Design and Analysis of Algorithms

PREPARED BY

Mrs.JEYA SHANTHI A,MCA,M.PHIL

DEPARTMENT OF COMPUTER SCIENCE

SYLLABUS

Objectives: The objective of the course is to teach techniques for effective problem solving in computing. The use of different paradigms of problem solving will be used to illustrate clever and efficient ways to solve a given problem. In each case emphasis will be placed on rigorously proving correctness of the algorithm.

UNIT –I: ALGORITHM AND ANALYSIS Objective: Understanding various algorithm design techniques. Elementary Data Structures: Stack – Queues – Trees – Priority Queue – Graphs – What is an Algorithm? – Algorithm Specification – Performance Analysis: Space Complexity – Time Complexity – Asymptotic Notation – Randomized Algorithms.

UNIT – II: DIVIDE AND CONQUER Objective: This technique is the basis of efficient algorithms for all kinds of problems. General Method – Binary Search – Recurrence Equation for Divide and Conquer – Finding the Maximum and Minimum— Merge Sort – Quick Sort – Performance Measurement – Randomized Sorting Algorithm – Selection Sort – A Worst Case Optimal Algorithm – Implementation of Select2 – Stassen's Matrix Multiplications.

UNIT – III: THE GREEDY METHOD Objective: This is a simple approach which tries to find the best solution at every step. The General Method – Container Loading – Knapsack Problem – Tree Vertex Splitting – Job Sequencing with Deadlines – Minimum Cost Spanning Trees – Prim's Algorithm – Kruskal's Algorithm – An optimal Randomized Algorithm – Optimal Storage on Tapes – Optimal Merge Pattern – Single Source Shortest Paths.

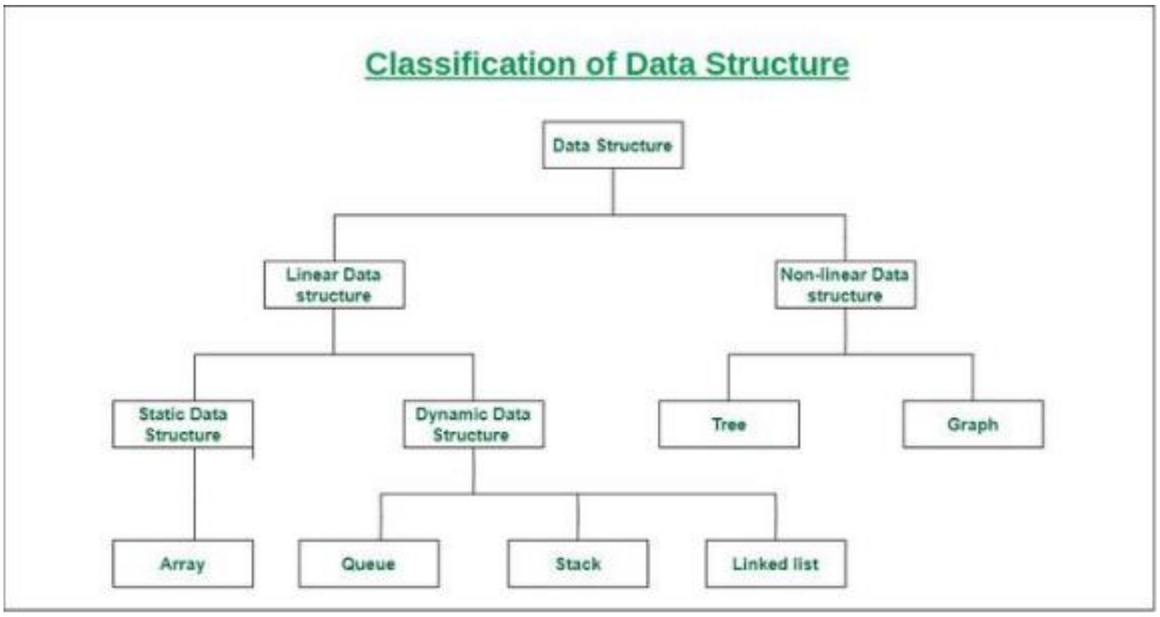
UNIT – IV: DYNAMIC POGRAMMING, TRAVERSAL & SEARCHING Objective: Providing a general insight into the dynamic programming approach. The General Method – Multistage Graphs – All Pair Shortest Path – Optimal Binary Search Trees – String Editing – 0/1 Knapsack – Reliability Design – The Traveling Salesperson Problem. Techniques for Binary Trees – Techniques for Graphs – BFS – DFS.

UNIT – V: BACKTRACKING & BRANCH AND BOUND Objective: Algorithm design paradigm for discrete and combinatorial optimization problems. The General Method – The 8– Queens Problem – Sum of Subsets– Graph Coloring – Hamiltonian Cycles – Branch and Bound: General Method – LC Branch and Bound – FIFO Branch and Bound.

UNIT:I

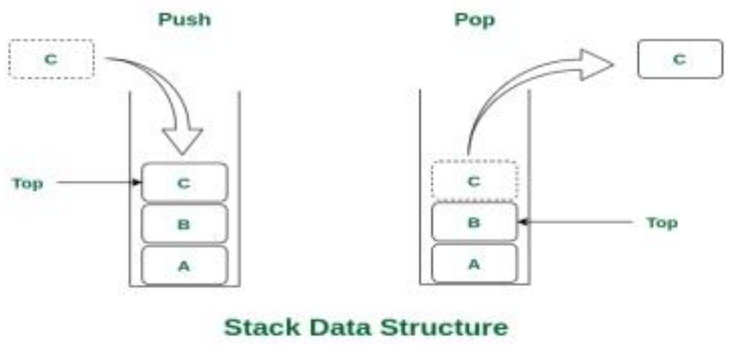
Elementary data structures

Elementary data structures such as stacks, queues, lists, and heaps will be the \of-the-shelf" components we build our algorithm from.

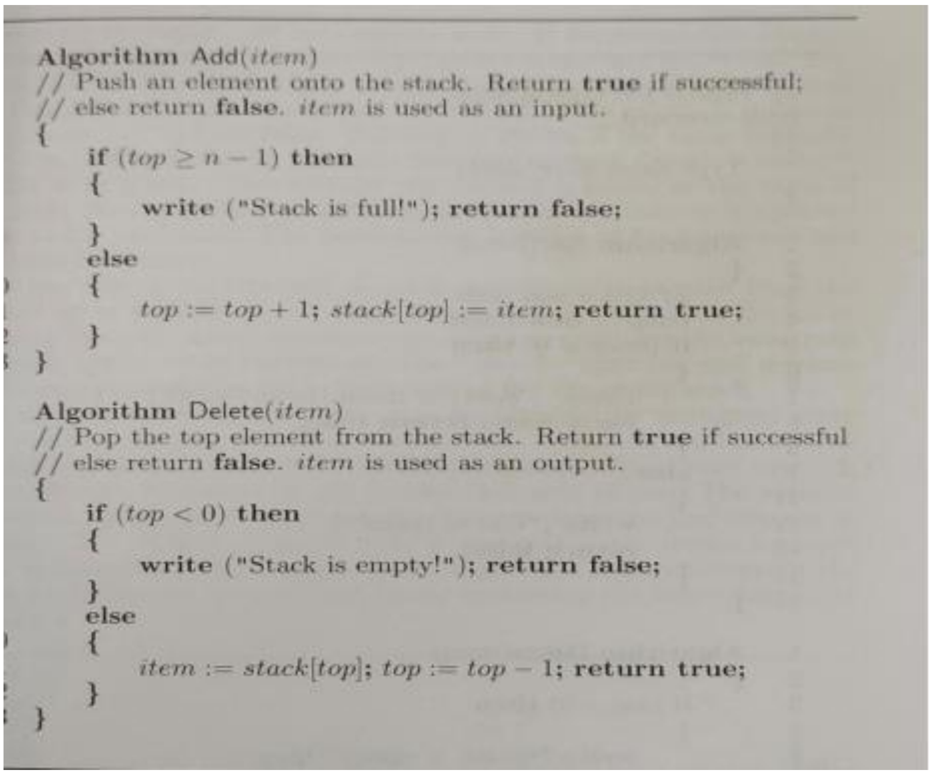


Stack

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last. LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.



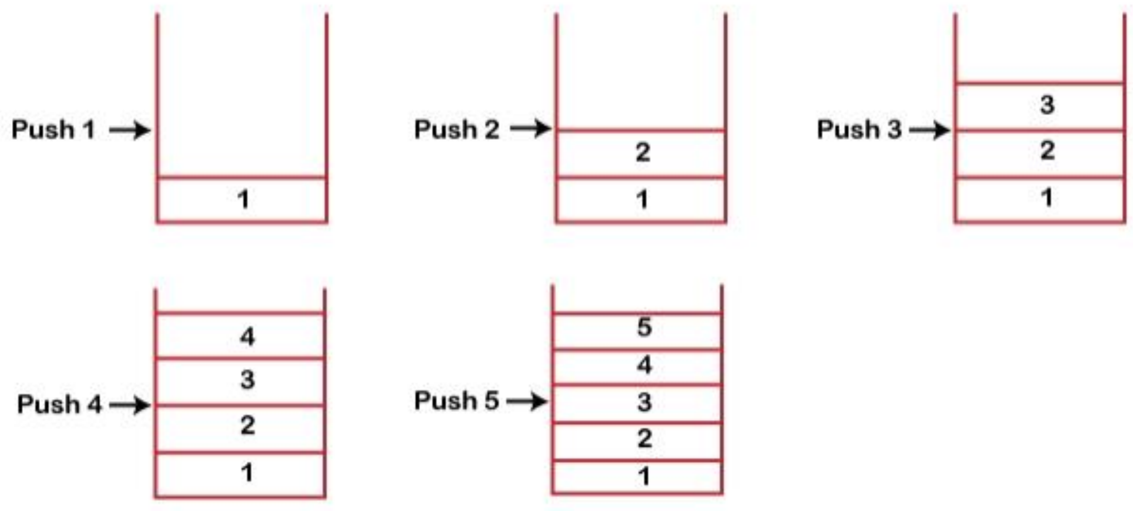
```
Algorithm Add(item)
// Push an element onto the stack. Return true if successful;
// else return false. item is used as an input.
{
    if ($top \geq n - 1$) then
    {
        write ("Stack is full!"); return false;
    }
    else
    {
        $top := top + 1$; $stack[top] := item$; return true;
    }
}

Algorithm Delete(item)
// Pop the top element from the stack. Return true if successful
// else return false. item is used as an output.
{
    if ($top < 0$) then
    {
        write ("Stack is empty!"); return false;
    }
    else
    {
        $item := stack[top]$; $top := top - 1$; return true;
    }
}
```

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Algorithm

1 − Checks if the stack is full.

2 − If the stack is full, produces an error and exit.

3 − If the stack is not full, increments top to point next empty space.

4 − Adds data element to the stack location, where top is pointing.

5 − Returns success.

Standard Stack Operations

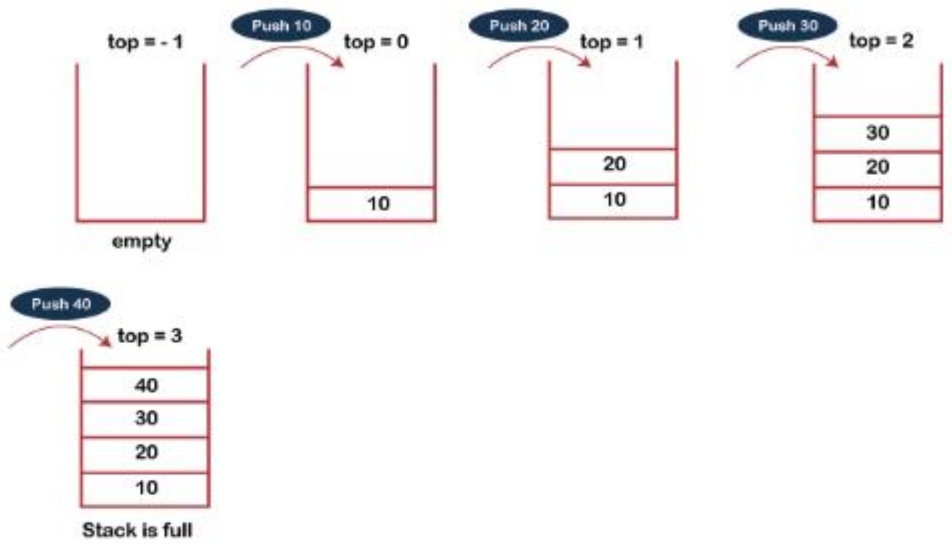
The following are some common operations implemented on the stack:

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the *overflow* condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the *max* size of the stack.



Insertion: push()

push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

Algorithm

1 − Checks if the stack is full.

2 − If the stack is full, produces an error and exit.

3 − If the stack is not full, increments top to point next empty space.

4 − Adds data element to the stack location, where top is pointing.

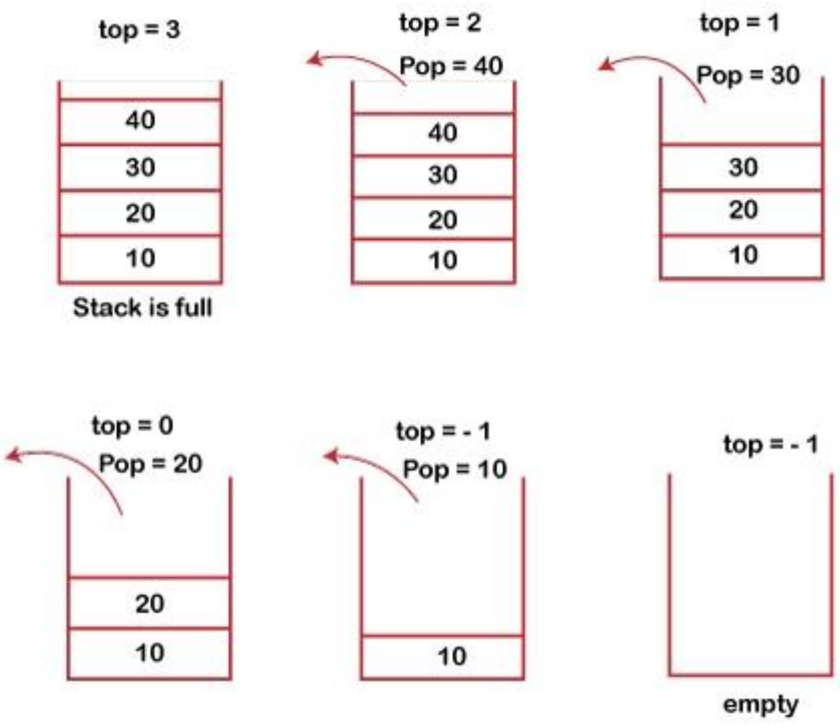
5 − Returns success.

POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the *underflow* condition occurs.
- If the stack is not empty, we first access the element which is pointed by the *top*
- Once the pop operation is performed, the top is decremented by 1,

- i.e., top=top-1.



-

Deletion: pop()

pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

Algorithm

1 − Checks if the stack is empty.

2 − If the stack is empty, produces an error and exit.

3 − If the stack is not empty, accesses the data element at which top is pointing.

4 − Decreases the value of top by 1.

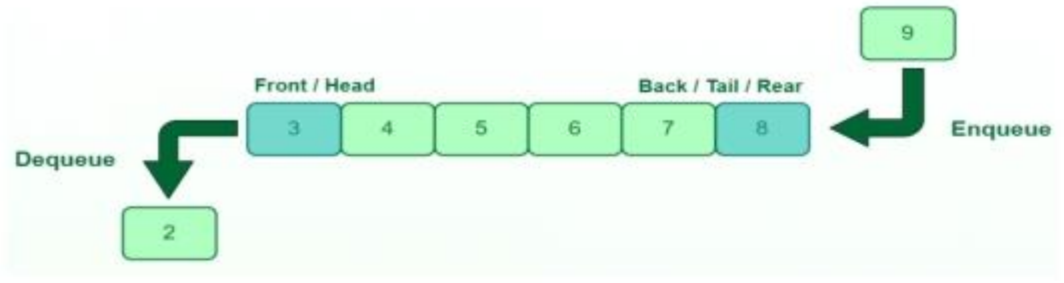
5 − Returns success.

QUEUE

What is Queue Data Structure

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

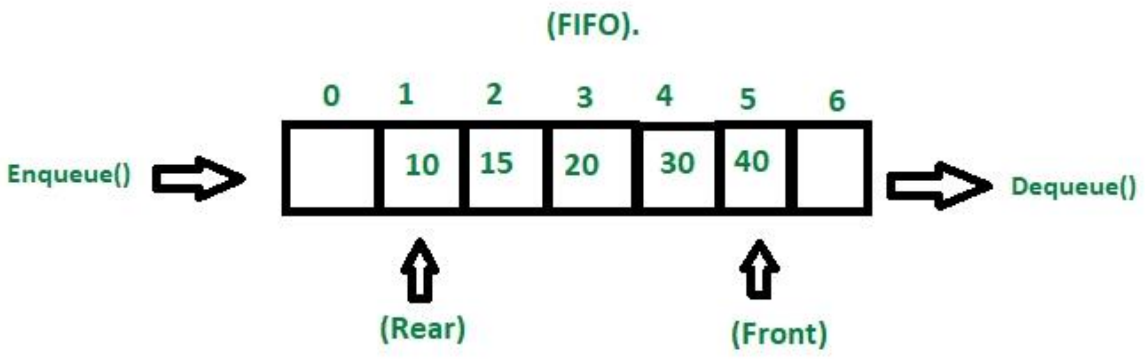
We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



Queue Data Structure

FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue(sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue. See the below figure.



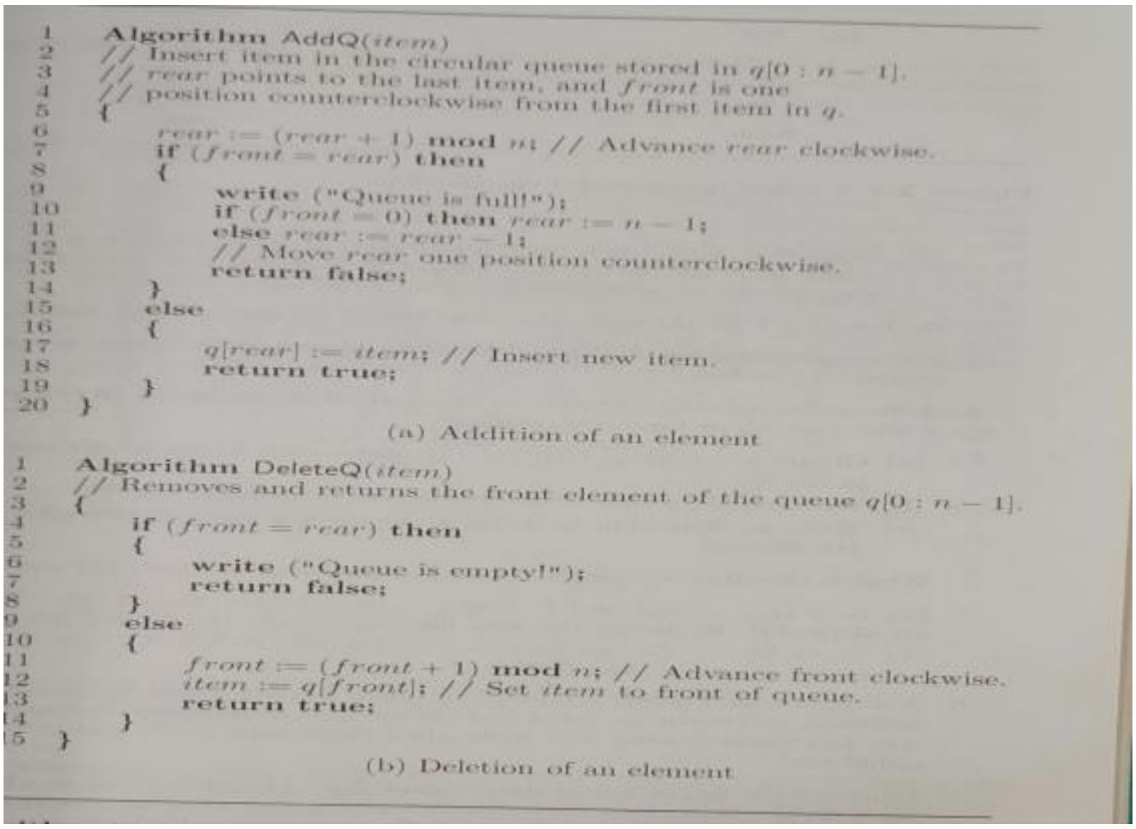
Characteristics of Queue:

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

Queue Representation:

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

- Queue: the name of the array storing queue elements.
- Front: the index where the first element is stored in the array representing the queue.
- Rear: the index where the last element is stored in an array representing the queue.



```
Algorithm AddQ(item)
// Insert item in the circular queue stored in q[0 : n - 1].
// rear points to the last item, and front is one
// position counterclockwise from the first item in q.
{
    rear := (rear + 1) mod n; // Advance rear clockwise.
    if (front = rear) then
    {
        write ("Queue is full!");
        if (front = 0) then rear := n - 1;
        else rear := rear - 1;
        // Move rear one position counterclockwise.
        return false;
    }
    else
    {
        q[rear] := item; // Insert new item.
        return true;
    }
}
```

(a) Addition of an element

```
Algorithm DeleteQ(item)
// Removes and returns the front element of the queue q[0 : n - 1].
{
    if (front = rear) then
    {
        write ("Queue is empty!");
        return false;
    }
    else
    {
        front := (front + 1) mod n; // Advance front clockwise.
        item := q[front]; // Set item to front of queue.
        return true;
    }
}
```

(b) Deletion of an element

Basic Operations

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers − front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Insertion operation: enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the stack. The following algorithm describes the enqueue() operation in a simpler way.

Algorithm

```
1 − START

2 – Check if the queue is full.

3 − If the queue is full, produce overflow error and exit.

4 − If the queue is not full, increment rear pointer to point the next empty space.

5 − Add data element to the queue location, where the rear is pointing.

6 − return success.

7 – END
```

Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

```
1 – START

2 − Check if the queue is empty.

3 − If the queue is empty, produce underflow error and exit.

4 − If the queue is not empty, access the data where front is pointing.

5 − Increment front pointer to point to the next available data element.
```

6 − Return success.

7 – END

The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm

1 – START

2 – Return the element at the front of the queue

3 – END

Full() Operation

The isFull() operation verifies whether the stack is full.

Algorithm

1 – START

2 – If the count of queue elements equals the queue size, return true

3 – Otherwise, return false

4 – END

Queue is full!

The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1 – START

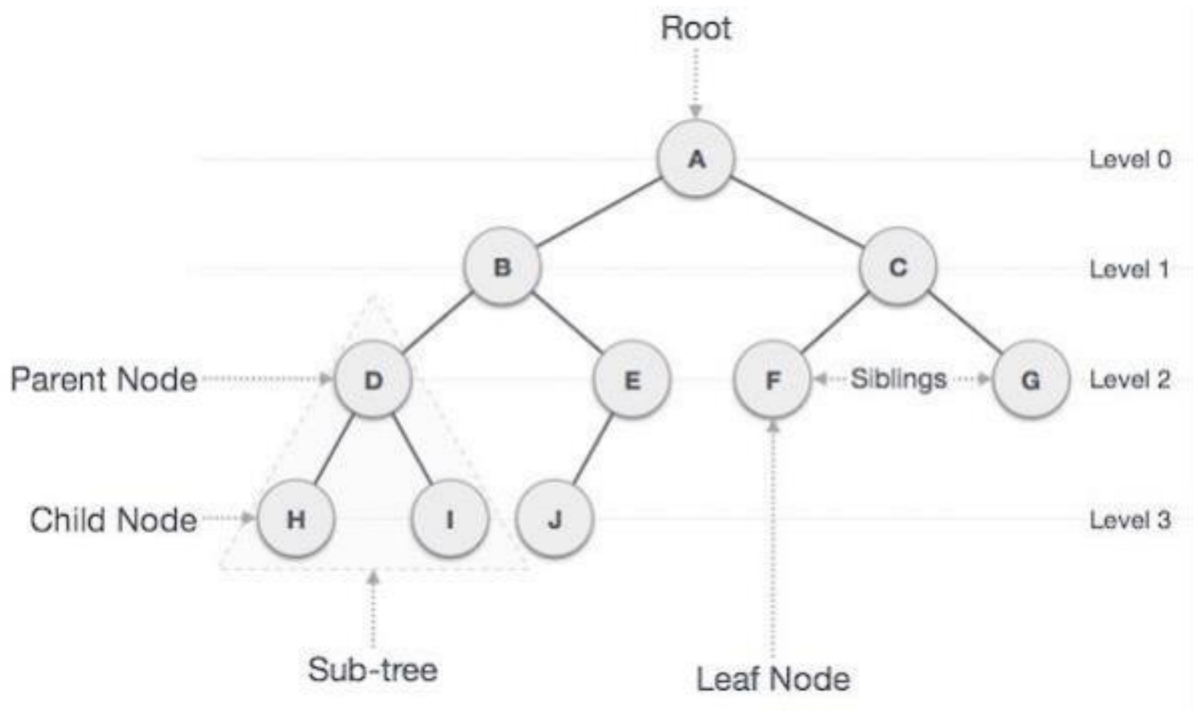
2 – If the count of queue elements equals zero, return true

3 – Otherwise, return false

4 – END

Tree

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 from 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.

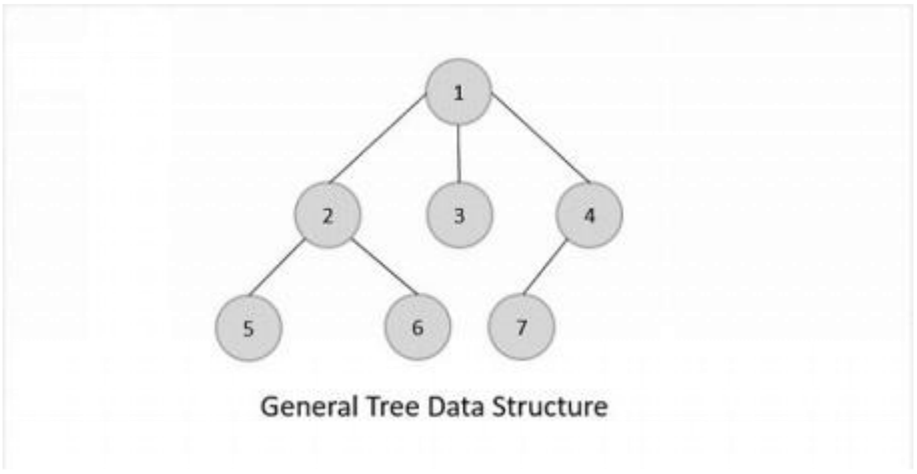
The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.

Advantages of Tree Data Structure:

- Tree offer Efficient Searching Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it easy to organize and navigate large amounts of information.
- The recursive nature of trees makes them easy to traverse and manipulate using recursive algorithms.

Disadvantages of Tree Data Structure:

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.



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

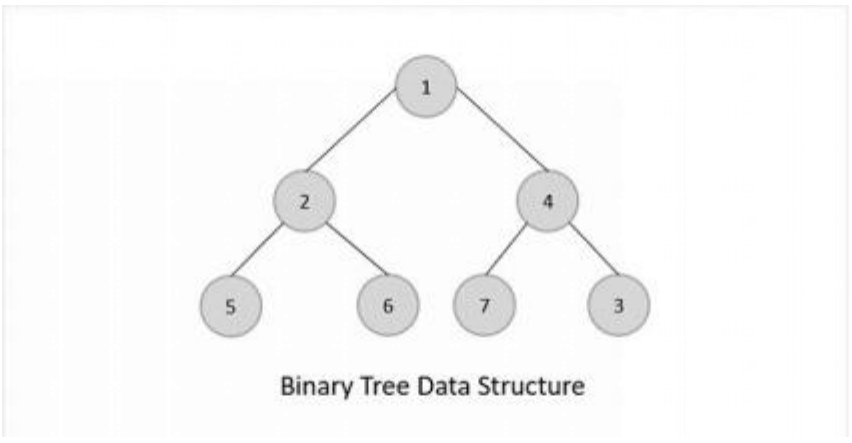
- A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

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.

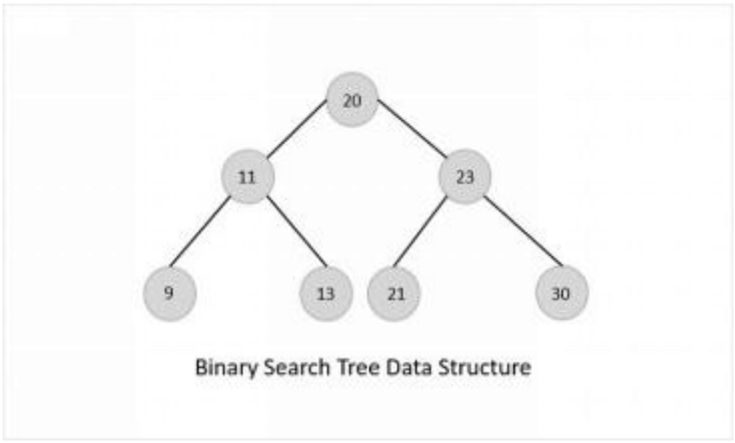


Binary Tree Data Structure

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.



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.

priority queue

A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

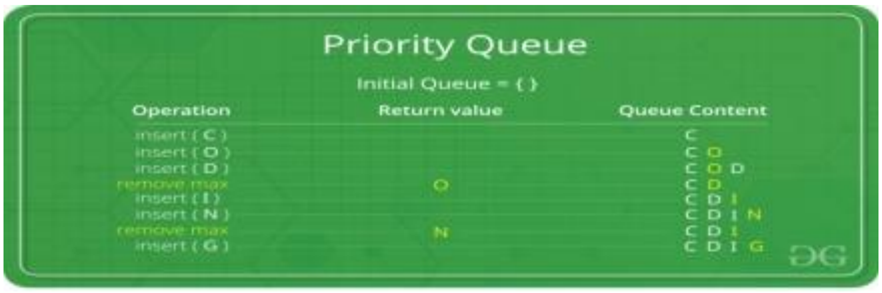
In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back.

Properties of Priority Queue

So, a priority Queue is an extension of the queue with the following properties.

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

In the below priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.



Priority Queue

Initial Queue = { }

Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

Priority assigned to the elements in a Priority Queue

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

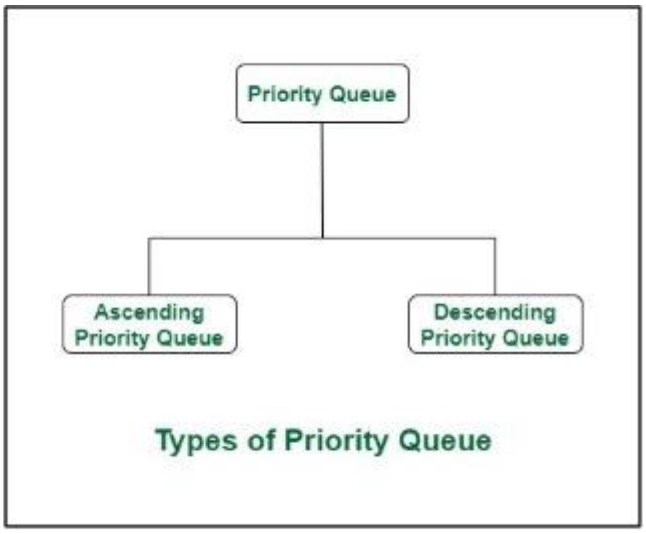
Types of Priority Queue:

1) Ascending Order Priority Queue

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 4 will remove from the queue and dequeue returns 4.

2) Descending order Priority Queue

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.



Difference between Priority Queue and Normal Queue

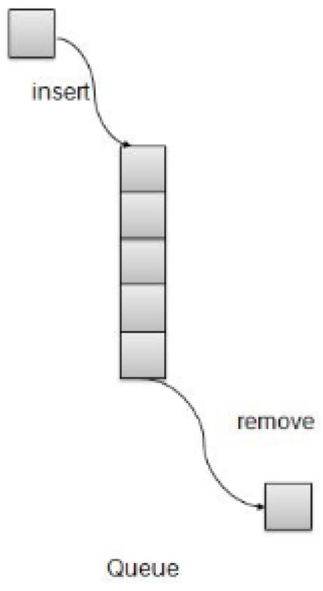
There is no priority attached to elements in a queue, the rule of first-in-first-out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- insert / enqueue − add an item to the rear of the queue.
- remove / dequeue − remove an item from the front of the queue.

Priority Queue Representation

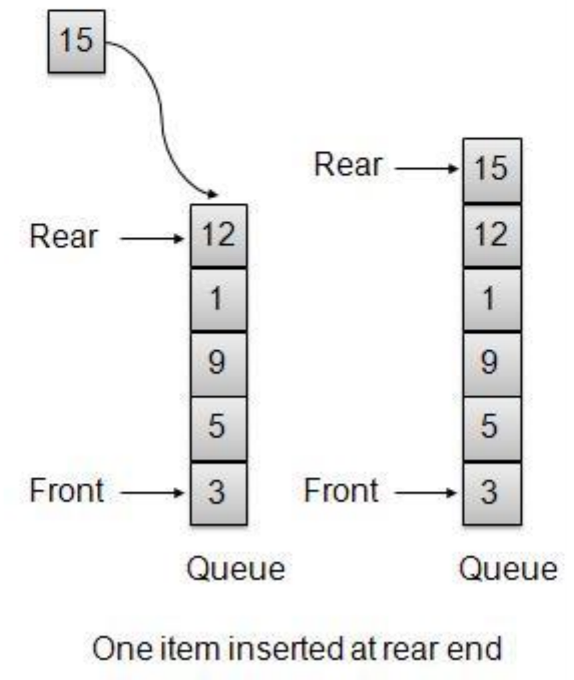


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- Peek − get the element at front of the queue.
- isFull − check if queue is full.
- isEmpty − check if queue is empty.

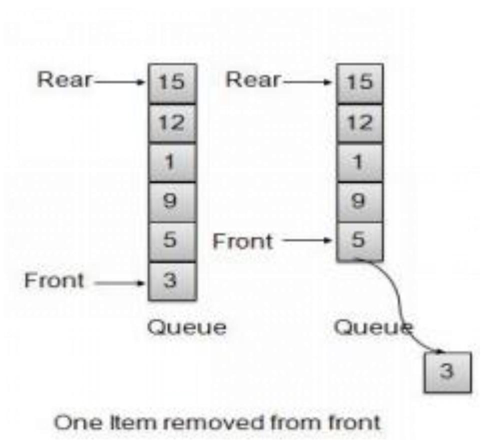
Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



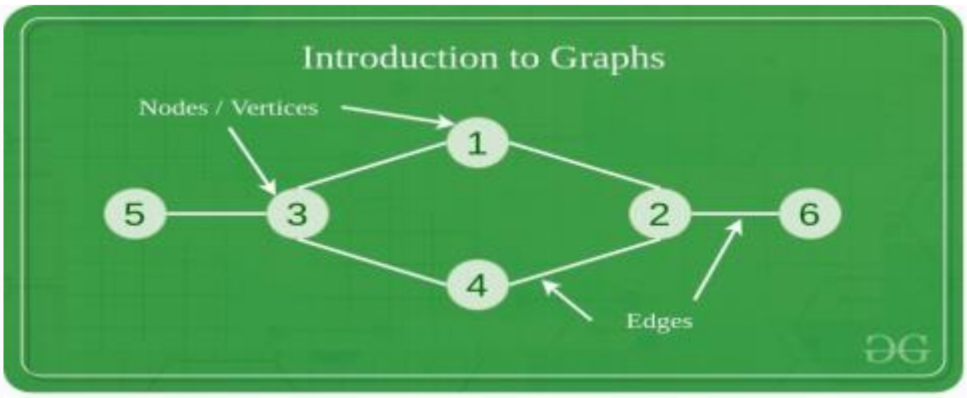
Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



<u>What is Graph Data Structure</u>

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

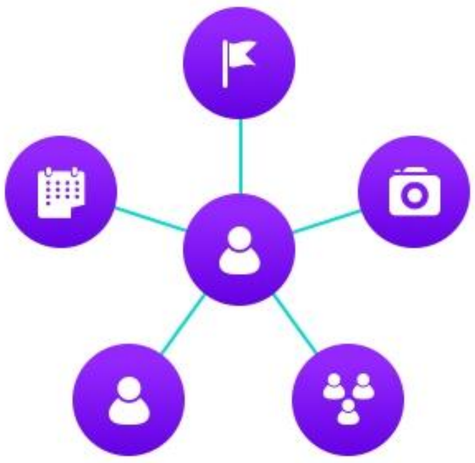


Graph Data Stucture

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.



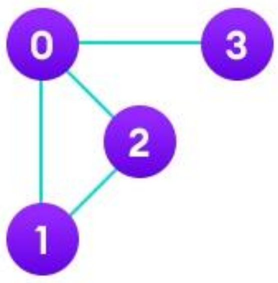
Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V

- A collection of edges E, represented as ordered pairs of vertices (u,v)



Vertices and edges

In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Graph Terminology

- Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- Directed Graph: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

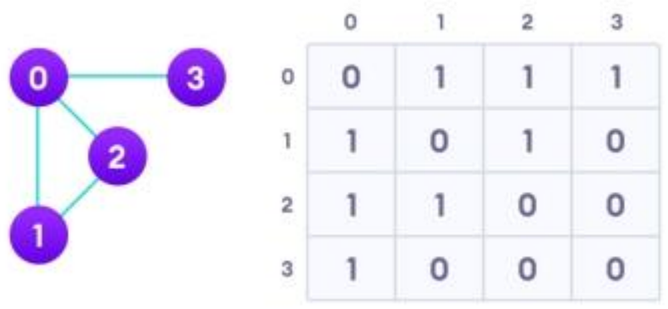
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

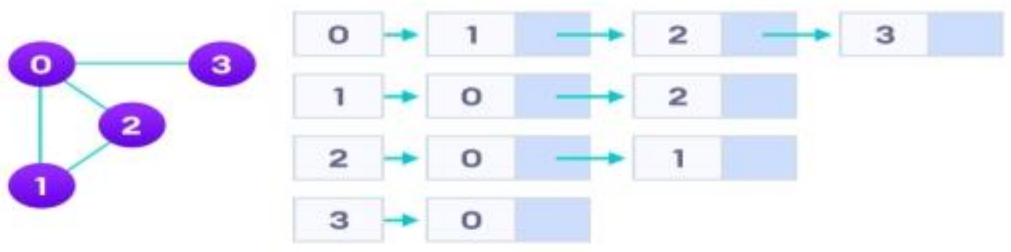
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

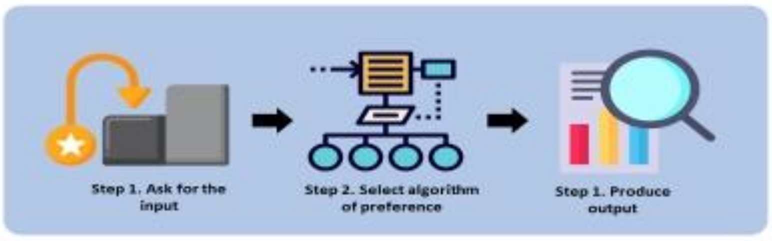
An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Data Structures - Algorithms Basics

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

Search − Algorithm to search an item in a data structure.
Sort − Algorithm to sort items in a certain order.
Insert − Algorithm to insert item in a data structure.
Update − Algorithm to update an existing item in a data structure.
Delete − Algorithm to delete an existing item from a data structure.


Step 1. Ask for the input
Step 2. Select algorithm of preference
Step 1. Produce output

•

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

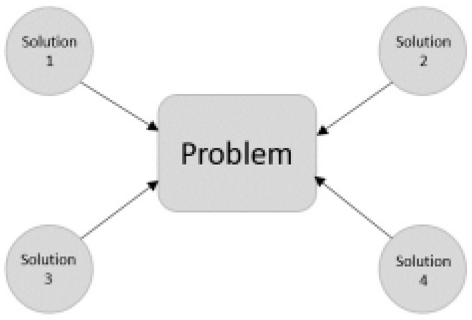
Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
Input − An algorithm should have 0 or more well-defined inputs.
Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
Finiteness − Algorithms must terminate after a finite number of steps.
Feasibility − Should be feasible with the available resources.
Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm vs Program: Difference Between Program and Algorithm

Algorithm	Program
An algorithm is a list of steps to solve a given problem.	A program is software code that eventually translates to machine code that the computer can understand and execute to solve a given problem.

An algorithm is written using plain natural language English phrases.	The software program is written using the programming language statements. It includes expressions and statements that adhere to the programming language syntax. For example: • C • C++ • C# • Java, • Kotlin • Python, etc.
Algorithms are easy to write and understand.	Software program code is understood by the developers/ programmers who know the programming language. Programmers only write programs.
An algorithm is a generalized solution to a problem that the computer can solve.	A software program is a specialized solution to a problem that the computer can solve.
Algorithms are written in informal language.	Software programs are written only in programming languages.
No rules are to be followed.	Programming language syntax rules must be followed while writing programs.

Performance analysis

Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.it is must be a efficiant program to exscute less time to compleate the program When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

Performance analysis have two types of complexity

(I) Space complexity

(II)Time complexity

<u>What Is Space Complexity?</u>

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

It has three types

I) Instruction space complexity

II) Data space complexity

III) Environment space complexity

<u>Instruction Space</u> is used to save compiled instruction in the memory. <u>Environmental Stack</u> is used to storing the addresses while a module calls another module or functions during execution.

<u>Data space</u> is used to store data, variables, and constants which are stored by the program and it is updated during execution.

Calculation types of space complexity

I) constant complexity

II) Linear space complexity

constant complexity

An algorithm has constant time complexity if it takes the same time regardless of the number of inputs. (Reading time: under 1 minute) If an algorithm's time complexity is constant, it means that it will always run in the same amount of time, no matter the input size.

Ex :

```
Int a

{

Int a*a

}
```

Linear space complexity

As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be Linear Space Complexity. If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

Ex:for loops,arrays and values

What Is Time Complexity?

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Constant time – O (1)

An algorithm is said to have constant time with order O (1) when it is not dependent on the input size n. Irrespective of the input size n, the runtime will always be the same.

Linear time – O(n)

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, with this order O(n).

Logarithmic time – O (log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size.

Quadratic time – O (n^2)

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)*O(n) = O(n^2) order.

Similarly, if there are 'm' loops defined in the function, then the order is given by O (n ^ m), which are called polynomial time complexity functions.

What Are Asymptotic Notations?

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows

What Are Asymptotic Notations

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slowAsymptotic notations are classified into five types:

1. Big-Oh (O) notation

2. Big Omega (Ω) notation

3. Big Theta (Θ) notation

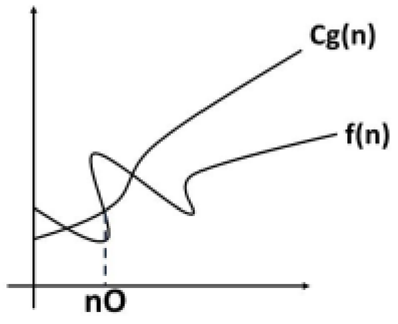
4. ο − Little Oh Notation
5. ω − Little Omega Notation

1. Big-Oh (O) Notation

Paul Bachmann invented the big-O notation in 1894. He inadvertently introduced this notation in his discussion of function approximation.

'n' denotes the upper bound value. If a function is O(n), it is also O(n2) and O(n3).

It is the most widely used notation for Asymptotic analysis. It specifies the upper bound of a function, i.e., the maximum time required by an algorithm or the worst-case time complexity. In other words, it returns the highest possible output value (big-O) for a given input.



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

The two different function of f(n) and g(n) but f(n) grows with same rate slower than g(n)

F(n) ≤cg(n)

n≥no

C>0,no≥1

F(n)=Og(n)

Logically broken down

F(n)=Ocg(n)

g(n)is an asymtotic upper bound

Example

f(n)=3n+2

G(n)=Og(n)

f(n)≤cg(n),c>0,n0≥1

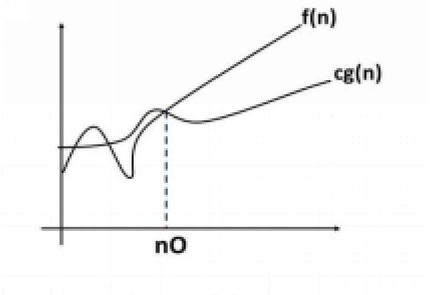
3n+2≤4n

Assume c=4,n≥2

. Big-Omega (Ω) notation

Big-Omega is an Asymptotic Notation for the best case or a floor growth rate for a given function. It gives you an asymptotic lower bound on the growth rate of an algorithm's runtime.

From the definition: The function f(n) is Ω (g(n)) if there exists a positive number c and N, such that f(n) >= cg(n) for all n >= N.



f(n) = Ω(g(n))

f(n) =Ωg(n)

n≥no

C>0,no≥1

F(n)≥cg(n)

Denoted by

F(n)=Ωg(n)

g(n)is an asymtotic lower bound

Example

f(n)=3n+2

g(n)=n

f(n)≥cg(n),c>0,n0≥1

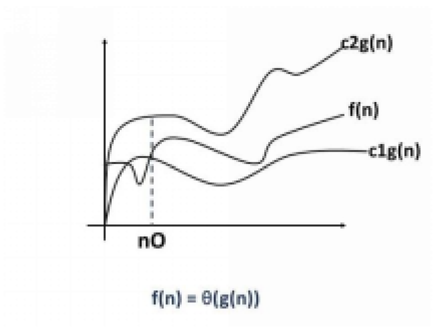
3n+2≥cn

3n+2≥n,no≥1

3n+2=Ωn

3. Big-Theta (Θ) notation

Big theta defines a function's lower and upper bounds, i.e., it exists as both, most, and least boundaries for a given input value.



n∞ f(n) =c

G(n)

F(n) ≤Θcg(n)

C1,c2>0

n≥no

Example

f(n)=3n+2.G(n)=n

C1g(n) ≤f(n)≤c2g(n)

f(n)≤c2g(n),

f(n)≥c1g(n)

3n+2≤4(n)

n0≥1 condition satisfied

Little o asymptotic notation

Slower growth rate

Big-O is used as a tight upper bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper bound. “Little-o” (o()) notation is used to describe an upper bound that cannot be tight.

Definition: Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for any real constant $c > 0$, there exists an integer constant n0 ≥ 1 such that $0 \leq f(n) < c*g(n)$. little o and little omega notations

Thus, little o() means loose upper-bound of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

In mathematical relation, f(n) = o(g(n)) means lim f(n)/g(n) = 0 n→∞

f(n) grows slower than g(n)

F(n)/g(n)=o

F(n)=og(n)

Little ω asymptotic notation

Faster growth rate

F(n)grows faster than g(n)

F(n)/g(n)=∞

n∞ condition has to be satisfied formally stated as

F(n)= ω g(n)

Best Case, Worst Case, and Average Case in Asymptotic Analysis

Best Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time. In this case, the execution time serves as a lower bound on the algorithm's time complexity.

Average Case: You add the running times for each possible input combination and take the average in the average case. Here, the execution time serves as both a lower and upper bound on the algorithm's time complexity.

Worst Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible. In this case, the execution time serves as an upper bound on the algorithm's time complexity.

You will now see how to calculate space and time complexity after grasping the significance of space and time complexity.

Randomized algorithms

Randomized algorithms use random numbers or choices to decide their next step. We use these algorithms to reduce space and time complexity.

- There are two types of randomized algorithms:
- 1)Las vegas 2)monte carlo
- A Las Vegas algorithm is a randomized algorithm that always gives the correct result but gambles with resources. Las Vegas Algorithms (LV)Always give the correct answer. But slow (Comparatively) and need resources.

e.g: randomized quicksort, randomized selection

Las Vegas algorithms always return correct results or fail to give one; however, its runtime may vary. An upper bound can be defined for its runtime.

Relation with the Monte-Carlo Algorithms:

The Las-Vegas algorithm can be differentiated with the Monte-carlo algorithms in which the resources used to find out the solution are bounded but it does not give guarantee that the solution obtained is accurate.

In some applications by making early termination a Las-Vegas algorithm can be converted into Monte-Carlo algorithm.

Complexity Analysis:

The complexity class of given problem which is solved by using a Las-Vegas algorithms is expect that the given problem is solved with zero error probability and in polynomial time.

This zero error probability polynomial time is also called as ZPP which is obtained as follows,

Randomized polynomial time algorithm always provide correct output when the correct output is no, but with a certain probability bounded away from one when the answer is yes. These kinds of decision problem can be included in class RP i.e. randomized where polynomial time.

That is how we can solve given problem in expected polynomial time by using Las-Vegas algorithm. Generally there is no upper bound for Las-vegas algorithm related to worst case run time.

```
Algorithm search repeat[A]

{

For I=0;I<n:I++

{

If A[I]=A[j]

Return true

}

}
```

<u>Algorithm lv</u>

```
Algorithm lv-search repeat [A]

While true

{

I=random mod n+1

If(Inot=j
A[I]=A[j]

}

}

Return true
```

Monte Carlo simulations are a broad class of algorithms that use repeated random sampling to obtain numerical results.

- Monte Carlo simulations are typically used to simulate the behaviour of other systems.
- Monte Carlo algorithms, on the other hand, are randomized algorithms whose output may be incorrect with a certain, typically small, probability.

Not always 100% correct. But fast.e.g: Karger’s algorithm (Min cut)

A randomized algorithm that always produce correct result with only variation from one aun to another being its running time is known as Las-Vegas algorithm.

Monte-Carlo algorithms

The Monte-Carlo algorithms work in a fixed running time; however, it does not guarantee correct results. One way to control its runtime is by limiting the number of iterations.

Applications and Scope:

The Monte-carlo methods has wider range of applications. It uses in various areas like physical science, Design and visuals, Finance and business, Telecommunication etc. In general Monte carlo methods are used in mathematics. By generating random numbers we can solve the various problem. The problems which are complex in nature or difficult to solve are solved by using Monte-carlo algorithms. Monte carlo integration is the most common application of Monte-carlo algorithm.

The deterministic algorithm provides a correct solution but it takes long time or its runtime is large. This run-time can be improved by using the Monte carlo integration algorithms. There are various methods used for integration by using Monte-carlo methods such as,

i) Direct sampling methods which includes the stratified sampling, recursive stratified sampling, importance sampling.

ii) Random walk Monte-carlo algorithm which is used to find out the integration for given problem.
Consider a tool that basically does sorting. Let the tool be used by many users and there are few users who always use tool for already sorted array. If the tool uses simple (not randomized) QuickSort, then those few users are always going to face worst case situation. On the other hand if the tool uses Randomized QuickSort, then there is no user that always gets worst case. Everybody gets expected O(n Log n) time.

```
Algorithm search repeat(A,a)

{

For(I=0;I<n-1;I++)

If (A(i)=a)

Return true

}
```

<u>Mc -Algorithm</u>

```
Algorithm Mc search(A,a,x)

{

I=0,flag=false

While(I<=x)

{

I =random()mod n+1
i=I+1
If(element is found)
}
Flag=true
Return flag
}
```

uses

Randomized search algorithms: These are algorithms that use randomness to search for solutions to problems. Examples include genetic algorithms and simulated annealing.

Randomized data structures: These are data structures that use randomness to improve their performance. Examples include skip lists and hash tables.

Randomized load balancing: These are algorithms used to distribute load across a network of computers, using randomness to avoid overloading any one computer.

Randomized encryption: These are algorithms used to encrypt and decrypt data, using randomness to make it difficult for an attacker to decrypt the data without the correct key.

UNIT-II

Divide and Conquer Algorithm

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. Learn about recursion in different programming languages:

How Divide and Conquer Algorithms Work

Here are the steps involved:

Divide: Divide the given problem into sub-problems using recursion.

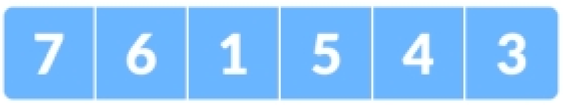
Conquer: Solve the smaller sub-problems recursively. If the sub problem is small enough, then solve it directly.

Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

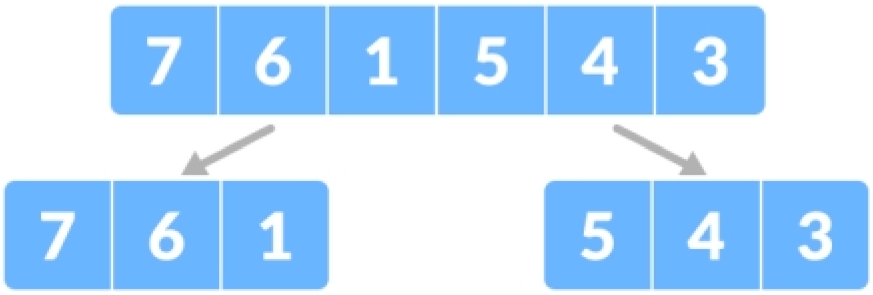
Let us understand this concept with the help of an example.

Here, we will sort an array using the divide and conquer approach (ie. merge sort).

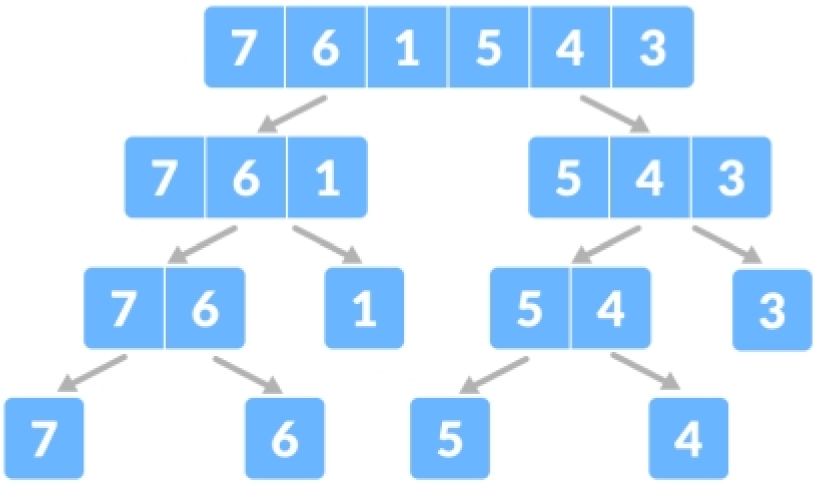
Let the given array be:



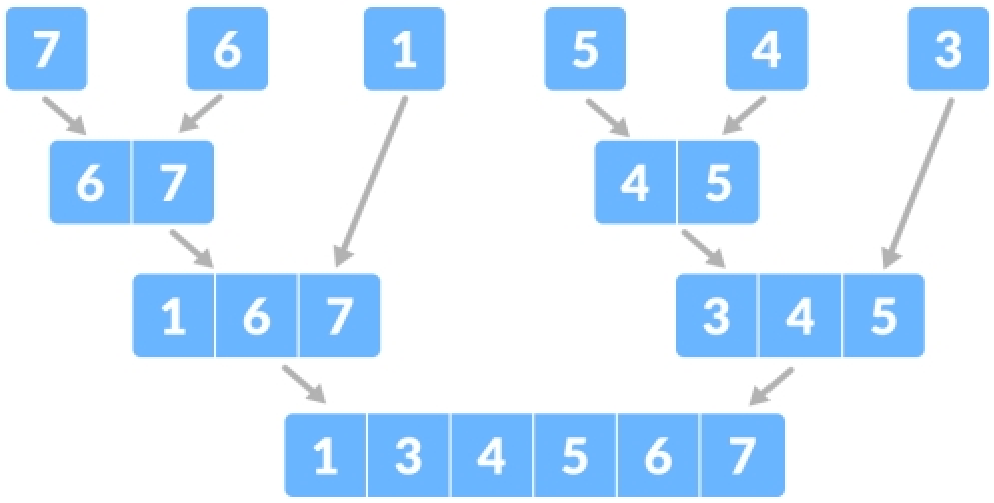
Divide the array into two halves



Again, divide each subpart recursively into two halves until you get individual elements.



Now, combine the individual elements in a sorted manner. Here, conquer and combine steps go side by side.



Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

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

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Let us take an example to find the time complexity of a recursive problem.

For a merge sort, the equation can be written as:

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

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

Where,

a = 2 (each time, a problem is divided into 2 subproblems)

$n/b = n/2$ (size of each sub problem is half of the input)

$f(n)$ = time taken to divide the problem and merging the subproblems

$T(n/2) = O(n \log n)$ (To understand this, please refer to the master theorem.)

Now, $T(n) = 2T(n \log n) + O(n)$

$\approx O(n \log n)$

Advantages of Divide and Conquer Algorithm

- The complexity for the multiplication of two matrices using the naive method is O(n3), whereas using the divide and conquer approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

Divide and Conquer Applications

- Binary Search
- Merge Sort
- Quick Sort

Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Conditions for when to apply Binary Search in a Data Structure:

To apply Binary Search algorithm:

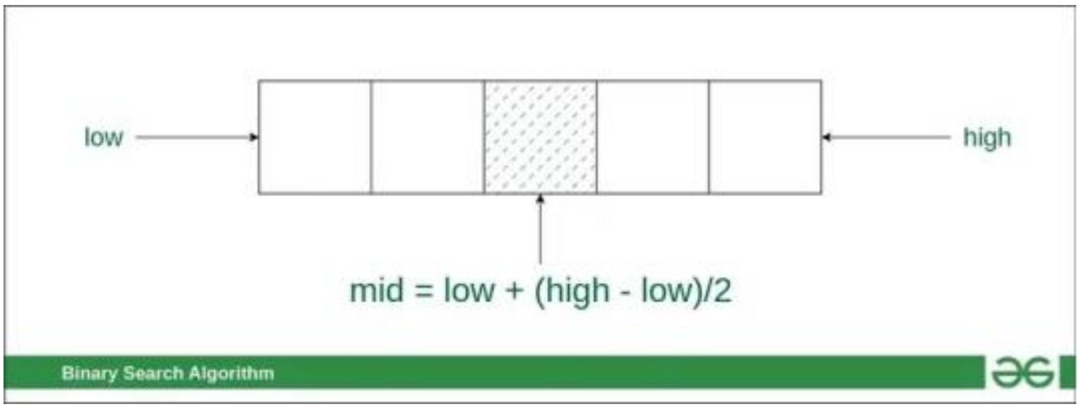
The data structure must be sorted.

Access to any element of the data structure takes constant time.

Binary Search Algorithm:

In this algorithm,

Divide the search space into two halves by finding the middle index "mid".



Compare the middle element of the search space with the key.

(I)If the key is found at middle element, the process is terminated.

(II)If the key is not found at middle element, choose which half will be used as the next search space.

(III)If the key is smaller than the middle element, then the left side is used for next search.

(IV)If the key is larger than the middle element, then the right side is used for next search.

This process is continued until the key is found or the total search space is exhausted.

Advantages of Binary Search:

Binary search is faster than linear search, especially for large arrays.

More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.

Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

The array should be sorted.

Binary search requires that the data structure being searched be stored in contiguous memory locations.

Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Applications of Binary Search:

Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.

It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.

It can be used for searching a database.

Max-Min Problem

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.

In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required. This approach is simple but it does (n – 1) comparisons for finding max and the same number of comparisons for finding the min. It results in a total of 2(n – 1) comparisons. Using a divide and conquer approach, we can reduce the number of comparisons.

Divide and conquer approach for Max. Min problem works in three stages.

1.If a1 is the only element in the array, a1 is the maximum and minimum.

2.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.

3.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.

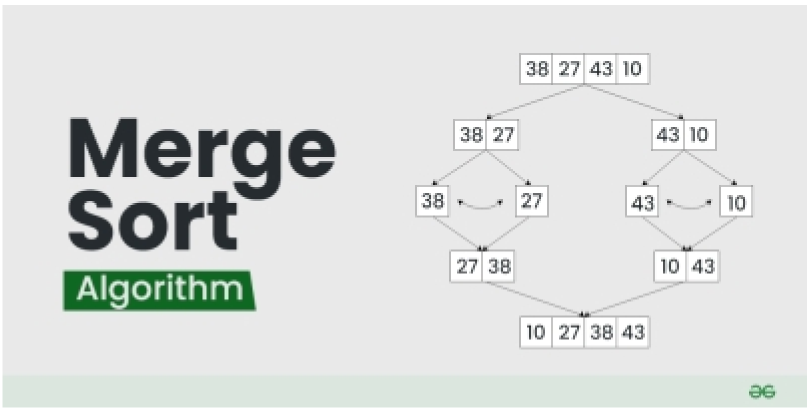
This algorithm makes the given problem easier as it divides the given problem into sub-problems which makes it easier to solve and then solving each of them individually and combining all the solution of all sub-problem into one to solve the original problem.

Disadvantages of Divide and Conquer Algorithm

1. Most of the divide and conquer design uses the concept of recursion therefore it requires high memory management. 2. Memory overuse is possible by an explicit stack.
Merge sort

is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



How does Merge Sort work

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Complexity Analysis of Merge Sort:

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).

External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.

Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Advantages of Merge Sort:

Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.

Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.

Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

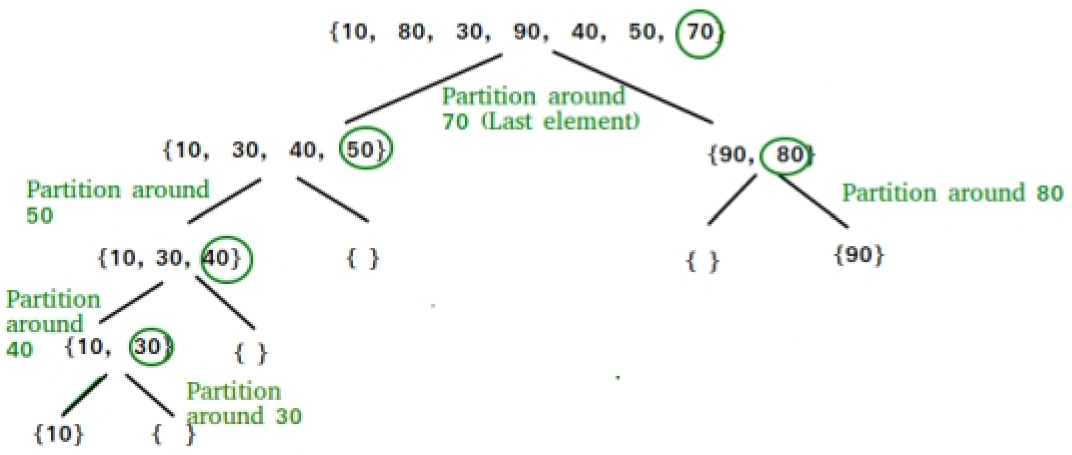
Quick Sort

is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



Time Complexity:

Best Case: Ω (N log (N))

The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array

into roughly equal halves.
In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

Average Case: θ (N log (N))
Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

Worst Case: O(N2)
The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Auxiliary Space: O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

Advantages of Quick Sort:

It is a divide-and-conquer algorithm that makes it easier to solve problems.

It is efficient on large data sets.

It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.

It is not a good choice for small data sets.

It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

What is performance measurements of algorithm

Time and space complexity are the two main measures for calculating algorithm efficiency, determining how many resources are needed on a machine to process it. Where time measures how long it takes to process the algorithm, space measures how much memory is used.
Time Complexity and Space Complexity

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

There are two such methods used, time complexity and space complexity which are discussed below:

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

Definition–The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called time complexity of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Space Complexity:

Definition –

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

It is the amount of memory needed for the completion of an algorithm.

To estimate the memory requirement we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

\

What is a Randomized Algorithm

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot

Selection sort

is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is O(N2) as there are two nested loops:

One loop to select an element of Array one by one = O(N)

Another loop to compare that element with every other Array element = O(N)

Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

Simple and easy to understand.

Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

Selection sort has a time complexity of O(n^2) in the worst and average case.

Does not work well on large datasets.

Does not preserve the relative order of items with equal keys which means it is not stable.

Worst, Average and Best Case Analysis of Algorithms

.

Popular Notations in Complexity Analysis of Algorithms

1. Big-O Notation

We define an algorithm's worst-case time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.

2. Omega Notation

It defines the best case of an algorithm's time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.

3. Theta Notation

It defines the average case of an algorithm's time complexity, the Theta notation defines when the set of functions lies in both O(expression) and Omega(expression), then Theta notation is used. This is how we define a time complexity average case for an algorithm.

<u>**Measurement of Complexity of an Algorithm**</u>

Based on the above three notations of Time Complexity there are three cases to analyze an algorithm:

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be O(n).

2. Best Case Analysis (Very Rarely used)

In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Omega(1)$

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity. Average Case Time = \sum_{i=1}^{n}\frac{\theta (i)}{(n+1)} = \frac{\theta (\frac{(n+1)*(n+2)}{2})}{(n+1)} = \theta (n)

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is $O(n^3)$, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from $O(n^3)$ to $O(n^{\log 7})$.

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes O(N2) time. So time complexity can be written as

T(N) = 7T(N/2) + O(N2)

From Master's Theorem, time complexity of above method is

O(NLog7) which is approximately O(N2.8074)

Generally Strassen's Method is not preferred for practical applications for following reasons.

The constants used in Strassen's method are high and for a typical application Naive method works better. For Sparse matrices, there are better methods especially designed for them.

The submatrices in recursion take extra space. Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

Unit-III

Greedy Method

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

- Candidate set: A solution that is created from the set is known as a candidate set.
- Selection function: This function is used to choose the candidate or subset which can be added in the solution.
- Feasibility function: A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- Objective function: A function is used to assign the value to the solution or the partial solution.
- Solution function: This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Pseudo code of Greedy Algorithm

```
Algorithm Greedy (a, n)
{
  Solution : = 0;
 for i = 0 to n do
 {
    x: = select(a);
   if feasible(solution, x)
   {
     Solution: = union(solution , x)
   }
    return solution;
 } }
```

- The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

AREAS OF APPLICAION

1.FIDING SHORTEST PATH

2.FINDING MAXIMUM SPANNING TREE

3.JOB SEQUENCING WITH DEADINE

4.KNAPSACK PROBLEM AND CONAINER LOADING

CONTAINER LOADING

The greedy algorithm constructs the loading plan of a single container layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size L x W with its initial position at height 0. At each step, the algorithm picks the lowest usable surface and then determines the box type to be packed onto the surface, the number of the boxes and the rectangle area the boxes to be packed onto, by the procedure select layer.

```
                procedure greedy heuristic()
list of surface := initial surface of L x W at height 0 list of box type := all box types
while (there exist usable surfaces) and (not all boxes are packed) do
select the lowest usable surface as current surface set depth := 0
```

Knapsack Problem

An efficient solution is to use the Greedy approach.

The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

<u>This problem can be solved with the help of using two techniques:</u>

- Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
- Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.
- Applications
- Few of the many real-world applications of the knapsack problem are −
- Cutting raw materials without losing too much material
- Picking through the investments and portfolios
- Selecting assets of asset-backed securitization
- Generating keys for the Merkle-Hellman algorithm
- Cognitive Radio Networks
- Power Allocation
- Network selection for mobile nodes

Tree vertex splitting problem greedy method

_ What is the tree vertex splitting problem?

The tree vertex splitting problem is a combinatorial optimization problem that can be stated as follows: Given a weighted tree T = (V, E, w), where V is the set of vertices, E is the set of edges, and w is a function that assigns a positive weight to each edge, and a tolerance limit δ, find a minimum cardinality subset X ⊆ V such that the diameter of the tree T/X obtained by contracting each vertex in X to a single point is at most δ. The diameter of a tree is defined as the maximum distance between any two vertices in the tree, where the distance between two vertices is the sum of the weights of the edges on the path connecting them.

The tree vertex splitting problem can be seen as a generalization of the minimum spanning tree problem, where the goal is to find a subset of edges that connects all the vertices with minimum total weight. In TVSP, we are allowed to contract some vertices to reduce the diameter of the tree, but we want to minimize the number of vertices that are contracted. The problem has applications in network design, where we want to minimize the number of booster stations that are needed to ensure that the signal quality between any two nodes does not fall below a certain threshold.

How can we solve TVSP using a greedy method

A greedy method is a heuristic technique that makes locally optimal choices at each step, hoping that they will lead to a globally optimal solution. A greedy method works well on optimization problems that have two properties: greedy-choice property and optimal substructure. The greedy-choice property means that there is always an optimal solution that contains the first choice made by the greedy method. The optimal substructure means that an optimal solution to the problem contains optimal solutions to subproblems.

The advantages of TVSP using a greedy method are:

It is simple and easy to implement.

It runs in polynomial time.

It always finds a feasible solution if one exists.

The disadvantages of TVSP using a greedy method are:

It does not guarantee to find an optimal solution. In some cases, it may find a solution that is far from optimal.

It is sensitive to the order of the edges. Different orders may lead to different solutions with different cardinalities.

job Sequencing with Deadline

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

Job Scheduling Algorithm

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

Algorithm

Find the maximum deadline value from the input set of jobs.

Once, the deadline is decided, arrange the jobs in descending order of their profits.

Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.

The selected set of jobs are the output.

What are the advantages of job sequencing

In production, the purpose of job sequencing is to minimize the production time and costs, by telling a production facility when to make, with which staff, and on which equipment. Job sequence aims to maximize the efficiency and free flow of the production at a minimum costs and minimum time.

What are the advantages of job sequencing

In production, the purpose of job sequencing is to minimize the production time and costs, by telling a production facility when to make, with which staff, and on which equipment. Job sequence aims to maximize the efficiency and free flow of the production at a minimum costs and minimum time.

Minimum Spanning Tree in Data Structures

A spanning tree is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are n number of vertices, the spanning tree should have n - 1 number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Properties of a Spanning Tree:

The spanning tree holds the below-mentioned principles:

The number of vertices (V) in the graph and the spanning tree is the same.

There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices (E = V-1).

The spanning tree should not be disconnected, as in there should only be a single source of component, not more than that.

The spanning tree should be acyclic, which means there would not be any cycle in the tree.

The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.

There can be many possible spanning trees for a graph.

Algorithms to find Minimum Spanning Tree:

There are several algorithms to find the minimum spanning tree from a given graph, some of them are listed below:

Kruskal's Minimum Spanning Tree Algorithm:

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below:

- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

This algorithm can be implemented efficiently using a DSU (Disjoint-Set) data structure to keep track of the connected components of the graph. This is used in a variety of practical applications such as network design, clustering, and data analysis.

> Follow the article on "Kruskal's Minimum Spanning Tree Algorithm" for a better understanding and implementation of the algorithm.

Prim's Minimum Spanning Tree Algorithm:

This is also a greedy algorithm. This algorithm has the following workflow:

- It starts by selecting an arbitrary vertex and then adding it to the MST.
- Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST.
- This process is continued until all the vertices are included in the MST.

To efficiently select the minimum weight edge for each iteration, this algorithm uses priority_queue to store the vertices sorted by their minimum edge weight currently. It also simultaneously keeps track of the MST using an array or other data structure suitable considering the data type it is storing.

This algorithm can be used in various scenarios such as image segmentation based on color, texture, or other features. For Routing, as in finding the shortest path between two points for a delivery truck to follow.

> *Follow the article on “Prim’s Minimum Spanning Tree Algorithm” for a better understanding and implementation of this algorithm.*

Boruvka’s Minimum Spanning Tree Algorithm:

This is also a graph traversal algorithm used to find the minimum spanning tree of a connected, undirected graph. This is one of the oldest algorithms. The algorithm works by iteratively building the minimum spanning tree, starting with each vertex in the graph as its own tree. In each iteration, the algorithm finds the cheapest edge that connects a tree to another tree, and adds that edge to the minimum spanning tree. This is almost similar to the Prim’s algorithm for finding the minimum spanning tree. The algorithm has the following workflow:

- Initialize a forest of trees, with each vertex in the graph as its own tree.
- For each tree in the forest:
- Find the cheapest edge that connects it to another tree. Add these edges to the minimum spanning tree.
- Update the forest by merging the trees connected by the added edges.
- Repeat the above steps until the forest contains only one tree, which is the minimum spanning tree.

The algorithm can be implemented using a data structure such as a priority queue to efficiently find the cheapest edge between trees. Boruvka’s algorithm is a simple and easy-to-implement algorithm for finding minimum spanning trees, but it may not be as efficient as other algorithms for large graphs with many edges.

Applications of Minimum Spanning Trees:

- Network design: Spanning trees can be used in network design to find the minimum number of connections required to connect all nodes. Minimum spanning trees, in particular, can help minimize the cost of the connections by selecting the cheapest edges.
- Image processing: Spanning trees can be used in image processing to identify regions of similar intensity or color, which can be useful for segmentation and classification tasks.
- Biology: Spanning trees and minimum spanning trees can be used in biology to construct phylogenetic trees to represent evolutionary relationships among species or genes.
- Social network analysis: Spanning trees and minimum spanning trees can be used in social network analysis to identify important connections and relationships among individuals or groups.

Optimal Storage on Tapes

Optimal Storage on Tapes Problem: Given n programs $P_1, P_2, \ldots, P_n$ of length $L_1, L_2, \ldots, L_n$ respectively, store them on a tap of length L such that Mean Retrieval Time (MRT) is a minimum. The retrieval time of the jth program is a summation of the length of first j programs on tap. Let Tj be the time to retrieve program Pj. The retrieval time of P_j is computed as,

$$T_j = \sum_{k=1}^{j} \underset{\substack{\uparrow \\ \text{Length of } k^{th} \text{ program}}}{L_k}$$

or ith problem

Mean retrieval time of n programs is the average time required to retrieve any program. It is required to store programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on single tape only.

There are many permutations of programs. Each gives a different MRT. Consider three programs (P1, P2, P3) with a length of (L1, L2, L3) = (5, 10, 2).

Storage on Multiple Tapes

- This is the problem of minimizing MRT on retrieval of the program from multiple tapes.
- Instead of a single tape, programs are to be stored on multiple tapes. Greedy algorithm solves this problem in a similar way. It sorts the programs according to increasing length of program and stores the program in one by one in each tape.

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a p-record file and a q-record file requires possibly p + q record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files $\{f_1, f_2, f_3, \ldots, f_n\}$. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Dijkstra's Algorithm – Single Source Shortest Path Algorithm

Dijkstra's Algorithm is also known as Single Source Shortest Path (SSSP) problem. It is used to find the shortest path from source node to destination node in graph.

The graph is widely accepted data structure to represent distance map. The distance between cities effectively represented using graph.

Dijkstra proposed an efficient way to find the single source shortest path from the weighted graph. For a given source vertex s, the algorithm finds the shortest path to every other vertex v in the graph.

Assumption : Weight of all edges is non-negative.

Steps of the Dijkstra's algorithm are explained here:

1. Initializes the distance of source vertex to zero and remaining all other vertices to infinity.

2. Set source node to current node and put remaining all nodes in the list of unvisited vertex list. Compute the tentative distance of all immediate neighbour vertex of the current node.

3. If the newly computed value is smaller than the old value, then update it.

For example, C is the current node, whose distance from source S is dist (S, C) = 5.

> Consider N is the neighbour of C and weight of edge (C, N) is 3. So the distance of N from source via C would be 8.
>
> If the distance of N from source was already computed and if it is greater than 8 then relax edge (S, N) and update it to 8, otherwise don't update it.

Weight updating in Dijkstra's algorithm

4. When all the neighbours of a current node are explored, mark it as visited. Remove it from unvisited vertex list. Mark the vertex from unvisited vertex list with minimum distance and repeat the procedure.

5. Stop when the destination node is tested or when unvisited vertex list becomes empty.

Disadvantages of Dijkstra Algorithm

- It conducts a blind scan, which takes a lot of processing time.
- It is unable to manage sharp edges. As a result, acyclic graphs are produced, and the ideal shortest path is frequently impossible to find.
- What is single source shortest path problem and its advantages?
-
- The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph. Algorithms such as Breadth-First-Search (BFS) for unweighted graphs or Dijkstra [1] solve this problem.

Unit-1V

Multistage Graph (Shortest Path)

A Multistage graph is a directed, weighted graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).
The vertices of a multistage graph are divided into n number of disjoint subsets S = { S1 , S2 , S3 ……….. Sn }, where S1 is the source and Sn is the sink (destination). The cardinality of S1 and Sn are equal to 1. i.e., |S1| = |Sn| = 1.
We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Now there are various strategies we can apply :-

The Brute force method of finding all possible paths between Source and Destination and then finding the minimum. That's the WORST possible strategy.

Dijkstra's Algorithm of Single Source shortest paths. This method will find shortest paths from source to all other nodes which is not required in this case. So it will take a lot of time and it doesn't even use the SPECIAL feature that this MULTI-STAGE graph has.

Simple Greedy Method – At each node, choose the shortest outgoing path. If we apply this approach to the example graph given above we get the solution as 1 + 4 + 18 = 23. But a quick look at the graph will show much shorter paths available than 23. So the greedy method fails !

The best option is Dynamic Programming. So we need to find Optimal Sub-structure, Recursive Equations and Overlapping Sub-problems.

Time and Auxiliary Space

The time complexity of the multistage graph shortest path algorithm depends on the number of vertices and the number of stages in the graph. The outer loop iterates over the stages, which takes O(k) time. The inner loop iterates over the vertices in each stage, and for each vertex, it examines its adjacent vertices. Since the graph is represented as an adjacency list, this takes O(E) time, where E is the number of edges in the graph. Therefore, the total time complexity of the algorithm is O(kE).

The space complexity of the algorithm is O(V), where V is the number of vertices in the graph. This is because we store the shortest path distances for each vertex in a list of size V. Additionally, we store the graph as an adjacency list, which also requires O(V) space.

All-Pairs Shortest Paths

The literal meaning of this problem is that we want to compute the shortest path from every node in the graph to every other node. Putting it formally:

Let G = (V, E) where,

G represents the given graph,

V is the set of all vertices in the graph G

E represents the set of all edges in the graph G

The All Pair Shortest Path problem wants to compute the shortest path from each vertex v ∈ V to every other vertex u ∈ V.

Requirement of All Pair Shortest Path

There are various scenarios where we can apply the All Pair Shortest Path approach to reach the solution. Let us see some of them:

1. This algorithm is used in finding the transitive closure of directed graphs. You can go through *Transitive closure of a Graph* article to understand how the Floyd Warshall Algorithm (we will discuss this shortly) is used to compute the transitive closure of a directed graph.

2. Inversion of real matrices

3. To check if a given graph is bipartite or not. You can check out this problem statement from Coding Ninjas Studio to gather more knowledge around this problem.

4. Faster computation of PathFinder Networks

Apart from these scenarios there are numerous coding problems which can be solved using the application of any All Pair Shortest Path algorithm. These coding techniques can either involve the idea of All Pair Shortest Path as the central idea with minor modifications or this All Pair Shortest Path concept may be one of its components to get to the final solution.

Algorithms to Find All Pair Shortest Path

There are mainly two algorithms to compute all pair shortest paths in a given graph. We have elaborate articles explaining each of these algorithms. These are:

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is a dynamic programming algorithm which is used to compute the all pair shortest paths in a directed or undirected graph. The time and space complexities of this algorithm are O(V^3) and O(V^2) respectively where V is the number of vertices in the graph.

Detailed explanation of this algorithm can be found in Coding Ninjas Studio article: Floyd-Warshall Algorithm

Johnson's Algorithm

Johnson's algorithm is a combination of two well-known algorithms for computing shortest paths in a graph for one node: Dijkstra's algorithm and Bellman-Ford algorithm. It uses both these algorithms to arrive at the final solution. The time complexity of this algorithm is O((V^2)logV + VE) where V is the number of vertices in the graph and E represents the count of edges in the graph.

Optimal Binary Search Tree

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2). Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.

The OBST is a useful data structure in applications where the keys have different probabilities of being searched for. It can be used to improve the efficiency of searching and retrieval operations in databases, compilers, and other computer programs.

Given a sorted array key [0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches for keys[i]. Construct a binary search tree of all keys such

that the total cost of all the searches is as small as possible.
Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

String editing

There are two strings given. The first string is the source string and the second string is the target string. In this program, we have to find how many possible edits are needed to convert first string to the second string.

The edit of strings can be either Insert some elements, delete something from the first string or modify something to convert into the second string.

Reliability Design Problem in Dynamic Programming

Let's say, we have to set up a system consisting of D_1, D_2, D_3, …………, and D_n devices, each device has some costs C_1, C_2, C_3, …….., C_n. Each device has a reliability of 0.9 then the entire system has reliability which is equal to the product of the reliabilities of all devices i.e., $\pi r_i = (0.9)^4$.

It means that 35% of the system has a chance to fail, due to the failure of any one device. the problem is that we want to construct a system whose reliability is maximum. How it can be done so? we can think that we can take more than one copy of each device so that if one device fails we can use the copy of that device, which means we can connect the devices parallel.

Travelling Salesman Problem using Dynamic Programming

Travelling Salesman Problem (TSP):

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.
Time Complexity : $O(n^2*2^n)$ where $O(n* 2^n)$ are maximum number of unique subproblems/states and O(n) for transition (through for loop as in code) in every states.
Auxiliary Space: $O(n*2^n)$, where n is number of Nodes/Cities here.

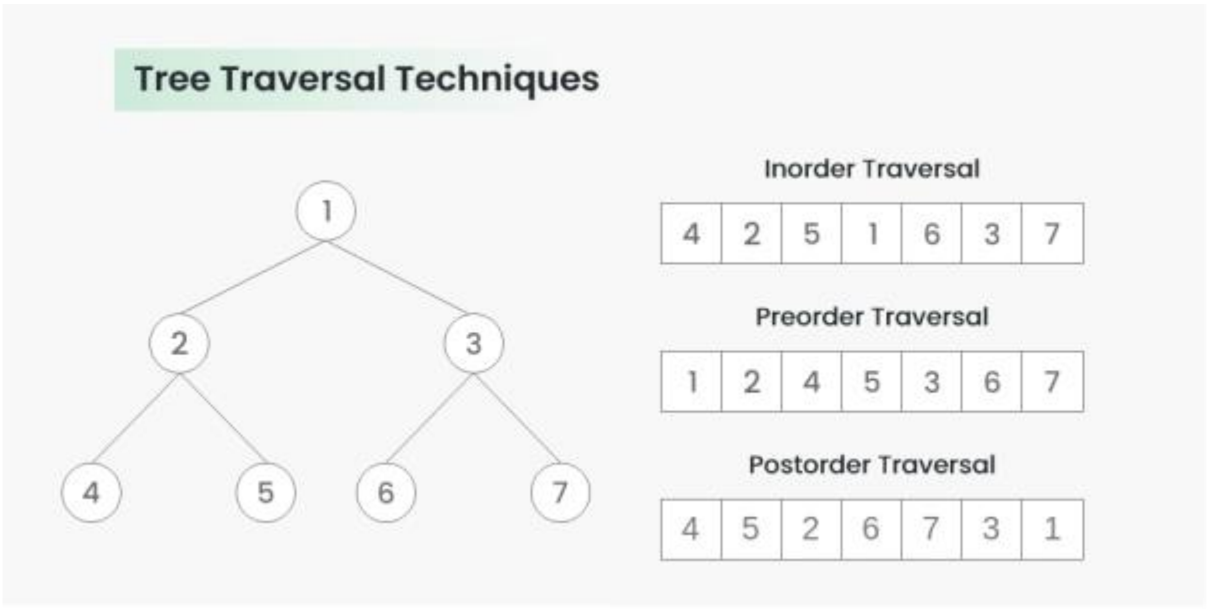
For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them. Using the above recurrence relation, we can write a dynamic programming-based solution. There are at most O($n*2^n$) subproblems, and each one takes linear time to solve. The total running time is therefore O(n^2*2^n). The time complexity is much less than O(n!) but still exponential. The space required is also exponential. So this approach is also infeasible even for a slightly higher number of vertices. We will soon be discussing approximate algorithms for the traveling salesman problem.

Tree Traversal Techniques – Data Structure and Algorithm Tutorials

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

A Tree Data Structure can be traversed in following ways:

1. Depth First Search or DFS
    1. Inorder Traversal
    2. Preorder Traversal
    3. Postorder Traversal
2. Level Order Traversal or Breadth First Search or BFS
3. Boundary Traversal
4. Diagonal Traversal



Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right->subtree)

Uses of Inorder Traversal:

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

Preorder Traversal:

Algorithm Preorder(tree)

1. *Visit the root.*
2. *Traverse the left subtree, i.e., call Preorder(left->subtree)*
3. *Traverse the right subtree, i.e., call Preorder(right->subtree)*

Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

Postorder Traversal:

Algorithm Postorder(tree)

1. *Traverse the left subtree, i.e., call Postorder(left->subtree)*
2. *Traverse the right subtree, i.e., call Postorder(right->subtree)*
3. *Visit the root*

Uses of Postorder:

Postorder traversal is used to delete the tree. Please see the question for the deletion of a tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree

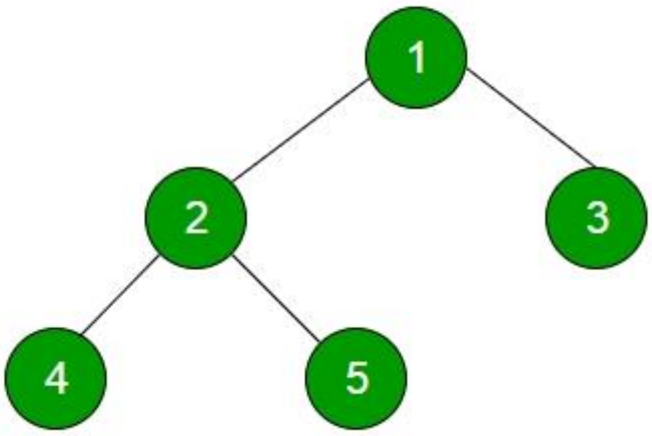
Some other Tree Traversals Techniques:

Some of the other tree traversals are:

Level Order Treversal

For each node, first, the node is visited and then it's child nodes are put in a FIFO queue. Then again the first node is popped out and then it's child nodes are put in a FIFO queue and repeat until queue becomes empty.

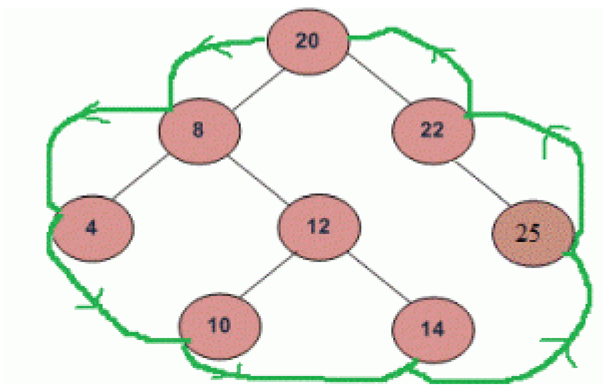
Example:



Boundary Traversal

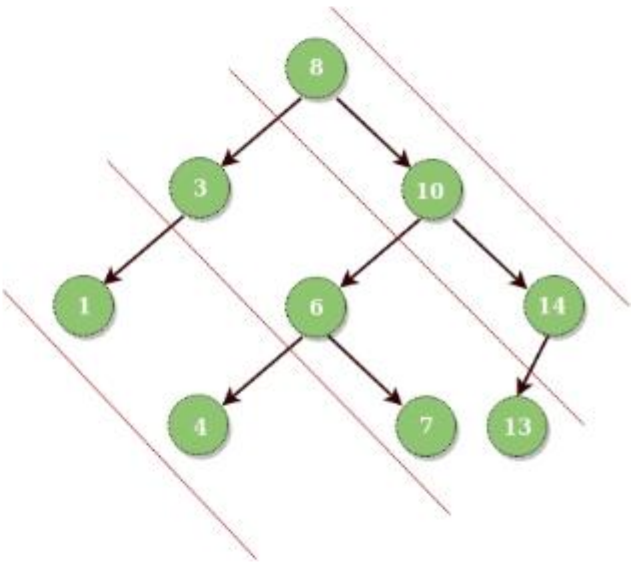
The Boundary Traversal of a Tree includes:

1. left boundary (nodes on left excluding leaf nodes)
2. leaves (consist of only the leaf nodes)
3. right boundary (nodes on right excluding leaf nodes)



Diagonal Traversal

In the Diagonal Traversal of a Tree, all the nodes in a single diagonal will be printed one by one.



Diagonal Traversal of binary tree:

8 10 14

3 6 7 13

1 4

Graph Data Structure And Algorithms

Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

Breadth First Search or BFS for a Graph

The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

How does BFS work?

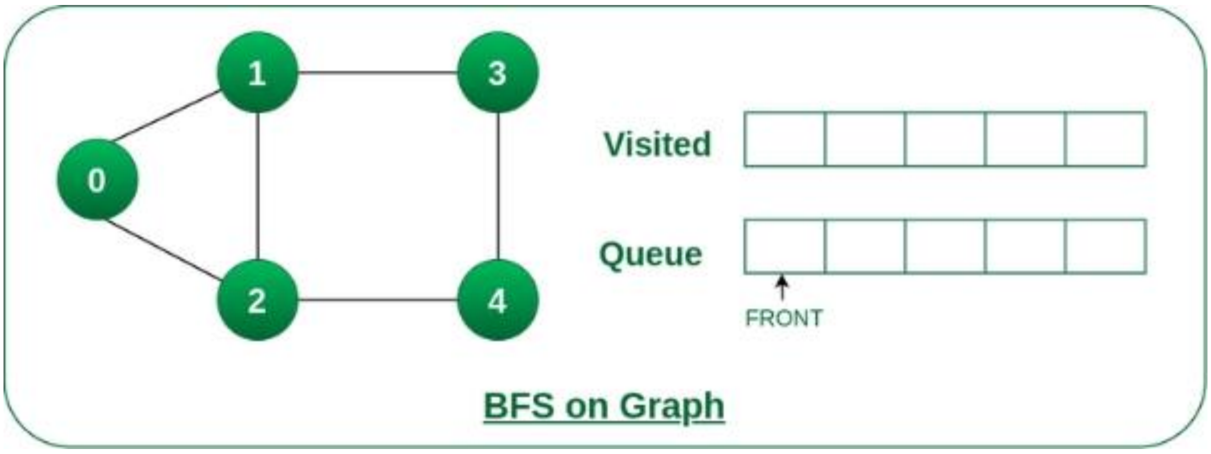
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

Illustration:

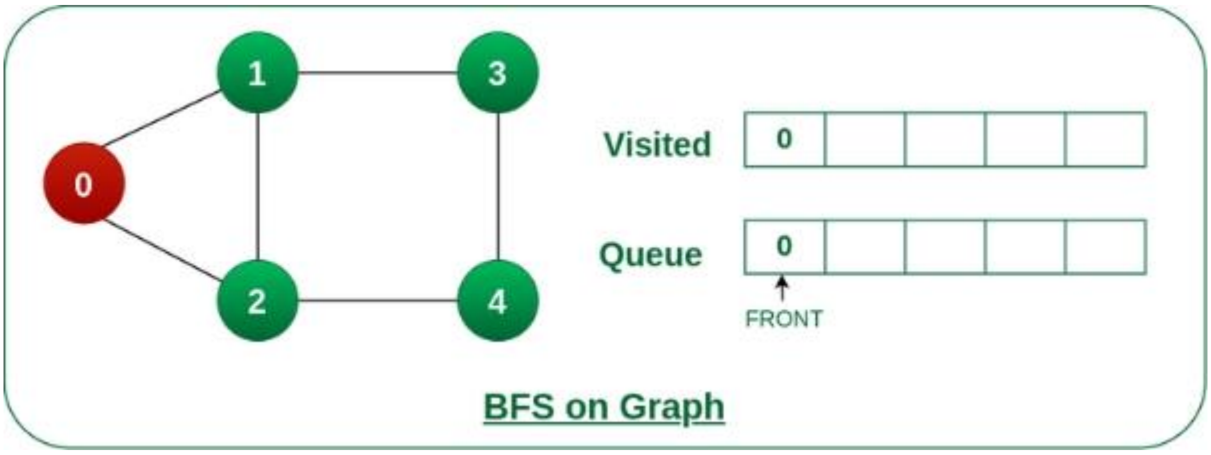
Let us understand the working of the algorithm with the help of the following example.

Step1: Initially queue and visited arrays are empty.



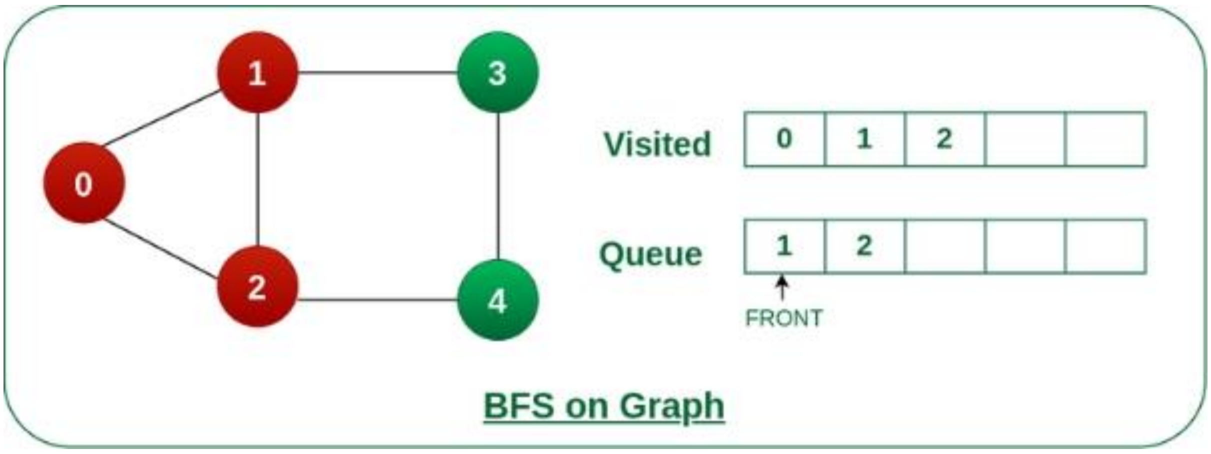
Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.



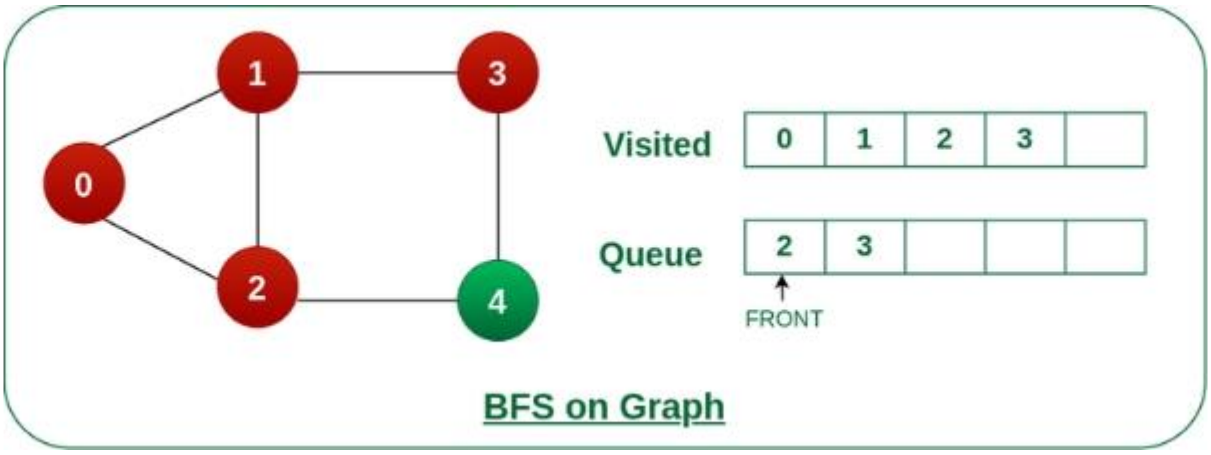
Push node 0 into queue and mark it visited.

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



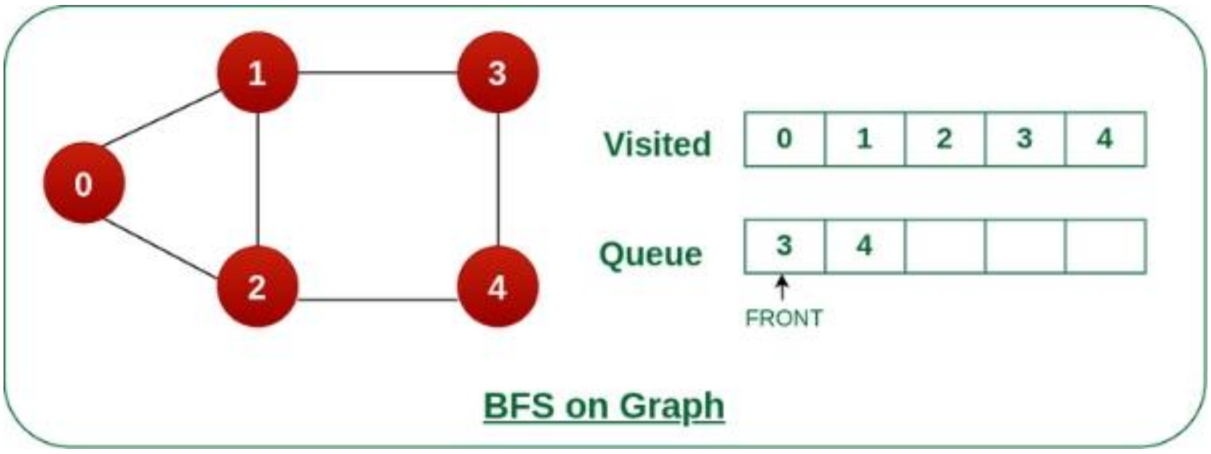
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



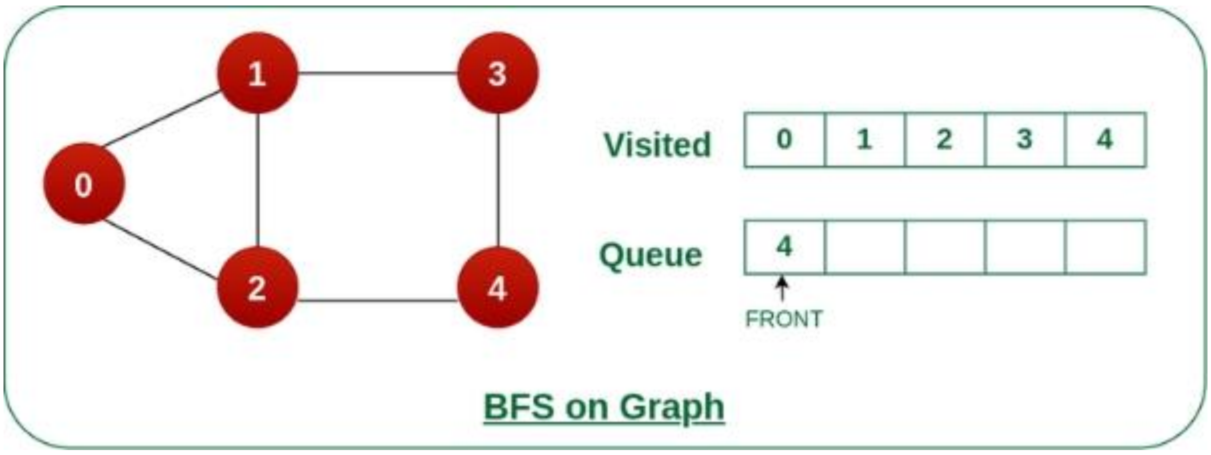
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



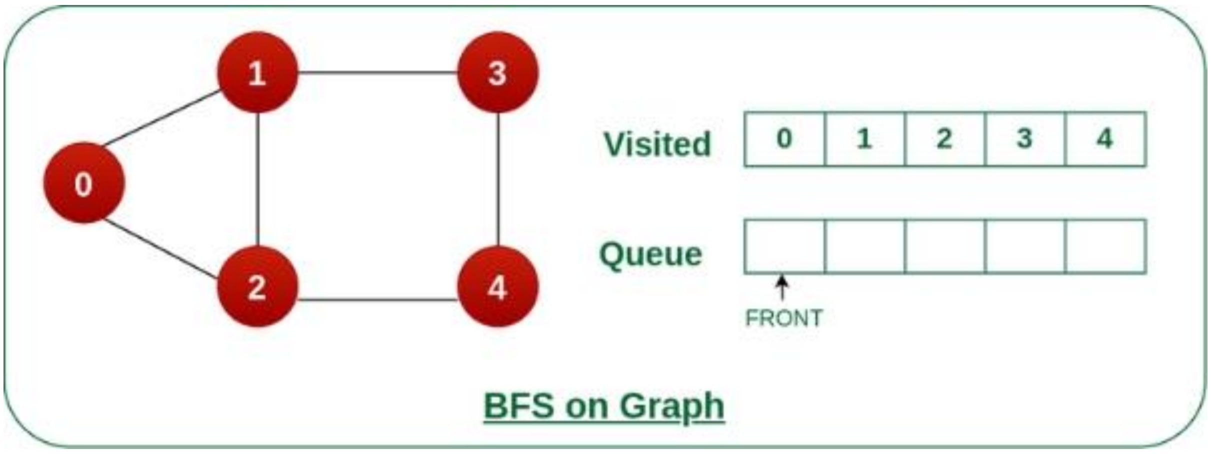
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Depth First Search (DFS)

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

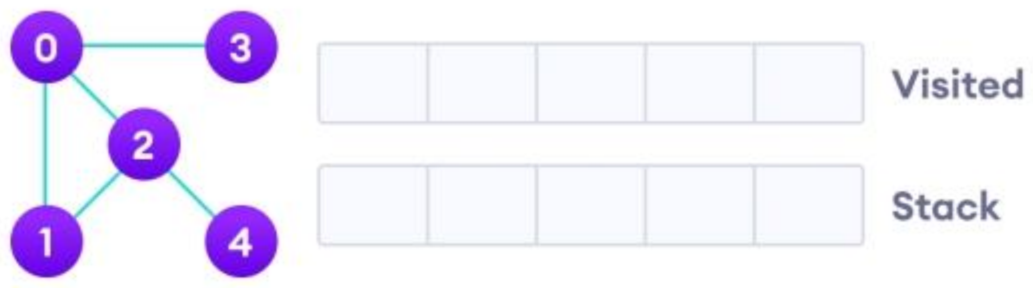
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

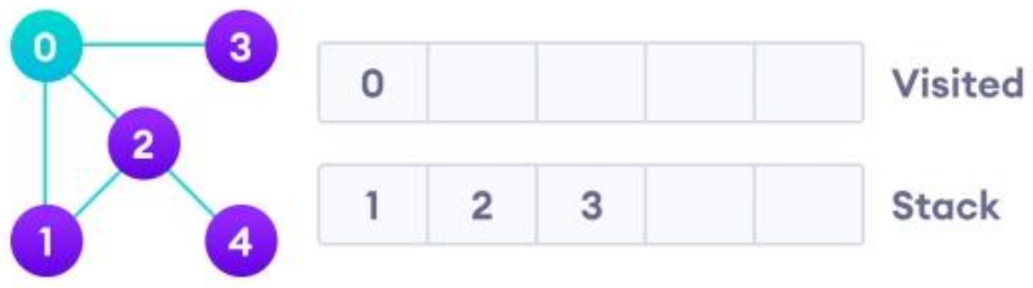
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



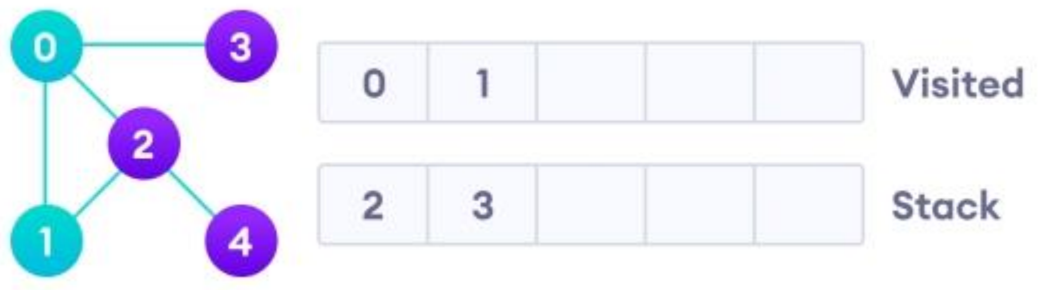
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



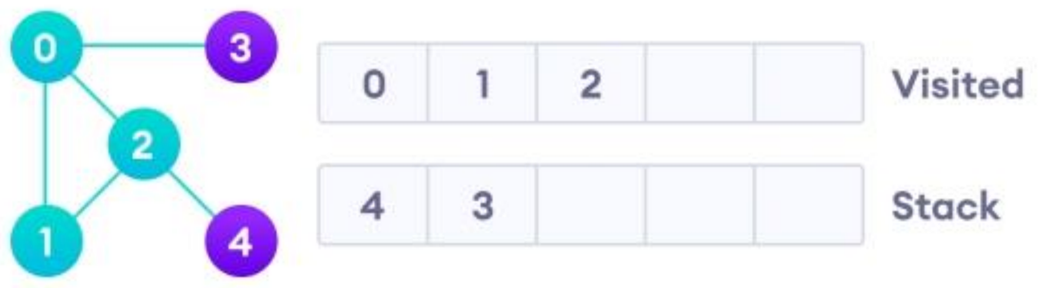
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

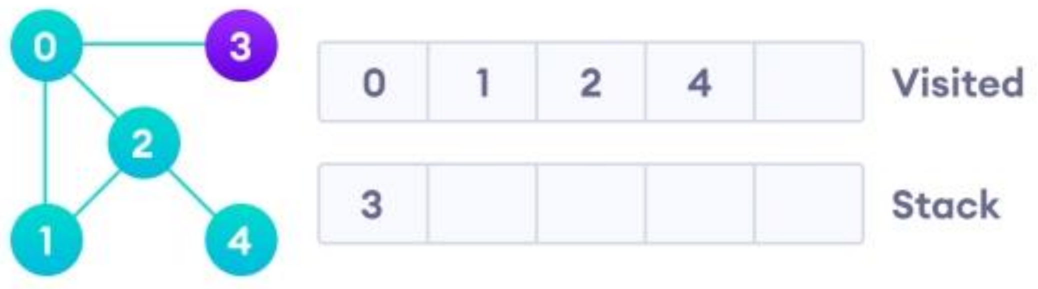


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

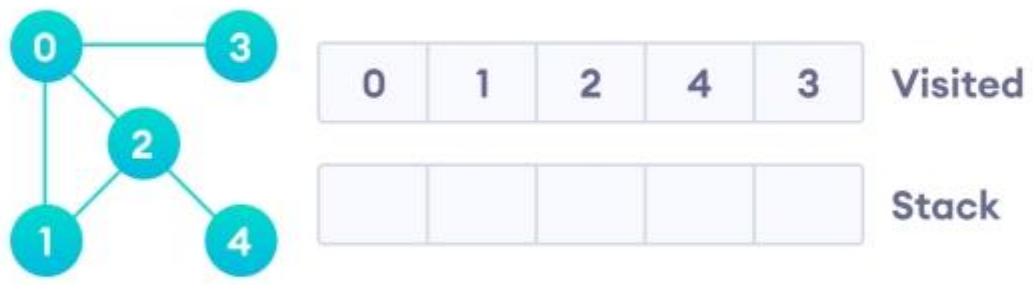


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

Introduction to Backtracking – Data Structure and Algorithm Tutorials

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree). Backtracking can also be said as an improvement to the brute force approach. So basically, the idea behind the backtracking technique is that it searches for a solution to a problem among all the available options. Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options. There also might be a case where none of the options will give you the solution and hence we understand that backtracking won't give any solution to that particular problem. We can also say that backtracking is a form of recursion. This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or we reach the final state. So we can conclude that backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.
According to the wiki definition,

Backtracking
can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.
Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Basic terminologies:Solution vector:

The desired solution X to a problem instance P of input size n is as a vector of candidate solutions that are selected from some finite set of possible solutions S.

Thus, a solution can be represented as an n-tuple (X1, X2, …, Xn) and its partial solution is given as (X1, X2,…, Xi) where i<n.

E.g. for a 4-queens problem X = {2,4,1,3) is a solution vector.

- Constraints:

Constraints are the rules to confine the solution vector (X1, X2…… Xa).

They determine the values of candidate solutions and their relationship with each other. There are two types of constraints:

(1) implicit constraints

(2) explicit constraints.

1. *Implicit constraints:*

These are the rules that identify the tuples in the solution space S that satisfy the specified criterion function of a problem instance P. They give the directives of relating all candidate solutions x's to each other.

E.g., in the case of the N-queens problem, all xi’s must be distinct satisfying the criterion function of non- attacking queens, in the case of the 0/1 knapsack problem all x’s with value ‘I’ must represent the item giving overall maximum profit and having total weight S knapsack capacity.

2.Explicit constraints:
These are the rules by which all candidate solutions x_{i} ^ prime s are restricted to take on values only from a specified set in a problem instance P. Explicit constraints vary with the instances of the problem.

Eg., in case of the N-queens problem, if N = 4 then x i in \{1, 2, 3, 4\} and if then x i in[1, 2, 3, 4, 5, 6 N = 8 7,8); in case of 0/1 knapsack problem xe \{0, 1\} where x_{i} = 0 represents the exclusion of an item i and x_{i} = 1 represents the inclusion of an item i.

- Solution space:

All candidate solutions xi’s satisfying the explicit constraints form the solution space S of a problem instance P. In a state space tree, all paths from the root node to a leaf node describe the solution space.

Eg of in the case of N-queens problem, all n! orderings (x_{1}, x_{2} ,…,x n) form the solution space of that problem instance.

- State space tree:

A representation of the solution space S of a problem instance P in the form of a tree is defined as the state space tree.

It facilitates systematic search in the solution space to determine the desired solution to a problem.

A solution space of a given problem can be represented by different state space trees.

- State space:

The state space of a problem is described by all paths from a root node to other nodes in a state space tree.

- Problem state:

Each node in a state space tree describes a problem state or a partial solution formed by making choices from the root of the tree to that node.

Solution states:

These are the problem states producing a tuple in the solution space S. At every internal node, the solution states are partitioned into disjoint sub-solution spaces. In a state space tree for a variable tuple size, all nodes are solution states.

In a state space tree for a fixed tuple size, only the leaf nodes are solution states.

- Answer states:

These are the solution states that satisfy the implicit constraints.

These states thus describe the desired solution-tuple (or answer-tuple).

- Promising node:

A node is promising if it eventually leads to the desired solution.

A promising node corresponds to a partial solution that is still feasible.

Any time the partial node becomes infeasible, that branch will no longer be pursued.

- Non-promising node:

A node is non-promising if it eventually leads to a state that cannot produce the desired solution. A non-promising node corresponds to a partial solution that shows infeasibility to get a complete solution.

Such nodes are killed by a bounding function without further exploration.

- Live node:

It is a node that has been generated and all of whose children are not yet been generated.

E-node:

It is a live node whose children are currently being generated.

- Dead node:

A node that is either not to be expanded further or for which all its children have been generated is known as a dead node.

- Depth first node generation:

Here, the latest live node becomes the next E-node.The moment a new child of the current E-node is generated, that child will be the new E-node. In backtracking, a state space tree is constructed by using the depth first node generation approach.

‣ Bounding function:

It is also known as a “validity function”, or “criterion function”, or “promising function”.

It is an optimization function B(x1, x2. Xa) which is to be either maximized or minimized for a given problem instance P.

It optimizes the search of a solution vector (X1, X2,… Xn) in the solution space S of a problem instance P.

It helps to reject the candidate solutions not leading to the desired solution to the problem. Thus, it kills the live nodes without exploring their children if constraints are not satisfied.

Eg, in the case of the knapsack problem, the criterion function is the maximization of the profit by filling a knapsack.

Static trees:

These are the state space trees whose tree formulation is independent of the problem instance being solved.

- Dynamic trees:

These are the state space trees whose tree formulation varies with the problem instance being solved.

Difference between Recursion and Backtracking:
In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Unit V

8 Queen Problem Using Backtracking

Backtracking is a recursive approach for solving any problem where we must search among all the possible solutions following some constraints. More precisely, we can say that it is an improvement over the brute force technique. In this blog, we will learn one
We will retain the backtracking approach and go through its time optimization. Let's start with the problem statement of solving 8 queens problem .
Problem Statement
Given an 8x8 chess board, you must place 8 queens on the board so that no two queens attack each other. Print all possible matrices satisfying the conditions with positions with queens marked with '1' and empty spaces with '0'. You must solve the 8 queens problem using backtracking.

- Note 1: A queen can move vertically, horizontally and diagonally in any number of steps.
- Note 2: You can also go through the N-Queen Problem for the general approach to solving this problem.

objective function in 8 queens problem

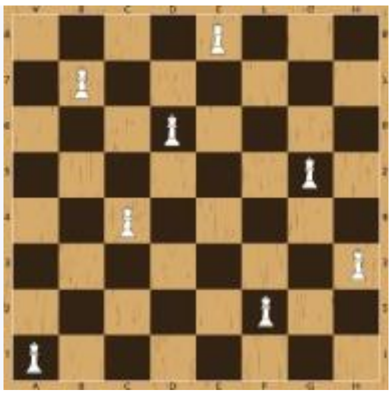
The objective function will count the number of queens that are positioned in a place where they cannot be attacked. Given that queens move vertically, it's reasonable to say that no queen should be placed in the same vertical and thus we can represent the position of each queen in a simple array of 8 positions.

complexity of the 8 queens problem

A simple brute-force solution would be to generate all possible chess boards with 8 queens. Accordingly, there would be N^2 positions to place the first queen, N^2 – 1 position to place the second queen and so on. The total time complexity, in this case, would be O(N^(2N)), which is too high.

Sample Example

Example: One possible solution to the 8 queens problem using backtracking is shown below. In the first row, the queen is at E8 square, so we have to make sure no queen is in column E and row 8 and also along its diagonals. Similarly, for the second row, the queen is on the B7 square, thus, we have to secure its horizontal, vertical, and diagonal squares. The same pattern is followed for the rest of the queens.



Output:

```
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

```
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
```

Bruteforce Approach

One brute-force approach to solving this problem is as follows:

- Generate all possible permutations of the numbers 1 to 8, representing the columns on the chessboard.
- For each permutation, check if it represents a valid solution by checking that no two queens are in the same row or diagonal.
- If a valid solution is found, print the board layout.

While this approach works for small numbers, it quickly becomes inefficient for larger sizes as the number of permutations to check grows exponentially. More efficient algorithms, such as backtracking or genetic algorithms, can be used to solve the problem in a more optimized way.

Backtracking Approach

This approach rejects all further moves if the solution is declined at any step, goes back to the previous step and explores other options.

Algorithm

Let's go through the steps below to understand how this algorithm of solving the 8 queens problem using backtracking works:

- Step 1: Traverse all the rows in one column at a time and try to place the queen in that position.
- Step 2: After coming to a new square in the left column, traverse to its left horizontal direction to see if any queen is already placed in that row or not. If a queen is found, then move to other rows to search for a possible position for the queen.
- Step 3: Like step 2, check the upper and lower left diagonals. We do not check the right side because it's impossible to find a queen on that side of the board yet.
- Step 4: If the process succeeds, i.e. a queen is not found, mark the position as '1' and move ahead.
- Step 5: Recursively use the above-listed steps to reach the last column. Print the solution matrix if a queen is successfully placed in the last column.
- Step 6: Backtrack to find other solutions after printing one possible solution.

Subset Sum Problem using Backtracking

Subset sum can also be thought of as a special case of the [0–1 Knapsack problem](). For each item, there are two possibilities:

> *Include the current element in the subset and recur for the remaining elements with the remaining Sum.*

> *Exclude the current element from the subset and recur for the remaining elements.*

Finally, if Sum becomes 0 then print the elements of current subset. The recursion's base case would be when no items are left, or the sum becomes negative, then simply return.

In this problem, there is a given set with some integer elements. And another some value is also provided, we have to find a subset of the given set whose sum is the same as the given sum value.

Here backtracking approach is used for trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.

Input and Output

```
Input:

This algorithm takes a set of numbers, and a sum value.

The Set: {10, 7, 5, 18, 12, 20, 15}

The sum Value: 35

Output:

All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.

{10,  7,  18}

{10,  5,  20}

{5,  18,  12}

{20,  15}
```

Algorithm

```
subsetSum(set, subset, n, subSize, total, node, sum)
```

Input − The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

Output − All possible subsets whose sum is the same as the given sum.

```
Begin
  if total = sum, then
    display the subset
    //go for finding next subset
    subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)
    return
  else
    for all element i in the set, do
      subset[subSize] := set[i]
      subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)
    done
End
```

<u>Advantage</u>

Real-world applications: The Subset Sum Problem has practical applications in various fields, including finance, where it is used to determine the minimum number of coins to make a given change. It is also used in resource allocation problems, scheduling problems, and bin packing problems.

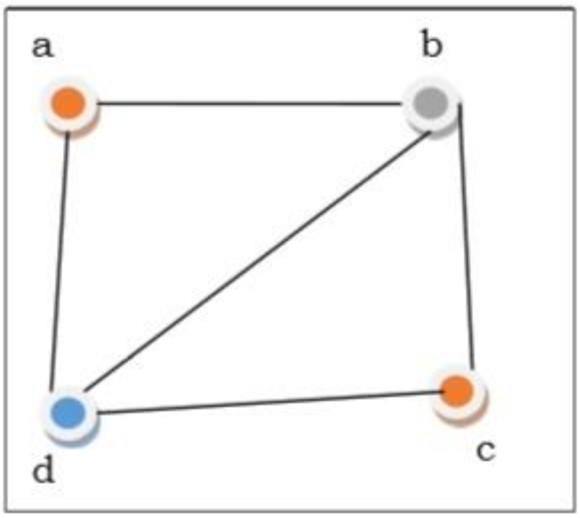
The Graph Coloring

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

Method to Color a Graph

The steps required to color a graph G with n number of vertices are as follows −

Step 1 − Arrange the vertices of the graph in some order.
Step 2 − Choose the first vertex and color it with the first color.
Step 3 − Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.



In the above figure, at first vertex a is colored red. As the adjacent vertices of vertex a are again adjacent, vertex b and vertex d are colored with different color, green and blue respectively. Then vertex c is colored as red as no adjacent vertex of c is colored red. Hence, we could color the graph by 3 colors. Hence, the chromatic number of the graph is 3.

Applications of Graph Coloring

Some applications of graph coloring include −

- Register Allocation
- Map Coloring
- Bipartite Graph Checking

- Mobile Radio Frequency Assignment
- Making time table, etc.

Hamiltonian Cycle

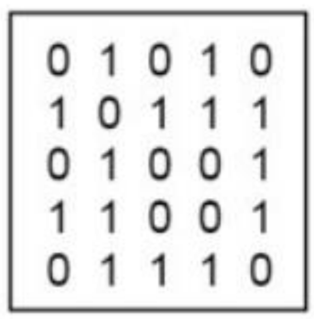
In an undirected graph, the Hamiltonian path is a path, that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, that there is an edge from the last vertex to the first vertex.

In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

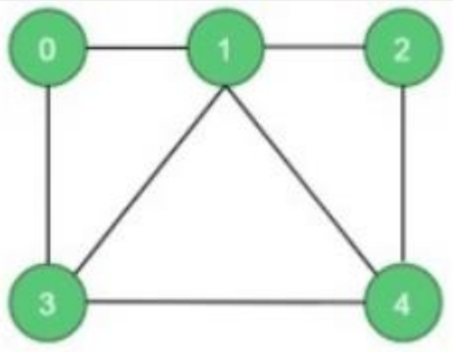
Input and Output

Input:

The adjacency matrix of a graph G(V, E).



0	1	0	1	0
1	0	1	1	1
0	1	0	0	1
1	1	0	0	1
0	1	1	1	0



Output:

The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.

If one graph has no Hamiltonian path, the algorithm should return false.

Algorithm

isValid(v, k)

Input − Vertex v and position k.

Output − Checks whether placing v in the position k is valid or not.

```
Begin

   if there is no edge between node(k-1) to v, then

      return false

   if v is already taken, then

      return false

   return true; //otherwise it is valid

End
```

cycleFound(node k)

Input − node of the graph.

Output − True when there is a Hamiltonian Cycle, otherwise false.

```
Begin

   if all nodes are included, then

      if there is an edge between nodes k and 0, then

         return true

      else

         return false;



   for all vertex v except starting point, do

      if isValid(v, k), then //when v is a valid edge

         add v into the path

         if cycleFound(k+1) is true, then

            return true

         otherwise remove v from the path
```

```
  done

  return false

End
```

Branch and bound

What is Branch and Bound Algorithm?

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.

A branch and bound algorithm provide an optimal solution to an NP-Hard problem by exploring the entire search space. Through the exploration of the entire search space, a branch and bound algorithm identify possible candidates for solutions step-by-step.

There are many optimization problems in computer science, many of which have a finite number of the feasible shortest path in a graph or minimum spanning tree that can be solved in polynomial time. Typically, these problems require a worst-case scenario of all possible permutations. The branch and bound algorithm create branches and bounds for the best solution.

Different search techniques in branch and bound:

The Branch algorithms incorporate different search techniques to traverse a state space tree. Different search techniques used in B&B are listed below:

1. LC search
2. BFS
3. DFS
4. 1. LC search (Least Cost Search):
5. It uses a heuristic cost function to compute the bound values at each node. Nodes are added to the list of live nodes as soon as they get generated. The node with the least value of a cost function selected as a next E-node.

6. 2.BFS(Breadth First Search):
   It is also known as a FIFO search.
   It maintains the list of live nodes in first-in-first-out order i.e, in a queue, The live nodes are searched in the FIFO order to make them next E-nodes.
7. 3. DFS (Depth First Search):
   It is also known as a LIFO search.
   It maintains the list of live nodes in last-in-first-out order i.e. in a stack.

When to apply Branch and Bound Algorithm

Branch and bound is an effective solution to some problems, which we have already discussed. We'll discuss all such cases where branching and binding are appropriate in this section.

- It is appropriate to use a branch and bound approach if the given problem is discrete optimization. Discrete optimization refers to problems in which the variables belong to the discrete set. Examples of such problems include 0-1 Integer Programming and Network Flow problems.
- When it comes to combinatory optimization problems, branch and bound work well. An optimization problem is optimized by combinatory optimization by finding its maximum or minimum based on its objective function. The combinatory optimization problems include Boolean Satisfiability and Integer Linear Programming.

Basic Concepts of Branch and Bound:

‣ Generation of a state space tree:

As in the case of backtracking, B&B generates a state space tree to efficiently search the solution space of a given problem instance.

In B&B, all children of an E-node in a state space tree are produced before any live node gets converted in an E-node. Thus, the E-node remains an E-node until i becomes a dead node.

- Evaluation of a candidate solution:

Unlike backtracking, B&B needs additional factors evaluate a candidate solution:

1. A way to assign a bound on the best values of the given criterion functions to each node in a state space tree: It is produced by the addition of further components to the partial solution given by that node.
2. The best values of a given criterion function obtained so far: It describes the upper bound for the maximization problem and the lower bound for the minimization problem.

- A feasible solution is defined by the problem states that satisfy all the given constraints.
- An optimal solution is a feasible solution, which produces the best value of a given objective function.
- Bounding function :

It optimizes the search for a solution vector in the solution space of a given problem instance.

It is a heuristic function that evaluates the lower and upper bounds on the possible solutions at each node. The bound values are used to search the partial solutions leading to an optimal solution. If a node does not produce a solution better than the best solution obtained thus far, then it is abandoned without further exploration.

The algorithm then branches to another path to get a better solution. The desired solution to the problem is the value of the best solution produced so far.

‣ *The reasons to dismiss a search path at the current node :*

(i) The bound value of the node is lower than the upper bound in the case of the maximization problem and higher than the lower bound in the case of the minimization problem. (i.e. the bound value of the ade is not better than the value of the best solution obtained until that node).

(ii) The node represents infeasible solutions, de violation of the constraints of the problem.

(iii) The node represents a subset of a feasible solution containing a single point. In this case, if the latest solution is better than the best solution obtained so far the best solution is modified to the value of a feasible solution at that node.

Types of Branch and Bound Solutions:
The solution of the Branch and the bound problem can be represented in two ways:

- Variable size solution: Using this solution, we can find the subset of the given set that gives the optimized solution to the given problem. For example, if we have to select a combination of elements from {A, B, C, D} that optimizes the given problem, and it is found that A and B together give the best solution, then the solution will be {A, B}.
- Fixed-size solution: There are 0s and 1s in this solution, with the digit at the ith position indicating whether the ith element should be included, for the above example, the solution will be given by {1, 1, 0, 0}, here 1 represent that we have select the element which at ith position and 0 represent we don’t select the element at ith position.

Classification of Branch and Bound Problems:
The Branch and Bound method can be classified into three types based on the order in which the state space tree is searched.

1. FIFO Branch and Bound

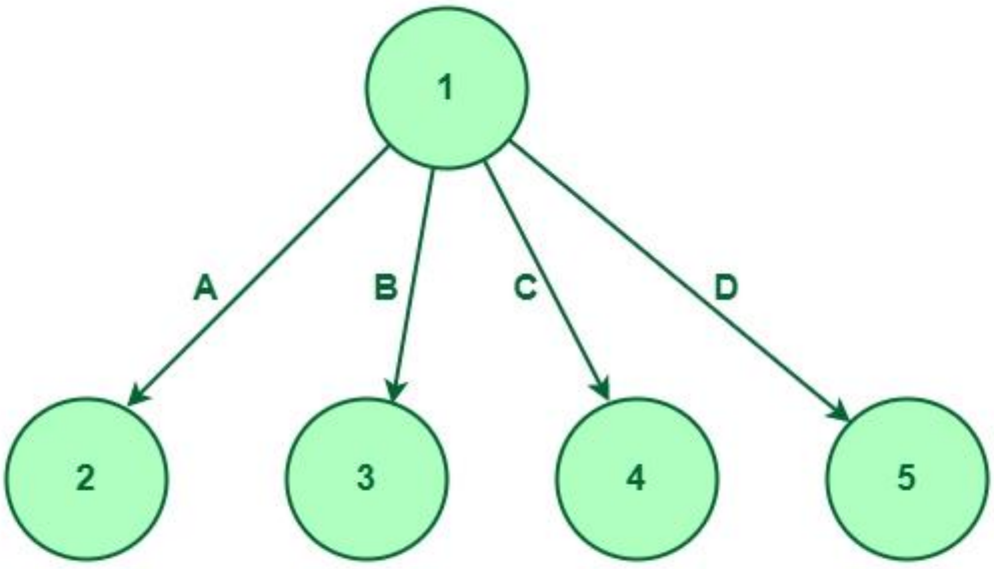
2. LIFO Branch and Bound
3. Least Cost-Branch and Bound

We will now discuss each of these methods in more detail. To denote the solutions in these methods, we will use the variable solution method.

1. FIFO Branch and Bound

First-In-First-Out is an approach to the branch and bound problem that uses the queue approach to create a state-space tree. In this case, the breadth-first search is performed, that is, the elements at a certain level are all searched, and then the elements at the next level are searched, starting with the first child of the first node at the previous level.

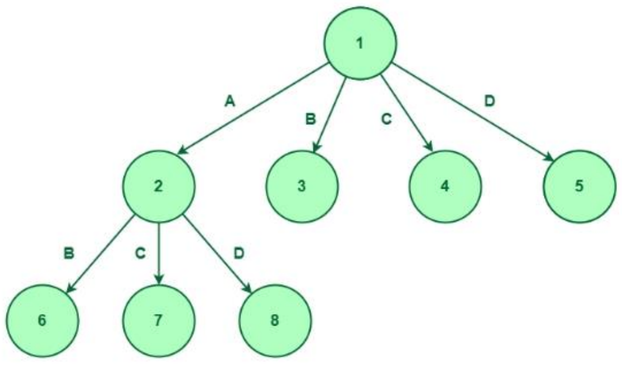
For a given set {A, B, C, D}, the state space tree will be constructed as follows :



State Space tree for set {A, B, C, D}

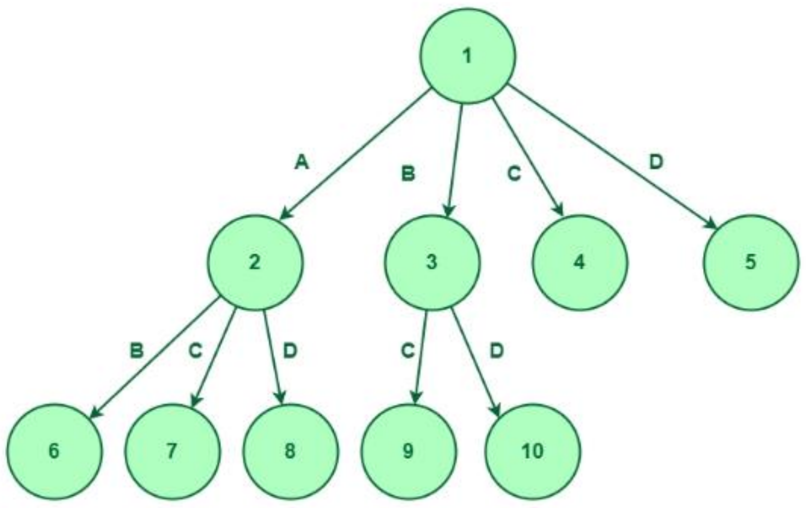
The above diagram shows that we first consider element A, then element B, then element C and finally we'll consider the last element which is D. We are performing BFS while exploring the nodes.

So, once the first level is completed. We'll consider the first element, then we can consider either B, C, or D. If we follow the route then it says that we are doing elements A and D so we will not consider elements B and C. If we select the elements A and D only, then it says that we are selecting elements A and D and we are not considering elements B and C.



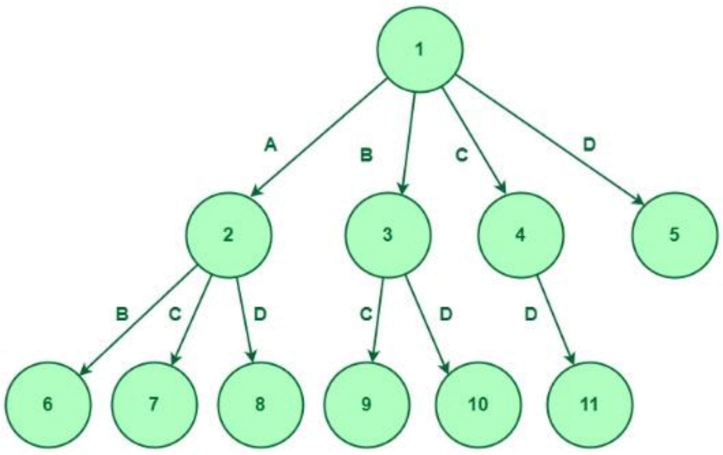
Selecting element A

Now, we will expand node 3, as we have considered element B and not considered element A, so, we have two options to explore that is elements C and D. Let’s create nodes 9 and 10 for elements C and D respectively.



Considered element B and not considered element A

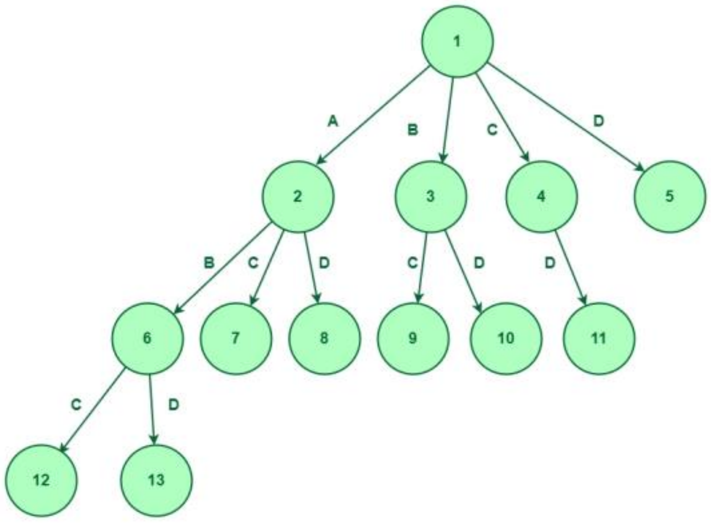
Now, we will expand node 4 as we have only considered elements C and not considered elements A and B, so, we have only one option to explore which is element D. Let’s create node 11 for D.



Considered elements C and not considered elements A and B

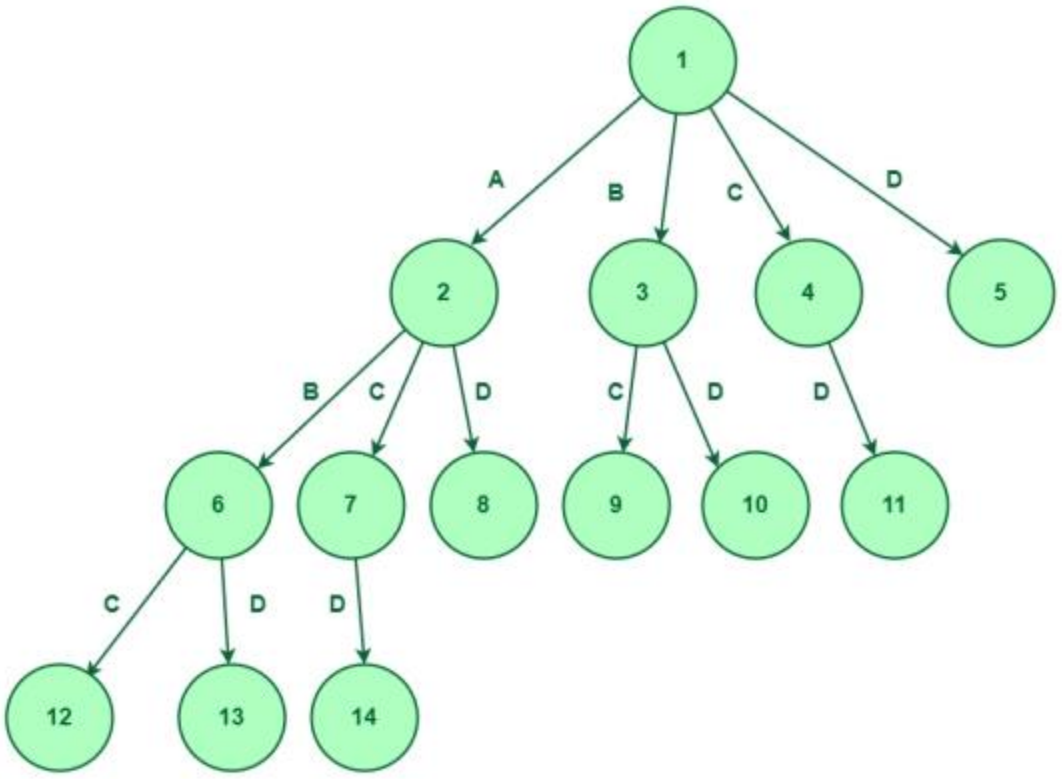
Till node 5, we have only considered elements D, and not selected elements A, B, and C. So, We have no more elements to explore, Therefore on node 5, there won't be any expansion.

Now, we will expand node 6 as we have considered elements A and B, so, we have only two option to explore that is element C and D. Let's create node 12 and 13 for C and D respectively.



Expand node 6

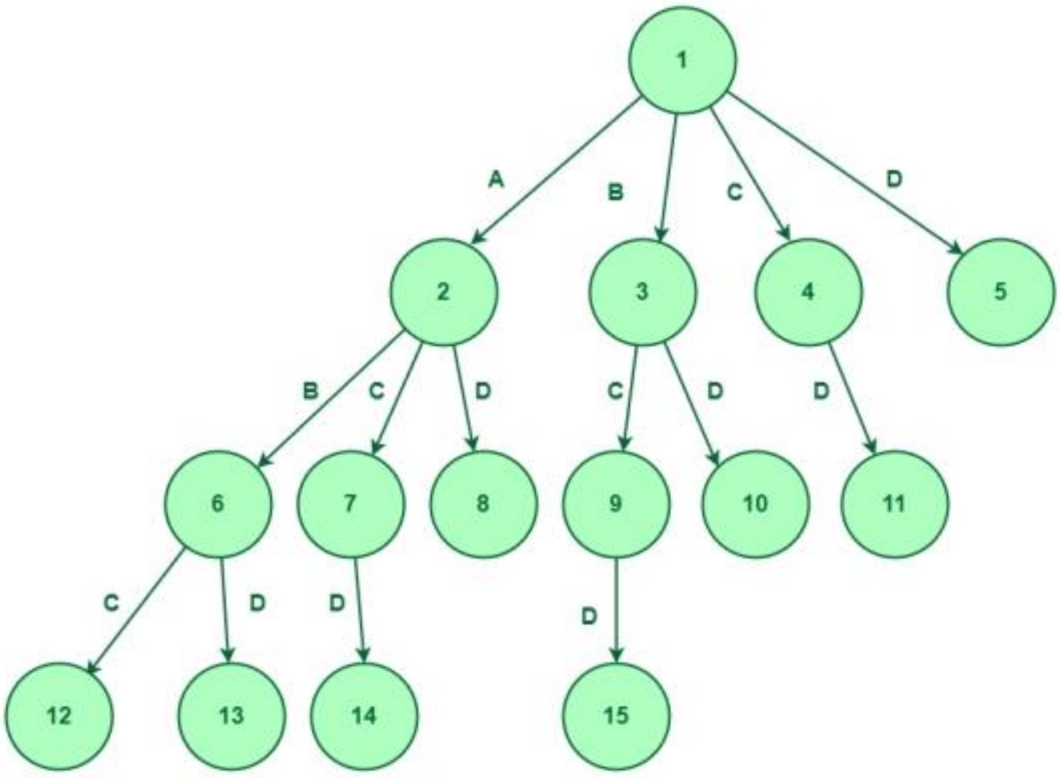
Now, we will expand node 7 as we have considered elements A and C and not consider element B, so, we have only one option to explore which is element D. Let's create node 14 for D.



Expand node 7

Till node 8, we have considered elements A and D, and not selected elements B and C, So, We have no more elements to explore, Therefore on node 8, there won’t be any expansion.

Now, we will expand node 9 as we have considered elements B and C and not considered element A, so, we have only one option to explore which is element D. Let’s create node 15 for D.

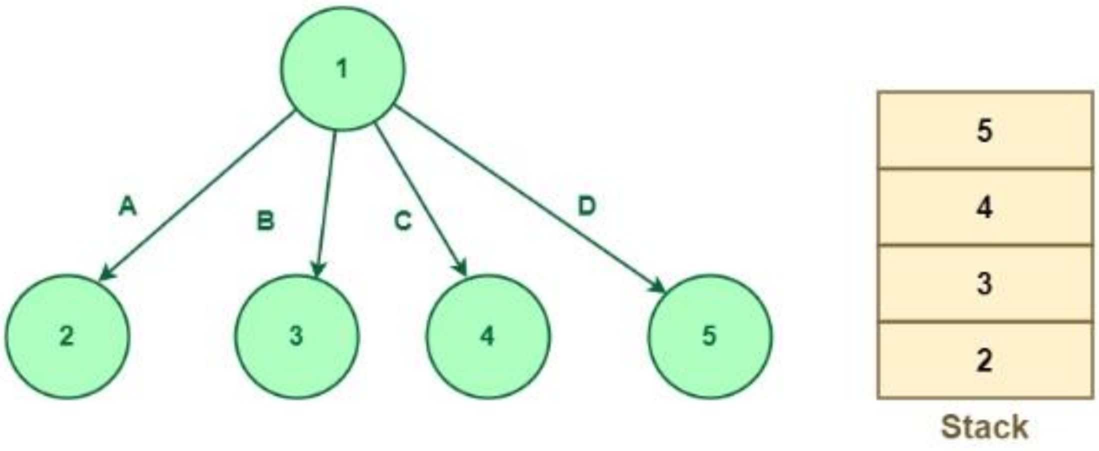


Expand node 9

2. LIFO Branch and Bound

The Last-In-First-Out approach for this problem uses stack in creating the state space tree. When nodes are added to a state space tree, they are added to a stack. After all nodes of a level have been added, we pop the topmost element from the stack and explore it.

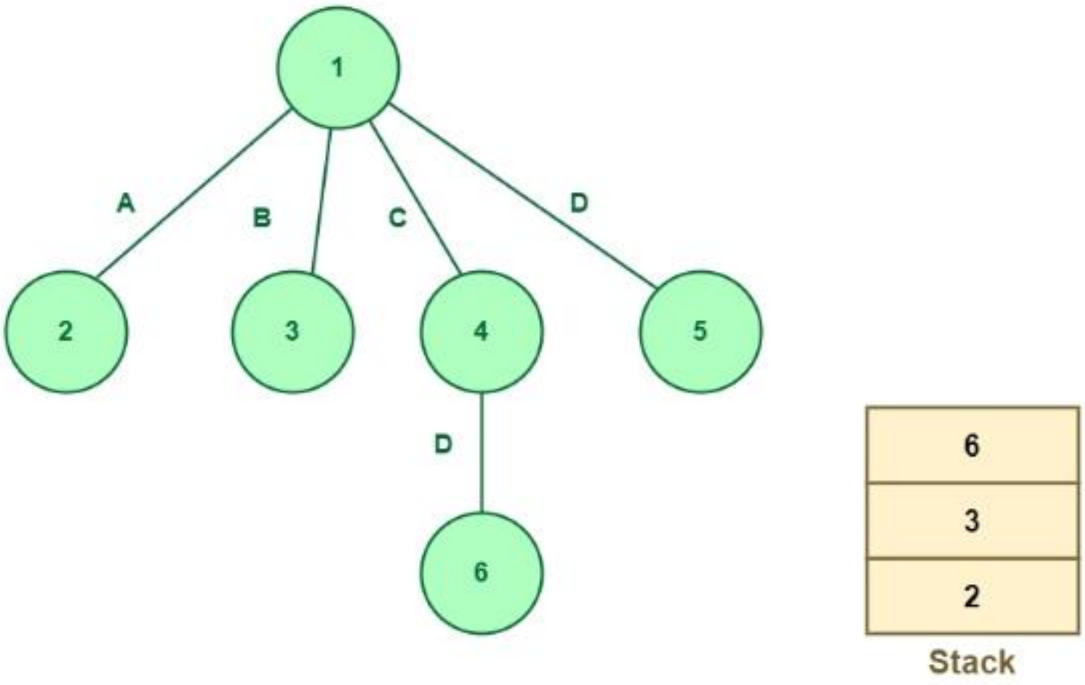
For a given set {A, B, C, D}, the state space tree will be constructed as follows :



State space tree for element {A, B, C, D}

Now the expansion would be based on the node that appears on the top of the stack. Since node 5 appears on the top of the stack, so we will expand node 5. We will pop out node 5 from the stack. Since node 5 is in the last element, i.e., D so there is no further scope for expansion.

The next node that appears on the top of the stack is node 4. Pop-out node 4 and expand. On expansion, element D will be considered and node 6 will be added to the stack shown below:

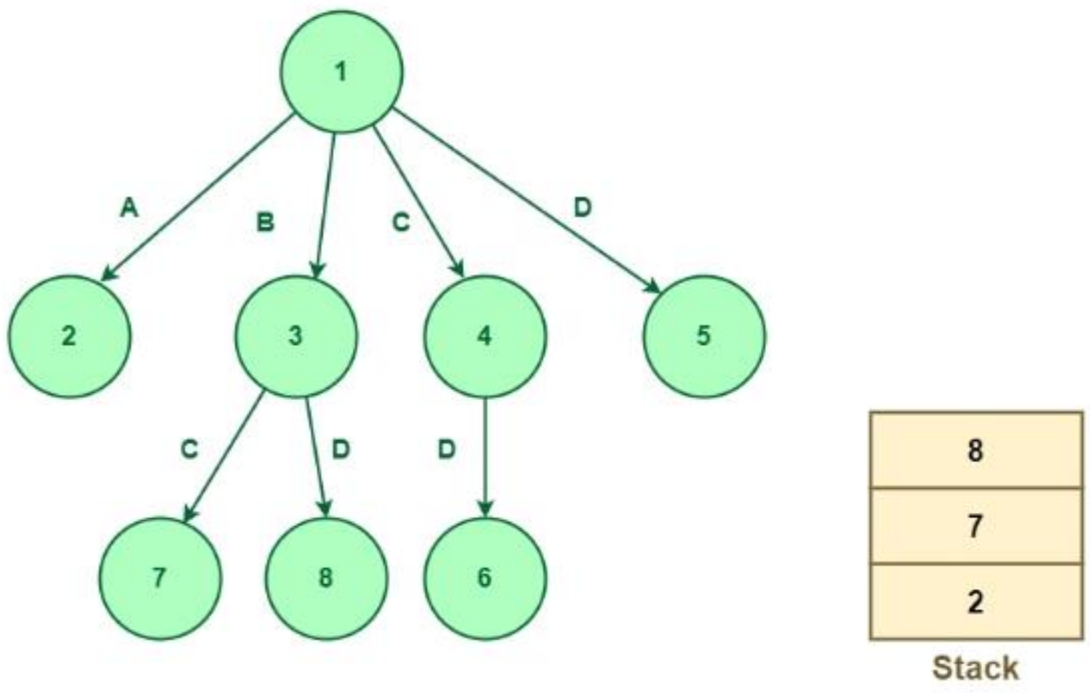


Expand node 4

The next node is 6 which is to be expanded. Pop-out node 6 and expand. Since node 6 is in the last element, i.e., D so there is no further scope for expansion.

The next node to be expanded is node 3. Since node 3 works on element B so node 3 will be expanded to two nodes, i.e., 7 and 8 working on elements C and D respectively. Nodes 7 and 8 will be pushed into the stack.

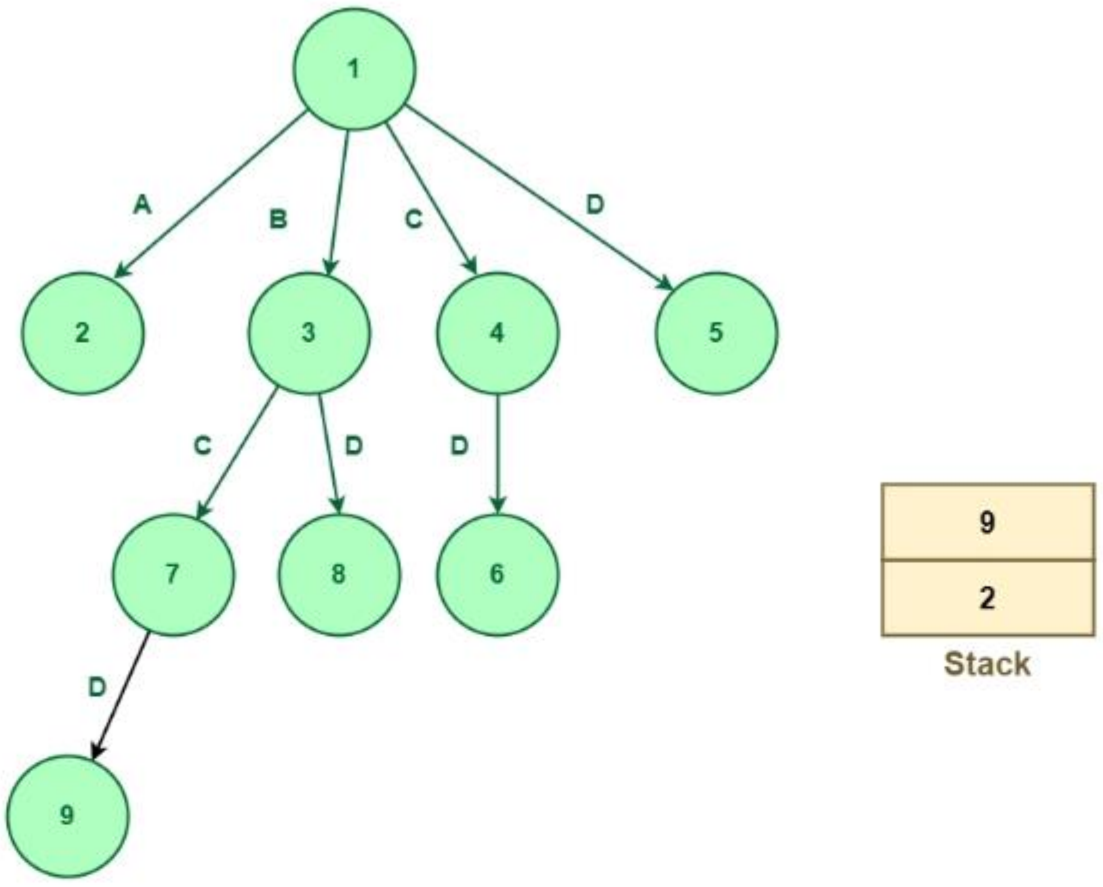
The next node that appears on the top of the stack is node 8. Pop-out node 8 and expand. Since node 8 works on element D so there is no further scope for the expansion.



Expand node 3

The next node that appears on the top of the stack is node 7. Pop-out node 7 and expand. Since node 7 works on element C so node 7 will be further expanded to node 9 which works on element D and node 9 will be pushed into the stack.

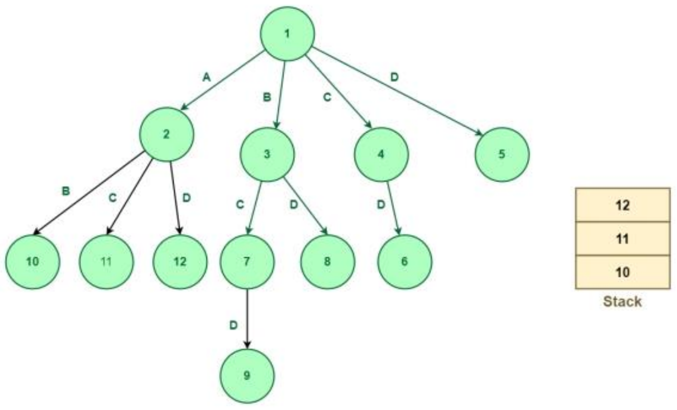
The next node is 6 which is to be expanded. Pop-out node 6 and expand. Since node 6 is in the last element, i.e., D so there is no further scope for expansion.



Expand node 7

The next node that appears on the top of the stack is node 9. Since node 9 works on element D, there is no further scope for expansion.

The next node that appears on the top of the stack is node 2. Since node 2 works on the element A so it means that node 2 can be further expanded. It can be expanded up to three nodes named 10, 11, 12 working on elements B, C, and D respectively. There new nodes will be pushed into the stack shown as below:



Expand node 2

In the above method, we explored all the nodes using the stack that follows the LIFO principle.

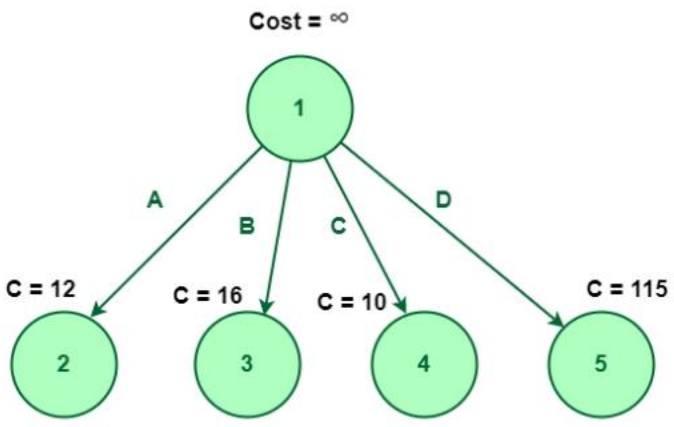
3. Least Cost-Branch and Bound

To explore the state space tree, this method uses the cost function. The previous two methods also calculate the cost function at each node but the cost is not been used for further exploration.

In this technique, nodes are explored based on their costs, the cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.
Now, Consider a node whose cost has been determined. If this value is greater than U0, this node or its children will not be able to give a solution. As a result, we can kill this node and not explore its further branches. As a result, this method prevents us from exploring cases that are not worth it, which makes it more efficient for us.

Let’s first consider node 1 having cost infinity shown below:

In the following diagram, node 1 is expanded into four nodes named 2, 3, 4, and 5.

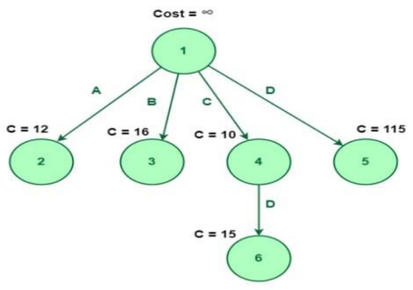


Node 1 is expanded into four nodes named 2, 3, 4, and 5

Assume that cost of the nodes 2, 3, 4, and 5 are 12, 16, 10, and 315 respectively. In this method, we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 4. So, we will explore node 4 having a cost of 10.

During exploring node 4 which is element C, we can notice that there is only one possible element that remains unexplored which is D (i.e, we already decided not to select elements A, and B). So, it will get expanded to one single element D, let’s say this node number is 6.

So, it will get expanded to one single element D, let’s say this node number is 6.



Exploring node 4 which is element C

Now, Node 6 has no element left to explore. So, there is no further scope for expansion. Hence the element {C, D} is the optimal way to choose for the least cost.

REFERENCES:

References

1. M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions,* Dover, New York, 1972.
2. A. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms,* Addison-Wesley, Reading, MA, 1975.
3. B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, and S. Watt. *Maple VLibrary Reference Manual,* Springer-Verlag, New York, 1991. Also *Maple User Manual,* Maplesoft, Waterloo, Ontario, 2012.