**Data Structures and Algorithm**

**Physical Data Structures**:

Array

LinkedList

**Logical Data Structures**:

Graph

Stack

Queue

Tree

**Recursion:**

Uses stack for storing functions which are called.

For every recursion to execute properly there must be a base case defined properly which checks for the for initial conditions and checks when the recursion is needed to be broken.

**Recursion vs Iteration:**

Iteration has upper hand over recursion in space management

Iteration is time efficient than recursion.

**When to use recursion?**

Divide n Conquer

Sorting- quick sort, merge sort

Tree- traversal, searching, sorting, Deletion

Stack

Dynamic Programming

**Notations:**

O(n) – Worst time of an algorithm

Ω(n) - best time of an algorithm

ɵ(n) - Average time taken by the algorithm

**Time Complexity of a recursive algorithm:**

**FindBiggest(A,n):** // n is the size of the array …..T(n)

Static highest = Integer.Min; // O(1)

If (n== -1): // O(1)

Return highest; //O(1)

Else if (A[n]>highest) //O(1)

Highest=A[n]; //O(1)

Return FindBiggest(A,n-1); // T(n-1)

T(n) = T(n-1) +O (1)

BASE condition: T (-1) = O (1)

T (n-1) = T ((n-1)-1) + O (1) …….2

T(n-2) = T ((n-2)-1) + O(1) ………(3)

Now replacing T(n-1) with equations (2) and (3)…..

We get,

T(n) = 3 + T (n-3)

Following the pattern, we can say that,

T(n) = K + T(n-K)

Now putting k=n+1 we get,

T(n) = n+1 +T(-1);

Therefore, the time complexity of recursive function is o(n).

**Recursive algo #2:**

**Find a num in a sorted array**

**Search (int arr[], int start, int end, int val):…….. T(n)**

If(start==end): ………O(1)

If(arr[start]==val): ……………….O(1)

Return start. ……………………” ”

Else:

Print (“ Not available’)…………….” ”

Else:

Int mid = end+start/2; ………………O(1)

If(arr[mid]==val) ………………O(1)

Return mid;

If(arr[mid]> val) ………………O(1)

Search(arr, val, start, mid); ……………………….**T(n/2)**

Else if(arr[mid]<val) ………………….O(1)

Search(arr, val,mid, end); ………………………….**T(n/2)**

**Time Complexity:**

T(n) = O(1) + T(n/2); ………………(1)

Base condition: T(1) = 1

T(n/2) = T(n/4)+O(1) ………………….(2)

T(n/4) = T(n/8) +O(1) ………………….(3)

Therefore, using 2 and 3 in 1:

T(n) = T(n/8) + 3

T(n) = T(n/2^k) + k

Putting k= log n/2;

T(n)= T(1) + log (n)

Time complexity of the algorithm is O(log n).

**Time complexity** of recursive functions can be found using **back substitution** or **master’s theorem.**

**Array**

1D array: has only cloumns

Ex: arr[i]

2D array: has rows and coulmns

Ex: arr[row][column]

3D array has depth, rows and columns

Ex: arr[depth][row][column]

**Array representation in memory:**

1-D array:

For 1D array the compiler allocates 10 contiguous cells of memory into the ram randomly.

2-D array:

For 2D array with m rows and n columns, in the memory this will be stored as a flat 1D array according to rows, like first rows all column are saved together in a line and then it goes to the nest row’s elements.

3-D array:

For 3-D array it will randomly start allocating memory address and stored first depth wise, with all 2d array elements present at depth zero are stored first and similarly stored otherwise.

When to use array?

**Use:** for storing multiple similar type of data, for random access,

**Avoid:** for storing non similar type of data, when we don’t know the size of data to be stored in advance

Practical use:

* Dynamic Programming
* Hashing and hash tables

**Linked List:**

Linear data structure where each element is a separate object with each object having data and address of the next object.

**Types of linked list**:

Singly linked list

Singly Circular linked list ( last node points to the first node)

Doubly linked list

Doubly circular linked list (last node points to first node and first node points to the last node)

**Components of a linked lists:**

**Node**: contains data and reference to next node

**Head**: reference to first node in the list

**Tail**: reference to last node list (generally for time efficiency)

**Need of linked Lists**:

Single linked list: Remove of add data to the run time.

**Storing of linked lists in memory:**

We cannot access the nodes of a linked list directly. The first node which is pointed by the head node is stored randomly in the memory. And the next node is stored somewhere randomly in the memory.

**Circular Linked List:**

We must simultaneously update he head and tail node for a circular linked list.

**Time Complexity:**

Creation – O(1); space complexity- O(1)

Insertion – O(n) ; space complexity -O(1)

Traverse – O(n) ; space complexity – O(1)

Searching – O(n) ; space complexity – O(1)

Deletion – O(n) ; space complexity – O(1)

Entire deletion of list – O(1); space complexity – O(1)

**Double Linked List:**

Each Node in a double linked list has three components: next Node pointer pointing to the next node of the list, prev Node pointer pointing to the previous node in the list and data variable to store the data given to the node.

**Time complexity:**

Creation – O(1); space complexity- O(1)

Insertion – O(n) ; space complexity -O(1)

Traverse – O(n) ; space complexity – O(1)

Searching – O(n) ; space complexity – O(1)

Deletion – O(n) ; space complexity – O(1)

Entire deletion of list – O(n) // traversing each node and updating the next and prev node as null; space complexity – O(1)

**Circular Double Linked List:**

Circular double linked list is double linked list in which the head’s previous points to tail and tail’s next reference points to the head.

**Time Complexity:**

Creation – O(1); space complexity- O(1)

Insertion – O(n) ; space complexity -O(1)

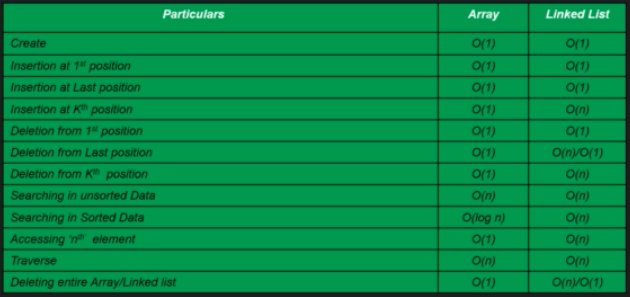
Traverse – O(n) ; space complexity – O(1)

Searching – O(n) ; space complexity – O(1)

Deletion – O(n) ; space complexity – O(1)

Entire deletion of list – O(1); space complexity – O(1)

SL vs SDLL vs DLL vs CDLL:



**Practical uses of Linked list:**

Winows os tab switching – circular linked list.

Windows photo viewer – circular double linked list

**Stack:**

Stack is a logical data structure where all elements are stacked up together one above the another. A stack generally follows LIFO( Last in First Out) method where last element is taken out first.

Ex: previous button in browser uses stack data structure to store previously visited website.

**Major Operations in stack data structure:**

Push- insert an element

Pop- remove the top element

Peek- shows the top element of stack

isEmpty – checks if stack is empty.

**Stack can be implemented using array and linked list.**

push – O(1)

pop - O(1)

peek – O(1)

isEmpty – O(1)

**Application of Stack:**

* Cannot be easily corrupted;
* Helps to manage data in LIFO way

Avoid:

* Random access is not possible, rectifying of element in stack is costly

**Queue Data-structure:**

New addition of member happens at the end of the queue.

First element in queue is removed from queue first

Follows FIFO method (First In First Out)

**Application:** Billing counter

Queue can be implemented using array (Linear queue, circular queue) and LinkedList.

**Operations in a queue:**

Creation of queue - O(1); space complexity: O(n) // creation of array of size n

Enqueue – O(1); space complexity: O(1)

Dequeue – O(1); space complexity: O(1)

peekIn – O(1) ; Space complexity: O(1)

isEmpty() – O(1); space complexity: O(1)

isFull() (optional, only in array) –

delete() -

**Circular Queue (array):**

In linear queue after dequeuing some of the space remain empty and is not used, circular queue helps in utilizing those array spaces.

Creation of circular queue is similar to linear queue.

**Working:**

For example there is a blank array, and then we enqueue few elements into the array and now we try to remove( dequeue) some elements from the array, as we do that the top starts pointing to the other elements gradually. Even if the array is empty in the beginning, we cannot add more elements to the linear array as last space of the array is already filled.

**Array vs linked List**:

Linked list implementation of queue is more space efficient than array implementation of queue.

**Binary Tree:**

A binary tree represents data in a hierarchical form. It has maximum of two children.

Every node has two components: data and reference to its sub-categories.

Every tree has one root node and other smaller subtrees: left subtree and right subtree.

**Root:** A Node with no parents

**Edge:** link from parent to child

**Leaf:** node with no children

**Sibling:** nodes with same parent node

**Depth:** the length of path from root to that node

**Height:** length of the path from that node to the deepest node.

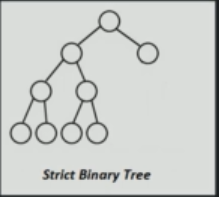
**Predecessor:** immediate previous neighbor of the node.

**Successor:** immediate next neighbor of the node.

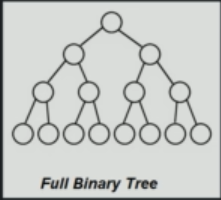
**Application:** Huffman coding, Heap, expression parsing.

**Types:**

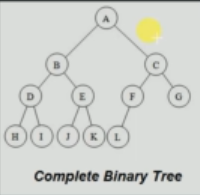
**Strict:**  Has either 2 or no children nodes.



**Full:** Each non leaf node has two children and each leaf node are in same level.



**Complete:** All levels are filled except the last level and all keys in the last level are as left as possible.



**Representation of Tree:**

* Through linked list

Each node has three elements, data, left pointer, and right pointer.

* Through array

In array the first element 1 stored in 1st index and left child is stored in index [2i] and right child is stored in index [2i+1].

**Binary tree (Linked List)**

Creation: O (1); space complexity: O (1)

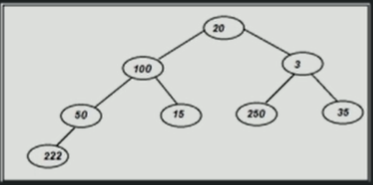
Insertion: O(n); space complexity: O(n)

Traversal: O(n)

Search: O(n); space complexity: O(n)

Deletion: O(n); space complexity: O(n)

**Traversal in a binary tree:**



**Preorder traversal: O(n); space complexity: O(n)**

Root -> left -> right

Here: 20,100,50,222,15,3,250,35

**In-order traversal: O(n); space complexity: O(n)**

Left ->Root -> right

Here: 222, 50,100,15,20,250,3,35

**Post-order traversal: O(n); space complexity: O(n)**

Left -> right -> root

Here: 222,50,15,100,250,35,3,20

**Level order traversal: O(n); Space complexity: O(n) since we are using queue to insert nodes**

Here: 20, 100,3,50,15,250,35,222

We will use queue.

**Algorithm:**

Pass the root Node as a argument to the level order traversal function.

Create a blank queue and pass the root node

Check if the queue is empty, if not add its children into the queue

Dequeue the elements

**Note:** Queue is faster than system stack

**Deletion:**

For deletion of any value in the tree, we swap the deleting value with the deepest node value and then delete the deepest node.

**Binary Tree (Array implementation):**

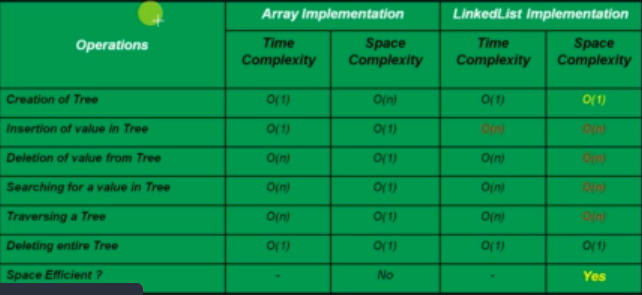
**Creation:** O(1); space complexity: O(n)

**Insertion:** O(1); space complexity: O(1)

**Search**: O(n) ; Space complexity: O(1)

**Traversal:** O(n) ; Space complexity: O(1)

**Deletion:** O(n) ; Space complexity: O(1)



**Binary Search Trees:**

A binary search tree is a binary tree in which its left node is smaller than the parent node and its right node is always greater than root node.

**Operations in Binary Search tree:**

Creation: O(1); space: O(1)

Search: O(log n); space: O(log n) // due to the recursive call

Traversal:  **Depth first Search:**

**Preorder traversal: O(n); space complexity: O(n)**

Root -> Left-> right

**In-order traversal: O(n); space complexity: O(n)**

Left ->Root -> right

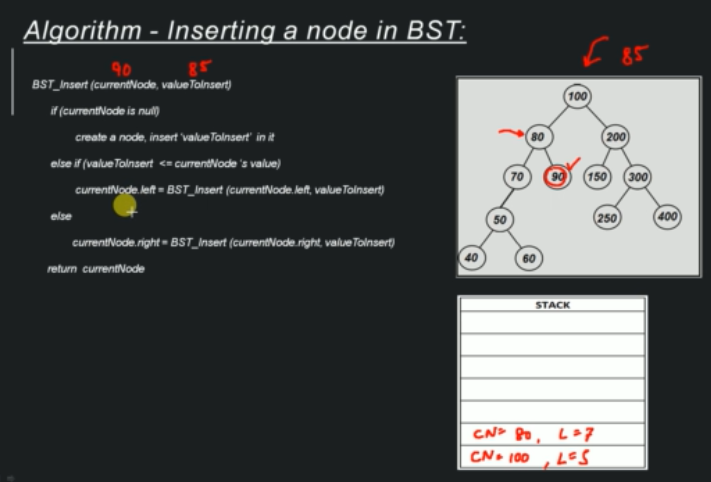
**Post-order traversal: O(n); space complexity: O(n)**

Left -> right -> root

**Level order traversal: O(n); Space complexity: O(n) since we are using queue to insert nodes**

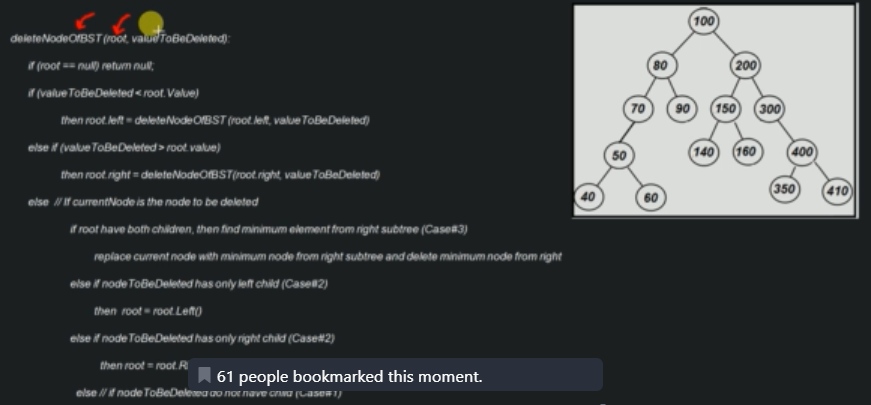
**Inserting a node into the Binary search tree:**

Inserting a node into the binary search tree is done by recursively calling the function and updating the stack with the root node value and line number.



**Time complexity**: O(log n) ; **space complexity**: O(log n)

**Deleting a node:**



**Time complexity**: O (log n); **space complexity**: O(log n)

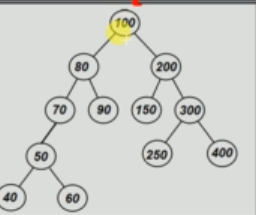
**AVL Tree:**

**Need of AVL tree:**

In AVL tree the height is more balanced and it gets easy to traverse in the tree. It helps in increase the performance of the BST. It helps in maintaining the balance whenever a node is deleted or inserted into the tree. It uses a concept rotation to protect the tree from being skewed.

**What is AVL tree:**

An AVL tree is a balanced Binary search tree where height of the immediate subtrees of any node differs by at most one.



Here height of node 300 is 3 and height of node 200 is 2 and the difference between their height is 1. Hence, the tree is balanced.

Similarly, for subtrees of node 200, height of node 150 is 0 and height of right subtree is 1 and their difference is 1 and here still the BST is balanced.

And here in subtrees of 80 is 70 and 90 and their heights are 2 and 0 which concludes that their difference is not 1 so the BST is not an AVL tree.

TO restore the property of avl tree at any point where height differs more than one then rebalancing is done to restore this property through rotation.

The height of the a subtree node which has no value is considered as -1.

**Operations:**

**Creation**: O(1); space: O(1)

**Search:** O(log n); space: O(log n) // due to the recursive call

Traversal:  **Depth first Search:**

**Preorder traversal: O(n); space complexity: O(n)**

Root -> Left-> right

**In-order traversal: O(n); space complexity: O(n)**

Left ->Root -> right

**Post-order traversal: O(n); space complexity: O(n)**

Left -> right -> root

**Level order traversal: O(n**); **Space complexity**: O(n) // since we are using queue to insert nodes

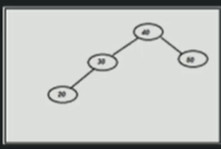
**Insertion in AVL tree:**

Cases:

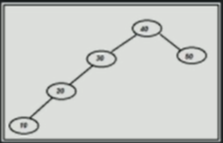
1. when rotation is not required:

Same algorithm as insertion in BST.

1. When rotation is required:
2. Conditions:
3. **Left Left condition (LL)**



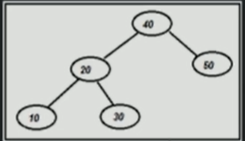
If we want to insert “10” in the BST, so after insertion the tree looks like below:



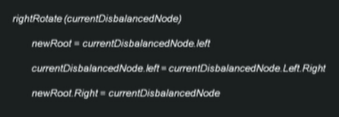
The tree is not balanced

Here as we look at the node which is disbalanced we find that the grandchild 10 creates the disbalance. So, here we went left and again a left, so it is a left- left condition.

Here we do a **right rotation** from the disbalanced node and the new BST looks like below:

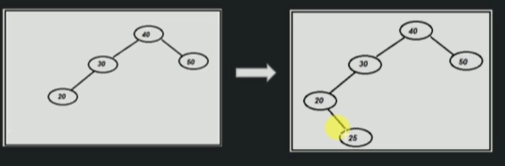


**Right rotation algorithm:**



Time complexity: O(1); space complexity: O(1)

1. **Left Right condition (LR)**



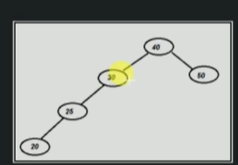
When we insert the node 25 at the tree then the new node gets inserted in as the right child of 20.

Here at node 30, its left child height is 1 and other child has the height as -1, the difference is 2. Hence, the tree is unbalanced. We can observe that the left- right node from the current node is causing disbalance.

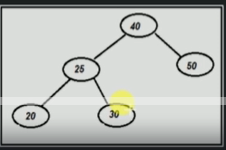
Here we will do **left rotation** first and then **right rotation**.

The left rotation is done on 20 instead of node 30.

After eft rotation it looks like below:

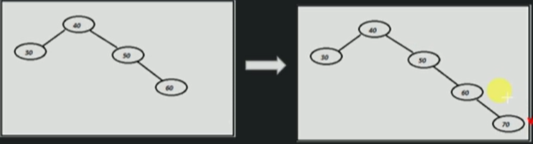


Now we do right rotation in the node 30 and after the right rotation it looks like below:



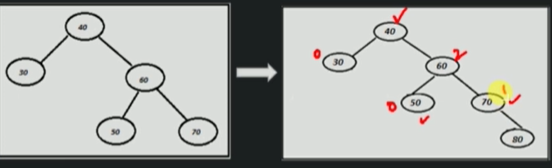
Time complexity: O(1); space complexity: O(1)

1. **Right Right condition (RR)**



Here as we check the height of each node, on doing so we find that the node 30 is disbalanced due to insertion of 70 which is its grandchild and can be reached by going right and then right of the unbalanced node.

Here the grandchild with max height is problem creator so we perform **left rotation** on the unbalanced node.

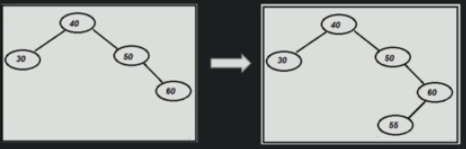


Here node 40 has two grand child and they are 70 and 50, the height of 50 is 0 and height of 70 is 1 so the operation occurs in the grandchild with max height.

Time complexity: O(1); space complexity: O(1)

1. **Right Left condition (RL)**

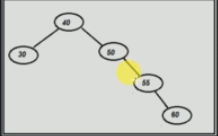
It is the mirror image of left-right condition.



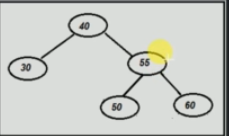
On inserting the node 55 the tree gets disbalanced at node 50 so we perform rotation on the grandchild node with maximum height which in this case is only node 55 which is accessible from the disbalanced through left- right traversal.

So, we perform **right rotation** on the unbalanced node’s right child and then **left rotation** on the unbalanced node.

Tree after right rotation



Tree after left rotation:



Time complexity: O(1); space complexity: O(1)

Insertion in AVL tree- O (log n); Space complexity: O(1)

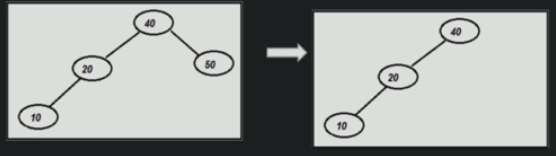
**Deletion from an AVL tree:**

3 cases :

1. Tree is not present
2. No rotation is required (like BST)
3. Is leaf node
4. Has one child
5. Has two children
6. Rotation is required.

LL condition:

After deletion of node 50:



Here we can see that node 40 is unbalanced so the grandchild of 40 is 10 which is in left- left part so we do the right rotation. Similarly, whenever we delete a node we will check the above three condition.

Time complexity: O(log n); space complexity: O(log n)

**Heap:**

Binary Heap is a special tree with special properties.

**Heap property:**

The value of any given node in a heap must be less than the value of its children(**Min-heap**)

The value of any given node in a heap must be greater than value of its children(**max-heap**).

All the levels in a binary heap is filled except possibly the last level, where all the nodes are as much left as possible.

**Practical use of Binary Heap:**

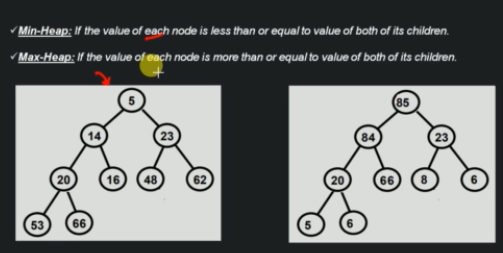
Prim’s Algorithm

Heap sort

Priority queue

Types:

Min-heap;



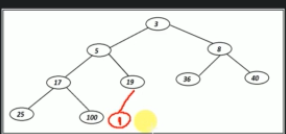
**Implementation of Heap:**

Implementation by array

Implementation by reference

**Insertion in heap:**

While inserting in Heap we must always check the heap property, here:



As we can see we are adding node 1 into the heap and as we do that we can see that binary heap is not maintained so we swap node 1 with node 19 and similarly it goes on and we keep doing that unless the heap property is maintained again.

Time complexity: O (log n);

Space complexity: O (log n);

**Deletion of root from a Heap:**

We can only delete **the root node of the heap** which is an inherent property of the Heap.

Time Complexity: O(log n); Space complexity: O(log n)

**Why avoid reference-based implementation be avoided in heap?**

The main problem occurs while extraction the root node and while extracting the element we need to move to last element and due to reference based implementation we may take O(n) time complexity instead of O (log n).

**Trie Data structure:**

* It is search tree, used to store/search strings in space/time efficient ways
* In it any node of a trie can store non repetitive multiple characters.
* Every node stores the link of the next character of the string.
* Every node keeps a track of the end of the string.

**Need of trie:**

Spelling checker

Autocomplete string

**Operation:**

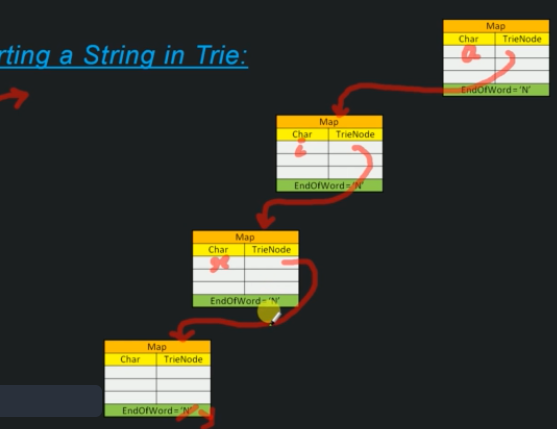
Creation- Create a blank root node



Logical and physical representation of a trie.

**Insertion-**

**case 1:** Trie is blank with inserting of string **“air”**



Here we can see that for each character for the string air we create a node and store each character in that node and also we create one extra node to store the end of the word, representing that the string is over and marked by Boolean end of word as true.

**Case 2**- new string’s prefix is already present in the Trie.- “aio”

Since character ‘a’ and ‘i’ are already present in the Trie so we move and store the letter ‘o’ in the node with r and create a blank node again and connect it to letter o and mark the Boolean end of tree in the new node as yes to denote end of string.

**Case-3**- new string’s prefix is already present as a complete string- “airk”

Since String “air is already present in the trie, we go ahead and store the character k in the blank node containing the Boolean as yes and from there connect the letter k to a new blank node and mark its endofword as true.

**Case 4:** string is already present in the trie

In this case we don’t need to do any extra work and we can just check it in the Trie.

**Search-**

Case-1 : String does not exist (”xyz”)

Here we will check if the root node of the trie contains the character ‘x’ or not , if not then we return as false.

Case 2 : String exists in trie

Here we keep checking the node and its child nodes until we reach the end of word as true

Case 3- current string is aprefix of the other string -“ab”

Here we keep checking and we find that ab is here in the tree but the next node ( child node) has end of word Boolean as false which means that the string is not present as a complete string.

**Deletion-**

Note- In trie deletion happens from leaf to root.

**Case 1**: deletion of string if the its prefix is same as other word prefix (BCDE, BCKG)

First, we check if the node is present in the trie, if there then we keep deleting from the end node and keep checking if the node has any dependencies. If at all there is any dependencies, we can delete the character from the node instead of the complete node.

**Case -2:** The word is a prefix of another word (BCDE, BCDEF)

Here we will just update the end of word from node containing the last letter of the later to false and hence the word gets deleted.

**Case-3**: some other word is a prefix of this word (BCDE, BC)

First, we check if the node is present in the trie, if it is there then we keep deleting from the end node and keep checking if the node has any dependencies. If at all there is any dependencies, we can delete the character from the node instead of the complete node. And change the end of word Boolean as false.

Case 4: Nobody is dependent on the current word

Practical uses:



**Hashing:**

Method of sorting and indexing data, it allows large amount of data to be indexed using keys commonly created by formulae.

**Hash Functions:**

Characteristic of a good hash function:

It distributes all hash values uniformly across the hash table

The hash function uses all the input data.

**Collision:**

A collision occurs when a hash function produces the same hash value for different input data.

Types of Collision:

**Direct chaining method:**

In this method instead of storing the string in the array we create a linked list where we store the data and the array stores the address of the linked list.

As new words produce the same hash function, we create a new node to store the new string and connect it with the array.

**Open addressing:**

**Linear probing:** Here we get a hash value for a string and store it in the array and if t all different string gives same hash value then we store the value in the next array index and similar fashion goes on.

**Quadratic probing:** Here we get a hash value for a string and store it in the array and if at all different string gives same hash value then to avoid collision in the array we add squares of natural number to the hash key and check if that location in the array is free or not.

**Double hashing:** In double hashing there are two hash functions. Anytime there is a collision in the hash values then a new hash value is created by the secondary hash functions and then is added to the old hash value created by the former hash function. If there is collision after that, then, we multiply the hash value created by the secondary function with natural numbers from 2 (2,3,4,….) and check if the cell in the array is empty or not, if not then we add a new integer to the later hash value and check again. If the new hash value is beyond the cell size of the array then we reach the cell address by taking a modulus of the total new hash value by the new hash value formed by multiplying the natural num to the later hash value.

**If the hash tale is full:**

In case of open addressing (linear probing) if the hash table is full then we need increase the size of the array by twice and put the values in the new array again by applying the hash functions. This process continues.

**Comparison of Collision resolution techniques:**

**Direct chaining**- no fear of exhausting the hash table

Cons- linked list might get too big and searching for values might take quite long time.

**Open addressing** – easy implementation

Con- exhausting of hash table which leads to processing of the hash function again and increase time complexity.

* If the input size is already known, then we should go for “open addressing” else we can go with any of the two.
* If the deletion operation happens quite a times in the hash table then we can go for direct chaining.

**Practical uses:**

Password verification in servers- converts password into hash value to store it in the server.

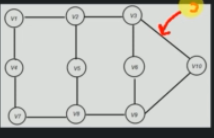
File system – The address of the file is passed as a key to the hash function and arrives with a hash value, the system uses it to store it physically in the hard disk. The address like cylinder number, sector and disk all are stores in the hash table at the cell with address as the hash value.



**Graphs**



Graph- a pair of sets (V,E) where V is the vertices and e is set of edges connecting the pair of vertices.



**Need of graphs:**

Shortest path finding.

Vertices- nodes of the graph

Edges- connect a pair of vertices

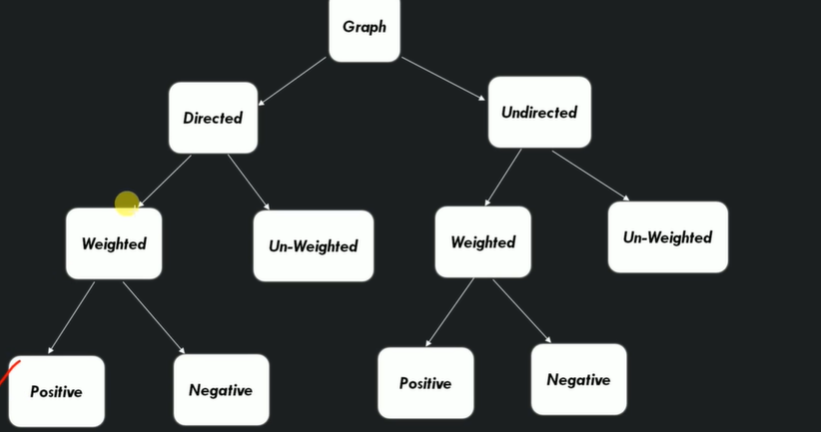
Unweighted graph- a graph having no weight associated with the edges of the graph

Undirected graph- if the edge has no direction defined for the edge

Cyclic graph – a graph having at least one cycle.

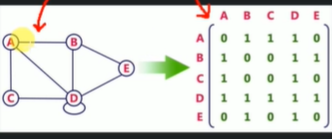
A directed acyclic graph is known as tree.

**Types of Graph:**



**Graph representation:**

1. Adjacency matrix- a 2D matrix



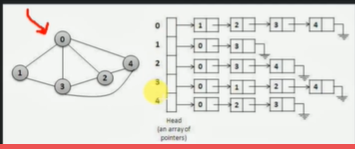
Here the size of the matrix is equal to the number of vertices in the graph.

To be used when there are maximum number of edges in the graph.

1. Adjacency list: - a linked list

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes a set of neighbors for each vertex.

To be used when the number of edges are not complete and is few.

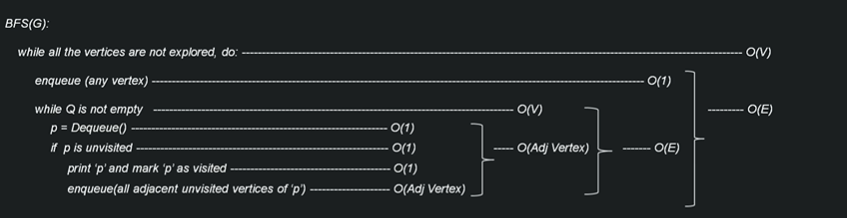


**Breath First Search:**

**It is a method of graph traversal (process of visiting each vertex of the graph)**

* Starts from arbitrary node and starts traversing other node of a graph and explores the neighbor nodes first, before moving to the next level neighbors
* Nodes are traversed in a level order
* Uses queue data structures to print the vertices in level order

**Algorithm:**



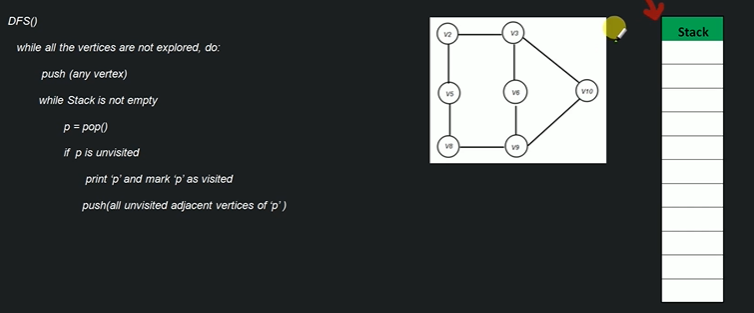
Time complexity: O(V+E); space complexity: O(V+E)

**Depth First Search:**

It is traversing the graph data structures; it goes as far as possible along each edge before backtracking.

It uses stack data structure for traversing.

**Algortihm:**



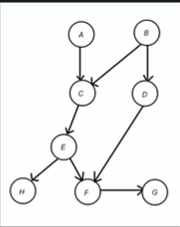
Time complexity: O(E+V); space complexity: O(E+V)



**Topological sort:**

Sorts given action in such a way that there is a dependency of one action on another action then the dependent action always comes later then the parent action.

**Algorithm:**



If a vertex is dependent on another vertex then go to that vertex first and then come back to that vertex else push that vertex to the stack.

As we move to each vertex, after pushing it to the stack , we mark it as visited and go back to the previous stack , this continues till all the vertex are not in the stack and then pop put each element from the stack and print them.

Time complexity: O(v+e) ; space complexiity: o(e)



**Single source Shortest path Problem (SSS)**

Finding a path between a given vertex to all other vertex in the graph such that the distance between them is minimum.

Various algorithms used here are:

BFS, DFS, Dijkstra, Bellman Ford

**Using** **BFS**:

Here the algorithm is same as the BFS but with extra parent node variable to keep a check of all the variables which are visited and their parent node from where visit was made.

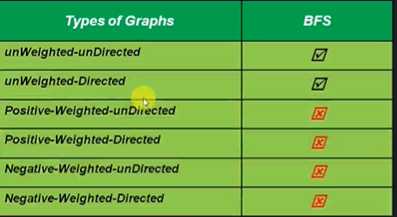
An extra parent variable is introduced to keep the track of the path.



TIME complexity: O(E)

Space Complexity: O(E)

**Why BFS is not optimum for SSS:**

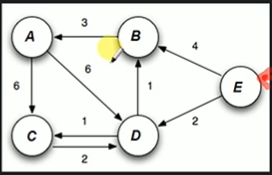


**Reason**: In case of weighted graph, we cannot use BFS for SSS as because BFS searches in **breadth** **way** ignoring the weight of the paths and there could be a better way to reach the destination keeping all the weights in the mind.

**Why DFS does not work in SSS?**

**Reason**: DFS has tendency to move far away from source so it cannot search the shortest path.

**Dijkstra Algorithm**:



Let us try to traverse from vertex E to all other vertices

E-A: 6; E-B: 4;…etc.

**Algorithm**:



Here set all the distance of the other vertices as infinite (large value) and the source vertices as 0.

**Time** **Complexity**: O(v^2)

**Space** **Complexity**: O(v)

Dijkstra algorthim

**Sorting:**

**Quick sort:**

Divide n conquer algorithm

At each step, a pivot is found such that all the smaller elements are left of it and all larger elements are to the right of it.

In space algorithm

3 variables are kept in check in quick sort, i.e., i,j,p.

p- pivot in the array, usually the last element

j- first element of the array, and then keeps traversing

i – j-i

we keep checking if the element in the jth index is more than the pivot, if not, then we swap the jth element with the (i+1)th element.

Time complexity: O(n log n)

Space complexity: O(n)

Not a stable algorithm.

Java uses quick sort algorithm.

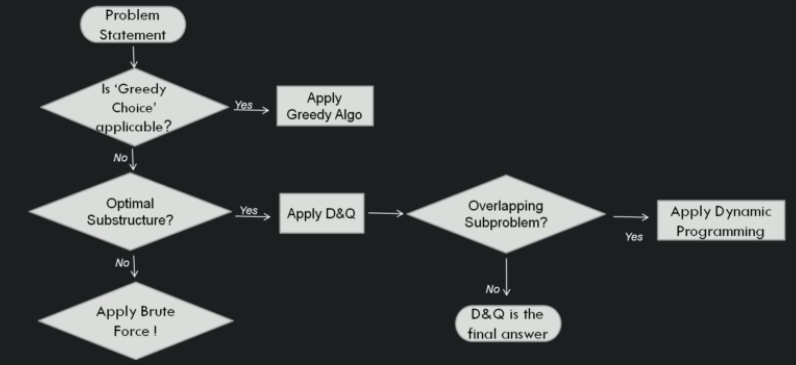
**Heap Sort algorithm:**

* Organizes data into a special sorted data in a heap
* Removes topmost array from the heap and inserts into the array
* Works best with array, does not work with linked list.



**Magic Framework:**

-A flow chart which helps us in better decision making and solve the problem statement.



**Greedy Algorithms:**

Greedy algorithm is a algorithmic paradigm that builds up solution piece by piece.

Chooses the next piece which offers the most obvious and immediate benefit

Greedy fist perfectly for those solution in which choosing a locally optimal solution also leads to globally optimal solutions.

Greedy choice property:

Insertion sort

Selection sort

Topological sort

Prim’s algorithm

Kruskal’s algorithm