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.

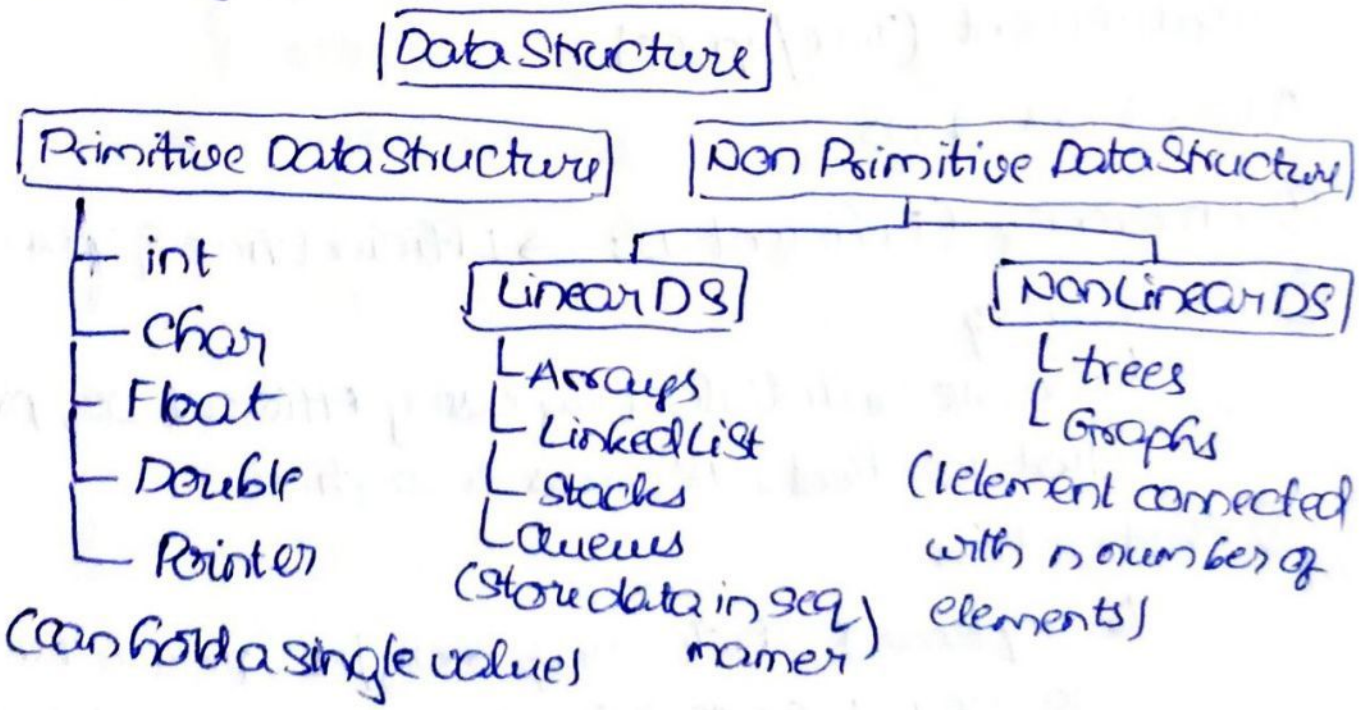
Examples

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 data type?

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



Data structures

↳ Static DS (Size allocated at compile time)
Maximum size is Fixed

↳ Dynamic DS (Size allocated at the 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 a particular ADT
* Ex: Stack (ADT) created/implemented using LL (DS)

ADT ⇒ Blueprint

DS ⇒ Implementation

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

Advantages of DS

1. Efficiency (Efficient DS → Efficient time & Space)
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: Elementry 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 Elementry Item: DataItem which are unable to Divide (EID)

⑤ Entity: Object that has a distinct identity (Person, Place)

⑥ Attribute: Characteristic of an Entity

Data structures => Allows us store & organise data

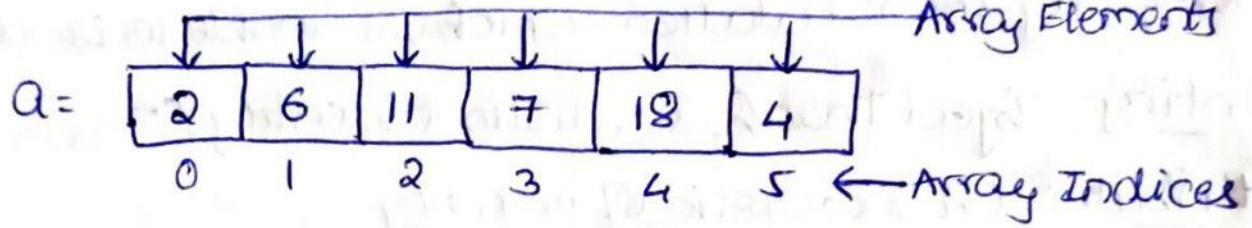
Algorithms => process the data meaningfully

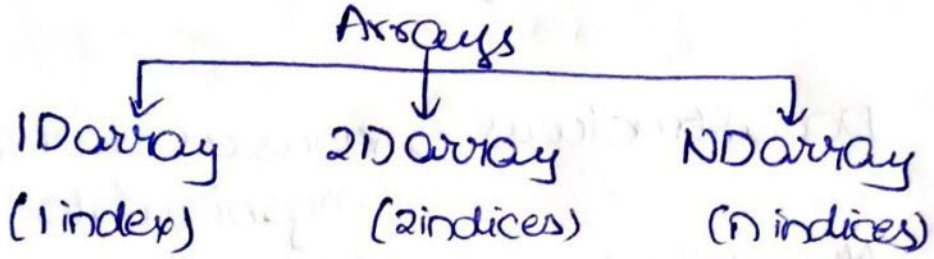
* Non Premitive Data Structure are data structures derived from Primitive DS
* NPDS forms set of data elements that is either Group of
  homogeneous / Hetrogeneous Dat structure
  - Same DT forms as grp
  - Diff DT forms as grp

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 auxilary storage for other data structs
③ Store data elements of Binary tree of Fixed count

② Linked List

* Singly linked list:
* Doubly Linked list
* Circular Linked list

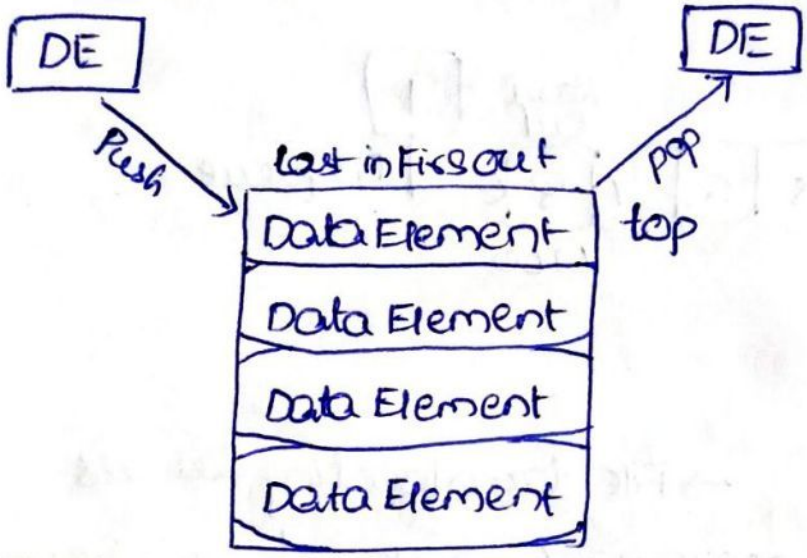
Applications of Linked List:

* 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:

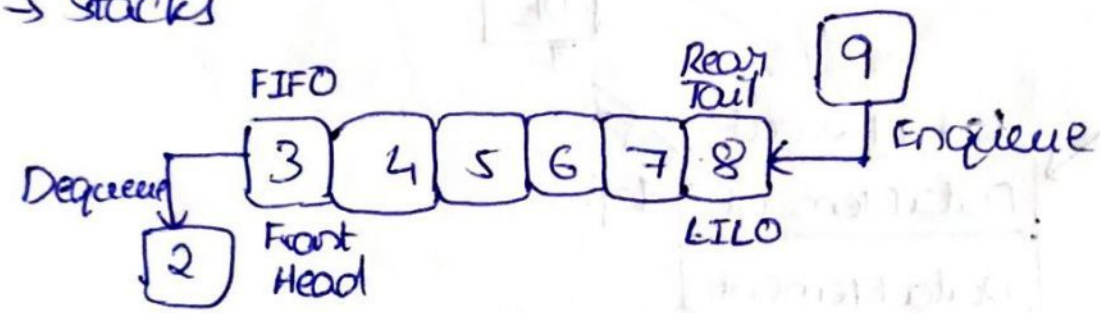
* Temporary storage for recursive operations, function calls nested operations
* Evaluate arithmetic Expressions
* Infix Exp → Postfix Exp
* Match paranthesis
* Reverse a string
* Backtracking
* DFS in graph & tree traversal
* undo/Redo-func

Queues

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

Can be implemented by
→ Arrays
→ LinkedLists
→ Stacks

Real life Examples
ticket counter
Escalator
car wash



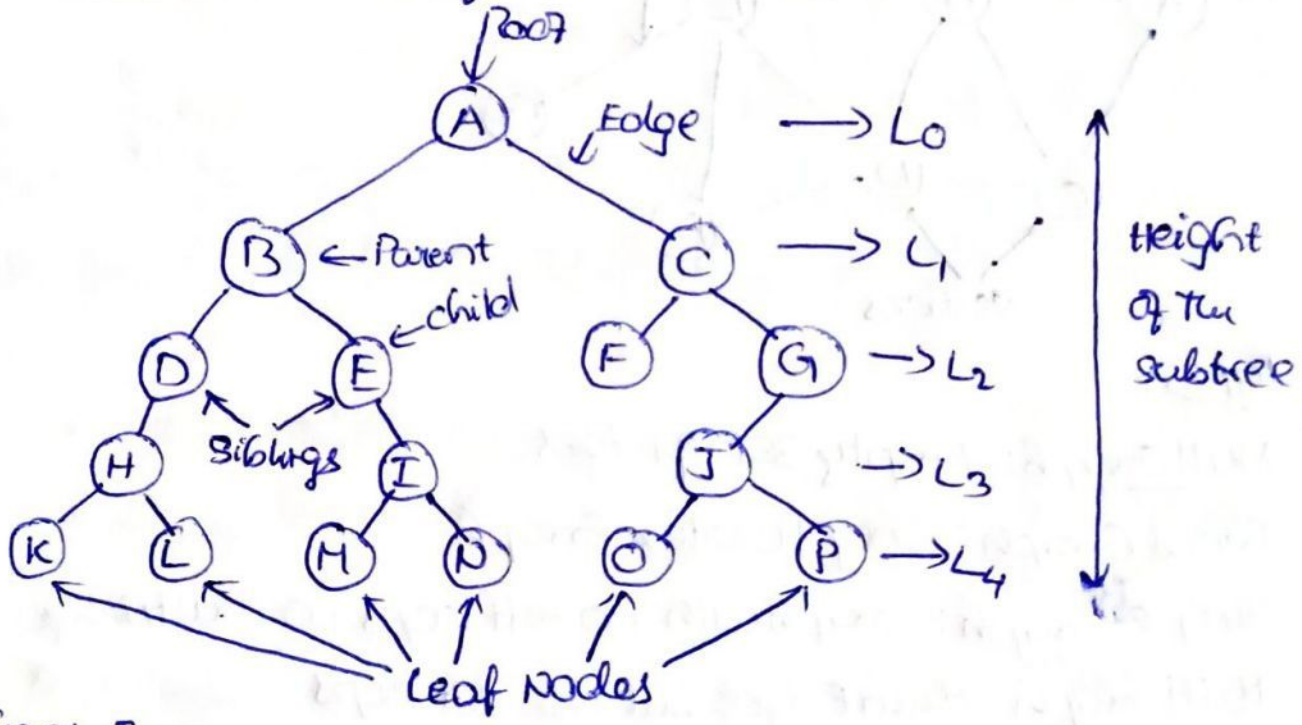
Applications
→ BFS in Graphs
→ File downloading Queues
→ Job scheduling operations
→ Handling interrupts generated by user apps
→ CPU/Job/Disk Scheduling

Non Linear Data Structure

* Data elements not arranged in sequential order.
* Insertion & Removal is not that easy, hierachial 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 → atmost 2 children

Binary Search Tree: can maintain sorted list of nums

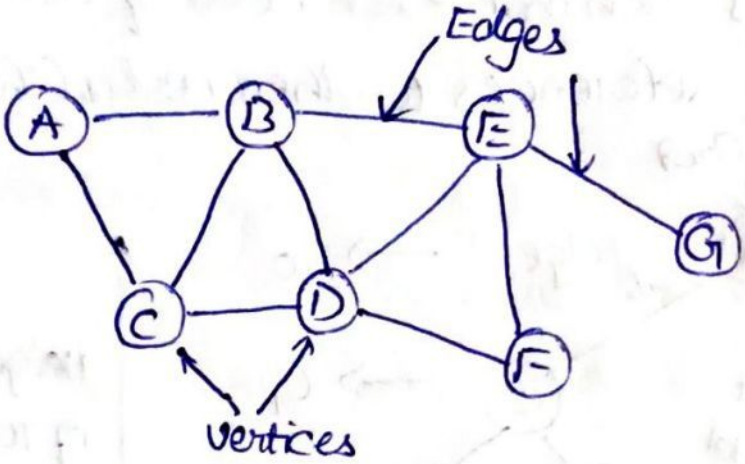
AVL Tree: Self Balancing Binary Search tree, Each node have Balance Factor (-1, 0, 1)

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

Graphs

* Finite nodes or vertices and the edges connecting them.

$G = (V, E)$ => 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 on 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

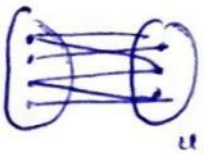
Cycle graph: Atleast 3 vertices and edges form a cycle

Cyclic graph: Atleast one cycle exists

Acyclic graph: zero cycles

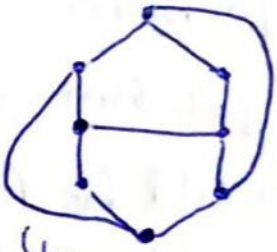
Finite / Infinite Graph: Finite / Infinite num of vertices / Edges

Bipartite Graph: vertices can be divided into 2 sets
SetA vertices can be connected to SB 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

Basic Operations of DS

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 of LL, user don't know the implementation that is main idea of Abstract Datatype

Applications of DS

→ Rep info of DB
→ search through org data
→ Generate the Data
→ Encrypt & Decrypt the data

Algorithm

* set of rules required to perform calculations or some other problem solving operations Especially by computer.
  → Flow chart
  → Pseudo code

Why Algorithms

* Scalability:
  we need to scale down a real world big prob 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 are
→ Modularity (Break problems into small chunks)
→ Correctness (Generate precise output with precise inp)
→ Maintainability (designed in simple structured way)

Some algorithmic approaches

① Brute Force algorithm:

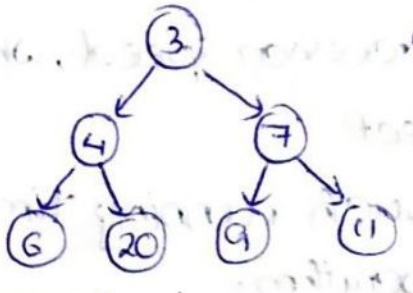
* searches all the possibilities to provide the required solution
  → Optimized method: Take best solution out of all solutions
  → Sacrificing: stops at first solution, doesn't care about optimized or not

② Divide and conquer

→ Divide bigger problem into smaller and solve them and merge the outputs 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 value
→ one of the case that would fail



Greedy Answer: 3+7+21 = 21
(local optimal)
Optimal answer: 3+4+20 = 27

When to use?

* Global optimum can be reached using local optimum
* 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 sub problems using memorization
* Find the optimal solution out of these subprob
* Reuse the result of subproblems, to not execute twice

⑤ Branch and bound algorithm:
* can be applied to only integer probles
* method of solving optimization problems by breaking them down into smaller sub problems and bounding 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

@ Priori analysis: Concider processor speed, which gave no implementation Effect

⑤ Posterior 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

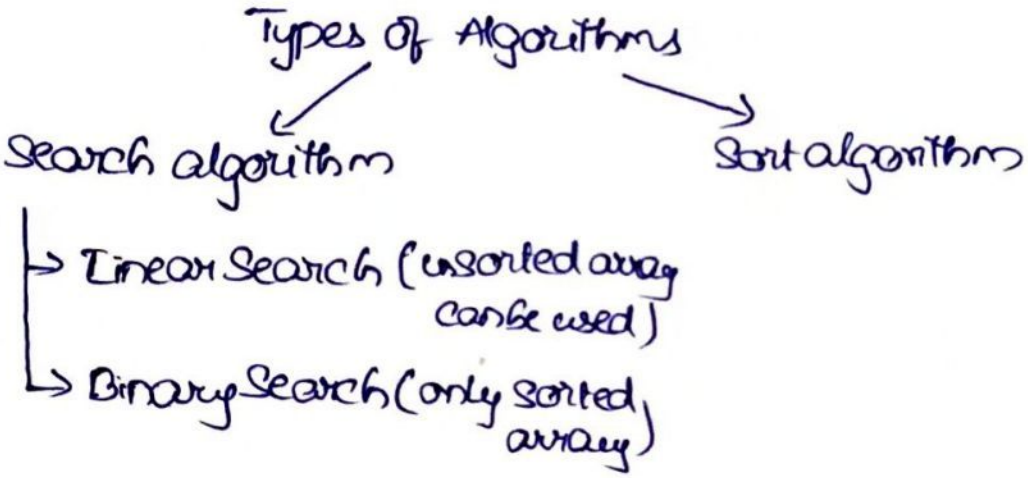
```
sum=0
for i in 1 to n:
    sum=sum+i;      } O(n)
return sum;         → O(1)
```

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.

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

Space complexity: Auxilary 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, Linkelist 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, bestcase and worst case scenario of an algorithm.

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

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

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

So T.C should be taken based on worst case

$n^3 \Rightarrow$ More Time complexity

$n^2 \Rightarrow$ comparitively less time complexity

If we have an algo which solves in n^2 & n^3 then we must go for n^2 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(logN)	log	⇒ Finding an entry in a sorted array
O(N)	linear	⇒ Finding an entry in an unsorted array
O(NlogN)	nlogn	⇒ Sorting n items by 'divide & conquer'
$O(N^2)$	quadratic	⇒ Shortest path b/w two nodes in graph
$O(N^3)$	cubic	⇒ Simultaneous linear equations
$O(2^N)$	exponential	⇒ Tower of Hanoi problem

Pointer

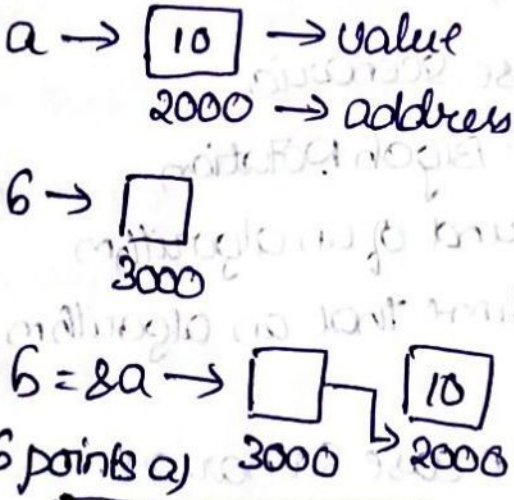
* Pointer is used to points the address of the value stored anywhere in computer memory
* obtaining such value using pointer is called dereferencing the pointer :

Mainly useful in

1. Lookup tables
2. Traversing String
3. control tables
4. Tree Structures

Pointer can be used in ways

1) Pointer arthematic
   => ++, --, +, -
2) Array of Pointers
3) pointer to pointer
4) passing pointer to function
5) Return pointer from func inc




```
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 structur_nm e1, e2, e3;
```

Advantages

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

Arrays

* collection of similar types of data items stored in contiguous memory locations.
* randomly accessable 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] = base address + size * (i - 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-1]=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)

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 arrays

Declare 2D array: int arr[max_rows][max_columns]

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

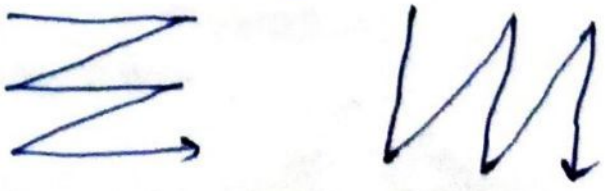
we must declare the size of the 2Dimensional array while we do initialization. This is not same while the case of single dimensional array.

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

Mapping 2D array to 1D array:

row idx ↓ \ → col idx	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

① Row Major ordering ② col major ordering



Locate address of 2D array : row/column 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

3×4 (m×n)

Row major order:

Addr(A[2][1]) = [Lo + [i×n + j] × w] (indices starts from 0)

Lo => Base address

Addr(A[2][1]) = [100 + [2×4 + 1] × 2]

Addr(A[2][1]) = 100 + 18 = 118

Addr(A[i][j]) = Lo + [(i-1)×n + (j-1)] w [index starts with 1]

Column Major order

Add[A[i][j]] = Lo + [j×m + i] × w (index starts with 0)

Add[A[1][2]] = 100 + [2×3 + 1] × 2

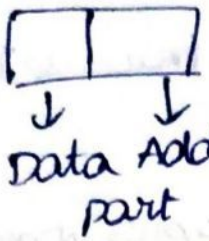
= 100 + 14

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 pointer to null
* Every link contains a connection to another link.

Why linkid 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 structure
* 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 xtra than array)
* Traversal and random accessing is complicated
  - Difficult, if we want to access last element, we need to traverse the whole LL. no possible of random accessing.
* Reverse traversing is needed for Backtracking but occupies more

Applications

* polynomial representation & sparse matrix repr
* Implement
  - → Stack → Queue → Tree
  - → Adjancy list → Dynamic Memory alloc

Time Complexity

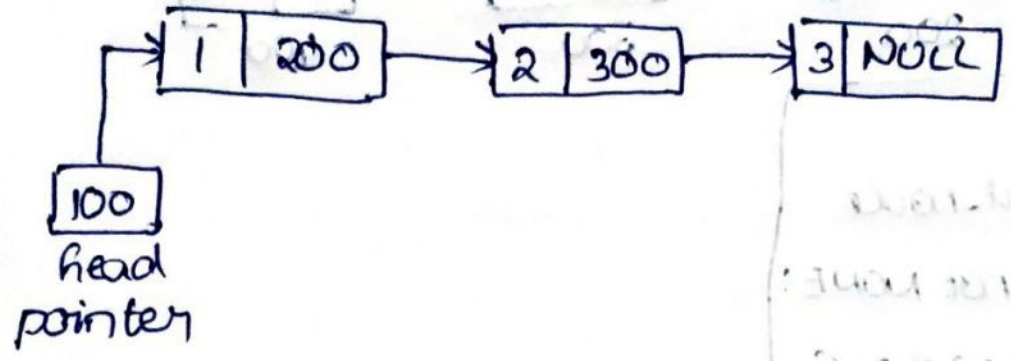
	Avg case	worst case
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

Space Complexity

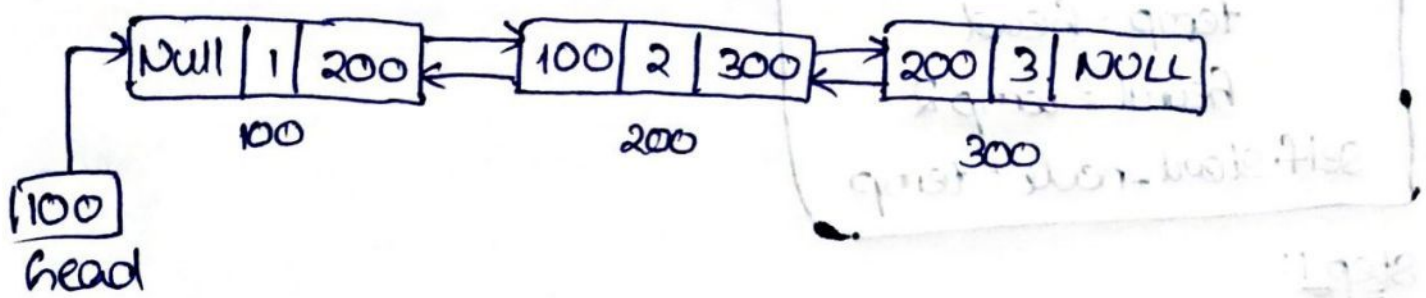
* Space Complexity of Linked List is O(n)

Types of Linked Lists

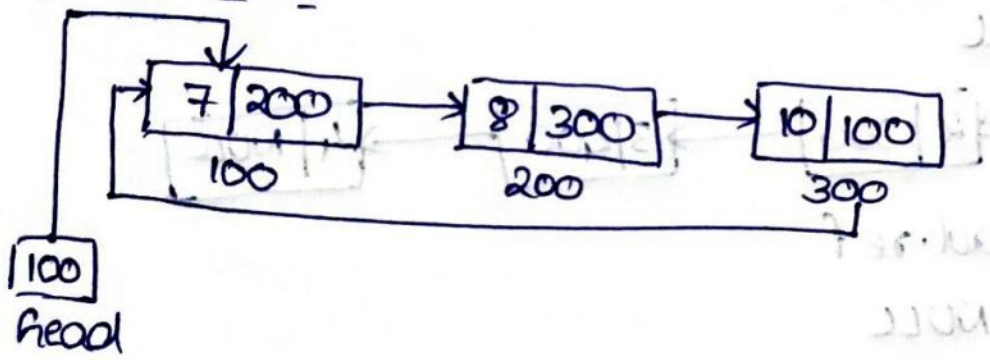
Singly Linked List (Single Link)



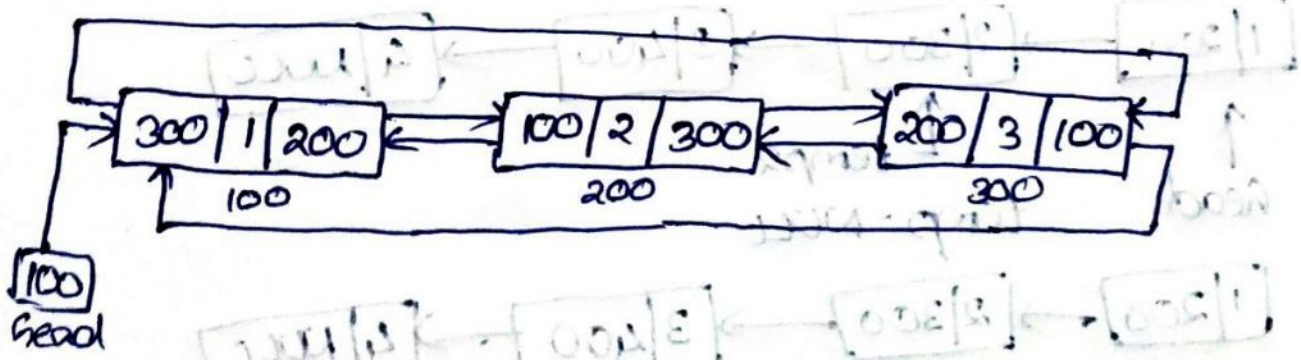
Doubly Linked List



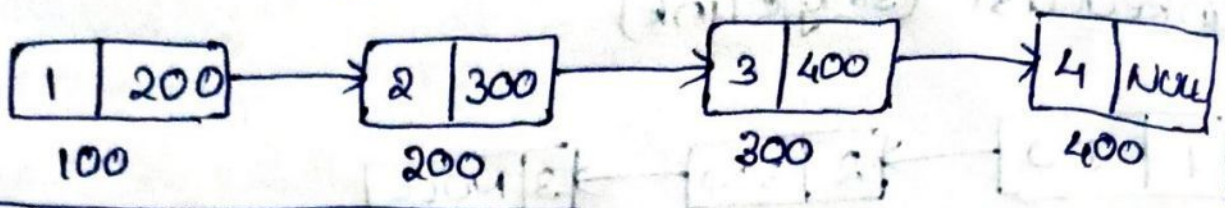
Circular Linked List



Doubly Circular Linked List



Reverse Linked List

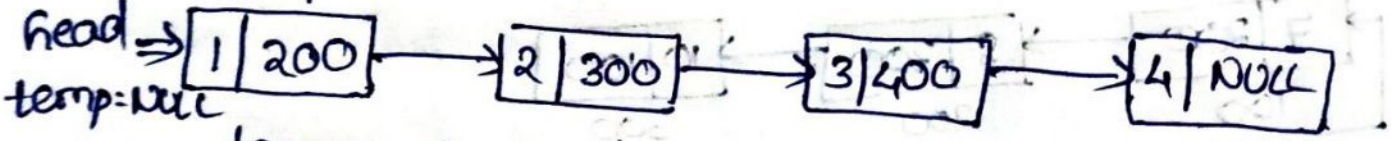



```
temp = NULL
head = self.start_node
while head is not NONE:
        temp2 = head.ref
        head.ref = temp
        temp = head
        head = temp2
self.start_node = temp
```

step 1:

head = start_node

temp = NULL



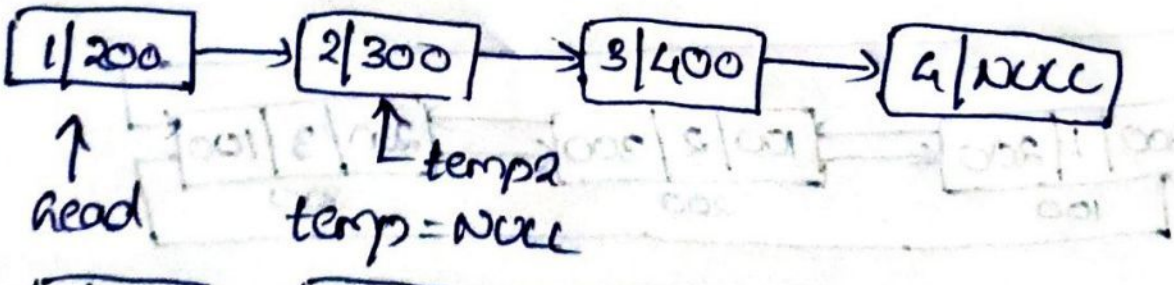
temp2 = head.ref

head.ref = NULL

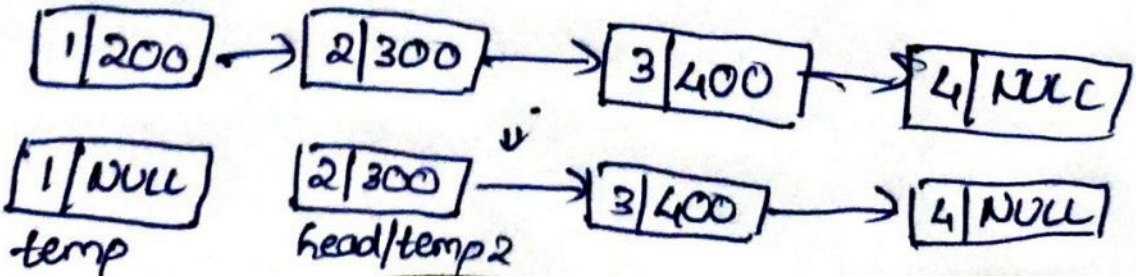
temp = head

head = temp2

step2



step3:



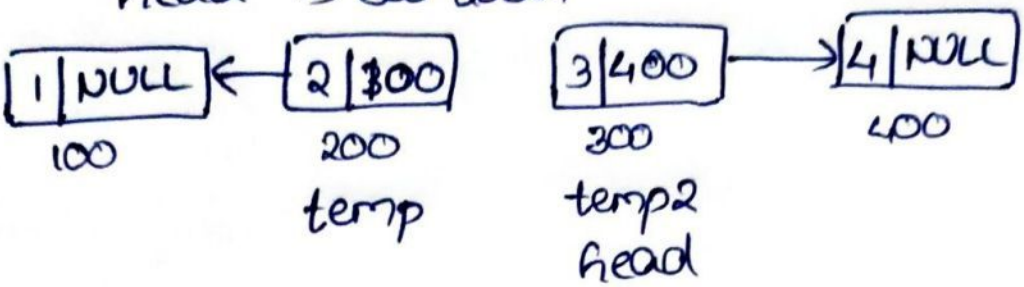
step4:

temp2 => 300 addr

head.ref => 100 addr

temp => 200 addr

head => 300 addr



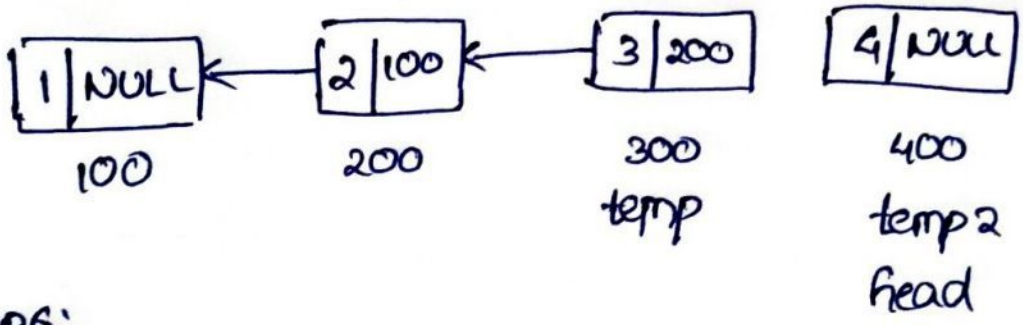
step5:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



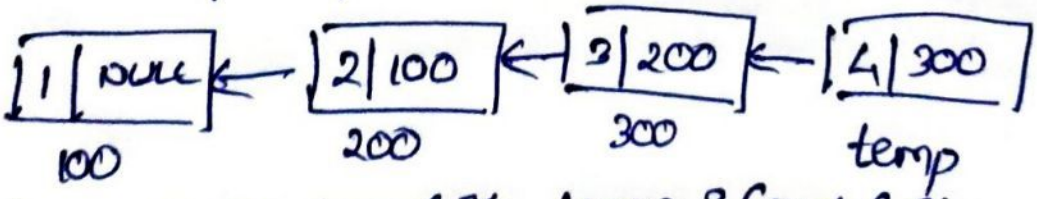
step6:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



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

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

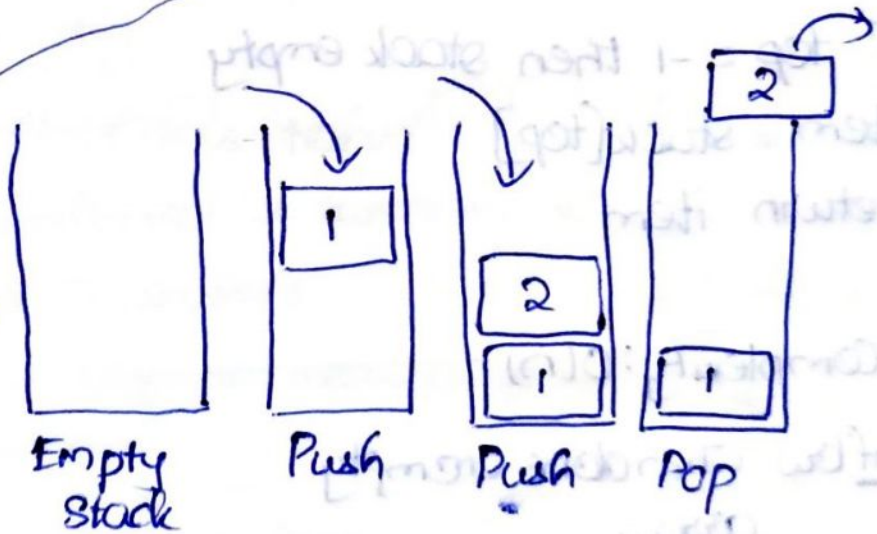
	Average	Worst
Access	O(n)	O(n)
Search	O(n)	O(n)
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)

Space complexity for SLL: O(n)

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.

push()	count()
pop()	change()
isEmpty()	display()
isFull()	
peek()	



Applications

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

Stack (arrays)

Push operation (insert)

```
begin
    if top = n then stack full
    top = top + 1
    stack (top) := item;
end
```

Time Complexity: O(1)

Overflow: inserted more than the limit

Pop operation (Delete)

```
begin
    if top = 0 then stack empty
    item = stack (top)
    top = top - 1
end
```

Time Complexity: O(1)

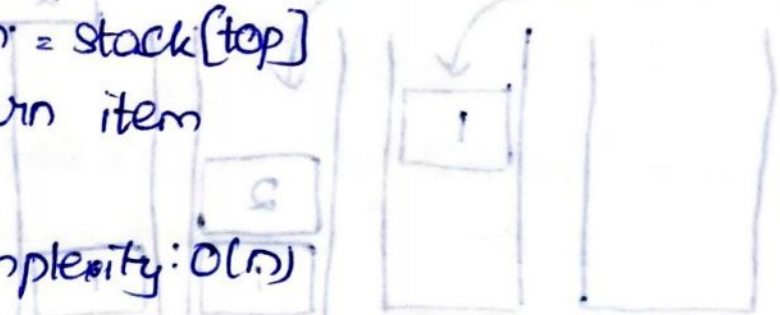
underflow: removing more than the limit (From my stack)

Peek operation (Display top)

```
begin
    if top = -1 then stack empty
    item = stack[top]
    return item
end
```

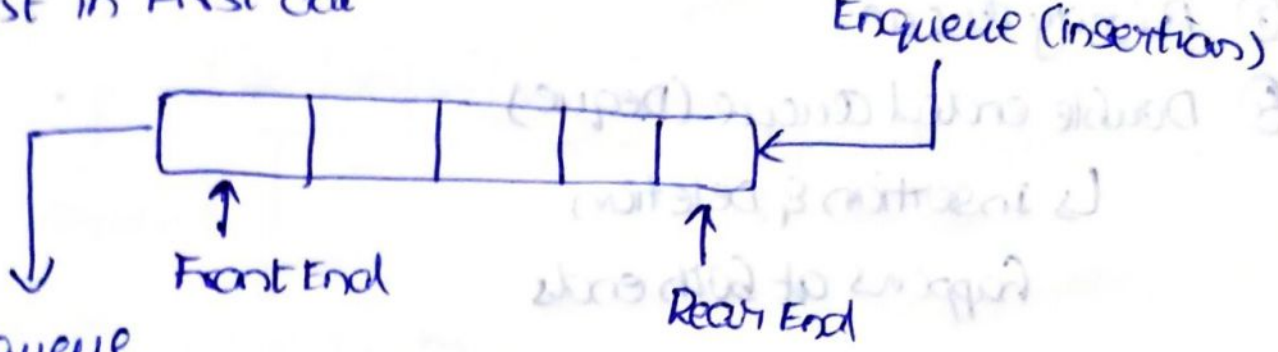
Time Complexity: O(n)

underflow: when done on empty array



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 IO, sockets
* used as buffers (Eg: MP3 mediaplayer, CDplayer)
* playlists in media player.

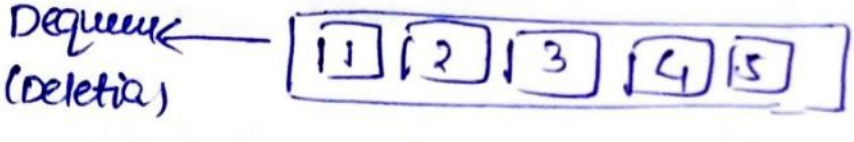
	Queue	
	Average	worst
Access	O(n)	O(n)
Search	O(n)	O(n)
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)

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:



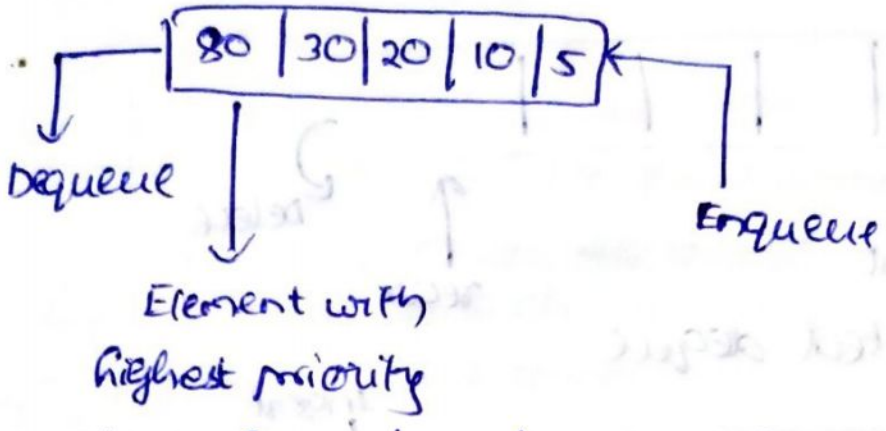
* Even if 3 of 5 elements are removed, even if space is available, we can't enter new elements
* overflow condition: If the rear pointer is pointing to the last element of the queue

Circular Queue:

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

Priority Queue

* Elements are arranged based on priority.
* Every element have priority associated with it. If 2 are of same priority, FIFO rule applied.



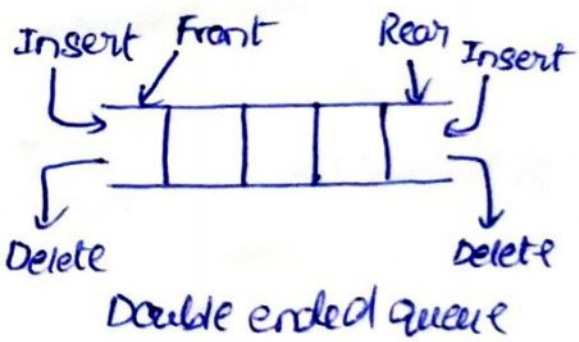
Insertion: Based on the arrival
Deletion: Based on the priority

Ascending priority queue: 7 | 5 | 3
Insert in any order 7,5,3
Smallest can be deleted first 3,5,7

Descending priority queue: 7 | 3 | 5
Insert in any order 7,3,5
Delete largest first 7,5,3

Deque (Double ended queue)

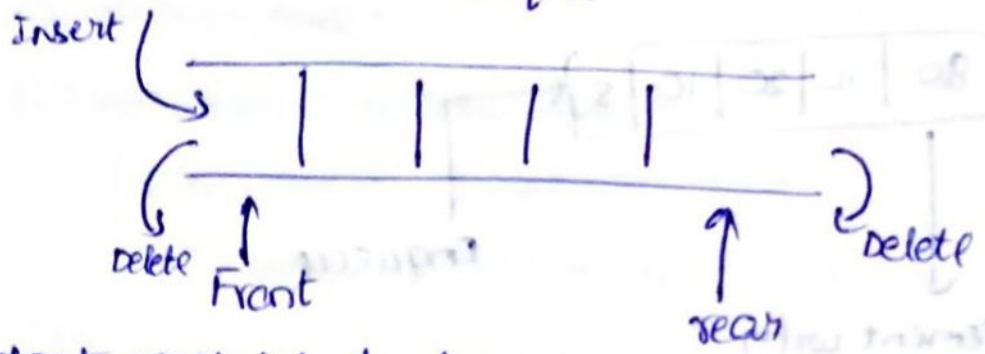
* Can insert and delete from both front and rear ends of the queue.
* palindrome checker
* Deque not performs FIFO, as this is illogical for queue rules
* Deque only performs LIFO, basically it can act as stack & queue



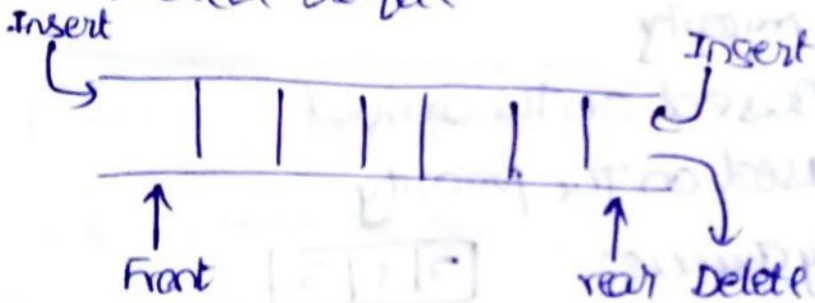
Double ended queue

Types of deque

* Input restricted deque
* output restricted deque
* Input restricted deque



* output restricted deque



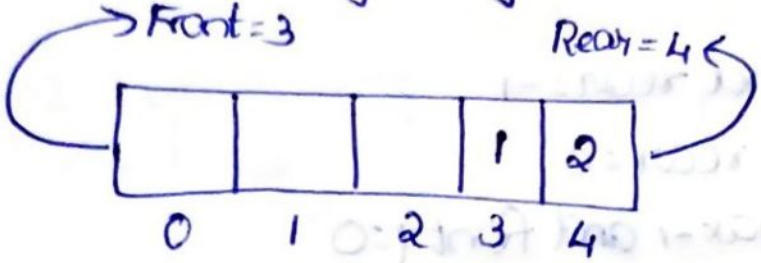
operations performed on queue

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

Queue can be implemented through array and linkedlist

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

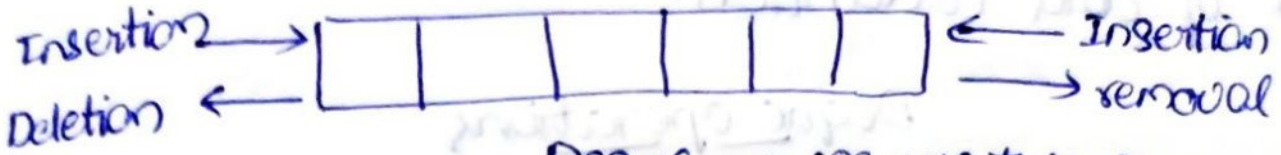
```
Step_1: if (rear+1)%.MAX = FRONT
            write "Overflow"
            Goto Step4
        [END OF IF]
Step_2: if front = -1 and rear = -1
            set front = rear = 0
        else if rear = max-1 and front != 0
            set rear = 0
        else
            set Rear = (rear+1)%.max
        end of if
Step_3: set Queue[Rear] = val
Step_4: exit
```

Deque an element in circular queue

```
Step_1: if front = -1
            write "underflow"
            Goto Step4
        end of if
Step_2: set val = Queue[Front]
Step_3: if front = rear
            set front = rear = -1
        else
            if front = max-1
                set front = 0
            else
                set front = 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.

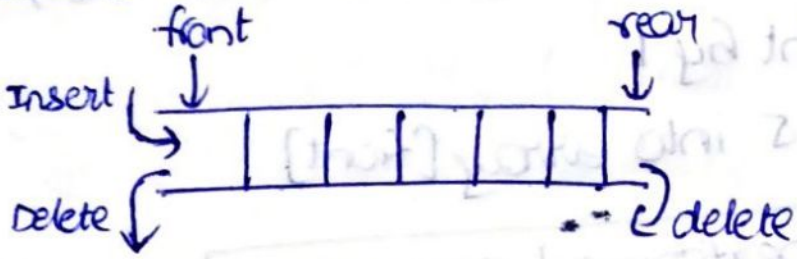


Deque

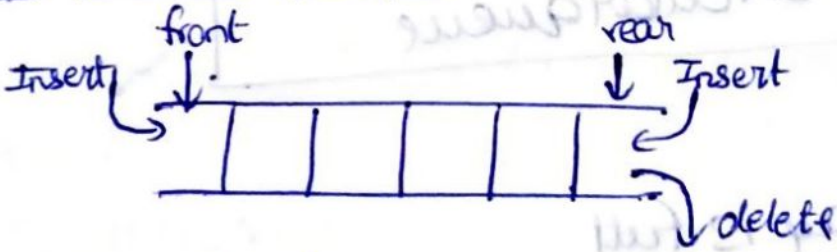
① Input restricted queue
② output restricted queue

* we use this when we need quicker append and pop operations from both ends
* O(1) time complexity for append and pop from both ends
* List performs this action in O(n) time complexity
* we are having an rotate function in which we rotate the deque array to left/right

"Input restricted" queue



"output restricted" queue



Applications of deque

* Deque can be used as both stack and queue as it supports both operations.
* Can be used as palindrom 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 seperate threads, without a risk of data corruption.

Deque operations

Insert at front

* Check the position of front
* If front < 1, reinitialize 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 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) (ie 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 (ie. front = n-1), set go the 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 conditon)
* 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

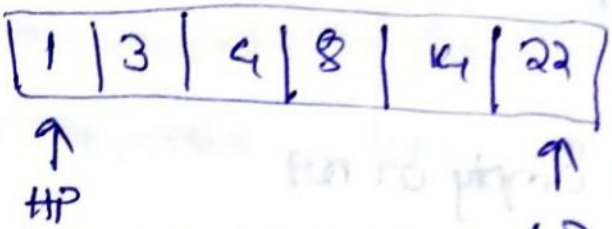
* (If front = 0 and rear = n-1) or (front = rear + 1)

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

Priority Queue

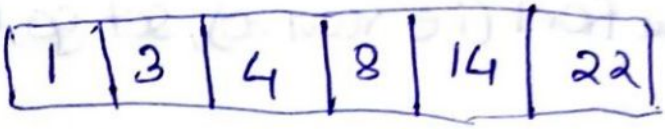
* Priority Queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority.
* thighest priority will come to the first of the queue. It will be removed first.
* Allows only comparable elements, arranged either in ascending order / descending order.

P. Queue



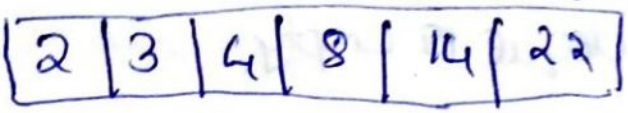
* thighest priority will be deleted first than the lowest ones
* If two elements in priority queue are of same priority then they will be arranged in FIFO principle.

Example:



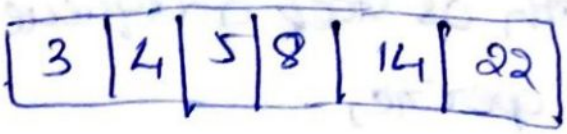
poll() => removes 1

add(2) => add 2 in front of 3 (asc order)



poll() => removes 2

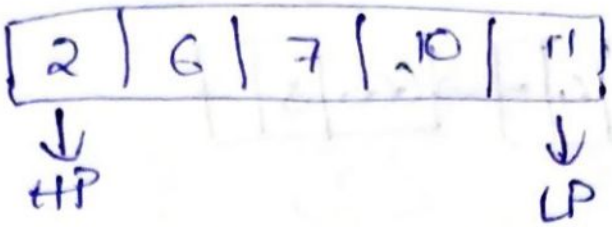
add(5) => adds in blw 4 & 8 (asc order)



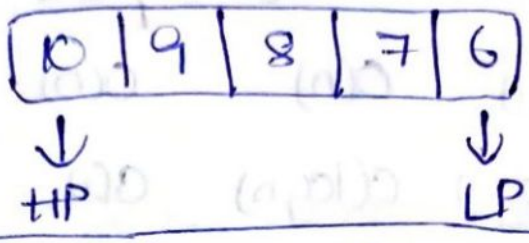
Types

① Ascending order priority queue

* lower value given highest priority



② Descending order priority queue



high PN = thigher Priority
↳ DSC PQ

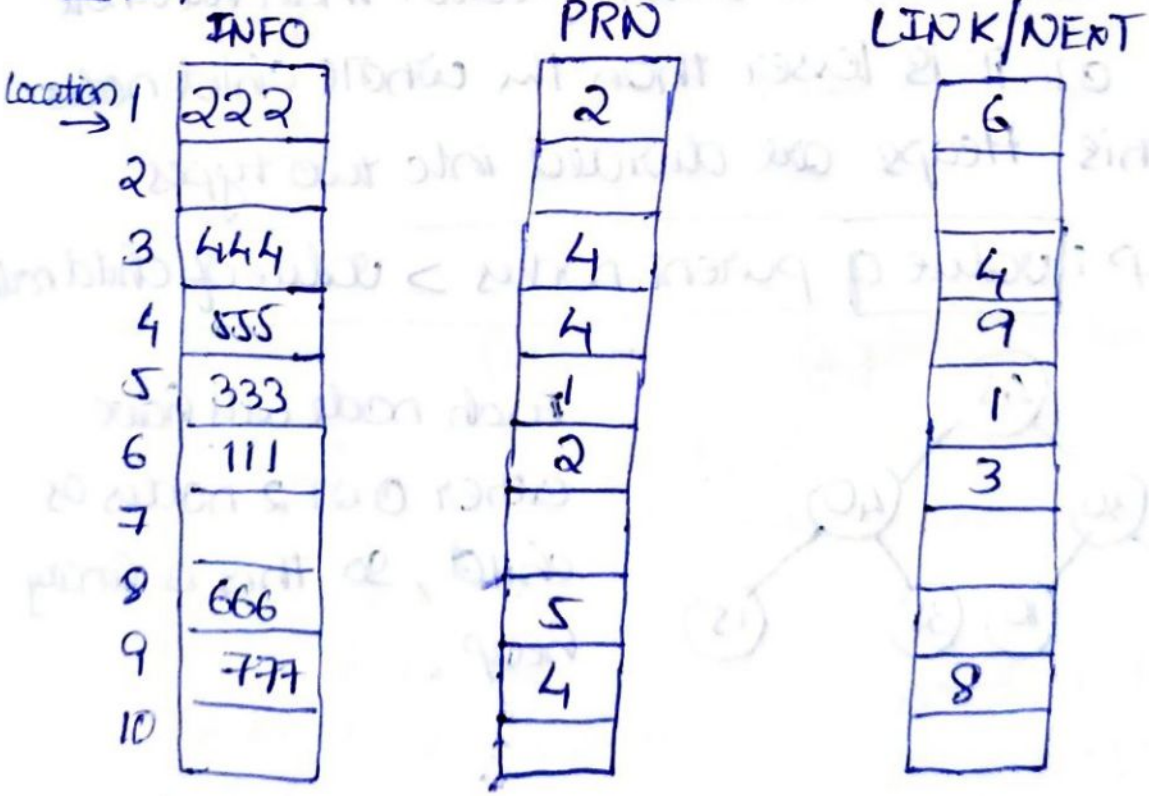
Lower priority number = highest priority. → ASPQ

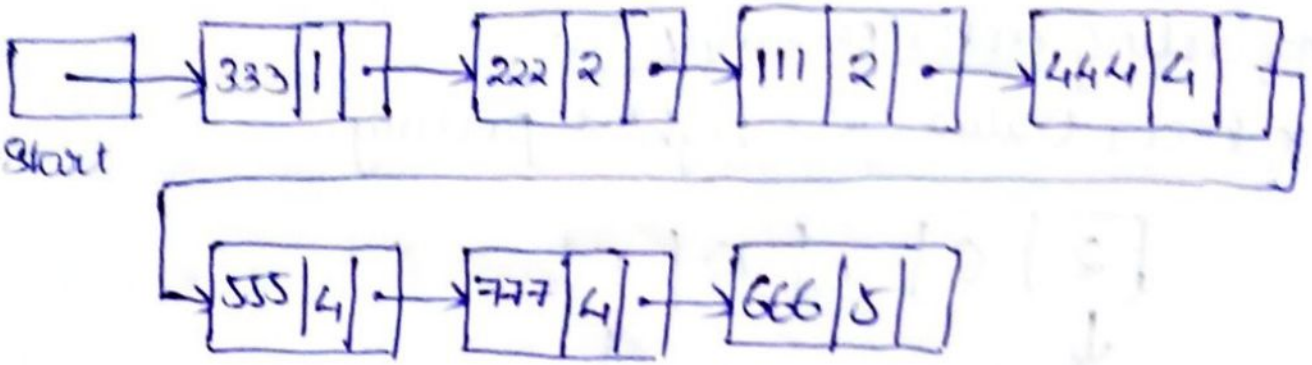
Priority queue's implementation

① arrays ② Linked List ③ heap data structure

④ Binary Search tree

Examples



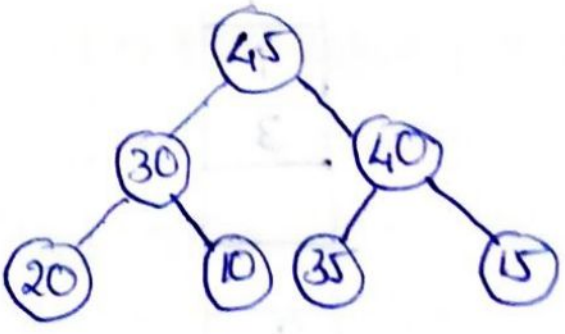


Complexity analysis

Implementation	add	remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(log n)	O(log n)	O(1)
Binary search tree	O(log n)	O(log n)	O(1)

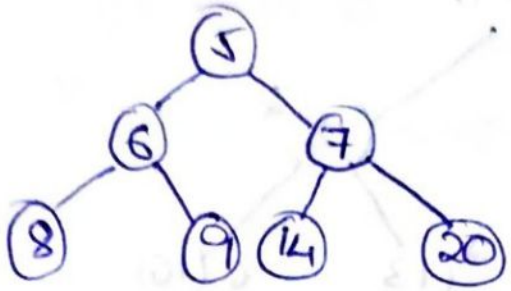
* Priority queue implementation through heap is efficient
* Heap is tree based data structure that forms a complete binary tree and satisfies heap property
* value of the root node is either greater than the whole child nodes or it is lesser than the whole child nodes
* Based on this Heaps are divided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

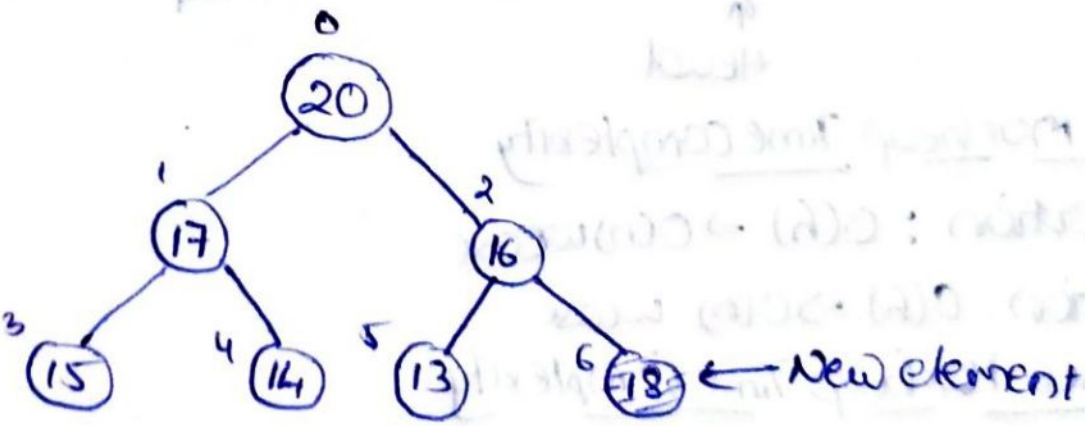
→ Min heap: value of parent node is less than value of child nodes



Operations on priority Queue

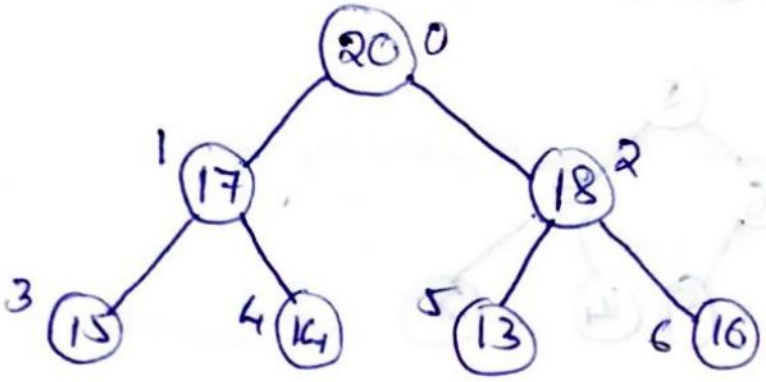
* Inserting an element in priority queue (max heap)
  → If we insert an element in priority queue, it will move to the empty slot by looking from top to bottom and left to right.
  → If the element is not in a correct place then it is compared with the parent node, if it is found out of order, elements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

⇓ Heapify



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

Insertion: adding element last of PQ array ⇒ O(1)

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

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

Insertion: adding a node at ~~first~~ last ⇒ O(1)

Deletion: deleting a node at last ⇒ O(n)

array ⇒

16	3	5	6	9	2	11	4	7

Linked list ⇒

BST Maxheap Time complexity

Insertion : O(h) ⇒ O(n) wCase

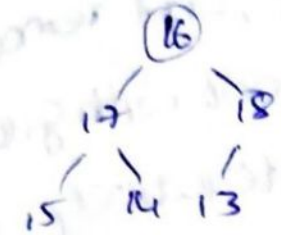
Deletion : O(h) ⇒ O(n) wcase

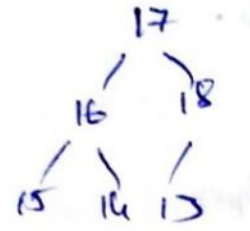
AVL Tree Maxheap Time Complexity

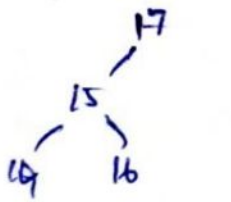
Balanced tree

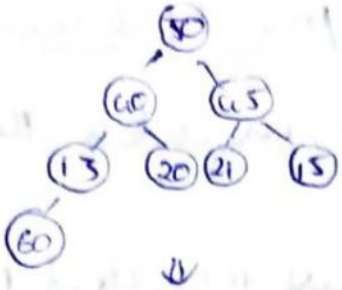
Insertion : O(h) ⇒ O(log n)

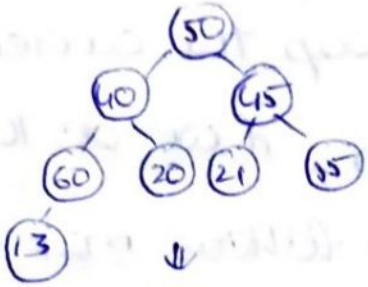
Deletion : O(h) ⇒ O(log n)

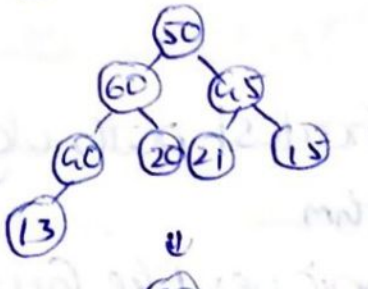


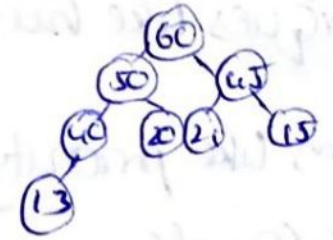






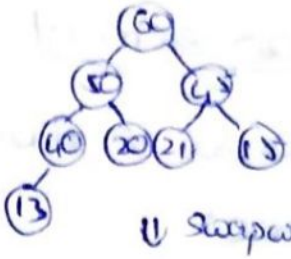


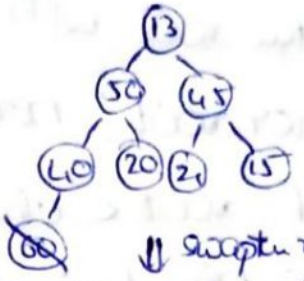


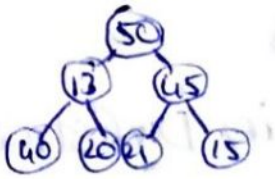


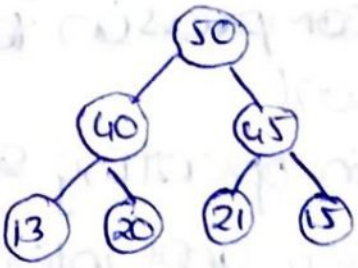
Insertion

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 root < children, if we do this iteratively we get a new tree which follows max heap.

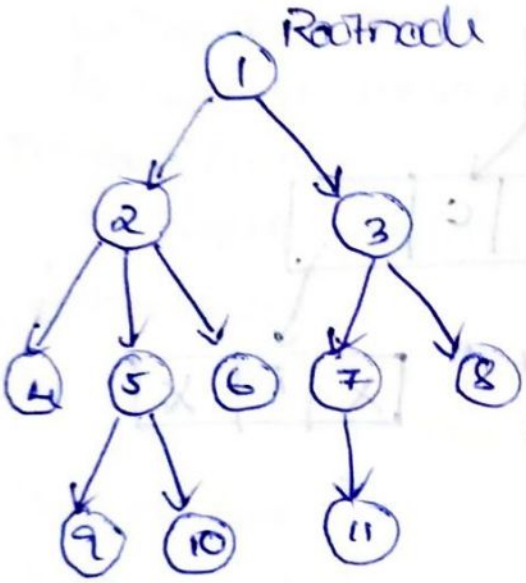
Applications

* 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 hierarchial data.
* root is at top, Trees are effecient to store elements in hierarchial way.
* Non linear data structure

Tree Example:



→ ⇒ Links

Root:
* Top most node in tree hierarchy

Child node:
* If a node is descendent of any node

Siblings:
* node that have same parent

Leaf node:
* node which doesn't have any descendents

Internal node:
* node atleast have one child node

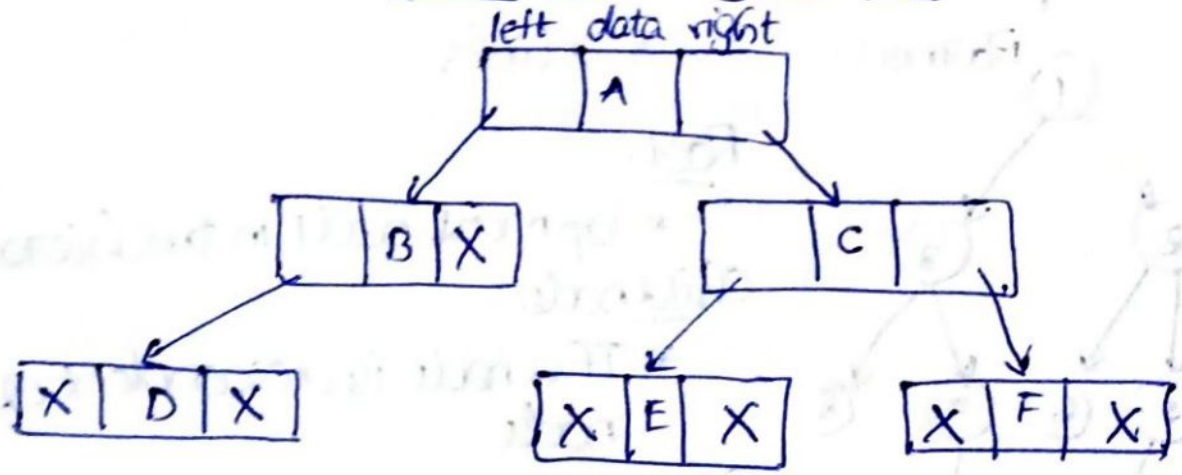
Ancestor node:
* Predecessor node on path from root to that node.
* 1, 2, 5 are ancestor nodes of node 10

Descendent
* immediate successor of the given node

Properties of tree data structure

* Recursive data structure
* number of edges
  - If we have n nodes ⇒ 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 data of the node
Field 3 consists address of right node

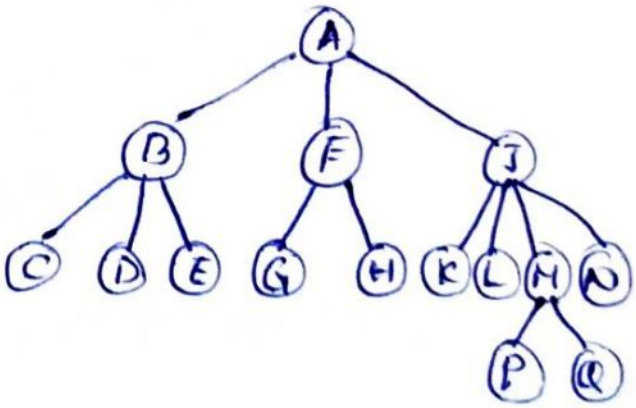
Applications of trees

* Storing naturally hierarchial data
  - ↳ Ex: File system (Files, Folders)
* Organize data
  - ↳ Effecient insertion, deletion, searching only possible in trees.
  - ↳ logN 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

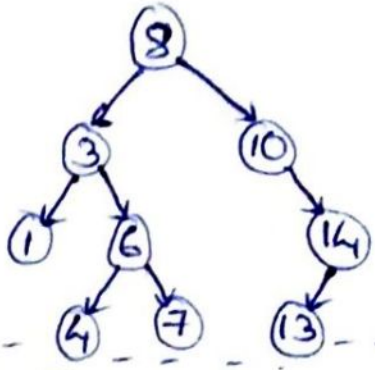
① General Tree

* 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)



Binary Search tree

* A node can be connected to utmost two child nodes
* Every node in the left subtree must contain a value less than the value of the root node, and value of each node in right subtree must be bigger than the value of root node.

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 btw the height of left subtree and height of right subtree
* It should be either 0, +, -

Red Black tree

* It is binary search tree
* Binary search average case $\log_2 n$
  Best case O(1)
  Worst case 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

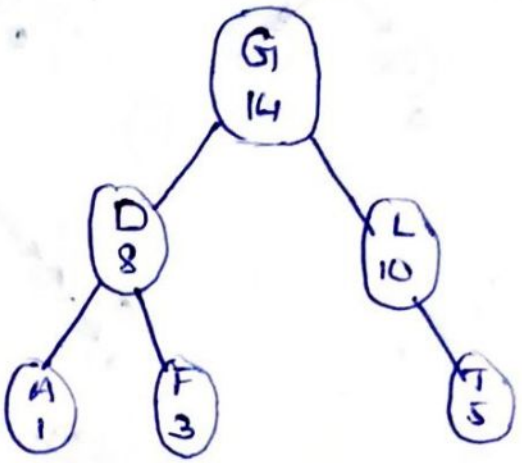
Splay tree

* 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.

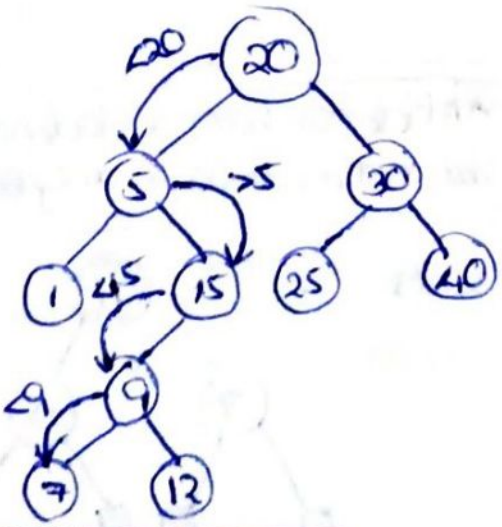
Treaps

* 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

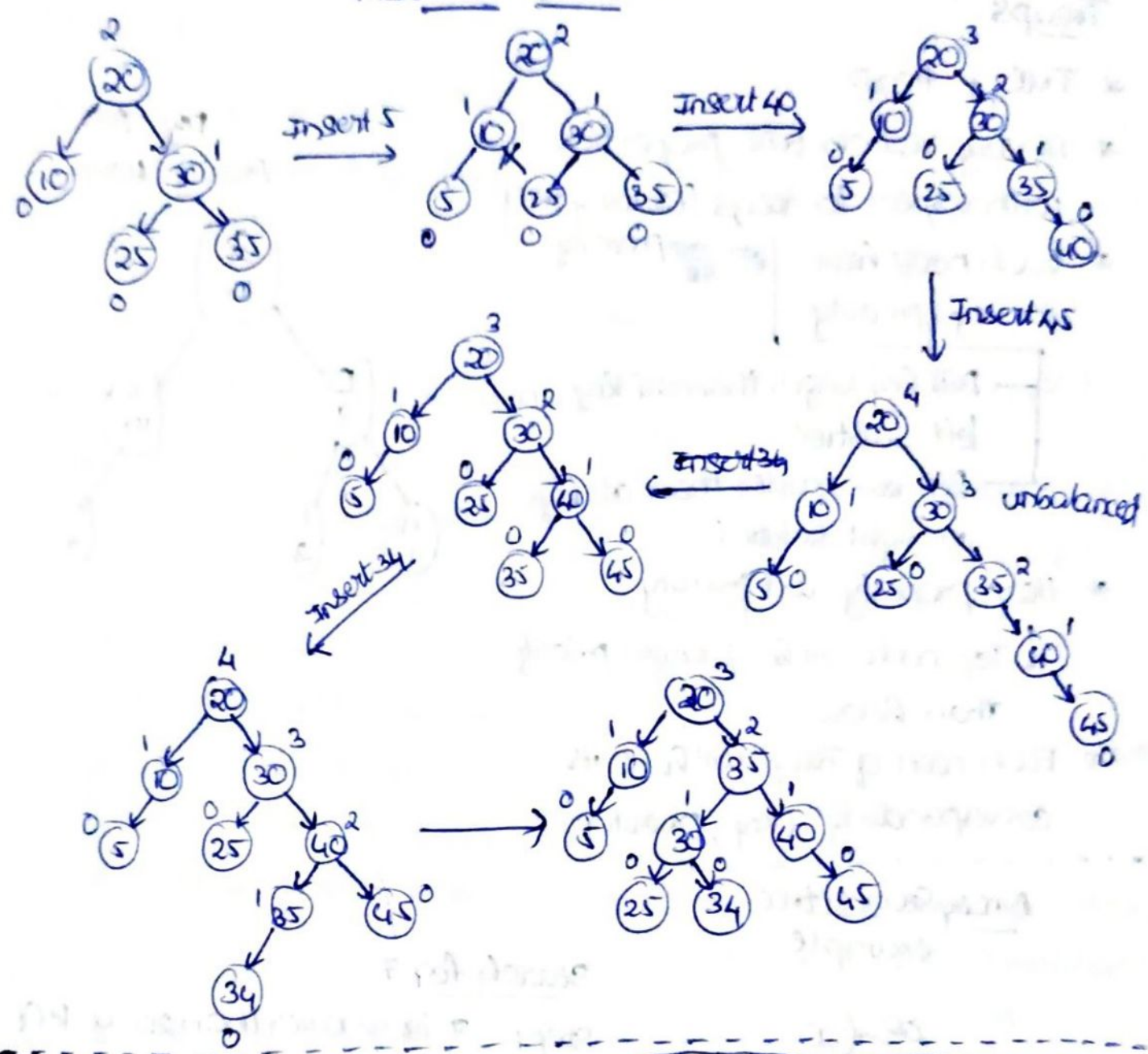


Searching in BST

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

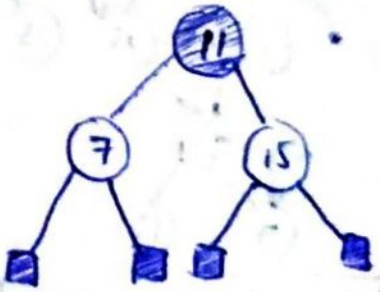
Insertion in AVL tree



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 => O(log n)

Properties of Binary tree

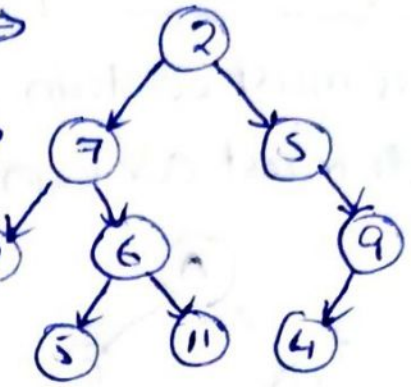
① maximum number of nodes at level l is $\boxed{2^{l-1}}$

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

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

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

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



② Total number of nodes present in binary tree

For example,

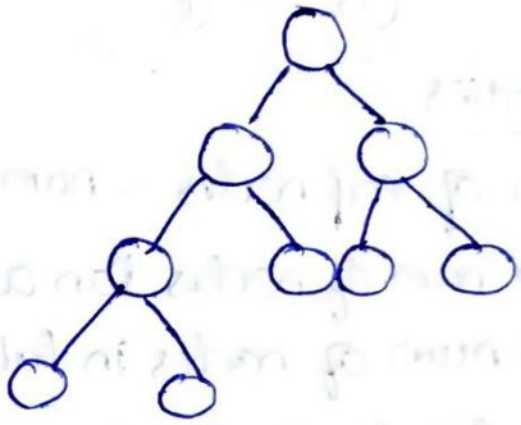
For a binary tree with which height 3, we will have max of 2^{h-1} nodes.

$n=1$

$n=2$

$n=3$

$n=h$



$$\sum_{l=1}^{l=h} 2^{l-1} = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}$$

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

② 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 a full binary tree $2 \times h - 1$

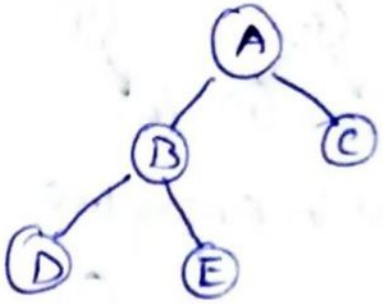
⑥ Maximum height of a full binary tree can be computed as

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

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

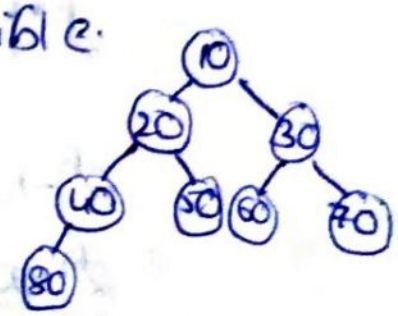


Properties

* num of leaf nodes = num of internal nodes + 1
* max num of nodes can a binary tree can have $\Rightarrow 2^h - 1$
* min num of nodes in full binary tree $\Rightarrow 2h - 1$
* 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.
* nodes should be added from left



Properties

* max num of nodes $\Rightarrow 2^h - 1$
* 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 = the sum of the number of nodes at each level from 0 to d.

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

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

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

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

* Finding depth based on total number of nodes

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

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

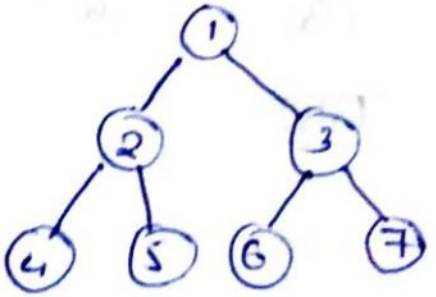
$$\log_2 2^{d+1} = \log(S+1)$$

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

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

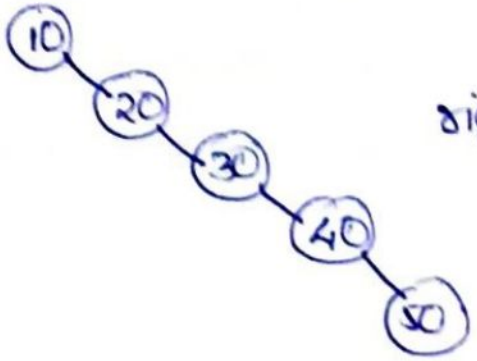
- Perfect Binary tree

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

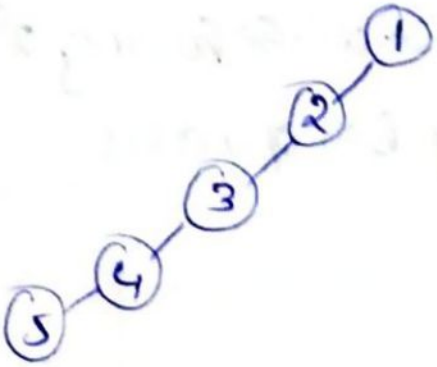


Degenarate Binary tree

* All internal nodes have only one children



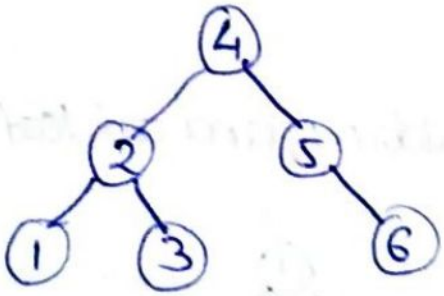
right skewed tree



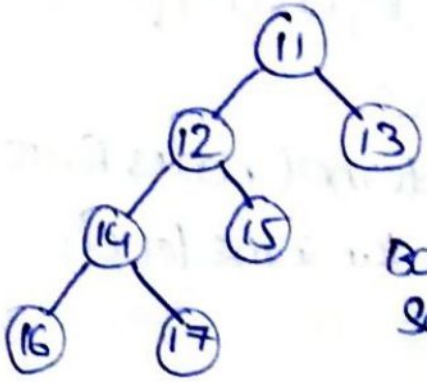
left skewed tree

Balanced Binary tree

* Tree in which both left and right trees differ by atmost 1
* Examples: AVL trees & Red Black tree



BBT



Balance Factor is >1
So, not a BBT

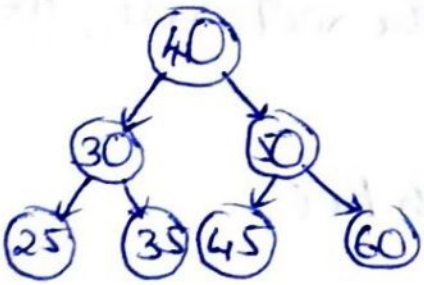
Not BBT

Properties of binary search tree

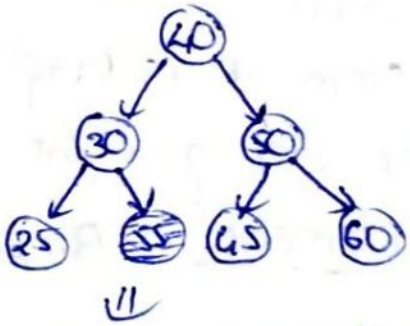
1. no duplicate elements in binary search tree
2. element at the left child of node is always less than the element at the current node
3. left subtree of a node has all elements less than the current node.
4. Element at the right child of a node is always greater than the element at the current node.
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

BST



Not BST



40 < 35 ⇒ Actually BST mean values of left subtree must be lesser than the root node. Here its the opposite, so not BST

Advantages of BST

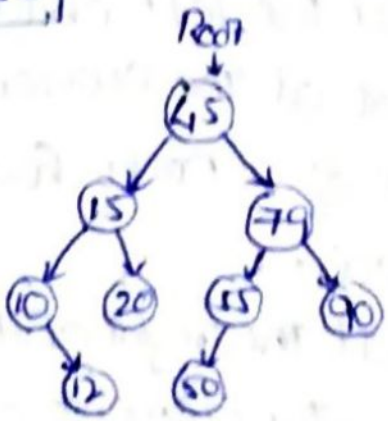
* Searching in BST is easier, as for each and every node, half search tree will be decreased.
* Insertion, deletion are faster in BST.

Array:

45	15	79	90	10	55	12	20	50

Insertion

Covert as BST



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 untill you get a leaf node to insert a new node
  or
  suitible 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 => if (item = root->data) or (root = NULL)
                    return root
              elseif (item < root->data)
                    return Search (root->left, item)
              else
                    return Search (root->right, item)
              endif
    Step 2 => END
```

Deletion in Binary Search tree

3 cases in Deleting a node from a tree

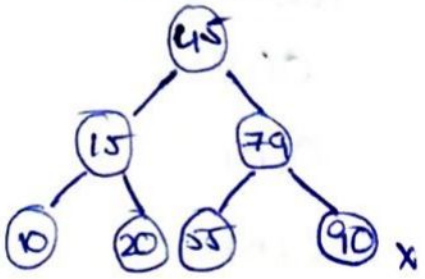
1. node to be deleted is a leaf node
2. node to be deleted has only child
3. node to be deleted has two children

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

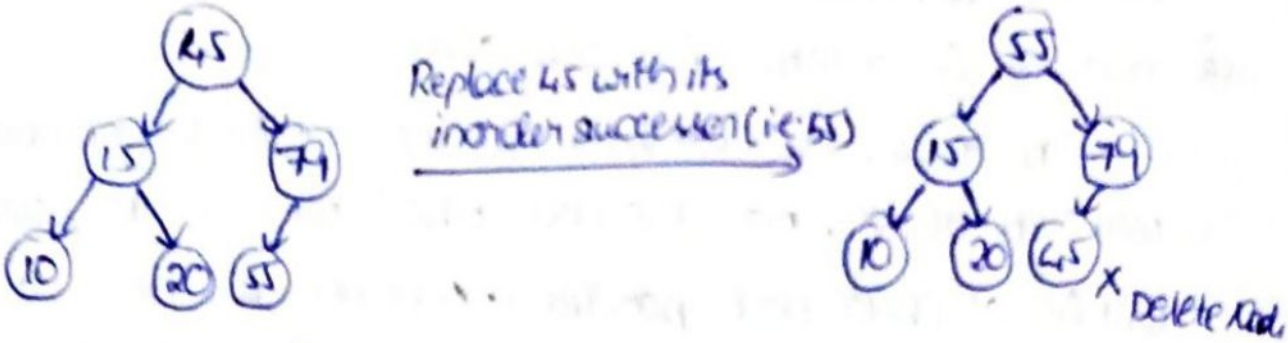
Delete 79 => swap 79 with it's child node and delete the new child.
i.e: 79 node

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.



	Time complexity	
	Best/Average case	worst case
Insertion	O(log n)	O(n)
Deletion	O(logn)	O(n)
Search	O(logn)	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 ascending order in inorder traversal

AVL Tree

* named after its founders "GM Adelson-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 balaced if the balance factor of each node is in between -1 to 1. Else tree is unbalanced or need to be balanced.

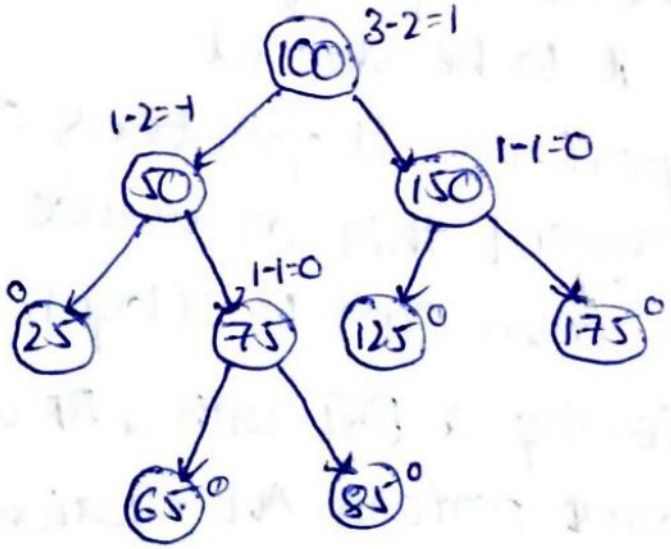
Balance Factor (k) = height(left(k)) - height(right(k))

1 ⇒ left subtree is one level higher than the right subtree

0 ⇒ left subtree and right subtree are of same height

-1 ⇒ left subtree is one level lower than the right

Example



AVL Tree

AVL Tree Time Complexity

Algorithm	Average & worst case T.C
Space	O(n)
Search	O(logn)
Insert	O(logn)
Delete	O(logn)

* 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(logn) 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.

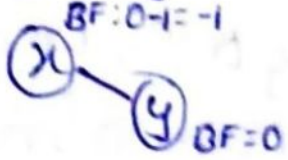
Example ①:

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

Step1 : insert x

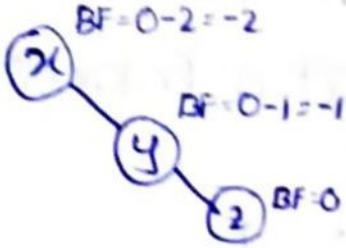
Balanced

Step2: insert y



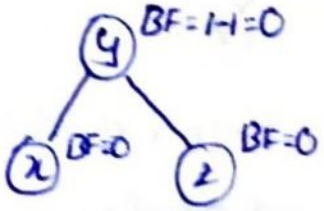
Balanced

Step3: insert z



(Not Balanced)

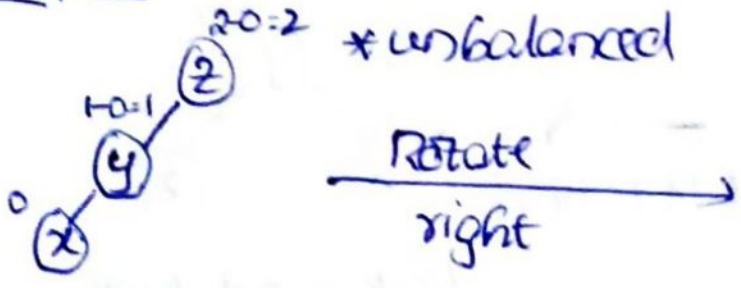
⇓ Rotate to make it Balance



* Rotate left

* Balanced

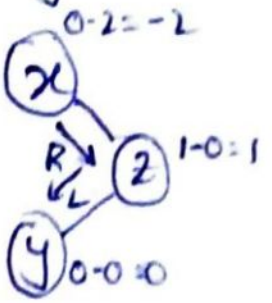
Example 2:



* unbalanced

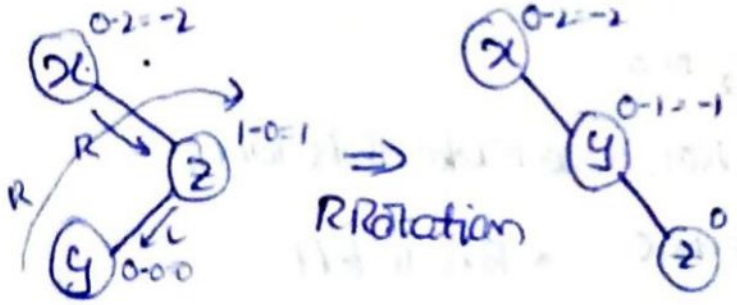
Example 3:

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

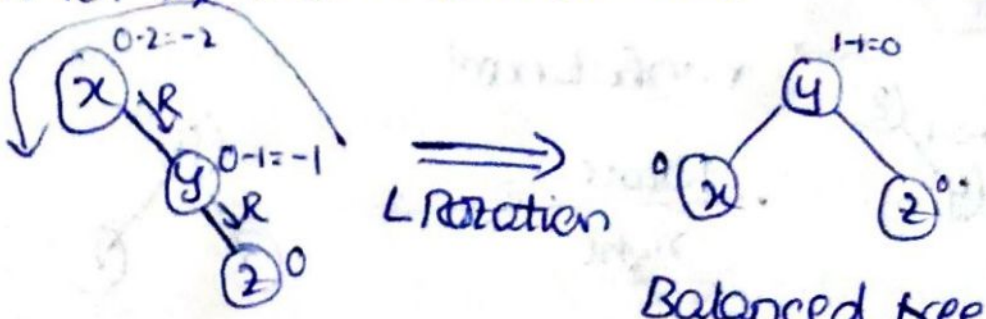


* Follows BST
* Doesn't follow AVL tree

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.

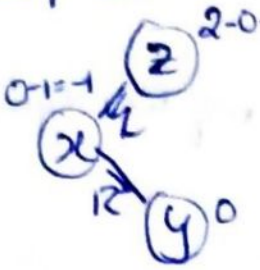


Balanced tree

AVL tree

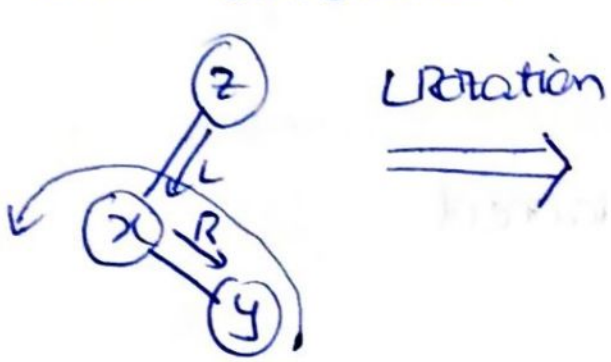
Binary Search tree

Example: z x y

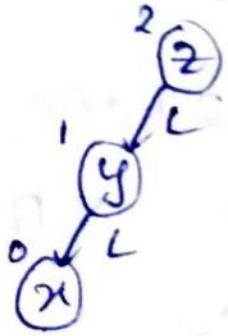


* unbalanced
* not AVL tree
* Left Right Rotation

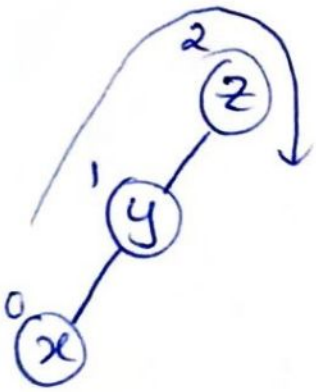
Reform LR Rotation



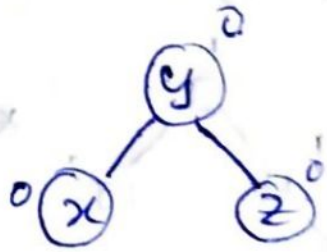
LRotation ⟹



* unbalanced



RRotation ⟹



---

when tree is unbalanced because of

RR ⟹ LRotation

LL ⟹ RRotation

RL ⟹ R_{subtree} $L_{\text{full tree under unbalanced node}}$ Rotation

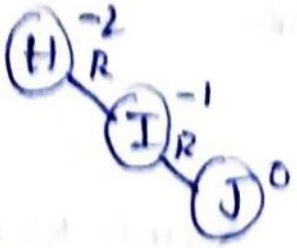
LR ⟹ L_{subtree} $R_{\text{unbalanced nodes tree}}$ Rotation

These are possible 4 cases in AVL tree

AVL tree example

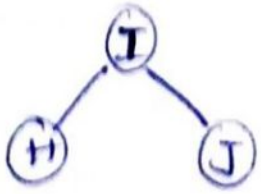
* H I J B A E C F D G K L

Step 1: Insert H I J (at least 3 nodes, req to negate BF)



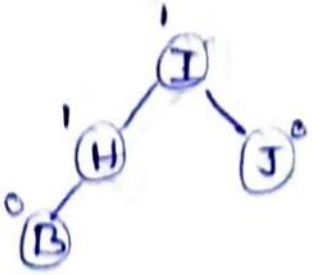
* unbalanced

Step 2: Perform LL Rotation



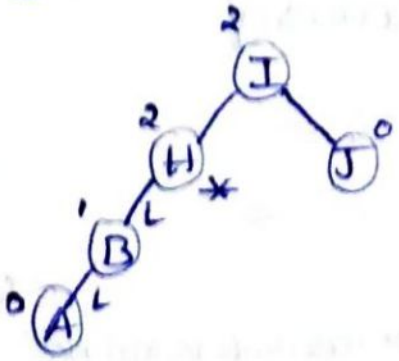
* Balanced

Step 3 Insert B



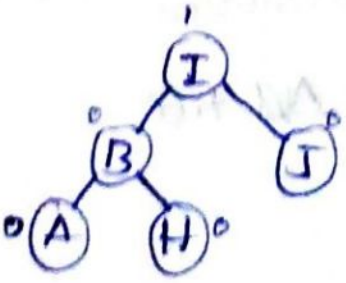
* Balanced

Step 4: Insert A



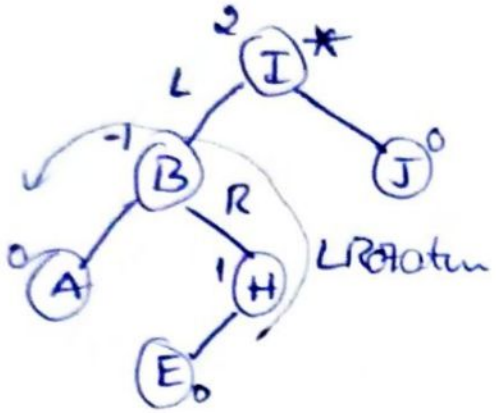
* unbalanced

Step 5: Perform RR Rotation

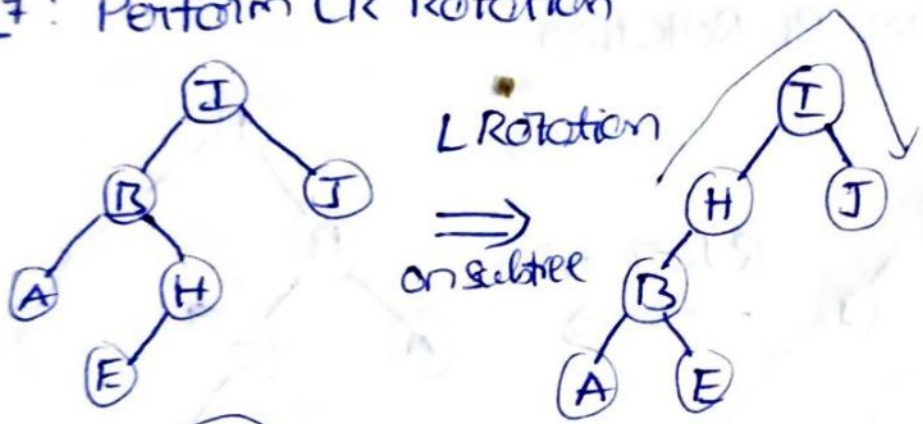


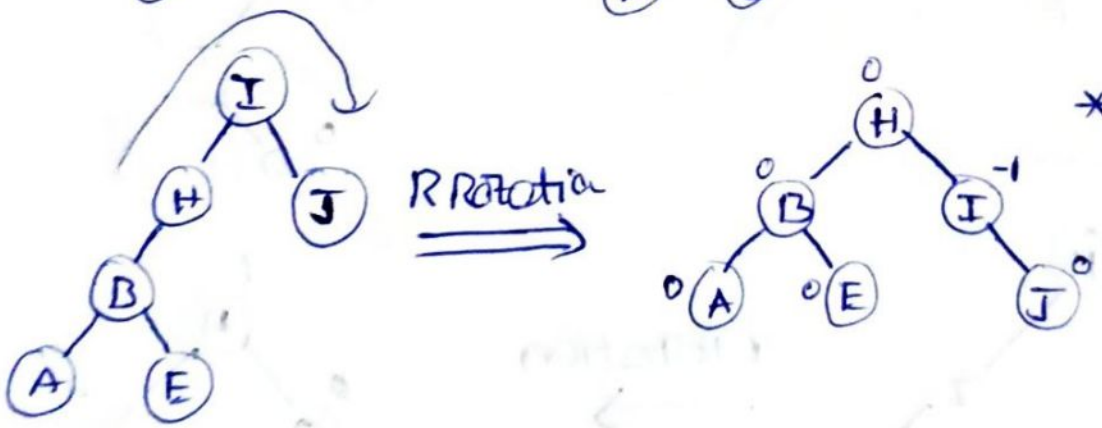
* Balanced

Step 6: Insert E

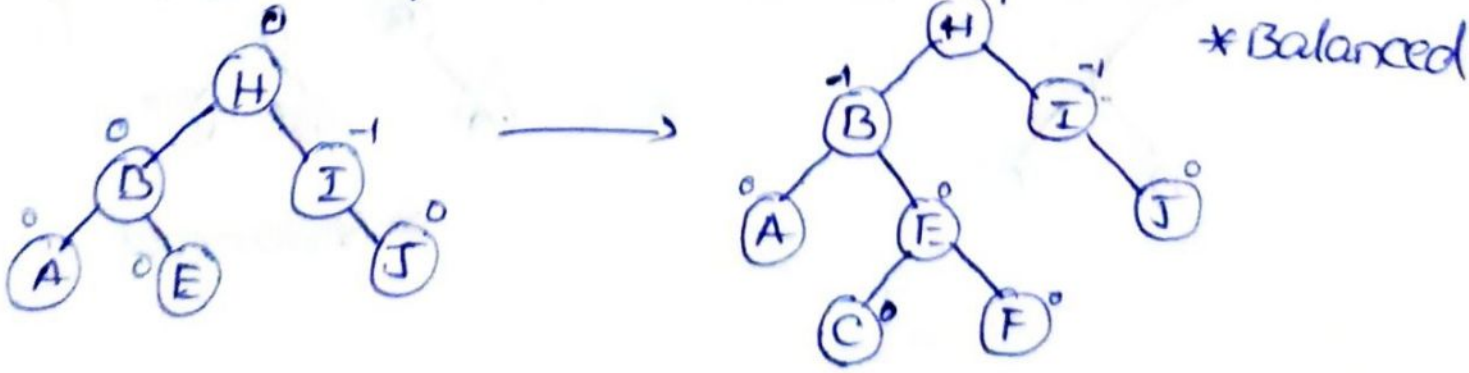


Step_7: Perform LR Rotation

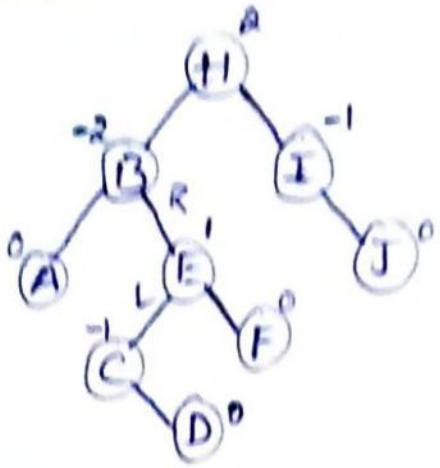




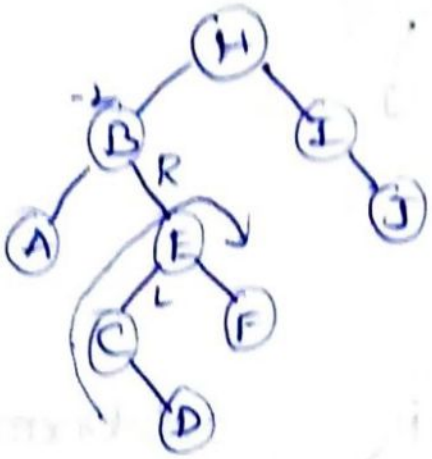
Step 8: Insert C & F



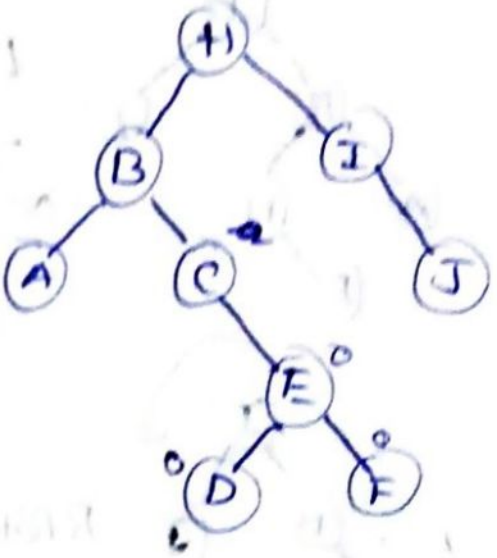
Step 9: Insert D

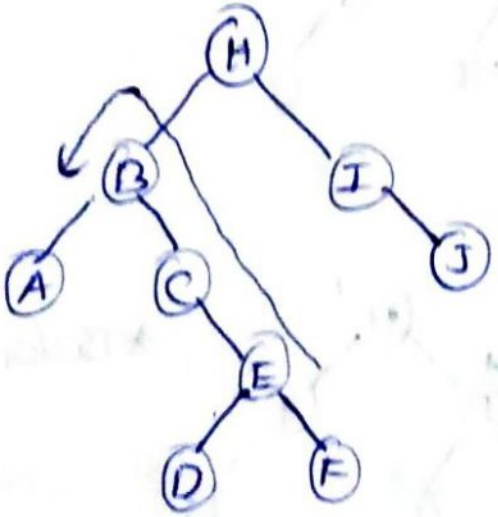


Step 10: Perform RL Rotation

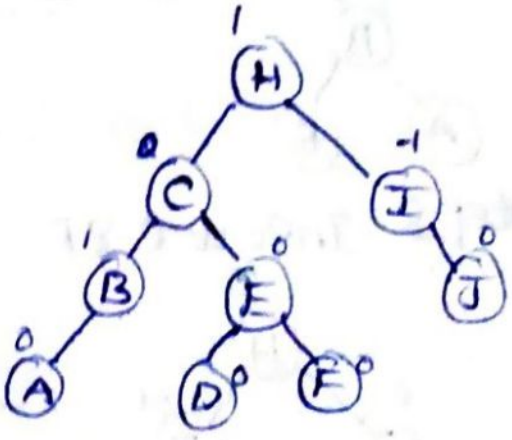


R Rotation ⟹



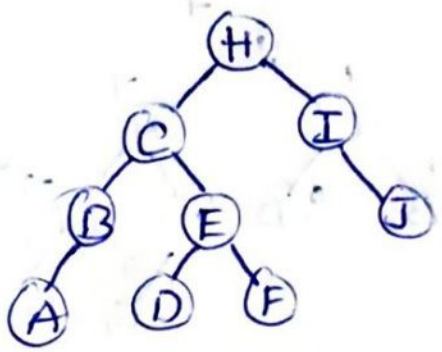


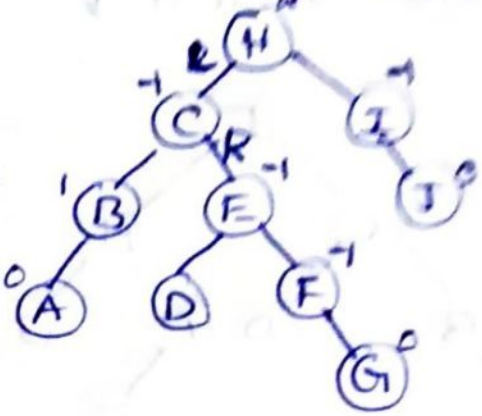
L Rotation ⟹



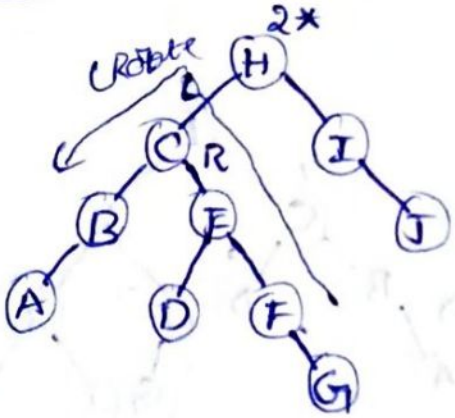
* Balanced

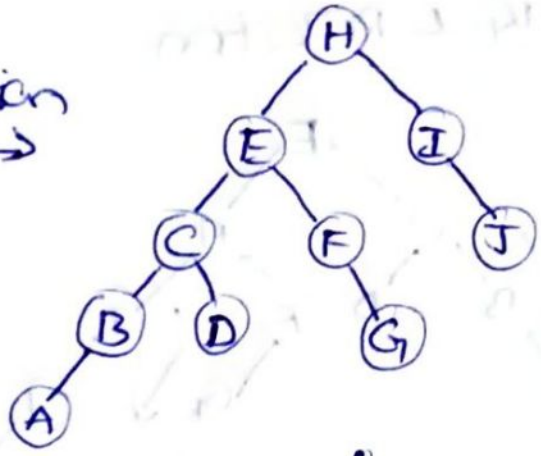
Step 11: Insert G

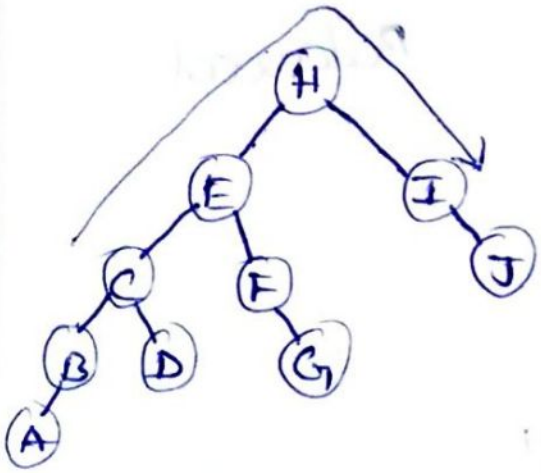


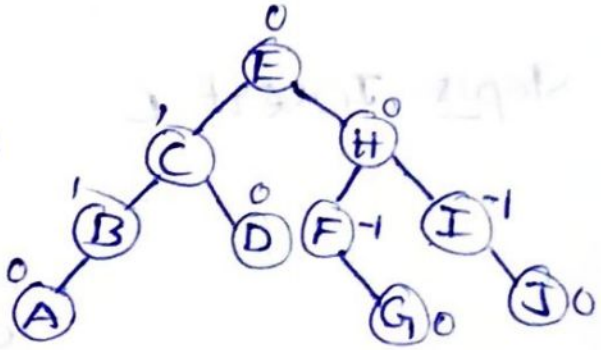


Step 12: RLRRotation



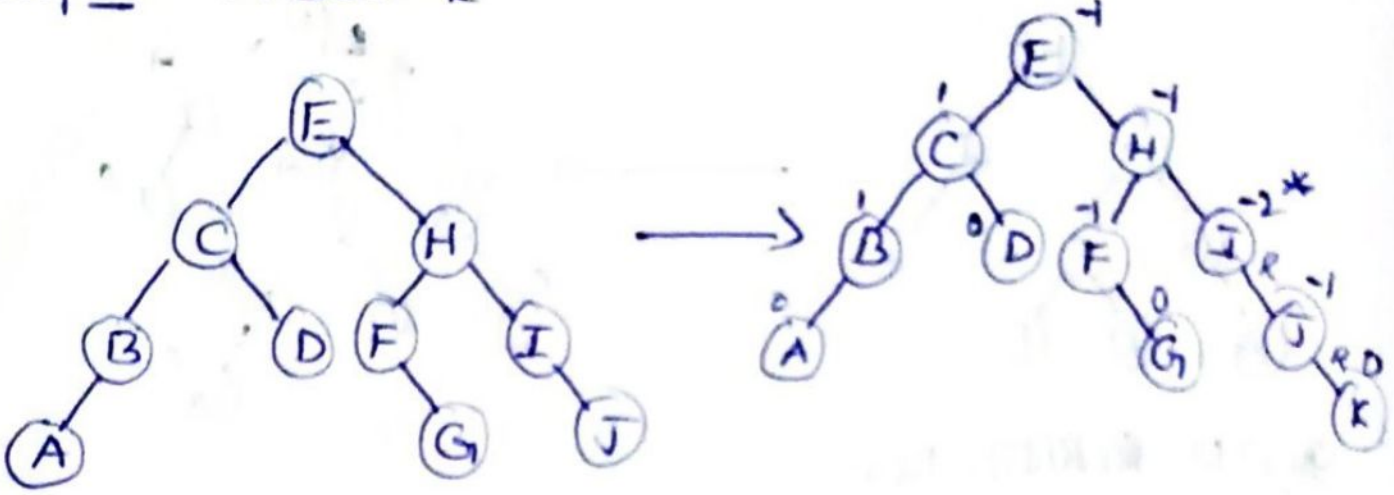




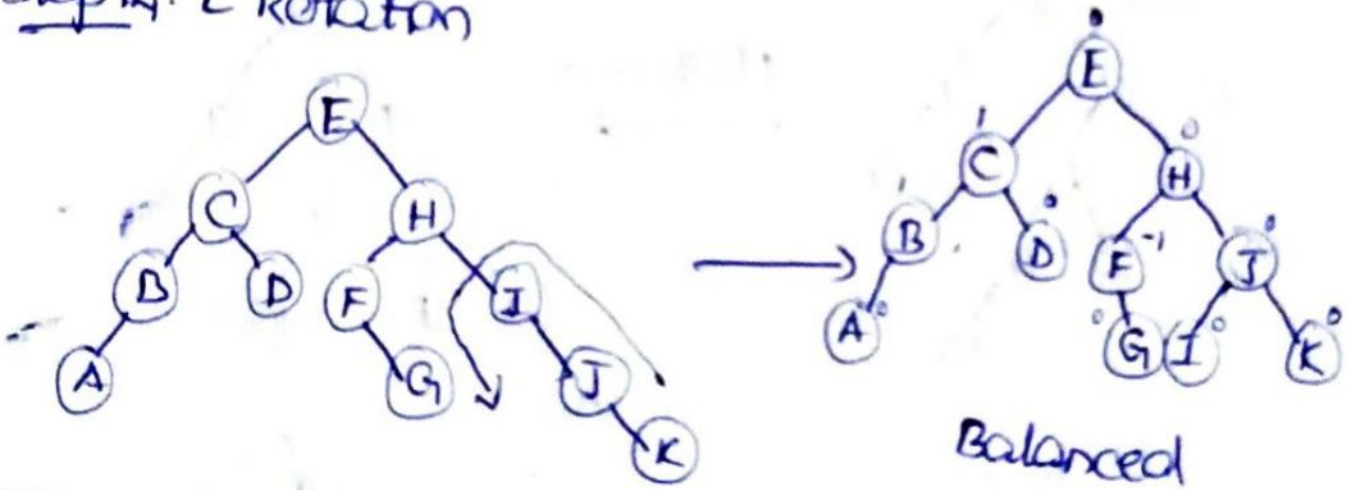


* Balanced

Step 13: Insert K

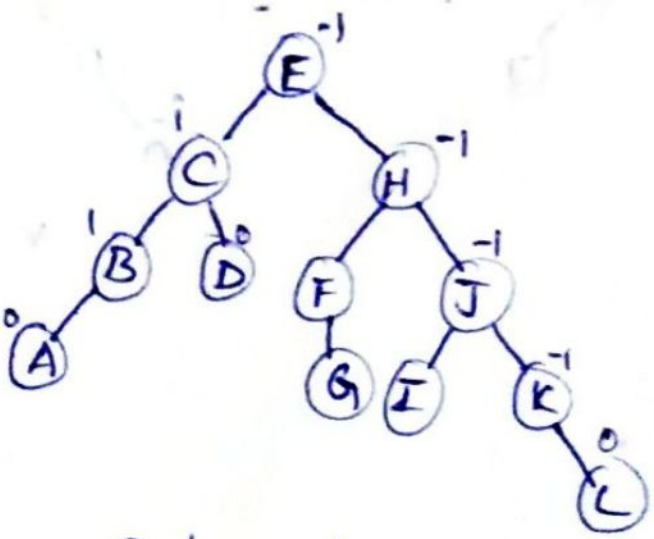


Step 14: L Rotation



Balanced

Step 15: Insert L



Balanced

Final AVL Tree

BTree

* specialized m-way tree
* widely used for disk access
* balanced m-way tree
* maintains sorted data
* All leaf nodes must be at same level
* 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

B Tree of order m has following properties

* Every node has max m children
* Minimum children for leaf → 0

  root → 2

  internal nodes → $\lceil \frac{m}{2} \rceil$
* Every node has max (m-1) keys
* minimum keys : rootnode → 1

  all other nodes → $\lceil \frac{m}{2} \rceil - 1$
* 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 B tree 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 provide fast access to the actual data stored on disks, in general it is time consuming process
* un indexed & 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 comparitively 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

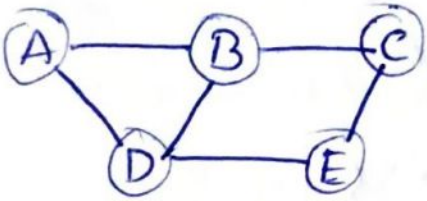
Insertion/Deletion is similar to BST, if any of these operations violate properties of B/B+ tree, we can split/merge child nodes with parent node or among siblings.

Graphs

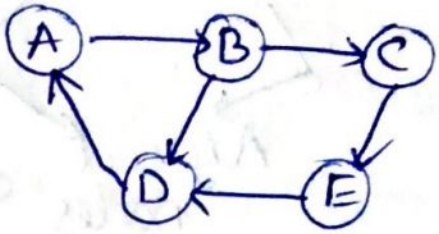
* Defined as group of vertices and edges that are used to connect these vertices.
* also called as cyclic tree
* Graph G can be defined as an ordered set G(V,E). where V(G) represents set of vertices and E(G) represents set of Edges that are used to connect these vertices.

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→B, B→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 is same as terminal node.

$V_0 = V_n$

Simple path (closed)

* 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:

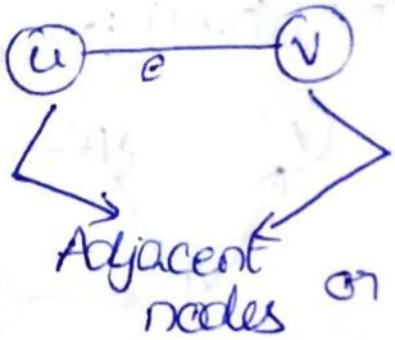
* every node is connected with all other nodes
* contain $\frac{n(n-1)}{2}$ edges
* n is number of nodes in the graph.

Weighted graph

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

Digraph ⇒ Directed graph

Adjacent nodes

* 

Adjacent nodes or neighbours

Degree of the node

* number of edges that are connected with that no
* Degree 0 ⇒ Isolated node

Graph representation

2 ways to store graphs in computer memory

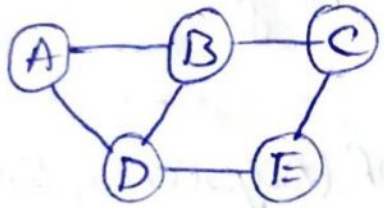
* sequential repr (i.e, Adjancy Matrix)
* Linked list repr (i.e, Adjancy List)

1. Adjancy Matrix
2. Adjancy List

Adjancy Matrix

* used to represent undirected graph, directed graph, weighted directed and undirected graph.
* adj[i][j] = w ⇒ Edge exists from i to j with "w"
* 0 ⇒ no association between the nodes
* 1 ⇒ Existence of path between two nodes

Example

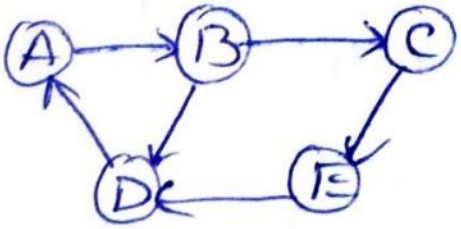


undirected graph

	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

Adjancy Matrix

* If there is any self loop in the graph then there will be '1' on the daigonal of Adjancy Matrix.



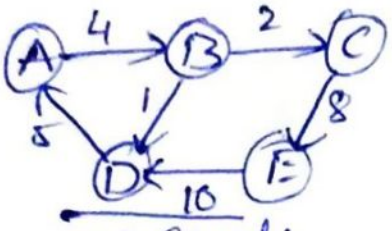
Directed graph

	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 ⇒ All diagonal entries are 0 in Adj. Mat.

Adjoney Matrix for weighted directed graph

* Entries in adjancy matrix are weights between



weighted directed graph

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

Adjacency Matrix

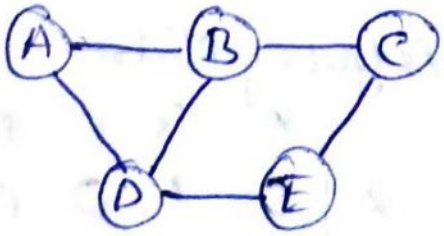
Properties

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

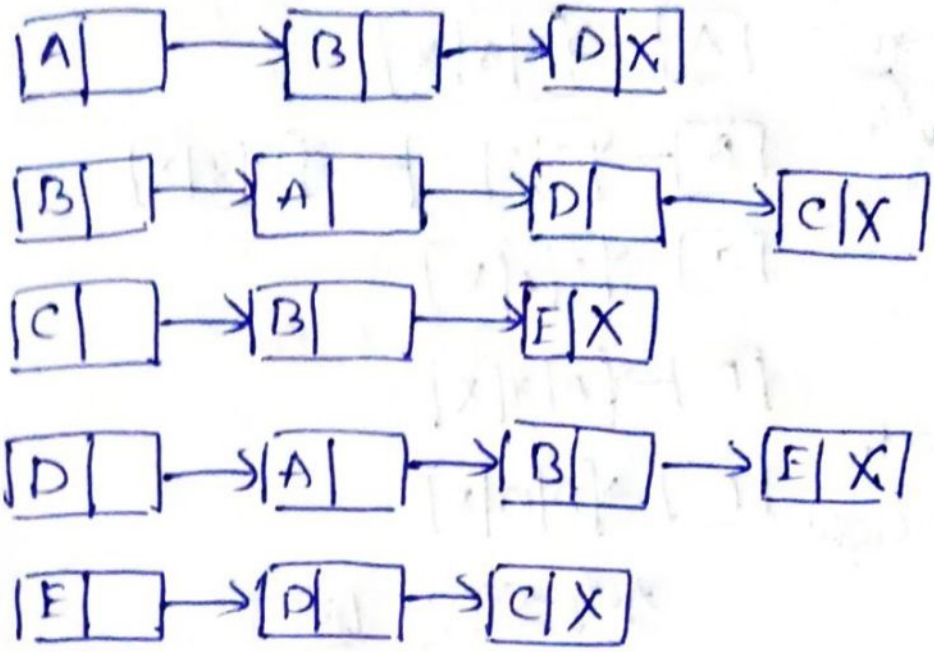
Linked List implementation of graph (Adjacency list)

* Efficient interms of storage
* we only have to store value of edges for a vertex

Example:



undirected graph

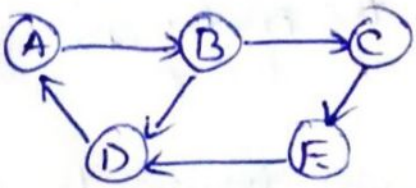


Adjacency list

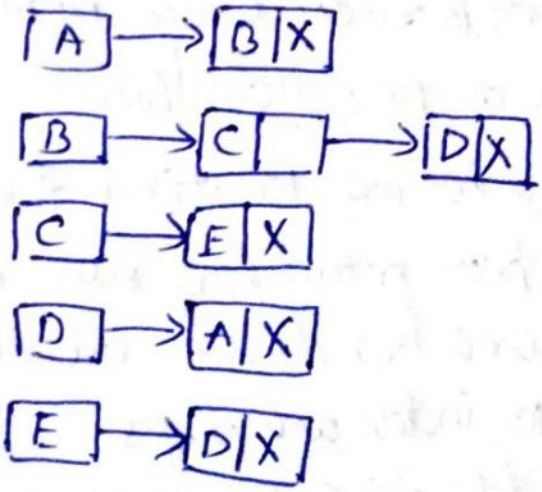
we will have a L.L on each of vertice which is connected or have info of all its adjacent nodes.

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

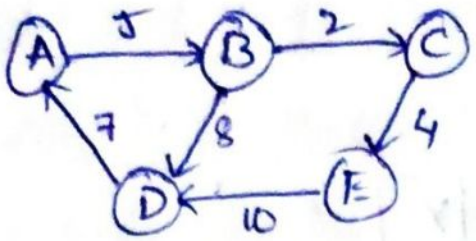
If all adjacent nodes of a node are visited store the next value of a node as NULL



Directed graph

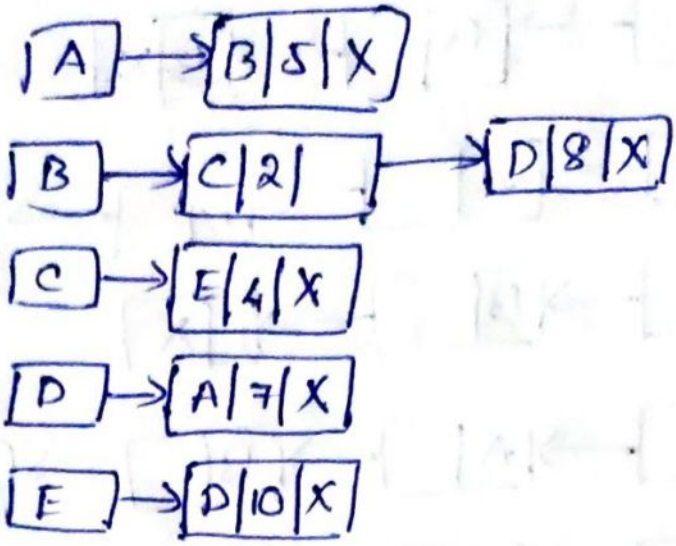


Adjacency list



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 approch 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

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

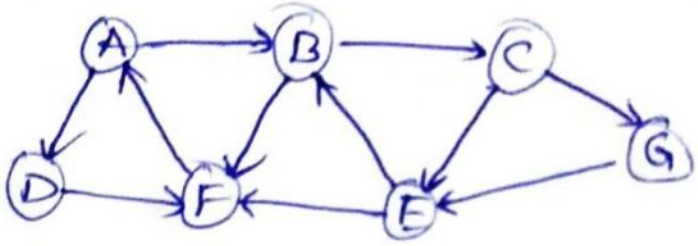
Step4: Dequeue a node N. Process it and set its status = 3 in processed state

Step5: Enqueue all the neighbours of N, that are in the ready state (whose STATUS = 1) and set their status = 2 (waitsta

[END OF LOOP]

Step6: EXIT

Example



Adjacency List

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

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

Queue 1 => all the nodes that are to be processed

Queue 2 => all the nodes that are processed and del from Q1

Steps:

Step1: Add A into Q_1

$Q_1 = \{A\}$

$Q_2 = \{NULL\}$

Step2: A is processed, insert neighbours of A into Q_1 which are unvisited

$Q_1 = \{B, D\}$

$Q_2 = \{A\}$

Step3: B is processed, add neighbours of B into Q_1

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

$Q_2 = \{A, B\}$

Step4: 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.

Step5: C is processed, add neigh of C into Q_1

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

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

Step6: F is processed

$Q_1 = \{E, G\}$

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

Step7: E is processed, and we get the shortest path b/w

$A \rightarrow E$

$Q_1 = \{G\}$

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

Path => $A \rightarrow B \rightarrow D \rightarrow C \rightarrow F \rightarrow E$

Complexity of BFS algorithm

TimeC => $O(V+E)$ => worst case, as it visits every V & E

SpaceC => $O(V)$ V => Vertices / nodes

E => 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.

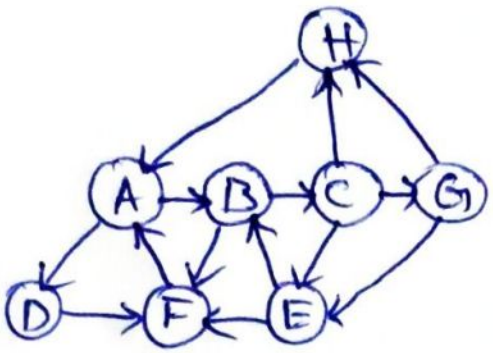
Steps

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

Step1

Stack : H

Step2

print H
Stack A

Step3

print A
Stack B, D

Step4

print D
Stack BF

Step5

print F
Stack B

Step6

print B
Stack C

Step7

print C
Stack EG

Step8

print G
Stack E

Step9

print E
Stack □

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 guarentee 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-1 edges → Condition

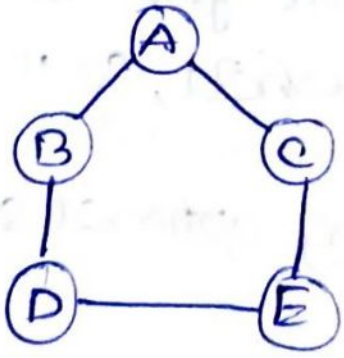
* Maximum number of spanning trees possible for complete undirected graph $\Rightarrow n^{n-2}$

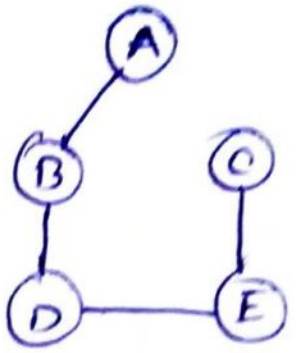
n = number of vertices in graph

Applications of spanning tree

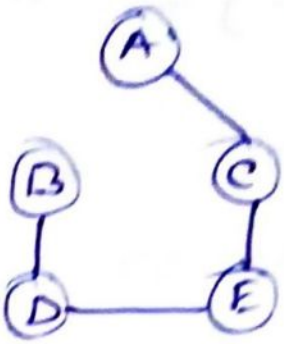
* cluster analysis
* Civil network planning
* Computer network routing protocol

Example:

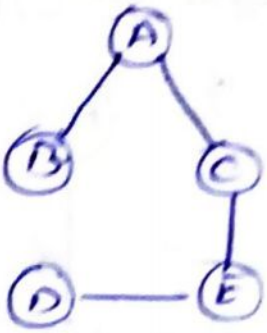




spanning tree 1



spanning tree 2



spanning tree 3

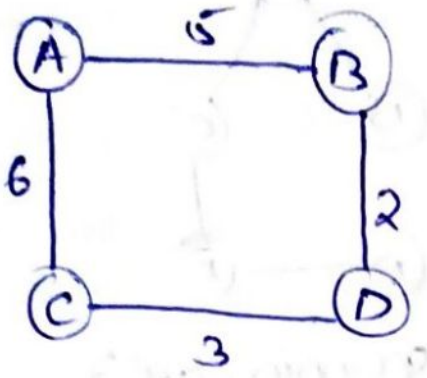
Properties

* 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 => num of edges

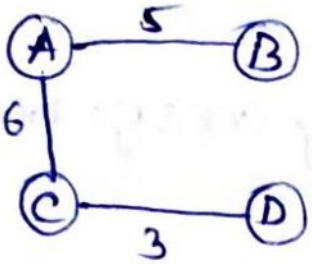
  n => num of vertices
* Can only be created for connected graph
* no spanning tree for disconnected graph.
* MST => path weight should be minimum

Minimum Spanning Tree Example

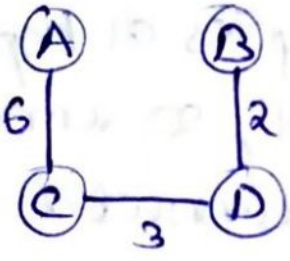


weighted graph

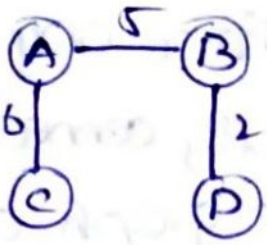
Possible spanning trees are,



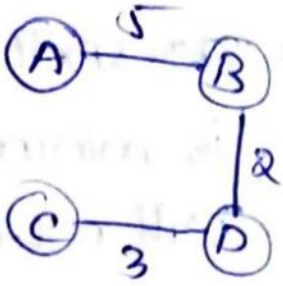
Sum = 14



Sum = 11



Sum = 13



Sum = 10

[MST]

Applications of MST

* Design water supply networks
* Design Telecommunication networks
* Find paths in the map

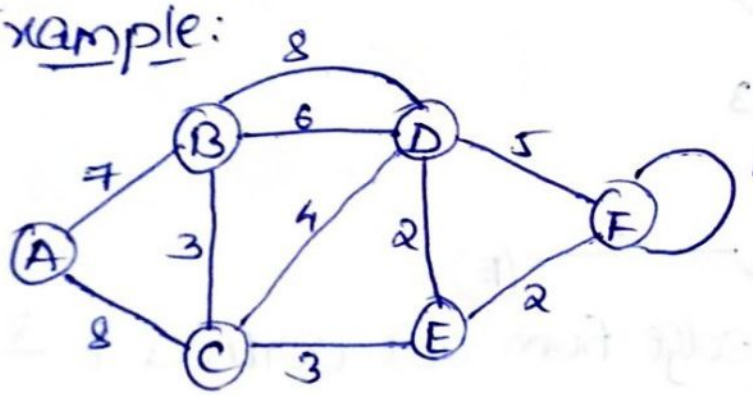
MST can be found by following algo's:

* Prim's Algorithm
* Kruskal's Algorithm

Prim's algorithm:

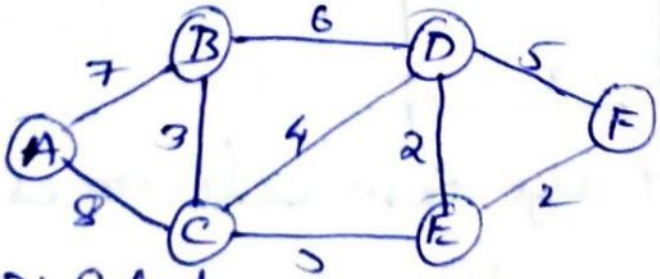
* used to calculate min spanning tree
* select a root vertex, and then select a shortest edge connected to that vertex, and then we need to do the same process of select a shortest edge of previously selected vertex's
* Generate the minimum cost spanning tree starting from a root node.
* prerequisites,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

Example:



Find MST:

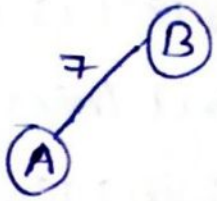
Step 1: remove self loops and parallel edges



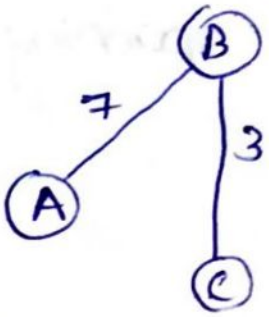
Step 2: Select any vertex ⇒ A

Step 3: Find an available and minimum output edge from the available vertices of the graph

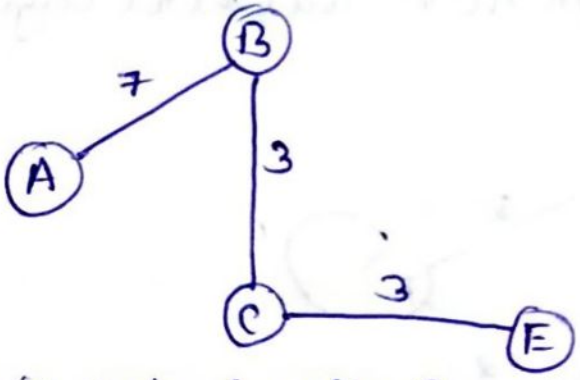
↳ A $\xrightarrow{7}$ B



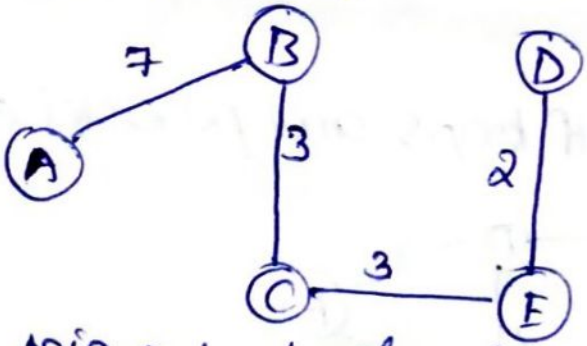
Step 4: Find min output edge from avl vertices B $\xrightarrow{3}$ C



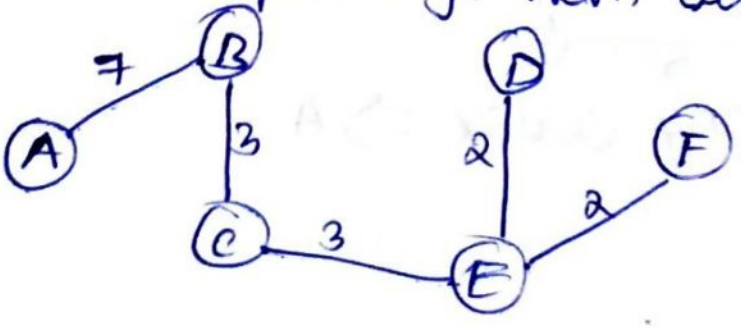
Step 5: Find min output edge from avl vertices C $\xrightarrow{3}$ E



Step 6: Find min output edge from avl vertices E $\xrightarrow{2}$ D



Step 7: Find min output edge from all vertices E $\xrightarrow{2}$ 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

$V' = V$ → number of vertices of MST & complete will be same

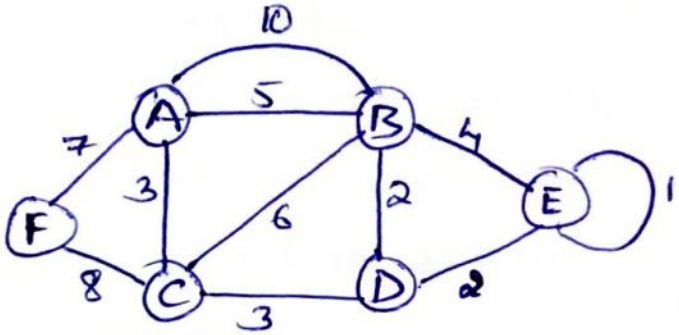
$E' \subset E$ → MST is subset of complete graph.

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

V → num of vertices of MST

E' → num of edges of MST

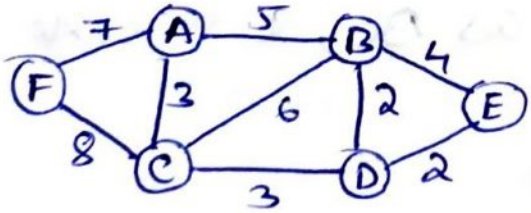
Kruskal's algorithm



Step 1:

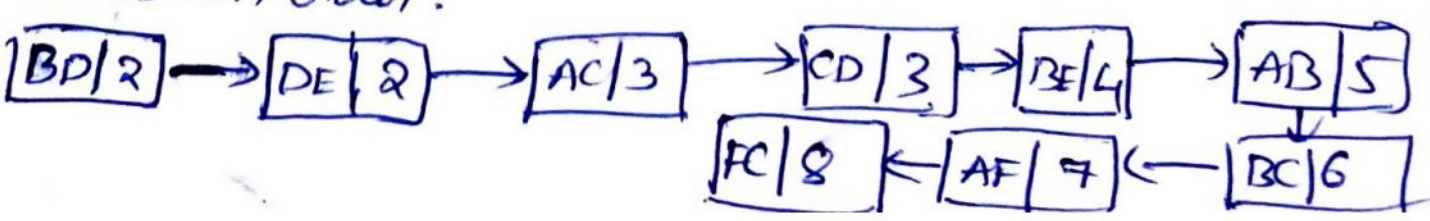
① Remove self loops

② Remove parallel edges



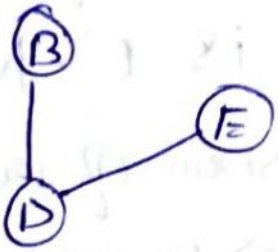
Step 2: we need to select minimum edges in a sequence and build a path such that there shouldn't be any cycle

Selection order:

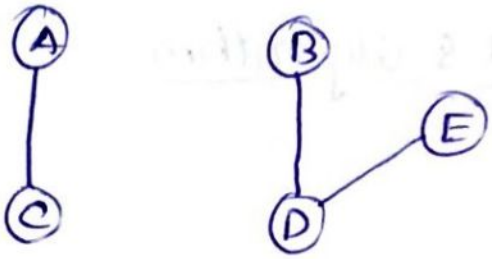


Step 3 : select BD

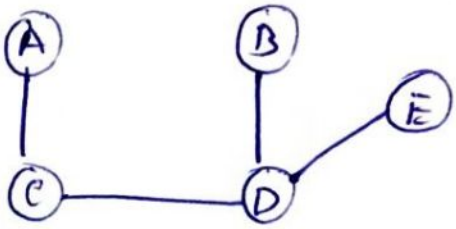
Step 4: Select DE



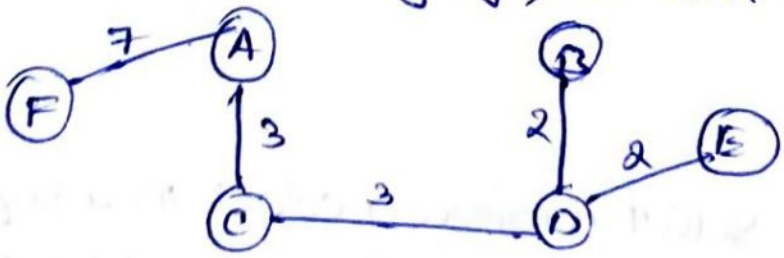
Step 5: Select AC



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 Kruskal's 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 ⇒ O(n)
* worst case T.C ⇒ O(n) * Best case ⇒ 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 = 41

K = 41

Iteration starts from K → 70 to K → 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 → Best Case O(1); → Average / Worst Case O(log n) |

* 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 ⇒ | end = mid - 1 |
* If the value that we want is greater than that of mid value then make ⇒ | start = mid + 1 |

Important points

* BS can implement through Iterative & Recursive approach

Example

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

step1: | 10 | 12 | 24 | 29 | 39 | 40 | 51 | 56 | 59 | K ≠ mid ⇒ K > mid
↑mid ⇒ move right ⇒ start = mid + 1

step2: | 10 | 12 | 24 | 29 | 39 | 40 | 51 | 56 | 59 | K ≠ mid ⇒ K > mid
↑mid ⇒ move right ⇒ start = mid + 1

step3: | 10 | 12 | 24 | 29 | 39 | 40 | 51 | 56 | 59 | K = mid, return True as element found
↑mid

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

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
```

Example:

arr = | 13 | 32 | 26 | 35 | 10 |
0 1 2 3 4

1st iteration

| 13 | 32 | 26 | 35 | 10 | comp 0 & 1 |
0 1 2 3 4 | 0 < 1, skip

| 13 | 32 | 26 | 35 | 10 | comp 1 & 2 |
0 1 2 3 4 | 1 > 2, swap

| 13 | 26 | 32 | 35 | 10 | comp 2 & 3 |
0 1 2 3 4 | 2 < 3, skip

| 13 | 26 | 32 | 35 | 10 | comp 3 & 4 |
0 1 2 3 4 | 3 > 4, swap

| 13 | 26 | 32 | 10 | 35 | nth element sorted (*)

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

Similarly

2nd iteration | 13 | 26 | 10 | 32 | 35 | n-1th ele sorted

3rd iteration | 13 | 10 | 26 | 32 | 35 | n-2th ele sorted

4th iteration | 10 | 13 | 26 | 32 | 35 | 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 any where lesser greater than elements of sorted array inclued the card in sorted p

---

* worst case time complexity $\Rightarrow O(n^2)$
* lesser efficient than other sorting algo's like
  - → heapsort → merge sort
  - → quick sort

Advantages

* Simple implementation
* Efficient for small datasets
* Appropirate for partially sorted datasets

Algorithm

Step1: If element is first element, assume it as sorted and ret
Step2: pick the next element, and store seperately in a key.
Step3: Now, compare the key with all elements in sorted array.
Step4: If element in sorted array is smaller than curre 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 ⇒ no swap (sorted already) ⇒ Store 12 in sorted array

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

compare 1 & 2 ⇒ 31 ≠ 25 ⇒ swap ⇒ compare & store 25 in sorted array

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

Compare 2 & 3 ⇒ 31 ≠ 8 ⇒ swap ⇒ compare & store 8 in sorted array

A = {[12, 25, 8], 31, 32, 17} 25 ≠ 8 ⇒ swap

A = {[12, 8, 25], 31, 32, 17} 12 ≠ 8 ⇒ swap

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

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

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

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

compare 4&5 ⟹ 32 ≠ 17 ⟹ swap ⟹ compare & store 17 in sorted array

A = {[8, 12, 25, 31, 17], 32} 31 ≠ 17 swap

A = {[8, 12, 25, 17, 31], 32} 25 ≠ 17 swap

A = {[8, 12, 17, 25, 31], 32} 8, 12 < 17 ⟹ no swap

compare 5 with sorted array:

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

or

[8, 12, 17, 25, 31] < 32 (Already sorted)

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

Time complexity :

Best case ⟹ no sorting req ⟹ O(n)

Average case ⟹ partialy sorted ⟹ O(n²)

worst case ⟹ complete unsorted ⟹ O(n²)

Space Complexity :

only variable required for swapping ⟹ O(1)

Insertion sort uses/applications

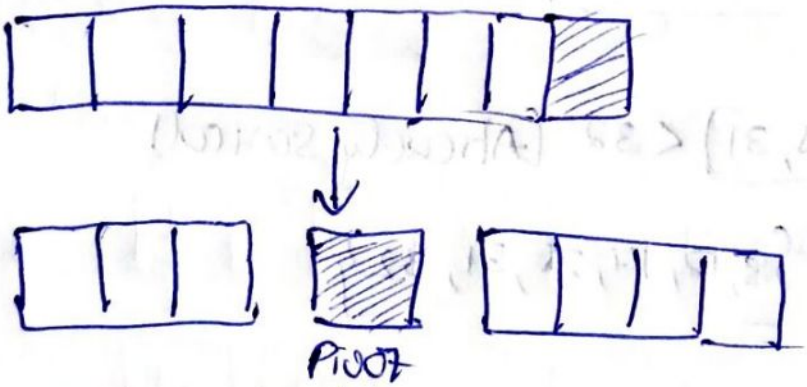
used when

* array has small number of elements
* partialy 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 seperation 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 does the above operation recursively such that subarray's are almost divided into single valued array (len(arr) = 1)

Example:



Quick sort

* Faster and highly effecient sorting.
* n log n comparisions 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 ⇒ pivot is middle or near to middle element ⇒ $O(n \log n)$

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

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

worst TC of QS 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.

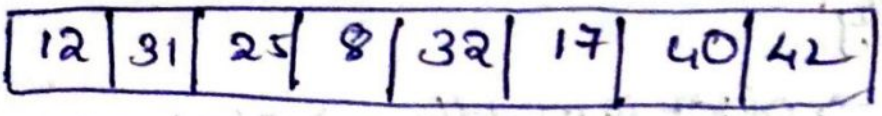
Space complexity: $O(\log n)$

Merge Sort algorithm

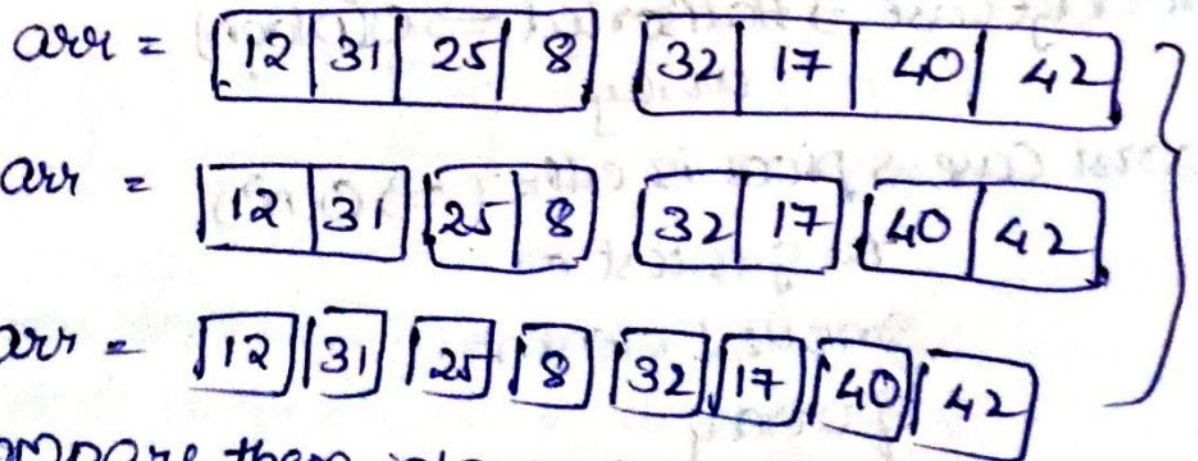
* Follows divide and conquer approach.
* Efficient sorting algorithm.
* Divides the given list into two equal halves, calls itself for the two halves and then merges the sorted halves.
* It will be divided into halves until the list cannot be divided further.

Example:

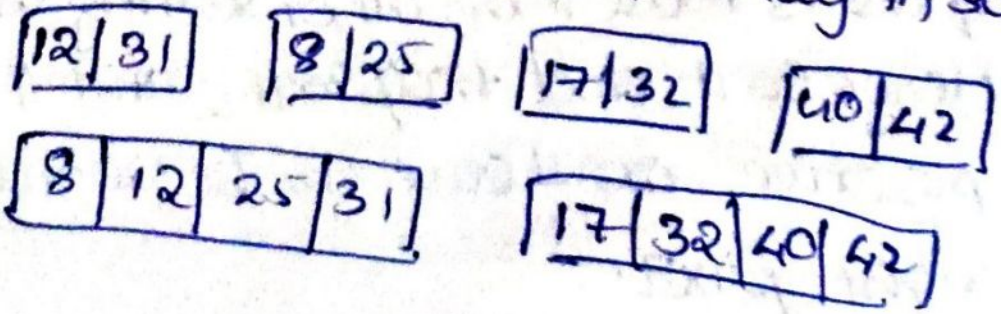
① Lets take an unsorted array



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



③ Compare them into another array in sorted order



Final Merging

8	12	17	25	31	32	40	42

Merge Sort Time Complexity

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

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

worst case ⇒ Totally unsorted ⇒ $O(n \log n)$

Space Complexity : $O(n)$

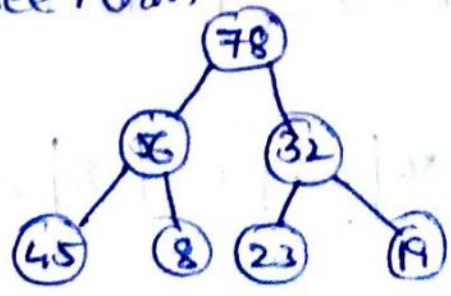
HeapSort algorithm

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

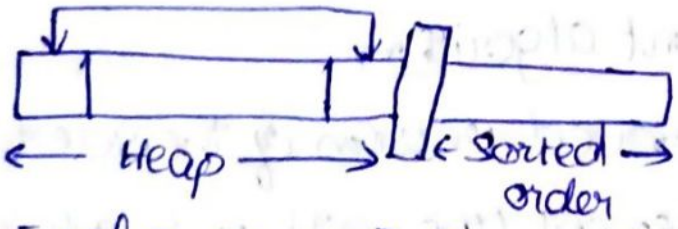
78	56	32	45	8	23	19

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

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

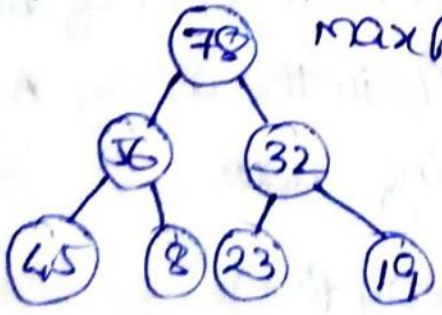
To sort in ascending order we use max heap property on the tree.

Heapsort exchange process

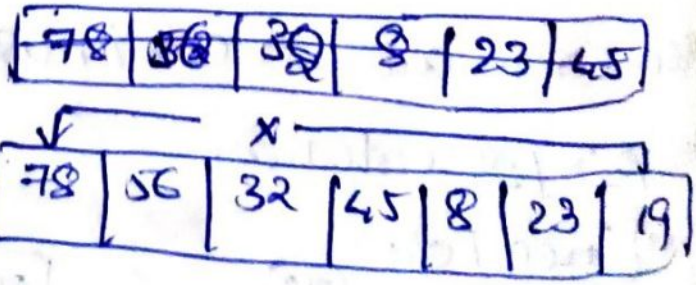


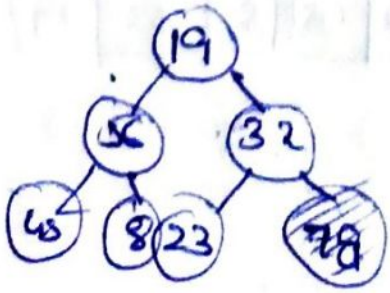
Exchange root of max heap, which is also the highest element of the unsorted array / tree with the last element of the unsorted array and merge that node into the sorted array.

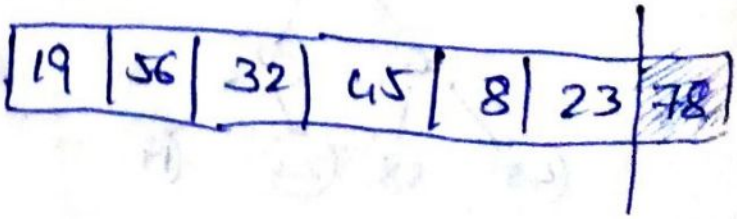
Example:



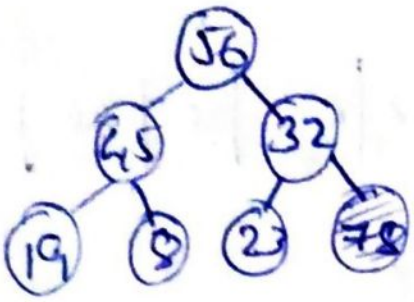
max heap

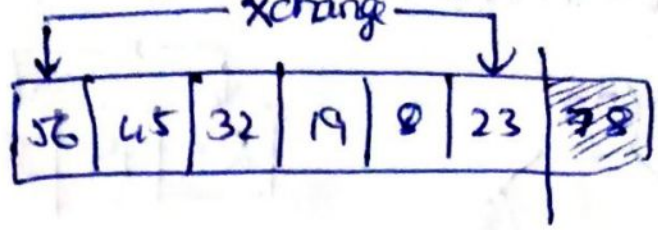


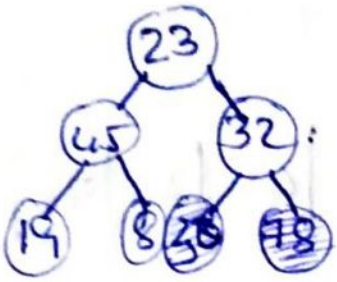


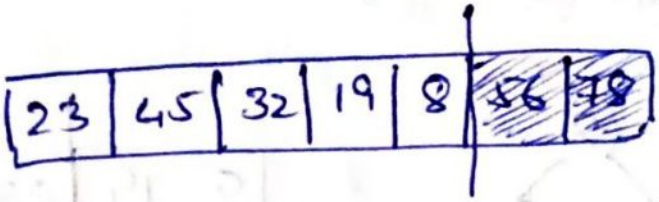


violates max heap property, so heapify

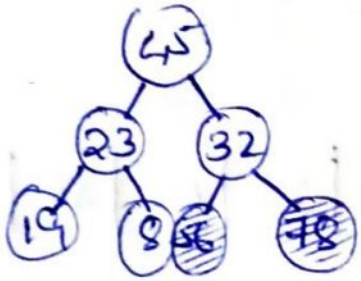


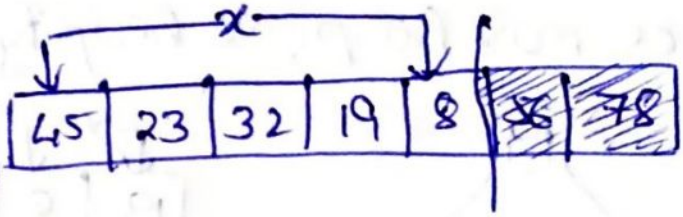


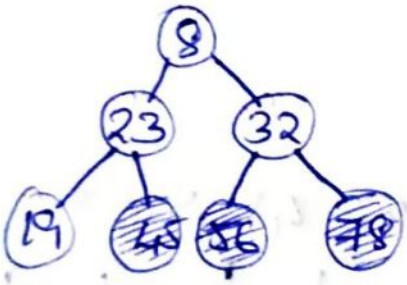




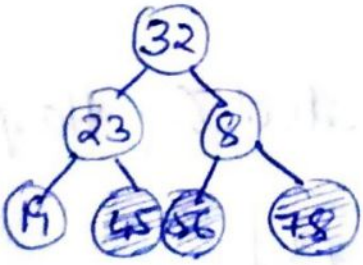
violates max heap property, so heapify

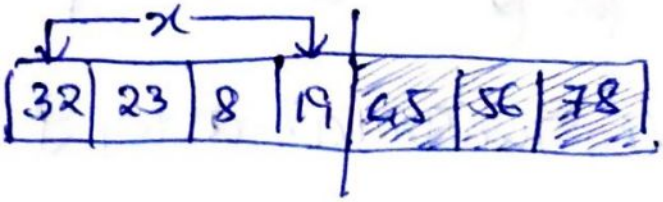


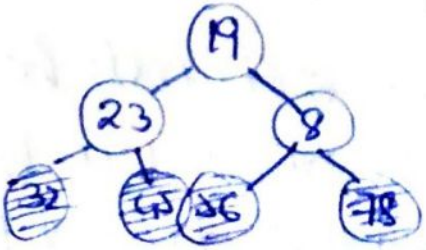


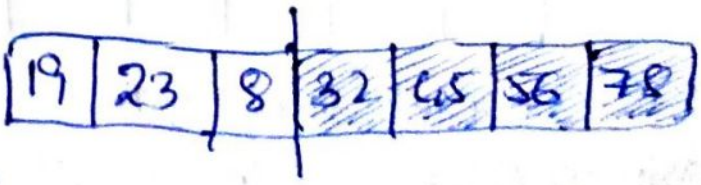


violates max heap property, so heapify

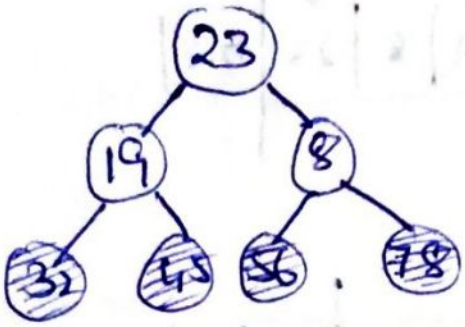


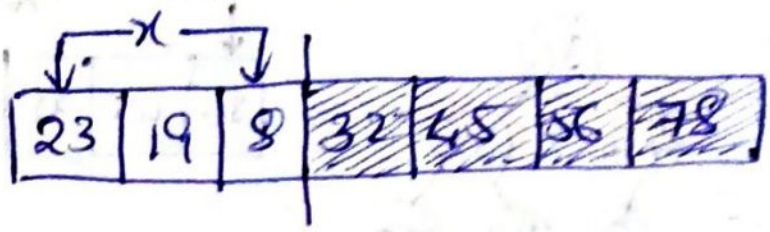


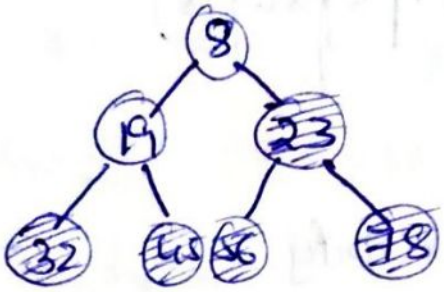


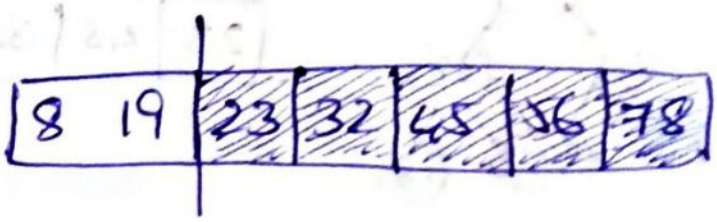


violates max heap, so heapify

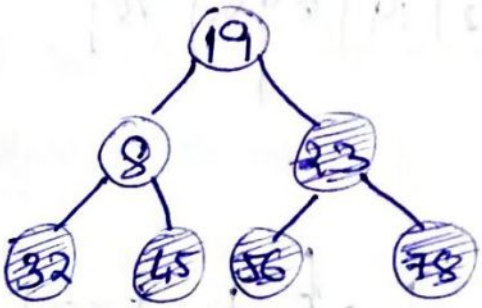


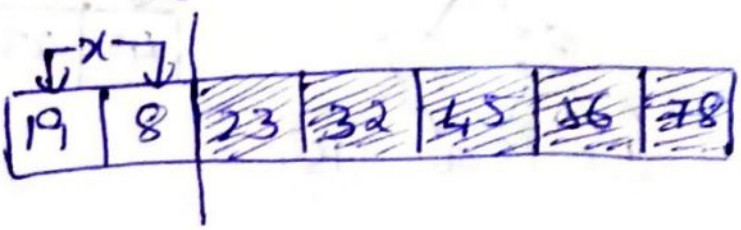


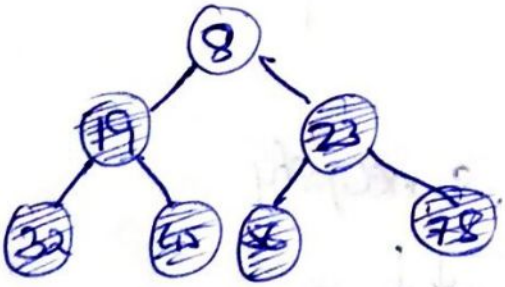


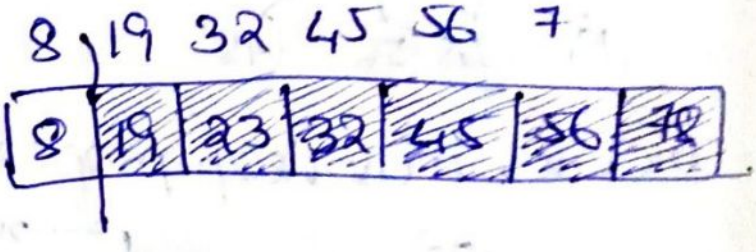


violates max heap, so heapify









Finally the sorted array generated by HeapSort

8	19	23	32	45	56	78

Heap is complete binary tree, where each node has utmost 2 children. In CBT 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 in the 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

$\left\lfloor \frac{i-1}{2} \right\rfloor \Rightarrow$ parent of i

Working of HeapSort

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 heapifing]

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 tree from the root to the leaf node by making a multiple of $\log(n)$ comparisions and swaps.

working of HeapSort

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 heapifing]

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

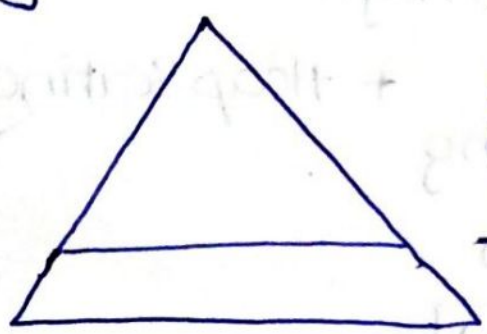
To fully heapify an element whose subtrees are already max heaps, we need to compare the element with its left and right children pushing it downwards until it reaches a point where both its children are smaller than it. In worst case, we need to move an element down the line from root to the leaf node by making a multiple of $\log(n)$ comparisions 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 follows max heap, we do heapification on $n/2$ elements.

So, we need

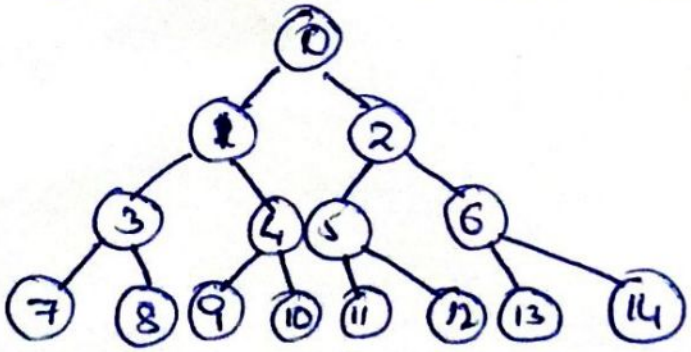
$\log(n)$ comparisions for $n/2$ elements to heapify the array to max heap $\Rightarrow \frac{n}{2} \times \log(n) \sim n\log n$



root & intermediary nodes

$\Rightarrow$ leaf nodes

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



leaf nodes $\Rightarrow 7$

Tot elements $\Rightarrow 14$

$n = 14$

we need to do heapification on nodes in this order of nodes. 6 → 5 → 4 → 3 → 2 → 1 → 0, it means,

from $\frac{n}{2}^{th}$ node (index $= \frac{n}{2} - 1$) to the root node (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~~ 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 $n \cdot \log(n)$

So, $TC = n\log n + n\log n$

Initial heapifying + Heap Sorting

$= 2n\log n$

$TC = O(n\log n)$

Space Complexity $O(1)$

Counting Sort

* non comparision based algorithm.
* Rest algorithms such as quicksort, HeapSort rest of those are comparision based
* counting sort, RadixSort, bucket sort are of non comparision 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 mean, we are defining the range of values that must be present inside the array.

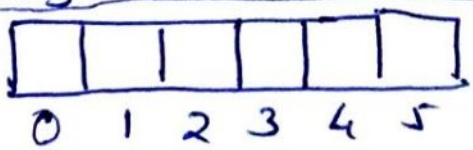
For example, if Range is 5, all the values that are present in the array are between 1 to 5.

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

Example:

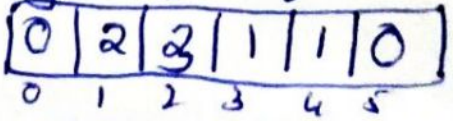
Range = 5

counting sort is basically a freq counter of elements of array using an extra array.

extra array = 

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

By applying counting sort, freq array

extra array = 

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

Time Complexity

$O(n+k)$ ⇒ 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

$O(k)$ ⇒ 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 ⇒ $O(n+k)$ ⇒ already sorted

Average T.C ⇒ $O(n+k)$ ⇒ not asc not desc

Worst Case T.C ⇒ $O(n+k)$ ⇒ 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 0s
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 ⇒ max = 9

Count array of size max+1 ⇒ 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

Cummulative count

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

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

Example:

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

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

CC 1

For Element 2

Important points

Value of array are elements

Value of ccarray is index of element in the output

Original array

2	9	7	4	1	8	4

CCarray

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

2-1=1

output (Sorted)

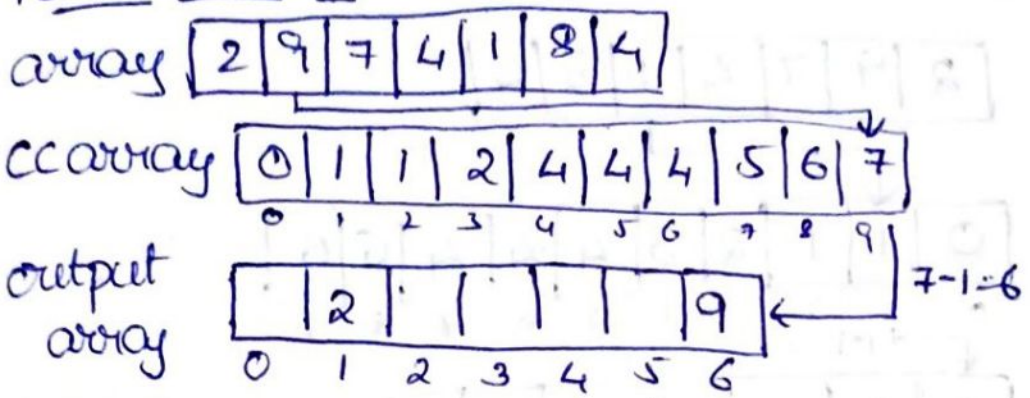
	0	1	2	3	4	5	6
		2					

* 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.

CCarray becomes

0	1	1	2	4	4	4	5	6	7

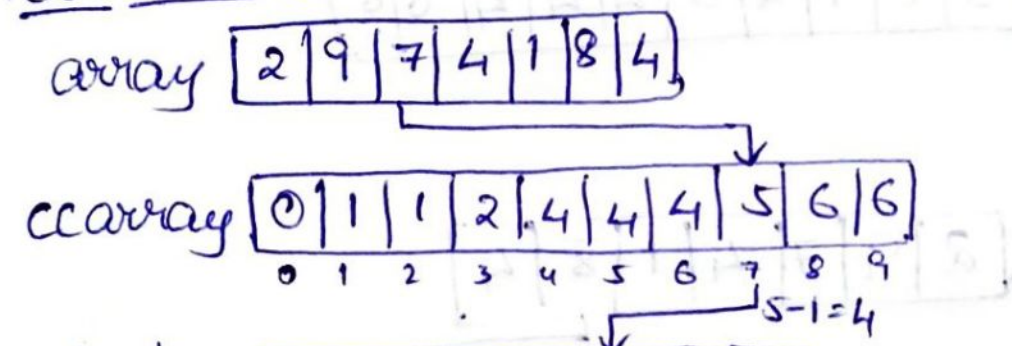
For element 9



ccarray becomes

0	1	1	2	4	4	4	5	6	6

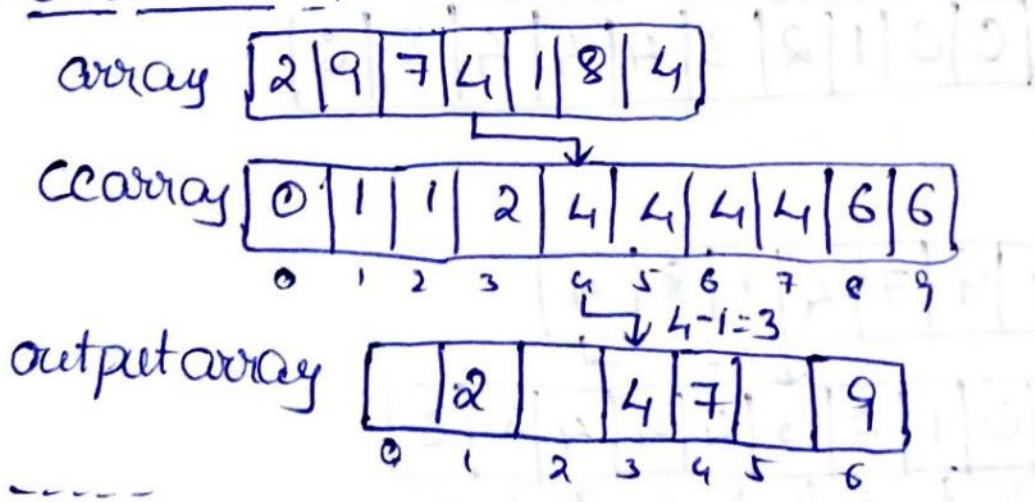
For element 7



ccarray becomes

0	1	1	2	4	4	4	4	6	6

For element 4

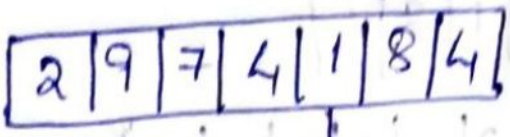


ccarray becomes

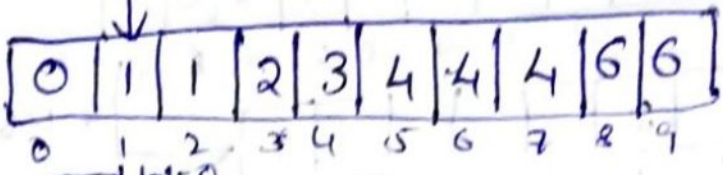
0	1	1	2	3	4	4	4	6	6

For element 1

original array

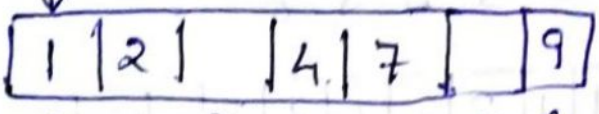


cc array



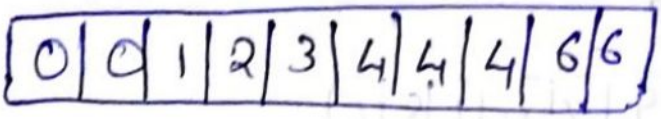
1-1=0

output



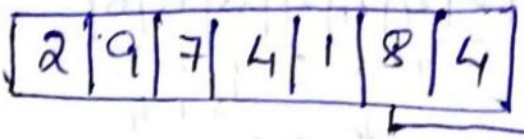
---

cc array becomes

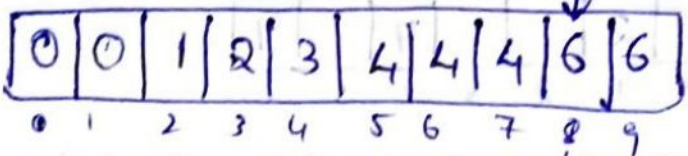


For element 8

original array

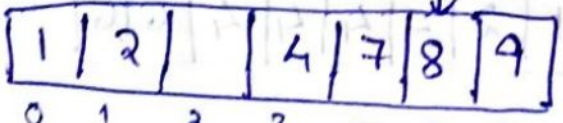


cc array



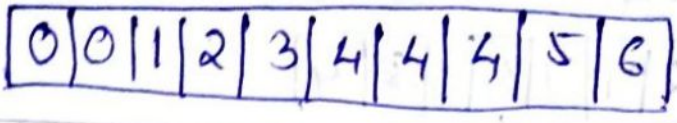
6-1=5

output



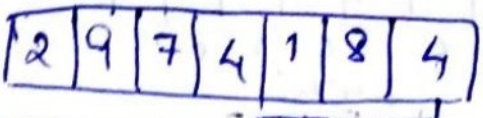
---

cc array becomes

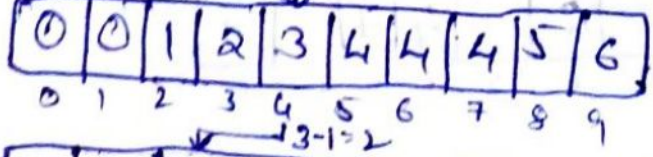


For element 4

orig array

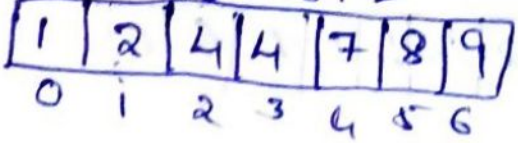


cc array



output

3-1=2

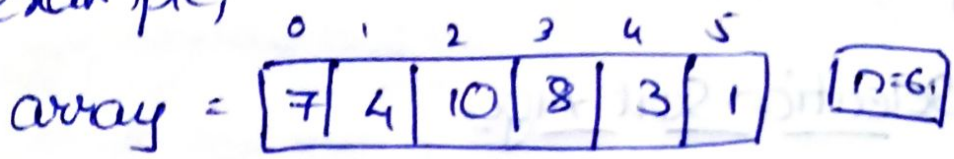


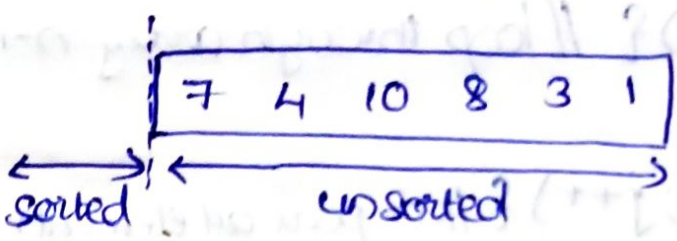
⇒ sorted by count sort

Selection Sort

We will have two assumed array's out of original array. They are sorted array and unsorted array.

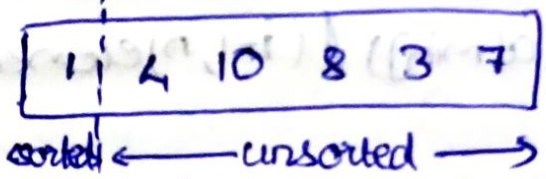
For example,



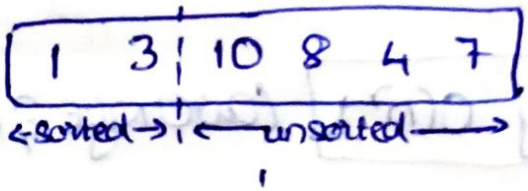


Main theme of algorithm is swap the first element of unsorted array with the minimum element of unsorted array, after sorting make the swapped first element of unsorted array an element of sorted array

Pass 1

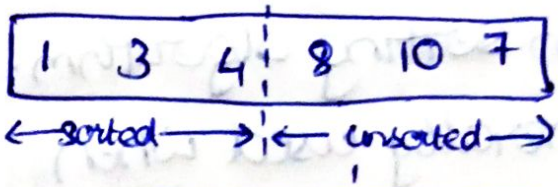


Pass 2



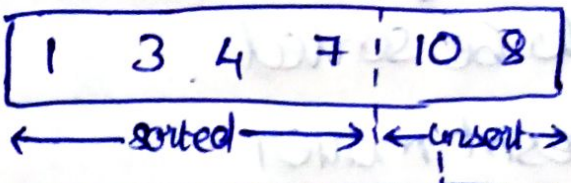
min is 3, so swap 3 with 4

Pass 3



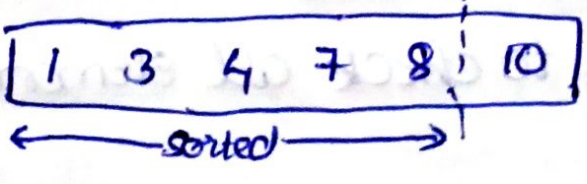
min is 4, so swap 10 with 4

Pass 4



min is 7, so swap 8 with 7

Pass 5



min is 8, so swap 10 with 8

Array is Sorted, so maximum passes we need to make an array sorted is "n-1"

we require two loops to implement this algorithm

For loop ① ⇒ iterate through n-1 passes

For loop ② ⇒ Find minimum element of each pass (inner loop)

Selection Sort Algo

```
for(i=0; i<n-1; i++){ // loop through array n-1 passes
    int min=i;
    for(j=i+1; j<n; j++){ //compare all elements of array to find min element among i+1th element to nth element
        if(a[j] <a[min])
        { min=j
        }
    }
    if (min != i){ // If we found some other min element among (i+1, n) elements
        swap(a[i], a[min])
    }
}
```

Time complexity $O(n^2)$ (average & worst) Best case

* Inplace comparision sorting algorithm.

Selection sort is generally used when

① small array is to be sorted

② swapping cost doesn't matter

③ It is compulsory to check all elements.

Best case → no sorting required → $O(n)$
as already sorted

Average Case → array is not properly → $O(n^2)$,
ascending & descending

worst case → array elements in reverse → $O(n^2)$
sorted order

Space complexity $O(1)$

Radix Sort

* not a comparision based sorting algorithm

For example,

assume this is the array to be sorted

~~[015, 1, 321, 010~~

15, 1, 321, 10, 802, 2, 123, 90, 109, 11

To sort an array in using Bucket sort, we must append '0's at most significant digit of a number. we do this process such that all elements of array are of same length.

For example, find the highest valued element of the array. i.e 802

802 ⟹ 3 digits

pad all elements with 0's, such that every element of array should be of length of the max element of array. Here that is 3. Then modified array would be.

015, 001, 321, 010, 802, 002, 123, 090, 109, 011

To sort this array with Radix Sort, we required 10 buckets (0 to 9) and n passes, where n is the length of the largest element of array.

For sorting string array we would require 26 buckets and n passes. n is the length of largest string of array

For integers, we compare them from LSB to MSB
For strings, we compare them from MSB to LSB

For array,

[015, 001, 321, 010, 802, 002, 123, 090, 109, 011]

Pass 1: [Fill using one's place]

0	010, 090
1	001, 321, 011
2	802, 002
3	123
4	
5	015
6	
7	
8	
9	109

array after 1st pass

[010, 090, 001, 321, 011, 802, 002, 123, 015, 109

Pass 2: [Fill using ten's place]

0	001, 802, 002, 109
1	010, 011, 015
2	321, 123
3	
4	
5	
6	
7	
8	
9	090

array after 2nd pass
[001, 802, 002, 109, 010, 011, 015, 321, 123, 090]

Time Complexity

$O(d \times (n+b))$

d => number of digits in max valued number in array

n => no. of numbers in array

b => base bucket, for num = 10, alpha = 26

Pass 3 [Fill using 100ths place]

0	001, 002, 010, 011, 015, 090
1	109, 123
2	
3	321
4	
5	
6	
7	
8	802
9	

After 3 passes,

The sorted array is (sorted by Radix Sort)

[1, 2, 10, 11, 15, 90, 109, 123, 321, 802]

* Digit by digit sorting algorithm
* Here in Radix sort, we internally uses counting sort algorithm to sort the elements at/by the significance index.
* First we find max element inside the array, and then we use counting sort to sort the elements based on one's, ten's, hundred's index.

Time Complexity of Radix Sort

Best Case : no sorting required $\Omega(n+k)$

Average Case : not proper ascending or descending $\Theta(nk)$

worst Case : sorted in reverse order $O(nk)$

It is non comparitive sorting algorithm of $O(nk)$ linear time complexity which is better than comparitive based sorting algorithm of $O(n \log n)$

Space Complexity : $O(n+k)$

Bucket Sort

* non comparision based algorithm
* Radix & counting sort are also non comparision based
* For best use, the inputs of array should be uniformly and independently distributed across [0,1] to get a running time of O(n)

Bucket Sort algorithm

In this sorting algorithm buckets are created to put elements into them. Then we apply some sorting algorithm (insertion sort) to sort the elements in each bucket. Finally take out and join them to get sorted array.

Example:

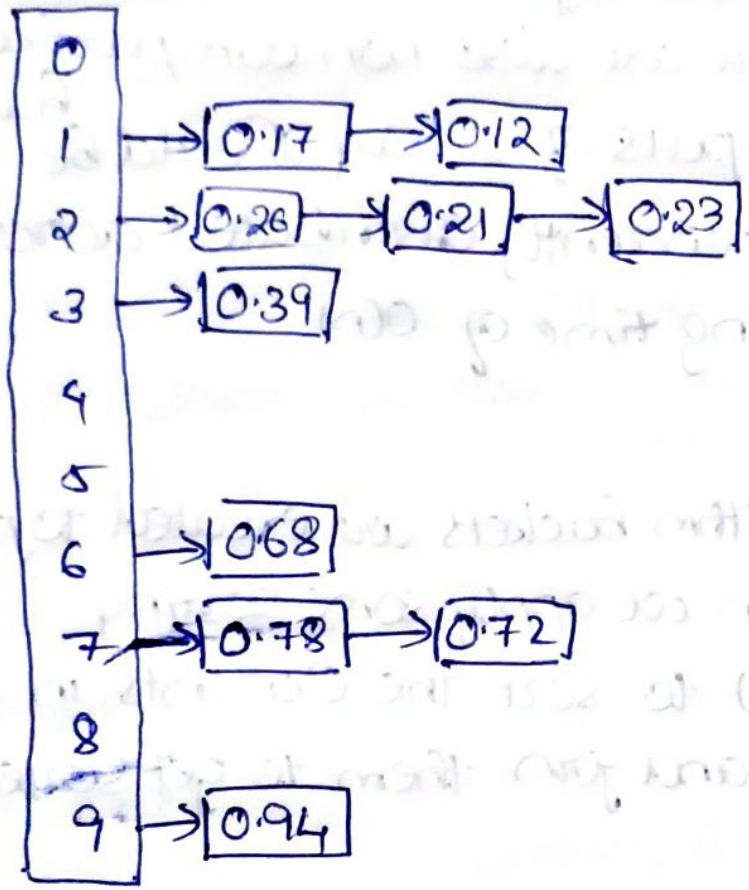
Consider array,

array = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]

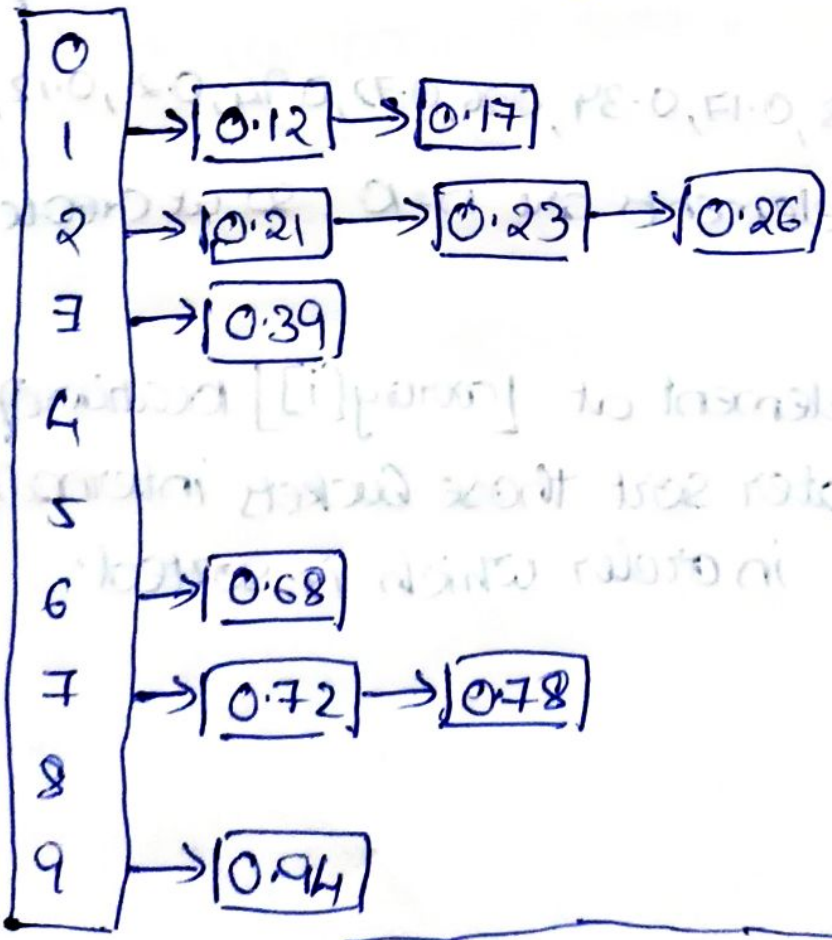
Total number of elements are n=10, so we create 10 buckets

Store each ith element at [array[i]] location of bucket sort. And later sort those buckets internally and print them out in order which is sorted.

array = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]



Sort among themselves using insertion sort



Sorted array [0.12 0.17 0.21 0.23 0.26 0.39 0.68 0.72 0.78 0.94]

Basic procedure of Bucket sort

1. First partition the range into a fixed number of buckets
2. Then, toss every element into its appropriate bucket
3. After that, sort each bucket individually by applying a sorting algorithm
4. And at last, concatenate all the sorted buckets.

Advantages of bucket sort

Bucket sort reduces the no. of comparisions

Fast because of the uniform distribution of elements

Limitations of bucket sort

not useful for large array's because it increases cost.

not an inplace algorithm, occupies an extra space to sort buckets

Mostly used for

1. array with floating point values.
2. array with input distributed uniformly over range.

Time Complexity

Best Case:

* when no sorting req
* uniformly distr in buckets
* Ele in buckets are also sorted.
* Even we use Ins sort in array we will have linear T.C as it is already sorted

$O(n+k)$

Average case

* not properly ascending/desc but elem are uniform dist
* no Ins sort needed in specific, distributing into buckets itself sort the array

$O(n+k)$

Worst case

* Elements are of close range so they are placed in same bucket.
* If elements of these buckets are in reverse order it takes $O(n^2)$ time for ins sort

Space Complexity

$O(n \times k)$