to find the shortest path with minimum edges i. e. if there a multiple short paths with same cost then choose the one with the minimum number of edges.

Consider the graph given below:



Example

There are two paths from vertex **0** to vertex **3** with a weight of 12:

0 -> 1 -> 2 -> 3

0 -> 4 -> 3

Since Dijkstra's algorithm is a greedy algorithm that seeks the minimum weighted vertex on every iteration, so the original Dijkstra's algorithm will output the first path but the result should be the second path as it contains minimum number of edges.

Examples:

Input: graph[][] = { {0, 1, INFINITY, INFINITY, 10},
{1, 0, 4, INFINITY, INFINITY},
{INFINITY, 4, 0, 7, INFINITY},
{INFINITY, INFINITY, 7, 0, 2},
{10, INFINITY, INFINITY, 2, 0} };
Output: 0->4->3
INFINITY here shows that u and v are not neighbors

Input: graph[][] = { {0, 5, INFINITY, INFINITY},
{5, 0, 5, 10},
{INFINITY, 5, 0, 5},
{INFINITY, 10, 5, 0} };
Output: 0->1->3

Approach: The idea of the algorithm is to use the original Dijkstra's algorithm, but also to keep track of the length of the paths by an array that stores the length of the paths from the source vertex, so if we find a shorter path with the same weight, then we will take it.

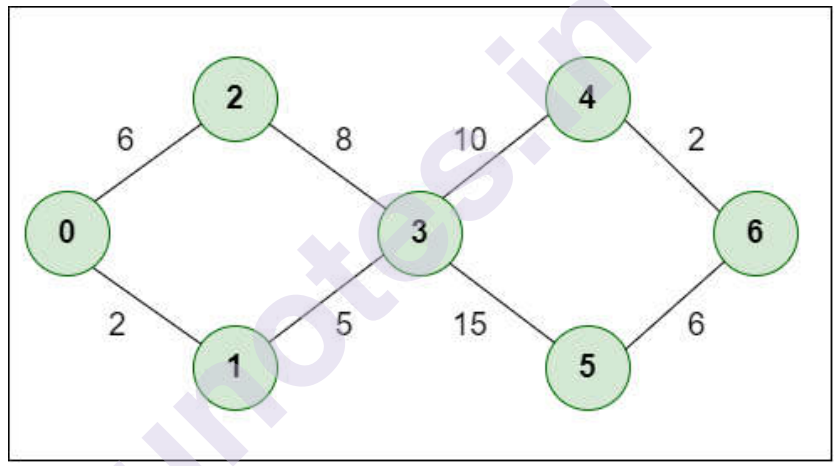
Complexity Analysis:

- **Time Complexity**: O(V^2) where V is the number of vertices and E is the number of edges.
- **Auxiliary Space**: O(V + E)

4. Dijkstra's shortest path algorithm using Set

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph

Example:



Example

Output: 0 2 6 7 17 22 19

Below is algorithm based on set data structure.

1. Initialize distances of all vertices as infinite.
2. Create an empty set. Every item of set is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs.
3. Insert source vertex into the set and make its distance as 0.
4. While Set doesn't become empty, do following

   a) Extract minimum distance vertex from Set.

   Let the extracted vertex be u.

   b) Loop through all adjacent of u and do

   following for every vertex v.

// If there is a shorter path to v through u.

If dist[v] >dist[u] + weight(u, v

(i) Update distance of v, i.e., do dist[v] = dist[u] + weight(u, v)

(i) If v is in set, update its distance in set by removing it first, then inserting with new distance

(ii) If v is not in set, then insert it in set with new distance

5) Print distance array **dist[**] to print all shortest paths.

Complexity Analysis:

- **Time Complexity**: O(ELogV), where V is the number of vertices and E is the number of edges.
- **Auxiliary Space**: O(V2)

Complexity Analysis of Dijkstra's Algorithm:

The time complexity of Dijkstra's algorithm depends on the data structure used for the priority queue. Here is a breakdown of the time complexity based on different implementations:

- **Using an unsorted list** as the priority queue: **$O(V^2)$**, where V is the number of vertices in the graph. In each iteration, the algorithm searches for the vertex with the smallest distance among all unvisited vertices, which takes O(V) time. This operation is performed V times, resulting in a time complexity of O(V^2).
- **Using a sorted list** or a **binary heap** as the priority queue: **O(E + V log V)**, where E is the number of edges in the graph. In each iteration, the algorithm extracts the vertex with the smallest distance from the priority queue, which takes O(log V) time. The distance updates for the neighboring vertices take O(E) time in total. This operation is performed V times, resulting in a time complexity of O(V log V + E log V). Since E can be at most V^2, the time complexity is O(E + V log V).

Dijkstra's Algorithms vs Bellman-Ford Algorithm

Dijkstra's algorithm and Bellman-Ford algorithm are both used to find the shortest path in a weighted graph, but they have some key differences. Here are the main differences between Dijkstra's algorithm and Bellman-Ford algorithm:

Feature:	Dijkstra's	Bellman Ford
Optimization	optimized for finding the shortest path between a single source node and all other nodes in a graph with non-negative edge weights	Bellman-Ford algorithm is optimized for finding the shortest path between a single source node and all other nodes in a graph with negative edge weights.
Relaxation	Dijkstra's algorithm uses a greedy approach where it chooses the node with the smallest distance and updates its neighbors	the Bellman-Ford algorithm relaxes all edges in each iteration, updating the distance of each node by considering all possible paths to that node
Time Complexity	Dijkstra's algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.	Bellman-Ford algorithm has a time complexity of O(VE), where V is the number of vertices and E is the number of edges in the graph.
Negative Weights	Dijkstra's algorithm does not work with graphs that have negative edge weights, as it assumes that all edge weights are non-negative.	Bellman-Ford algorithm can handle negative edge weights and can detect negative-weight cycles in the graph.

Dijkstra's algorithm vs Floyd-Warshall Algorithm

Dijkstra's algorithm and Floyd-Warshall algorithm are both used to find the shortest path in a weighted graph, but they have some key differences. Here are the main differences between Dijkstra's algorithm and Floyd-Warshall algorithm:

Feature:	Dijkstra's	Floyd-Warshall Algorithm
Optimization	Optimized for finding the shortest path between a single source node and all other nodes in a graph with non-negative edge weights	Floyd-Warshall algorithm is optimized for finding the shortest path between all pairs of nodes in a graph.
Technique	Dijkstra's algorithm is a single-source shortest path algorithm that uses a greedy approach and calculates the shortest path from the source node to all other nodes in the graph.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.
Time Complexity	Dijkstra's algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.
Negative Weights	Dijkstra's algorithm does not work with graphs that have negative edge weights, as it assumes that all edge weights are non-negative.	Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that uses dynamic programming to calculate the shortest path between all pairs of nodes in the graph.

Dijkstra's algorithm vs A* Algorithm

Dijkstra's algorithm and A* algorithm are both used to find the shortest path in a weighted graph, but they have some key differences. Here are the main differences between Dijkstra's algorithm and A* algorithm:

Feature:		A* Algorithm
Search Technique	Optimized for finding the shortest path between a	A* algorithm is an informed search algorithm that uses a

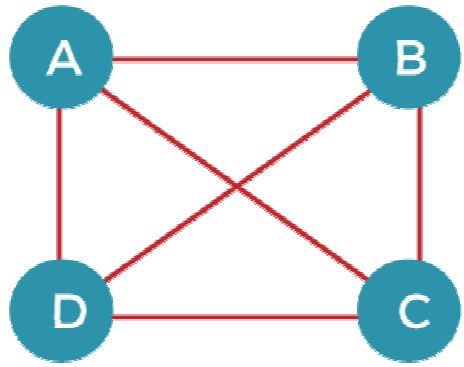
Feature:		A* Algorithm
	single source node and all other nodes in a graph with non-negative edge weights	heuristic function to guide the search towards the goal node.
Heuristic Function	Dijkstra's algorithm, does not use any heuristic function and considers all the nodes in the graph.	A* algorithm uses a heuristic function that estimates the distance from the current node to the goal node. This heuristic function is admissible, meaning that it never overestimates the actual distance to the goal node
Time Complexity	Dijkstra's algorithm has a time complexity of O(V^2) for a dense graph and O(E log V) for a sparse graph, where V is the number of vertices and E is the number of edges in the graph.	The time complexity of A* algorithm depends on the quality of the heuristic function.
Application	Dijkstra's algorithm is used in many applications such as routing algorithms, GPS navigation systems, and network analysis	. A* algorithm is commonly used in pathfinding and graph traversal problems, such as video games, robotics, and planning algorithms.

Conclusion: Overall, Dijkstra's algorithm is a simple and efficient way to find the shortest path in a graph with non-negative edge weights. However, it may not work well with graphs that have negative edge weights or cycles. In such cases, more advanced algorithms such as the Bellman-Ford algorithm or the Floyd-Warshall algorithm may be used.

Minimal Spanning Tree

Before knowing about the minimum spanning tree, we should know about the spanning tree.

To understand the concept of spanning tree, consider the below graph:



The above graph can be represented as G(V, E), where 'V' is the number of vertices, and 'E' is the number of edges. The spanning tree of the above graph would be represented as G`(V`, E`). In this case, V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different. The number of edges in the spanning tree is the subset of the number of edges in the original graph. Therefore, the number of edges can be written as:

E` € E

It can also be written as:

E` = |V| - 1

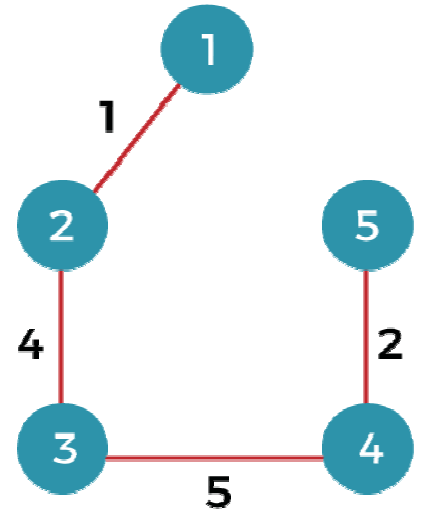
Two conditions exist in the spanning tree, which is as follows:

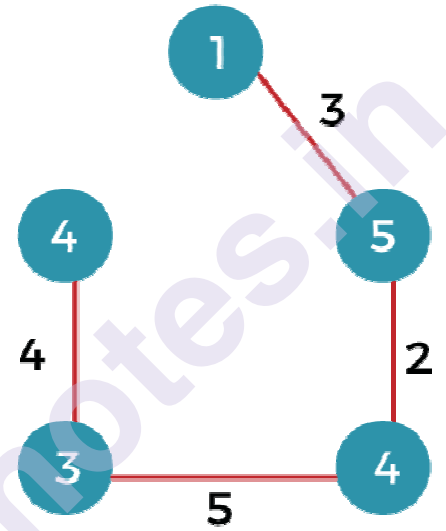
- The number of vertices in the spanning tree would be the same as the number of vertices in the original graph. **V` = V**
- The number of edges in the spanning tree would be equal to the number of edges minus 1. **E` = |V| - 1**
- The spanning tree should not contain any cycle.
- The spanning tree should not be disconnected.

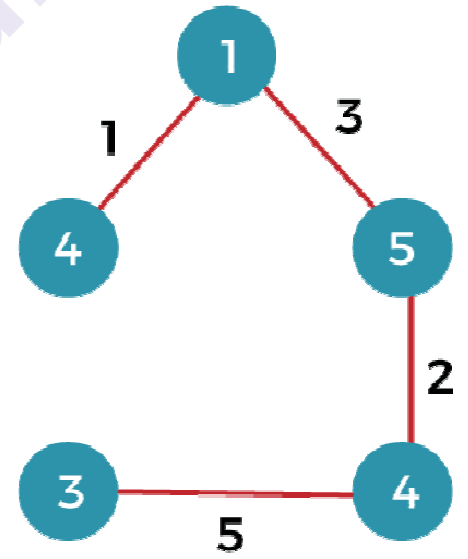
Note: A graph can have more than one spanning tree.

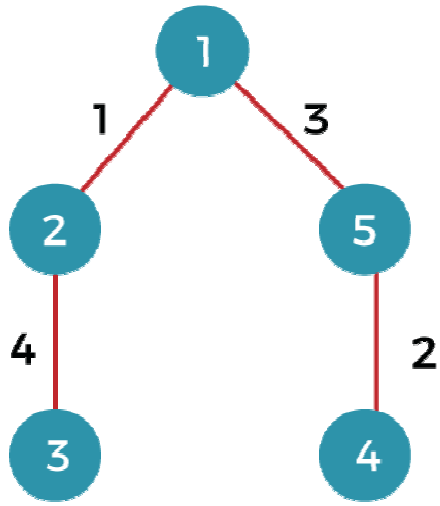
Consider the below graph:

The above graph contains 5 vertices. As we know, the vertices in the spanning tree would be the same as the graph; therefore, V` is equal 5. The number of edges in the spanning tree would be equal to (5 - 1), i.e., 4. The following are the possible spanning trees:









What is a minimum spanning tree?

The minimum spanning tree is a spanning tree whose sum of the edges is minimum. Consider the below graph that contains the edge weight:

The following are the spanning trees that we can make from the above graph.

- The first spanning tree is a tree in which we have removed the edge between the vertices 1 and 5 shown as below:
  The sum of the edges of the above tree is (1 + 4 + 5 + 2): 12
- The second spanning tree is a tree in which we have removed the edge between the vertices 1 and 2 shown as below:
  The sum of the edges of the above tree is (3 + 2 + 5 + 4) : 14
- The third spanning tree is a tree in which we have removed the edge between the vertices 2 and 3 shown as below:
  The sum of the edges of the above tree is (1 + 3 + 2 + 5) : 11
- The fourth spanning tree is a tree in which we have removed the edge between the vertices 3 and 4 shown as below:
  The sum of the edges of the above tree is (1 + 3 + 2 + 4) : 10. The edge cost 10 is minimum so it is a minimum spanning tree.

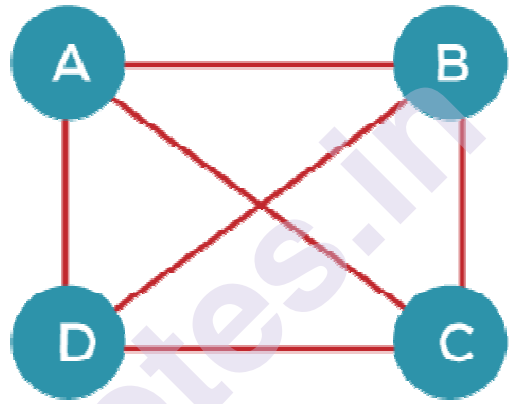
General properties of minimum spanning tree:

- If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.
- If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.
- In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.

- A complete undirected graph can have an n^{n-2} number of spanning trees.
- Every connected and undirected graph contains atleast one spanning tree.
- The disconnected graph does not have any spanning tree.
- In a complete graph, we can remove maximum (e-n+1) edges to construct a spanning tree.

Let's understand the last property through an example.

Consider the complete graph which is given below:



The number of spanning trees that can be made from the above complete graph equals to $n^{n-2} = 4^{4-2} = 16$.

Therefore, 16 spanning trees can be created from the above graph.

The maximum number of edges that can be removed to construct a spanning tree equals to e-n+1 = 6 - 4 + 1 = 3.

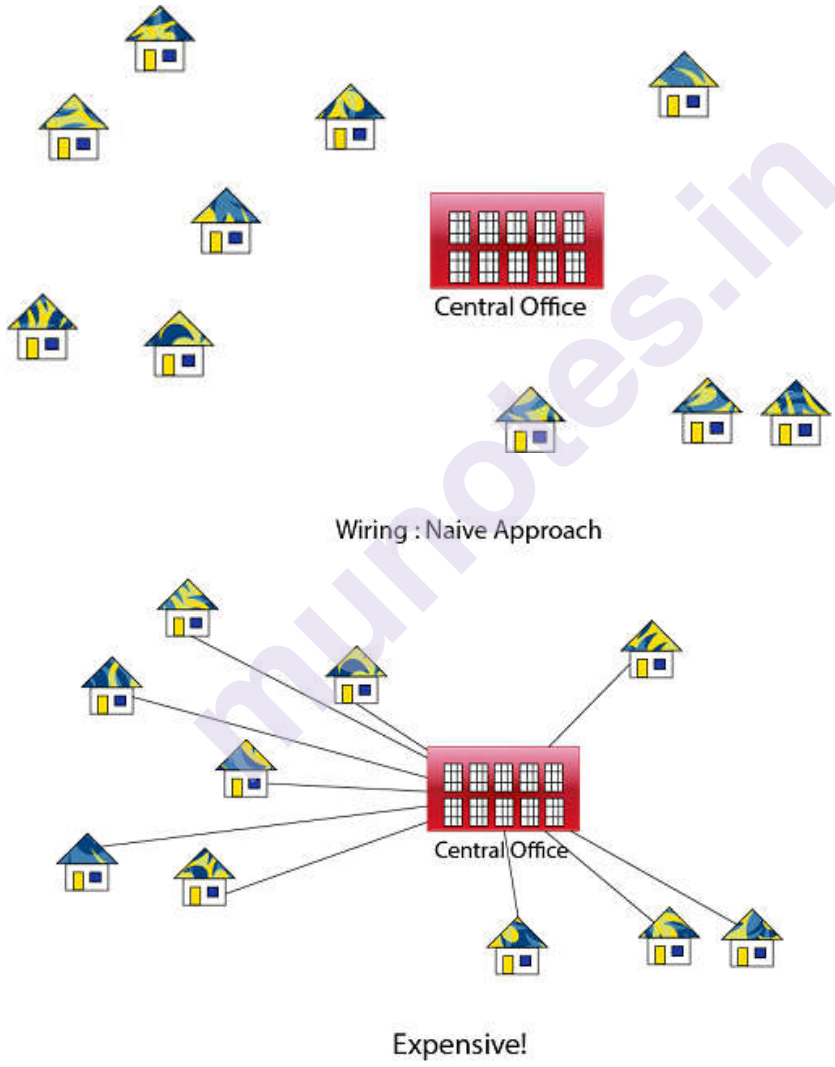
Application of Minimum Spanning Tree

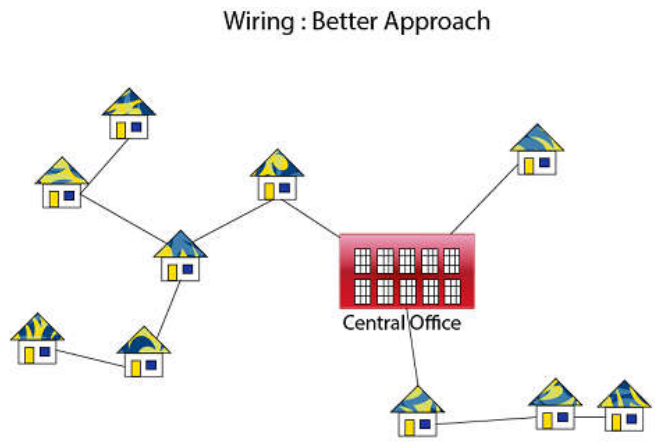
1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.
   The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.

5. Suppose you want to apply a set of houses with
   - Electric Power
   - Water
   - Telephone lines
   - Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

For Example, Problem laying Telephone Wire.





Minimize the total length of wire connecting the customers

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

1. A ← ∅
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)

7. then A ← A ∪ {(u, v)}

8. UNION (u, v)

9. return A

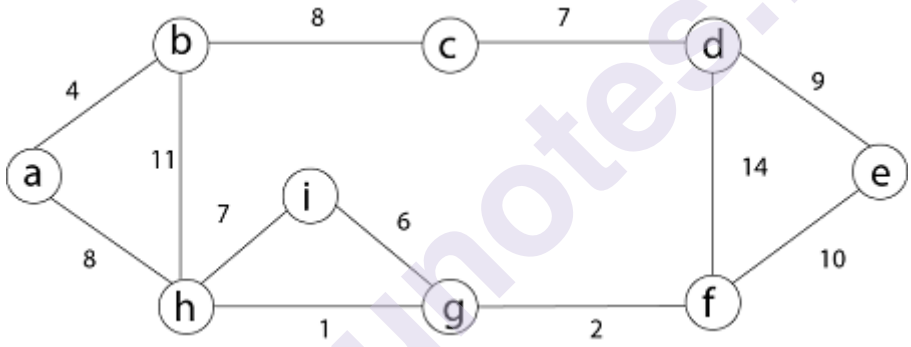
Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2E$, so log V is O (log E).

Thus the total time is

1. O (E log E) = O (E log V).

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

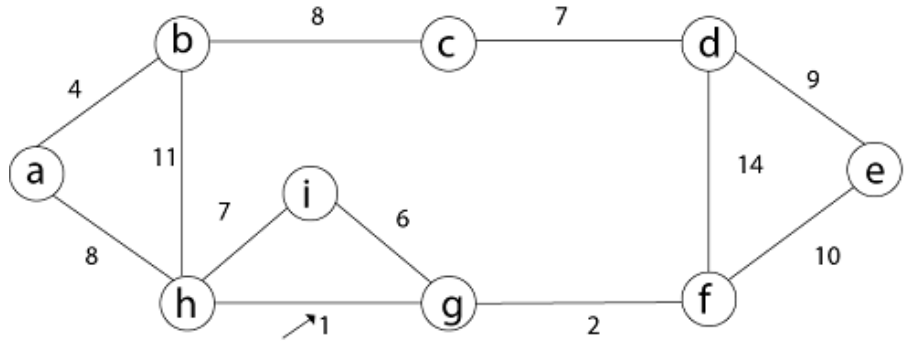
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

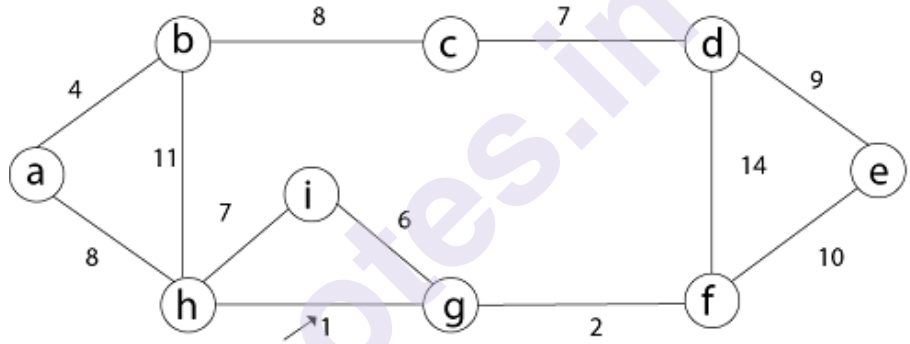
Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is

added to A, and the vertices in two trees are merged in by union procedure.

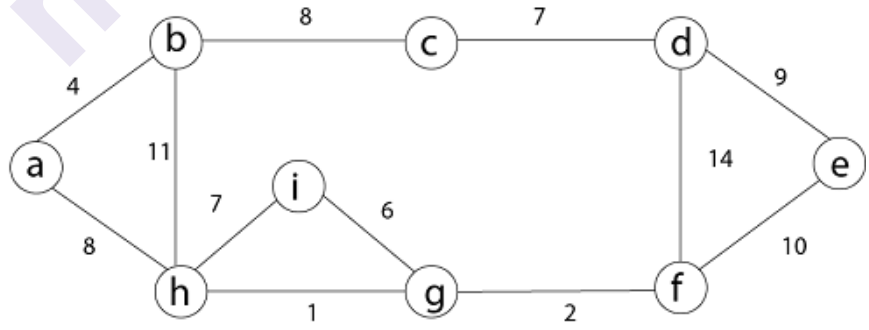
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

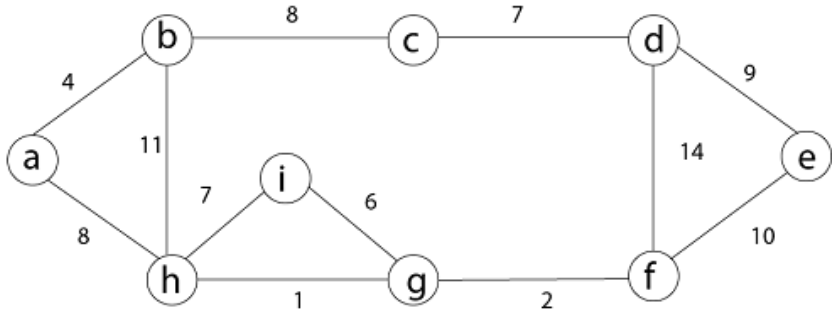


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



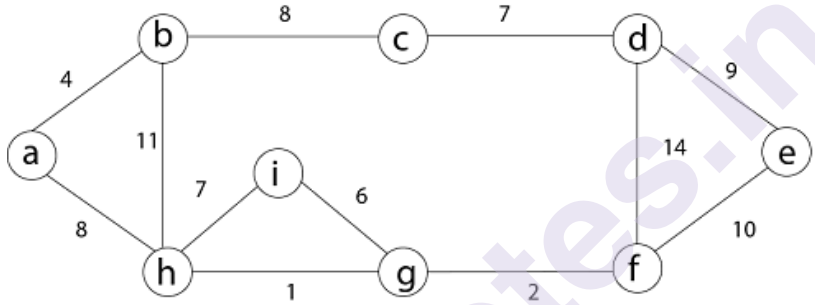
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



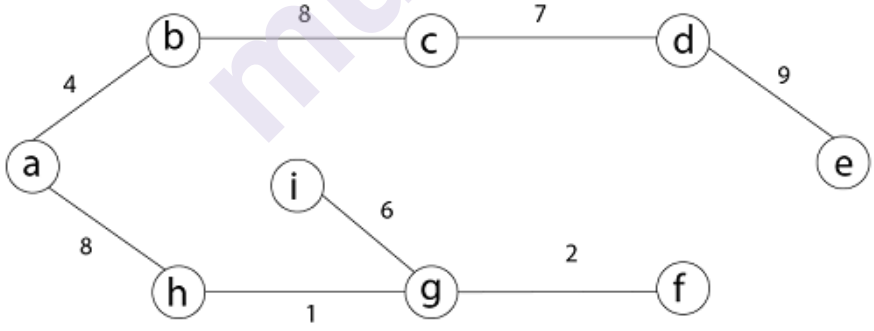
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.

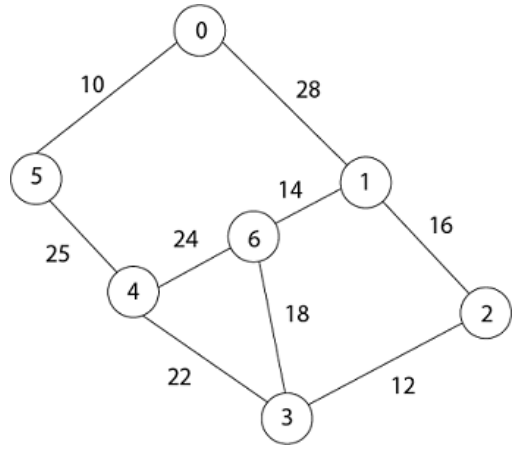
a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.

b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
6. Q ← V [G]
7. While Q ?∅
8. do u ← EXTRACT - MIN (Q)
9. for each v ∈Adj [u]
10. do if v ∈ Q and w (u, v) < key [v]
11. then π [v] ← u
12. key [v] ← w (u, v)

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



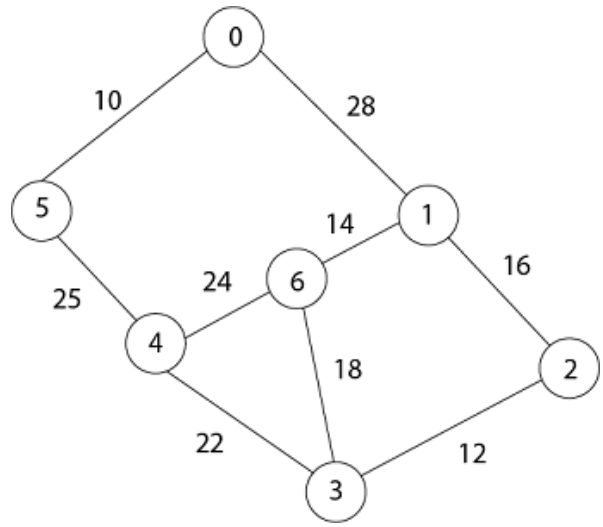
Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

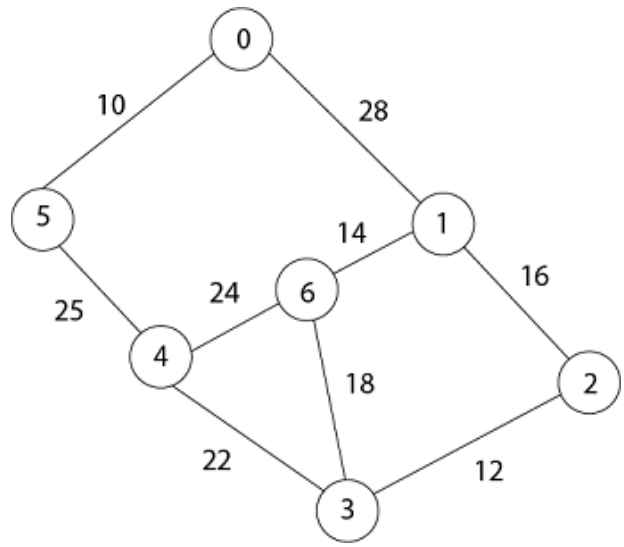
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞ π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞ key [6] = ∞
3. w (4,3) = 22 w (4,6) = 24
4. w (u, v) < key [v] w (u, v) < key [v]
5. w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

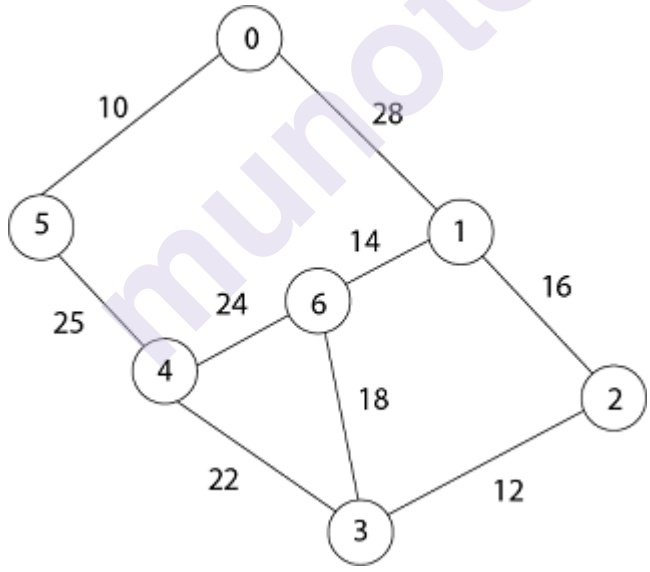
And the parent of 3, 6 as 4.

1. π[3]= 4 π[6]= 4

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

1. u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
2. u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}
2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞ key [6] = 24
5. w (3, 2) = 12 w (3, 6) = 18
6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

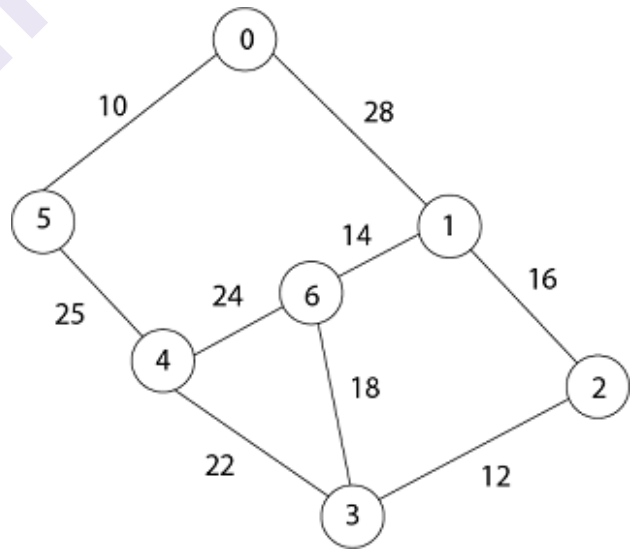
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

1. u = EXTRACT_MIN (2, 6)
2. u = 2 [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}
2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

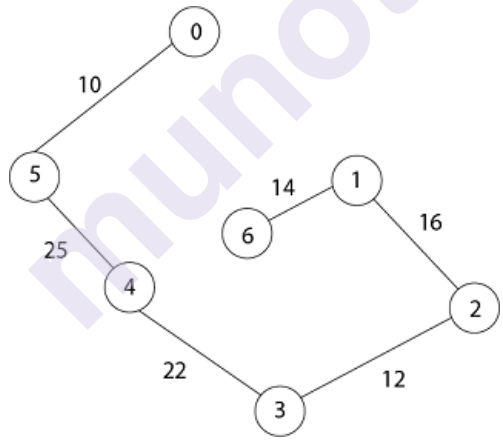
1. Π [6] = 1

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6
2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

4.9 QUESTION BANK AND EXCERCISE

1. What is Data Structure: Types, Classifications and Applications
2. Applications, Advantages and Disadvantages of Directed Graph
3. Applications, Advantages and Disadvantages of Unweighted Graph
4. Graph Coloring | Set 1 (Introduction and Applications)
5. Applications, Advantages and Disadvantages of Weighted Graph

6. Applications, Advantages and Disadvantages of Graph
7. Check whether the structure of given Graph is same as the game of Rock-Paper-Scissor
8. Maximum number of edges that N-vertex graph can have such that graph is Triangle free | Mantel's Theorem
9. Python Program to Find Independent Sets in a Graph using Graph Coloring
10. What is graphs? Enlist Types of Graphs.
11. Explain Shortest Path First Algorithms?
12. What is BFS & DFS?
13. Explain minimal spanning tree with examples?
14. Explain Graph Representation & Graph Traversal?
15. Enlist Applications of Graphs?
16. Explain Topological Sort in detail?
17. Explain Graph Algorithms in details?
18. Write a short note on Weighted Graphs & Directed graphs?
19. Expalin What is Dijkstra's Algorithm? Introduction to Dijkstra's Shortest Path Algorithm.
20. What is Graph Colouring ?

4.10 CONCLUSION

In this article, you came across plenty of graph coloring algorithms and techniques that find their day-to-day applications in all instances of real life. You learned how to implement them according to situations, and hence the pseudo code helped you process the information strategically and efficiently. Graph algorithms are considered an essential aspect in the field confined not only to solve problems using data structures but also in general tasks like Google Maps and Apple Maps. However, a beginner might find it hard to implement Graph algorithms because of their complex nature. Hence, it is highly recommended to go through this article since it covers everything from scratch.

5

SELECTION ALGORITHMS

UnitStructure

5.0 OBJECTIVES

A sorting algorithm is a method for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest value or shortest-to-longest distance. Sorting algorithms take lists of items as input data, perform specific operations on those lists and deliver ordered arrays as output.

5.1 INTRODUCTION OF SELECTION SORT ALGORITHMS

What Is a Selection Sort Algorithm? Selection sort is an effective and efficient sort algorithm based on comparison operations. It adds one element in each iteration. You need to select the smallest element in the array and move it to the beginning of the array by swapping with the front element.

Selection Algorithm is an algorithm for finding the kth smallest (or largest) number in a list or an array. That number is called the kth order statistic. It includes the various cases for finding the minimum, maximum and median elements in a list or an array. For finding the minimum (or maximum) element by iterating through the list, we keep the track of current minimum (or maximum) elements that occur so far and it is related to the selection sort. Below are the different ways of selecting the kth smallest(or largest) element in an unordered list:

5.2 WHAT ARE THE SELECTION ALGORITHM

1. Selection by Sorting

Sorting the list or an array then selecting the required element can make selection easy. This method is inefficient for selecting a single element but is efficient when many selections need to be made from an array for which it only needs sorting of an array. For selection in a linked list is O(n) even if the linked list is sorted due to lack of random access. Instead of sorting the entire list or an array, we can use partial sorting to select the kth smallest(or largest) element in a list or an array. Then the kth smallest(or largest) is the largest(or smallest) element of the partially sorted list. This takes O(1) to access in an array and O(k) to access in a list.

- **Unordered Partial Sorting**

Unordered partial sorting is a sorting algorithm in such a way that first k element are in sorted order and rest element are in random order. For finding the kth smallest(or largest) element. The time complexity reduces to O(k log k). But since K ≤ n, The asymptotic Time Complexity converges to O(n). Note: Due to the possibility of equality of elements, one must not include the element less than or equals to the kth element while sorting a list or an array, as element greater than the kth element may also be equal to the kth smallest element.

- **Partial selection sort**

The concept used in Selection Sort helps us to partially sort the array up to kth smallest(or largest) element for finding the kth smallest(or largest) element in an array. Thus a partial selection sort yields a simple selection algorithm that takes O(k*n) time to sort the array. This is asymptotically inefficient but can be sufficiently efficient if k is small, and is easy to implement.
Below is the algorithm for partial selection sort:

- functionpartialSelectionSort(arr[0..n], k) {
- fori in [0, k){
- minIndex = i
- minValue = arr[i]
- for j in [i+1, n){
- if (arr[j] <minValue) then
- minIndex = j
- minValue = arr[j]
- swap(arr[i], arr[minIndex])

- }
- }
- returnarr[k]
- }

2. Partition Based Selection:

For partition-based selection, the Quick select Algorithm is used. It is a variant of quicksort algorithm. In both, we choose a pivot element and using the partition step from the quicksort algorithm arranges all the elements smaller than the pivot on its left and the elements greater than it on its right. But while Quicksort recurses on both sides of the partition, Quickselect only recurses on one side, the side on which the desired kth element is present. The partition-based algorithms are done in place, which results in partially sorting the data. They can be done out of place by not changing the original data at the cost of O(n) auxiliary space.

3. Median selected as pivot:

A median-selection algorithm can be used to perform a selection algorithm or sorting algorithm, by selecting the median of the array as the pivot element in Quickselect or Quicksort algorithm. In practice the overhead of pivot computation is significant, so these algorithms are generally not used, but this technique is of theoretical interest in relating selection and sorting algorithms. The median of an array is the best pivot for sorting of an array because it evenly divides the data into two parts., and thus guarantees optimal sorting, assuming the selection algorithm is optimal.

4. selection sort in python:

In this tutorial, we will implement the selection sort algorithm in Python. It is quite straightforward algorithm using the less swapping.

In this algorithm, we select the smallest element from an unsorted array in each pass and swap with the beginning of the unsorted array. This process will continue until all the elements are placed at right place. It is simple and an in-place comparison sorting algorithm.

5.3 WORKING OF SELECTION SORT

The following are the steps to explain the working of the Selection sort in Python.

Let's take an unsorted array to apply the selection sort algorithm.

[30, 10, 12, 8, 15, 1]

Step - 1: Get the length of the array.

length = len(array) → 6

Step - 2: First, we set the first element as minimum element.

Step - 3: Now compare the minimum with the second element. If the second element is smaller than the first, we assign it as a minimum.

Again we compare the second element to the third and if the third element is smaller than second, assign it as minimum. This process goes on until we find the last element.

Step - 4: After each iteration, minimum element is swapped in front of the unsorted array.

Step - 5: The second to third steps are repeated until we get the sorted array.

Selection Sort Algorithm

The selection sort algorithm as follows.

Algorithm

1. selection_sort(array)
2. repeat (0, length - 1) times
3. set the first unsorted element as the minimum
4. **for** each of the unsorted elements
5. **if** element < currentMinimum
6. set element as **new** minimum
7. swap minimum with first unsorted position
8. end selection_sort

Selection Sort Program using Python

The following code snippet shows the selection sort algorithm implementation using Python.

Code -

```
1. def selection_sort(array):
2.     length = len(array)
3.
4.     for i in range(length-1):
5.         minIndex = i
6.
7.         for j in range(i+1, length):
8.             if array[j]<array[minIndex]:
```

```
9.                minIndex = j
10.
11.            array[i], array[minIndex] = array[minIndex], array[i]
12.
13.
14.        return array
15.    array = [21,6,9,33,3]
16.
17.    print("The sorted array is: ", selection_sort(array))
```

Output:

```
The sorted array is:  [3, 6, 9, 21, 33]
```

Explanation -

Let's understand the above code -

- First, we define the **selection_sort()** function that takes array as an argument.
- In the function, we get the length of the array which used to determine the number of passes to be made comparing values.
- As we can see that, we use two loops - outer and inner loop. The outer loop uses to iterate through the values of the list. This loop will iterate to 0 to (length-1). So the first iteration will be perform (5-1) or 4 times. In each iteration, the value of the variable i is assigned to the variable
- The inner loop uses to compare the each value of right-side element to the other value on the leftmost element. So the second loop starts its iteration from i+1. It will only pick the value that is unsorted.
- Find the minimum element in the unsorted list and update the minIndex position.
- Place the value at the beginning of the array.
- Once the iteration is completed, the sorted array is returned.
- At last we create an unsorted array and pass to the **selection_sort()** It prints the sorted array.

Time Complexity of Selection Sort

Time complexity is an essential in term of how much time an algorithm take to sort it. In the selection sort, there are two loops. The outer loop runs for the n times (n is a total number of element).

The inner loop is also executed for n times. It compares the rest of the value to outer loop value. So, there is n*n times of execution. Hence the time complexity of merge sort algorithm is $O(n^2)$.

The time complexity can be categorized into three categories.

5.4 LINEAR SELECTION ALGORITHM

A linear search is the simplest approach employed to search for an element in a data set. It examines each element until it finds a match, starting at the beginning of the data set, until the end. The search is finished and terminated once the target element is located. If it finds no match, the algorithm must terminate its execution and return an appropriate result. The linear search algorithm is easy to implement and efficient in two scenarios:

- When the list contains lesser elements
- When searching for a single element in an unordered array

What Is Searching?

Searching is the process of fetching a specific element in a collection of elements. The collection can be an array or a linked list. If you find the element in the list, the process is considered successful, and it returns the location of that element.

And in contrast, if you do not find the element, it deems the search unsuccessful.

Two prominent search strategies are extensively used to find a specific item on a list. However, the algorithm chosen is determined by the list's organization.

1. Linear Search
2. Binary Search

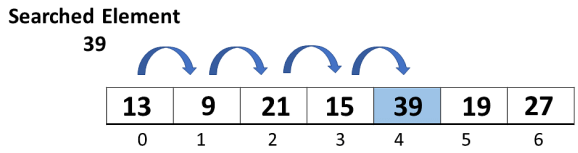
Moving ahead in this tutorial, you will understand what exactly a linear search algorithm is.

What Is a Linear Search Algorithm?

Linear search, often known as sequential search, is the most basic search technique. In this type of search, you go through the entire list and try to fetch a match for a single element. If you find a match, then the address of the matching target element is returned.

On the other hand, if the element is not found, then it returns a NULL value.

Following is a step-by-step approach employed to perform Linear Search Algorithm.



The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6: If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

So far, you have explored the fundamental definition and the working terminology of the Linear Search Algorithm.

Algorithm and Pseudocode of Linear Search Algorithm

Algorithm of the Linear Search Algorithm

Linear Search (Array Arr, Value a) // Arr is the name of the array, and a is the searched element.

Step 1: Set i to 0 // i is the index of an array which starts from 0

Step 2: ifi> n then go to step 7 // n is the number of elements in array

Step 3: if Arr[i] = a then go to step 6

Step 4: Set i to i + 1

Step 5: Goto step 2

Step 6: Print element a found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode of Linear Search Algorithm

Start

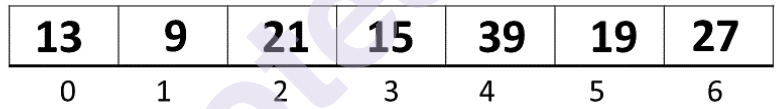
```
linear_search ( Array , value)

For each element in the array

   If (searched element == value)

      Return's the searched lament location

   end if

end for

end
```

Following your understanding of the linear search algorithm and pseudocode, in the next segment, you will go through the practical implementation of the algorithm.

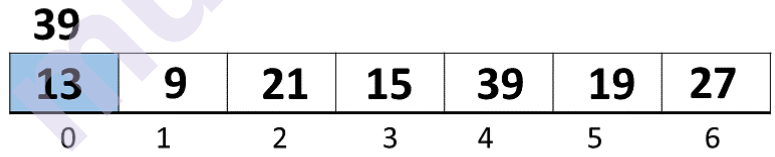
Example of Linear Search Algorithm

Consider an array of size 7 with elements 13, 9, 21, 15, 39, 19, and 27 that starts with 0 and ends with size minus one, 6.

Search element = 39

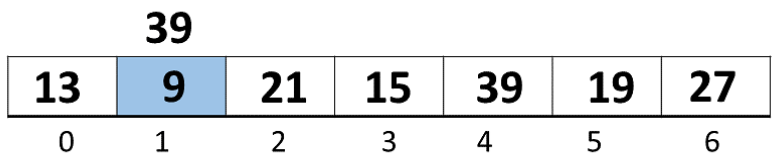


Step 1: The searched element 39 is compared to the first element of an array, which is 13.



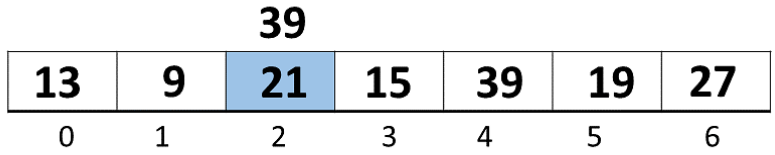
The match is not found, you now move on to the next element and try to implement a comparison.

Step 2: Now, search element 39 is compared to the second element of an array, 9.



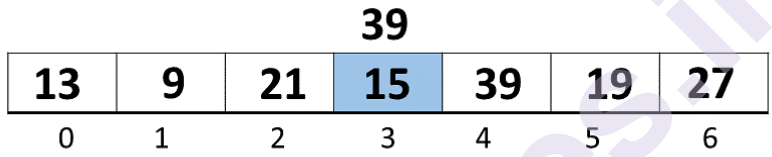
As both are not matching, you will continue the search.

Step 3: Now, search element 39 is compared with the third element, which is 21.



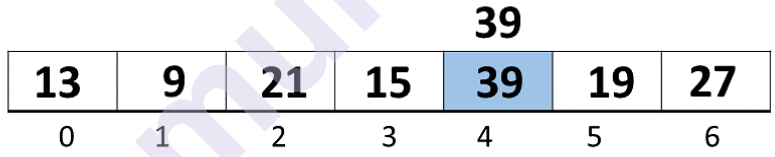
Again, both the elements are not matching, you move onto the next following element.

Step 4; Next, search element 39 is compared with the fourth element, which is 15.



As both are not matching, you move on to the next element.

Step 5: Next, search element 39 is compared with the fifth element 39.



A perfect match is found, you stop comparing any further elements and terminate the Linear Search Algorithm and display the element found at location 4.

Followed by the practical implementation, you will move on to the complexity of the linear search algorithm.

The Complexity of Linear Search Algorithm

You have three different complexities faced while performing Linear Search Algorithm, they are mentioned as follows.

1. Best Case
2. Worst Case
3. Average Case

You will learn about each one of them in a bit more detail.

Best Case Complexity

1. The element being searched could be found in the first position.
2. In this case, the search ends with a single successful comparison.
3. Thus, in the best-case scenario, the linear search algorithm performs O(1) operations.

Worst Case Complexity

- The element being searched may be at the last position in the array or not at all.
- In the first case, the search succeeds in 'n' comparisons.
- In the next case, the search fails after 'n' comparisons.
- Thus, in the worst-case scenario, the linear search algorithm performs O(n) operations.

Average Case Complexity

When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is O(n).

Next, you will learn about the Space Complexity of Linear Search Algorithm.

Space Complexity of Linear Search Algorithm

The linear search algorithm takes up no extra space; its space complexity is O(n) for an array of n elements.

Now that you've grasped the complexity of the linear search algorithm, look at some of its applications.

Application of Linear Search Algorithm

The linear search algorithm has the following applications:

- Linear search can be applied to both single-dimensional and multi-dimensional arrays.
- Linear search is easy to implement and effective when the array contains only a few elements.
- Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

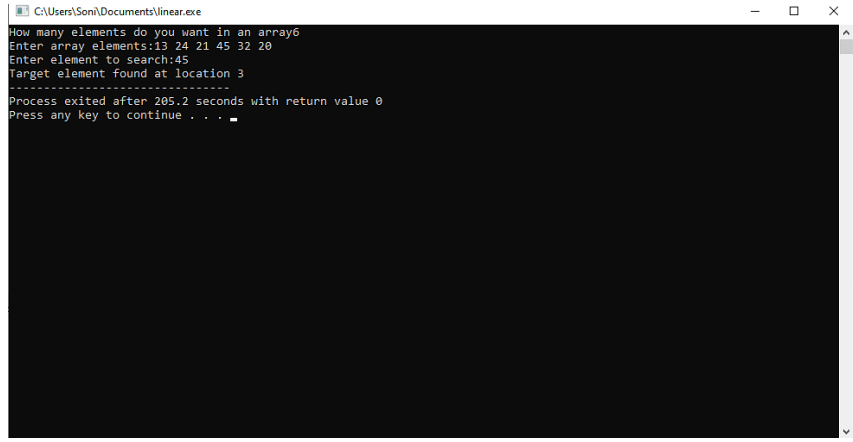
Finally, in this tutorial, you will implement the linear search algorithm in code.

Also Read: Arrays in Data Structure

Code Implementation of Linear Search Algorithm

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int main()
{
   int array[50],i,target,num;
   printf("How many elements do you want in the array");
   scanf("%d",&num);
   printf("Enter array elements:");
   for(i=0;i<num;++i)
      scanf("%d",&array[i]);
   printf("Enter element to search:");
   scanf("%d",&target);
   for(i=0;i<num;++i)
      if(array[i]==target)
         break;
   if(i<num)
      printf("Target element found at location %d",i);
   else
      printf("Target element not found in an array");
   return 0;
}
```

The output of linear search algorithm code:

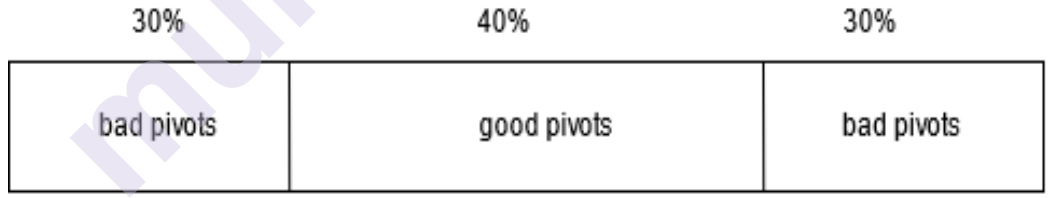


5.5 MEDIAN OF MEDIANS ALGORITHM

Median of Medians Algorithm

It is a divide and conquer algorithm in that, it returns a pivot that in the worst case will divide a list of unsorted elements into sub-problems of size 3n/10 and 7n/10 assuming we choose a sublist size of 5.

It guarantees a good pivot that in the worst case will give a pivot in the range between 30th and 70th percentile of the list of size n. For a pivot to be considered good it is essential for it to be around the middle, 30-70% guarantees the pivot will be around the middle 40% of the list.



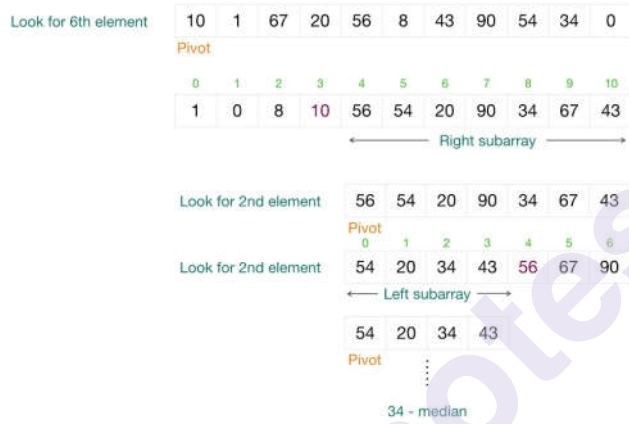
The algorithm is as follows,

1. Divide the list into sublists if size n, assume 5.
2. Initialize an empty array M to store medians we obtain from smaller sublists.
3. Loop through the whole list in sizes of 5, assuming our list is divisible by
4. For n/5 sublists, use select brute-force subroutine to select a median m, which is in the 3rd rank out of 5 elements.
5. Append medians obtained from the sublists to the array M.

6. Use quickSelect subroutine to find the true median from array M, The median obtained is the viable pivot.
7. Terminate the algorithm once the base case is hit, that is, when the sublist becomes small enough. Use Select brute-force subroutine to find the median.

Note: We used chunks of size 5 because selecting a median from a list whose size is an odd number is easier. Even numbers require additional computation. We could also select 7 or any other odd number as we shall see in the proofs below.

Here is the procedure pictorially;



Pseudocode of Median of Medians Algorithm

```
medianOfMedians(arr[1...n])
if(n <5returnSelect(arr[1...n], n/2))
   Let M be an empty list
```

```
   For i from 0 to n/5-1:
      Let m =Select(arr[5i +1...5i+5],3)
      Add m to M
   Return QuickSelect(M[1...n/5], n/10)
End medianOfMedians
```

It is recursive, it calls QuickSelect which in turn will call Median Of Medians.

Time and Space Complexity of Median of Medians Algorithm

This algorithm runs in **O(n)** linear time complexity, we traverse the list once to find medians in sublists and another time to find the true median to be used as a pivot.

The space complexity is **O(logn)** , memory used will be proportional to the size of the lists.

Proofs: Why is this pivot good?

1. **Lemma:**

The median of medians will return a pivot element that is greater than and less than at least 30% of all elements in the whole list.

proof:
Array M consists of n/5 medians of sub lists of size 5, these elements in list M is greater than and less than at-least two elements in the original list.

QuickSelect will return a true median that represents the whole list which is greater than and less than n/5/2 elements of list M and since each one of the M elements is greater than and less than at least two other elements in their previous sublists, therefore the true median is greater than and less than at least 3n/10, 30 percentile of elements of the whole list.

2. Lemma:

With that, we guarantee that quickselect finds a good pivot in linear time O(n) which in turn guarantees quickSort's worst case to be O(nlogn).

proof:

Recurrence relation;

$T(n) = \{c \quad if(n \leq 1)$

$T(n/5) + T(7n/10) + dn \qquad if(n > 1)$

Theorem:

1. $T(n) \leq k \cdot n \quad$ for $n \geq n'$

2. $T(n) = T(n/5) + T(7n/10) + dn$

Base Case:

$n = 1$

$T(1) = c \leq k \cdot 1$ therefore $k \geq c$

Induction Hypothesis:

Assume, $T(j) \leq b \cdot k \quad for(k \leq n)$

Induction Step:

$T(n) = T(n5) + T(7n/10) + dn$

$T(n) \leq k \cdot n/5 + k \cdot 7n/10 + dn = 9/10kn + dn$

By induction;

$T(n) \leq k \cdot n \quad \text{for } n \geq 1$

The above proof worked because $n/5 + 7n/10 < 1$,we split the original list in chunks of 5 assuming the original list is divisible by 5. We could also use another odd number provided the above equation results in a number below 1, then our theorem will perform its operations in O(n) linear time.

Therefore we get a big theta(n) time complexity for QuickSelect which proves using this heuristic for QuickSelect ad QuickSort improves worst case to O(n) and O(nlogn) for the respective algorithms.

1.6 Find the Kth smallest element in the sorted generated array

Given an array **arr[]** of **N** elements and an integer **K**, the task is to generate an **B[]** with the following rules:

1. Copy elements **arr[1…N]**, **N** times to array **B[]**.
2. Copy elements **arr[1…N/2]**, **2*N** times to array **B[]**.
3. Copy elements **arr[1…N/4]**, **3*N** times to array **B[]**.
4. Similarly, until only no element is left to be copied to array B[].

Finally print the **K**th smallest element from the array **B[]**. If **K** is out of bounds of **B[]** then return **-1**.

Examples:
Input: *arr[] = {1, 2, 3}, K = 4*
Output: *1*
{1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 1, 1, 1, 1, 1} is the required array B[] {1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3} in the sorted form where 1 is the 4th smallest element. ***Input:*** *arr[] = {2, 4, 5, 1}, K = 13*
Output: *2*

Approach:
Maintain a Count_Array where we must store the count of times every element occurs in array B[]. It can be done for range of elements by adding the count at start index and subtracting the same count at end index + 1 location.

1. Take cumulative sum of count array.
2. Maintain all elements of arr[] with their count in Array B[] along with their counts and sort them based on element value.
3. Traverse through vector and see which element has Kth position in B[] as per their individual counts.

4. If K is out of bounds of B[] then return -1.

Below is the implementation of the above approach:

```
# Python3 implementation of the approach
# Function to return the Kth element in B[]
def solve(Array, N, K) :
    # Initialize the count Array
    count_Arr = [0]*(N + 2) ;
    factor = 1;
    size = N;
    # Reduce N repeatedly to half its value
    while (size) :
        start = 1;
        end = size;
        # Add count to start
        count_Arr[1] += factor * N;
        # Subtract same count after end index
        count_Arr[end + 1] -= factor * N;
        factor += 1;
        size //= 2;
    for i in range(2, N + 1) :
        count_Arr[i] += count_Arr[i - 1];
    # Store each element of Array[] with their count
    element = [];
        for i in range(N) :
        element.append(( Array[i], count_Arr[i + 1] ));
    # Sort the elements wrt value
    element.sort();
    start = 1;
    for i in range(N) :
```

```
            end = start + element[i][1] - 1;
            # If Kth element is in range of element[i]
            # return element[i]
            if (K >= start and K <= end) :
                return element[i][0];
            start += element[i][1];
        # If K is out of bound
        return -1;
# Driver code
if __name__ == "__main__" :
        arr = [ 2, 4, 5, 1 ];
        N = len(arr);
        K = 13;
        print(solve(arr, N, K));
        # This code is contributed by AnkitRai01
```

Output:

2

Time Complexity: O(N * log N)

Auxiliary Space: O(N)

Find the kth element in the series generated by the given N ranges

Given **N** non-overlapping ranges **L[]** and **R[]** where the every range starts after the previous range ends i.e. **L[i] > R[i – 1]** for all valid **i**. The task is to find the **K^{th}** element in the series which is formed after sorting all the elements in all the given ranges in ascending order.

Examples:
Input: *L[] = {1, 8, 21}, R[] = {4, 10, 23}, K = 6*
Output: *9*
The generated series will be 1, 2, 3, 4, 8, 9, 10, 21, 22, 23
And the 6^{th} element is 9
Input: *L[] = {2, 11, 31}, R[] = {7, 15, 43}, K = 13*
Output: *32*

Approach: The idea is to use binary search. An array **total** to store the number of integers that are present upto **i^{th}** index, now with the help of this array find out the index in which the **k^{th}** integer will lie. Suppose that index is **j**, now compute the position of the **k^{th}** smallest integer in the interval **L[j] to R[j]** and find the **k^{th}** smallest integer using binary search where **low** will be **L[j]** and **high** will be **R[j]**. Below is the implementation of the above approach:

```
# Python3 implementation of the approach

# Function to return the kth element

# of the required series

def getKthElement(n, k, L, R):

    l = 1

    h = n

    # To store the number of integers that lie

    # upto the ith index

    total=[0 for i in range(n + 1)]

    total[0] = 0

    # Compute the number of integers

    for i in range(n):

        total[i + 1] = total[i] + (R[i] - L[i]) + 1

    # Stores the index, lying from 1

    # to n,

    index = -1

    # Using binary search, find the index

    # in which the kth element will lie

    while (l <= h):

        m = (l + h) // 2

        if (total[m] > k):

            index = m

            h = m - 1

        elif (total[m] < k):

            l = m + 1
```

```
            else :
                index = m
                break
        l = L[index - 1]
        h = R[index - 1]
        # Find the position of the kth element
        # in the interval in which it lies
        x = k - total[index - 1]
        while (l <= h):
            m = (l + h) // 2
            if ((m - L[index - 1]) + 1 == x):
                return m
            elif ((m - L[index - 1]) + 1 > x):
                h = m - 1
            else:
                l = m + 1
# Driver code
L=[ 1, 8, 21]
R=[4, 10, 23]
n = len(L)
k = 6
print(getKthElement(n, k, L, R))
```

Output:

9

Time Complexity: O(N) **Auxiliary Space:** O(N)

5.6 FIND THE KTH SMALLEST ELEMENT FROM AN ARRAY USING SORTING

Finding the*kth* smallest element of an array (efficiently) may seem a little tricky at first; however, it can easily be found using a min-heap or by sorting the array.

Algorithm

The following steps are involved in finding the*kth* smallest element using sorting.

1. Sort the given array in ascending order using a sorting algorithm like merge sort, bubble sort, or heap sort.
2. Return the element at index $k-1$ in the sorted array.

The illustration below further explains this concept:

Implementation

The following code snippet implements the above algorithm in C++:

```
#include <iostream>
#include<algorithm>
using namespace std;
int main() {
  int arr[] = {10, 1, 0, 8, 7, 2};
  int size = sizeof(arr)/sizeof(arr[0]);
  // Sorting the array
  sort(arr, arr + size);
  // Finding the third smallest element in the array
cout<< "The third smallest element in the array is: "<<
arr[2]<<endl;
  return 0;
}
```


Find the Kth smallest element in the sorted generated array

Given an array **arr[]** of **N** elements and an integer **K**, the task is to generate an **B[]** with the following rules:

1. Copy elements **arr[1…N]**, **N** times to array **B[]**.
2. Copy elements **arr[1…N/2]**, **2*N** times to array **B[]**.
3. Copy elements **arr[1…N/4]**, **3*N** times to array **B[]**.
4. Similarly, until only no element is left to be copied to array B[].

Finally print the **K**th smallest element from the array **B[]**. If **K** is out of bounds of **B[]** then return **-1**.

Examples:
Input: *arr[] = {1, 2, 3}, K = 4*
Output: *1*
{1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 1, 1, 1, 1, 1} is the required array B[]

{1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3} in the sorted form where 1 is the 4th smallest element.
Input: *arr[] = {2, 4, 5, 1}, K = 13*
Output: *2*
Approach:

1. Maintain a Count_Array where we must store the count of times every element occurs in array B[]. It can be done for range of elements by adding the count at start index and subtracting the same count at end index + 1 location.
2. Take cumulative sum of count array.
3. Maintain all elements of arr[] with their count in Array B[] along with their counts and sort them based on element value.
4. Traverse through vector and see which element has Kth position in B[] as per their individual counts.
5. If K is out of bounds of B[] then return -1.

Below is the implementation of the above approach:

```
# Python3 implementation of the approach

# Function to return the Kth element in B[]

def solve(Array, N, K) :

    # Initialize the count Array

    count_Arr=[0]*(N +2) ;

    factor =1;

    size =N;

     # Reduce N repeatedly to half its value

    while(size) :

        start =1;

        end =size;

         # Add count to start

        count_Arr[1] +=factor *N;

         # Subtract same count after end index

        count_Arr[end +1] -=factor *N;

        factor +=1;
```

```
        size //=2;
    foriinrange(2, N +1) :
        count_Arr[i] +=count_Arr[i-1];
    # Store each element of Array[] with their count
    element =[];
        foriinrange(N) :
        element.append(( Array[i], count_Arr[i+1] ));
    # Sort the elements wrt value
    element.sort();
    start =1;
    foriinrange(N) :
        end =start +element[i][1] -1;
        # If Kth element is in range of element[i]
        # return element[i]
        if(K >=start andK <=end) :
            returnelement[i][0];
        start +=element[i][1];
    # If K is out of bound
    return-1;
  # Driver code
if__name__ =="__main__":
    arr=[ 2, 4, 5, 1];
    N =len(arr);
    K =13;
    print(solve(arr, N, K));
    # This code is contributed by AnkitRai01
```

Output:

2

Time Complexity: O(N * log N)

Auxiliary Space: O(N)

Exercise end up

1. What is meant by sorting?
2. What are the two main classifications of sorting based on the source of data?
3. What is meant by external and internal sorting?
4. What is the purpose of quick sort?
5. What is the advantage of quick sort?
6. What is the purpose of insertion sort?
7. Define merge sort.
8. What are the advantages of merge sort?
9. What is linear search?
10. What is binary search?
11. Differentiate linear search and binary search.
12. Differentiate quick sort and merge
13. Give the advantage of merge sort
14. Distinguish quick sort and insertion sort.
15. Define sorting.
16. Narrate insertion sort with example
17. List examples for various sorting
18. Give the advantage of Merge sort
19. List linear search and binary search with example
20. Narrate insertion sort with example

1. Write and trace the following algorithms with suitable example.

I. Breadth first traversal II. Depth first traversal

2. Sort the sequence 3,1,4,1,5,9,2,6,5 using insertion sort.

3. Give short notes of : (i) Merge sort with suitable example.

4. Write an algorithm to sort 'n' numbers using quicksort.
5. Show how the following numbers are sorted using quicksort : 42, 28, 90,2, 56. 39, 12, 87 5. Sort the sequence 13,11, 74,37,85,39,22,56,25 using Insertion sort.
6. Explain the operation and implementation of merge sort.
7. Explain the operation and implementation of external sorting.
8. Write quick sort algorithm
9. Trace the quick sort algorithm for the following list of numbers. 90,77,60,99,55,88,66
10. Write down the merge sort algorithm and give its worst case, best case and average case analysis.
11. Explain linear search algorithm with an example.
12. Explain linear search & binary search algorithm in detail.
13. Briefly differentiate linear search algorithm with binary search algorithm.

5.7 LIST OF REFERENCES

List of reference materials used in this book.

- Quicksort, C.A.R. Hoare, Comput. J., 5 (1962), p. 10-15.
- Sorting, Lloyd Allison, csse.monash.edu.au/~lloyd/tildeAlgDS/Sort.
- Sequential and parallel sorting algorithms, H.W. Lang, iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm.

5.8 BIBILOGRAPHY

1 In computer science, selection sort is an in-place comparison sorting algorithm. It has an O(n^2) time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

2 The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

3 The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort.

4 Example

Here is an example of this sort algorithm sorting five elements:

Sorted sublist	Unsorted sublist	Least element in unsorted list
()	(11, 25, 12, 22, 64)	11
(11)	(25, 12, 22, 64)	12
(11, 12)	(25, 22, 64)	22
(11, 12, 22)	(25, 64)	25
(11, 12, 22, 25)	(64)	64
(11, 12, 22, 25, 64)	()	

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 12 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 22 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 25 64

Conclusion of selection sort algorithms

Conclusion: Selection sort is sorting algorithm known by its simplicity. Unfortunately, it lacks efficiency on huge lists of items, and also, it does not stop unless the number of iterations has been achieved (n-1, n is the number of elements) even though the list is already sorted.

❖❖❖❖

6

ALGORITHM AND DESIGN TECHNIQUES

Unit Structure :

6.0 OBJECTIVES

After this chapter, you will be

- Able to Understand Algorithm design and techniques.
- Able to develop your own versions for a given computational task and to compare and contrast their performance.
- Able to apply important algorithmic design paradigms and methods of analysis.
- Able to select appropriate algorithms for a given problem, integrate multiple algorithms effectively.

6.1 INTRODUCTION

Before solving a new problem, the general tendency is to look for the similarity of the current problem to other problems for which we have solutions. This helps us in getting the solution easily. In this chapter, we will see different ways of classifying the algorithms and in subsequent chapters we will focus on a few of them (Greedy, Divide and Conquer, Dynamic Programming).

6.2 ALGORITHM & IT'S CLASSIFICATION

The word Algorithm means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations " Or " A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations". Therefore an Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input. The algorithms can be classified in various ways. They are:

1. Implementation Method
2. Design Method
3. Design Approaches
4. Other Classifications

- The different algorithms in each classification method are discussed below.

6.2.1 Classification By Implementation Method

An algorithm may be implemented according to different basical principles. There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

- **Recursion or Iteration:**

  A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/ or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.

  Example: The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.

- **Exact or Approximate:**

  Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem. Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

- **Serial or Parallel or Distributed Algorithms:**

  In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel

algorithms are distributed on different machines, then they are known as distributed algorithms.

6.2.2 Classification By Design Method

There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

- Greedy Method: Greedy algorithms work in stages. In each stage, a decision is made that is good at that point. It assumes that the local best selection also makes for the global optimal solution. In the greedy method, at each step, a decision is made to choose the ***local optimum*****,** without thinking about the future consequences. Example: Fractional Knapsack, Activity Selec

- Divide and Conquer: The D & C strategy solves a problem by:

  - **Divide:** Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.

  - **Recursion:** Recursively solving these sub problems.

  - **Conquer:** Appropriately combining their answers.

The Divide and Conquer strategy involves dividing the problem into sub problem, recursively solving them, and then recombining them for the final answer. Example: Merge sort, Quick sort.

- **Dynamic Programming:**

  The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table. Dynamic programming (DP) and memoization work together. The difference between DP and divide and conquer is that in the case of the latter there is no dependency among the sub problems, whereas in DP there will be an overlap of sub-problems. By using memoization [maintaining a table for already solved sub problems] , DP reduces the exponential complexity to polynomial complexity for many problems.

  Example: 0-1 Knapsack, subset-sum problem.

- **Linear Programming:**

  In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs. Example: Maximum flow of Directed Graph.

- **Reduction (Transform and Conquer):**

  In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer.*

- **Backtracking:**

  This technique is very useful in solving combinatorial problems that have *a **single unique solution***. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution. Example: N-queen problem, maize problem.

- Branch and Bound: This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution. Example: Job sequencing, Travelling salesman problem.

6.2.3 Classification By Design Approaches

There are two approaches for designing an algorithm these approaches include

1. Top-Down Approach :
2. Bottom-up approach

- Top-Down Approach: In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.

- Bottom-up approach: The bottom-up approach is also known as the reverse of top-down approaches. In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.

6.2.4 Other Classifications

Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

- **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.

  Example: Randomized Quicksort Algorithm.

- **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis. Example: Some algorithms take O(n), while some take exponential time.

- **Classification by Research Area:** In computer science each field has its own problems and needs efficient algorithms. Examples: search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, geometric algorithms, combinatorial algorithms, machine learning, cryptography, parallel algorithms, data compression algorithms, parsing techniques, and more.

- **Branch and Bound Enumeration and Backtracking:** These are mostly used in Artificial Intelligence.

6.3 SUMMARY

- In this chapter we have foremost seen an introduction about algorithm. There are many ways of classifying algorithms and a few of them we have discuss in brief. We have classify algorithm into Implementation Method, Design Approaches, Design Method and few other types of classifications.

6.4 QUESTIONS

1. Explain Classification by implementation method.
2. Explain classification by design approaches.
3. Explain classification by design method.
4. Explain Other Classification types.

6.5 REFERENCES

1. Data Structure and Algorithmic Thinking with Python, Narasimha Karumanchi , CareerMonk Publications, 2016.
2. Introduction to Algorithm, Thomas H Cormen, PHI

7

GREEDY ALGORITHMS

Unit Structure :

7.0 OBJECTIVES

- In an algorithm design there is no one 'silver bullet' that is a cure for all computation problems. Different problems require the use of different kinds of techniques.
- To be able to be a good programmer we must know to use all the different techniques based on the type of problem.
- After this chapter the learners will be able to understand how to build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

7.1 INTRODUCTION

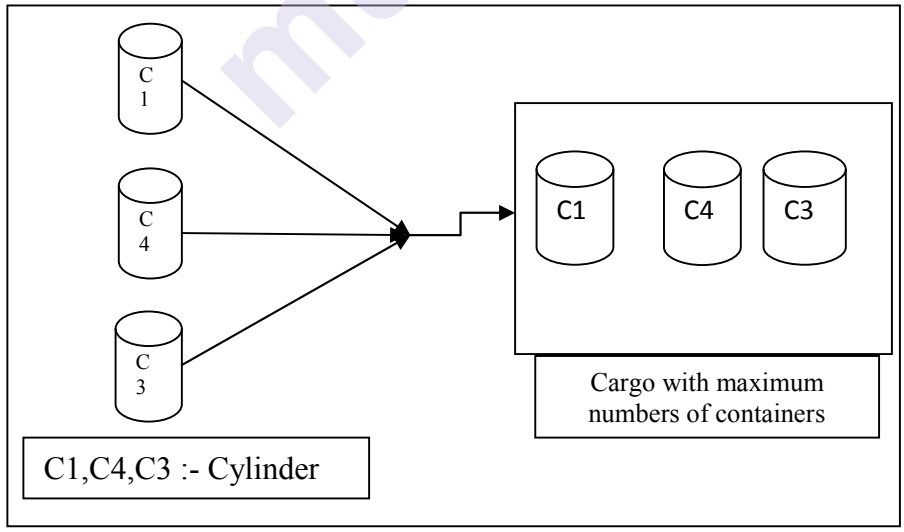
Let us start our discussion with simple theory that will give us an understanding of the Greedy technique. In the game of Chess, every time we make a decision about a move, we have to also think about the future consequences. Whereas, in the game of Tennis (or Volleyball), our action is based on the immediate situation. This means that in some cases making a decision that looks right at that moment gives the best solution (Greedy), but in other cases it doesn't. The Greedy technique is best suited for looking at the immediate situation.

7.2 WHAT IS GREEDY ALGORITHM?

- It simply means to pick up a choice/solution that seems the best at the moment **(being greedy)**. This technique is best suited when we want an immediate situation. It helps to solve optimization problems i.e. which gives either minimum results or maximum results.

- A solution satisfying the condition in the problem is a feasible solution. The solution having minimum cost out of all possible feasible solutions is the optimal solution i.e. it is the best solution.

- The goal of the greedy algorithm is to find the optimal solution. There can be only 1 optimal solution. For Example: Let us consider the example of the container loading problem. In a container loading problem, a large ship is loaded with cargo. The cargo has containers of equal sizes but different weights. The cargo capacity is of the ship is cp. We need to load the containers in the ship in such a way that the ship has maximum containers. For example, let us say there are a total of 4 containers with weights: w1=20, w2=75, w3=40, w4=20. Let cp = 80. Then, to load maximum containers we will load container-1,3, 4.

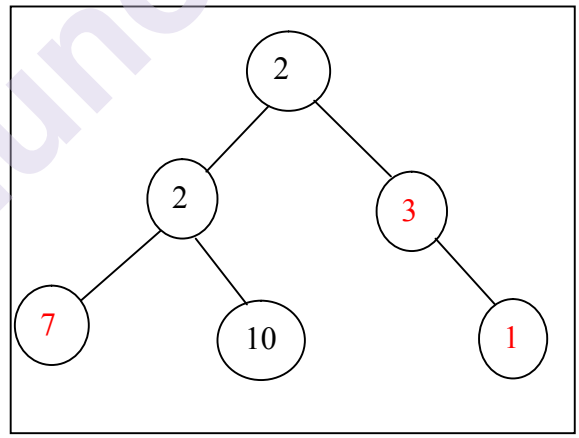


- We will load the containers in the order of increasing weights so that maximum containers load inside the ship. Here, the order of loading is container1→container4→container3.

7.3 GREEDY STRATEGY

- Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some local best is chosen. It assumes that a local good selection makes for a global optimal solution.
- The choice made by the greedy approach does not consider future data and choices. In some cases making a decision that looks right at that moment gives the best solution (Greedy), but in other cases, it doesn't.
- The greedy technique is used for optimization problems (where we have to find the maximum or minimum of something). The Greedy technique is best suited for looking at the immediate situation. Let us study Greedy Strategy with an example:

1. Let's start with the root node 20. The weight of the right child is 3 and the weight of the left child is 2.
2. Our problem is to find the largest path. And, the optimal solution at the moment is 3. So, the greedy algorithm will choose 3.
3. Finally the weight of an only child of 3 is 1. This gives us our final result 20 + 3 + 1 = 24.
4. However, it is not the optimal solution. There is another path that carries more weight (20 + 2 + 10 = 32) as shown in the image below.



- Therefore, greedy algorithms do not always give an optimal/feasible solution.

7.3.1 Elements Of Greedy Algorithms

- Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision. The two basic properties of optimal Greedy algorithms are:

- Greedy choice property.
- Optimal substructure.

- **Greedy choice property:-** This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

- **Optimal substructure:-** A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solutions to solve larger problems.

7.3.2 Does greedy always work?

- Making locally optimal choices does not always work. Hence, Greedy algorithms will not always give the best solutions.

7.3.3 Why to use the greedy algorithm?

- In many cases greedy algorithm is the most intuitive approach to solve a given optimization problem, that is, it is often the first guessed solution and is easy to formulate.

- Correct greedy algorithms are often more powerful and fast as compared to other approaches such as dynamic programming. This is because the greedy approach has to consider only one locally optimal choice while moving ahead with the solution, while dynamic programming must check all possible cases.

- Greedy algorithms work for a wide range of problems (such listed here), many of which have real-life applications in fields such as designing networking protocols and scheduling multiple events.

7.4 ADVANTAGES AND DISADVANTAGES OF GREEDY METHOD

- Easy to understand
- Easy to code
- We do not have to spend time re-examining the already computed values.
- Typically have less time complexity.
- **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.

- Greedy algorithms can be used for optimization purposes or finding close to optimization in case of Hard problems.
- The **difficult part** is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct.
- The local optimal solution may not always be globally optimal.
- Not applicable for many problems

7.5 GREEDY APPLICATIONS

- Sorting: Selection sort, Topological sort.
- Priority Queues: Heap sort.
- Huffman coding compression algorithm.
- Prim's and Kruskal's algorithms.
- Shortest path in Weighted Graph [Dijkstra's].
- Coin change problem.
- Fractional Knapsack problem.
- Disjoint sets-UNION by size and UNION by height (or rank).
- Job scheduling algorithm.
- Greedy techniques can be used as an approximation algorithm for complex problems.

7.6 UNDERSTANDING GREEDY TECHNIQUE

7.6.1 Huffman Coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. When reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character ′e′ is used 10 times more frequently than the character ′q′. It would then be advantageous for us to instead use a 7 bit code for e and a 9 bit code for q because that could reduce our overall message length.

- On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for

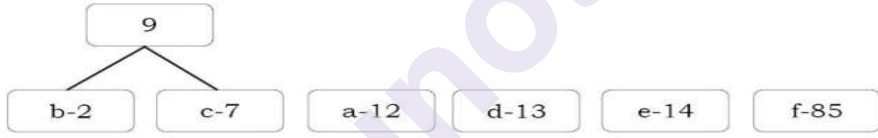
less frequent characters and groups of characters. Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

- **An ExampleLet's assume that after scanning a file we find the following character frequencies**

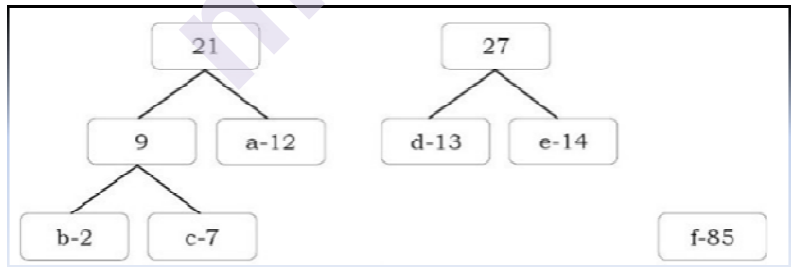
Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

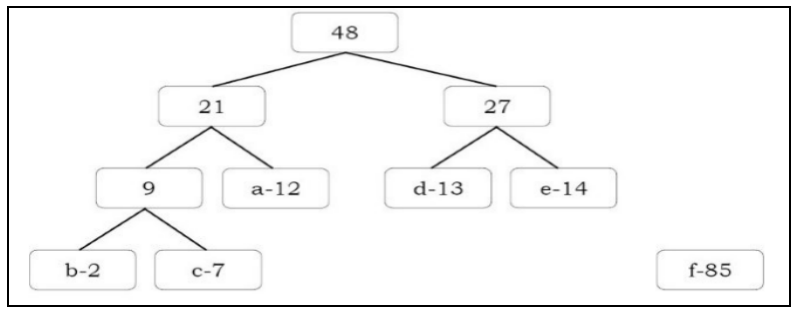
Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below):-

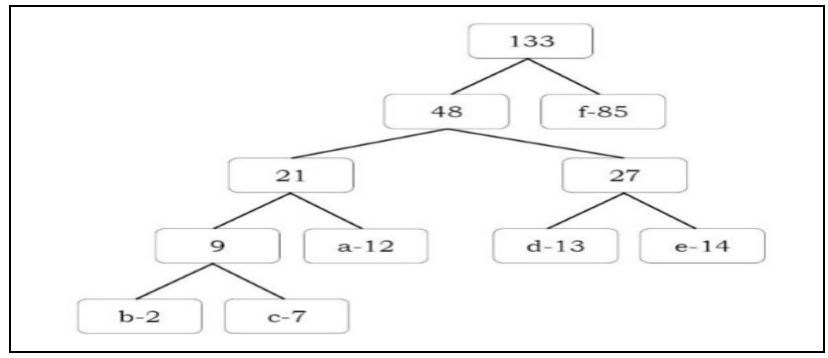
- The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like this:



- Repeat this process until only one tree is left:







- Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

➢ Calculating Bits Saved

- Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that the number of bits that are used to store the data using the Huffman code. In the above example, since we have six characters, let's assume each character is stored with a three bit code.

- Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is 3 * 133 = 399. Using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

- Thus, we saved 399 – 238 = 161 bits, or nearly 40% of the storage space.

```
HuffmanCodingAlgorithm(int A[], int n) {
    Initialize a priority queue, PQ, to contain the n elements in A;
    struct BinaryTreeNode *temp;
    for (i = 1; i<n; i++) {
        temp  = (struct *)malloc(sizeof(BinaryTreeNode));
        temp→left = Delete-Min(PQ);
        temp→right = Delete-Min(PQ);
        temp→data = temp→left→data + temp→right→data;
        Insert temp to PQ;
    }
    return PQ;
}
```

- Time Complexity: O(nlogn), since there will be one build_heap, 2n – 2 delete_mins, and n – 2 inserts, on a priority queue that never has more than n elements.

7.6.2 Fractional knapsack problem

- The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.
- There are basically three approaches to solve the problem:
  - The first approach is to select the item based on the maximum profit.
  - The second approach is to select the item based on the minimum weight.
  - The third approach is to calculate the ratio of profit/weight.
- It is an optimization problem in which we can select fractional items rather than binary choices. The objective is to obtain a filling knapsack that maximizes the total profit earned. Consider the below example:

OBJECTS	1	2	3	4	5	6	7
PROFIT(p)	5	10	15	7	8	9	4
WEIGHT(w)	1	3	5	4	1	3	2
W:Weight of the knapsack	15						
n (no of items)	7						

➢ **First approach:** The first approach is to select the item based on the maximum profit.

Object	Profit	Weight	Remaining weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
7	7 * ¾ = 5.25	3	3 - 3 = 0

- **The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25**

➢ **Second approach:** The second approach is to select the item based on the minimum weight.

Object	Profit	Weight	Remaining weight
1	5	1	15 - 1 = 14
5	7	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	15 * 1/5 = 3	1	1 - 1 = 0

- **In this case, the total profit would be equal to (5 + 7 + 4 + 10 + 9 + 7 + 3) = 45.**

Third approach:

- In the third approach, we will calculate the ratio of profit/weight.
- In this case, we first calculate the profit/weight ratio.
  1. Object 1: 5/1 = 5
  2. Object 2: 10/3 = 3. 33
  3. Object 3: 15/5 = 3
  4. Object 4: 7/4 = 1.7
  5. Object 5: 8/1 = 8

6. Object 6: 9/3 = 3
7. Object 7: 4/2 = 2

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0

- As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.
- In the first approach, the maximum profit is 47.25. The maximum profit in the second approach is 45. The maximum profit in the third approach is 51. Therefore, we can say that the third approach, i.e., maximum profit/weight ratio is the best approach among all the approaches.

7.7.3 Minimum Coin Change Problem

- The Minimum Coin Change problem is actually a variation of the problem where you find whether a change of the given amount exists or not
- Given a set of coins and a value, we have to find the minimum number of coins which satisfies the value.
- Example
  - coins[] = {5,10,20,25}
  - value = 50
- Possible Solutions:- {coin * count}
  - {5 * 10} = 50 [10 coins].
  - {5 * 8 + 10 * 1} = 50 [9 coins] goes on.
  - {10 * 5} = 50 [5 coins].
  - {20 * 2 + 10 * 1} = 50 [3 coins].
  - {20 * 2 + 5 * 2} = 50 [4 coins].
  - {25 * 2} = 50 [2 coins] etc etc
- Best Solution:- Two 25 rupees. Total coins two. {25 * 2 }=50.

7.7 SUMMARY

In this chapter you learned what a greedy programming paradigm is and discovered different techniques and steps to build a greedy solution. The chapter also discussed applications and mentions the advantages and disadvantages of greedy algorithm. Towards the end the chapter introduced different examples to understand greedy techniques.

7.8 QUESTIONS

1. Explain Greedy Strategy with an example.
2. What are the elements of Greedy Strategy.
3. Give advantages and disadvantages of greedy method.
4. Explain in brief Huffman coding.
5. Write a note on Fractional KnapSack Problem.

7.9 REFERENCES

1. Data Structure and Algorithmic Thinking with Python, Narasimha Karumanchi, CareerMonk Publications, 2017.
2. Introduction to Algorithm, Thomas H Cormen, PHI.
3. https://www.javatpoint.com/fractional-knapsack-problem.
4. https://www.geeksforgeeks.org/
5. https://www.codechef.com/wiki/

8

DIVIDE AND CONQUER ALGORITHMS

Unit Structure :

8.0 OBJECTIVE

- After this chapter learners will understand about divide and conquer a powerful algorithm design technique.
- Learners will be able to solve many important problems such as mergesort, quicksort, calculating Fibonacci numbers, and performing matrix multiplication.
- Learners will be able to solve recursion based problem.

8.1 INTRODUCTION

- In the Greedy chapter, we have seen that for many problems the Greedy strategy failed to provide optimal solutions. Among those problems, there are some that can be easily solved by using the Divide and Conquer (D & C) technique. Divide and Conquer is an important algorithm design technique based on recursion.

The D & C algorithm works by recursively breaking down a problem into two or more sub problems of the same type, until they become simple enough to be solved directly. The solutions to the sub problems are then combined to give a solution to the original problem.

8.2 WHAT IS DIVIDE AND CONQUER STRATEGY?

Many useful algorithms are recursive in structure, i.e., to solve the given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow a divide-and-conquer algorithm. The divide and conquer algorithm involves three steps at each level of recursion.

- The D & C strategy solves a problem by:
  - **Divide:** Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
  - **Recursion:** Recursively solving these sub problems.
  - **Conquer:** Appropriately combining their answer.

8.3 DOES DIVIDE AND CONQUER ALWAYS WORK?

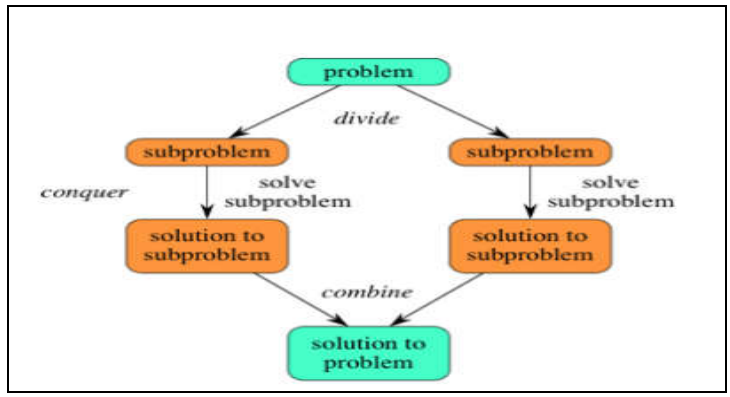
It's not possible to solve all the problems with the Divide & Conquer technique. As per the definition of D & C, the recursion solves the subproblems which are of the same type. For all problems it is not possible to find the subproblems which are the same size and D & C is not a choice for all problems.

8.4 DIVIDE AND CONQUER VISUALIZATION

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into small pieces (sub-problem), decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

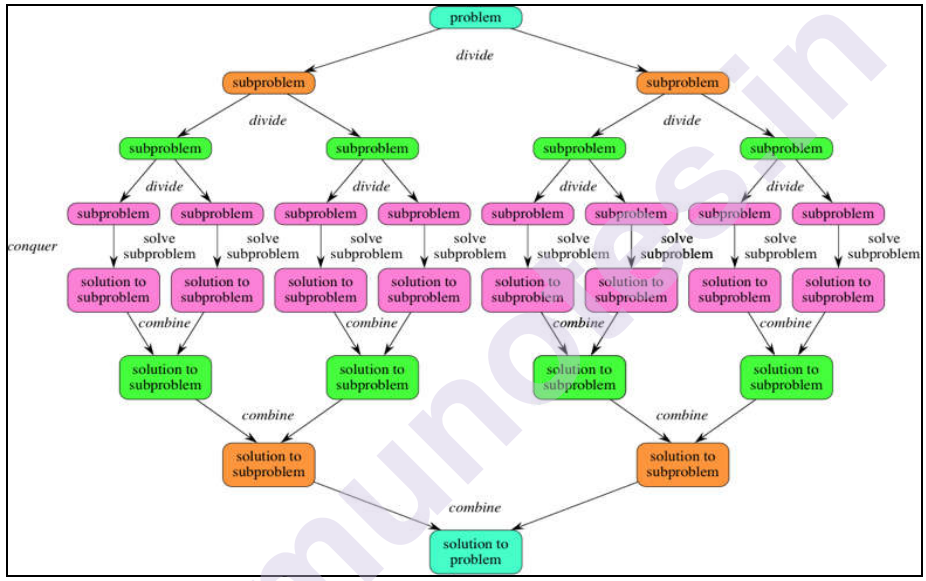
- For better understanding, consider the following visualization. Assume that n is the size of the original problem. As described above, we can see that the problem is divided into sub problems with each of size n/b (for some constant b). We solve the sub problems recursively and combine their solutions to get the solution for the original problem.

- Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



Source:
https://www.khanacademy.org

- If we expand out two more recursive steps, it looks like this:



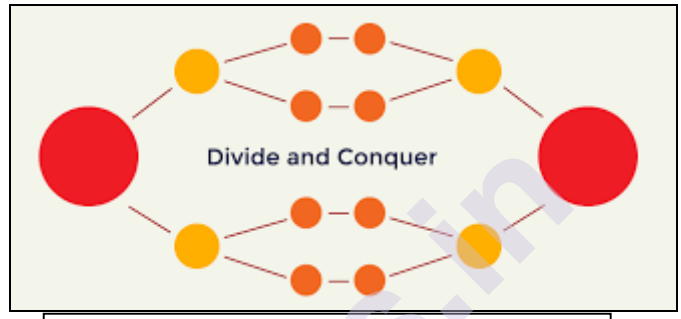
Source:
https://www.khanacademy.org

8.5 UNDERSTANDING DIVIDE AND CONQUER

- For a clear understanding of D & C, let us consider a story. There was an old man who was a rich farmer and had seven sons. He was afraid that when he died, his land and his possessions would be divided among his seven sons, and that they would quarrel with one another. So he gathered them together and showed them seven sticks that he had tied together and told them that anyone who could break the bundle would inherit everything. They all tried, but no one could break the bundle. Then the old man untied the bundle and broke the sticks one by one. The brothers decided that they should stay together and work together and succeed together.

- The moral for problem solvers is different. If we can't solve the problem, divide it into parts, and solve one part at a time. Therefore a divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems, solving the sub-problems, and combining them to get the desired output. To use the divide and conquer algorithm, recursion is used.

- The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity are solved directly.



Source https://mobisoftinfotech.com/resources/blog/

- Example of Divide and Conquer Algorithm:- Here, we will sort an array using the divide and conquer approach, i.e. merge sort.

1. Assume the given array is:

6	5	1	4	3	2

2. Break the array into two segments:- Recursively split each subpart again into two parts until you have separate elements.

3. Merge the individual parts in an ordered manner. Here, the steps to conquer and merge go hand in hand.

8.6 ADVANTAGES OF DIVIDE AND CONQUER (D&C)

8.6.1 Advantages Of Divide And Conquer (D&C)

1. **Solving difficult problems:** D & C is a powerful method for solving difficult problems. As an example, consider the Tower of Hanoi problem. This requires breaking the problem into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that subproblems can be combined again is a major difficulty in designing a new algorithm. For many such problems D & C provides a simple solution.

2. **Parallelism:** Since D & C allows us to solve the subproblems independently, this allows for execution in multiprocessor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because different subproblems can be executed on different processors.

3. **Memory access:** D & C algorithms naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache, without accessing the slower main memory.

8.6.2 DISADVANTAGES OF DIVIDE AND CONQUER (D&C)

- One disadvantage of this approach is that recursion is slow. This is beacause of the overhead of the repeated subproblem calls. Also the algorithm need stack for storing the calls. Actually this depends upon the implementation style. With large enough recursive base cases , the overhead of recursion can become negligible for many problems.

- Another problem with D & C is that, for some problems, it may be more complicated than an iterative approach. For example, to add n numbers, a simple loop to add them up in sequence is much easier than a D & C approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.

- An explicit stack may overuse the space.

- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

8.7 MASTER THEOREM

- As stated above, in the D & C method, we solve the sub problems recursively. All problems are generally defined in terms of recursive definitions. These recursive problems can easily be solved using Master theorem.

- The above algorithm divides the problem into a subproblems, each of size n/b and solve them recursively to compute the problem and the extra work done for problem is given by f(n), i.e., the time to create the subproblems and combine their results in the above procedure.

- So, according to master theorem the runtime of the above algorithm can be expressed as:

$T(n) = aT(n/b) + f(n)$ where n = size of the problem

a = number of subproblems in the recursion and a >= 1.

n/b = size of each subproblem.

f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

- If the recurrence is of the form $T(n)=aT\frac{n}{b} + \theta(n^k \log^p n)$, where a ≥ 1, b > 1, k ≥ 0 and p is a real number, then the complexity can be directly given as:

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log\log n)$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$
3) If $a < b^k$
   a. If $p > 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1. Example-1: Binary Search – T(n) = T(n/2) + O(1)

   a = 1, b = 2, k = 0 and p = 0

   b^k = 1. So, a = b^k and p > -1 [Case 2.(a)]

   T(n) = θ($n^{\log_b a} \log^{p+1} n$)

   T(n) = θ(logn)

2. Example-2: Merge Sort – T(n) = 2T(n/2) + O(n)

   a = 2, b = 2, k = 1, p = 0

   b^k = 2. So, a = b^k and p > -1 [Case 2.(a)]

   T(n) = θ($n^{\log_b a} \log^{p+1} n$)

   T(n) = θ(nlogn)

3. Example-3: T(n) = 3T(n/2) + n2

   a = 3, b = 2, k = 2, p = 0

   b^k = 4. So, a < b^k and p = 0 [Case 3.(a)]

   T(n) = θ($n^k \log^p$n)

   T(n) = θ(n^2)

8.8 DIVIDE AND CONQUER APPLICATIONS

1. **Binary Search:** Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array.

2. **Merge Sort:** Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm. Here, a problem is divided into multiple sub-problems. Each sub-problem is

solved individually. Finally, sub-problems are combined to form the final solution.

3. **Quick Sort:** Quicksort is a sorting algorithm based on the divide and conquer approach where

   a. An array is divided into subarrays by selecting a pivot element (element selected from the array).

   b. While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

   c. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

4. **Median Finding:** The median-of-medians algorithm is a deterministic linear-time selection algorithm. The algorithm works by dividing a list into sublists and then determines the approximate median in each of the sublists. Then, it takes those medians and puts them into a list and finds the median of that list. It uses that median value as a pivot and compares other elements of the list against the pivot. If an element is less than the pivot value, the element is placed to the left of the pivot, and if the element has a value greater than the pivot, it is placed to the right. The algorithm recurses on the list, honing in on the value it is looking for.

5. **Min and Max Finding:** Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach. Divide and conquer approach for Max. Min problem works in three stages.

   a. If a1 is the only element in the array, a1 is the maximum and minimum.

   b. If the array contains only two elements a1 and a2, then the single comparison between two elements can decide the minimum and maximum of them.

   c. If there are more than two elements, the algorithm divides the array from the middle and creates two subproblems. Both subproblems are treated as an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.

After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build the solution of a parent problem.

6. **Matrix Multiplication:** Matrix multiplication is an important operation in many mathematical and image processing applications. Suppose we want to multiply two matrices A and B, each of size n x n, multiplication is operation is defined as

$$\begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} * \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

$$C_{11} = A_{11}\ B_{11} + A_{12}\ B_{21}$$

$$C_{12} = A_{11}\ B_{12} + A_{12}\ B_{22}$$

$$C_{21} = A_{21}\ B_{11} + A_{22}\ B_{21}$$

$$C_{22} = A_{21}\ B_{12} + A_{22}\ B_{22}$$

This approach is very costly as it required 8 multiplications and 4 additions.

8.9 DIVIDE AND CONQUER: PROBLEMS & SOLUTIONS

Problem-1 Let us consider an algorithm A which solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time. What is the complexity of this algorithm?

Solution: Let us assume that the input size is n and T(n) defines the solution to the given problem. As per the description, the algorithm divides the problem into 5 sub problems with each of size $\frac{n}{2}$. So we need to solve 5T($\frac{n}{2}$). subproblems. After solving these sub problems, the given array (linear time) is scanned to combine these solutions. The total recurrence algorithm for this problem can be given as: T(n)=5T ($\frac{n}{2}$) + O(n) . Using the Master theorem (of D & C), we get the complexity O($n^{log\frac{5}{2}}$) $\approx$ O(n^{2+}) $\approx$ O(n^3).

Problem 2:- We are given two sorted lists of size n. Give an algorithm for finding the median element in the union of the two lists.

Solution: We use the Merge Sort process. Use merge procedure of merge sort. Keep track of the count while comparing elements of two arrays. If the count becomes n (since there are 2n elements), we have reached the median. Take the average of the elements at indexes n – 1 and n in the merged array.

Time Complexity: O(n).

Problem 3:- Can we give the algorithm if the size of the two lists are not the same?

Solution: The solution is similar to the previous problem. Let us assume that the lengths of two

lists are m and n. In this case we need to stop when the counter reaches (m + n)/2.

Time Complexity: O((m + n)/2).

Problem 4:- Can we improve the time complexity of Problem-2 to O(logn)?

Solution: Yes, using the D & C approach. Let us assume that the given two lists are L1 and L2.

Algorithm:

1. Find the medians of the given sorted input arrays L1[] and L2[]. Assume that those medians are m1 and m2.

2. If m1 and m2 are equal then return m1 (or m2).

3. If m1 is greater than m2, then the final median will be below two sub arrays.

4. From first element of L1 to m1.

5. From m2 to last element of L2.

6. If m2 is greater than m1, then median is present in one of the two sub arrays below.

8. From m1 to last element of L1.

8. From first element of L2 to m2.

9. Repeat the above process until the size of both the sub arrays becomes 2.

10. If size of the two arrays is 2, then use the formula below to get the median.

11. Median = (max(L1[0],L2[0]) + min(L1[1],L2[1])/2

Problem 5:- We are testing "unbreakable" laptops and our goal is to find out how unbreakable they really are. In particular, we work in an n-story building and want to find out the lowest floor from which we can drop the laptop without breaking it (call this "the ceiling"). Suppose we are given two laptops and want to find the highest ceiling possible. Give an algorithm that minimizes the number of tries we need to make f(n) (hopefully, f(n) is sub-linear, as a linear f(n) yields a trivial solution).

Solution: For the given problem, we cannot use binary search as we cannot divide the problem and solve it recursively. Let us take an example for understanding the scenario. Let us say 14 is the answer. That means we need 14 drops to find the answer. First we drop from height 14, and if

it breaks we try all floors from 1 to 13. If it doesn't break then we are left 13 drops, so we will drop it from $14 + 13 + 1 = 28^{th}$ floor. The reason being if it breaks at the 28^{th} floor we can try all the floors from 15 to 27 in 12 drops (total of 14 drops). If it did not break, then we are left with 11

drops and we can try to figure out the floor in 14 drops. From the above example, it can be seen that we first tried with a gap of 14 floors, and then followed by 13 floors, then 12 and so on. So if the answer is k then we are trying the intervals at k, k – 1, k – 2 ....1. Given that the number of floors is n, we have to relate these two. Since the maximum floor from which we can try is n, the total skips should be less than n.

This gives:

$$= k + (k-1) + (k-2) + \text{----} + 1 \leq n$$

$$= \frac{k(k+1)}{2} \leq n$$

$$= k \leq \sqrt{n}$$

8.10 SUMMARY

- In this chapter we studied the Divide and Conquer strategy(D&C) which works in three stages as follows:
- Divide: This involves dividing the problem into smaller sub-problems.
- Conquer: Solve sub-problems by calling recursively until solved.
- Combine: Combine the sub-problems to get the final solution of the whole problem.
- We have also understood various advantages and disadvantages of this strategy.
- We studied about various algorithms under D&C techniques.
- At the end some problems and its solution using D&C technique was mentioned.

8.11 QUESTIONS

1. Write a short note on Divide and Conquer Strategy.
2. Explain advantages and disadvantages of divide and conquer.
3. Explain disadvantages of divide and conquer.
4. Explain any 5 applications of divide and conquer.
5. Describe master theorem in detail.

8.12 REFERENCES

1. Data Structure and Algorithmic Thinking with Python, Narasimha Karumanchi, CareerMonk Publications, 2016.
2. Introduction to Algorithm, Thomas H Cormen, PHI.
3. https://www.javatpoint.com/fractional-knapsack-problem.
4. https://www.geeksforgeeks.org/
5. https://www.codechef.com/wiki/

❖❖❖❖

9

DYNAMIC PROGRAMMING

Unit Structure

9.0 OBJECTIVE

- To gain an understanding of the technique of dynamic programming. To be able to adapt it to new problems. To be able to use dynamic programming to develop new logic.
- Towards the end of this chapter, you will have a better understanding of the recursion and dynamic programming approach
- Learners will also be able to analyze and solve various application using dynamic strategy.
- Learners will also be able to analyze the problem and will be able to recognize the order in which the sub-problems are solved and will start solving from the trivial subproblem, up towards the given problem.

- Learners will be able to reason about different dynamic algorithm correctness and complexity

9.1 INTRODUCTION

In this chapter we will try to solve the problems for which we failed to get the optimal solutions

using other techniques (say, Divide & Conquer and Greedy methods). Dynamic Programming (DP) is a simple technique but it can be difficult to master. One easy way to identify and solve DP problems is by solving as many problems as possible. The term Programming is not related to coding but it is from literature, and means filling tables (similar to Linear Programming).

9.2 WHAT IS DYNAMIC PROGRAMMING STRATEGY?

- Dynamic programming and memoization work together. The main difference between dynamic programming and divide and conquer is that in the case of the latter, sub problems are independent, whereas in DP there can be an overlap of sub problems. By using memorization [maintaining a table of sub problems already solved], dynamic programming reduces the exponential complexity to polynomial complexity (O(n^2),O(n^3),etc) for many problems.
- The major components of DP are:
  - **Recursion:** Solves sub problems recursively.
  - **Memoization:** Stores already computed values in table (Memoization means caching).
- Dynamic Programming = Recursion + Memoization.
- Therefore, dynamic programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

9.2.1 What is Recursion?

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.
- Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself

and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process.

- So we can say that every time the function calls itself with a simpler version of the original problem.

9.2.2 Difference between a dynamic programming algorithm and recursion?

- In dynamic programming, problems are solved by breaking them down into smaller ones to solve the larger ones, while recursion is when a function is called and executed by itself. While dynamic programming can function without making use of recursion techniques, since the purpose of dynamic programming is to optimize and accelerate the process, programmers usually make use of recursion techniques to accelerate and turn the process efficiently.
- When a function can execute a specific task by calling itself, receive the name of the recursive function. In order to perform and accomplish the work, this function calls itself when it has to be executed.
- Using dynamic programming, you can break a problem into smaller parts, called subproblems, to solve it. Dynamic programming involves solving the problem for the first time, then using memoization to store the solutions.
- Therefore, the main difference between the two techniques is their intended use; recursion is used to automate a function, whereas dynamic programming is an optimization technique used to solve problems.
- Recursive functions recognize when they are needed, execute themselves, then stop working. When the function identifies the moment it is needed, it calls itself and is executed; this is called a recursive case. As a result, the function must stop once the task is completed, known as the base case.
- By establishing states, dynamic programming recognizes the problem and divides it into sub-problems in order to solve the whole scene. After solving these sub-problems, or variables, the programmer must establish a mathematical relationship between them. Last but not least, these solutions and results are stored as algorithms, so they can be accessed in the future without having to solve the whole problem again.

9.3 PROPERTIES OF DYNAMIC PROGRAMMING STRATEGY

- The two dynamic programming properties which can tell whether it can solve the given problem or not are:
    - **Optimal substructure:** Optimal substructure property materializes when you can arrive at an optimal solution after constructing all the

other solutions that occurred from every subproblem you solved. The solution you calculate from each overlap applies to the overall problem in order to function and optimize recursion. In the example of the Fibonacci sequence, each subproblem contains a solution that you can apply to each successive subproblem to find the next number in the series, making the entire problem display optimal substructure property. Therefore we can conclude it means that an optimal solution to a problem contains optimal solutions to sub problems.

- **Overlapping sub problems:** Subproblems are smaller variations of an original, larger problem. For example, in the Fibonacci sequence, each number in the series is the sum of its two preceding numbers (0, 1, 1, 2, 3, 5, 8,...). If you want to calculate the nth Fibonacci value in the sequence, you can break down the entire problem into smaller subproblems. These subproblems then overlap with one another as you find solutions by solving the same subproblem repeatedly. The overlap in subproblems occurs with any problem, which allows you to apply dynamic programming to break down complex programming tasks into smaller parts. Therefore we can conclude that a recursive solution contains a small number of distinct sub problems repeated many times.

9.3.1 Can dynamic programming solve all problems?

- Like Greedy and Divide and Conquer techniques, DP cannot solve every problem. There are problems which cannot be solved by any algorithmic technique [Greedy, Divide and Conquer and Dynamic Programming].

- The difference between Dynamic Programming and straightforward recursion is in memorization of recursive calls. If the sub problems are independent and there is no repetition then memorization does not help, so dynamic programming is not a solution for all problems.

9.4 DYNAMIC PROGRAMMING APPROACHES

- Basically there are two approaches for solving DP problems:

1. Bottom-up dynamic programming.
2. Top-down dynamic programming

9.4.1 Bottom-Up Dynamic Programming

- In this method, we evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used.

- Analyze the problem and see in what order the subproblems are solved, and work your way up from the trivial subproblem to the given problem. This process ensures that the subproblems are solved before the main problem.

2. TOP-DOWN DYNAMIC PROGRAMMING

- In this method, the problem is broken into sub problems; each of these sub problems is solved; and the solutions remembered, in case they need to be solved. Also, we save each computed value as the final action of the recursive function, and as the first action we check if pre-computed value exists.

- You solve all the related sub-problems first instead of applying recursion. As bottom-up tabulation requires multiple solvencies, dynamic programming uses an n-dimensional table, where n represents a value of zero or greater. As you solve each subproblem within the table, you can then use the results to compute the original problem.

9.4.1 Bottom-Up Versus Top-Down Programming

- In bottom-up programming, the programmer has to select values to calculate and decide the order of calculation. In this case, all sub problems that might be needed are solved in advance and then used to build up solutions to larger problems. In top-down programming, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into sub problems, these sub problems are solved and the solutions remembered, in case they need to be solved again.

- Let us understand the difference between the two approaches with an example.

- Apply the Fibonacci sequence, where each number in the series represents the sum of the first two preceding numbers:

1. **Bottom-up Approach-** In this example, apply the Fibonacci sequence to break down the entire computation when you want to calculate the nth value in the series. With the same number sequence {0, 1, 1, 2, 3, 5, 8,...}, you can see that the next value in the series results in 13, since 5 and 8 give a sum of 13.

Using the bottom-up method to break down the entire problem Fib(n), apply the sequence equation [Fib(n - 1) + Fib(n - 2), when n > 1] for each subproblem you want to find an optimal solution for. Suppose you want to compute the next nth values when n = 23:

Fib(n) = Fib(n - 1) + Fib(n - 2) =

Fib(23) = Fib(23 - 1) + Fib(23 - 2) =

Fib(23) = Fib(22) + Fib(21) = Fib(43).

The result Fib(43) then applies to the computation of the next values in the series when n = 43:

Fib(n) = Fib(n - 1) + Fib(n - 2) =

Fib(43) = Fib(43 - 1) + Fib(43 - 2) =

Fib(n) = Fib(42) + Fib(41) = Fib(83) then

Fib(83) = Fib(83 - 1) + Fib(83 - 2) =

Fib(83) = Fib(82) + Fib(81) = Fib(163).

You can continue using this optimal solution for each subproblem to calculate the next values within the Fibonacci sequence, giving you the ability to break down the initial problem of Fib(n) = Fib(n - 1) + Fib(n - 2), when n > 1.

2. Top Down Approach: Understanding that the sum of the first two preceding values is the next number in the series can help you solve the entire problem when you want to calculate the nth Fibonacci number. Because you know the pattern for computing the future values in the series, you can apply a formula to determine the optimal solution without breaking down the problem into smaller subproblems. Using the appropriate formula when n > 1, you can compute an optimal solution with the top-down method when solving for an nth value of 13:

Fib(n) = Fib(n - 1) + Fib(n - 2) =

Fib(13) = Fib(13 - 1) + Fib(13 - 2) =

Fib(n) = Fib(12) + Fib(11) = 23

Using the top-down approach and applying memorization, you can cache the result of 23 in the database to input and call on your code string for additional tasks.

9.4.2 Which one is better?

- The top-down approach is typically recursive. It has the overhead of recursive calls and therefore is slower than the bottom-up approach.
- One might find the top-down approach easier to implement because we use an array of some sort of lookup table to store results during recursion. While for the bottom-up approach we need to define the order of iteration and define an n-dimensional table for storing results.
- The top-down approach might also run into stack overflow conditions in the case of a very deep recursion tree.

9.5 EXAMPLES OF DYNAMIC PROGRAMMING ALGORITHMS

1. **Many String algorithms** including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.

9.6 LONGEST COMMON SUBSTRING

- Let m and n be the lengths of the first and second strings respectively.
- A simple solution is to one by one consider all substrings of the first string and for every substring check if it is a substring in the second string. Keep track of the maximum length substring. There will be O(m^2) substrings and we can find whether a string is substring on another string in O(n) time (See this). So overall time complexity of this method would be O(n * m^2).
- Dynamic Programming can be used to find the longest common substring in O(m*n) time. The idea is to find the length of the longest common suffix for all substrings of both strings and store these lengths in a table.
- The longest common suffix has following optimal substructure property.
  - If last characters match, then we reduce both lengths by 1 :- LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 if X[m-1] = Y[n-1] .
  - If last characters do not match, then result is 0, i.e., :-

    LCSuff(X, Y, m, n) = 0 if (X[m-1] != Y[n-1]).
  - Now we consider suffixes of different substrings ending at different indexes. The maximum length Longest Common Suffix is the longest common substring.
  - LCSubStr(X, Y, m, n) = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m and 1 <= j <= n.
- **Given two strings 'X' and 'Y', find the length of the longest common substring.**

Input : X = "zxabcdezy", y = "yzabcdezx"

Output : 6

Explanation:

The longest common substring is "abcdez" and is of length 6.

EDIT DISTANCE

- Levenshtein (1966) introduced the edit distance between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) needed to transform one string into the other .
- The Edit distance is a problem to measure how much two strings are different from one another by counting the minimum number of operations required to convert one string into the other. Edit distance problem can be solved by many different approaches.But the most efficient approach to solve the Edit distance problem is Dynamic programming approach which takes the O(N * M) time complexity, where N and M are sizes of the strings.
- Edit distance has different definitions which uses different sets of string operations. Levenshtein distance operations is the basic set of operations which is used in Edit distance Problem.
- Operation allowed are:
  - Delete any character from the string.
  - Replace any character with any other.
  - Add any character into any part of the string.
- d(v,w) = MIN number of elementary operations to transform v → w **.**
- **Let us Transform** TGCATAT → ATCCGAT to understand the concept
  - TGCATAT (delete last T)
  - TGCATA (delete last A)
  - ATGCAT (insert A at front)
  - ATCCAT (substitute C for G)
  - ATCCGAT (insert G before last A) d(v,w)=5
- **EXAMPLE:- Input: str1 = "bat", str2 = "but"**
- **Output: d(v,w)=1 (**We can convert str1 into str2 by replacing 'a' with 'u'.)
- **Input: str1 = "sunday", str2 = "saturday" ; Output: d(v,w)=3**
- **Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using following operations:-**
- **Replace 'n' with 'r', insert t, insert a**

COMMON SUBSEQUENCE

- Given two sequences v = $v_1, v_2, \ldots, v_m$, and w = $w_1, w_2, \ldots, w_n$ a common subsequence of v and w is a sequence of positions in
- v: $1 < i_1 < i_2 < \ldots < i_t <$ m and a sequence of positions in
- w: $1 < j_1 < j_2 < \ldots < j_t <$ n such that the i_t-th letter of v is equal to the j_t-th letter of w.
- Example: v = ATGCCAT, w = TCGGGCTATC. Then take:

  $i_1 = 2, i_2 = 3, i_3 = 6, i_4 = 7$

  $j_1 = 1, j_2 = 3, , j_3 = 8, , j_4 = 9$
- This gives us that the common subsequence is TGAT.

2. Bellman-Ford algorithm :-

- The Bellman-Ford Algorithm determines the shortest route from a particular source vertex to every other weighted digraph vertices. The Bellman-Ford algorithm can handle graphs where some of the edge weights are negative numbers and produce a correct answer, unlike Dijkstra's algorithm, which does not confirm whether it makes the correct answer. However, it is much slower than Dijkstra's algorithm.
- The Bellman-Ford algorithm works by relaxation; that is, it gives approximate distances that better ones continuously replace until a solution is reached. The approximate distances are usually overestimated compared to the distance between the vertices. The replacement values reflect the minimum old value and the length of a newly found path.
- This algorithm terminates upon finding a negative cycle and thus can be applied to cycle-canceling techniques in network flow analysis.

3. Floyd-Warshall algorithm

- The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
- The Floyd-Warshall method uses a technique of dynamic programming to locate the shortest pathways. It determines the shortest route across all pairings of vertices in a graph with weights. Both directed and undirected weighted graphs can use it.
- This program compares each pair of vertices' potential routes through the graph. It gradually optimizes an estimate of the shortest route between two vertices to determine the shortest distance between two

vertices in a chart. With simple modifications to it, one can reconstruct the paths.

- This method for dynamic programming contains two subtypes:

- **Behavior with negative cycles**: Users can use the Floyd-Warshall algorithm to find negative cycles. You can do this by inspecting the diagonal path matrix for a negative number that would indicate the graph contains one negative cycle. In a negative cycle, the sum of the edges is a negative value; thus, there cannot be a shortest path between any pair of vertices. Exponentially huge numbers are generated if a negative cycle occurs during algorithm execution.

- **Time complexity:** The Floyd-Warshall algorithm has three loops, each with constant complexity. As a result, the Floyd-Warshall complexity has a time complexity of O(n3). Wherein n represents the number of network nodes.

4. Chain matrix multiplication

- If a chain of matrices is given, we have to find the minimum number of the correct sequence of matrices to multiply.

- We know that the matrix multiplication is associative, so four matrices ABCD, we can multiply A(BCD), (AB)(CD), (ABC)D, A(BC)D, in these sequences. Like these sequences, our task is to find which ordering is efficient to multiply.

- In the give Matrix chain multiplication problem: Determine the optimal parenthesization of a product of n matrices.

- Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.

- The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same.n input there is an array say arr, which contains arr[] = {1, 2, 3, 4}. It means the matrices are of the order (1 x 2), (2 x 3), (3 x 4). Input: The orders of the input matrices. {1, 2, 3, 4}. It means the matrices are {(1 x 2), (2 x 3), (3 x 4)}. Output: Minimum number of operations need multiply these three matrices. Here the result is 19.

- EXAMPLE: Lets take four matrices A, B, C, and D, we would have:

  - ((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))

  - However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product.

- For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then computing (AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations while computing A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
- Clearly, the first method is more efficient.

- Chain Matrix Multiplication Problem: Given a sequence of matrices A1, A2,...,An and dimensions p0, p1,...,pn where Ai is of dimension pi−1 × pi, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

- **Important Note**: This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

5. Subset Sum

- A subset is a set that contains the elements of a previously defined set. For eg. {a, b} is a subset of {a,b,c,e,f}. In this, you have to find a subset or the set of numbers from the given array that amount to the given value in the input.

- Given a set of N non-negative integers and a target sum S, determine if there exists a subset of the given set such that the sum of the elements of the subset equals to S. Return 1 if there exists such subset, otherwise return 0.

- Therefore a subset sum problem is that given a subset A of n positive integers and a value sum is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

- Let's look at the problem statement: "You are given an array of non-negative numbers and a value 'sum'. You have to find out whether a subset of the given array is present whose sum is equal to the given value."

- Input: {10, 0, 5, 8, 6, 2, 4} →15 Output: True

- Explanation: The sum of the subset {5,8,2} gives the sum as 15. Therefore, the answer comes out to be true.

10	0	5	8	6	2	4

- We create a boolean subset[][] and fill it in bottom up manner.

- subset[i][j] denotes if there is a subset of sum j with element at index i-1 as the last element

subset[i][j] = true if there is a subset with: * the i-th element as the last element * sum equal to j

- Base cases include:

j=0 then subset[i][0] will be true as the sum for empty set is 0.

If i=0, then subset[0][j] will be false, as with no elements, we can get no sum.

subset[i][0] = true as sum of {} = 0 subset[0][j] = false as with no elements we can get no sum

- If element at index i (E1) is greater than j, then subset[i][j] = false as we cannot get a subset of positive numbers with E1 as a member. If we include element at index i (E1), we get

subset[i][j] = subset[i-1][j-E1]; where E1 = array[i-1]

6. 0/1 Knapsack:-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.
- The Knapsack problem states- Which items should be placed into the knapsack such that- The value or profit obtained by putting the items into the knapsack is maximum and the weight limit of the knapsack does not exceed.
- 0/1 Knapsack Problem is one of the variant of Knapsack problem where s the name suggests, items are indivisible here.We can not take the fraction of any item. We have to either take an item completely or leave it completely.It is solved using dynamic programming approach.
- Step-01: Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns. Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3		W
0	0	0	0	0	……	0
1	0					
2	0					
	……					
n	0					

T-Table

- Step-02: Start filling the table row wise top to bottom from left to right.

  Use the following formula- :- T (i , j) = max { T (i-1 , j) , $value_i$ + T(i-1 , j – $weight_i$ +) } . Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j. This step leads to completely filling the table.Then, value of the last box represents the maximum possible value that can be put into the knapsack.

- Step-03: To identify the items that must be put into the knapsack to obtain that maximum profit, consider the last column of the table. Start scanning the entries from bottom to top.On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

- **Time Complexity:-** Each entry of the table requires constant time θ(1) for its computation.It takes θ(nw) time to fill (n+1)(w+1) table entries.It takes θ(n) time for tracing the solution since tracing process traces the n rows. Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

9.6 UNDERSTANDING DYNAMIC PROGRAMMING

- Dynamic Programming is mainly an optimization compared to simple recursion. The main idea is to decompose the original question into repeatable patterns and then store the results as many sub-answers. Therefore, we do not have to re-compute the pre-step answers when needed later. In terms of big O, this optimization method generally reduces time complexities from exponential to polynomial.

- You can imagine we have a tree of a problem and their sub-problems.

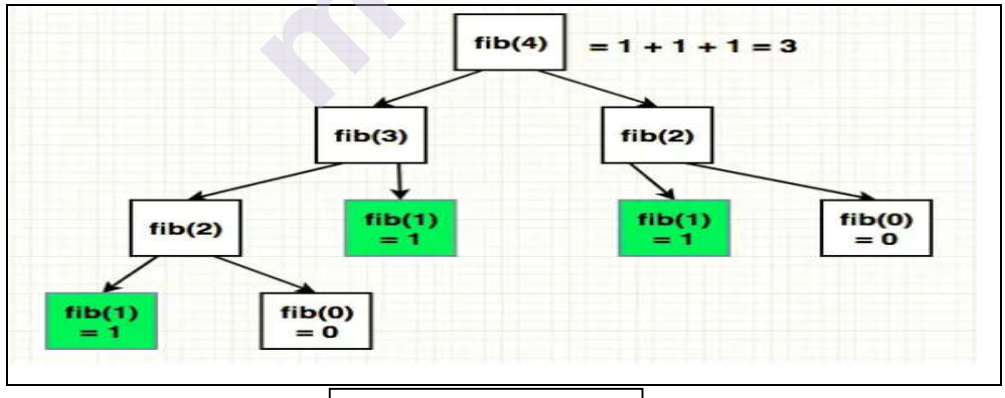
- We start with solving the "leaf" level problems and then move on to their "parent" problems and so on. We save the results as we solve sub-problems for future reference. Thereby avoiding working on the same sub-problem if encountered again.

- Dynamic Programming and Recursion are very similar. Both recursion and dynamic programming are starting with the base case where we initialize the start. he main difference is that, for recursion, we do not store any intermediate values whereas dynamic programming do utilize that.Let's get a bit deeper with the Fibonacci Number.

1. FIBONACCI NUMBER

- The Fibonacci numbers are the numbers in the following integer sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..
- In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ for n>1with seed values $F_0 = 0$ and $F_1 = 1$.
- Analyzing Time complexity of both solutions it was clear that the recursive solution was getting significantly slower for each additional number in the sequence due to exponential time growth. There is, however, a smart way of improving this solution and that is using Memoization.
- For our recursive solution we want to store the arguments of each function call as well as function's output, and reuse it later on if the function was called with the same arguments.Our recursive solution looked like this:

```
function fib(n) {
  if (n < 2) {
    return n;
  }
  return fib(n - 1) + fib(n - 2);
}
```



Source:

- From above visualization you can see the recursive implementation reaches the correct answer through the addition of 1s (and 0s). It's good to understand recursion by visualizing the call stack. In the case of fib(4), function fib is called 9 times. For a computer keeping track of the fib function 9 times is not a huge burden, so this is not a big deal. It will become a big deal as the entered index increases. Incrementing the

index by 1 raised the number of fib calls to 15. You can imagine what problem this will raise once we want to find the Fibonacci number at higher indexes. Its complexity in Big O notation is O(2^n).Now here comes memorization.

- In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- How does Memoization help? Calling fib(5) produces a call tree that calls the function on the same value many times:

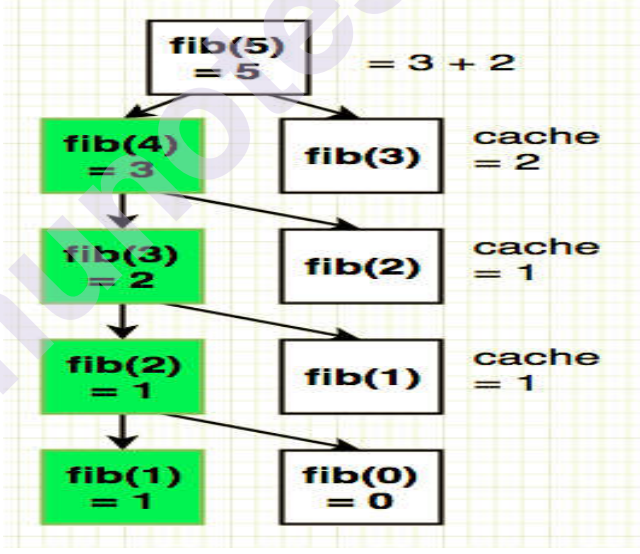
fib(5)

fib(4) + fib(3)

(fib(3) + fib(2)) + (fib(2) + fib(1))

((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))



Source:

- In the above example, fib(2) was calculated three times (overlapping of subproblems). If n is big, then many more values of fib (sub problems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same sub problems again and again we can store the previous calculated values and reduce the complexity. Memoization works like this: Start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is called twice with the same parameters, we simply look up the answer in the table.

- Hence while solving the problems using DP, try to figure out the following:
  - See how the problems are defined in terms of subproblems recursively.
  - See if we can use some table [memoization] to avoid the repeated calculations.

2. Factorial

- Let's create a factorial program using recursive functions. Until the value is not equal to zero, the recursive function will call itself. Factorial can be calculated using the following recursive formula.
  n! = n * (n – 1)!
  n! = 1 if n = 0 or n = 1
- This definition can easily be converted to implementation. Here the problem is finding the value of n!, and the sub-problem is finding the value of (n – l)!. In the recursive case, when n is greater than 1, the function calls itself to find the value of (n – l)! and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1.

```
int fact(int n){
{
  if (n == 1)   return 1;
   else  if (n==0) return 1l
   else  //recursive call
            return (n * fact(n-1));
}
```

- The recurrence implementation can be given as: T(n) = n × T(n – 1) ≈ O(n). Time Complexity: O(n). Space Complexity: O(n), recursive calls need a stack of size.
- In the recursion method as gien above fact(n) is calculated from fact(n -1) and n and nothing else. Instead of calling fact(n) every time, we can store the previous calculated values in a table and use these values to calculate a new value.

```
int[] factorialDP(int n)
{
        int fact[n+1],i,j; // factorials array
        fact[0]=1;

        for(i=1;i<=n;i++)
        {
                fact[i] = i * fact[i-1];
        }

        return fact;
}
```

- The above implementation clearly reduces the complexity as compared to recursive implementation. This is because if the fact(n) is already there, then we are not recalculating the value again. If we fill these newly computed values, then the subsequent calls further reduce the complexity.

9.7 LONGEST COMMON SUBSEQUENCE

- Given a sequence of elements, a subsequence of it can be obtained by removing zero or more elements from the sequence, preserving the relative order of the elements. Note that for a substring, the elements need to be contiguous in a given string, for a subsequence it need not be. Eg: S1="ABCDEFG" is the given string. "ACEG", "CDF" are subsequences, where as "AEC" is not. For a string of lenght n the total number of subsequences is 2n (Each character can be taken or not taken). Now the question is, what is the length of the longest subsequence that is common to the given two Strings S1 and S2. Lets denote length of S1 by N and length of S2 by M.

1. **BruteForce:-** Consider each of the 2N subsequences of S1 and check if its also a subsequence of S2, and take the longest of all such subsequences. Clearly, very time consuming.

2. **Recursion:-** Can we break the problem of finding the LCS of S1[1...N] and S2[1...M] in to smaller subproblems ?

- In dynamic programming, the phrase “largest common subsequence” (LCS) refers to the subsequence that is shared by all of the supplied sequences and is the one that is the longest. It is different from the challenge of finding the longest common substring in that the components of the LCS do not need to occupy consecutive locations within the original sequences to be considered part of that problem.

- The LCS is characterized by an optimal substructure and overlapping subproblem properties. This indicates that the issue may be split into many less complex sub-issues and worked on individually until a solution is found. The solutions to higher-level subproblems are often reused in lower-level subproblems, thus, overlapping subproblems.

- Therefore, when solving an LCS problem, it is more efficient to use a dynamic algorithm than a recursive algorithm. Dynamic programming stores the results of each function call so that it can be used in future calls, thus minimizing the need for redundant calls.

- For instance, consider the sequences (MNOP) and (MONMP). They have five length-2 common subsequences (MN), (MO), (MP), (NP), and (OP); two length-3 common subsequences (MNP) and (MOP); MNP and no longer frequent subsequences (MOP). Consequently, (MNP) and (MOP) are the largest shared subsequences. LCS can be applied in bioinformatics to the process of genome sequencing. Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
   If X[i] = Y[j]
      LCS[i][j] = 1 + LCS[i-1, j-1]
      Point an arrow to LCS[i][j]
   Else
      LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
      Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

- The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

- In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

9.8 SUMMARY

- Recursion which is calling a function within itself. Sub-problems might have to be solved multiple times when a problem is solved using recursion. At the same time, Dynamic programming is a technique where you store the result of the previous calculation to avoid calculating the same once again.

- One might find dynamic programming a bit intimidating initially. But if one understands the basics well, one can master dynamic programming problems. Having a strong programming foundation is key to getting comfortable with such problems. Applications of dynamic programming are common and relevant to everyday challenges, and mastering dynamic programming gives you the superpower to tackle them.

- There are numerous applications of dynamic programming in real life.

- Algorithms that are aimed at solving optimization problems use a dynamic programming approach. Examples of dynamic programming algorithms are string algorithms like the longest common subsequence, longest increasing subsequence, and the longest palindromic substring. Optimizing the order for chain matrix multiplication. The Bellman-Ford algorithm for finding the shortest distance in a graph.Many of which we have seen in brief in the chapter.

9.9 QUESTIONS

1. Write a short note on Dynamic Programming Strategy.
2. Difference Between A Dynamic Programming Algorithm And Recursion.
3. Explain properties of dynamic programming.
4. Bottom-Up Versus Top-Down Programming
5. Describe Longest common subsequence.
6. Explain Fibonacci number using dynmamic programming approach.

9.10 REFERENCES

1. Data Structure and Algorithmic Thinking with Python, Narasimha Karumanchi, CareerMonk Publications, 2016.
2. Introduction to Algorithm, Thomas H Cormen, PHI.
3. https://www.knowledgehut.com/blog/programming/dynamic-programming
4. https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/
5. https://en.wikipedia.org/wiki/Dynamic_programming
6. https://www.geeksforgeeks.org/
7. https://www.thelearningpoint.net/computer-science/dynamic-programming