DSA SELF PLACED COURSE BY GEEKSFORGEEKS

In partial fulfilment of the requirements for the degree of Bachelor of Technology

(Computer Science and Engineering)

Submitted by

Chalithar Anju Kumari

(12302101)

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

CERTIFICATE

GeeksforGeeks

CERTIFICATE
OF COURSE COMPLETION

THIS IS TO CERTIFY THAT

Chalithar Anju Kumari

has successfully completed a 16-week course on Data Structures and Algorithms - Self Paced [Online Course].

Sandeep Jain

Mr. Sandeep Jain
Founder & CEO, GeeksforGeeks
https://media.geeksforgeeks.org/courses/certificates/c7d9b8c288cacd7f7d1159f704e79a0c.pdf

www.geeksforgeeks.org

STUDENT DECLARATION

To whom so ever it may concern

I, Chalithar Anju Kumari (12302101), declare that the work done by me on "DSA" self-paced course by "geeksforgeeks" is a record of original work done as a partial fulfilment of the requirements for the award of the degree, Bachelor of technology.

ACKNOWLEDGEMENT

TABLE OF CONTENTS

INTRODUCTION

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory. To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types.

DATA STRUCTURES

Data Structures are a specialized means of organizing and storing data in computers in such a way that we can perform operations on the stored data more efficiently. Data structures have a wide and diverse scope of usage across the fields of Computer Science and Software Engineering.

ALGORITHMS

The word Algorithm means" A set of rules to be followed in calculations or other problem-solving operations" Or" A procedure for solving a mathematical problem in a finite number of steps that frequently by recursive operations ".

TECHNOLOGY LEARNT

Data structures and algorithms from basic to advanced level.

8 weeks long course to master the basics of DSA and practice coding questions and attempt assessment tests It comes with more than 200+ questions and contests helped learn all the concepts of Data structures and algorithms such as Arrays, Linked List, Trees, queues, Hashing, Searching, Matrix, Binary Search Tree, Heap, Graph, Greedy, Backtracking, Dynamic Programming, Asymptotic Notations and concepts of recursions and bit magic.

What is meant by Algorithm Analysis?

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?

To predict the behaviour of an algorithm without implementing it on a specific computer.

It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.

It is impossible to predict the exact behaviour of an algorithm. There are too many influencing factors.

The analysis is thus only an approximation; it is not perfect.

More importantly, by analysing different algorithms, we can compare them to determine the best one for our purpose.

1) Asymptotic Notations-:

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

- **O Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour.

  A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. $3n^3 + 6n^2 + 6000 = O(n^3)$ Dropping lower order terms is always fine because there will always be a no after which $O(n^3)$ has higher values than $On^2)$ irrespective of the constants involved. For a given function g(n), we denote O(g(n)) is following set of functions.

- **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
  If we use O notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

  1) The worst-case time complexity of Insertion Sort is O(n^2).
  2) The best-case time complexity of Insertion Sort is O(n).

- **Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three.

Worst, Average and Best-Case Time Complexities

It is important to analyse an algorithm after writing it to find its efficiency in terms of time and space in order to improve it if possible.

When it comes to analysing algorithms, the asymptotic analysis seems to be the best way possible to do so. This is because asymptotic analysis analyses algorithms in terms of the input size. It checks how are the time and space growing in terms of the input size.
We can have three cases to analyse an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Arrays -:

An array is a collection of items of the same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Generally, arrays are declared as: datatype array Name [array Size]; An array is distinguished from a normal variable by brackets [and].

Advantages of using arrays:

- Arrays allow random access of elements. This makes accessing elements by their position faster.
- Arrays have better cache locality that can make a pretty big difference in performance.

Arrays 4 Basic Operations

1) Traversal
2) Insertion
3) Deletion
4) Search

Traversal:

Visiting every element of an array once is known as traversing the array.

Insertion:

An element can be inserted in an array at a specific position. For this operation to succeed, the array must have enough capacity. Suppose we want to add an element 10 at index 2 in the below-illustrated array, then the elements after index 1 must get shifted to their adjacent right to make way for a new element.

Deletion:

An element at a specified position can be deleted, creating a void that needs to be fixed by shifting all the elements to their adjacent left. We can also bring the last element of the array to fill the void if the relative ordering is not important

Searching:

Searching can be done by traversing the array until the element to be searched is found O(n)

- **Linear Search**
- **Binary Search**

Linear Search

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

Approach for Linear or Sequential Search

- Start with index 0 and compare each element with the target
- If the target is found to be equal to the element, return its index
- If the target is not found, return -1

Binary Search

This type of searching algorithm is used to find the position of a specific value contained **in a sorted array**. The binary search algorithm works on the principle of divide and conquer and it is considered the best searching algorithm because it's faster to run.

Approach for Binary Search

- Compare the target element with the middle element of the array.

- If the target element is greater than the middle element, then the search continues in the right half.

- Else if the target element is less than the middle value, the search continues in the left half.

- This process is repeated until the middle element is equal to the target element, or the target element is not in the array

- If the target element is found, its index is returned, else -1 is returned.

Sorting

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

- Insertion Sort

- Selection Sort

- Bubble Sort

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.
The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.

- Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
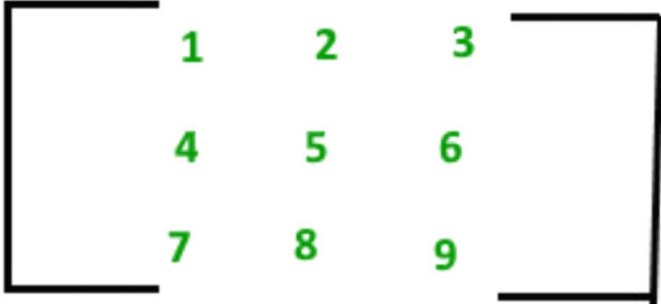
Bubble Sort

In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Matrix

A matrix represents a collection of numbers arranged in order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

A matrix with 9 elements is shown below:

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

The above Matrix M has 3 rows and 3 columns. Each element of matrix [M] can be referred to by its row and column number.

Properties of Matrix addition and multiplication:

1) A+B = B+A (Commutative)
2) (A+B) +C = A+ (B+C) (Associative)

3) AB ≠ BA (Not Commutative)
4) (AB) C = A (BC) (Associative)
5) A (B+C) = AB+AC (Distributive)

Terminologies

- **Square Matrix:** A square Matrix has as many rows as it has columns. i.e., no of rows = no of columns.
- **Symmetric matrix:** A square matrix is said to be symmetric if the transpose of original matrix is equal to its original matrix. i.e. $(A^T) = A$.
- **Skew-symmetric:** A skew-symmetric (or antisymmetric or antimetric [1]) matrix is a square matrix whose transpose equals its negativities. $(A^T) = -A$.

- **Diagonal Matrix:** A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. The term usually refers to square matrices.
- **Identity Matrix:** A square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. Identity matrix is denoted as I.

- **Orthogonal Matrix:** A matrix is said to be orthogonal if $AA^T = A^TA = I$.
- **Idempotent Matrix:** A matrix is said to be idempotent if $A^2 = A$.
- **Involuntary Matrix:** A matrix is said to be Involuntary if $A^2 = I$.
- **Singular Matrix**: A square matrix is said to be singular matrix if its determinant is zero i.e., $|A|=0$
- **Non-singular Matrix**: A square matrix is said to be non-singular matrix if its determinant is non-zero.

Matrix Operation

1) Matrices Addition

The addition of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. Here, m and n represent the number of rows and columns in the matrix respectively. The

elements of C's are sum of corresponding elements in A and B which can be shown as:

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 12 & 14 \end{bmatrix}$$

1) Matrices Subtraction

The subtraction of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. Here, m and n represent the number of rows and columns in the matrix respectively. The subtraction of two matrices A_{m*n} and B_{m*n} gives a matrix C_{m*n}. Here, m and n represent the number of rows and columns in the matrix respectively.

The elements of C's are difference of corresponding elements in A and B which can be represented as:

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

Matrices Multiplication

The multiplication of two matrices A_{m*n} and B_{n*p} gives a matrix C_{m*p}. It means number of columns in A must be equal to number of rows in B to calculate C=A*B. To calculate element c_{11}, multiply elements of 1st row of A with 1st column of B and add them $(5*1+6*4)$ which can be shown as:

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 29 & 40 \\ 44 & 61 \end{bmatrix}$$

Hashing

It is a method of storing and retrieving data from a database efficiently.

Suppose that we want to design a system for storing employee records keyed using phone numbers. And we want the following queries to be performed efficiently:

1. Insert a phone number and the corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and the related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. An array of phone numbers and records.
2. A linked list of phone numbers and records.
3. A balanced binary search tree with phone numbers as keys.
4. A direct access table.

Hash Function:

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as an index in the hash table.
A good hash function should have following properties:

1. It should be efficiently computable.
2. It should uniformly distribute the keys (Each table position be equally likely for each key).

For example, for phone numbers, a bad hash function is to take the first three digits. A better function will consider the last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function Equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:** The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine the table slots until the desired element is found or it is clear that the element is not present in the table.
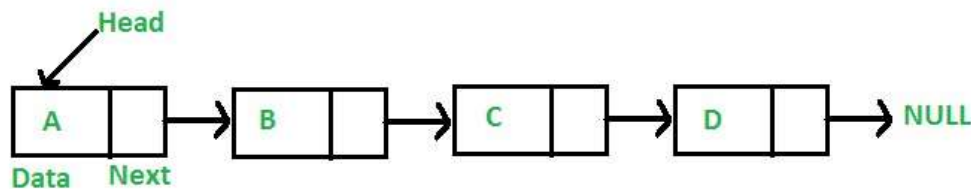
LINKED LIST

Linked Lists are linear or sequential data structures in which elements are stored at non-contiguous memory locations and are linked to each other using pointers.

Like arrays, linked lists are also linear data structures but in linked lists elements are not stored at contiguous memory locations. They can be stored anywhere in the memory but for sequential access, the nodes are linked to each other using pointers.

Each element in a linked list consists of two parts:

- **Data:** This part stores the data value of the node. That is the information to be stored at the current node.
- **Next Pointer:** This is the pointer variable or any other variable which stores the address of the next node in the memory.

Advantages of Linked Lists over Arrays:

Arrays can be used to store linear data of similar types, but arrays have the following limitations:
 The size of the arrays is fixed, so we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage. On the other hand, linked lists are dynamic and the size of the linked list can be incremented or decremented during runtime.

1. Inserting a new element in an array of elements is expensive, because a room has to be created for the new elements, and to create room, existing elements have to shift.

Disadvantages of Linked Lists:

- Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So, we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.

- Extra memory space for a pointer is required with each element of the list.

- Not cache friendly. Since array elements are present at contiguous locations, there is a locality of reference which is not there in the case of linked lists

STACK

The **Stack** is a linear data structure, which 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).

- The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order, the insertion takes place at the rear end of the stack and deletion occurs at the rear of the stack.

- The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

Mainly, the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they were pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.
- **is Empty:** Returns true if the stack is empty, else false.
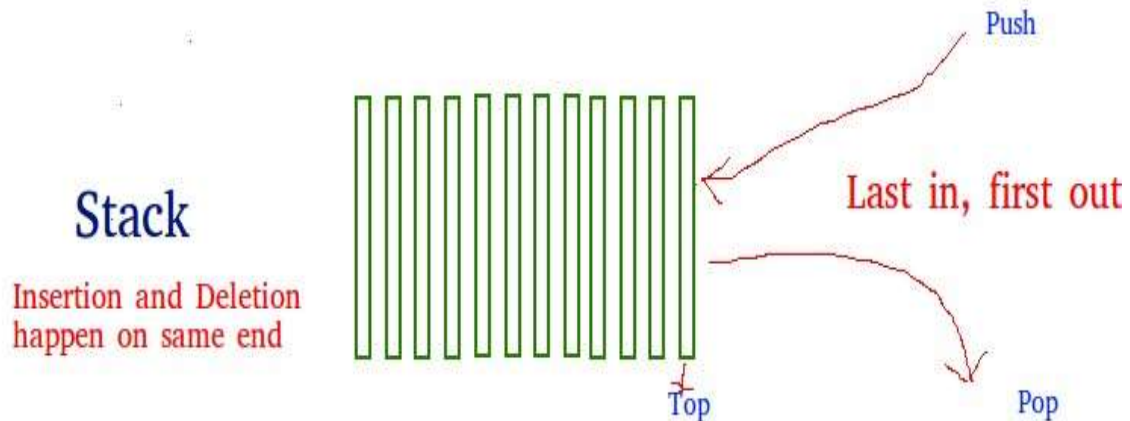
How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate that is at the top is the first one to be removed, i.e., the plate that 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/FILO order.

Time Complexities of operations on stack: The operations push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Implementation: There are two ways to implement a stack.

- Using array
- Using linked list

Queue

Like *Stack* data structure, *Queue* is also a linear data structure that follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO), which means that the element that is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer who came first is served first.

The difference between stacks and queues is in removing. In a stack, we remove the most recently added item; whereas, in a queue, we remove the least recently added item.
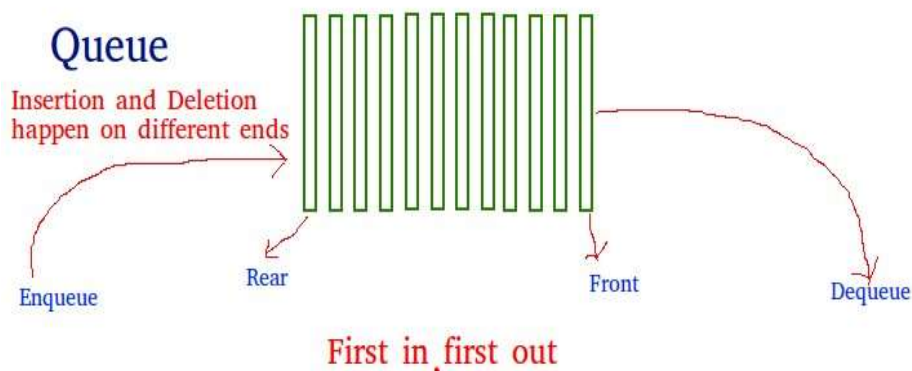
Operations on Queue: Mainly the following four basic operations are performed on queue:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition

- **Front:** Get the front item from queue.

- **Rear:** Get the last item from queue.

TREE

A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.

A Sample tree is as shown below:
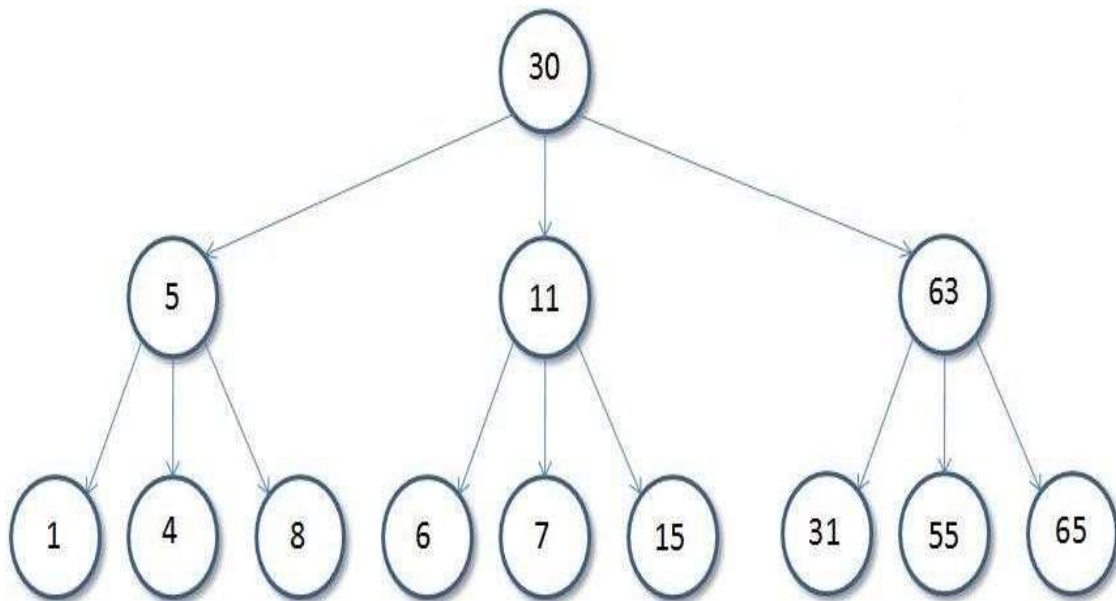
Basic Tree Terminologies:

- **Root**: The root of a tree is the first node of the tree. In the above image, the root node is the node 30.
- **Edge**: An edge is a link connecting any two nodes in the tree. For example, in the above image there is an edge between node **11** and **6.**
- **Siblings**: The children nodes of same parent are called siblings. That is, the nodes with same parent are called siblings. In the above tree, nodes 5, *11, and 63* are siblings.
- **Leaf Node**: A node is said to be the leaf node if it has no children. In the above tree, node **15** is one of the leaf nodes.
- **Height of a Tree**: Height of a tree is defined as the total number of levels in the tree or the length of the path from the root node to the node present at the last level. The above tree is of height **2.**

Below is a sample Binary Tree:
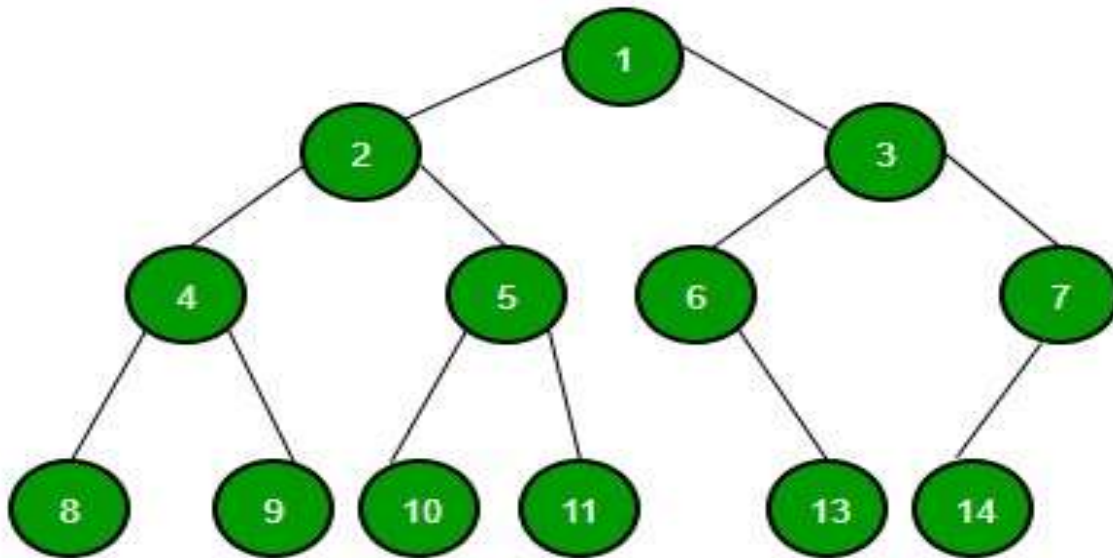
Types of Binary Trees: Based on the structure and number of parents and children nodes, a Binary Tree is classified into the following common types:

- **Full Binary Tree:** A Binary Tree is full if every node has either 0 or 2 children. The following are examples of a

full binary tree. We can also say that a full binary tree is a binary tree in which all nodes except leave nodes have two children.

- **Complete Binary Tree:** A Binary Tree is a complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

- **Perfect Binary Tree:** A Binary tree is a Perfect Binary Tree when all internal nodes have two children and all the leave nodes are at the same level.

Properties of a Binary Tree:

·

·   The maximum number of nodes at level 'l' of a binary tree is (2l - 1). Level of root is 1. This can be proved by induction. For root, l = 1, number of nodes = 21-1 = 1 Assume that the maximum number of nodes on level l is 2l-1. Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 * 2l-1.

·   Maximum number of nodes in a binary tree of height 'h' is (2h – 1). Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1. This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2h-1. This is a simple geometric series with h terms and sum of this series is 2h – 1. In some books, the height of the root is considered as 0. In that convention, the above formula becomes 2h+1 – 1.

- In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is Log2(N+1). This can be directly derived from point 2 above. If we consider the convention where the height of a leaf node is considered 0, then above formula for minimum possible height becomes Log2(N+1) – 1.

- A Binary Tree with L leaves has at least (Log2L + 1) levels. A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

HEAP

A Heap is a Tree-based data structure, which satisfies the below properties:

1. A Heap is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

2. A Heap is either Min Heap or Max Heap. In a Min-Heap, the key at root must be minimum among all keys present in the Binary Heap. The same property must be recursively true for all nodes in the Tree. Max Heap is similar to MinHeap.
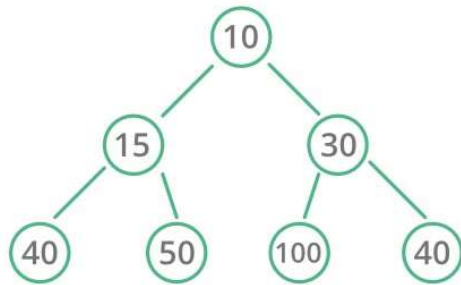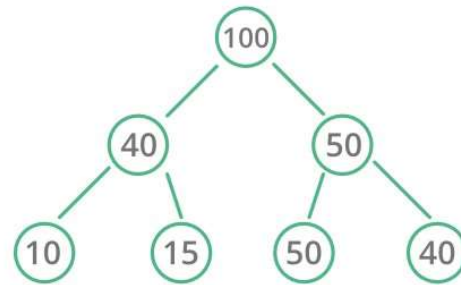
Binary Heap:

A Binary Heap is a heap where each node can have at most two children. In other words, a Binary Heap is a complete Binary Tree satisfying the above-mentioned properties.

Heap Data Structure

Min Heap

Max Heap

GRAPH

A *Graph* is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.

2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example, Google GPS

- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.
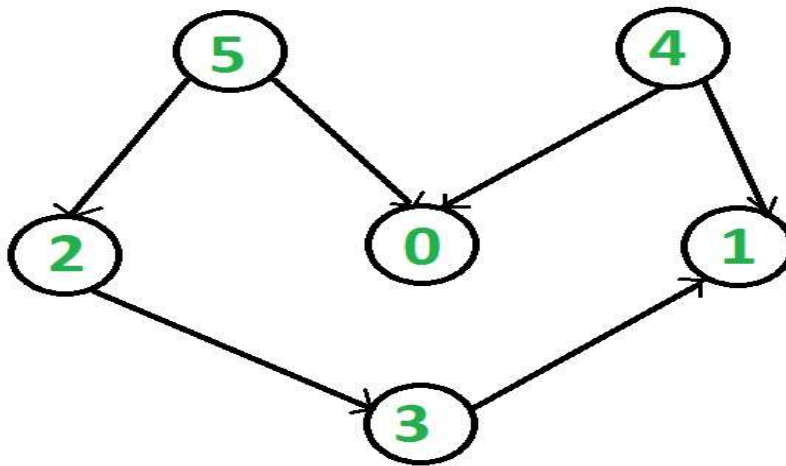
Directed and Undirected Graphs

- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.

  For Example, the below graph is a directed graph:
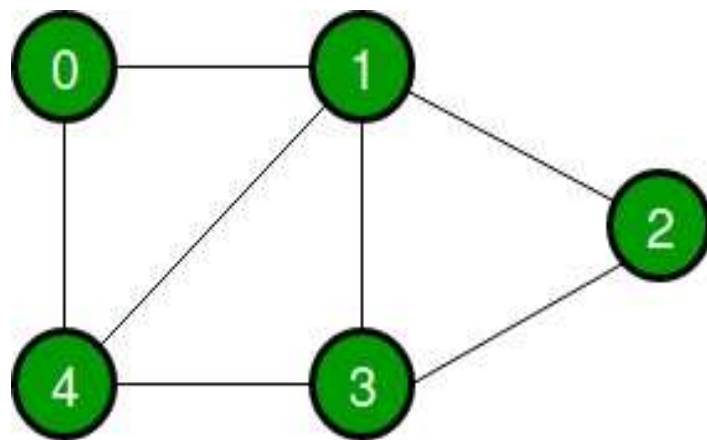
- .
- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u.**

  Following is an example of an undirected graph with 5 vertices:

Representing Graphs

Following two are the most commonly used representations of a graph:

Adjacency Matrix:

The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency List:

Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.

BACKTRACKING

A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach. This approach is used to solve problems that have multiple solutions

Backtracking Algorithm

1)To find all Hamiltonian Paths present in a graph

2)To solve the N Queen problem

3) Maze solving problem

4) The Knight's tour problem.

DYNAMIC PROGRAMMING

Dynamic Programming is an algorithmic approach to solve some complex problems easily and save time and number of comparisons by storing the results of past computations. The basic idea of dynamic programming is to store the results of previous calculation and reuse it in future instead of recalculating them.

We can also see Dynamic Programming as dividing a particular problem into subproblems and then storing the result of these subproblems to calculate the result of the actual problem.

How Dynamic Programming Works

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them. This technique of storing the value of subproblems is called memorization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

REASON TO CHOOSE DSA

My reason for choosing data structures and algorithms (DSA) is that programming is about DSA and it is the building blocks of the software development process. It enables you to learn to write efficient codes. It helps you to gain knowledge in order to solve complex problems a thorough knowledge of data structures and algorithms helps you to optimize and solve complex problems with ease and now a days data structures and algorithm is base of every software job interviews every company interview you face question related to data structures and algorithms are asked which makes it the most demanding technology to learn .

Project

Implementation of Tic-Tac-Toe game

Rules of the Game

- The game is to be played between two people (in this program between HUMAN and COMPUTER).

- One of the players chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').

- If no one wins, then the game is said to be draw

O	X	O
O	X	X
X	O	X

Initialize the Game Board: Start by initializing the Tic-Tac-Toe board, usually represented as a 3x3 grid. You can use a 2D array or any suitable data structure for this purpose.

Display the Board: Create a function to display the current state of the board so that players can see the game's progress.

Get Player Input: Alternately, prompt each player (X and O) for their move. They should specify the row and column where they want to place their symbol (X or O).

Check for Validity: Ensure that the selected move is valid by checking if the chosen cell is empty. If the cell is already occupied, ask the player to choose another cell.

Update the Board: If the move is valid, update the board with the player's symbol in the selected cell**.**

Check for a Win or Draw: After each move, check if the current player has won the game or if the game has ended in a draw. This involves checking rows, columns, and diagonals for three consecutive symbols**.**

Switch Players: Alternate between players for each turn.

Repeat: Continue steps 3-7 until the game ends (either a player wins, or the game is a draw).

MINIMAX ALGORITHM

The algorithm associated with the tic tac toe games is minimax algorithm it is a kind of backtracking algorithm that is used in decision making and game theory in order to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games such as tic-tac-toe, chess etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.

LEARNING OUTCOME

- Solve complex programming problems using advance algorithms techniques.

- Knowledge about various data structures and its uses

- algorithmic techniques for solving various problems with full flexibility of time.

- Understanding of dynamic programming and its related applications

<u>**BIBLIOGRAPHY**</u>

- https://www.programiz.com/dsa/dynamic-programming

- https://www.geeksforgeeks.org/courses/dsa-self-paced

<u>Thank You</u>