

Data Structures and Algorithms

What is Data Structure?

It is a way to store and organize the data, so that it can be used efficiently.

Examples

- ① Array ③ Structure ⑤ Stack ⑦ Graph
- ② pointer ④ LinkedList ⑥ Queue ⑧ Searching
- ⑨ Sorting

* DS is a set of algorithms that we can use in any PL to structure data in memory.

What is abstract datatype?

* To keep the data structured in memory, abstract data type concept is been introduced, the ADT is bound by set of rules.

Data Structure

Primitive Data Structure

- int
- char
- float
- double
- pointer

(can hold a single value)

Non Primitive Data Structure

- [Linear DS]
 - Arrays
 - LinkedList
 - Stacks
 - Queues

(store data in seq)
Linear

Non Linear DS

- trees
- graphs

(element connected with n number of elements)

Data Structures

- ↳ Static DS (size allocated at compile time)
Maximum size is Fixed
- ↳ Dynamic DS (size allocated at runtime)
Maximum size is Flexible

Operations on DS

- ① Searching
- ② Sorting
- ③ Insertion
- ④ updation
- ⑤ Deletion

Necessity of DS

- * Store the data efficiently in terms of time and space.
- * We require some DS to implement another DS or a particular ADT.
- * Ex: Stack (ADT) created / implemented using LL (DS).

ADT \Rightarrow Blueprint

DS \Rightarrow Implementation

- * Selection of DS to implement ADT depends on user requirement (Time / space)

Advantages of DS

- ① Efficiency (Efficient DS \rightarrow Efficient time & spaces)
- ② Reusability
 - ↳ Create an Interface by using Efficient DS, provide that to client, He uses everytime
- ③ Abstraction
 - ↳ Implement stack using array / LL, provide the interface to user, user don't have implementation details hence Abstraction

Main AIM: Store and retrieve as fast as possible

Basic Terminology

- ① Data: Elementary value or collection of values
- ② Data Item: Single unit of value
- ③ Group Items: Data items that have subordinate D.Items
Ex: Employee Name (First, Last, Middle Name)
- ④ Elementary Item: DataItem which are unable to Divide (EID)
- ⑤ Entity: Object that has a distinct identity (Person, Places)
- ⑥ Attribute: Characteristic of An Entity

Datastructures \Rightarrow Allows us store & organise data

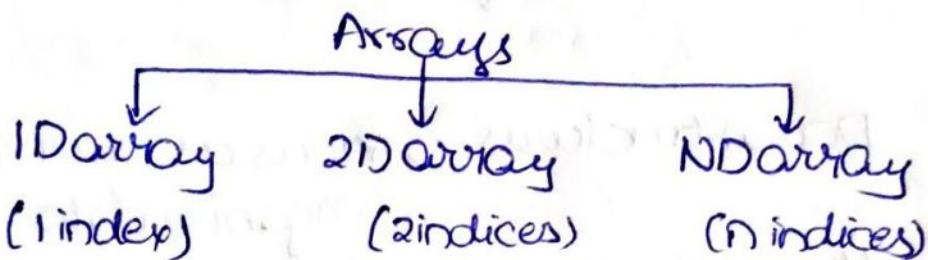
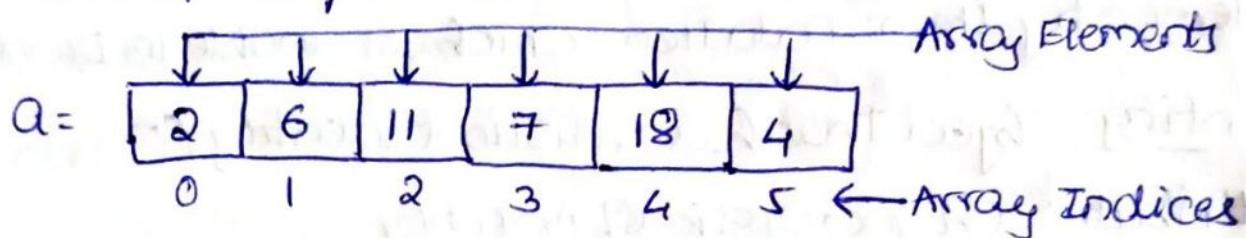
Algorithms \Rightarrow process the data meaningfully

- * Non Primitive Data Structure are data structures derived from Primitive DS
- * NPDS forms set of data elements that is either group of homogeneous/heterogeneous Data structure
 - ↳ same DT form as grp \hookrightarrow diff DT forms as grp

Linear Data Structures

① Array

- * Collect Multiple data elements of the same datatype into one variable.
- * Data is stored in contiguous memory location, so retrieving randomly through index based on array variable is possible.



Applications

- ① Store list of data elements of same type
- ② use as auxiliary storage for other data structures
- ③ store data elements of Binary tree of Fixed count

② Linked List

- * Singly linked list:
- * Doubly linked list
- * Circular linked list

Applications of Linked List:

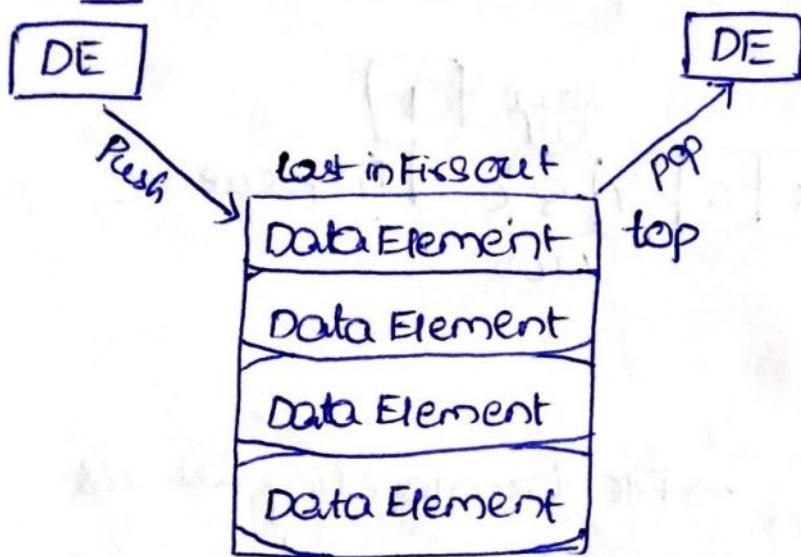
- * Helps us implement stacks, queues, binary trees and graphs of predefined size.
- * OS functionality, Dynamic Mem Manag
- * Slideshow functionality of PPT
- * First slide → last slide
- * DLL → Browser Front & Back navigation

Stack

- * Linear Data Structure that follows LIFO
- * Insertion & Deletion only from top end.
- * Implemented using
 - ↳ contiguous memory (Array)
 - ↳ Non contiguous Memory (linked list)

- * Can access only Stack's top at any time

operations:



Applications:

- * Temporary storage for recursive operations, function calls, nested operations
- * Evaluate arithmetic Expressions
- * Infix Exp \rightarrow Postfix Exp
- * Match parenthesis
- * Reverse a string
- * Backtracking
- * DFS in graph & tree traversal
- * undo/redo func

Queues

- * Linear Data Structure
- * Insertion done at one end
- * Deletion done at opposite end
- * First in First Out

Can be implemented by

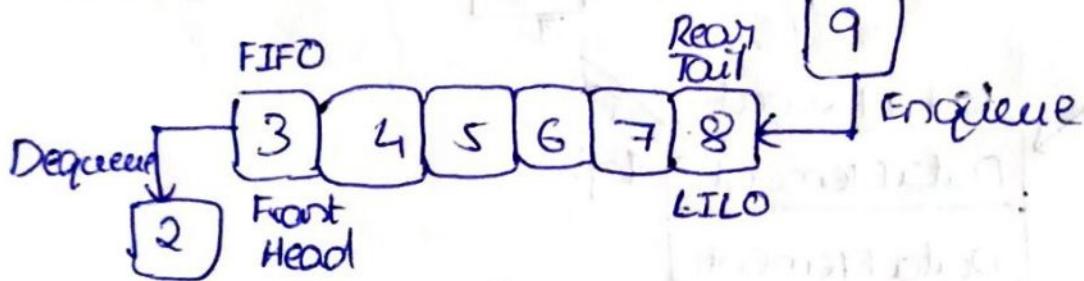
- Arrays
- Linked Lists
- Stacks

Real life Examples

ticket counter

Escalator

car wash



Applications

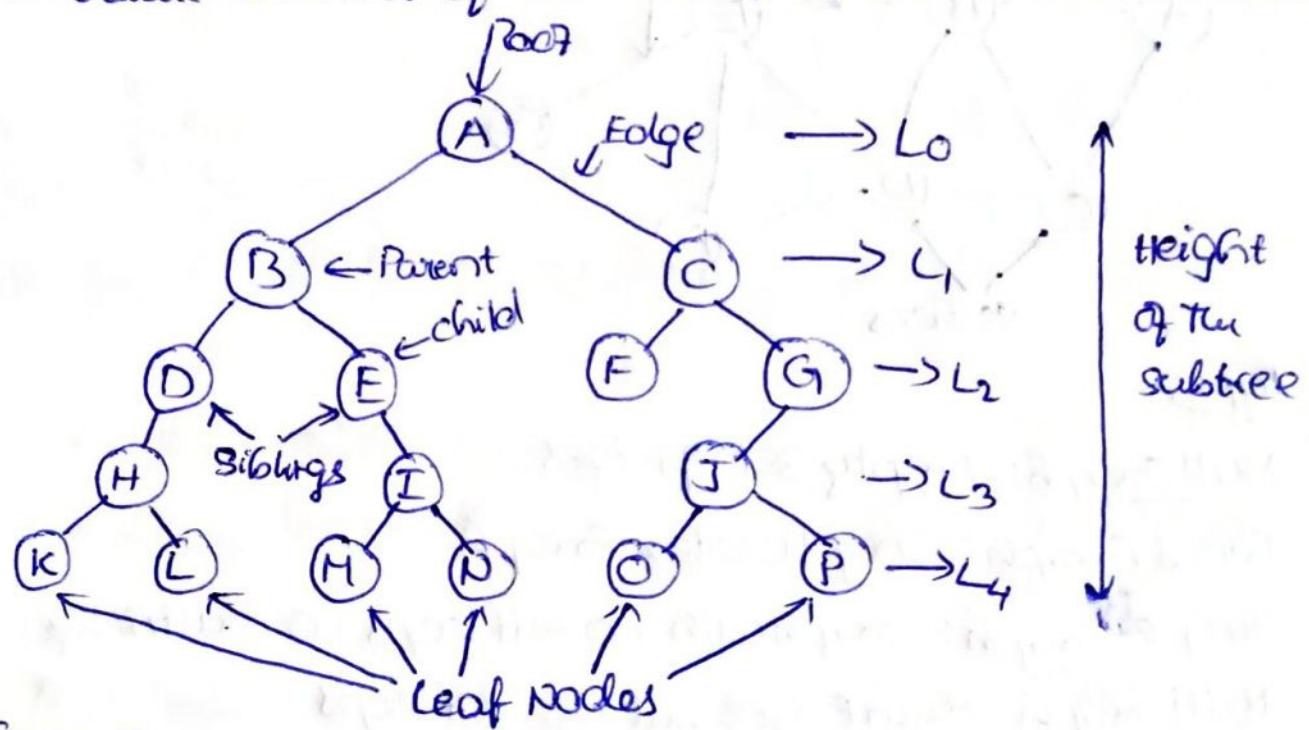
- BFS in Graphs
- Job scheduling operations
- CPU/Job/Disk Scheduling
- File download queues
- Handling interrupts generated by user apps

Non Linear Data Structure

- * Data elements not arranged in sequential order.
- * Insertion & Removal is not that easy, hierarchical dependency between data items

Trees

- * collection of Nodes such that each node of tree stores a value and list of references to other nodes (children)



Types of Tree:

Binary Tree: 1 Parent node \rightarrow atmost 2 children

Binary Search Tree: can maintain sorted list of nums

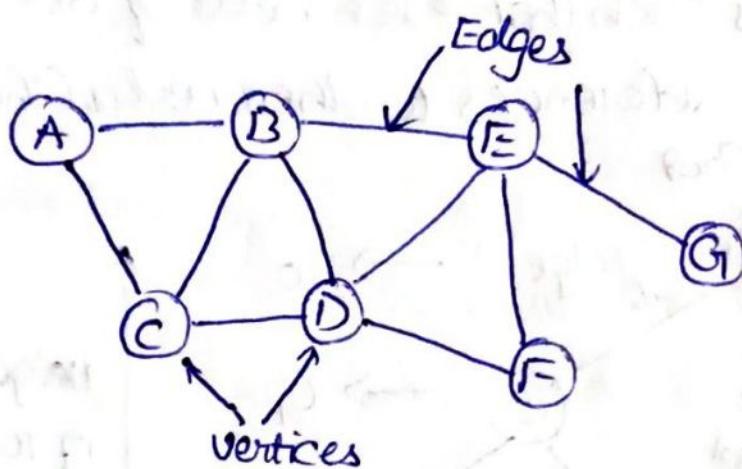
AVL Tree: Self Balancing Binary Search tree, Each node have Balance Factor (-1, 0, 1)

BTree: Similar to AVL Tree, Each node can have more than two children.

Graphs

* Finite nodes or vertices and the edges connecting them.

$G = (V, E) \Rightarrow$ set of vertices & Edges



Types

Null Graph: Empty set of edges

Trivial Graph: Only 1 vertex Graph

Simple Graph: Graph with no self loops no multi edges

Multi Graph: Multi edges but no self loops

Pseudo Graph: self loops & multi edges

Non-Directed Graph: non directed edges

Directed Graph: directed edges

Connected Graph: atleast a single path b/w every pair of vertices

Disconnected Graph: atleast one pair of vertices doesn't have edge

Regular Graph: all vertices have same degree

Complete Graph: all vertices have an edge b/w every pair of vertices

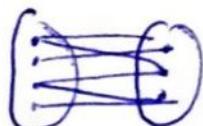
Cycle graph: At least 3 vertices and edges form a cycle

Cyclic graph: At least one cycle exists

Acyclic graph: zero cycles

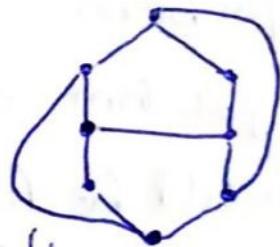
Finite / Infinite Graph: Finite / Infinite number of vertices / edges

Bipartite Graph: vertices can be divided into 2 sets
Set A vertices can be connected to Set B vertices



Planar graph: If we can "draw in a single plane" with
"two edges intersecting to Each Other"

Euler graph: All vertices are even degrees



Two Edges intersect
to Each Other

- ① Traversal
- ② Search
- ③ Insertion
- ④ Deletion
- ⑤ Sorting
- ⑥ Merge
- ⑦ Selection
- ⑧ update
- ⑨ splitting

Important points:

- * All Data Structures are Examples of ADT
- * If user want to store the data into memory, we provide array or LL, user don't know the implementation taken or this is main idea of Abstract Data type

Application of DS

- Rep info of DB
- search through org data
- Generate test data
- Encrypt & Decrypt data

Algorithm

* set of rules required to perform calculations or some other problem solving operations Especially by computer.

→ Flowchart

→ Pseudo code

Why Algorithms

* Scalability:

we need to scale down a real world big problem into small steps, which helps us to analyze the problem

* Algorithm says that each and every instruction should be followed in specific order to do a specific task.

Algorithms should consider these while creating one

→ Modularity (Break problems into small chunks)

→ Correctness (Generate precise output with precise input)

→ Maintainability (designed in simple structured way)

Some algorithmic approaches

① Brute Force algorithm:

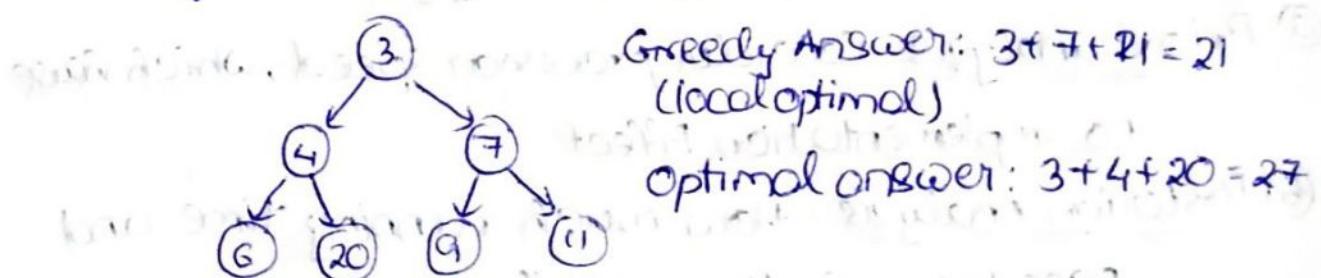
- * searches all the possibilities to provide the required solution
- Optimized method: Take best solution out of all solutions
- Sacrificing: stops at first solution, doesn't care about optimized or not

② Divide and conquer

- Divide bigger problem into smaller and solve them and merge the output's to get result of solution

③ Greedy Algorithm

- Problem solving approach of making the locally optimal choice at each stage with the hope of finding a globally optimum.
- May not provide Globally optimized values
- One of the case that would fail



When to use?

- * Global optimum can be reached using local optimism
- * optimal solution to a problem contains opt sol of subprob

Applications

- * Activity Selection prob
- * Huffman coding
- * Job sequencing
- * Fractional knapsack
- * Prim's Min Spanning

④ Dynamic Programming

- * Breaks problem into subproblems
- * Stores results of subproblems using memorization,
- * Find the optimal solution out of two subprob
- * Reuse the result of subproblems, to not execute twice.

⑤ Branch and bound algorithm:

- * can be applied to only integer problems
- * method of solving optimization problems by breaking them down into smaller sub problems and boundary function to eliminate subproblems that cannot contain the optimal solution.

⑥ Backtracking

- * Solves the problem recursively and remove the solution if it doesn't satisfy constraints of problem.

Algorithmic Analysis

① Priori analysis: Consider processor speed, which have no implementation Effect

② Posteriori Analysis: how much running time and space taken by the algorithm.

Algorithmic Complexity

① Time Complexity:

- * amount of time req to complete the execution
- * Denoted by Big(O) notation.
- * number of steps it may take to complete execution

sum = 0
for i in 1 to n:

$$\text{sum} = \text{sum} + i; \quad \left\{ O(n) \right.$$

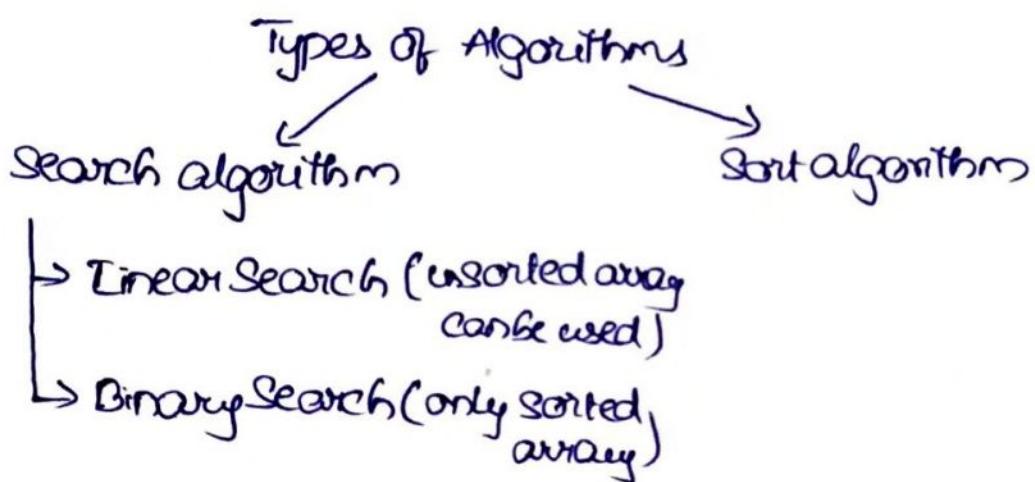
return sum; $\rightarrow O(1)$

② Space Complexity:

- * Amount of space required to solve a problem and produce an output.
- * Expressed in Big O notation.
- * Space is required by store program instructions, constant values, variable values, function calls, jumping statements etc.

Auxiliary space: Extra space required by algorithm, excluding the input size.

Space complexity: Auxiliary space + Input size



Asymptotic Analysis

we choose the best algorithm, based on time complexity.

Real time example:

If we need algorithm to add a element in front
then possible solution could be by array, Linklist.
But the best solution is through linked list because
in array's we need to push all elements to right to
add an element to the beginning array. But for
linked list we need to add a node in front and tag
the address with the second node.

* using Asymptotic analysis, we can conclude the
average case, best case and worst case scenario
of an algorithm.

Example: $f(n) = 5n^2 + 6n + 12$

For small values of $n \geq 12$ is best

For large values of n (worst case) $= 5n^2$ is best

So T.C should be taken based on worst case

$n^3 \Rightarrow$ More Time Complexity

$n^r \Rightarrow$ comparatively less time complexity

If we have an algo which solves in n^2 or n^3 then
we must go for n^r as less time. C than n^3

Time Complexity

- ① worst case
- ② Average case
- ③ Best case

Asymptotic notations used for calculating T.C

* Bigoh Notation (O)

- provides the order of growth of the function
- provides an upper bound on a function
- It Ensures that program that we written will be less or equal complexity of our upper bound.
- provides worst-time complexity

* Omega Notation (Ω)

- provides best case scenario
- Opposite action to Bigoh Notation
- provides lower bound of an algorithm
- provides faster time that an algorithm can run.

* Theta Notation (Θ)

- provides average case scenarios.
- provides realistic time complexity of an algorithm.

Common Asym notations

Constant	$O(1)$	→ Adding to the front of a linked list
$O(\log N)$	\log	→ Finding an entry in a sorted array
$O(N)$	linear	→ Finding an entry in an unsorted array
$O(N \log N)$	$n \log n$	→ Sorting n items by 'divide & conquer'
$O(N^2)$	quadratic	→ Shortest path b/w nodes in graph
$O(N^3)$	cubic	→ Simultaneous linear equations
$O(2^N)$	exponential	→ Tower of Hanoi problem

Pointer

- * Pointer is used to points the address of the value stored anywhere in computer memory
- * obtaining such value using pointer is called dereferencing the pointer.

Mainly useful in

① lookup tables

② Traversing String

③ control tables

④ Tree Structures

Pointer can be used in ways

1) Pointer arithmetic

$\Rightarrow ++, --, +, -$

2) Array of Pointers

3) pointer to pointer

4) passing pointer to function

5) Return pointer from function

$a \rightarrow [10] \rightarrow \text{value}$
 $2000 \rightarrow \text{address}$

$b \rightarrow []$
 3000

$b = \&a \rightarrow [] \rightarrow [10]$
 $(b \text{ points } a)$
 $3000 \rightarrow 2000$

Pointer
`int a=5
int *b;
b = &a;`

Pointer to Pointer
`int a=5;
int *b;
int **c;
b = &a;
c = &b;`

Structure

- * Structure is a composite data type that defines a grouped list of variables that are placed under one name in block of memory.
- * Allows different variables to be accessed by using a single pointer to structure.

```
struct structure_name {  
    datatype member-1;  
    datatype member2;  
    :  
    datatype member;  
};
```

```
struct structure_nm e1,e2,e3;
```

Advantages

- * can hold variables of different datatypes.
- * reuse the data layout across program.
- * used to implement LL, stacks, queues, trees, graphs etc.

Arrays

- * collection of similar types of data items stored in contiguous memory locations.
- * randomly accessible by using its index number.
- * Elements are of same datatype & same size
- * First element stored at smallest Memory location, second element is calculated Based on given base address and size of data element.

```
int array[10] = {35, 33, 42, 10, 4, 19, 27}  
↑      ↑      ↑      ↓  
Type   Name   Size      Elements
```

- * Index starts with 0
- * Sorting and searching a value in array is easier, fast to process values quickly, good for storing multiple value in a single variable
- * Types of indexing (zero based, one based, n based)

Byte address of element

$$A[i] = \text{base address} + \text{size} * (i - \text{first index})$$

Insert an element at particular index

```
x=50; //value  
pos=4; //index  
n++;  
for(i=n-1; i>=pos; i--)  
    arr[i]=arr[i-1];  
arr[pos]=x;
```

Time and space complexity of various array operations

Time Complexity:

	Average Case	worst case
Access	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$
insertion	$O(n)$	$O(n)$
deletion	$O(n)$	$O(n)$

Space Complexity

worst case $O(n)$

worst case $O(n)$

Advantages

* single name for group of variables of same type.

* Easy traversing and accessing through index.

Disadvantages

* Memory allocation is static, the size of array cannot be altered.

* Memory inefficient usage if we not use whole statically allocated

2D array

Declare 2D array: int arr[max-rows][max-columns]

Value can be accessed by: arr[i][j]

We must declare the size of the 2-dimensional array while we do initialization. This is not same while the case of single dimensional array.

int arr[2][2] = {{0, 1, 2}, {3}}

Mapping 2D array to 1D array:

→ cell index ① Row Major ordering ② Column ordering

↓ row idx 0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)



Locate address of 2D array, row / col Major

int A[3][4]

datatype size = 2

A	0	1	2	3
0	100	102	104	106
1	108	110	112	114
2	116	118	120	122

3x4
(mm)

Row major order:

$$\text{Addr}(A[2][1]) = [L_0 + (i \times n + j) \times w] \quad (\text{indices starts from 0})$$

$L_0 \Rightarrow$ Base address

$$\text{Addr}(A[2][1]) = [100 + (2 \times 4 + 1) \times 2]$$

$$\text{Addr}(A[2][1]) = 100 + 18 = 118$$

$$\text{Addr}(A[i][j]) = L_0 + [i \times n + (j - 1)] \times w \quad (\text{index starts with 1})$$

Column Major order

$$\text{Addr}(A[i][j]) = L_0 + [j \times m + i] \times w \quad (\text{index starts with 0})$$

$$\text{Addr}(A[1][2]) = 100 + (2 \times 3 + 1) \times 2$$

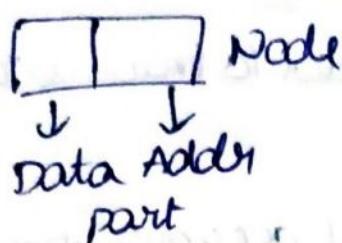
$$= 100 + 14$$

$$\text{Addr}(A[1][2]) = 114$$

If we want locate an element of a 2D array, then we must either use row major order / column major order

Linked Lists

- * Series of connected nodes
- * nodes that are randomly stored in memory



- * Last node of the list contains a pointer to null
- * Every link contains a connection to another link.

why linked lists over array?

- * Array the size should be prefixed, expand the size of array at runtime is not possible, Elements should be stored in contiguous memory. Inserting an element shift all its predecessors.
- * LL allocates memory dynamically, elements can be stored non contiguously and linked together with help of pointer. memory allocated at runtime.

Declare the structure of Node

```

class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None
  
```

we use classes to create a custom datatype similar to what we use for structure in c programming

Types

- ① Singly linked list
- ② Doubly linked list
- ③ Circular singly linked list
- ④ Circular doubly linked list

Advantages

- * Dynamic data structures
- * Ins & Del is easier
- * Memory efficient
↳ Memory used is dynamic and flexible
- * Implements other DS like stacks & queues

Disadvantages

- * Memory usage more than array. (pointer size is extra than array)
- * Traversal and random accessing is complicated.
Difficult, if we want to access last element, we need to traverse whole LL. no possible of random accessing.
- * Reverse traversing is needed for Backtracking but occupies more space.

Applications

- * Polynomial representation & sparse matrix representation
- * Implement
 - stack
 - queue
 - tree
- Adjacency list → Dynamic memory block

Time Complexity

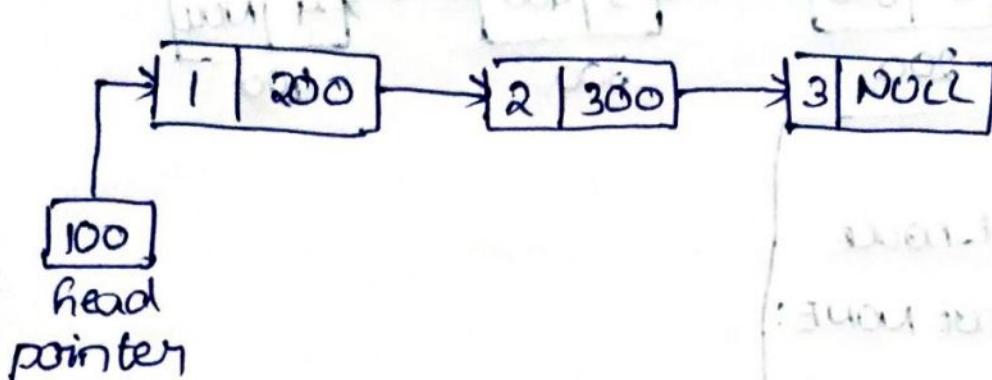
	<u>Avg Case</u>	<u>Worst Case</u>
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$

Space Complexity

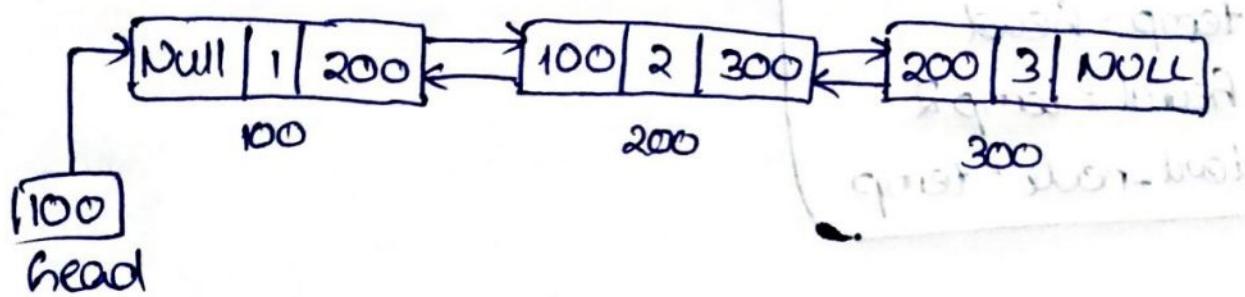
- * Space Complexity of Linked List is $O(n)$

Types of Linked Lists

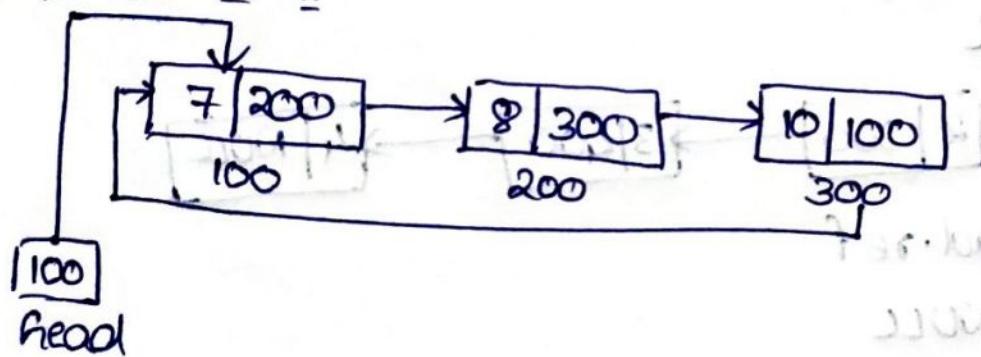
Singly Linked List (Single Link)



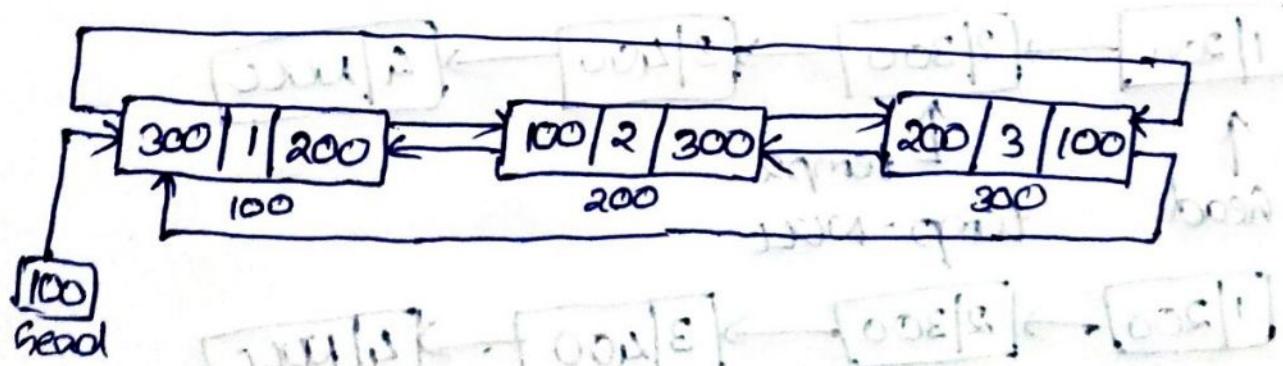
Doubly Linked List



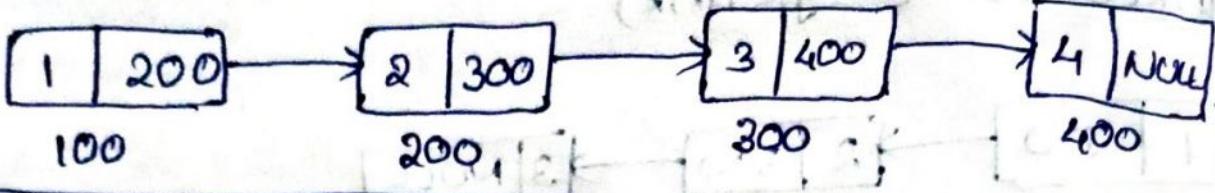
Circular Linked List



Doubly Circular Linked List



Reverse Linked List



temp = NULL

head = self.start-node

while head is not None:

 temp2 = head.ref

 head.ref = temp

 temp = head

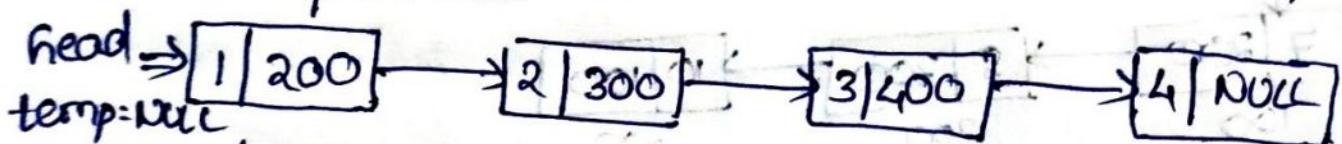
 head = temp2

self.start-node = temp

Step 1:

head = start-node

temp = NULL



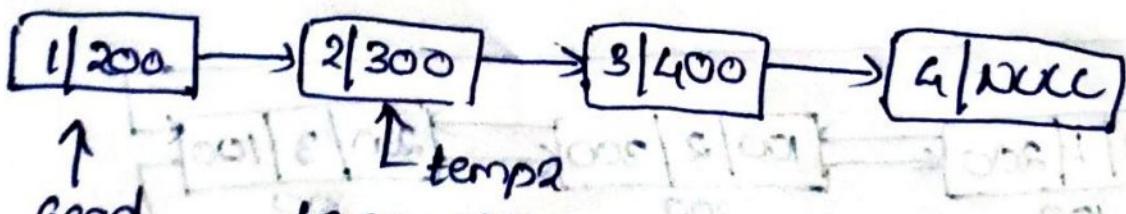
temp2 = head.ref

head.ref = NULL

temp = head

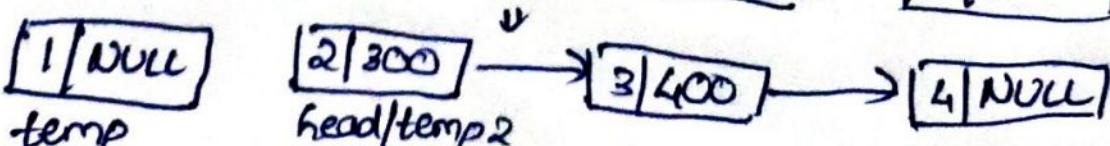
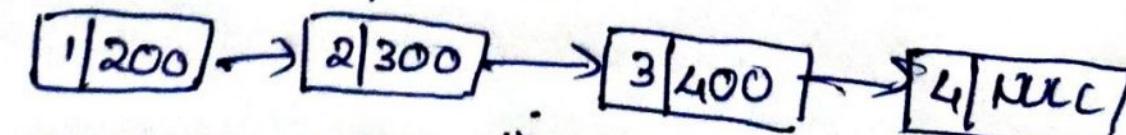
head = temp2

Step 2



temp2
head.ref = NULL

Step 3:



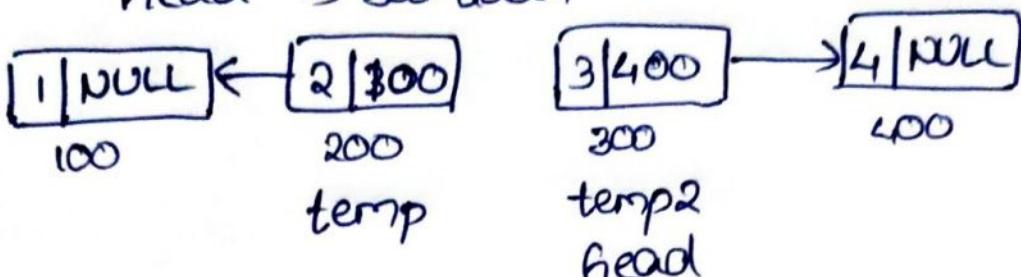
step4:

temp2 \Rightarrow 300 addr

head.ref \Rightarrow 100 addr

temp \Rightarrow 200 addr

head \Rightarrow 300 addr



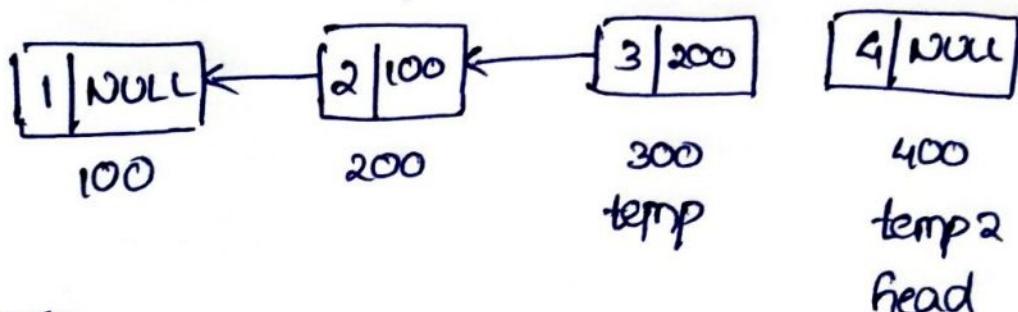
step5:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



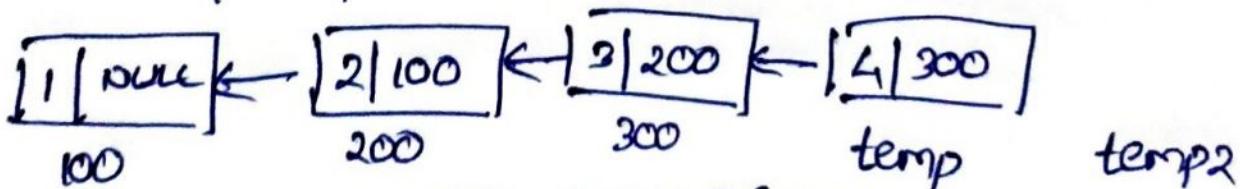
step6:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



At the end of the loop both temp2 & head both
points to same node

temp2

head

Linked list

- * It is a collection of objects called nodes that are randomly stored in the memory.
- * A list is not required to be contiguously present in the memory.
- * optimal utilization of space
- * list size is limited to memory size.

Single Linked List

Time Complexity

	Average	worst
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(n)$

best = constant

avg = linear

worst = linear

constant = best

Space complexity for SLL: $O(n)$

best = constant

avg = linear

worst = linear

constant = best



constant

avg

avg

avg

avg

best

avg

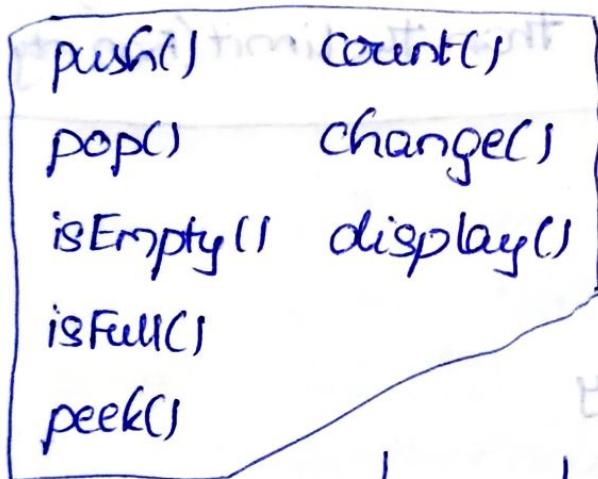
avg

avg

After inserting some new nodes at front it will increase size of array

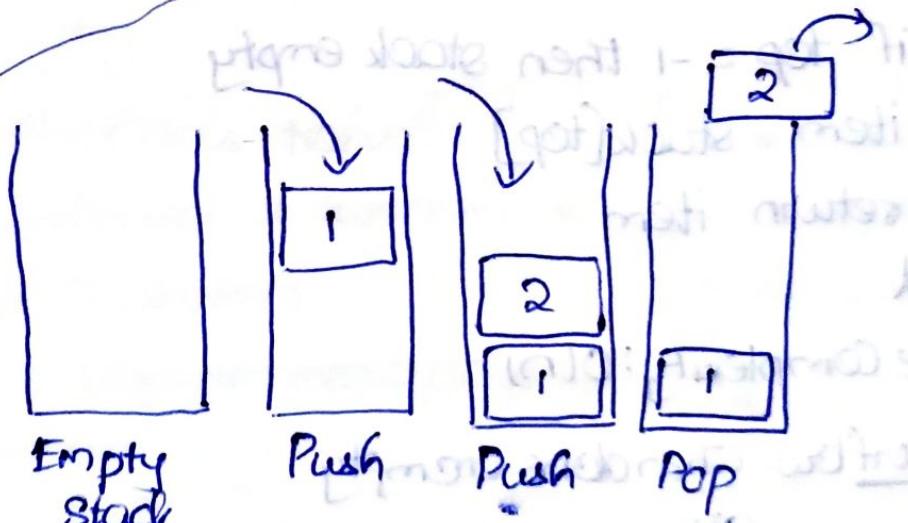
Stack

- * Linear data structure
- * Last in first out principle
- * Insertion & Deletion can be done from one end known as top of the stack.
- * Can store the elements of limited size
- * Abstract data type with pre defined capacity.
- * Filled from bottom to top , removed from top to bottom.



Applications

- * Balancing Symbols
- * String reversal
- * UNDO/REDO
- * Recursion
- * DFS
- * Backtracking
- * Expression conversion



Stack (arrays)

Push operation (Insert)

begin

if $\text{top} = n$ then stack full

$\text{top} = \text{top} + 1$

$\text{stack}[\text{top}] := \text{item}$

end

Time Complexity: $O(1)$

Overflow: inserted more than
the limit

Pop operation (Delete)

begin

if $\text{top} = 0$ then stack empty

$\text{item} = \text{stack}[\text{top}]$

$\text{top} = \text{top} - 1$

end

Time Complexity: $O(1)$

Underflow: removing more
than the limit (from my stack)

Peek operation (Display top)

begin

if $\text{top} = -1$ then stack empty

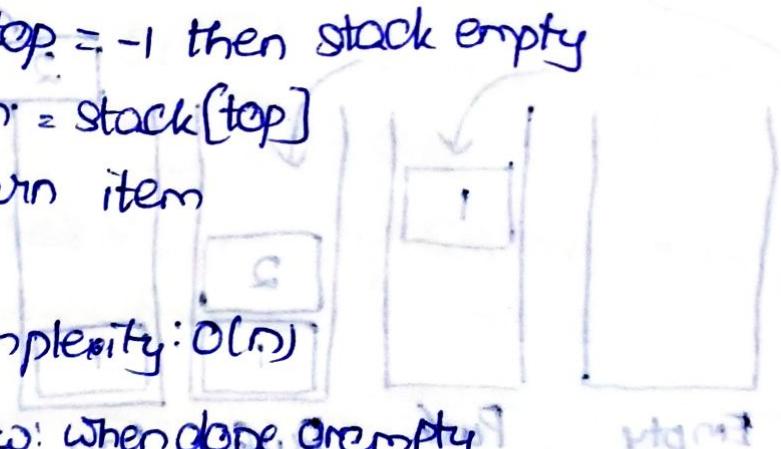
$\text{item} = \text{stack}[\text{top}]$

return item

end

Time Complexity: $O(1)$

Underflow: when done on empty
array

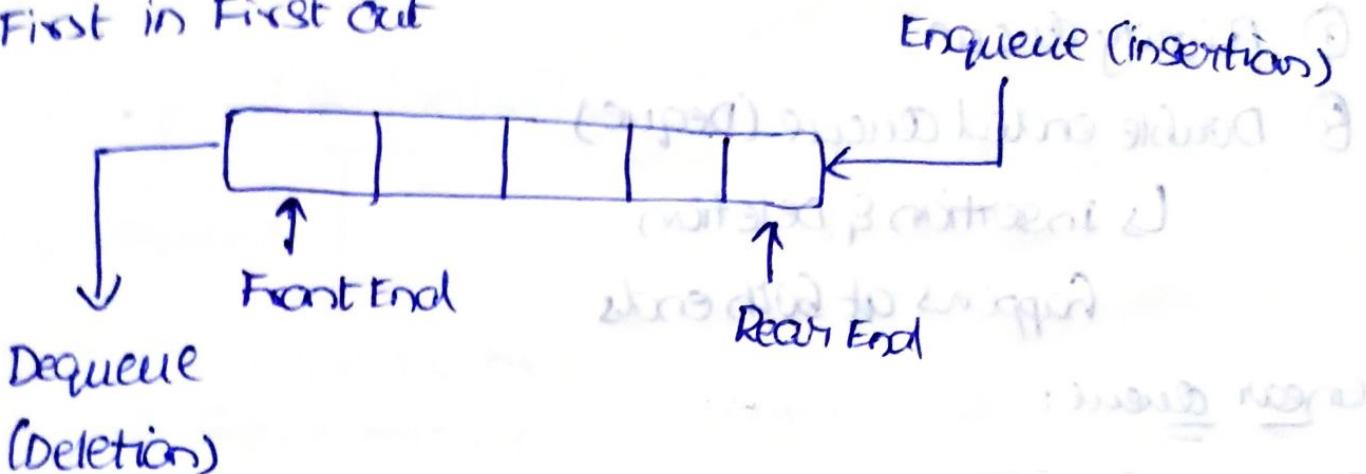


2. Implementation
 simple program *
 user input *
 data struct *
 function *
 2.10 *
 implementation
 ordered linked list *

Queue

- * ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

- * First in First Out



Applications of Queue

- * Mostly used for ordering of actions
- * used as waiting lists for a single shared resource like printer, disk, CPU.
- * used in asynchronous transfer of data (where data is not being transferred at the same rate b/w two processes)
Ex: pipes, files I/O, sockets
- * used as buffers (Eg: MP3 mediaplayer, cdplayer)
- * playlists in media player

Queue	
Average	worst
Access	$O(n)$
Search	$O(n)$
Insertion	$O(1)$
Deletion	$O(1)$

Space complexity (worst case): $O(n)$

Types of Queues

- ① Simple Queue
- ② Circular Queue
- ③ Priority Queue
- ④ Double ended queue (Deque)

↳ Insertion & Deletion

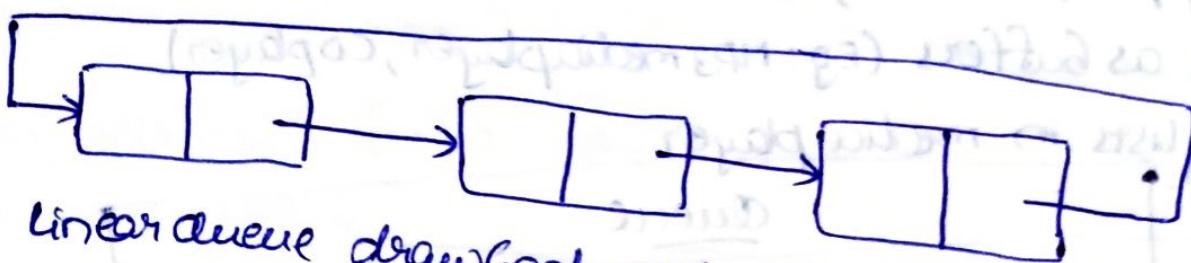
Happens at both ends

Linear Queue:



- * Even if 3 of 5 elements are removed, even if space is available, we can't enter new elements
- * overflow condition: If the rear pointer is pointing to the last element of the queue

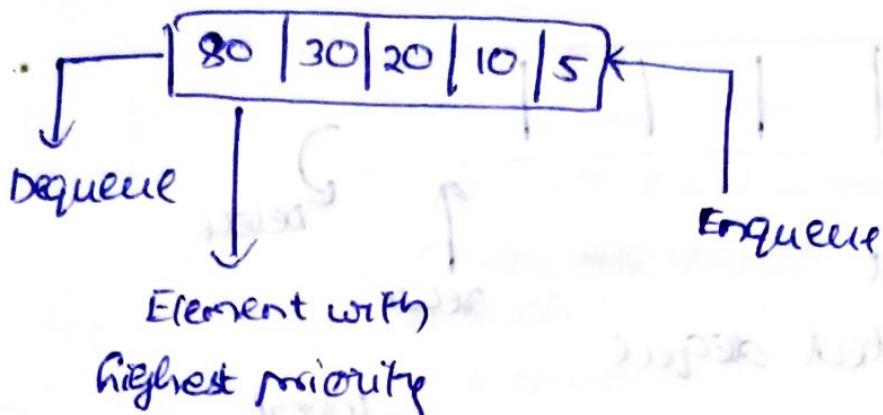
Circular Queue:



- * Linear queue drawback overridden by circular queue
- * Empty space can be utilized simply by incrementing value of rear.
- * Main advantage: Memory utilization

Priority Queue

- * Elements are arranged based on priority.
- * Every element have priority associated with it. If two are of same priority, FIFO rule applied.



Insertion: Based on the arrival

Deletion: Based on the priority

Ascending Priority Queue:

7	5	3
---	---	---

Insert in any order

7, 5, 3

Smallest can be deleted first

3, 5, 7

Descending Priority Queue:

1	7	3	5
---	---	---	---

Insert in any order

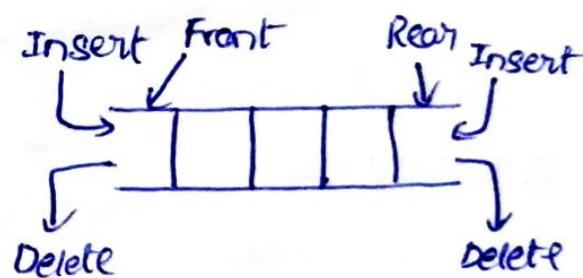
7, 3, 5

Delete largest first

7, 5, 3

Deque (Double ended queue)

- * Can insert and delete from both front and rear ends of the queue.
- * palindrome checker
- * Deque ^{not} performs FIFO, as this is illogical for queue rules
- * Deque Only performs LIFO, basically it can act as stack & queue

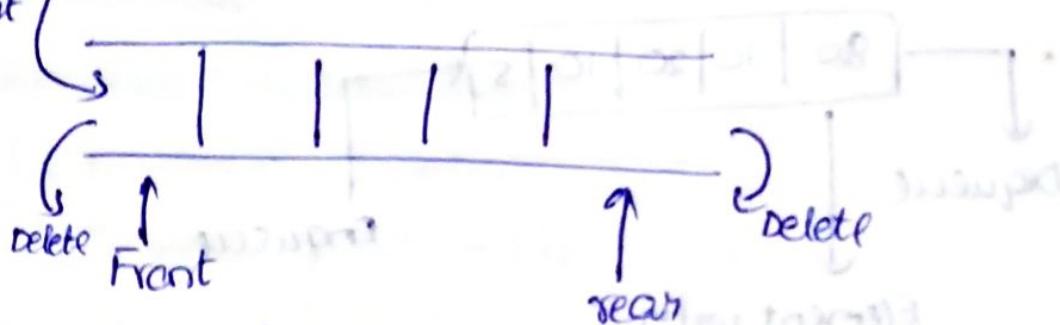


Double ended queue

Types of deque

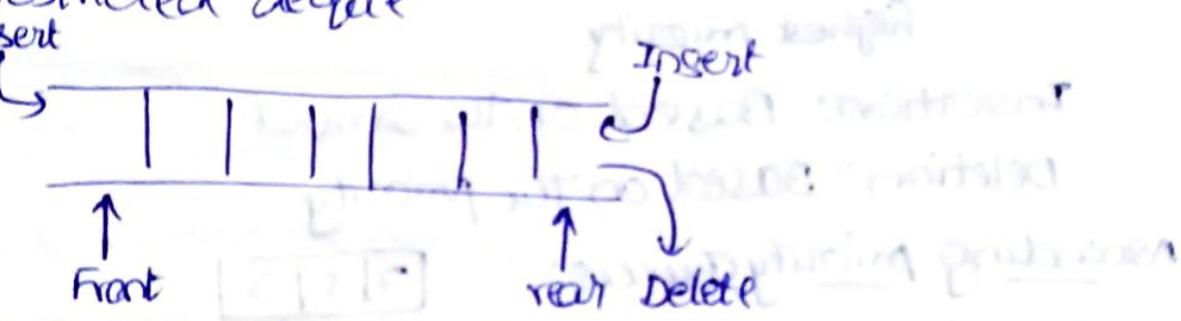
- * Input restricted deque
- * Output restricted deque
- * Input restricted deque

Insert



- * Output restricted deque

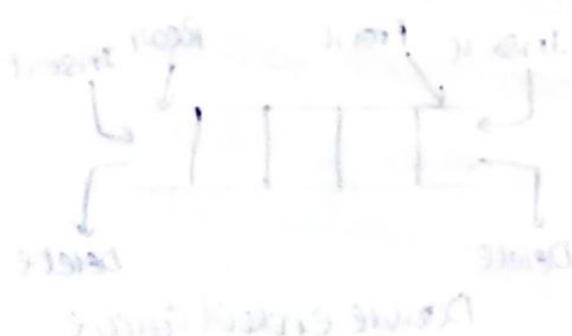
Insert



Operations performed on queue

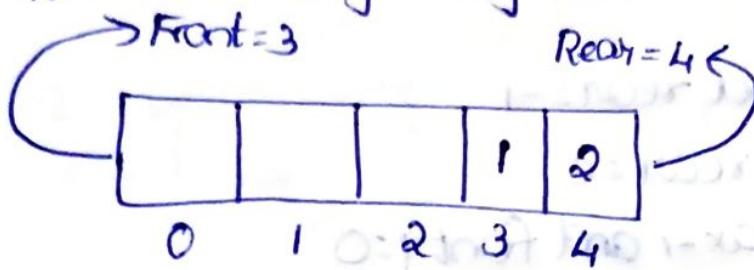
- ① Enqueue : insert at rear end
- ② Dequeue : delete from front end
- ③ Peek

Queue can be implemented through array and linked list



why circular queue introduced

- * one limitation
- * If the rear reaches to the end position of the queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized.



- * If we try to insert in such situation, we get "The queue is not empty", but the queue is empty.
- * To avoid such wastage of memory space, we need to shift elements to left for each update, but this process is resource costly.
The efficient approach for these situations is having a circular queue data structure.

* FIFO * Forms a circle * Ring Buffer

* last position is connected to the first position

Applications

- ① Memory Management
- ② CPU Scheduling
- ③ Traffic System

Complexity

- * Enqueue & dequeue operations of a circular queue is $O(1)$ for array implementations.

Insert an element in circular queue

Step_1: if $(\text{rear} + 1) \times \text{MAX} = \text{FRONT}$

 write "overflow"

 Goto Step4

[END OF IF]

Step_2: if $\text{front} = -1$ and $\text{rear} = -1$,

 set $\text{front} = \text{rear} = 0$

 else if $\text{rear} = \text{max} - 1$ and $\text{front} \neq 0$

 set $\text{rear} = 0$

 else

 set $\text{rear} = (\text{rear} + 1) \times \text{max}$

 end of if

Step_3: set $\text{Queue}[\text{rear}] = \text{val}$

Step_4: exit

Dequeue an element in circular queue

Step_1: if $\text{front} = -1$

 write "underflow"

 Goto Step4

end of if

Step_2: Set $\text{val} = \text{queue}[\text{front}]$

Step_3: if $\text{front} = \text{rear}$

 set $\text{front} = \text{rear} = -1$

else

 if $\text{front} = \text{max} - 1$

 set $\text{front} = 0$

 else

 set $\text{front} = \text{front} + 1$

 end of if

end of if

Step_4: exit

Deque (Double ended queue)

- * Deque is a linear data structure where the insertion and deletion operations are performed from both ends.
- * Doesn't follow the FIFO rule.

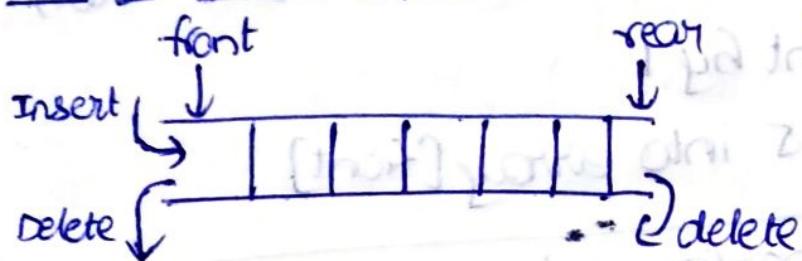


Deque

* we use this when we need

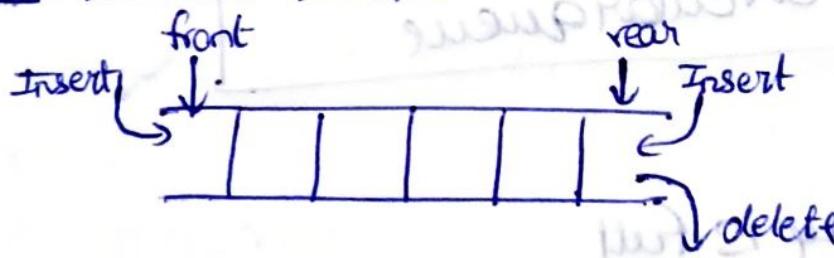
- ① input restricted queue quicker append and pop operations from both ends
- ② output restricted queue

"Input restricted" queue



* O(1) time complexity for append and pop from both ends

"output restricted" queue



* List performance action in O(n) time complexity

Applications of deque

- * Deque can be used as both stack and queue as it supports both operations.
- * Can be used as palindrome checker
- * In python, we can define maximum length of given dq using " maxlen " (we use this operation in page history)


```
four_numbers = deque([0, 1, 2, 3, 4], maxlen=4) # Discard 0
```

Deque's append(), appendleft(), pop(), popleft(), len(), operations are thread safe, so we could execute addition/removal of data from both ends of deque at the same time from separate threads, without a risk of data corruption.

Deque operations

Insert at front

- * Check the position of front
- * If $\text{front} < 1$, reinitialize $\text{front} = n - 1$ (last index)
- * Else, decrease front by 1
- * Add the new key 5 into array[front]

operations are performed assuming
the queue is circular queue

Insert at rear

- * Check if the array is full
- * If deque is full, reinitialize $\text{rear} = 0$
- * Else increase rear by 1
- * Add the new key to array[rear]

Delete from the front

- * check if deque is empty
- * If deque is empty deletion cannot be performed (underflow condition) (i.e. front = -1)
- * If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1
- * else if front is at the end (i.e. front = n-1), set go to front = 0
- * else, front = front + 1

Delete from the Rear

- * check the queue is empty or not
- * If the deque is empty (i.e. front = -1), deletion cannot be performed (underflow condition)
- * If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1 else follow the steps below
- * If rear is at the front (i.e. rear = 0), set go to the front rear = n-1
- * else, rear = rear - 1

check empty

* if front = -1, the deque is empty

check full

* (If front = 0 and rear = n-1) or (front = rear + 1)

Deque can be used both as stack and queue as it supports both FIFO and LIFO operations.

Priority Queue

- * Priority Queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority.
- * Highest priority will come to the first of the queue. It will be removed first.
- * Allows only comparable elements, arranged either in ascending order / descending order.

P. Queue

1	3	4	8	14	22
↑				↑	

HP
LP

- * highest priority will be deleted first than the lowest ones
- * If two elements in priority queue are of same priority then they will be arranged in FIFO principle.

Example:

1	3	4	8	14	22
---	---	---	---	----	----

poll() \Rightarrow removes 1

add(2) \Rightarrow add 2 in front of 3 (asc order)

2	3	4	8	14	22
---	---	---	---	----	----

poll() \Rightarrow removes 2

add(5) \Rightarrow adds in b/w 4 & 8 (asc order)

3	4	5	8	14	22
---	---	---	---	----	----

Types

① Ascending order priority queue

- * lower value given highest priority

2	6	7	10	11
↓			↓	

HP LP

② Descending order priority queue

10	9	8	7	6
↓			↓	

HP LP

High PN = higher Priority

↳ DSC PQ

Lower priority number = highest priority. → ASPOQ

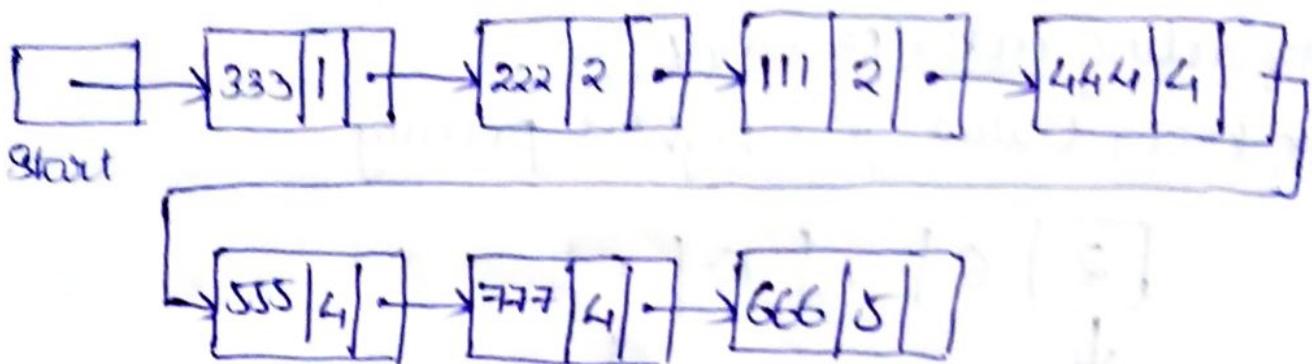
Priority queue's implementation

① arrays ② linked list ③ heap data structure

④ Binary Search tree

Examples

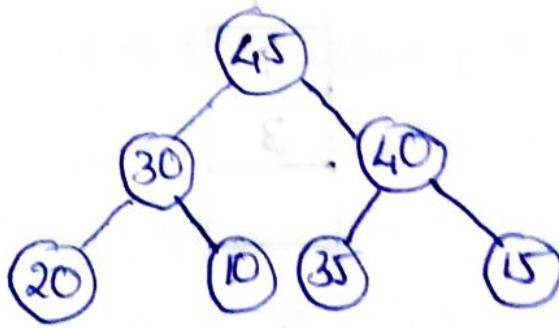
	INFO	PRN	LINK/NEXT
location 1	222	2	6
2			
3	444	4	4
4	555	4	9
5	333	1	1
6	111	2	3
7			
8	666	5	
9	777	4	
10			8



Complexity analysis

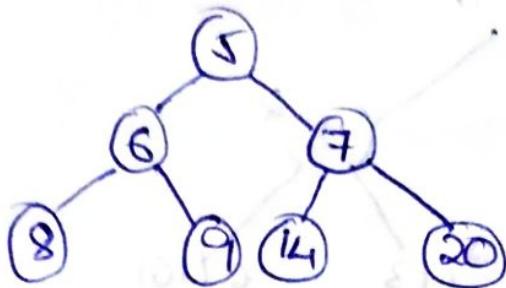
Implementation	add	remove	peek
Linked list	$O(n)$	$O(n)$	$O(n)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$
Binary searchtree	$O(\log n)$	$O(\log n)$	$O(1)$

- * Priority queue implementation through heap is efficient
- * Heap is tree based data structure that forms a complete binary tree and satisfies heap property.
- * value of the root node is either greater than the whole child nodes or it is lesser than the whole child nodes
- * Based on this Heaps are divided into two types
→ Max heap: value of parent nodes > value of child nodes



Each node will have either 0 or 2 nodes as child, so this is binary heap

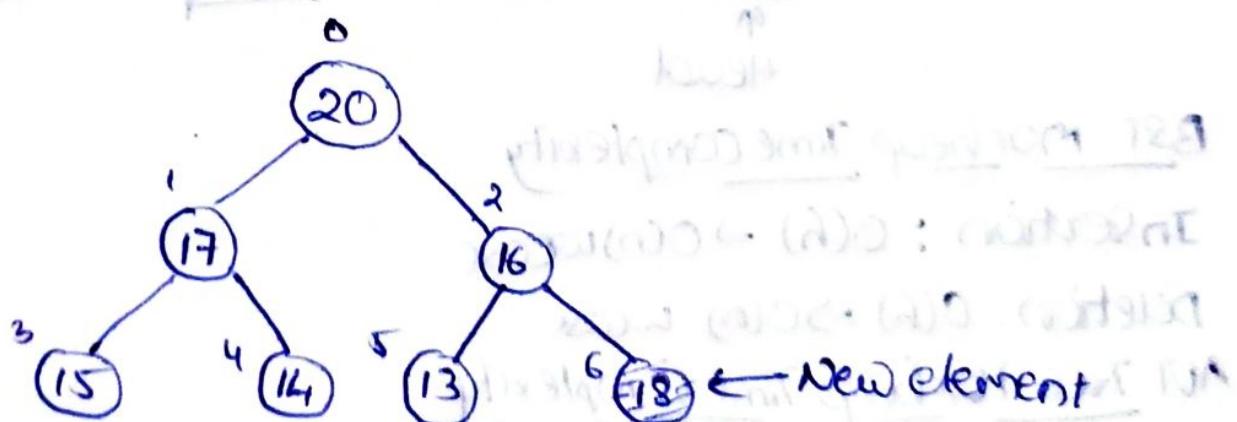
→ Min heap: value of parent node is less than value of child nodes



Operations on priority Queue

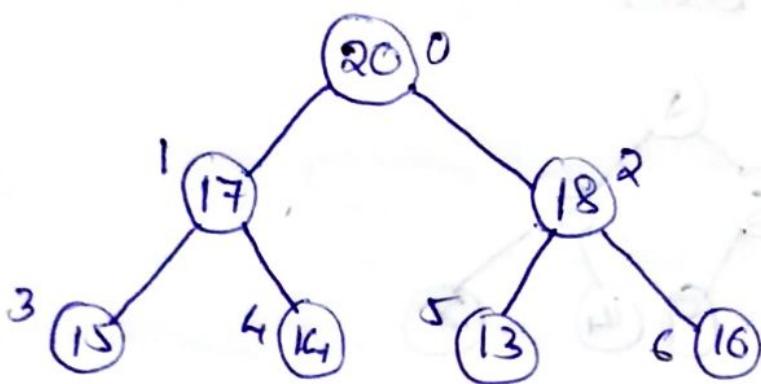
- * Inserting an element in priority queue (max heap)
 - If we insert an element in priority queue, it will move to the empty slot by looking from top to bottom and left to right
 - If the element is not in a correct place then it is compared with the parent node, if it is found out of order, elements are swapped. This process continues until the element is placed in correct position

Example



- New element is inserted at the end of the heap, where it can find the first empty place. Later insertion it checks if it is lesser than its parent (property of max heap). If it follows max heap, else swap nodes, do this until the heap follows max heap.

MaxHeap



Time complexity of Priority Queue (FIFO) array (MaxPQ)

Insertion: adding element last of PQ array $\Rightarrow O(1)$

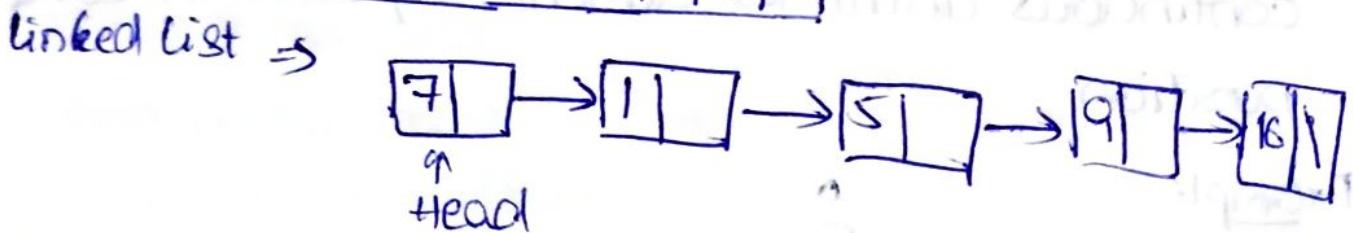
Deletion: deleting element first of PQ array and shifting $n-1$ elements to left $\Rightarrow O(n-1) \Rightarrow O(n)$

Time Complexity of PQ (FIFO) Linked List (maxPQ)

Insertion: adding a node at first $\Rightarrow O(1)$

Deletion: deleting a node at last $\Rightarrow O(n)$

array \Rightarrow [16 | 3 | 5 | 6 | 9 | 2 | 11 | 4 | 7]



BST Maxheap Time complexity

Insertion: $O(h) \Rightarrow O(n)$ worst case

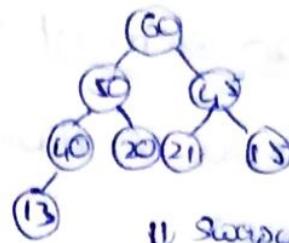
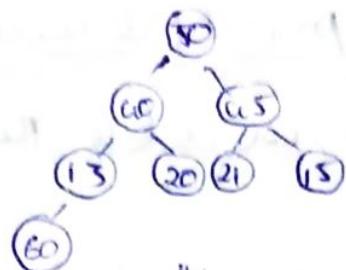
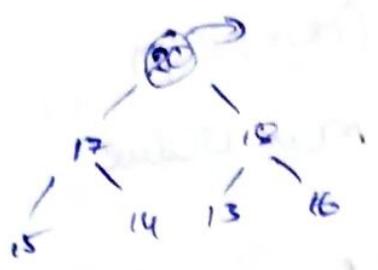
Deletion: $O(h) \Rightarrow O(n)$ worst

AVL Tree Maxheap Time Complexity

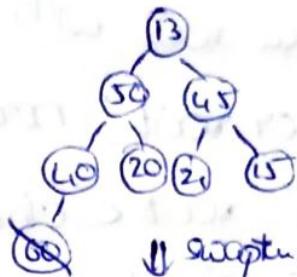
Balanced tree

Insertion: $O(h) \Rightarrow O(\log n)$

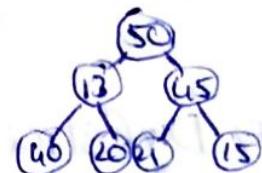
Deletion: $O(h) \Rightarrow O(\log n)$



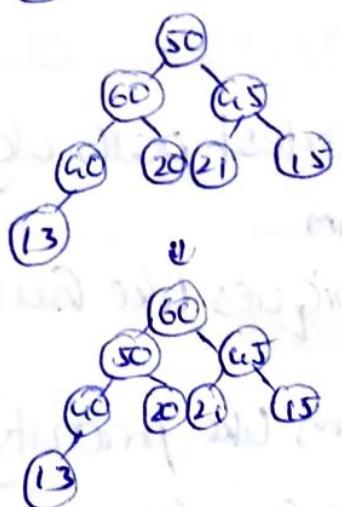
ii swap with last element



ii swap with root with max ele of its children

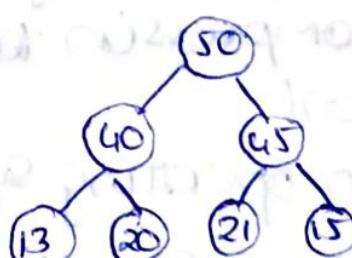


ii



Insertion

O(logn)



Deletion

* O(logn)

Deleting an element from Priority Queue (Max heap)

- * as we know in max heap, we delete the max value that is root
- * replace the root with the last inserted value, and remove the earlier root, now swap the current root with greater child if $\text{root} < \text{children}$, if we do this iteratively we get a new tree which follows max heap.

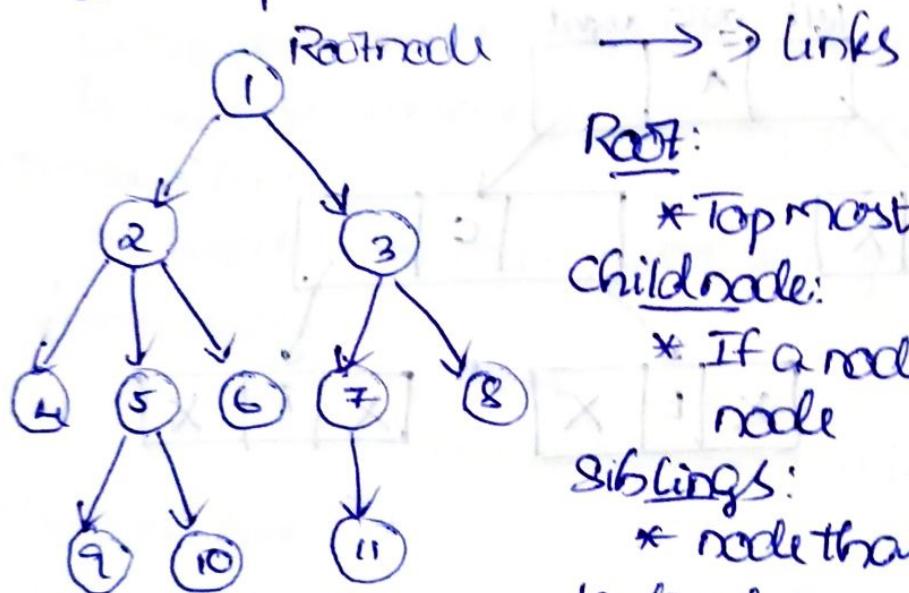
Applications

- * used in Dijkstra's shortest path algorithm
- * used in prim's algorithm
- * data compression techniques like Huffman coding
- * heap sort
- * used in operating systems like priority scheduling, load balancing and interrupt handling

Tree data structure

- * Linear data structure like an array, linked list, stack and queue in which all the elements are arranged in sequential manner.
- * Tree is also one of the data structure that represent hierarchical data.
- * Root is at top, Trees are efficient to store elements in hierarchical way.
- * Non linear data structure

Tree Example:



→ Links

Root:

- * Top most node in tree hierarchy

Child node:

- * If a node is descendant of any node

Siblings:

- * node that have same parent

Leaf node:

- * node which doesn't have any descendants

Internal node:

- * node atleast have one child node

Ancestor node:

- * Predecessor node on path from root to that node.

- * 1, 2, 5 are ancestor nodes of node 10

Descendant

- * immediate successor of the given node

Properties of tree data structure

- * Recursive data structure
- * Number of edges

↳ If we have n nodes $\Rightarrow n-1$ edges

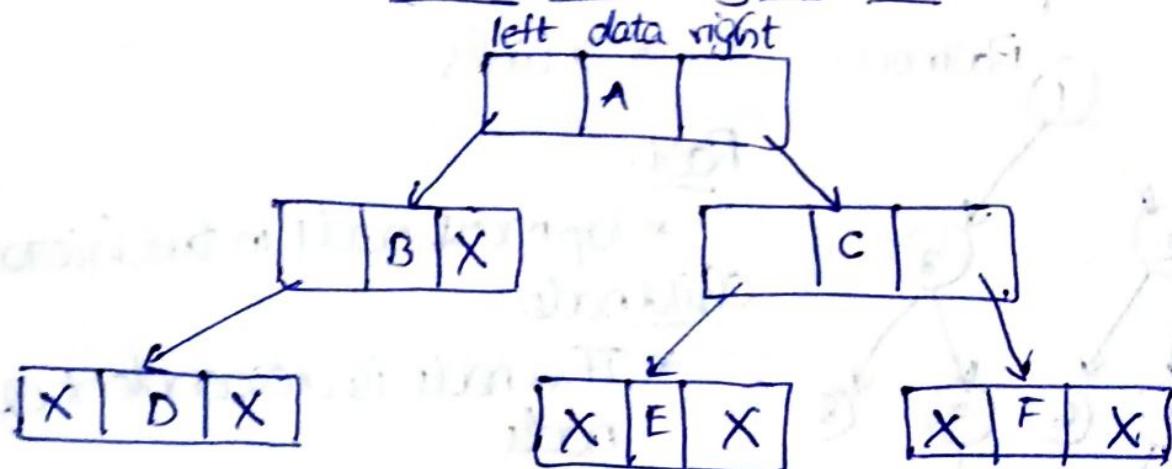
- * Depth of node x

↳ length of the path from the root to node x

- * Height of node x

↳ longest path from the node x to the leaf node

Tree in the memory looks like



node structure

struct node

{

 struct node *left;

 int data

 struct node *right;

}

Field 1 stores address of left node

Field 2 consists of data of the node

Field 3 consists of address of right node

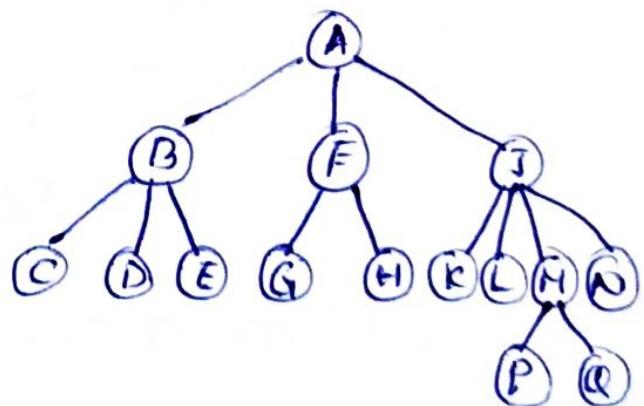
Applications of trees

- * Storing naturally hierarchical data
 - ↳ Ex: File System (Files, Folders)
 - * Organize data
 - ↳ Efficient insertion, deletion, searching only possible in trees.
 - ↳ $\log N$ time for searching an element.
 - * Trie
 - ↳ Special kind of tree, used to store the dictionary.
 - ↳ Fast and efficient for dynamic
 - * Heap
 - ↳ Tree data structure implemented using arrays.
 - ↳ used to implement priority queues
 - * B-Tree & B+tree
 - ↳ Tree data structure used to implement indexing in DB.
 - * Routing table
-

Tree data structure types

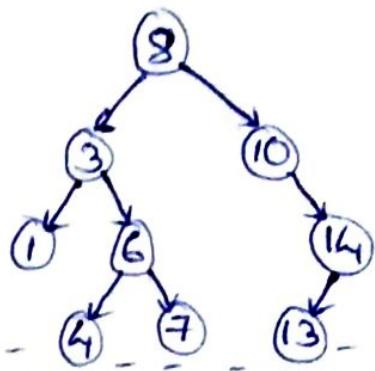
① General Tree

- * A node can have either 0 or max n number of nodes.
- * No restriction imposed on degree of node
- * Subtrees are unordered as they don't have any pattern in common.



Binary tree

- * Each node in a tree can have utmost two child nodes. (0/1/2)



AVL tree

- * Variant of the binary search tree.
- * satisfies both properties of binary tree and as well as binary search tree.
- * self balancing tree
- * Balancing the height of left subtree and right subtree based on balancing factor.

* Balancing factor means difference b/w the height of left subtree and height of right subtree

* It should be either 0, -1, +1

Splay tree

- * BST where recently accessed element is placed at root position of tree by performing some rotation operation.

* No explicit balance condition like AVL tree.

* Splay tree is a balanced tree but we can't consider that as height balanced tree.

Red Black tree

- * It is binary search tree
- * Binary search average case $\log_2 n$
- Bestcase $O(1)$
- Worstcase $O(n)$
- * It is self balancing binary search tree.
- * Number of rotations are predefined. max of 2 rotations.
- * Each node is of Red or Black
- * Similar to AVL tree but here we have color to node to balance

Treaps

- * Tree + heap
- * Binary search tree properties with respect to keys (following tree) property
- * Each node have its key & priority

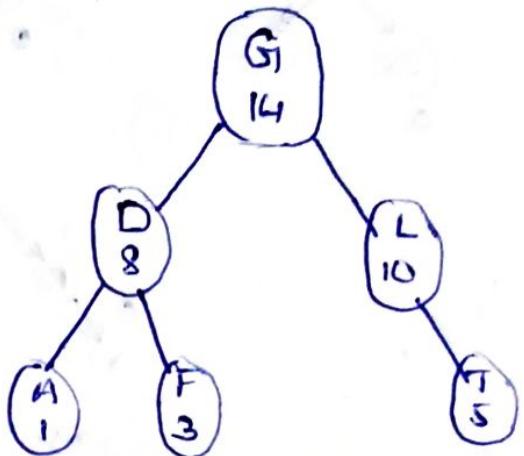
→ BST key larger than all keys in left subtree
 → BST key smaller than all keys in right subtree

- * Heap property wrt priority

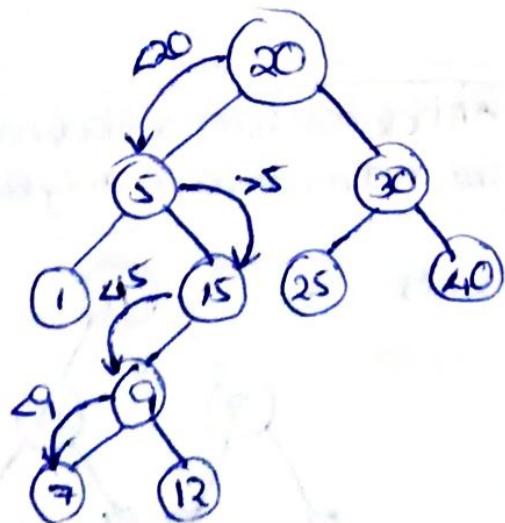
Top node will be of larger priority than below

- * Each node of Treap will have its corresponding (key, priority)

Keys: BST
 Priority: heap



Binary Search tree example

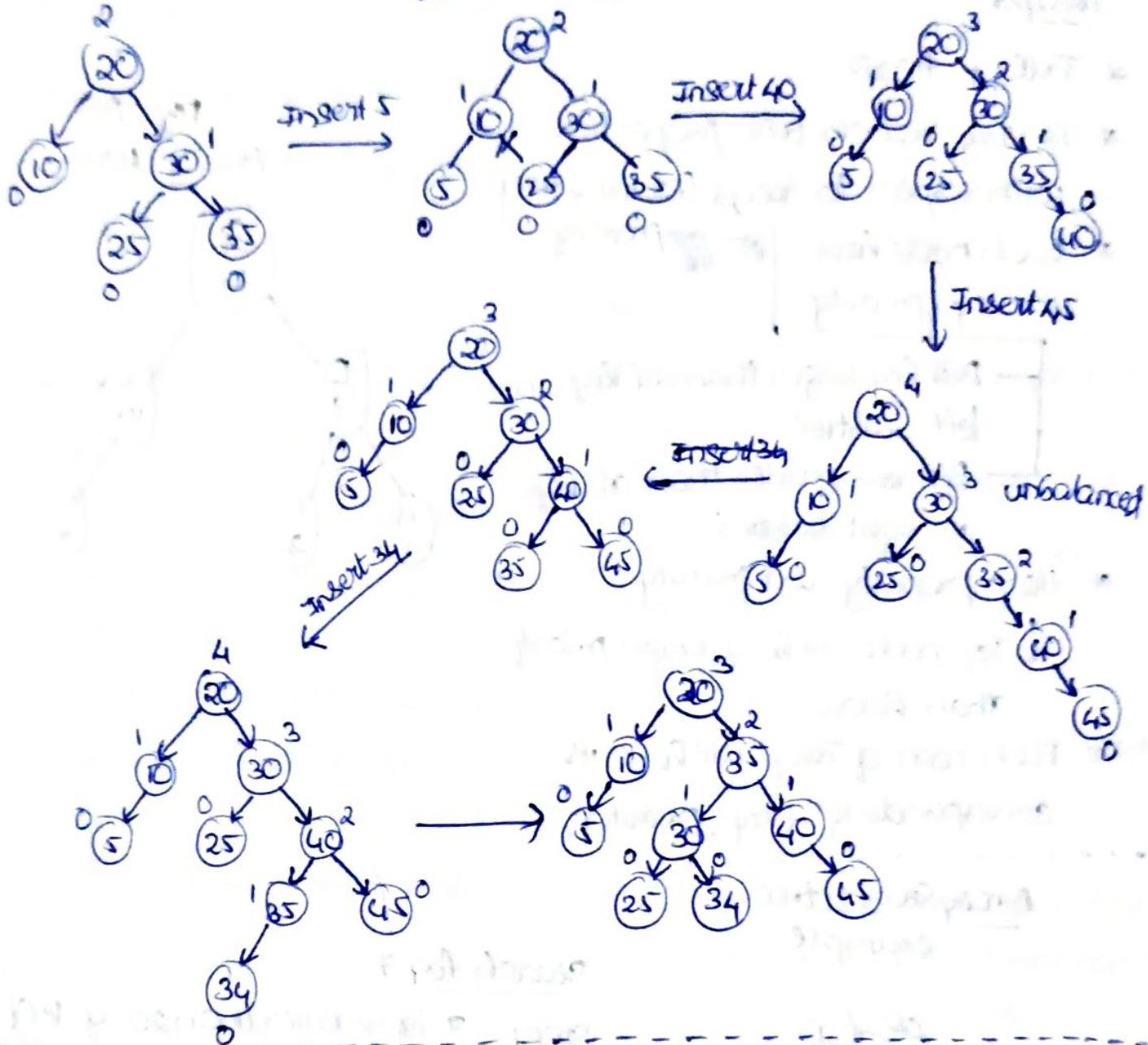


Searching in BST

search for 7

- Step 1: 7 is smaller than 20, go left
- Step 2: 7 is greater than 5, go right
- Step 3: 7 is smaller than 15, go left
- Step 4: 7 is smaller than 9, go left

Insertion in AVL tree



Red Black tree

A black can have a black node as next
but red must have only black as next

- * A self balancing binary search tree

- * Each node has extra bit, which represents its color (red or black)

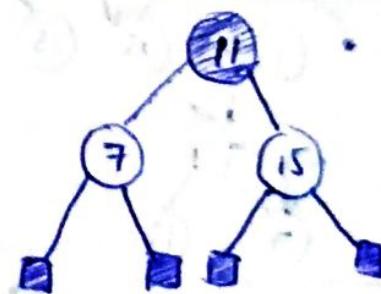
- * several properties

- ① Root property: Root should be black

- ② External property: Every external node is black

- ③ Red property: children of red node are black

- ④ Depth property: All external nodes have same black depth.



If any of these property failed, then it won't be a red black tree

Searching, Insertion, Deletion $\Rightarrow O(\log n)$

Properties of Binary tree

① maximum number of nodes at level l is 2^{l-1}

$$\text{max nodes } 2^{l-1} = 1 \quad l=1 \Rightarrow$$

$$\text{max nodes } 2^{l-1} = 2^{2-1} = 2 \quad l=2 \Rightarrow$$

$$\text{max nodes } 2^{l-1} = 2^{3-1} = 4 \quad l=3 \Rightarrow$$

$$\text{max nodes } 2^{l-1} = 2^{4-1} = 8 \quad l=4 \Rightarrow$$

② total number of nodes present in binary tree

For example,

For a binary tree with which height 3, we will have max of 2^{h-1} nodes.

$$\sum_{l=1}^{l=h} 2^{l-1} = 2^0 + 2^1 + 2^2 + \dots + 2^{h-1} \quad N=3$$

$$\boxed{\sum_{l=1}^{l=h} 2^{l-1} = 2^h - 1}$$

$N=1$

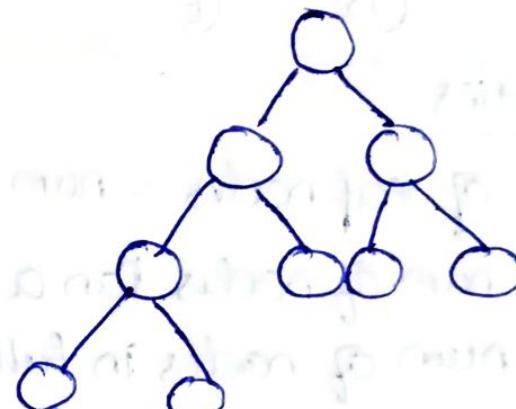
$N=2$

$N=3$

$N=4$

$N=5$

$N=6$



③ In a binary tree with N nodes, minimum possible height or minimum number of levels is $\log_2(N+1)$

$2^h - 1 \Rightarrow$ Max num of nodes can be present in a binary tree

$$2^h - 1 = N$$

$$2^h = N + 1$$

$$\boxed{h = \log_2(N+1)}$$

④ Number of leaf nodes is equal to number of internal nodes plus 1

⑤ Minimum number of nodes in the full binary tree $2 \times h - 1$

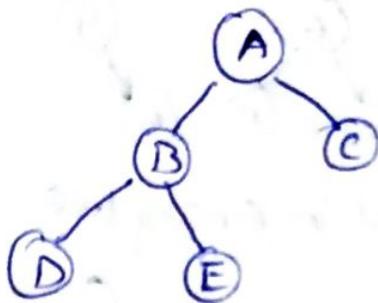
⑥ Maximum height of the full binary tree can be computed as

$$h = \frac{N+1}{2}$$

Types of Binary tree

① Full / Proper / Strict Binary tree

- * Each node must contain either 0 or 2 children.
- * Each node must contain 2 children except leaf nodes



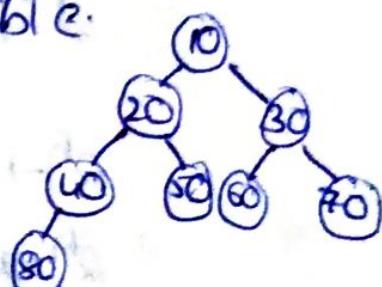
Properties

- * num of leaf nodes = num of internal nodes + 1
- * max num of nodes in a binary tree can have $\geq 2^h - 1$
- * min num of nodes in full binary tree $\Rightarrow 2^h - 1$
- * min height of full binary tree $\Rightarrow \log_2(n+1)$
- * max height of full binary tree $\Rightarrow h = \frac{n+1}{2}$

② Complete Binary tree

- * CBT is a tree in which all nodes are completely filled except the last level.
- * All the nodes must be as left as possible e.g.
- * nodes should be added from left

Properties



- * max num of nodes $\geq 2^h - 1$
- * Tree can contain at most 2^l nodes at level l
- * m nodes at level l, $2m$ nodes at level l+1
- * binary tree of depth d that contains exactly 2^d nodes at level d.

* Total num of nodes in CBT of depth d, 'S'

$S = \text{the sum of the number of nodes at each level from } 0 \text{ to } d.$

$$S = 2^0 + 2^1 + 2^2 + \dots + 2^d$$

$$S = \sum_{j=0}^d 2^j = 2^{d+1} - 1$$

$$\boxed{S = 2^{d+1} - 1}$$

* If we are having 2^d leaves in CBT, we will be having $2^d - 1$ non-leaves

* Finding depth based on total number of nodes

$$S = 2^{d+1} - 1$$

$$2^{d+1} = S + 1$$

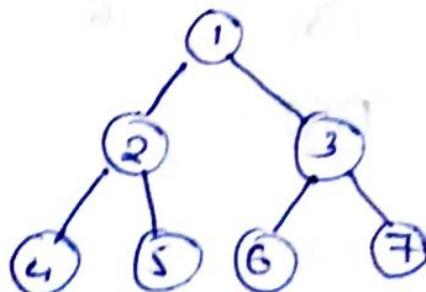
$$\log_2^{d+1} = \log(S+1)$$

$$d+1 = \log(S+1)$$

$$\boxed{d = \log(S+1) - 1}$$

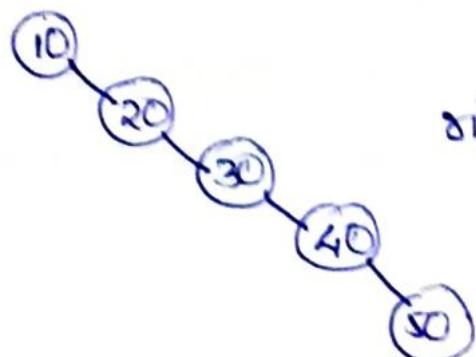
Perfect Binary Tree

* PBT if all internal nodes have 2 children, and all leaf nodes are at the same level.

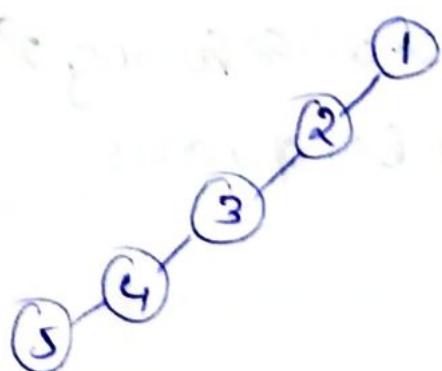


Degenerate Binary Tree

- * All internal nodes have only one children



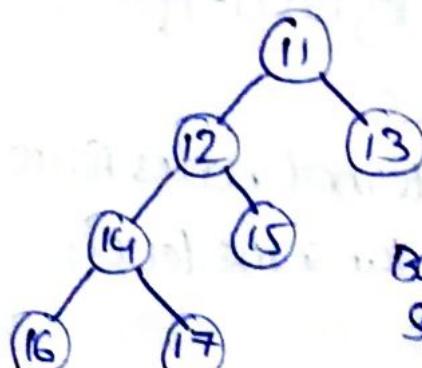
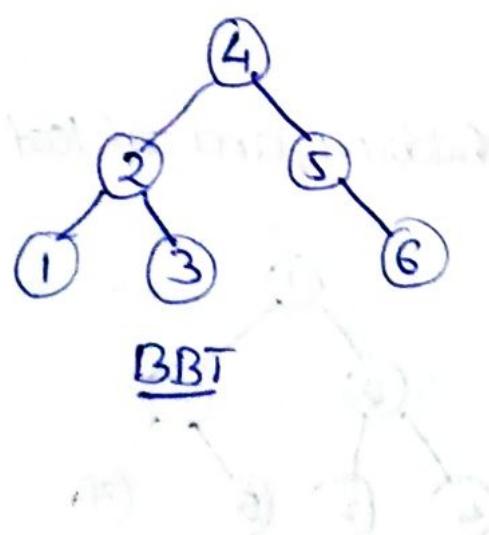
right skewed tree



left skewed tree

Balanced Binary Tree

- * Tree in which both left and right trees differ by at most 1 height.
- * Examples: AVL trees & Red Black tree

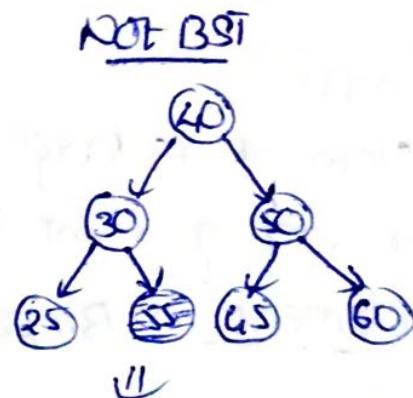
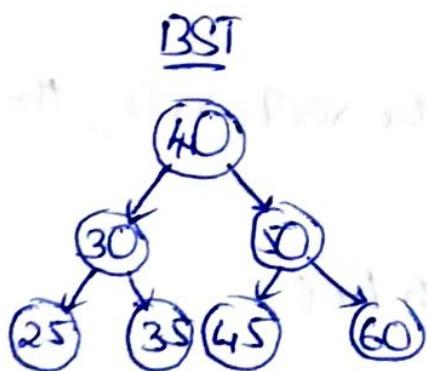


Balance Factor is > 1
So, not a BBT

Not BBT

Properties of binary search tree

- ① no duplicate elements in binary search tree
 - ② element at the left child of node is always less than the element at the current node
 - ③ left subtree of a node has all elements less than the current node.
 - ④ Element at the right child of a node is always greater than the element at the current node.
 - ⑤ Right subtree of node has all elements greater than the current node.
- If any of the above property not satisfied by any binary tree then , we can conclude it is not binary search tree.



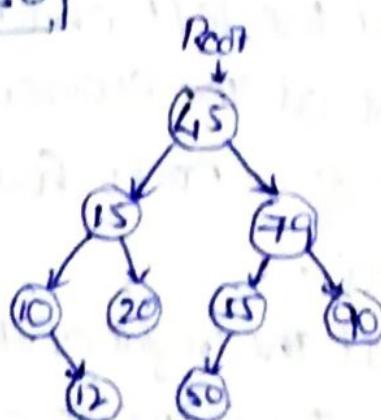
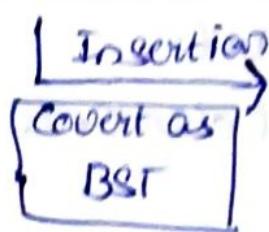
Advantages of BST

- * Searching in BST is easier, as for each and every node, half search tree will be decreased.
- * Insertion, deletion are faster in BST.

$40 < 55 \Rightarrow$ actually BST mean values of left subtree must be lesser than the root node . Here its the opposite, so not BST

Answe

45		15		79		90		10		55		12		20		50
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----



Insertion process

- * Insert first element of array as root
- * Then, read next element, if it is smaller than the root node, insert it as the root of left subtree, if we are having a root already traverse the nodes of BST until you get a leaf node to insert a new node
or
suitable node
- * If the element is larger than the root node, then insert it as the root of right subtree

Search an element in Binary Search tree

Search(root, item)

Step 1 \Rightarrow if (item == root \rightarrow data) or (root == null)

 return root

 elseif (item < root \rightarrow data)

 return Search (root \rightarrow left, item)

 else

 return Search (root \rightarrow right, item)

 endif

Step 2 \Rightarrow END

Deletion in Binary Search tree

3 cases in Deleting a node from a tree

① node to be deleted is a leaf node

② node to be deleted has only child

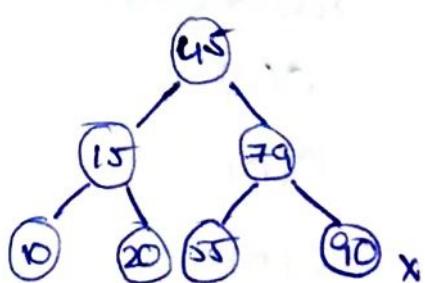
③ node to be deleted has two children

① node to be deleted is a leaf node

we have to replace the leaf node with null and simply free the allocated space.

② node to be deleted has only child

In this case, we have to replace the target node with its child , now the node which we want to delete will be at child node position. And now we need to delete the child node. Now we simply have to replace the child node with NULL and free up the allocated space.



case 1: Delete leaf node

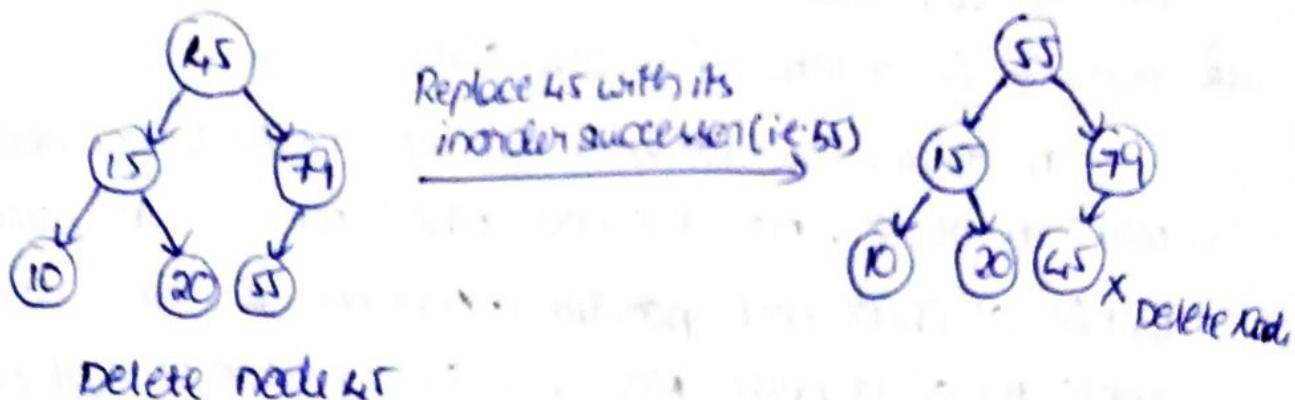


case 2: Delete node which have one child

I Delete 79 \Rightarrow swap 79 with its child node and delete the new child.
i.e: 79 node

When the node to be deleted has two children

- ① First, find the inorder successor of the node to be deleted.
- ② After that, replace that node with the inorder successor until the target node is placed at leaf of tree.
- ③ And at last, replace the node with NULL and free up the allocated space.



	<u>Time complexity</u>	
	<u>Best/Average case</u>	<u>worst case</u>
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$
search	$O(\log n)$	$O(n)$

* where n is number of nodes in given tree

Space complexity

Insertion | Deletion | Search $O(n)$

* Inorder traversal of binary search tree gives us a list of elements in ascending order. After insertion/deletion, the BST should be in ascending order in inorder traversal.

AVL Tree

- * named after its founders "GM Adel'son-Velsky and EM Landis" in 1962.
- * Height Balanced binary search tree (IMP)
- * Each node is associated with balance factor which is calculated by subtracting height of its right subtree from that of its left subtree.
- * Tree is balanced if the balance factor of each node is in between -1 to 1. Else tree is unbalanced or need to be balanced.

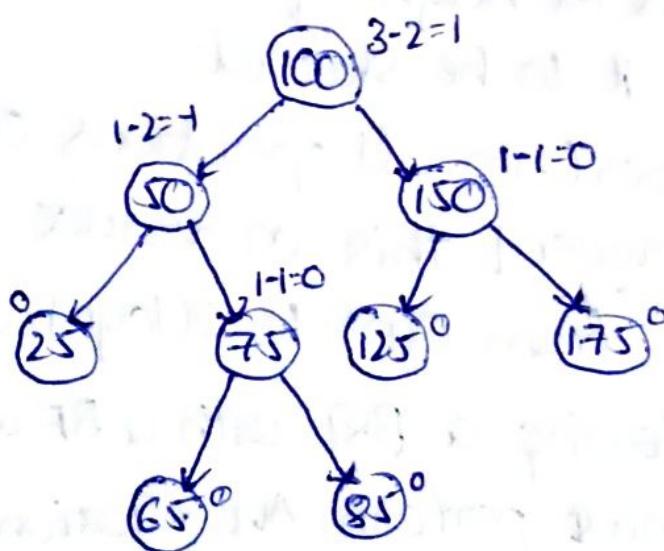
Balance Factor (K) = height(left(K)) - height(right(K))

1 \Rightarrow left subtree is one level higher than the right subtree

0 \Rightarrow left subtree and right subtree are of same height

-1 \Rightarrow left subtree is one level lower than the right subtree

Example



AVL Tree

AVL Tree Time Complexity

Algorithm

Average & worst case T.C

Space

$O(n)$

Search

$O(\log n)$

Insert-

$O(\log n)$

Delete

$O(\log n)$

- * All other operations of BST is similar to AVL tree, but insertion and deletion are different in some ways
- * Both insertion and deletion follow same process as BST, but after ins or del, we need to rebalance the tree, to satisfy properties of AVL tree.

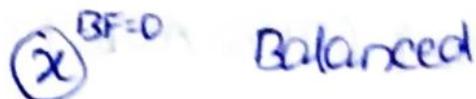
why AVL tree?

- * AVL tree controls the height of binary search tree by not letting it to be skewed.
- * BST takes to perform all operations $O(n)$ in worst case. So to prevent this or reduce or optimize the worst case from $O(n)$ to $O(\log n)$ we use AVL tree. Whenever, we are having a BST with a BF which is not $\{-1, 0, 1\}$ then we must perform AVL rotations to balance the BST to make AVL tree.

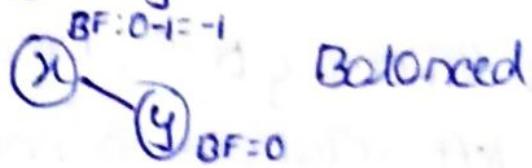
Example 1:

consider 3 nodes x, y, z where $x < y < z$, and lets build a tree, by following rules of AVL tree, it internally should follow properties of Binary Search tree.

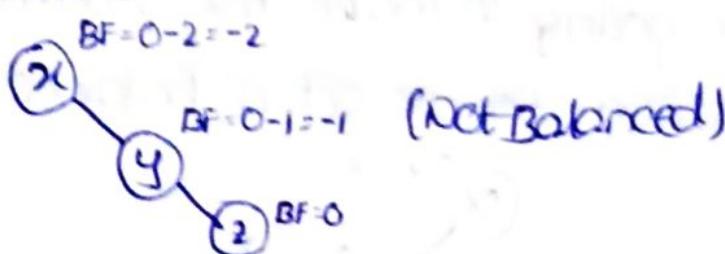
Step 1 : insert x



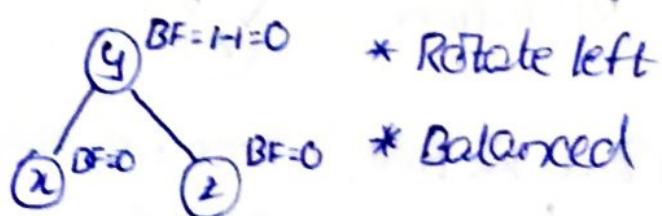
Step 2: insert y



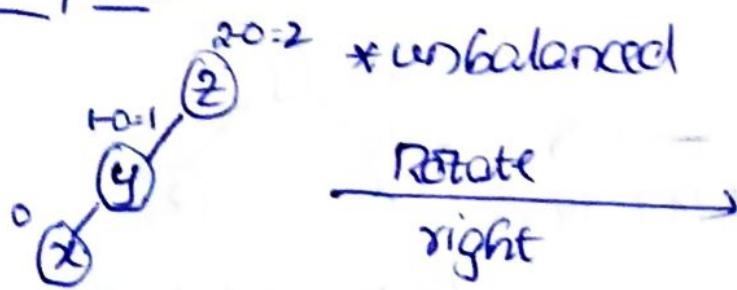
Step 3: insert z



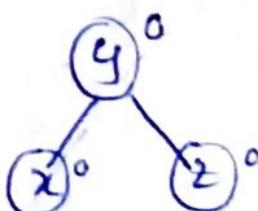
↓ Rotate to make it Balance



Example 2:

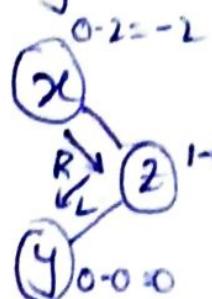


→
 Rotate right



Example 3:

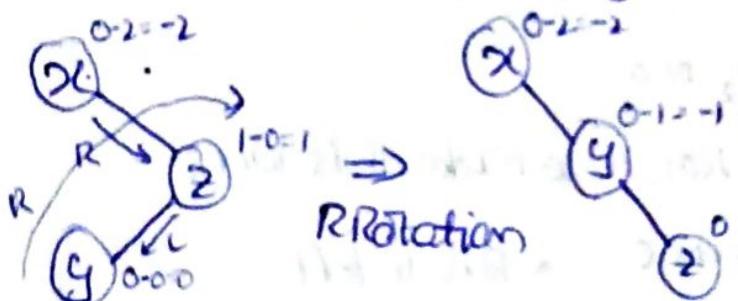
Suppose, lets consider x, z, y rotation, where we know $x < y < z$, then the tree would become like the following



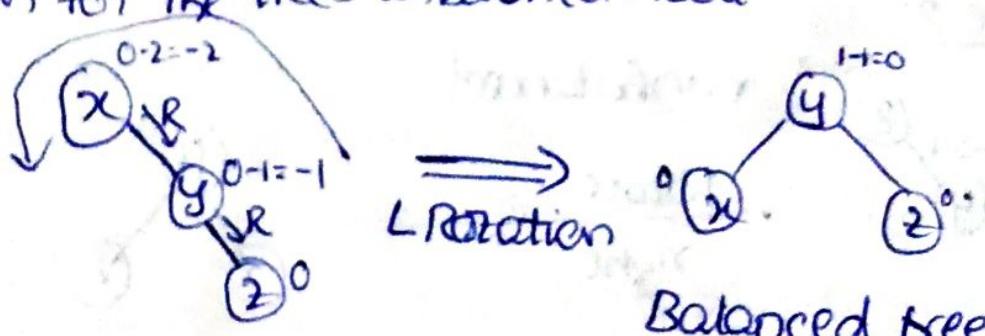
* Follows BST

* Doesn't follow AVL tree

As we see to get this following tree, we done Right rotation and then do a left rotation. So now, to make this tree we are going to make the subtree of unbalanced node into RL Rotation. So we get a balanced tree.



Now we got a RR tree, where we must perform left rotation for the tree's unbalanced node.

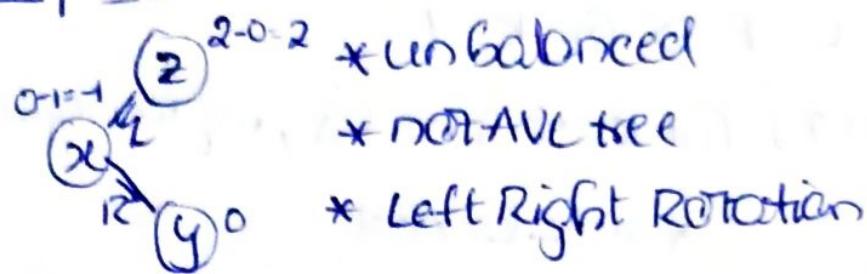


Balanced tree

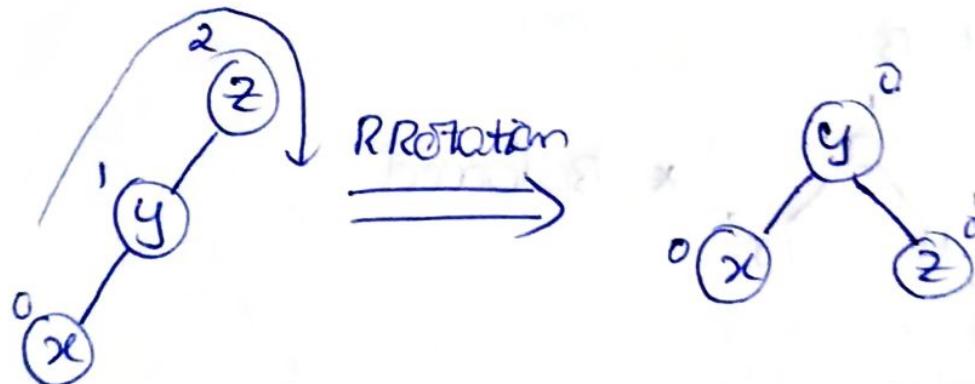
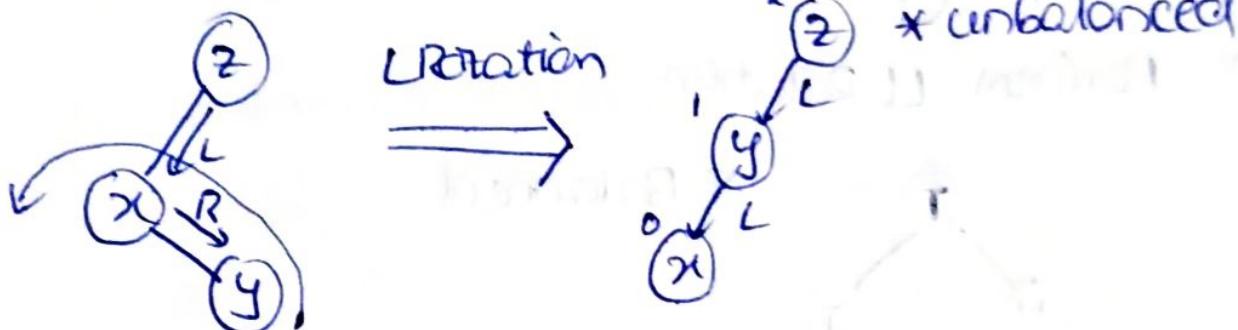
AVL tree

Binary Searchtree

Example 4: $z \times y$



Reform LR Rotation



when tree is unbalanced because of

RR \implies L Rotation

LL \implies R Rotation

RL \implies R_{subtree} L_{full tree under unbalanced node} Rotation

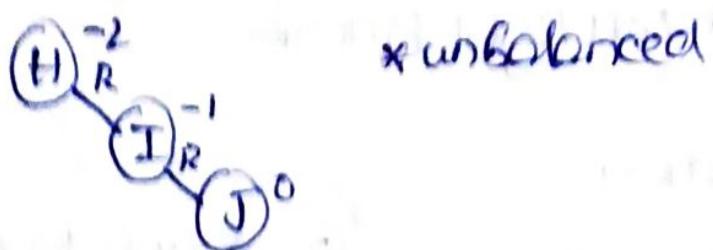
LR \implies L_{subtree} R_{unbalanced node's tree} Rotation

These are possible 4 cases in AVL tree

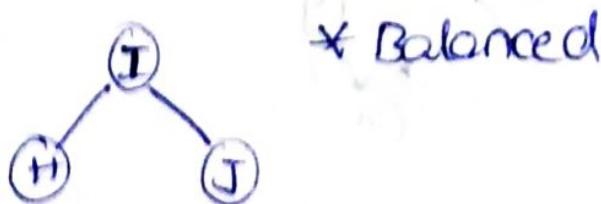
AVL tree example

* H I J B A E C F D G L K

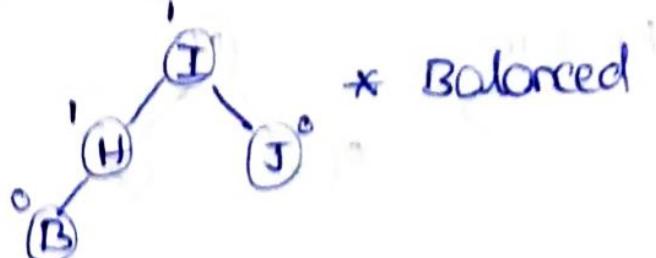
Step_1: Insert H I J (at least 3 nodes, req to negate BF)



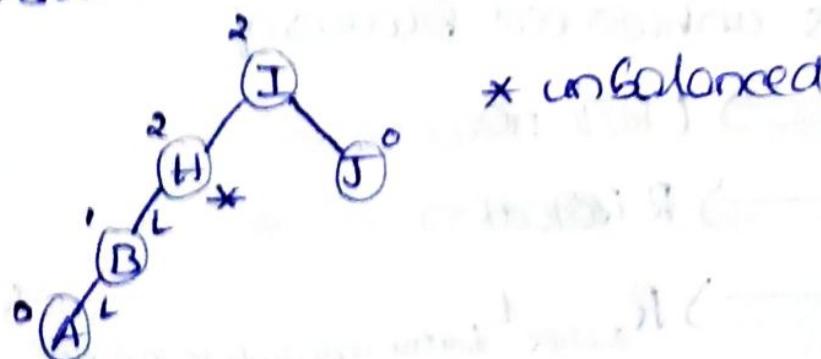
Step_2: Perform LL Rotation



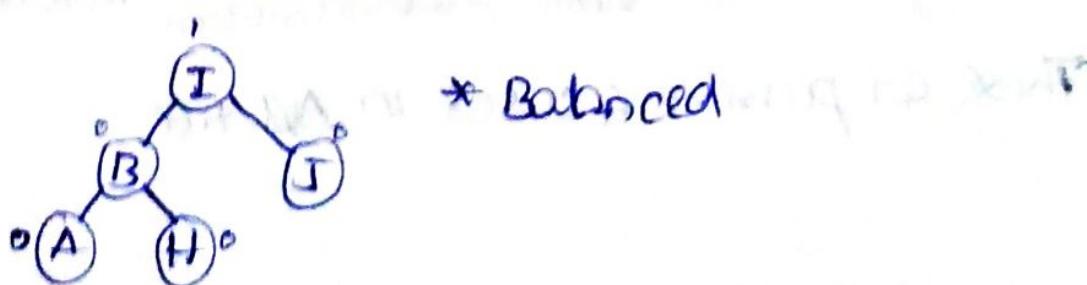
Step_3: Insert B



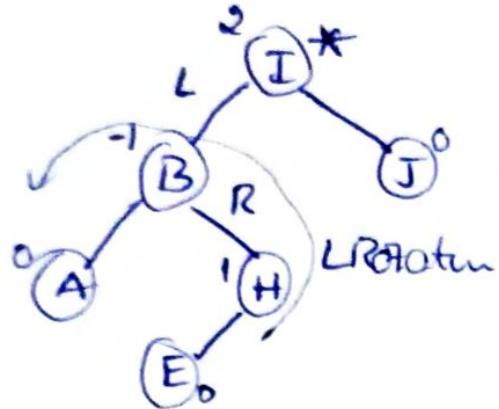
Step_4: Insert A



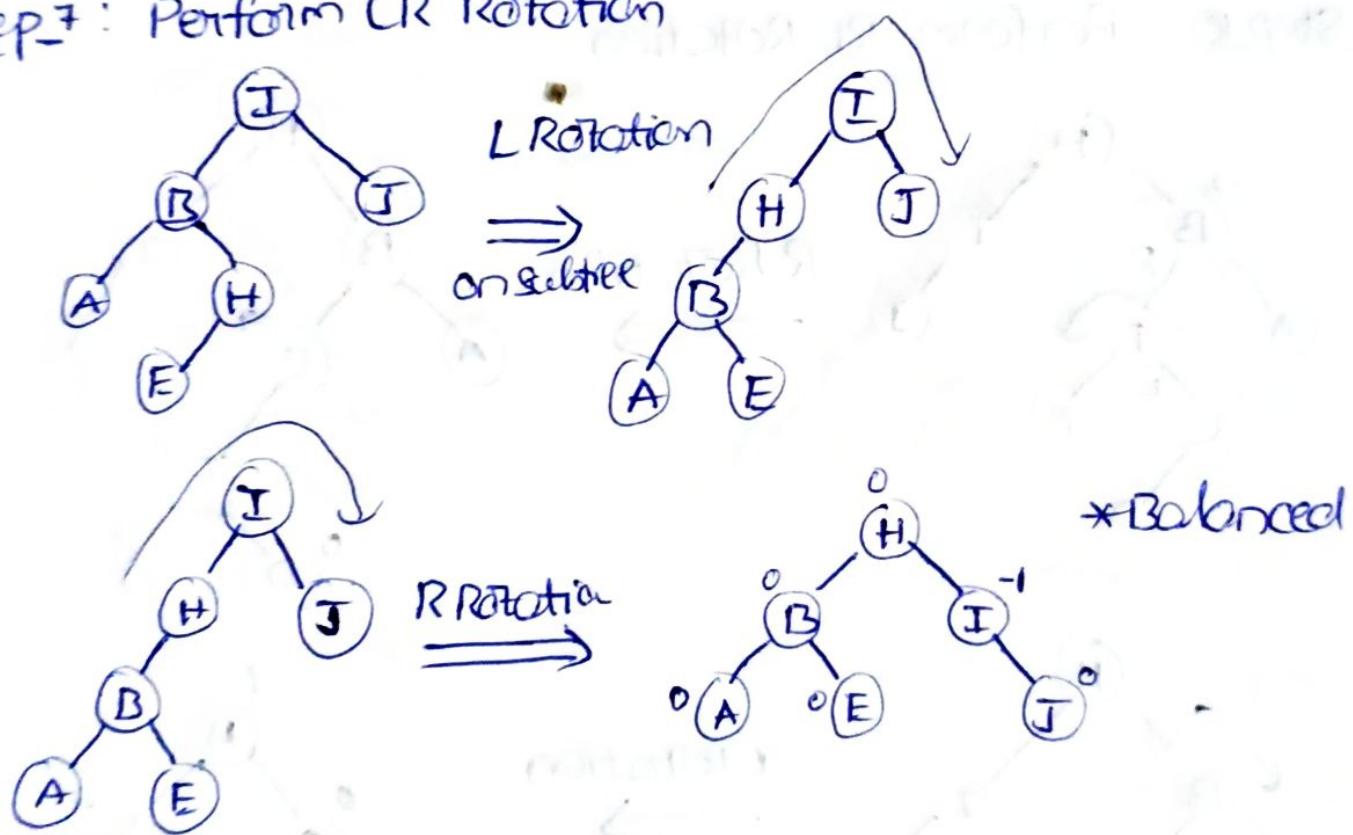
Step_5: Perform RR Rotation



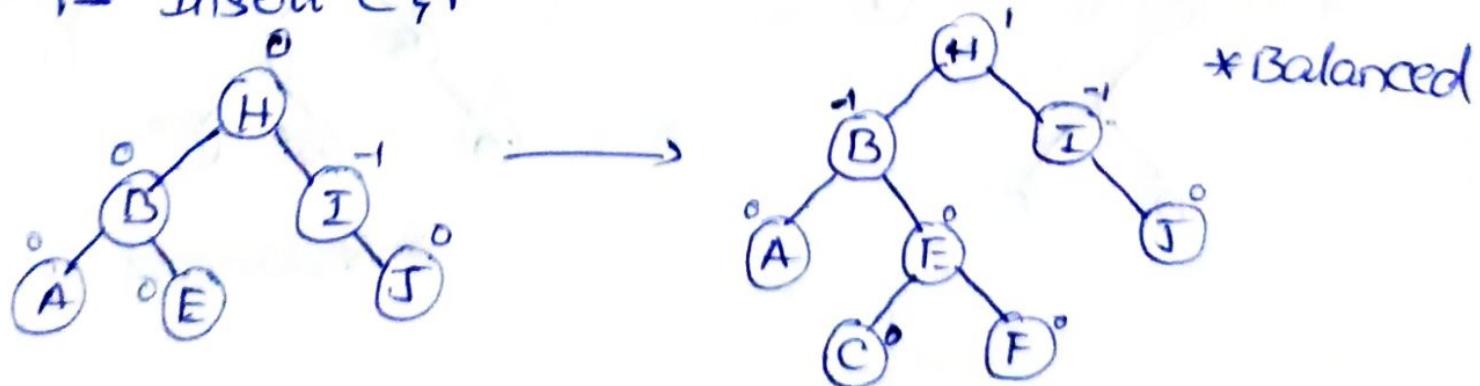
Step 6: Insert E



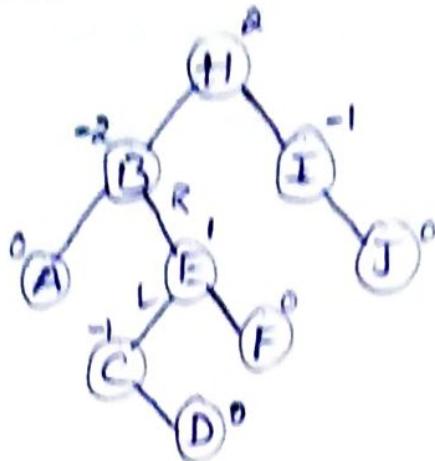
Step 7: Perform LR Rotation



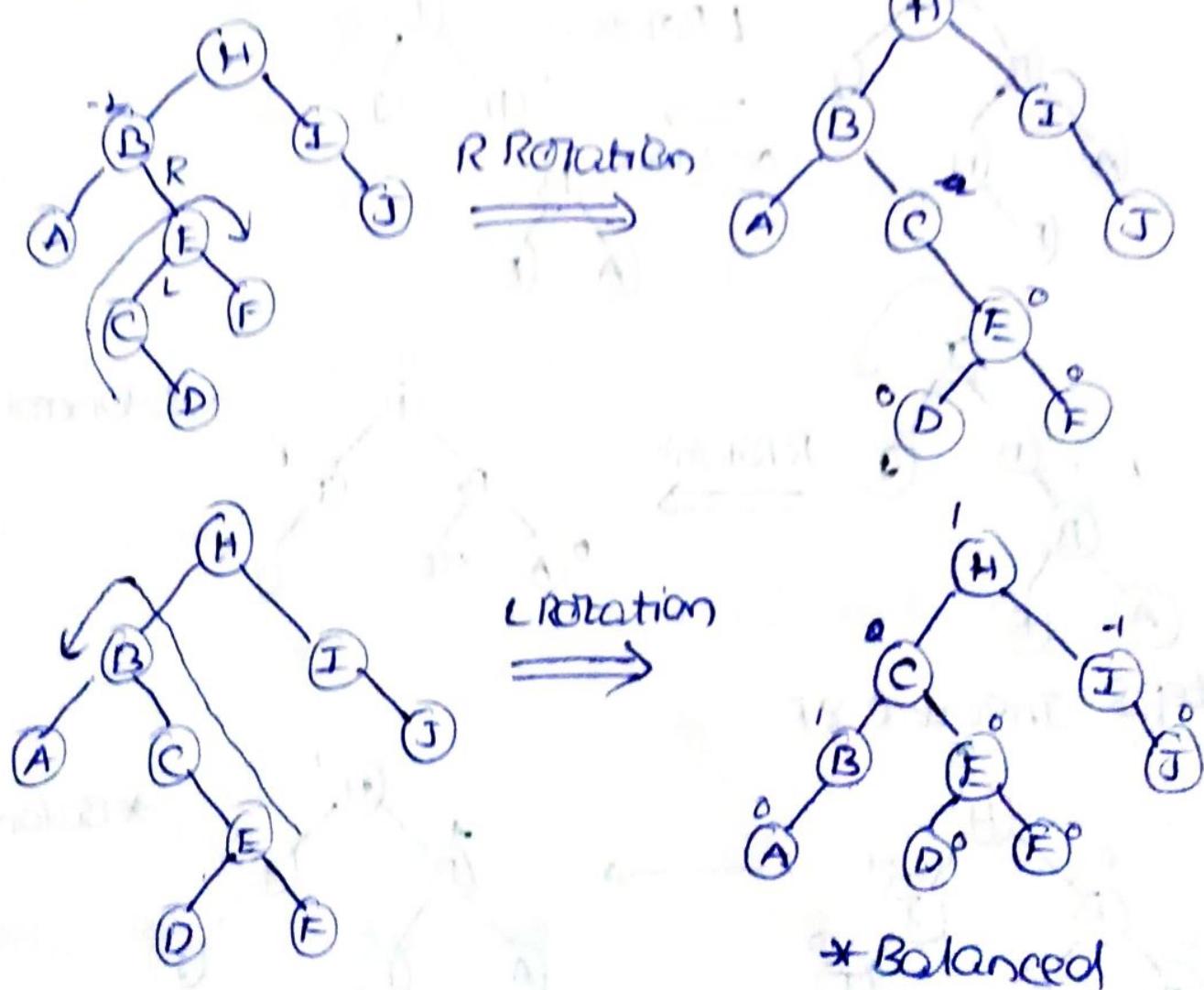
Step 8: Insert C & F



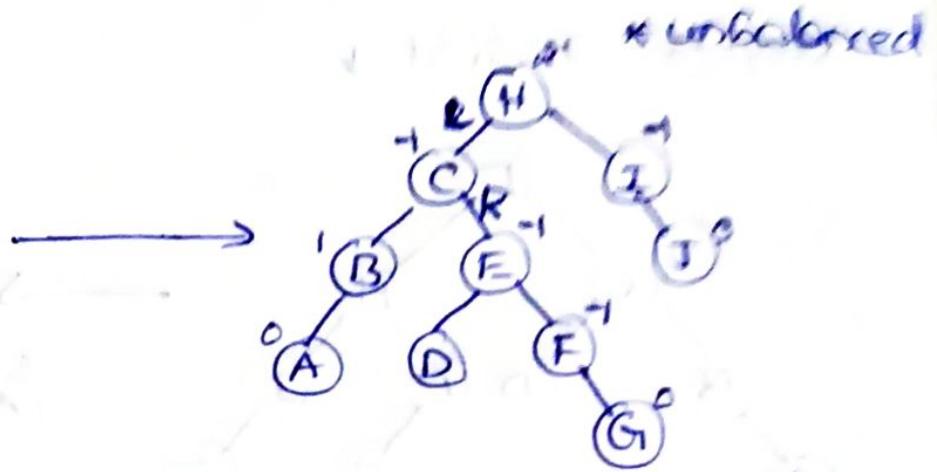
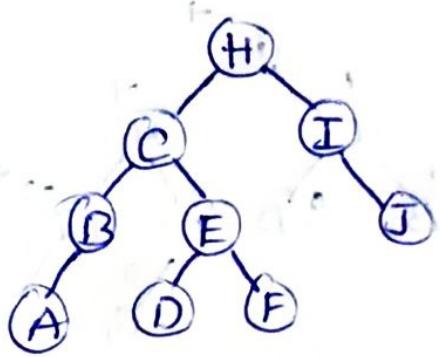
Step 9: Insert D



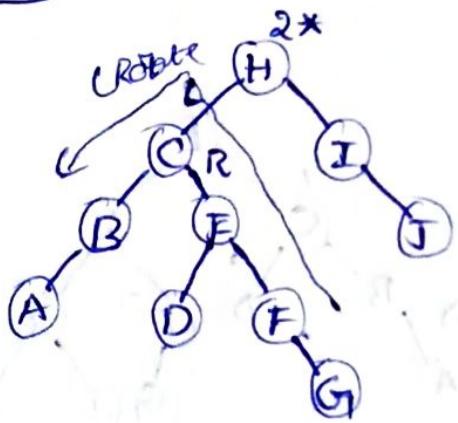
Step 10: Perform RL Rotation



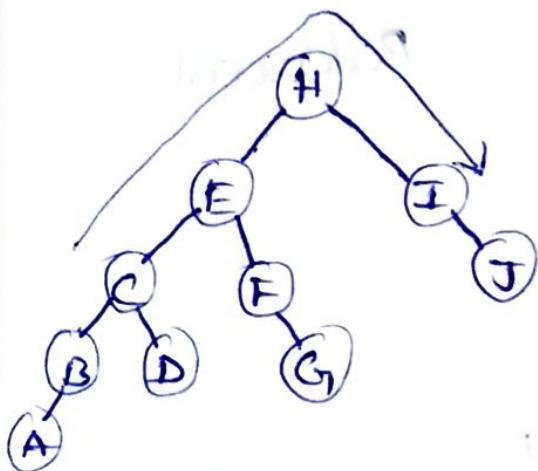
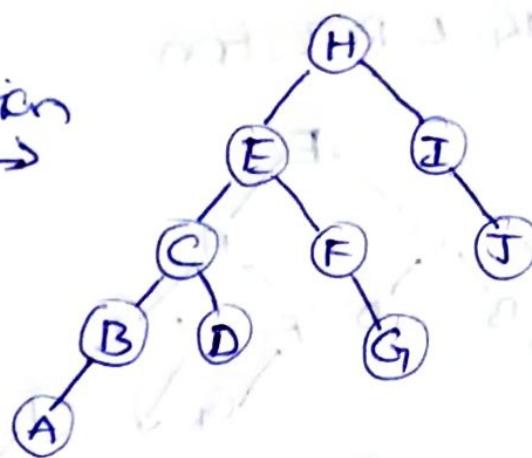
Step 11: Insert G



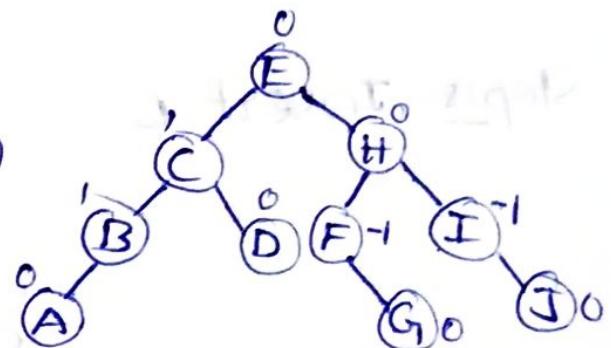
Step 12: RLRR Rotation



L Rotation

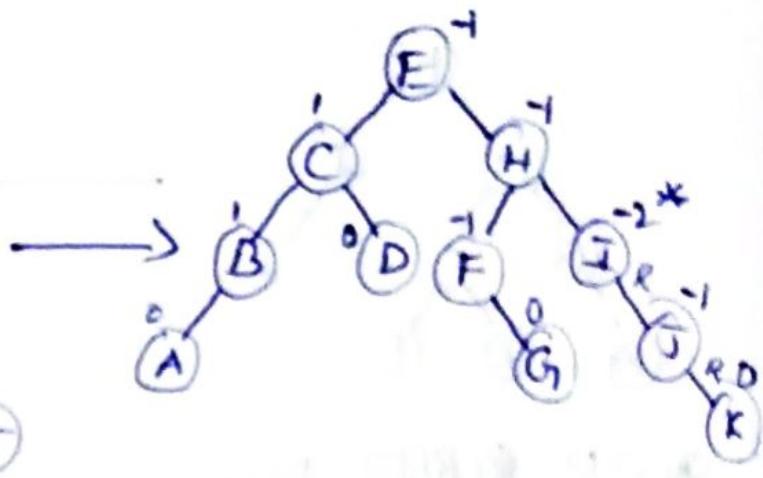
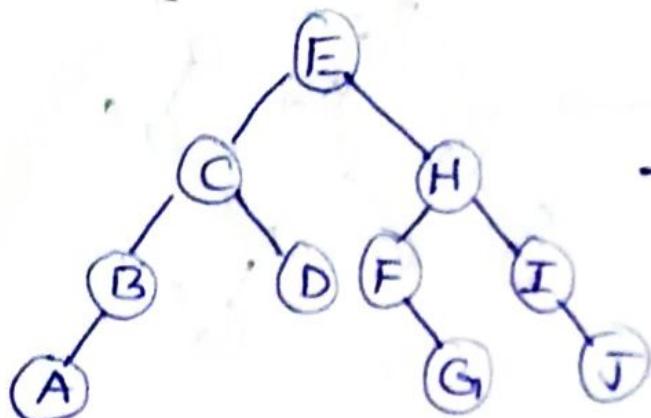


R Rotation

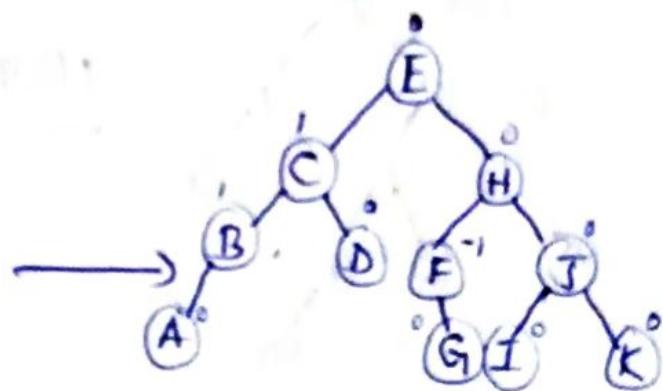
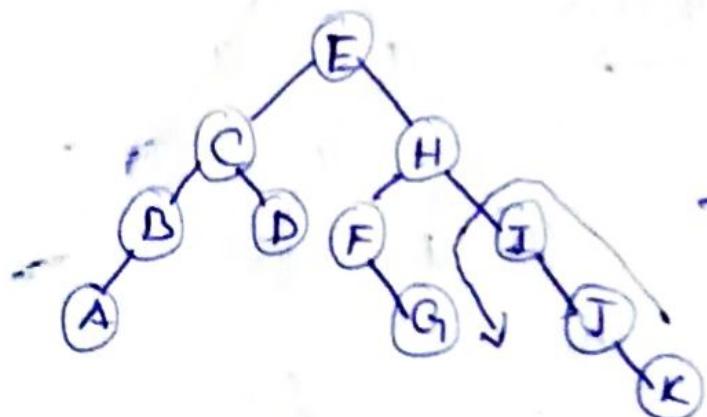


* Balanced

Step 13: Insert K

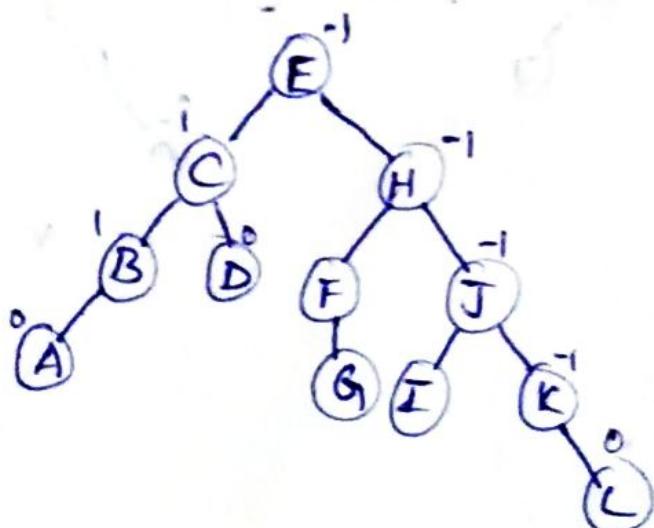


Step 14: L rotation



Balanced

Step 15: Insert L



Balanced

Final AVL Tree

BTree

- * specialized m-way tree
- * widely used for disk access
- * balanced m-way tree
- * Generalization of BST in which a node can have more than one key and more than 2 children
- * maintains sorted data
- * all leaf nodes must be at same level

BTree of order m has following properties

- * Every node has max m children
- * Minimum children for leaf $\rightarrow 0$

root $\rightarrow 2$

internal nodes $\rightarrow \lceil \frac{m}{2} \rceil$

- * Every node has max $(m-1)$ keys
- * minimum keys : rootnode $\rightarrow 1$
all other nodes $\rightarrow \lceil \frac{m}{2} \rceil - 1$
- * insertion should always be done on leaf node
- * capable to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- * While performing some operations on Btree, any property of Btree may violate such as number of min child a node can have. To maintain properties of Btree, the tree may split or join.
- * Example not given here in this book, please refer.

Application of Btree

- * Btree is used to index the data and provides fast access to the actual data stored on disks, in general it is time consuming process
 - * unindexed & unsorted $\Rightarrow O(n)$ worst case
 - * Indexed & sorted (Btree) $\Rightarrow O(\log n)$ worst case
-

B+ Tree

- * Extension of Btree (Search online for example)
- * Allows efficient insertion, deletion and search operations.
- * In Btree, keys and records both can be stored in the internal as well as leaf nodes.
- * In B+ tree, records can only be stored on the leaf nodes while internal nodes can only store the key values.
- * Leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries efficient
- * used to store large amount of data, so using main memory for storing keys to access records and leaf nodes are stored in secondary memory, as leaf nodes can contain large amount of data as this is actual data.
- * Internal nodes of B+ tree are called index nodes.
- * Faster search queries as data is stored only on leafs.

- * Redundant search keys are present in B+ tree, but not possible in Btree.
- * Search, Insertion & Deletion are comparatively faster in B+ tree because the records will only be on leaf nodes, when coming to Btree, records can be present in internal nodes, so operations are a bit slower in Btree

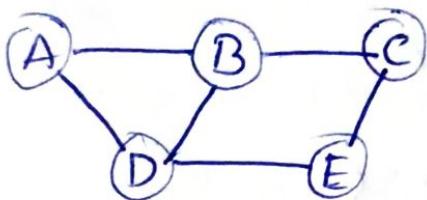
Insertion/Deletion is similar to BST, if any of these operations violate properties of B/B+ tree, we can split/merge child nodes with parent node or among siblings.

Graphs

- * Defined as group of vertices and edges that are used to connect these vertices.
- * also called as cyclic tree
- * Graph G_i can be defined as an ordered set $G_i(V, E)$, where $V(G_i)$ represents set of vertices and $E(G_i)$ represents set of edges that are used to connect these vertices.

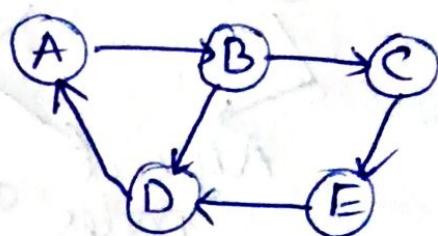
Example

A graph $G(V, E)$ with 5 vertices (A, B, C, D, E) and six edges $((A, B), (B, C), (C, E), (E, D), (D, B), (D, A))$ is shown below



undirected graph

($A \rightarrow B, B \rightarrow A$ possible)



Directed graph

Path: sequence of nodes that are followed in order to reach some terminal node v from the initial node u

Closed path: If initial node u is same as terminal node.

Simple path (closed)
 $v_0 = v_n$

* nodes of graph are distinct with exception $v_0 = v_n$

Cycle: Path which has no repeated edges or vertices except the first and last vertices.

Connected graph:

* Some path exists between every 2 vertices (u, v) in V .

* no isolated nodes in connected graph.

complete graph:

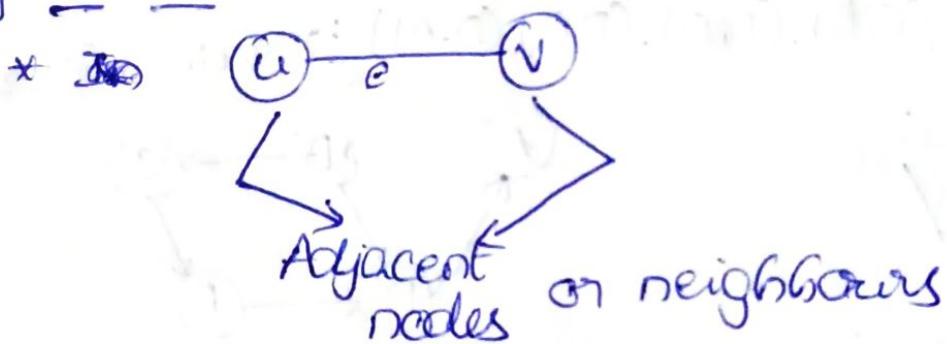
- * every node is connected with all other nodes
- * contain $\frac{n(n-1)}{2}$ edges
- * n is number of nodes in the graph.

weighted graph

- * Each edge is assigned with some length / weight
- * denoted by $w(e)$
- * value denotes cost of traversing the edge.

Digraph \Rightarrow Directed graph

Adjacent nodes



Degree of the node

- * number of edges that are connected with that node
- * Degree 0 \Rightarrow Isolated node

Graph representation

2 ways to store graphs in computer memory

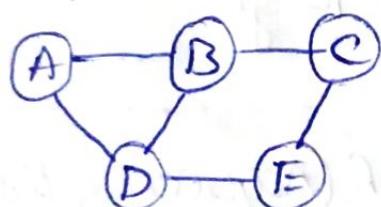
- * sequential repr (i.e, Adjacency Matrix)
- * linked list repr (i.e, Adjacency List)

- ① Adjacency Matrix
- ② Adjacency List

Adjacency Matrix

- * used to represent undirected graph, directed graph, weighted directed and undirected graph.
- * $\text{adj}[i][j] = \omega \Rightarrow$ Edge exists from i to j with " ω "
- * $0 \Rightarrow$ no association between the nodes
- * $1 \Rightarrow$ Existence of path between two nodes

Example

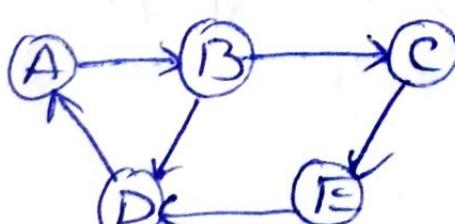


undirected

graph

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

- * If there is any self loop in the graph then there will be '1' on the diagonal of Adjacency Matrix.



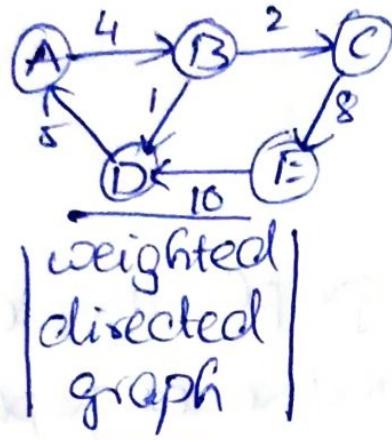
Directed graph

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

No self loop \Rightarrow All diagonal entries are 0 in Adj. M.

Adjacency Matrix for weighted directed graph

* Entries in adjacency matrix are weights between vertices



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

Properties

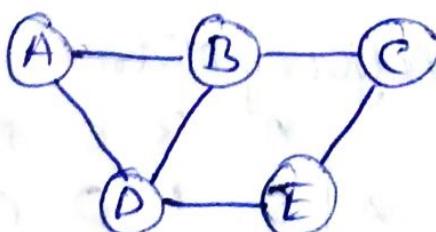
- * Easier to implement
- * Mostly used when the graph is dense and num of edges are large
- * consumes more space

Adjacency Matrix

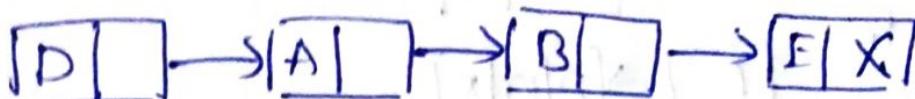
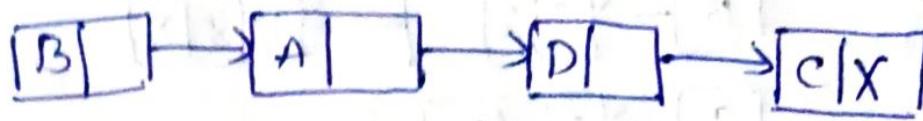
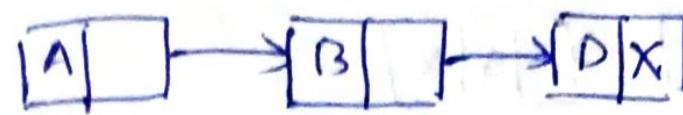
Linked List implementation of graph (Adjacency list)

- * Efficient in terms of storage
- * we only have to store value of edges for a vertex

Example:



undirected graph

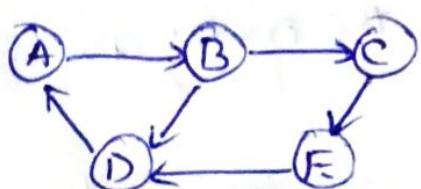


Adjacency list

we will have a LL on each of vertex which is connected or have info of all its adjacent nodes.

Ex: A is connected directly to B & D. So we have a LL, having these nodes as part of LL

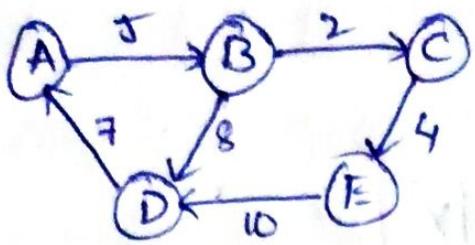
If all adjacent nodes of a node are visited store the next value of a node as NULL



Directed graph

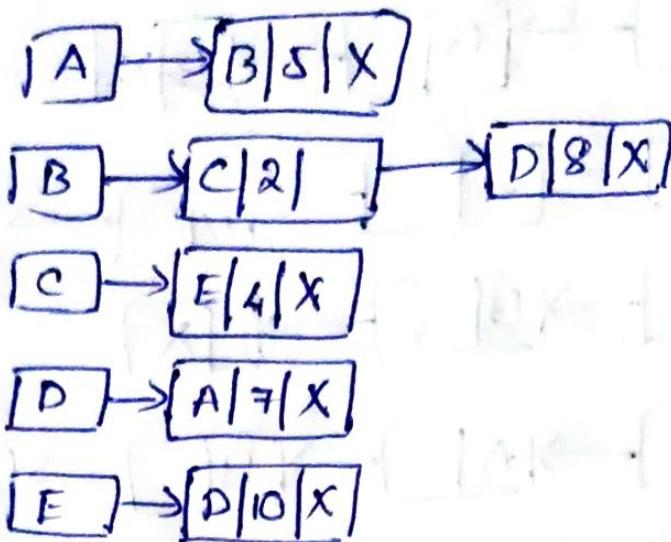


Adjacency list



weighted directed graph

- * contains an extra field that is weight of a node.



Adjacency list

BFS algorithm

- * Graph traversal algorithm
- * recursive algorithm to search all the vertices of a tree or graph data structure.
- * Two categories for any vertex : visited and non-visited
- * select a single node in a graph, and after that visits all the nodes adjacent to the selected node and perform the same approach recursively until all the nodes are visited.

Applications of BFS algorithm

- * Find neighboring locations from a given source location.
- * peer to peer networking used in BitTorrent, uTorrent.
- * used in web crawlers to create web page indexes. it is used to index web pages.
- * used to determine the shortest path and Minimum Spanning Tree

Algorithm:

Step1: Set status = 1 (Ready state) for each node in G

Step2: Enqueue the starting node A and set its status = 2
in waiting state

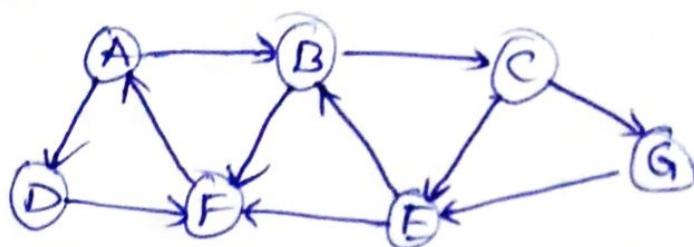
Step3: Repeat Steps 4 and 5 until Queue is empty

Step4: Dequeue a node N. Process it and set its status = 3
in processed state

Step5: Enqueue all the neighbours of N, that are in the ready
state (whose STATUS=1) and set their status = 2 (waits)
[END OF LOOP]

Step6: EXIT

Example



Adjacency List

A: B, D F: A

B: C, F G: E

C: G, E

D: F

E: B, F

Min Path from A to E, can be found
using BFS

Queue1 \Rightarrow all the nodes that are to be processed

Queue2 \Rightarrow all the nodes that are processed and del from Q1

Steps:

Step1: Add A into Q1

$$Q_1 = \{A\}$$

$$Q_2 = \{\text{null}\}$$

Step2: A is processed, insert neighbours of A into Q1 which
are unvisited

$$Q_1 = \{B, D\}$$

$$Q_2 = \{A\}$$

Step 3: B is processed, add neighbours of B into Q_1 ,

$$Q_1 = \{C, F, D\} \setminus \{B\}$$

$$Q_2 = \{A, B\}$$

Step 4: D is processed, add neigh of D into Q_1 ,

$$Q_1 = \{C, F\}$$

$$Q_2 = \{A, B, D\}$$

If neigh is already visited or in waiting state need not to be add another time.

Step 5: C is processed, add neigh of C into Q_1 ,

$$Q_1 = \{F, E, G\}$$

$$Q_2 = \{A, B, D, C\}$$

Step 6: F is processed

$$Q_1 = \{E, G\}$$

$$Q_2 = \{A, B, D, C, F\}$$

Step 7: E is processed, and we get the shortest path Cw

$$A \rightarrow E$$

$$Q_1 = \{G\}$$

$$Q_2 = \{A, B, D, C, F, E\}$$

$$\text{Path} \Rightarrow \boxed{A \rightarrow B \rightarrow D \rightarrow C \rightarrow F \rightarrow E}$$

Complexity of BFS algorithm

Time $C \Rightarrow O(V+E)$ \Rightarrow worst case, as it visits every $V \notin E$

Space $C \Rightarrow O(V)$

$V \Rightarrow$ Vertices / nodes

$E \Rightarrow$ Edges

DFS Algorithm

- * recursive algorithm to search all the vertices of a tree data structure of a graph.
- * starts with initial node of graph G and goes until we find goal node or the node with no children.

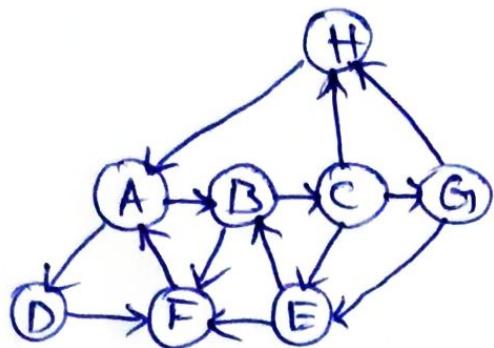
Steps

- ① First, create a stack with the total number of vertices in the graph.
- ② Now, choose any vertex as the starting point of traversal and push that vertex into the stack.
- ③ After that, push a non visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
- ④ Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on stack's top.
- ⑤ If no vertex is left, go back and pop a vertex from the stack.
- ⑥ Repeat steps 2,3,4 until stack is empty.

Applications

- * topological sorting
- * Find the paths between two vertices
- * used to detect cycles in the graph.
- * Determine if a graph is Bipartite or not

Example:



Adjacency list

A : B, D
 B : C, F
 C : E, G, H
 G : E, H
 E : B, F
 F : A
 D : F
 H : A

Step 1

Stack : H

Step 2

print H

Stack A

Step 3

print A

Stack B, D

Step 4

print D

Stack B, F

Step 5

print B

Stack B

Step 6

print B

Stack C

Step 7

print C

Stack E, G

Step 8

print G

Stack E

Step 9

print E

Stack \square

Time Complexity of DFS : $O(V+E)$

Space Complexity of DFS : $O(V)$

- * DFS occupies lesser memory than BFS
- * not guaranteed to find the solution
- * Issue comes with infinite graph to the left, a depth should be fixed as threshold, so it will not stuck in infinite execution
- * no guarantee in finding optimal solution

Spanning Tree

Properties

- * subgraph of an undirected connected graph.
- * If we can reach any vertex from any vertex it is connected graph.
- * It includes all the vertices along with least possible number of edges. If any vertex is missed then it is not spanning tree.

Should not be

- ① cycles
- ② disconnected

n vertices

$n-i$ edges

→ Condition

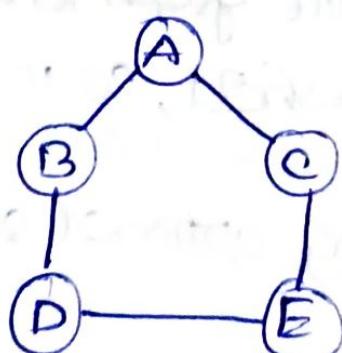
- * maximum number of spanning trees possible for complete undirected graph $\Rightarrow n^{n-2}$

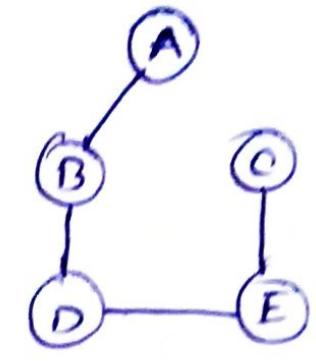
n = Number of vertices in graph

Applications of spanning tree

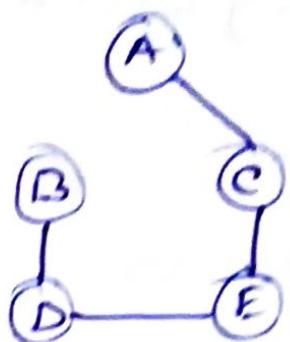
- * cluster analysis
- * civil network planning
- * computer network routing protocol

Example:

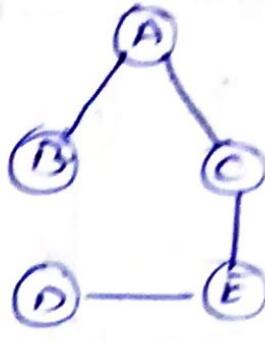




spanning tree 1



spanning tree 2



spanning tree 3

Properties

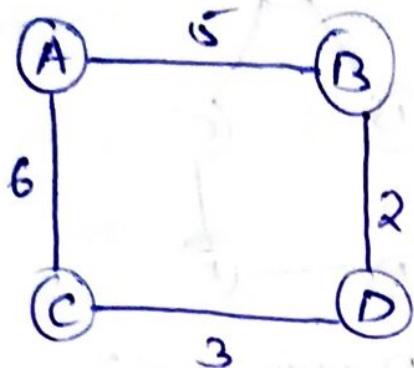
- * more than one spanning tree possible in Graph G
- * does not have any cycles or loop
- * is minimally connected, so removing one edge from tree will make graph disconnected.
- * maximally acyclic, so adding one edge to a tree will create a loop.
- * max of n^{n-2} number of spanning trees possible.
- * should only have $n-1$ edges.
- * If graph is a complete graph, then spanning tree can be constructed by removing $\max(e-n+1)$ edges.

$e \Rightarrow$ num of edges

$n \Rightarrow$ num of vertices

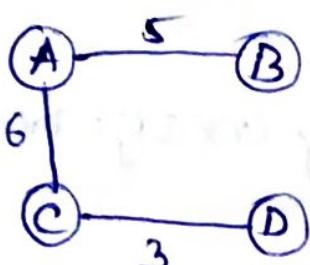
- * [Can only be created for connected graph]
- * [No spanning tree for disconnected graph]
- * [MST \Rightarrow path weight should be minimum]

Minimum Spanning Tree Example

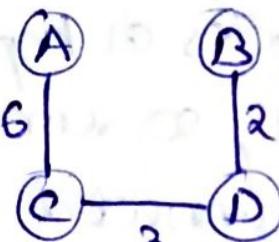


weighted graph

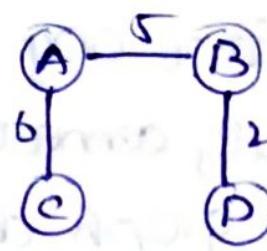
Possible spanning trees are,



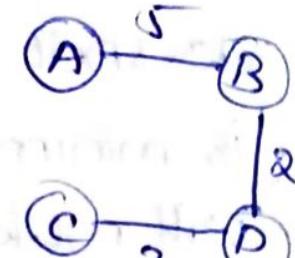
$$\text{Sum} = 14$$



$$\text{Sum} = 13$$



$$\text{Sum} = 10$$



$$\text{Sum} = 13$$

Applications of MST

[MST]

- * Design water supply networks
- * Design telecommunication networks
- * Find paths in the map

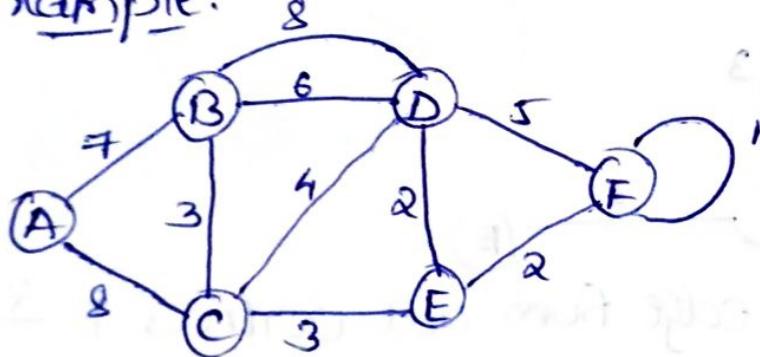
MST can be found by following algo's:

- * Prim's Algorithm
- * Krushkal's Algorithm

Prim's algorithm:

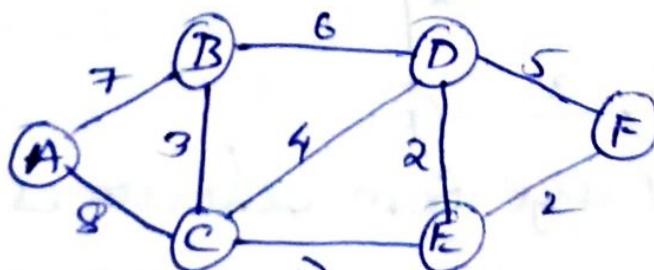
- * used to calculate min spanning tree
- * select a root vertex , and then select a shortest edge connected to that vertex , and then we need to do the same process of select a shortest edge of previously selected vertex's
- * generate the minimum cost spanning tree starting from a root node.
- * prerequisites ,
 - ① remove the self loops
 - ② remove the parallel edges
 - ③ Then, do/perform the above algorithm

Example:



Find MST:

step 1: remove self loops and parallel edges

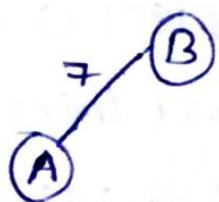


step 2: Select any vertex \Rightarrow A

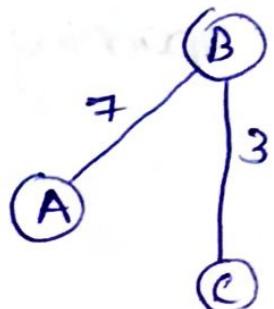
Step 3: Find an available and minimum output

edge from the available vertices of the graph

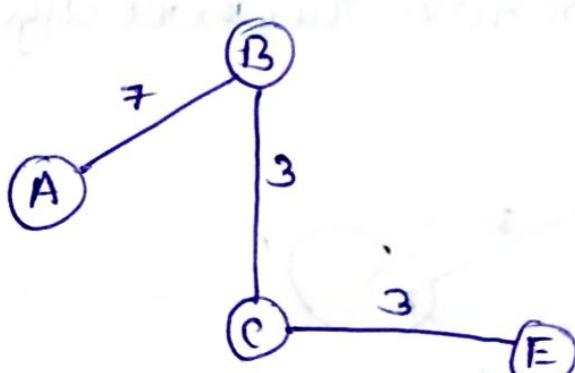
$$\hookrightarrow A \xrightarrow{?} B$$



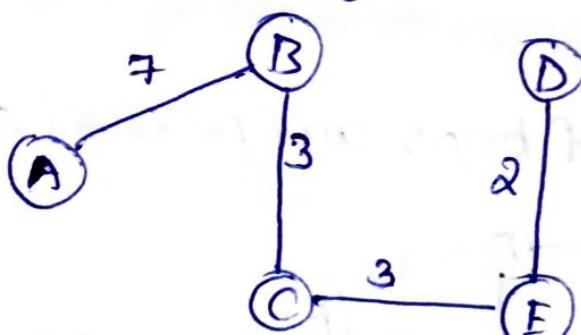
Step 4: Find min ^{output} edge from all vertices $B \xrightarrow{?} C$



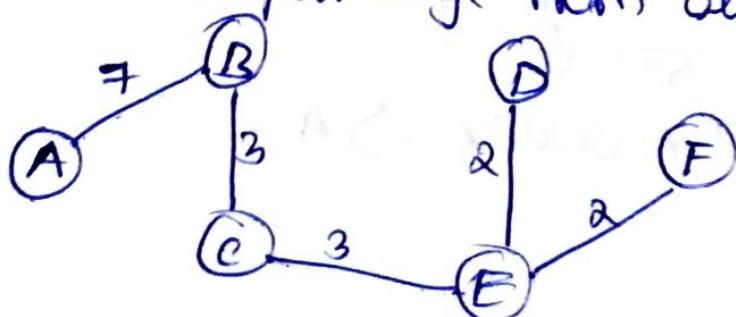
Step 5: Find min output edge from all vertices $C \xrightarrow{?} E$



Step 6: Find min output edge from all vertices $E \xrightarrow{?} D$



Step 7: Find min output edge from all vertices $E \xrightarrow{?} F$



If we assume $G(V, E)$ is complete graph and $G'(V', E')$ is a MST graph, then for a minimum spanning tree graph these would be true

$$\boxed{V' = V} \rightarrow \text{number of vertices of MST \& complete will be same}$$

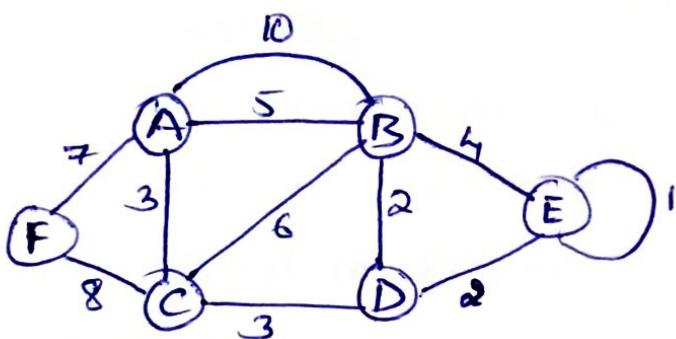
$$\boxed{E' \subset E} \rightarrow \text{MST is subset of complete graph.}$$

Num of edges of MST is $E' = |V| - 1$

$V \rightarrow$ num of vertices of MST

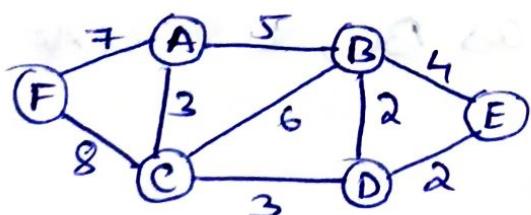
$E' \rightarrow$ num of edges of MST

Kruskal's algorithm



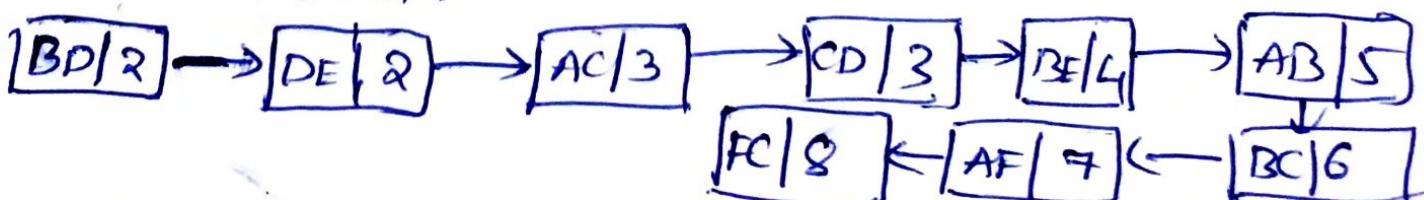
Step 1:

- ① Remove self loops
- ② Remove parallel edges

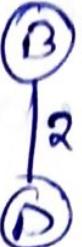


Step 2: we need to select minimum edges in a sequence and build a path such that there shouldn't be any cycle

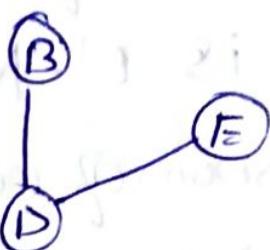
selection order:



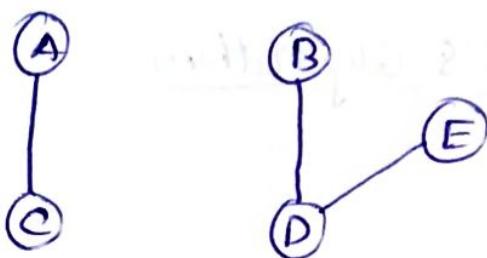
Step 3: select BD



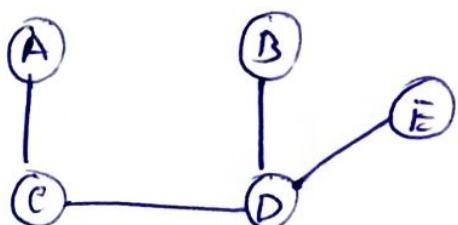
Step 4: select DE



Step 5: select AC



Step 6: Select CD



Step 7: we need to select BE, but we shouldn't make a cycle, so leave BE and see AB, AB also fall into this category, as well as BC, so select AF



MST by Kruskals Algorithm

Searching algorithm

Linear Search

- * Main aim is to find an element in the list
- * also called as sequential search algorithm
- * simplest searching algorithm.
- * mostly used to search an element from the unordered list * Avg case T.C $\Rightarrow O(n)$
- * Worst case T.C $\Rightarrow O(n)$ * Best case $\Rightarrow O(1)$

Algorithm:

→ Iterate through every element of array through for loop, in each iteration compare search element with current element, if element matches return index of corresponding element, if not found in entire array return -1

Example

Array:

70	40	30	4	57	41	25	14	52
----	----	----	---	----	----	----	----	----

$K = 40 \neq 41$

$| K = 41$

Iteration starts from $K \rightarrow 70$ to $K \rightarrow 41$
as element matched we return the index
if $K = 42$, as element NOT present in list, we
iterate through whole array and return -1
as element not found.

[Space Complexity : $O(1)$]

Binary Search

- * searching technique
- * works efficient on sorted lists
- * If we want to use this algorithm, we must ensure that list should be sorted [space complexity: $O(1)$]
- * Follows divide and conquer approach:
Approach: Time complexity $\begin{cases} \text{Best Case } O(1) \\ \text{Average/Worst Case } O(\log n) \end{cases}$
- * mainly depends on the center element
- * if mid value of the array is the element that we required, then return that position where we found the element.
- * If the value we want is lesser than that of mid value then make $\Rightarrow \underline{\text{end} = \text{mid} - 1}$
- * If the value that we want is greater than that of mid value then make $\Rightarrow \underline{\text{start} = \text{mid} + 1}$

Important points

- * BS can implement through Iterative & Recursive approach

Example

10	12	24	29	39	40	51	56	59
								$K=56$

Step 1: [10 12 24 29 39 40 51 56 59], $K \neq \text{mid} \Rightarrow K > \text{mid}$
 ↑mid $\Rightarrow \text{move right} \Rightarrow \text{start} = \text{mid} + 1$

Step 2: [10 12 24 29 39 40 51 56 59] $K < \text{mid} \Rightarrow K < \text{mid}$
 ↑mid $\Rightarrow \text{move right} \Rightarrow \text{start} = \text{mid} + 1$

Step 3: [10 12 24 29 39 40 51 56 59] $K = \text{mid}$, returns True
 ↑mid as element found

Sorting algorithms

Bubble sort

- * repeatedly swapping of adjacent elements until they are not in intended order.
- * Elements of an array move to end in each iteration.
- * poor performance [Space Complexity: $O(1)$]
- * Time complexity (Worst & Average) $\Rightarrow O(n^2)$ already sorted

Algorithm:

```

begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort

```

Imp: At the end of each pass or at end of each for iter we will help an element to find its position

1st iteration \Rightarrow nth element

2nd iteration \Rightarrow n-1th element

3rd iteration \Rightarrow n-2th element

!

Each iteration, we sort the array from the end

Example: Best case T.C $\Rightarrow O(n)$

arr =

13	32	26	35	10
0	1	2	3	4

1st iteration

13	32	26	35	10
0	1	2	3	4

 comp 0 < 1, skip

13	32	26	35	10
0	1	2	3	4

 comp 1 < 2, swap

13	26	32	35	10
0	1	2	3	4

 comp 2 < 3, skip

13	26	32	35	10
0	1	2	3	4

 comp 3 < 4, swap

13	26	32	10	35
0	1	2	3	4

 nth element sorted *

Similarly

2nd iteration

13	26	10	32	35
0	1	2	3	4

 n-1th ele sorted

3rd iteration

13	10	26	32	35
0	1	2	3	4

 n-2th ele sorted

4th iteration

10	13	26	32	35
0	1	2	3	4

 n-3th ele sorted

we need to perform only n-1 iter.

so, here we have to perform 4 iterations as we have 5 elements

optimized bubble sort

* To prevent performing operations on sorted arrg, we use this approach, this decreases execution time

Algorithm:

bubbleSort(array)

n = length(array)

repeat

swapped = false

for i=1 to n-1

if array[i-1] > array[i], then

swap(array[i-1], array[i])

swapped = true

end if

end for

n = n-1

until not swapped

end bubbleSort

Insertion Sort Algorithm

works similar to sorting of playing cards, we assume that the first element is sorted and arrange the rest of the elements w.r.t the sorted array (First card)

- * If the second card is greater than the first, leave wherever it is. If the second card is lesser than elements of the sorted array then move all pack of cards which are sorted to right. And if it lies anywhere lesser/greater than elements of sorted array include the card in sorted pack.
- * Worst case time complexity $\Rightarrow O(n^2)$
- * Lesser efficient than other sorting algo's like
 - heapsort
 - mergesort
 - quicksort

Advantages

- * Simple implementation
- * Appropriate for partially sorted datasets
- * Efficient for small datasets

Algorithm

- Step1: If element is first element, assume it as sorted and set 1
- Step2: pick the next element, and store separately in a key.
- Step3: Now, compare the key with all elements in sorted array.
- Step4: If element in sorted array is smaller than current then move right, else shift greater elements in array towards right
- Step5: insert the value
- Step6: Repeat until the array is sorted

making it simple insertion sort does two Operations

- ① Firstly compare first two elements of unsorted array
* Swap those, if they are not in correct order.
- ② Now, After swapping compare the first element of previous two elements and compare and arrange them in sorted array. Thus in each iteration we add an element to sorted array from unsorted array.

Example:

0 1 2 3 4 5

Array $A = \{12, 31, 25, 8, 32, 17\}$

compare 0 & 1 \Rightarrow no swap \Rightarrow store 12 in sorted array
(sorted already)

$A = \{\boxed{12}, 31, 25, 8, 32, 17\}$

compare 1 & 2 \Rightarrow 31 \neq 25 \Rightarrow swap \Rightarrow compare 8, store 25
in sorted array

$A = \{\boxed{12, 25}, 31, 8, 32, 17\}$

compare 2 & 3 \Rightarrow 31 \neq 8 \Rightarrow swap \Rightarrow compare 8, store 8 in
sorted array

$A = \{\boxed{12, 25, 8}, 31, 32, 17\}$ 25 \neq 8 \Rightarrow swap

$A = \{\boxed{12, 8, 25}, 31, 32, 17\}$ 12 \neq 8 \Rightarrow swap

$A = \{\boxed{8, 12, 25}, 31, 32, 17\}$

compare 3 & 4 \Rightarrow 31 < 32 \Rightarrow no swap \Rightarrow compare & store
31 in sorted array

$$A = \{ [8, 12, 25], 31, 32, 17 \}$$

$$A = \{ [8, 12, 25, 31], 32, 17 \}$$

compare 4 & 5 \Rightarrow 32 < 17 \Rightarrow swap \Rightarrow compare & store 17 in
sorted array

$$A = \{ [8, 12, 25, 31, 17], 32 \} \quad 31 \neq 17 \text{ swap}$$

$$A = \{ [8, 12, 25, 17, 31], 32 \} \quad 25 \neq 17 \text{ swap}$$

$$A = \{ [8, 12, 17, 25, 31], 32 \} \quad 8, 12 < 17 \Rightarrow \text{no swap}$$

compare 5 with sorted array:

$$32 > [8, 12, 17, 25, 31]$$

or

$$[8, 12, 17, 25, 31] < 32 \quad (\text{Already sorted})$$

Sorted array: $\{8, 12, 17, 25, 31, 32\}$

Time Complexity:

Best Case \Rightarrow no sorting req $\Rightarrow O(n)$

Average Case \Rightarrow partially sorted $\Rightarrow O(n^2)$

Worst Case \Rightarrow complete unsorted $\Rightarrow O(n^2)$

Space Complexity:

only variable required for swapping $\Rightarrow O(1)$

Insertion Sort uses / applies

used when

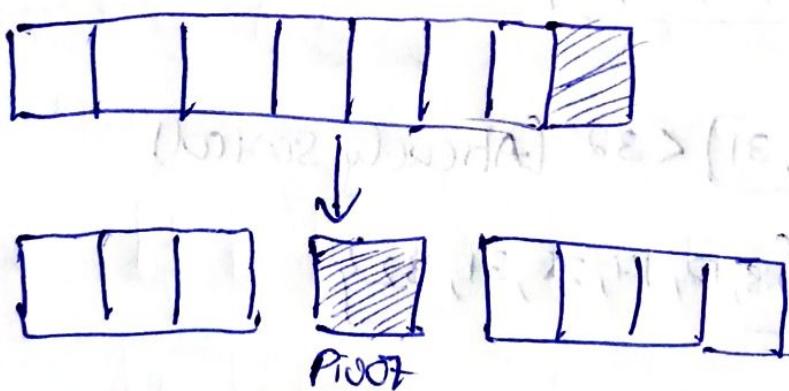
* array has small number of elements

* partially sorted arrays

Quick Sort algorithm

- * Sorting algorithm based on divide & conquer
- * we typically choose a pivot (in general) we choose pivot as end of the array
- * Pivot is basically a separation point for lesser elements to that of greater elements.
- * we move pivot among array such that all the elements left of pivot are lesser than that of all the elements right of pivot,
- * we do the above operation recursively such that subarray's are almost divided into single valued array (len(arr)=1)

Example:



Quick Sort

- * Faster and highly efficient sorting.
- * $n \log n$ comparisons in average case
- * Follow divide and conquer approach.
- * D&C method of breaking an algorithm into sub problems and then solving those subproblems and combine those results to solve the original problem.

Divide: pick pivot, then move all lesser elements of array to left of pivot and greater elements to the right of pivot.

Conquer: Recursively sort two subarrays with Q.S.

Combine: combine the already sorted array.

We can choose pivot as

- * pivot can be random
- * pivot can be either rightmost or leftmost element
- * pivot can be the median element.

Quicksort complexity:

Time complexity:

Best Case \Rightarrow pivot is middle $\Rightarrow O(n \log n)$
or near to middle element

Average Case \Rightarrow Halfsorted $\Rightarrow O(n \log n)$
array

Worst Case \Rightarrow pivot is either $\Rightarrow O(n^2)$
the greatest or
smallest element
of array

Worst TC of Q.S is more than other sorting algo's such as Merge Sort and Heapsort, still it is faster in practice, worst case can be avoided by choosing right pivot.

Space complexity: $O(\log n)$

Merge Sort algorithm

- * Follows divide and conquer approach.
- * Efficient sorting algorithm.
- * Divides the given list into two equal halves, calls itself for the two halves and then merges the sorted halves.
- * It will be divided into halves until the list cannot be divided further.

Example:

- ① Let's take an unsorted array

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

- ② According to merge sort, divide the array into halves until it can't be divided

arr =

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

arr =

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

Divide

arr =

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

- ③ Compare them into another array in sorted order

12	31	8	25	17	32	40	42
8	12	25	31	17	32	40	42

Final Merging

8	12	17	25	31	32	40	42
---	----	----	----	----	----	----	----

Merge Sort Time Complexity

Best case \Rightarrow Already sorted $\Rightarrow O(n \log n)$

Average Case \Rightarrow partially sorted $\Rightarrow O(n \log n)$

Worst Case \Rightarrow Totally unsorted $= O(n \log n)$

Space Complexity : $O(n)$

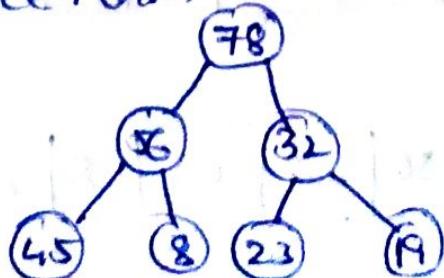
Heap Sort Algorithm

Heapsort is an improved version of the selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list. and later consider that node as sorted later on.

After exchanging the largest element with the last element, the tree will be not following "max heap", so heapify the tree and do this process all over again until there are no unsorted node left in the array. And elements are sorted in ascending order.

Heap representation

a) Tree Form



b) array form

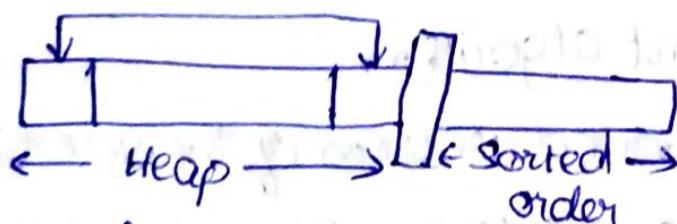
78	56	32	45	8	23	19
----	----	----	----	---	----	----

If parent value is greater than all of its children
↳ max heap

If parent value is smaller than all of its children
↳ min heap

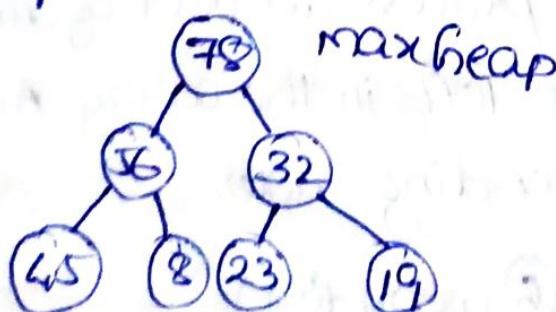
To sort in ascending order we use max heap property on the tree.

Heap sort exchange process



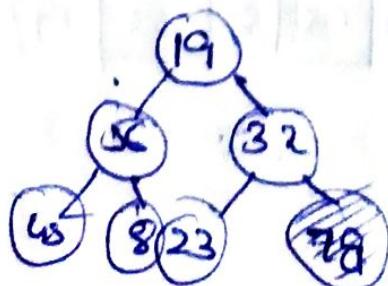
Exchange root of max heap, which is also the highest element of the unsorted array / tree with the last element of the unsorted array and merge that node into the sorted array.

Example:



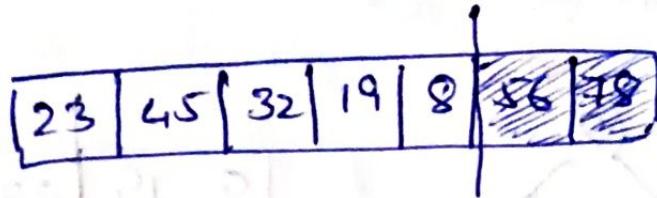
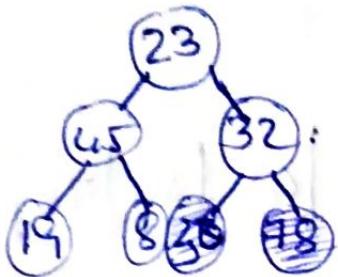
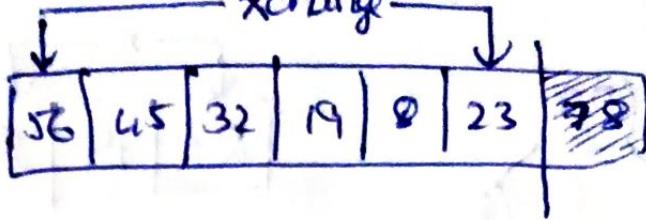
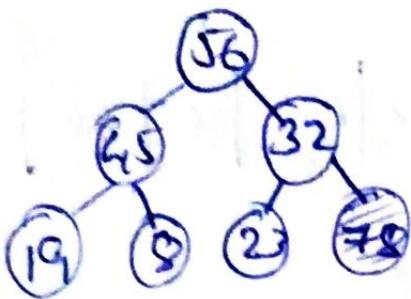
78	56	32	8	23	45
----	----	----	---	----	----

78	56	32	45	8	23	19
----	----	----	----	---	----	----

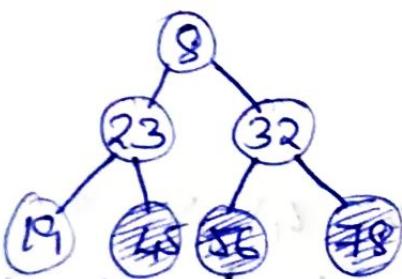
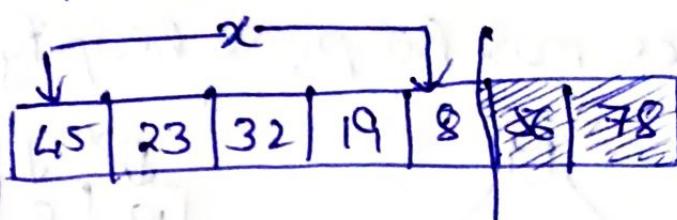
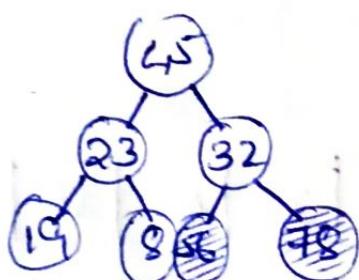


19	56	32	45	8	23	78
----	----	----	----	---	----	----

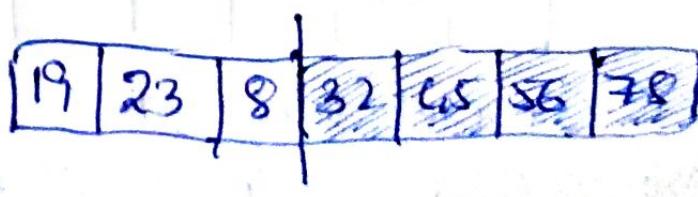
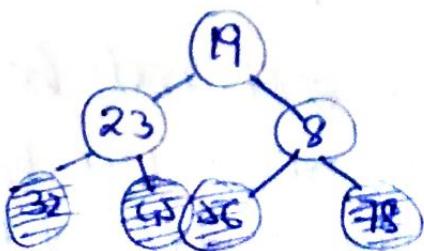
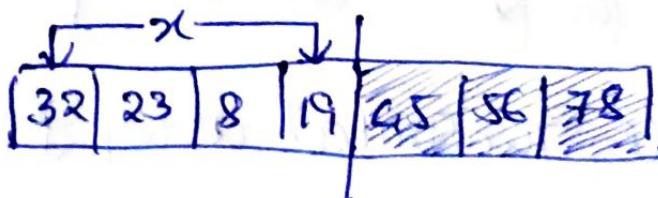
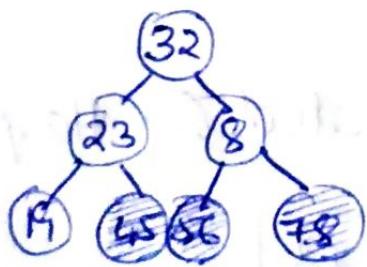
violates max heap property, so heapify



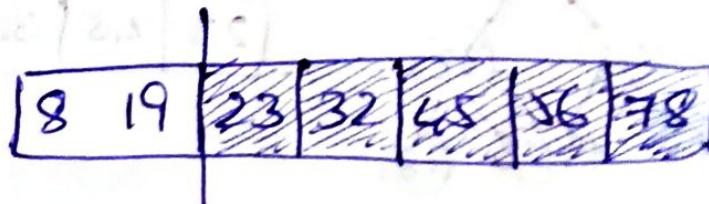
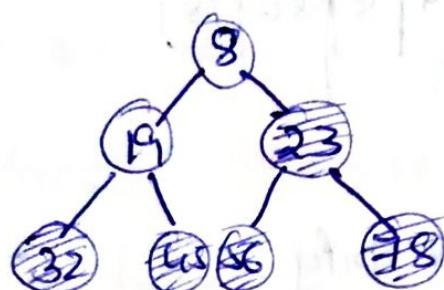
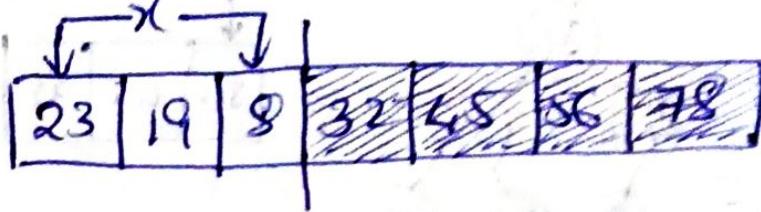
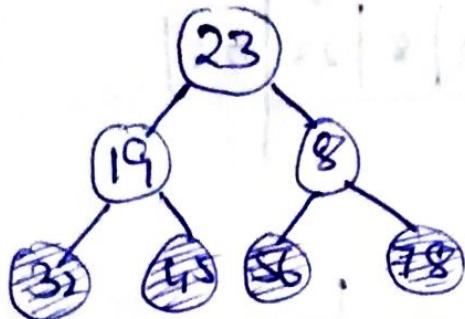
violates max heap property, so heapify



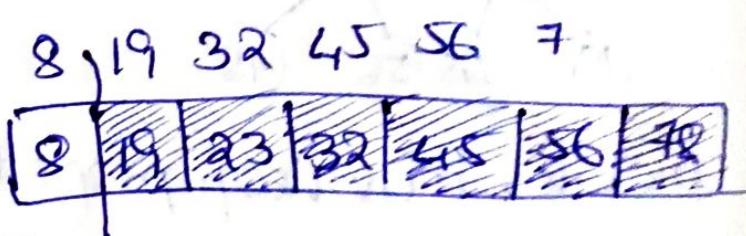
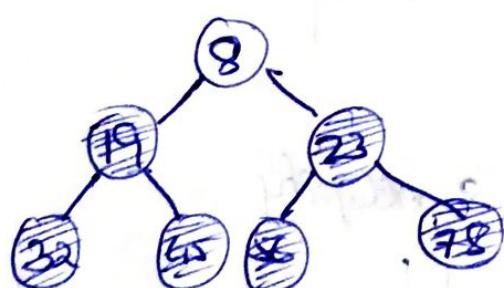
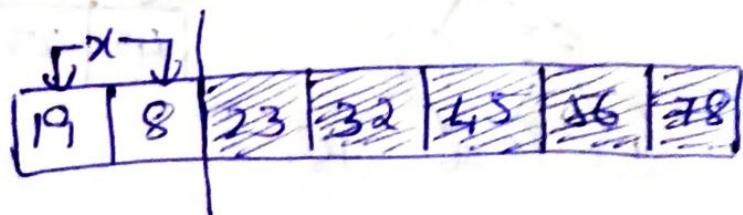
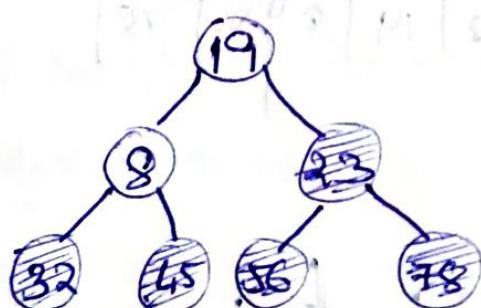
violates max heap property, so heapify



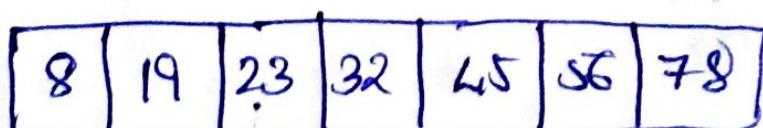
violates max heap, so heapify



violates max heap, so heapify



Finally the sorted array generated by Heapsort



Heap is 'complete binary tree', where each node has utmost 2 children. In OBT all levels except last level are completely filled. all nodes should be left justified.

Time Complexity:

Best Case: no sorting required $\Rightarrow O(n \log n)$

Average Case: no proper ascending/descending $\Rightarrow O(n \log n)$

Worst Case: When array is in reverse order $\Rightarrow O(n \log n)$

When we do ascending order it
is in descending order

Height of CBT having n elements: $\log n$

Space Complexity: $O(1)$

Relationship between array indexes and tree elements:

- * CBT has an interesting property that we can use to find the children and parents of any node.
- * If the index of any element in the array is i , the element at index $2i+1$ will become the left child and element at $2i+2$ index will become the right child.
- Also the parent of any index i is given by the lower bound of $(i-1)/2$

$i \Rightarrow$ current node

$2i+1 \Rightarrow$ left child of i

$2i+2 \Rightarrow$ right child of i

$\left[\frac{i-1}{2}\right] \Rightarrow$ parent of i

Working of heap Sort

- ① since the tree satisfies max heap property, then largest item is stored at the root node.
- ② swap: Remove the root element and put at the end of the array (n^{th} position)
Put the last item of the tree (heap) at the vacant place.
- ③ Remove: Reduce the size of the heap by 1
- ④ Heapify: Heapify the root element again, so that we have the highest element at root
- ⑤ process is repeated until all the items of the list are sorted.

Time Complexity analysis

Case 1: [Initial heapifying]

Height of CBT having n elements is $\log n$

To fully heapify an element whose subtrees are already max heaps, we need to compare the element with its left and right children pushing it downwards until it reaches a point where both its children are smaller than it. In worst case, we need to move an element down the line from the root to the leaf node by making a multiple of $\log(n)$ comparisons and swaps.

Working of Heap Sort

- ① since the tree satisfies max heap property, then largest item is stored at the root node.
- ② Swap: Remove the root element and put at the end of the array (n^{th} position)
Put the last item of the tree (heap) at the vacant place.
- ③ Remove: Reduce the size of the heap by 1
- ④ Heapify: Heapify the root element again, so that we have the highest element at root.
- ⑤ process is repeated until all the items of the list are sorted.

Time Complexity analysis

Case 1: [Initial Heapifying]

Height of CBT having n elements is $\log n$

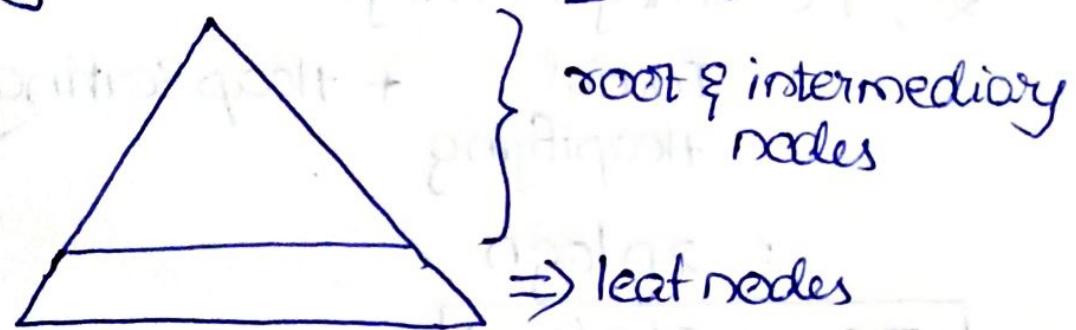
To fully heapify an element whose subtrees are already max heaps, we need to compare the element with its left and right children pushing it downwards until it reaches a point where both its children are smaller than it. In worst case, we need to move an element down the line from ~~the~~ root to the leaf node by making a multiple of $\log(n)$ comparisons and swaps.

In worst case, while we take an input from the user, we take it as an array which may or may not follow max heap. But to perform heapsort algorithm on array, that array must follow max heap, so we should heapify an array for each swap.

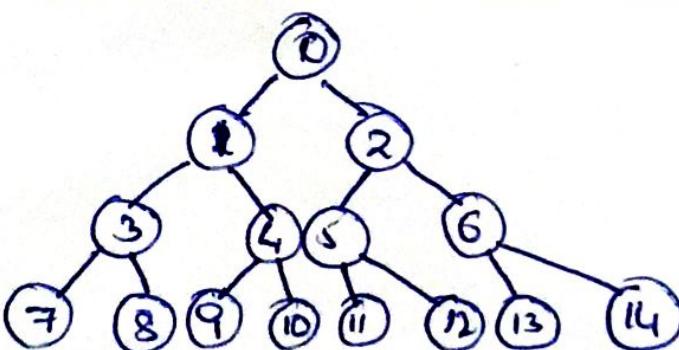
So to convert the user input array to array which follow max heap, we do heapification on $n/2$ elements.

so, we need

$\log(n)$ comparisions for $n/2$ elements to heapify the array to maxheap $\Rightarrow \frac{n}{2} \times \log(n) \sim n \log n$



One heapify operation set rights the 3 nodes, indeed itself and its children, we have to do the same operation on all its non leaf nodes, so the tree heapify



$$\begin{aligned}\text{leaf nodes} &\Rightarrow 7 \\ \text{Total elements} &\Rightarrow 14 \\ n &= 14\end{aligned}$$

We need to do heapification on nodes in this order of nodes: $6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$, it means, from $\frac{n}{2}^{\text{th}}$ node ($\text{index} = \frac{n}{2} - 1$) to The root node ($\text{index} = 0$)
node 6 to node 0

Case 2: [Heap Sort]

During sorting step, we exchange the root element with the last element and heapify the root element. We need to repeat this for n elements to make the tree a max heap followed times to make the tree a max heap followed For each element, this again takes $\log n$ worst time because we might have to bring the element all the way from the root to leaf. So the time complexity would be $n \cdot \log(n)$.

$$\text{so, } TC = n \log n + n \log n$$

Initial + Heap Sorting
Heapifying

$$= 2n \log n$$

$$TC = O(n \log n)$$

Space Complexity $O(1)$

Counting Sort

- * non comparison based algorithm.
- * Rest algorithms such as quicksort, Heapsort rest of those are comparison based
- ✓ Counting Sort, RadixSort, Bucket sort are non comparison based algorithm.
- * In Counting Sort, we are going to get two more inputs other than elements of array
 - ① Input size is 'n'
 - ② Range is 'k'

Range means, we are defining the range of values that must be present inside the array.

For example, if Range is 5, all the values that are present in the array are between 1 to 5.

Depends on the Range, we have to take an extra array such that the maximum index of array would be RangeUpperBound#1

Example:

Range = 5

Counting Sort is basically a freq counter of elements of array using an extra array.

extra array =

0	1	2	3	4	5

array = [2, 1, 2, 3, 1, 2, 4]

By applying Counting Sort, freq array

extra array =

0	2	3	1	1	0
0	1	2	3	4	5

, so loop through freq array and print i, freq[i] times, this gives $\Rightarrow 1, 1, 2, 2, 3, 4 \Rightarrow$ sorted

Time Complexity

$O(n+k)$ \Rightarrow To complete counting sort, we iterate through element array of size n , which generates freq array of size k . And to print the sorted output, we iterate through freq array of size k .

Space Complexity

$O(k)$ \Rightarrow we are using an extra array to store the frequency of elements which is of size ' k '

k is Range

- * It is not used as general purpose sorting algorithm.
- * It is effective when range is not greater than number of objects to be sorted (better than comp based algo's)
- * Can be used to sort negative input values.

Best Case T.C $\Rightarrow O(n+k)$ \Rightarrow already sorted

Average T.C $\Rightarrow O(n+k)$ \Rightarrow not ASC not DESC

Worst Case T.C $\Rightarrow O(n+k)$ \Rightarrow Reverse order

Counting Sort Algorithm

countingSort(array, n) // n is the size of array

max = find maximum element in the given array

create count array with size maximum + 1

initialize count array with all 0's

for i=0 to n

 find the count of every unique element and
 store that count at i^{th} position in the count arr

for j=1 to max

 Now, find the cumulative sum and store it in
 count array.

for i=n to 1

 Restore the array elements

 Decrease the count of every restored element by 1
end countingSort

Example

array =

2	9	7	4	1	8	4
---	---	---	---	---	---	---

Find max element $\Rightarrow \text{max} = 9$

Count array of size $\text{max}+1 \Rightarrow 10$

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

↓ accumulate

0	1	1	0	2	0	0	1	1	1
0	1	2	3	4	5	6	7	8	9

Cumulative count

0	1	2	2	4	4	4	5	6	7
0	1	2	3	4	5	6	7	8	9

We use cumulative count array, to know the position of the element in the output sorted array.

Example:

Original array = $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix}$

array = $\boxed{2 \mid 9 \mid 7 \mid 4 \mid 1 \mid 8 \mid 4}$

CC Array = $\boxed{\begin{matrix} 0 & 1 & 2 & 2 & 4 & 4 & 4 & 5 & 6 & 7 \end{matrix}} \quad \boxed{CC1}$

For Element 2

Important points

Value of array are elements

Value of ccarray is index of element in the output

Original array $\boxed{2 \mid 9 \mid 7 \mid 4 \mid 1 \mid 8 \mid 4}$

CC array $\boxed{\begin{matrix} 0 & 1 & 2 & 2 & 4 & 4 & 4 & 5 & 6 & 7 \end{matrix}}$

Output (sorted) $\boxed{\begin{matrix} 2 & | & | & | & | & | \end{matrix}}$

* 1st element of original array is 2, so visit element at index 2, this element is the index where element of original array resides in the output sorted array.. we need to store the element at (index-1) index of output sorted array.

CC array becomes $\boxed{0 \mid 1 \mid 2 \mid 4 \mid 4 \mid 4 \mid 5 \mid 6 \mid 7}$

For element 9

array $\boxed{2|9|7|4|1|8|4}$

carray $\boxed{0|1|1|2|4|4|4|5|6|7}$

output array $\boxed{2|1|1|1|9} \quad 7-1=6$

carray $\boxed{0|1|1|2|4|4|4|5|6|6}$

Becomes

For element 7

array $\boxed{2|9|7|4|1|8|4}$

carray $\boxed{0|1|1|2|4|4|4|5|6|6}$

output array $\boxed{2|1|1|7|9} \quad 5-1=4$

carray $\boxed{0|1|1|2|4|4|4|4|6|6}$

Becomes

For element 4

array $\boxed{2|9|7|4|1|8|4}$

carray $\boxed{0|1|1|2|4|4|4|4|6|6}$

output array $\boxed{2|4|7|9} \quad 4-1=3$

carray $\boxed{0|1|1|2|3|4|4|4|6|6}$

Becomes

For element 1

original array

2	9	7	4	1	8	4
---	---	---	---	---	---	---

carray

0	1	1	2	3	4	4	4	6	6
---	---	---	---	---	---	---	---	---	---

output

1	2	1	4	7	9
---	---	---	---	---	---

carray becomes

0	0	1	2	3	4	4	4	6	6
---	---	---	---	---	---	---	---	---	---

For element 8

original array

2	9	7	4	1	8	4
---	---	---	---	---	---	---

carray

0	0	1	2	3	4	4	4	6	6
---	---	---	---	---	---	---	---	---	---

output

1	2	1	4	7	8	9
---	---	---	---	---	---	---

carray becomes

0	0	1	2	3	4	4	4	5	6
---	---	---	---	---	---	---	---	---	---

For element 4

original

2	9	7	4	1	8	4
---	---	---	---	---	---	---

carray

0	0	1	2	3	4	4	4	5	6
---	---	---	---	---	---	---	---	---	---

output

1	2	4	4	7	8	9
---	---	---	---	---	---	---

sorted by
Count sort