Data Structures and Algorithms

-Manvith Prabhu

ARRAYS

Syntax: datatype arrayname[size];

Arrays are ordered arrangement of same type of data.

Insertion at a position

1) Start
2) for (i=n-1; i>=pos; i--)
   A[i+1] = A[i]
3) A[pos] = data
4) n = n+1
5) Stop

Deletion at a position

1) Start
2) data = A[pos]
3) for (i=pos; i<n; i++)
   A[i] = A[i+1]
4) n = n-1
5) Stop

Searching algorithms

1) Linear search:

1> Start
2> loc = -1
3> for (i=0; i<n; i++) O(n)
{ if (A[i] == search)
loc = i }
4> if (loc > -1)
print loc
else print search unsuccessful
5> Stop

2) Binary search algorithm (for sorted array)

1> Start $O(\log_2 n)$
2> B = lb E = UB loc = -1
3> while (B <= E)
{ mid = int (B+E)/2;
if (ele == a[mid]
loc = mid
else if (ele < a[mid])
E = mid - 1;
else B = mid + 1; }
4> if (loc > -1)
print loc
else print search unsuccessful
5> Stop

Sorting algorithms :

1> Selection sort :

1> Start
2> Input array
3> for (i=0; i<n-1; i++)
4> min = i;
5> for (j=i+1; j<n; j++)
    if (a[min] > a[j])
    { t = a[min]
    a[min] = a[j]
    a[j] = t }
6> output array
7> Stop.

$O(n^2)$

2> Bubble sort

1> Start
2> Input array
3> for (i=0; i<n-1; i++)
4> for(j=0; j<n-i-1; j++)
{ if (a[j] > a[j+1])
{ t = a[j];
a[j] = a[j+1];
a[j+1] = t; }
}
5> output array
6> Stop.

$O(n^2)$

3> Insertion sort

1> Start
2> Input array
3> for (i=1; i<n; i++)
for (j=i; j>0; j--)
{ if (a[j] < a[j-1])
{ t = a[j];
a[j] = a[j-1];
a[j-1] = t}
}
4> output array
5> Stop.

$O(n^2)$

4> Quick sort : Divide and conquer

$O(n^2)$

1> Start

2> Partition (A, lb, ub)

3> pivot = a[lb];
start = lb;
end = ub;

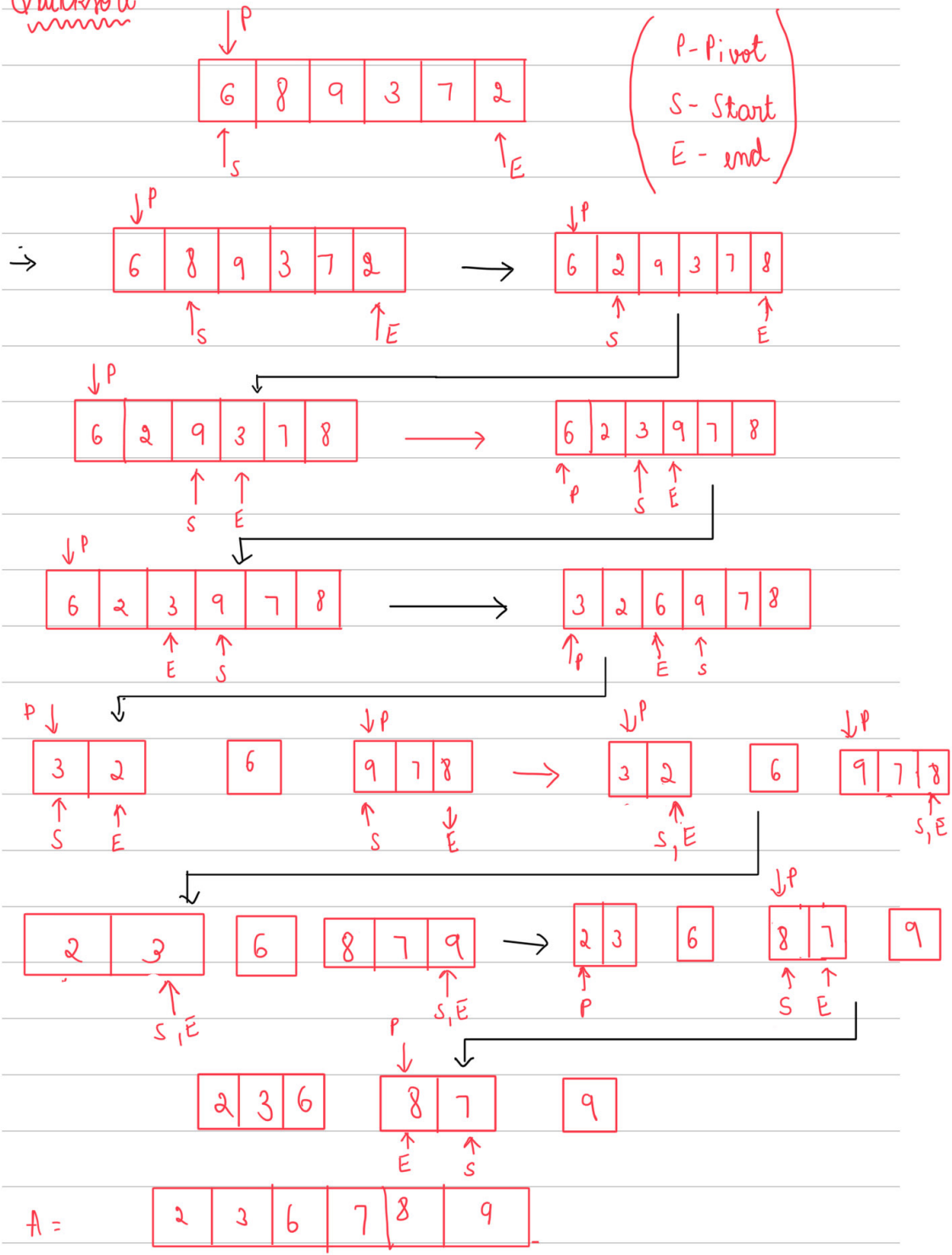
4> while (start < end)
while (a[start] <= pivot)
start ++
while (a[end] > pivot)
end --

5> if (start < end)
swap (a[start], a[end])
end while

6> if (start >= end)
swap (a[lb], a[end])
end if
return end

7> Quicksort (A, lb, ub)
{ if (lb < ub)
{ location = Partition (A, lb, ub)
Quicksort (A, lb, location - 1)
Quicksort (A, location+1, ub) } }

Quicksort

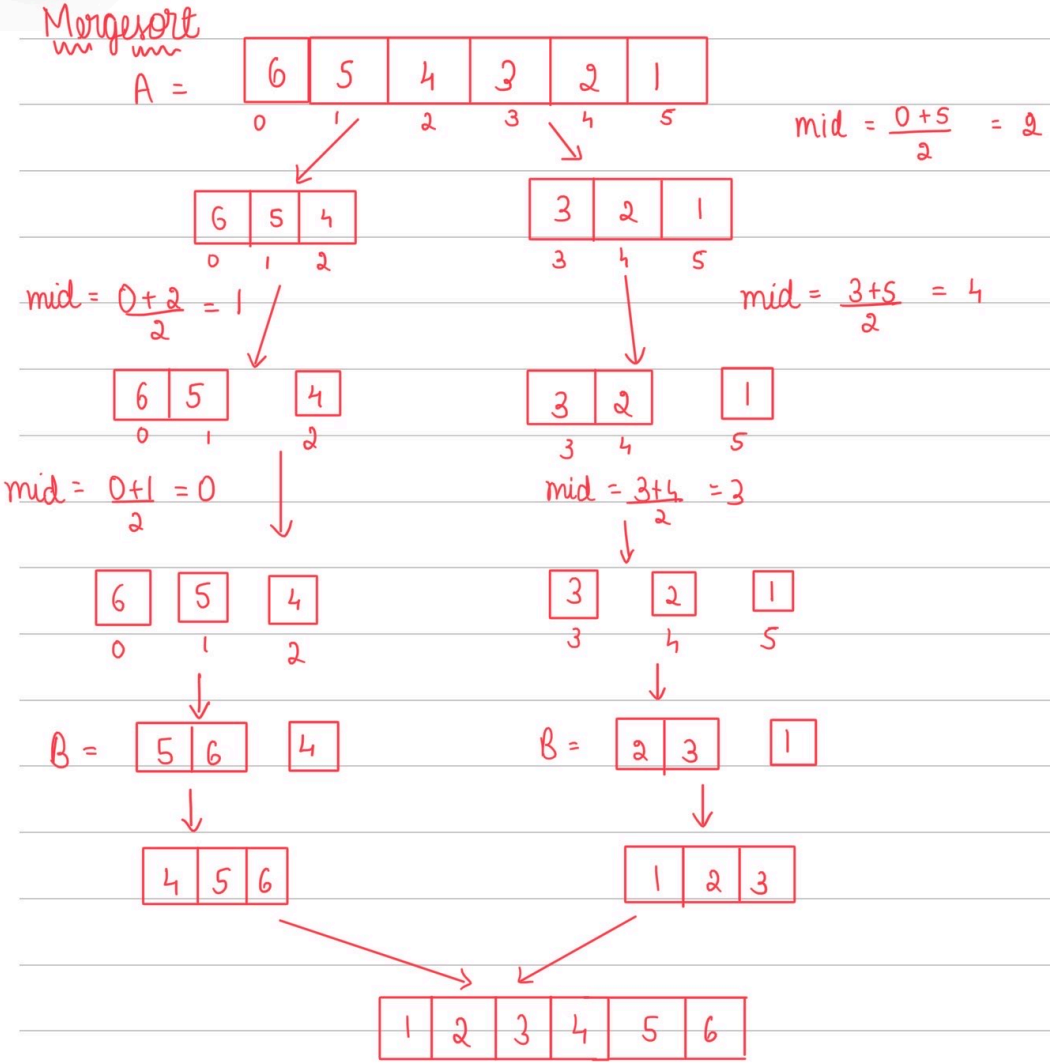


5> Merge sort : Divide & conquer (Find middle element)

$O(n \log_2 n)$

```
1> Start
2> Mergesort (A, lb, ub)
{ if (lb < ub)
   { mid = (lb+ub)/2;
     mergesort (A, lb, mid)
     mergesort (A, mid+1, ub)
   merge (A, lb, mid, ub)
}
3> merge (A, lb, mid, ub)
{ i = lb                          n = ub - lb + 1
   j = mid+1
   k = 0
while (i <= mid && j <= ub)
{ if (a[i] <= a[j])
   { b[k] = a[i]
      i++ }
 else { b[k] = a[j]
        j++ }
  k++ }
  if (i > mid)
  { for (j; j <= ub; j++)
      b[k] = a[j]; k++ }
 if (j > ub)
  { for (i; i <= mid; i++)
      b[k] = a[i]; k++ }
  for (x=0; x < n; x++)
    a[lb+x] = b[x]
```

Mergesort


A =
6 5 4 3 2 1
0 1 2 3 4 5
mid = 0+5/2 = 2
6 5 4
0 1 2
3 2 1
3 4 5
mid = 0+2/2 = 1
mid = 3+5/2 = 4
6 5
0 1
4
2
3 2
3 4
1
5
mid = 0+1/2 = 0
mid = 3+4/2 = 3
6
0
5
1
4
2
3
3
2
4
1
5
B =
5 6
4
B =
2 3
1
4 5 6
1 2 3
1 2 3 4 5 6

6> **Counting sort :**

$O(n)$

1> no comparison
2> check for reoccurance
3> Store in an array
4> Find the exact location to be placed.

Ex:

2	2	1	0	1	3	5	4	6	1	3

range $k = 0-6 : \Rightarrow k = 6$

$n = 11$

count array with all 0 elements. size $= k+1 = 7$

count array:

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

count[i] = count[i] + count[i-1]

count array :

1	4	6	8	9	10	11
0	1	2	3	4	5	6

Sorted array :

0	1	1	1	2	2	3	3	4	5	6
0	1	2	3	4	5	6	7	8	9	10

Algorithm:

1> Start
2> for (i=0 ; i<n ; i++)
    count [a[i]] ++ ;
3> for (i=1 ; i<k+1 ; i++)
    count (i) = count (i) + count (i-1)
4> for (i=n-1 ; i>=0 ; i--)
    B[-- count [a[i]]] = a[i]

7) Radix sort - sub routine

Ex:

008	326	100	073	169	456	011	079	132

Pass 1: counting sort for LSB:

count array

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

count(i) = count(i) + count(i-1)

count array:

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

B:

100	11	132	73	326	456	8	169	79
0	1	2	3	4	5	6	7	8

Pass 2: counting sort for 2nd bit:

count array:

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

count(i) = count(i) + count(i-1)

count array:

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

B:

100	8	11	326	132	456	169	73	79
0	1	2	3	4	5	6	7	8

Pass 3: counting sort for 3rd bit:

count array:

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

count(i) = count(i) + count(i-1)

count array :

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

B :

8	11	73	79	100	132	169	326	456
0	1	2	3	4	5	6	7	8

Algorithm :

1> Start

2> radix sort (A, n)

```
{ max = getmax (A, n)
  for ( pos =1 ; (max/pos)>0 ; pos * 10)
    { count sort (A, n, pos)
  }}
```

3> count sort (A, n, pos)

```
{  count [10 ] = {0};
   for (i=0 ; i <n ; i++)
       count [(A[i]/pos) %. 10 ] ++
   for (i=0 ;  i<10; i++)
       count [i] = count [i] + count [i-1];
   for (i= n-1; i>0 ; i--)
     B [-- count [(A[i]/pos) %. 10]] = A.[i]
   for  (i= 0;   i<n ; i++)
    A.[i] =  B[i]
}
```

Time complexity = $O(d * (n+b))$

b = base (here 10)
d = no. of digits
n = no. of elements

Conversion of infix to prefix / postfix

Priority: 1. ( ), [ ], { }
2. ^ → Right to left
3. / * → Left to right
4. + , - → Left to right

Rules:

1> operands → Print
2> (→ opening bracket → Push into stack.
3> If it is operator:
i> If stack is empty push it.
ii> If to top of stack is opening bracket, push it.
iii> If it has higher priority push it, else pop the top of stack and repeat step 3.
iv> If it has equal priority check associativity.
4> If it is closing bracket: pop all operators until closing bracket is encountered.

- To convert to prefix:
1> Reverse the given infix expression (convert closing braces to open braces and open braces to closing braces)
2> Convert it to postfix by applying the steps above.
3> Reverse the obtained postfix expression to get prefix expression.

Queues: • It is a linear abstract data type.
It is based on FIFO principle.
• It can be implemented using stack, linked list, arrays.

Enque Algorithm - Array

1> Start
2> if (rear == max-1)
overflow
3> if (rear == -1 && front == -1)
{ rear ++; front ++;
Queue[rear] = data }
end if
4> else { rear ++;
Queue[rear] = data; }
5> Stop

Deque algorithm - Array

1> Start
2> if (rear == -1 && front == -1)
underflow
3> if (front == rear)
{ data = Queue[front];
front = -1; rear = -1; }
end if

4) else { data = Queue[front]
front ++ ; }

5) Stop

Enque algorithm - Linked list

1) Start
2) if (front == NULL && rear == NULL)
{ n = new node
front = n rear = n }
end if
3) else { n = new node ;
rear → next = n ; rear = n ; }
4) Stop

Deque algorithm - Linked list

1) Start
2) if (front == rear)
{ t = front
front = NULL
rear = NULL
delete t } end if
3) else
{ t = front
front = front → next
delete t }
4) Stop

Circular queue :

Enque - array

1> Start
2> if (Front == (rear + 1) %. N)
    Overflow
3> if (rear == -1 && front == -1)
    { rear ++; front ++;
      Queue [rear] = data }
    end if
4> else if (rear == N - 1)
    { rear = 0; Queue [rear] = data }
    end if
5> else { rear ++;
      Queue [rear] = data ; }
6> Stop

Dequeue - array

1> Start
2> if (rear == -1 && front == -1)
    underflow
3> data = Queue [front]
    if (front == rear)
    front = -1
    rear = -1

4) if (front == n-1)
front = 0;
else front ++;
5) Stop.

Linked lists

Insertion:

1) Start
2) count = 0 , t = head
3) n = new node
n -> data = num
4) while (t != NULL)
count ++
t = t -> next
end while
5) if (pos == 1)
{n -> next = head
head = n }
6) if (pos == count + 1)
{ n -> next = null
t -> next = n }
7) if (pos <= count)
{ t = head;
for (i=1 ; i<pos-1; i++)
{t = t -> next }

```
        n->next = t-> next
        t-> next = n }
8> else  print "invalid  position"
9> Stop
```

Deletion:

```
1> Start
2> count = 0        t1 = head
3> while (t1 != NULL)
     count= count +1
     t2 = t1
    t1 = t1-> next
    end  while
4> if (pos == 1)
    { t1 = head
     print  t1-> data
      head = t1-> next
      delete  t1 }
5> if (pos == count )
    { print   t1-> data
        t2 -> next = NULL
       delete  t1 }
6> if (pos < count)
     t1 = head
  for ( i=1 ; i<pos ; i++)
     { t2 = t1
       t1 = t1-> next}
```

print t1->data
t2->next = t1->next
delete t1
}
7> else print "invalid position"
8> Stop.

Asymptotic analysis

1> O -> Worst case time complexity
2> Ω -> Best case time complexity
3> Θ -> When runtime is same for all cases.

Stacks:

1) Push :- array

1) Start
2) if (top == max-1)
   print overflow
3) else { top = top + 1
   stack [top] = value }
4) Stop

2) Pop :- array

1) Start
2) if (top == NULL)
   print underflow
3) else value = Stack [top]
4) top = top - 1
5) Stop

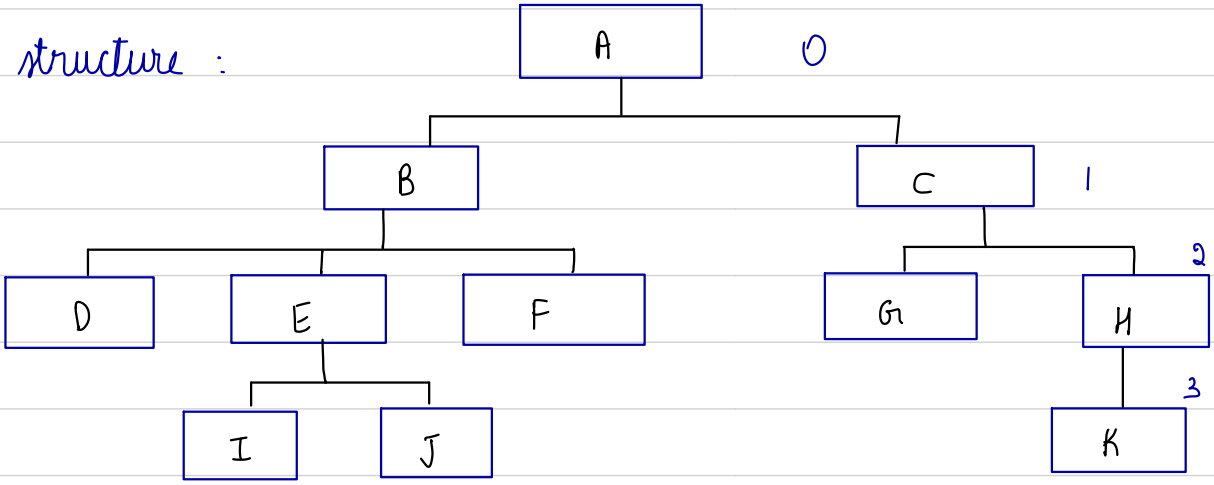
3) Peek :- array

1) Start
2) if (top == NULL)
   print underflow
3) else return stack [top]
4) Stop

Non linear data structures

Trees: (When there is hierarchy, trees are used)

General structure:



Root node: Nodes which do not have any parent (Eg: A)

Intermediate nodes: Nodes which are between root and leaf:
Eg: (B, E, C, H)

Leaf node: Nodes which do not have any child node.
Eg: (D, F, I, J, G, K)

Level / Depth: Distance between root and any node
Ex: Depth of G = 2

Height: Distance between root and last leaf node
Ex: Height of the tree = 3 (max depth)

Degree of a node: Number of child nodes that a node has.

Traversal of tree

1) Ordered tree: Left to right order is maintained in data.

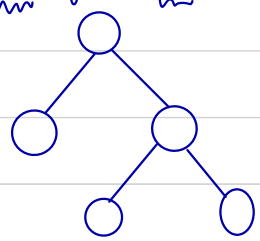
NOTE: Empty trees are possible.

- Binary tree: It is an ordered tree where every node will have atmost 'two' child nodes.

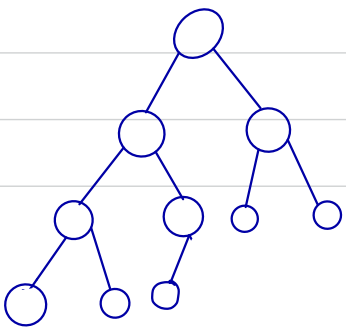
2) Unordered tree: No ordering is maintained.

Different types of Binary trees

i) Full Binary tree: Each node will have 0 or 2 child nodes.
Ex:

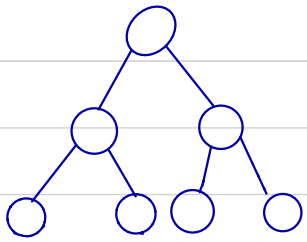


ii) Complete binary tree: It is a binary tree in which all levels except the last level is completely filled and all nodes in last level are as left as possible.
Ex:



iii) Perfect binary tree: All nodes have 2 child nodes and all leaf nodes are at same level.

Ex:



Tree traversal / tree walk

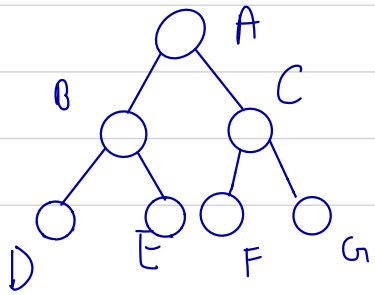
	First	Second	Last
1) Preorder Traversal	Root	left	right
2) Inorder traversal	left	root	right
3) Postorder traversal	left	right	root

1) Preorder traversal

Algorithm:

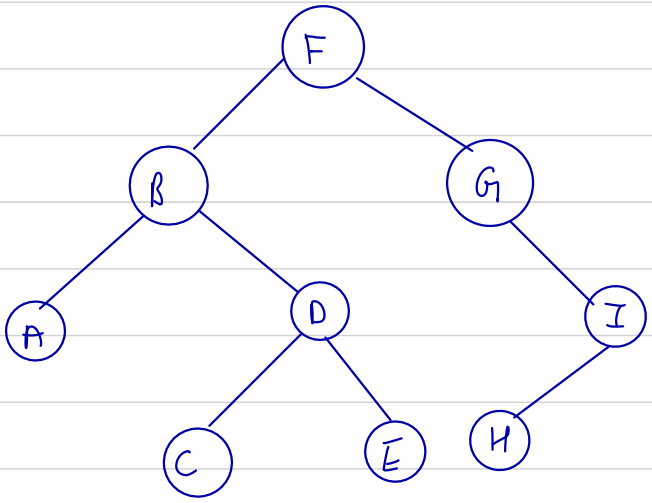
1) Start
2) Preorder (v)
3) If (v == null), Return
   visit (Root)
   Preorder (v. left child())
   Preorder (v. right child())

Ex a)



A → B → D → E → C → F → G

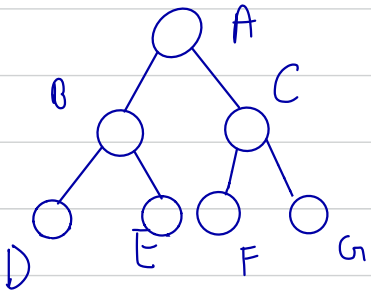
Ex b:



Ans: F → B → A → D → C → E → G → I → H

2) Inorder traversal

Ex a:

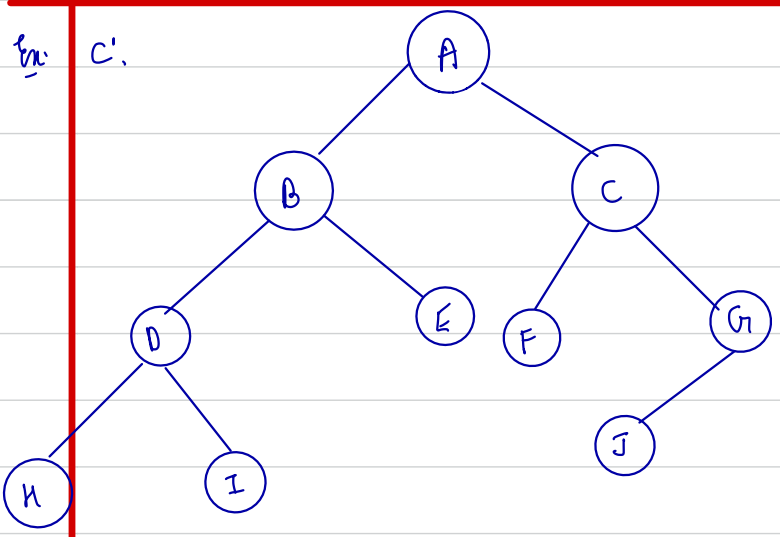


D → B → E → A → F → C → G

Algorithm

1) Start
2) Inorder (v)
   If (v == null), Return
3) Inorder (v.left())
   visit (root)
   Inorder (v.right())
4) Stop

Ex: c:



Preorder: A, B, D, H, I, E, C, F, G, J

Inorder: H, D, I, B, E, A, F, C, J, G

3> Postorder: For ex a: D, E, B, F, G, C, A

For ex b: A, C, E, D, B, H, I, G, F

For ex c: H, I, D, E, B, F, J, G, C, A

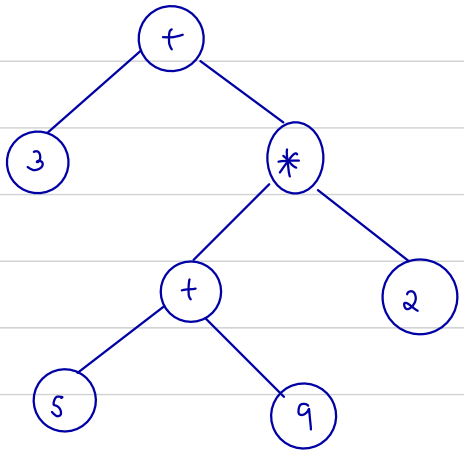
Algorithm:

1> Start
2> Postorder (v)
  If (v==null), return
3> Postorder (v.left)
  Postorder (v.right)
  visit (root)
4> Stop

Binary expression tree

- It is a binary tree which is used to represent mathematical expression, where each internal node and root node represents operator and leaf node represents operand.
- Inorder traversal is used.

Eg: 1>



Inorder

→ 3 + (5 + 9) * 2

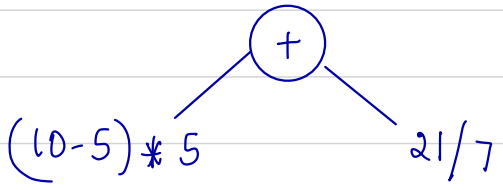
= 3 + 14 * 2

= 3 + 28

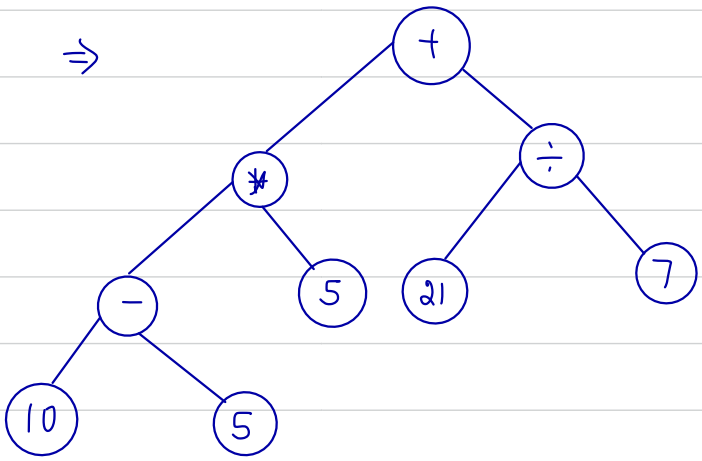
= 31

2> (10 - 5) * 5 + 21/7

Rules: 1> Root is last priority and associvity operator
2> Divide the expression accordingly
3> Again go to rule 1.



⇒



Eg 3: Write prefix & postfix for eg 1:

Ans: Prefix : + 3 * + 5 9 2

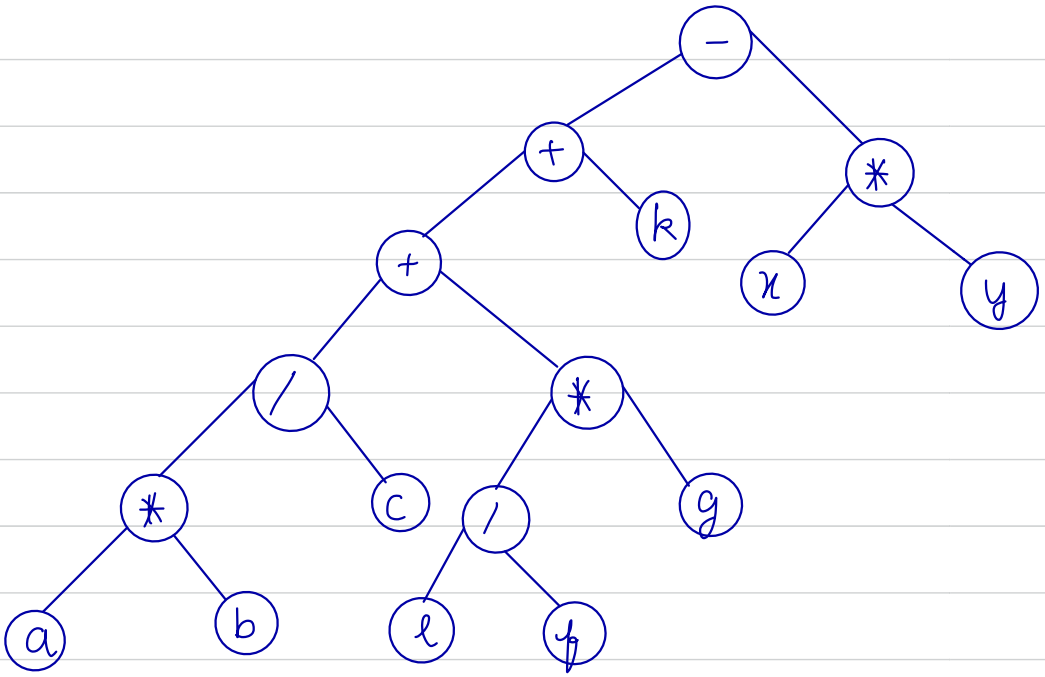
Postfix : 3 5 9 + 2 * +

Eg 4: Write prefix & postfix for eg 2:

Ans: + * − 10 5 5 ÷ 21 7 ← Preorder

Postorder : → 10 5 − 5 * 21 7 ÷ +

Eg 5: a * b / c + e / f * g + k − x * y

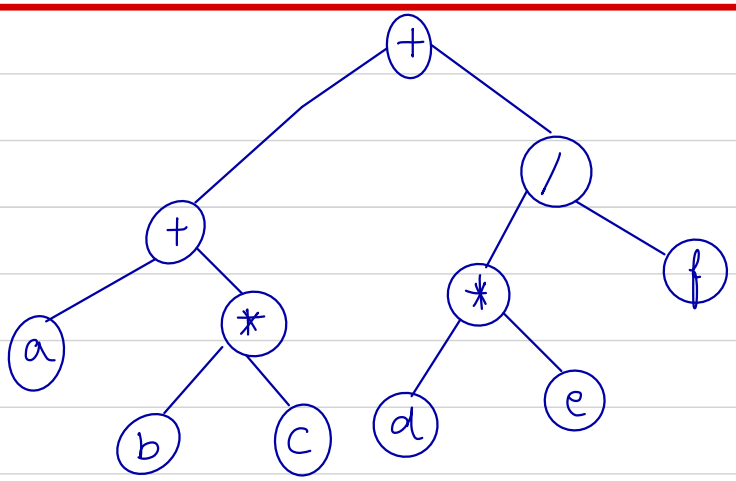


Prefix: − + + / * a b c * / e f g k * x y

Postfix: a b * c / e f / g * + k + x y * −

Eg 6: a + b * c + d * e / f

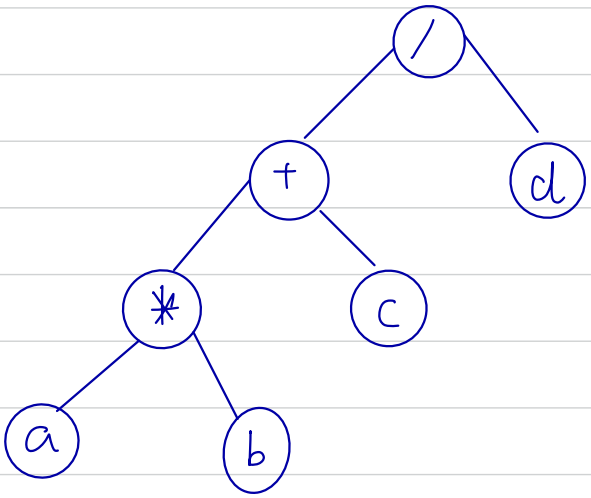
Eg6:



Prefix: + + a * b c / * d e f

Postfix: a b c * + d e * f / +

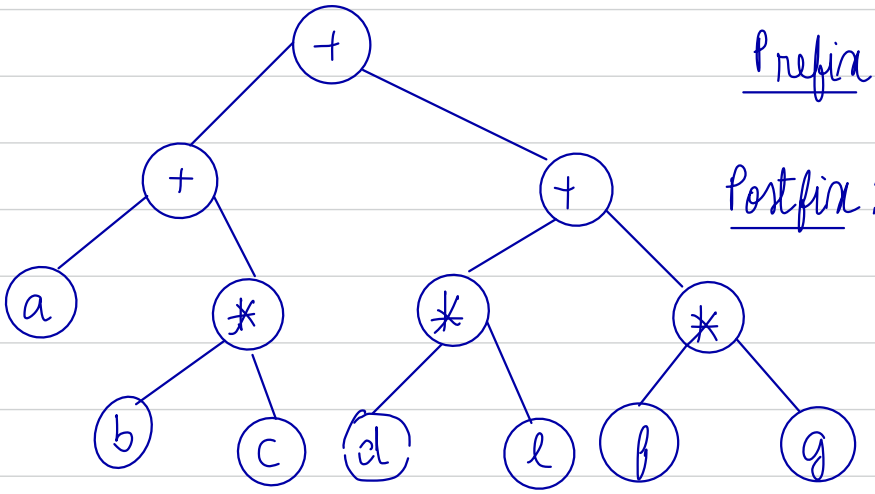
Eg7 $((a * b) + c) / d$



Prefix: / + * a b c d

Postfix: a b * c + d /

Eg8 $(a + b * c) + (d * e + f * g)$



Prefix: + + a * b c + * d e * f g

Postfix: a b c * + d e * f g * + +

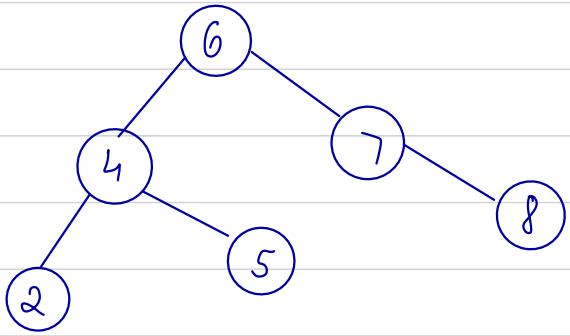
Binary search tree: (BST)

It is a node base tree which has the following property :-

→ Let x be a node in BST. If y is a node in left subtree of x then $y.key \leq x.key$ (key → value of node)

→ If y is a node in right subtree then $y.key \geq x.key$

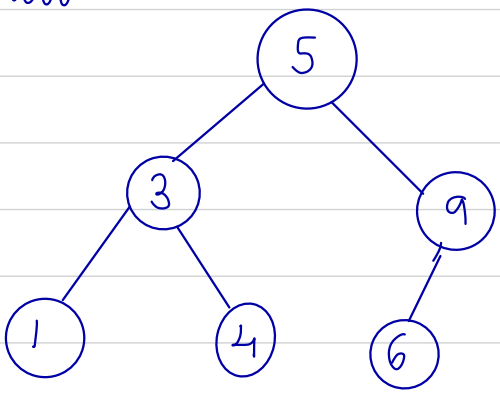
Ex:



1> 5, 3, 1, 9, 6, 4

↓ root

compare > root or < root

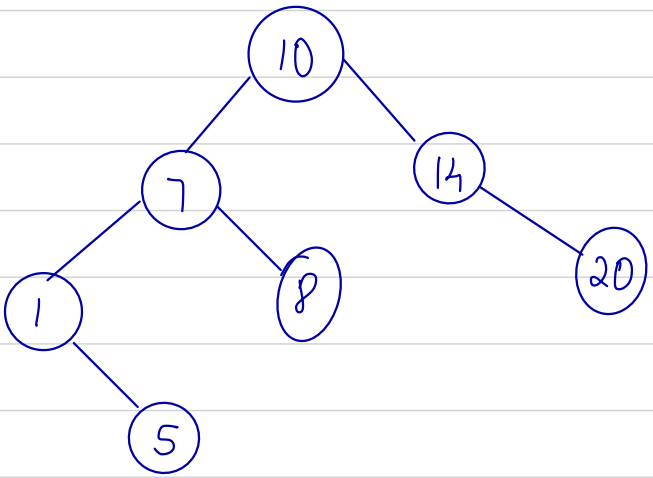


Preorder: 5 3 1 4 9 6
Inorder: 1 3 4 5 6 9
Postorder: 1 4 3 6 9 5

∴ Inorder always gives a sorted order.

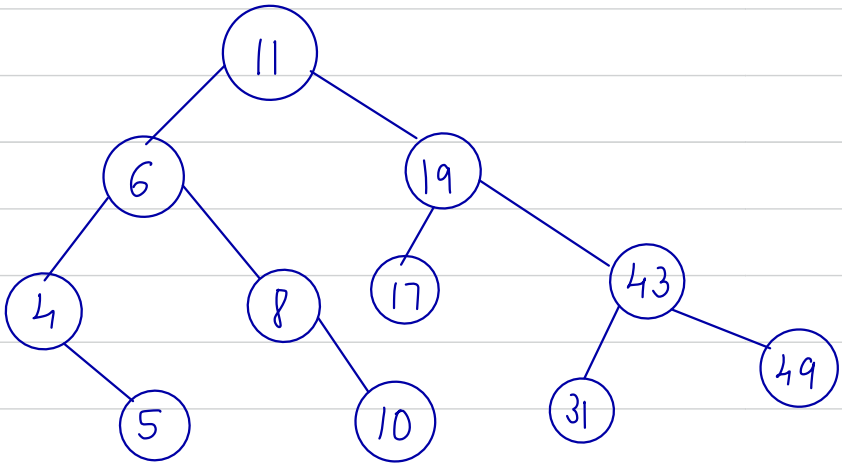
Skew tree: Nodes only towards one side i.e left skew tree or right skew tree.

2> 10, 7, 14, 20, 1, 5, 8



Inorder : 1 5 7 8 10 14 20

3> 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

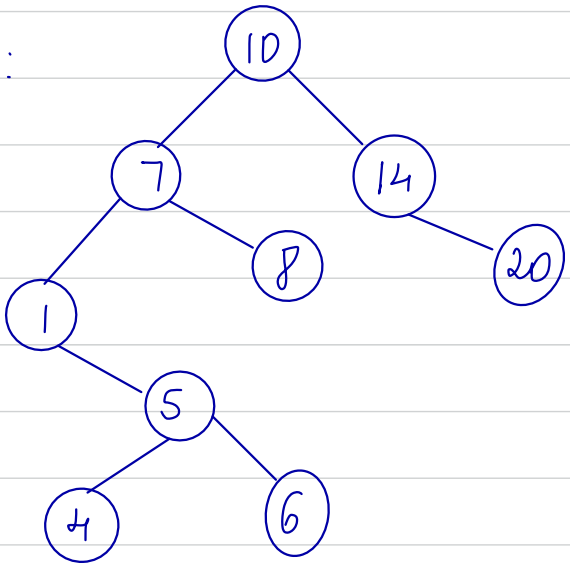


Inorder: 4 5 6 8 10 11 17 19 31 43 49

Insertion: Given a tree, insert a new node as per rules of creation

Deletion: If we want to remove any node with 2 child node, we can replace it with either with predecessor or successor of inorder travel

Eg: Remove 7:

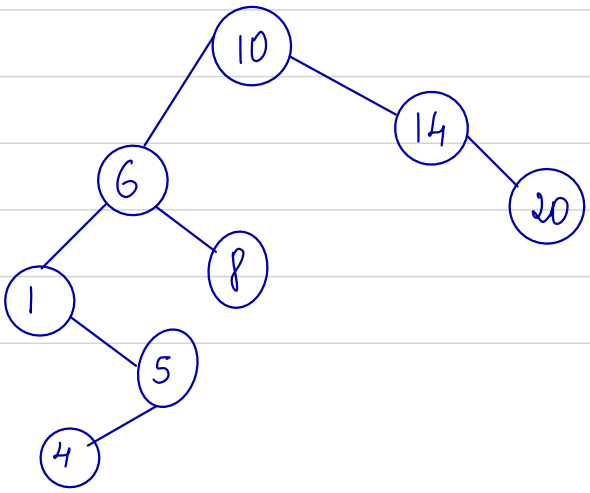


→

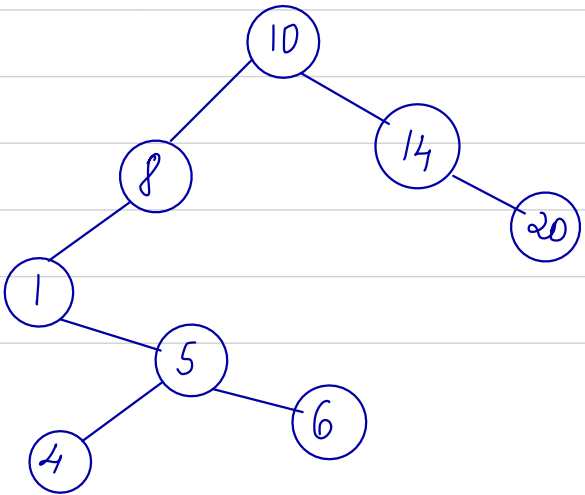
Inorder travel: 1 4 5 6 7 8 10 14 20

We can replace it 6 or 8.

Replace with 6



Replace with 8

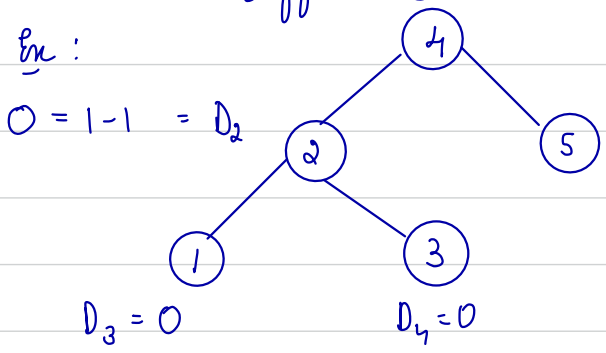


Search: If element is found then side, level should be shown.

Balanced binary search tree

- Also known as Height balanced BST.
- Max difference in height of subtrees is 1.

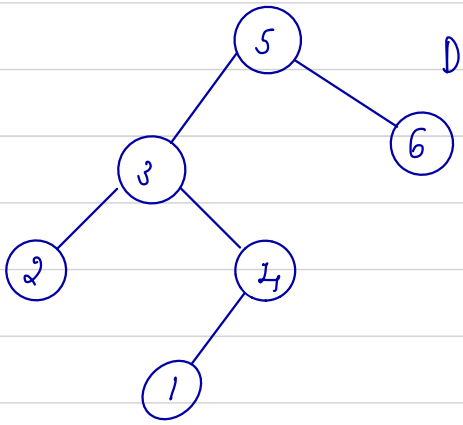
Ex:



Advantage: Reduces search time

- Balanced binary search tree is a BST in which height of left and right subtree is atmost one.

Q> Is it balanced?



⇒ Not balanced

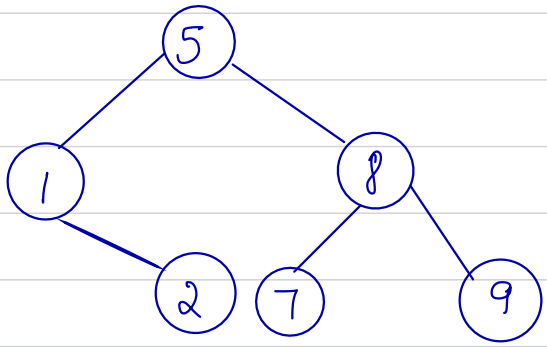
Creation of binary search tree

1> From a list of key values:

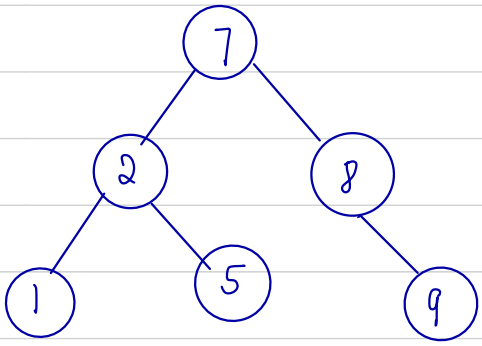
1> Sort the given values
2> Select the mid as root
3> Again select the mid of left & right sublists as roots of left and right subtrees and so on.

Ex: 7 5 2 9 8 1

Am Sort: 1 2 5 7 8 9



or



Priority queues & heaps

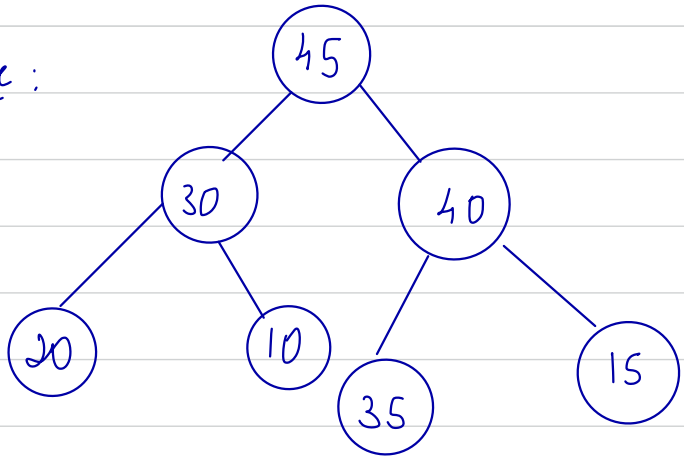
Operations on Priority queue

1> Insertion
2> Deletion
3 Peek
4> Remove

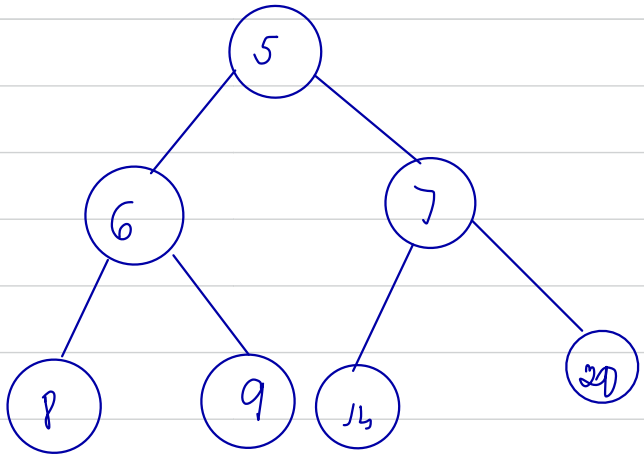
Priority queue is an abstract data type where every element has some priority value associated with it and priority determines the order in which elements are served or processed.

Heap: Heap is a tree type data structure that satisfies the property: if A is a parent node of B then A is ordered with respect to B for all nodes in the heap. Heap should be complete binary tree.

Ex:



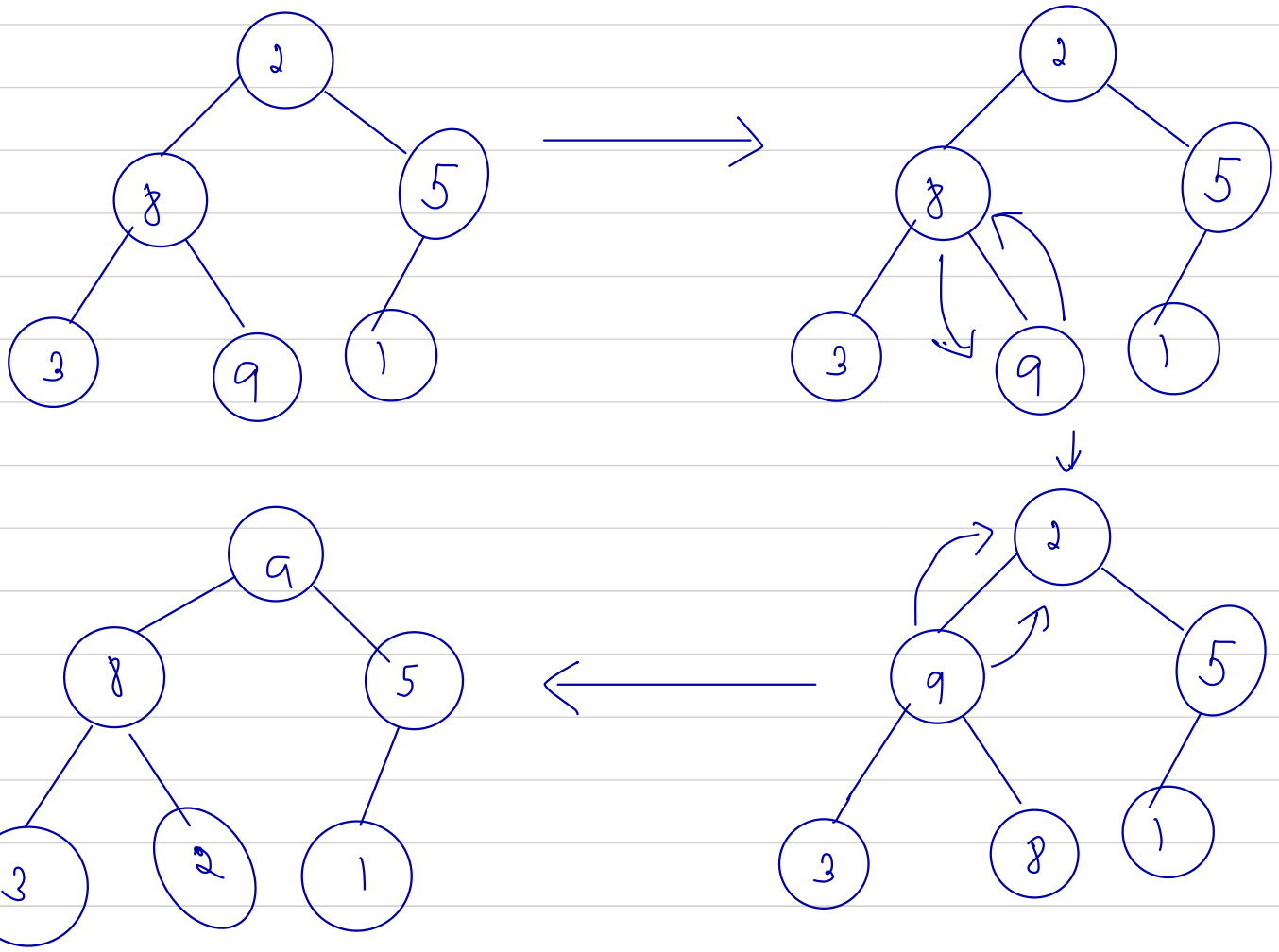
Max heap
Parent > leaf



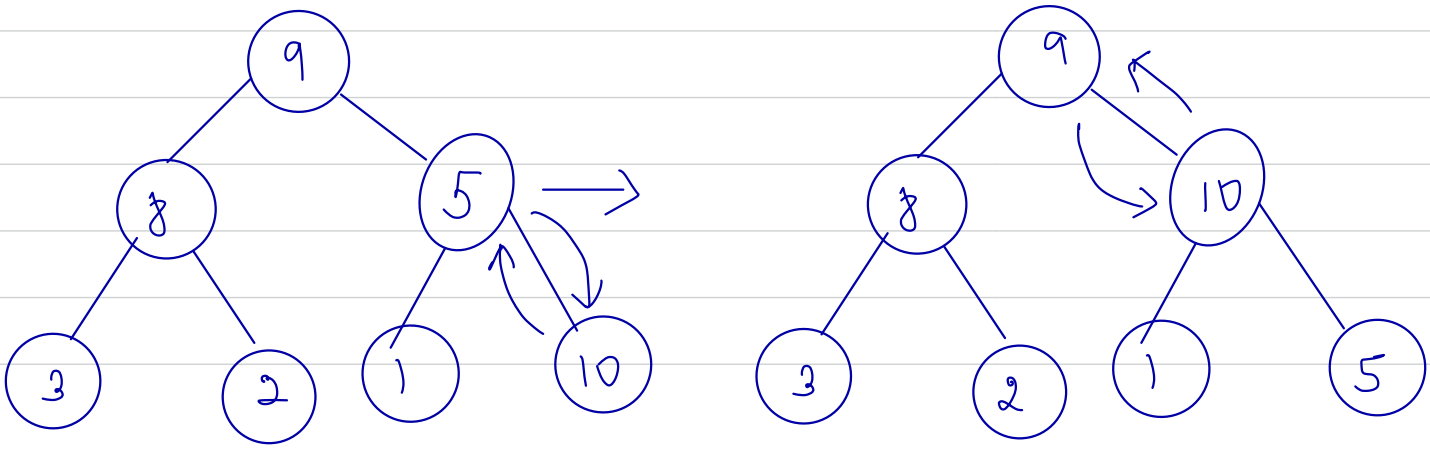
Min heap
Parent < leaf

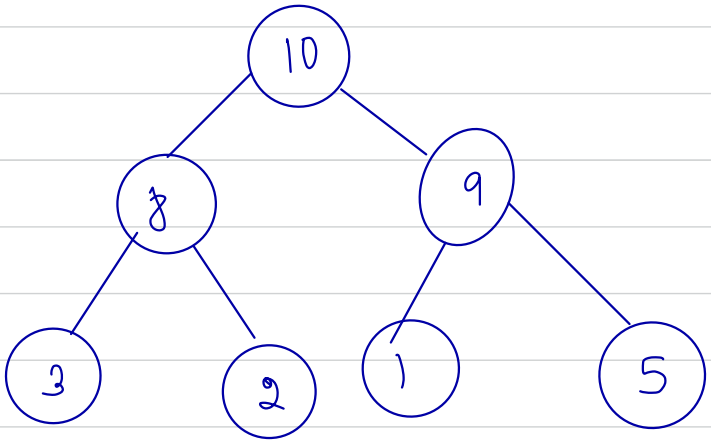
1> Creation of heap

2, 8, 5, 3, 9, 1



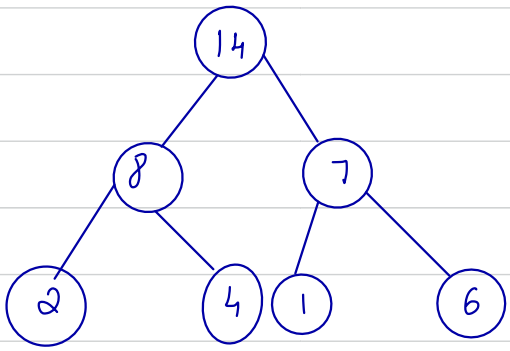
2> Insertion : Add 10 to above heap



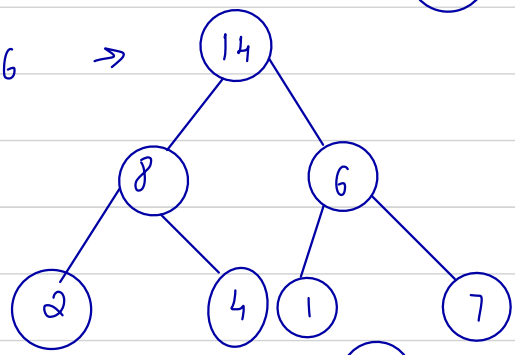


3> Peek: Gives value of root

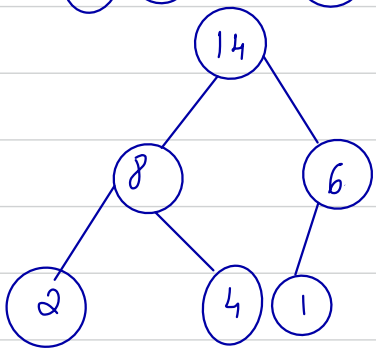
4> Deletion: Remove 7 in



Swap with 6 →



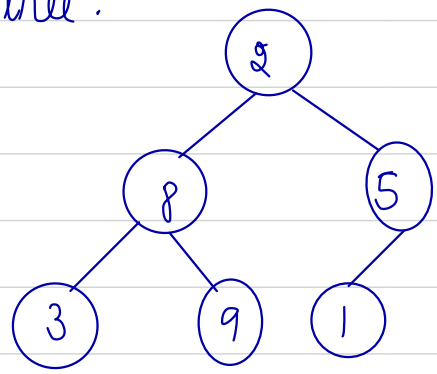
Remove 7 →



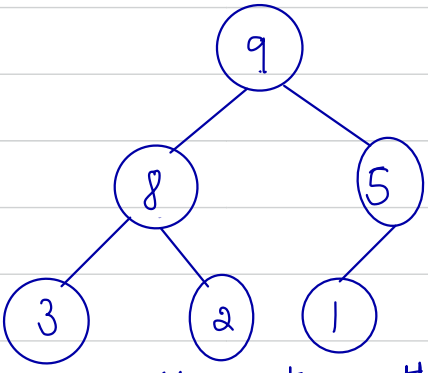
Heap Sort

Ex: 2, 8, 5, 3, 9, 1

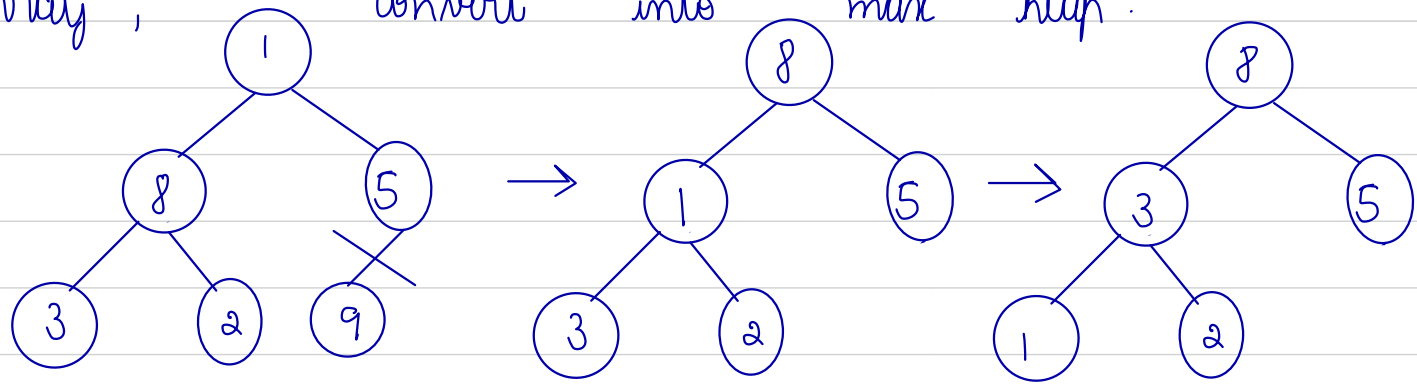
Step 1: Create complete tree:



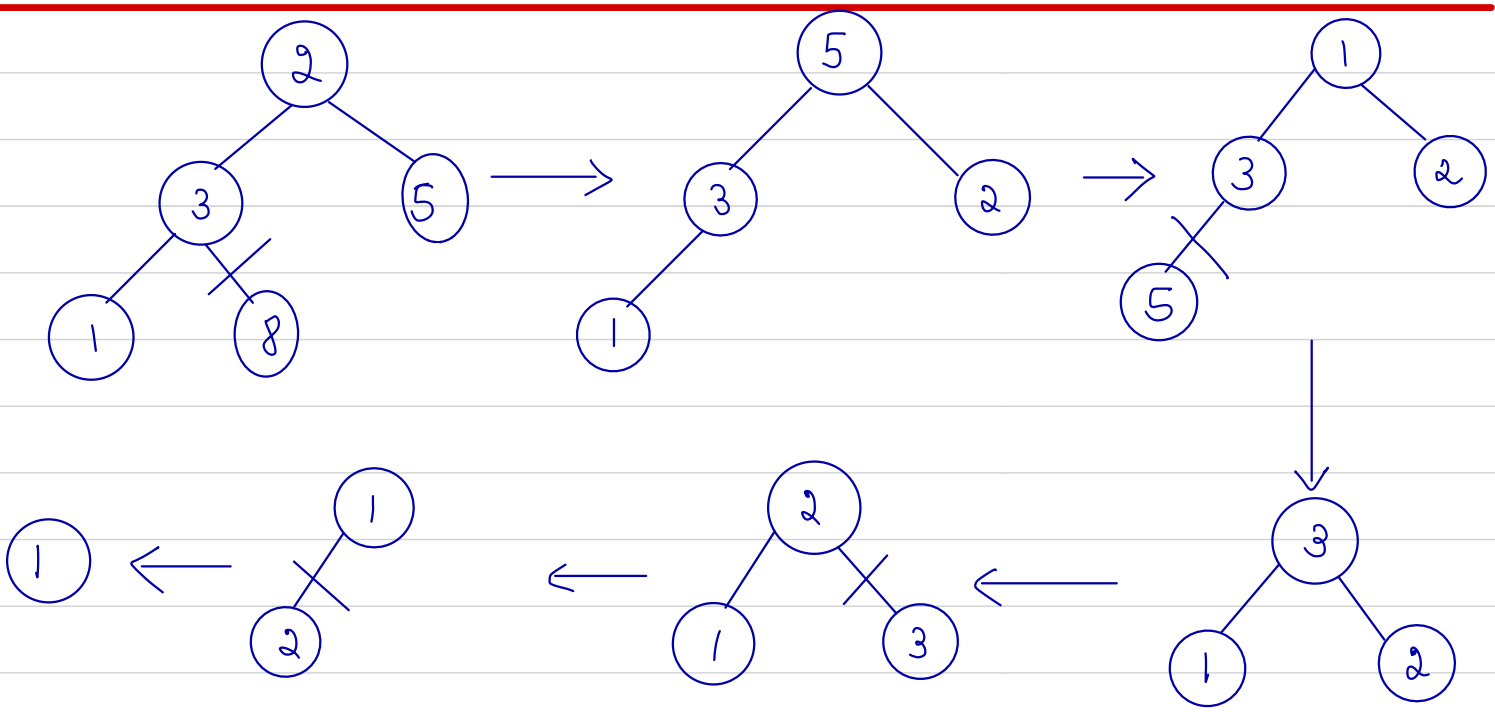
Step 2: Create max heap:



Step 3: Delete root and place it at the end of array, convert into max heap.



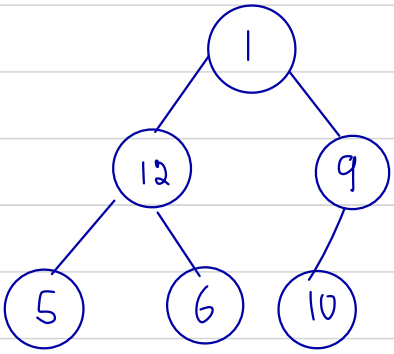
Step 4: Repeat step 3 till only one node is left in tree.



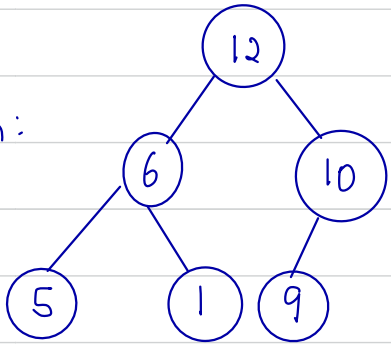
Array :

1	2	3	5	8	9

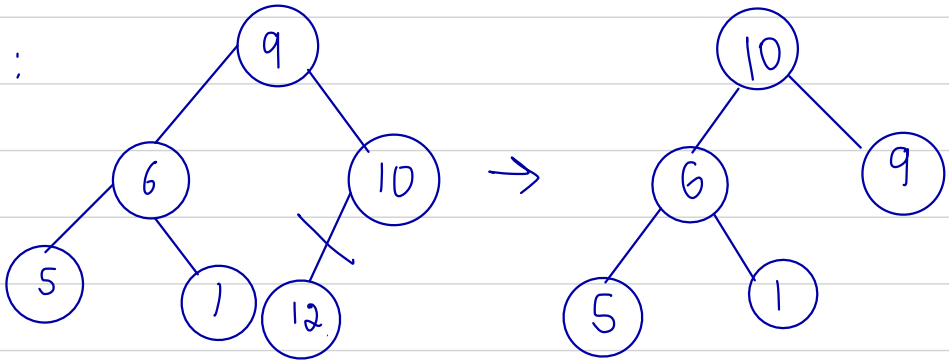
eg 2: 1, 12, 9, 5, 6, 10

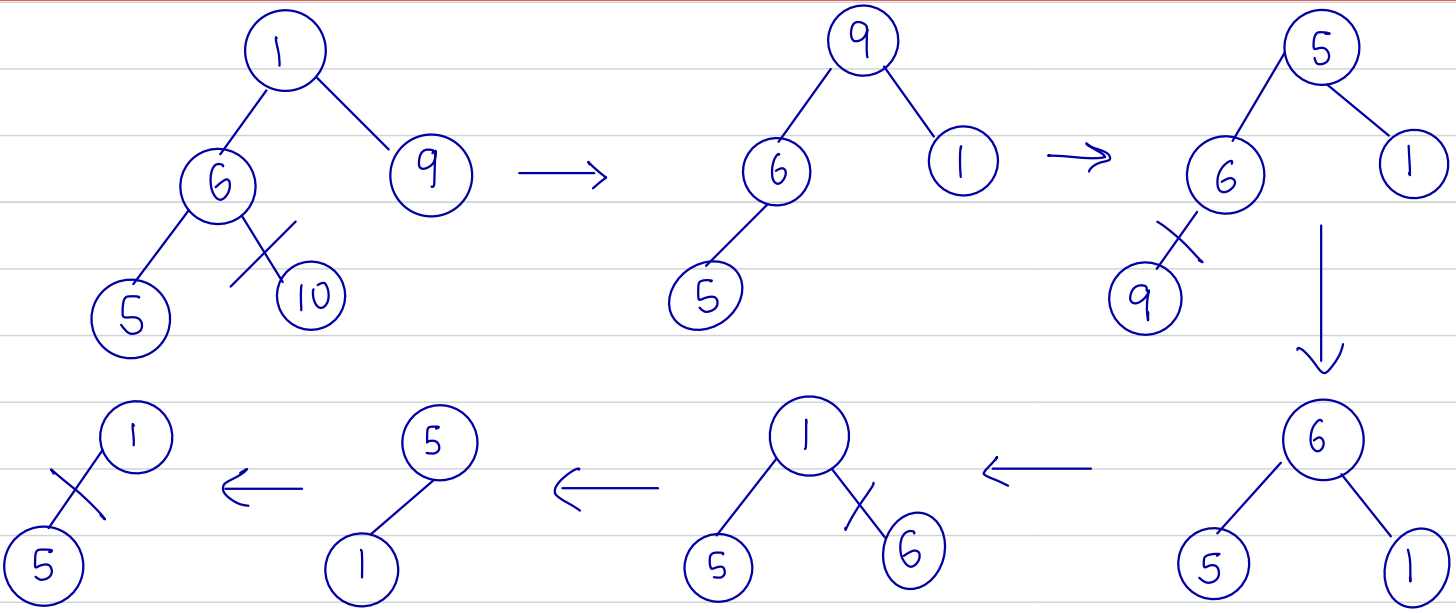


max heap:



Remove root :





Final array :

1	5	6	9	10	12

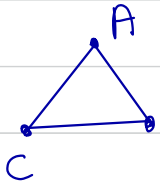
Graphs : It has vertices & edges (atleast one vertex)

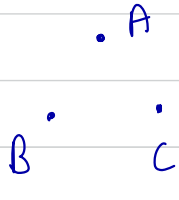
A simple graph $G(V, E)$ consists of a non empty set V of vertices and possibly empty set E of edges with each edge being set of 2 vertices from V. (non directed)

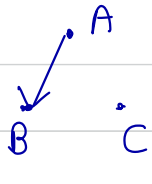
Directed graph : It $G(V, E)$ consists of a non empty set V of vertices and possibly empty set E of edges with each edge being set of 2 vertices from V such that for each edge, $\{v_i, v_j\} \neq \{v_j, v_i\}$

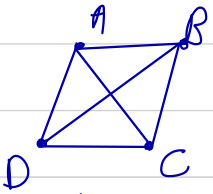
NOTE : $|V|$ = no. of vertices
$|E|$ = no. of edges

Eg :

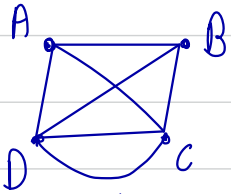




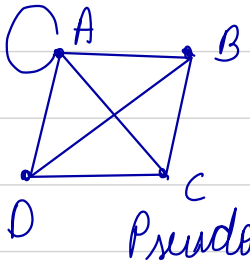




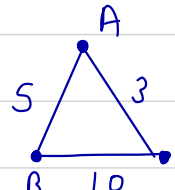
complete graph



multi graph



Pseudograph



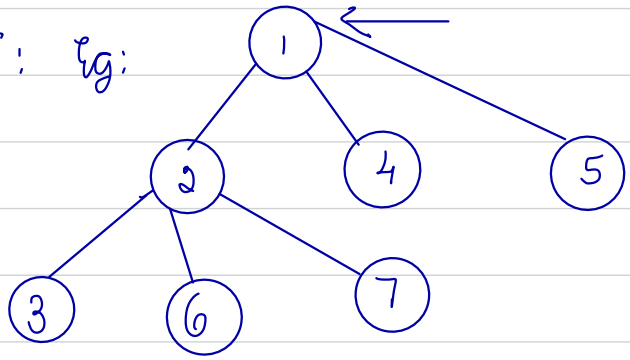
weighted graph

- In a graph if we can reach a node from itself without repitition it is called cyclic graph and if there is repitition it is called path

Graph traversal

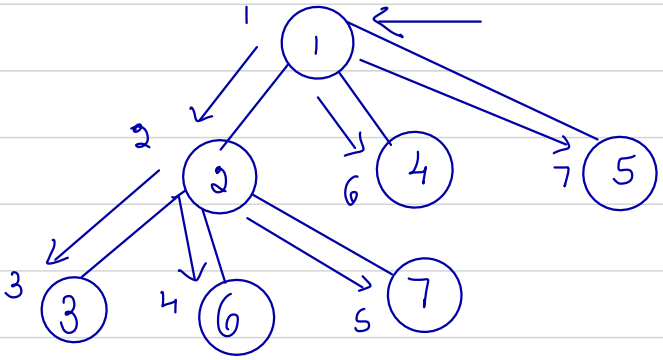
1) DFS - Depth first search
2) BFS - Breadth first search

1) **DFS**: Eg:

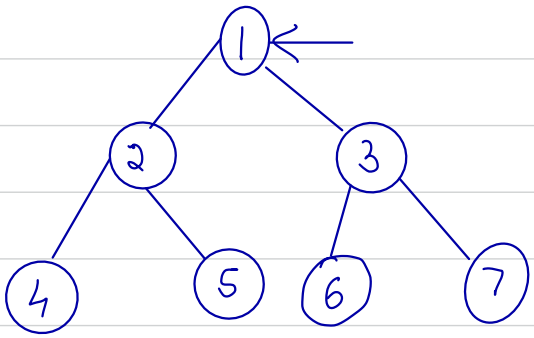


1 → 2 → 3 → 6 → 7 → 4 → 5

i.e



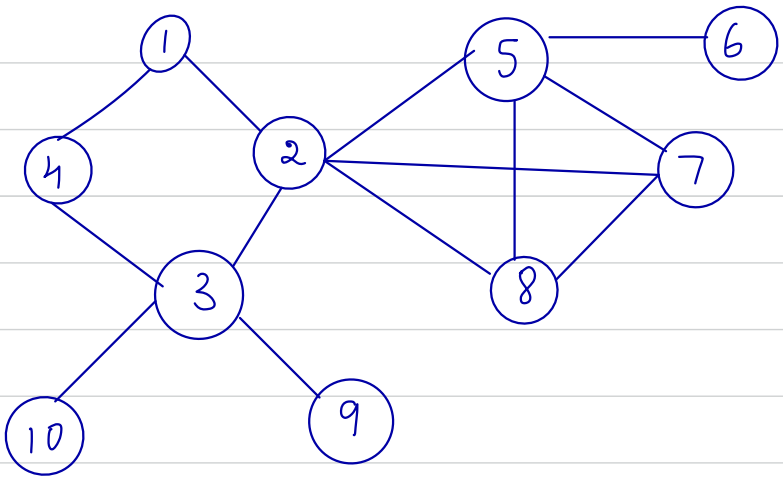
Eg 2:



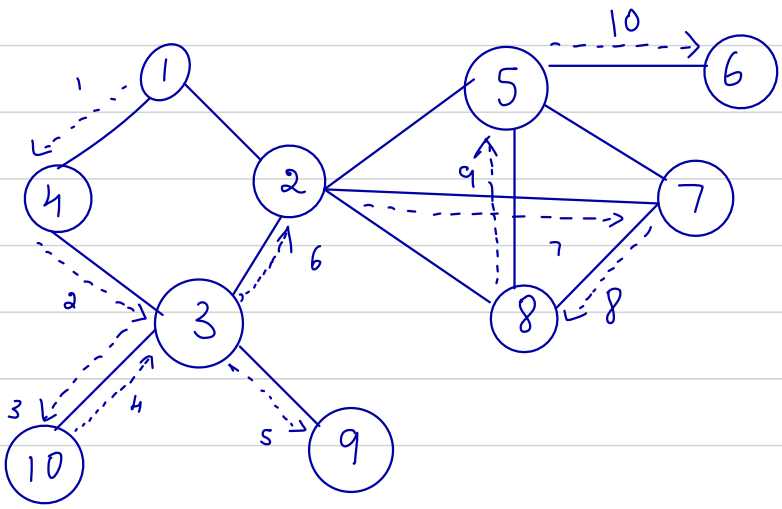
1 → 2 → 4 → 5 → 3 → 6 → 7

NOTE:
- In DFS, stacks are used.
- In BFS, queues are used.

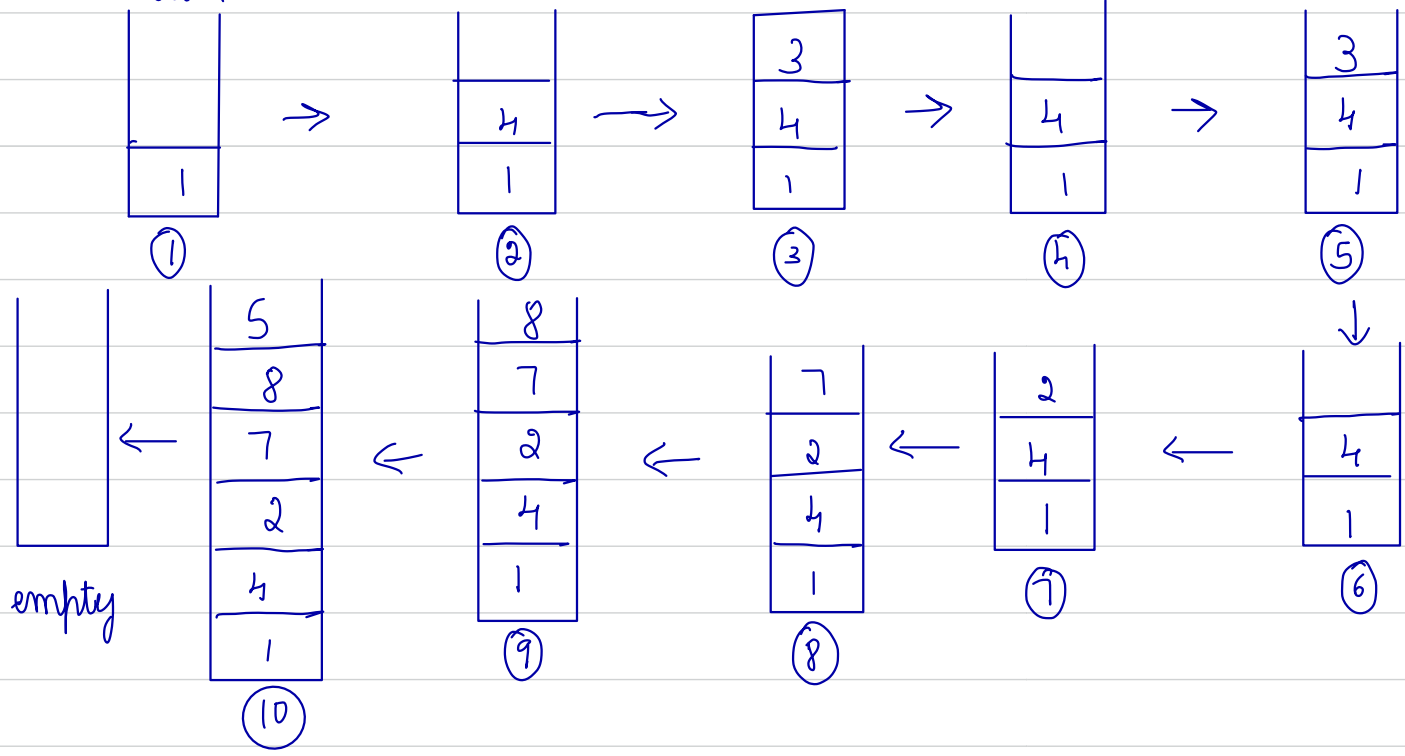
Eg 3:



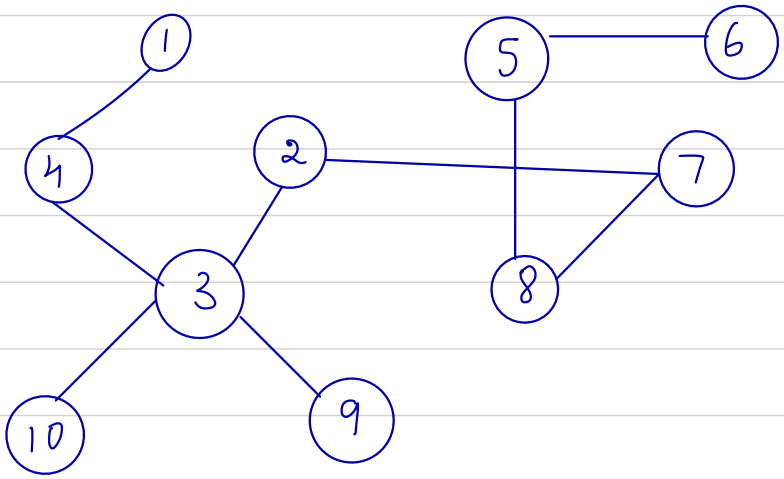
Ans



Stack

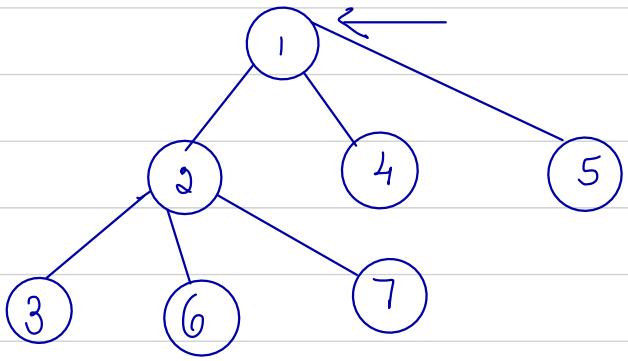


1 → 4 → 3 → 10 → 9 → 2 → 7 → 8 → 5 → 6

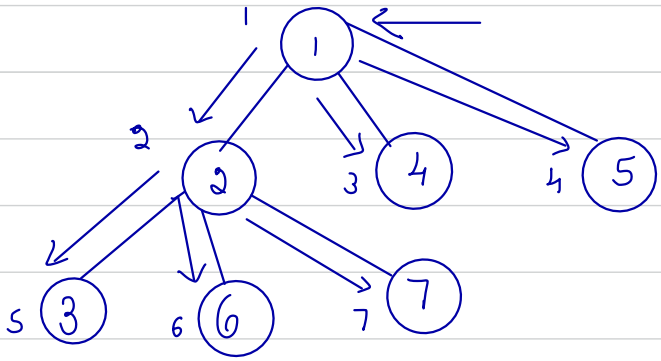


← Spanning tree

2> BFS: Eg 1:

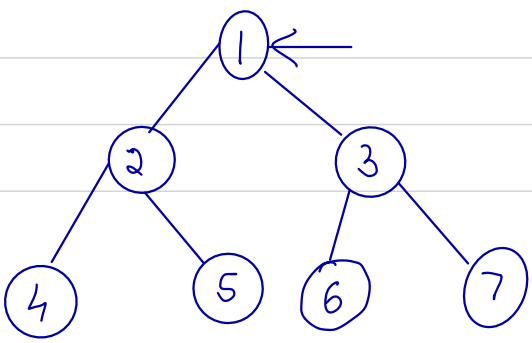


Ans 1 → 2 → 4 → 5 → 3 → 6 → 7



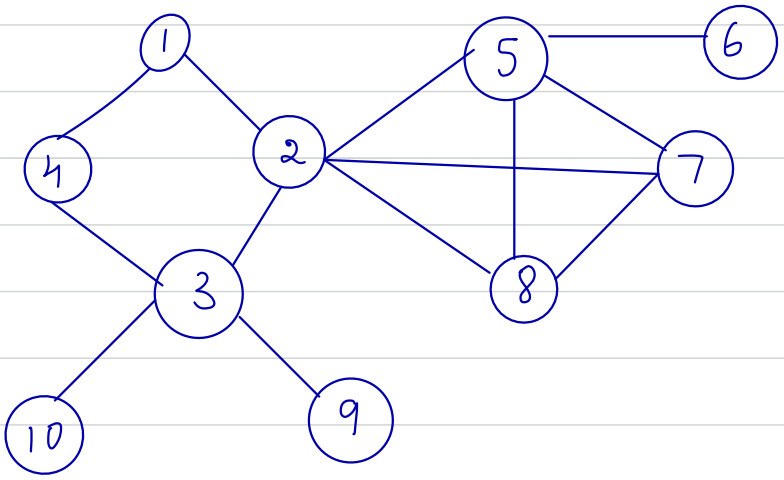
NOTE BFS is also called level ordering in trees.

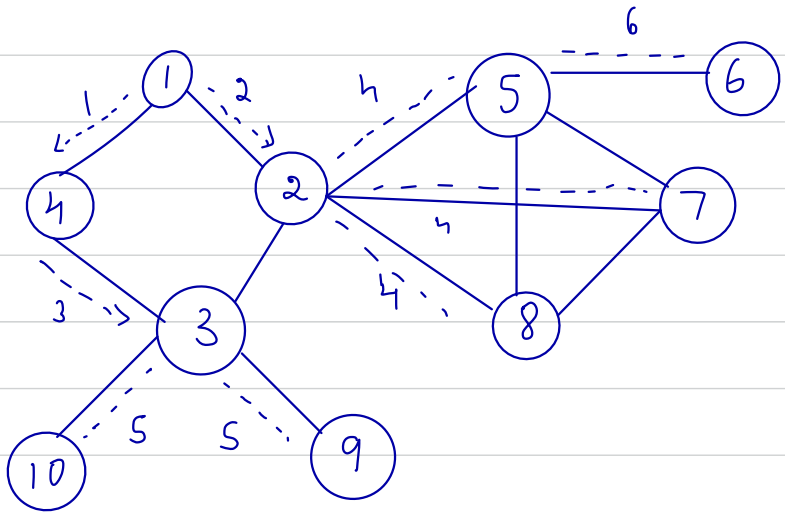
Eg 2:



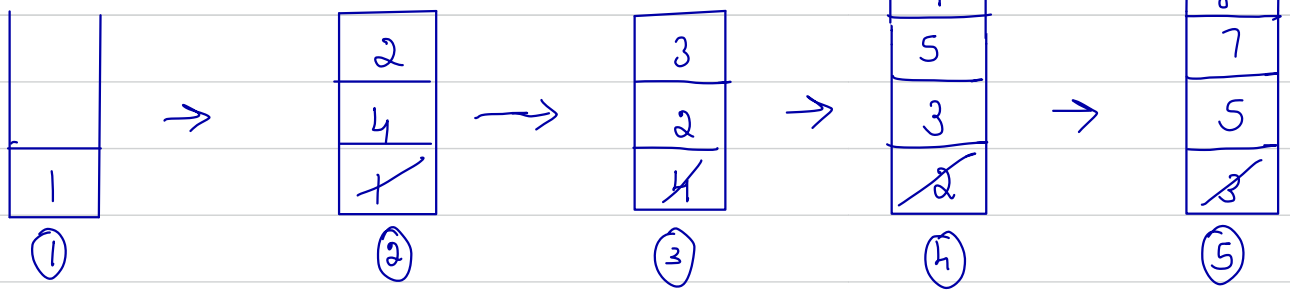
1 → 2 → 3 → 4 → 5 → 7 → 6

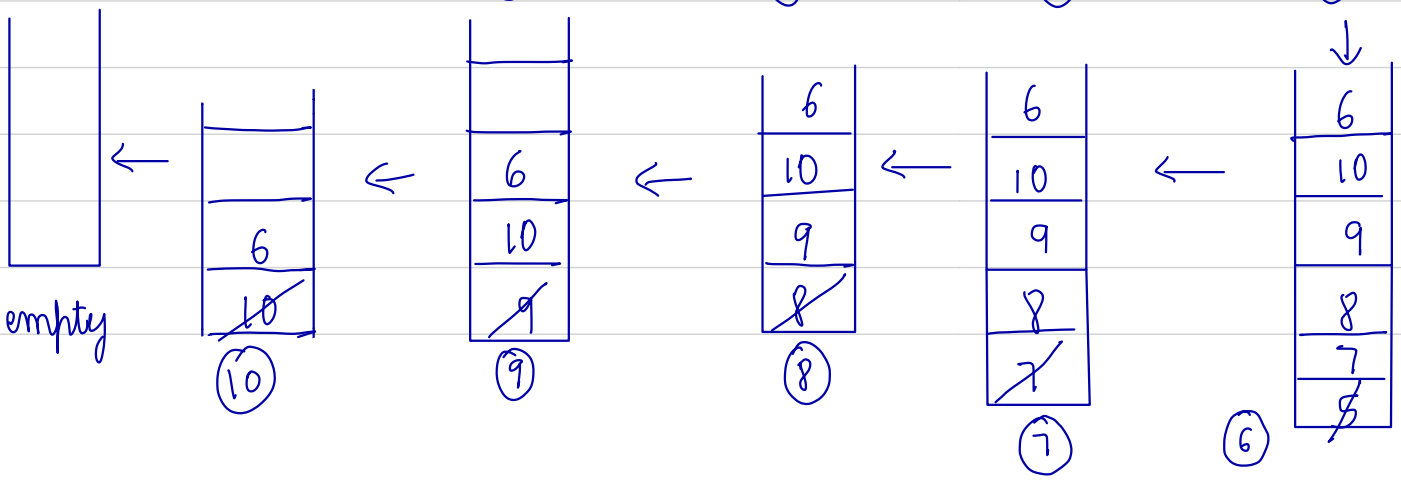
Eg 3:


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10
1
2
4
6
4
3
4
5
5

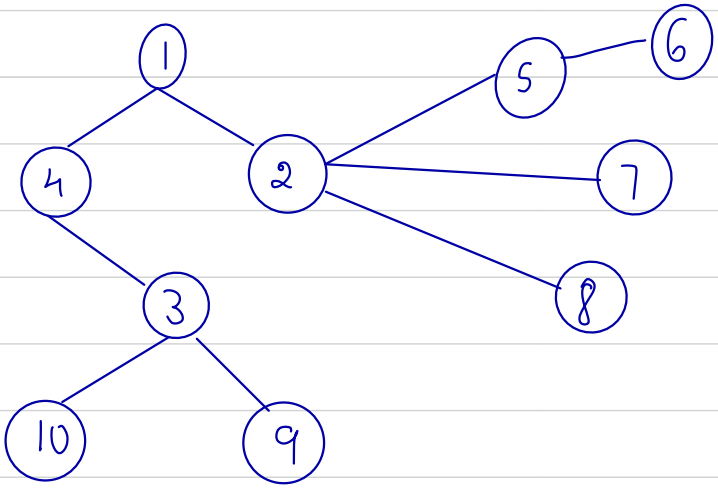
Queue:


1
2
4
x
3
2
4
8
7
5
3
2
10
9
8
7
5
3
①
②
③
④
⑤

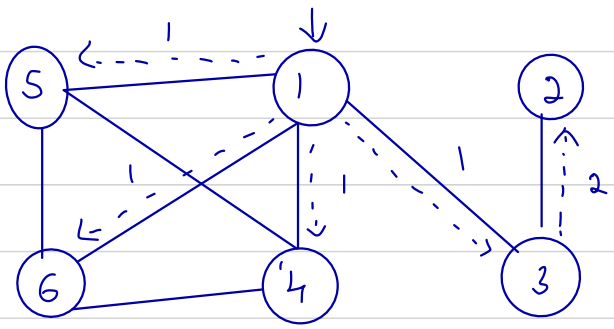

6
10
9
8
7
5
6
10
9
8
7
x
6
10
9
8
x
6
10
9
x
6
10
x
empty
⑥
⑦
⑧
⑨
⑩

1, 4, 2, 3, 5, 7, 8, 10, 9, 6

Spanning trees:



Eg:

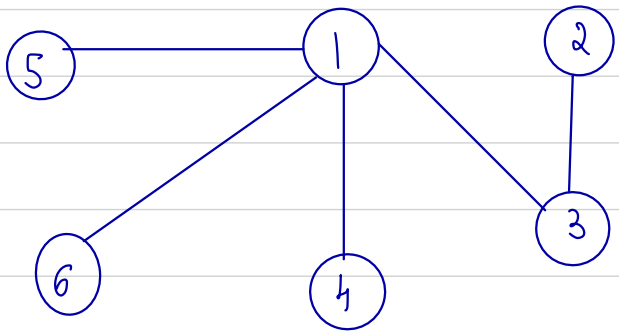


Queue:

~~1~~	~~3~~	4	5	6	2
1	1	1	1	1	2

Reached: 1, 3, 4, 5, 6, 2

Tree:



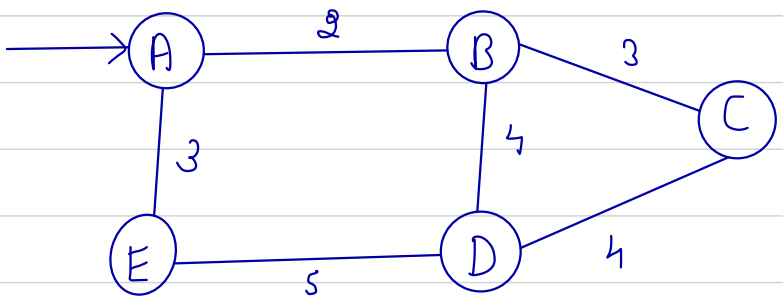
NOTE: Spanning tree satisfies following conditions:

1) It is a sub graph
2) It should have same number of vertices.
   i.e $|V'| = |V|$ ($V' \rightarrow$ vertex in tree)
3) Difference between number of edges & vertices is 1. i.e $|E'| = |V| - 1$ //

Greedy approach: To find the minimum spanning tree.

1) Prim's approach:

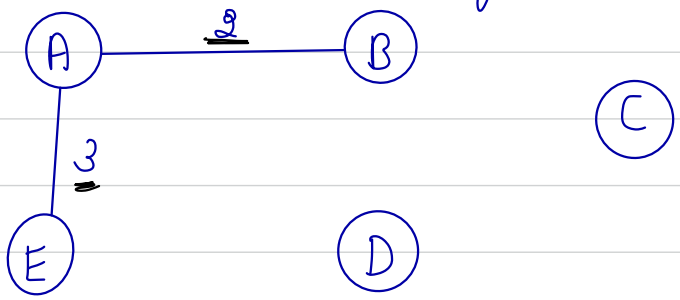
Eg:



Start from any random node.

1) Check the weights of connected edges (compare 3, 4, 3)

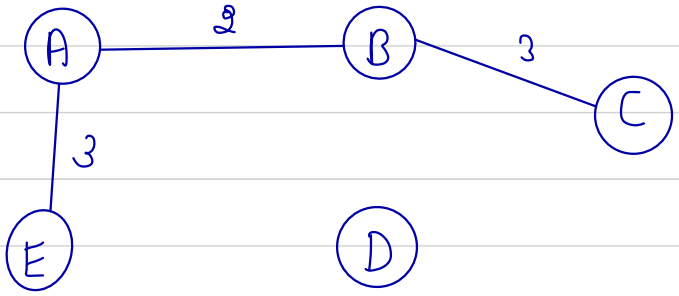
Step 1:



(If weights are same, select any)

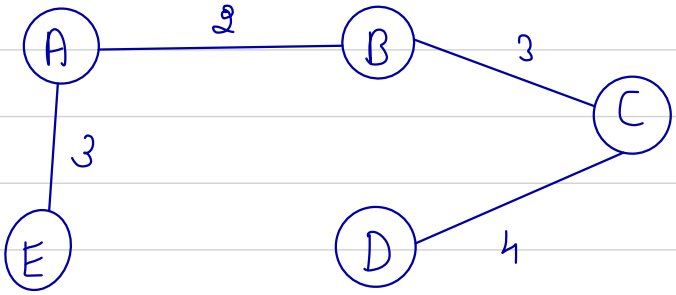
- keep checking weights of all adjacents nodes of the reached nodes

Step 2:



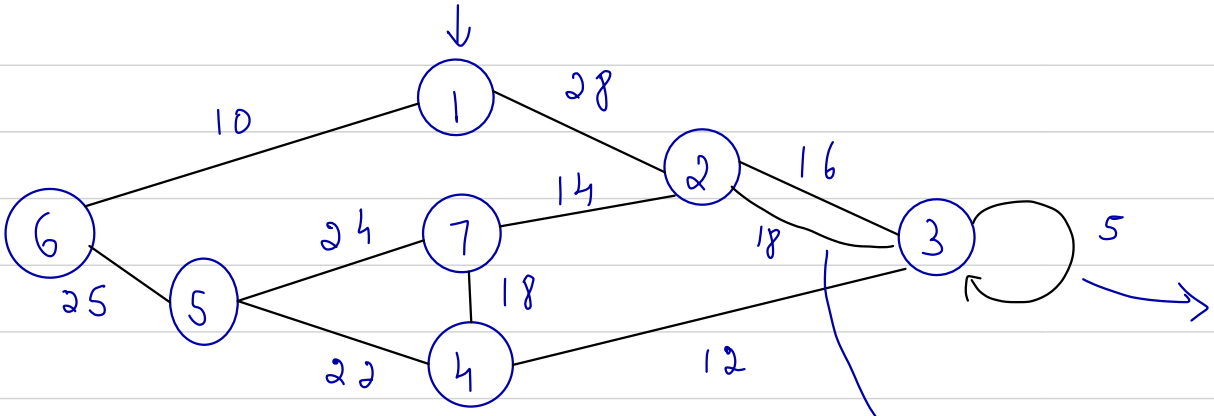
(compare 5, 4, 3)

Step 3:



(compare : 5, 4, 4)

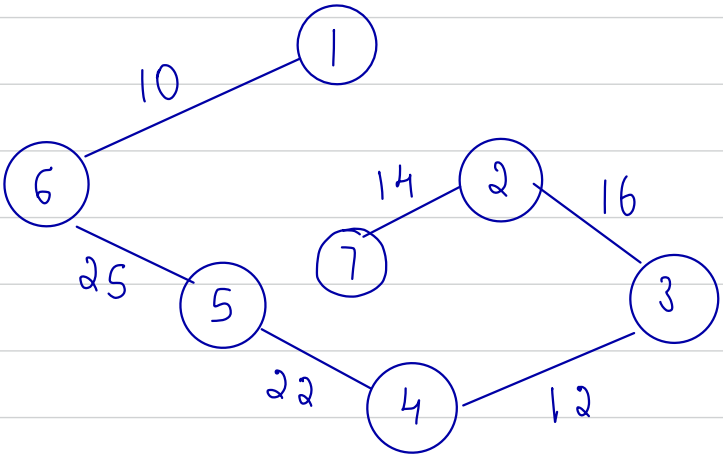
Eg2:



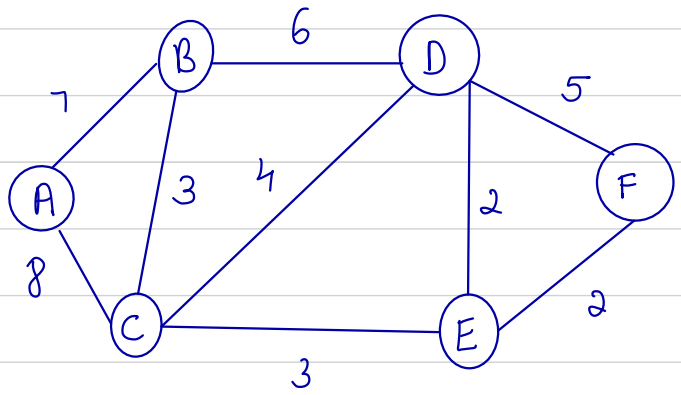
→ Don't look into loops (in Prim's)

→ Don't consider larger edge

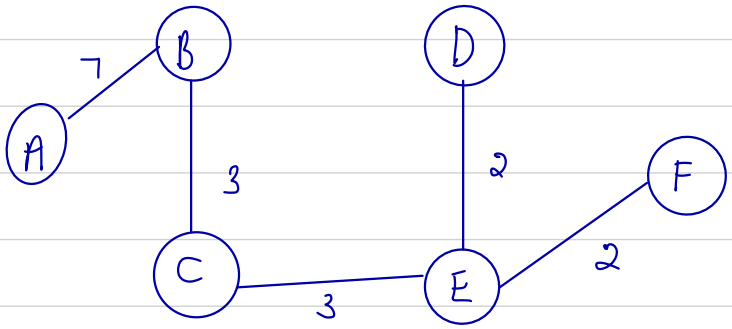
Ans



Eg3

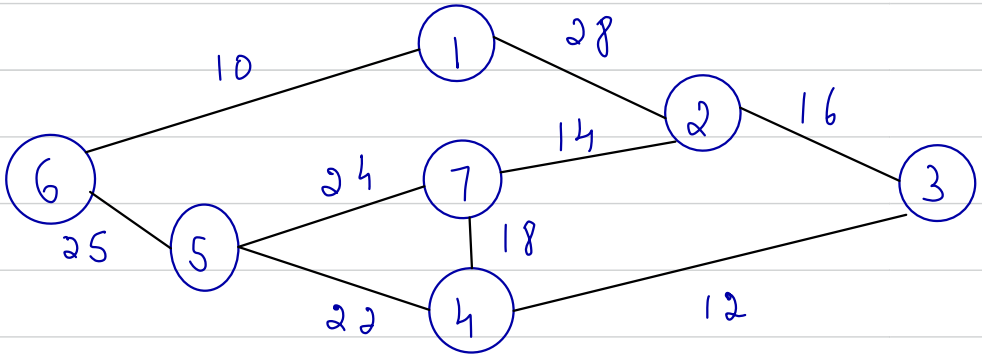


Ans



2> Kruskal's approach

Step 1: Write edges in ascending order of cost.



Edge: (1,6) (3,4) (2,7) (2,3) (4,7) (4,5) (5,7)

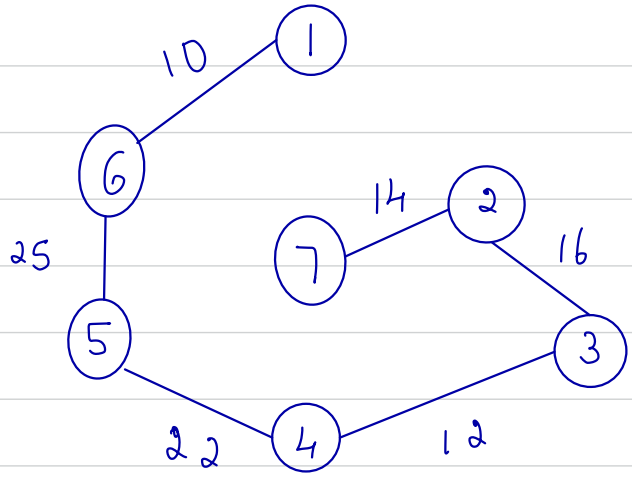
cost: 10 ✓ 12 ✓ 14 ✓ 16 ✓ 18 ✗ 22 ✓ 24 ✗

(5,6) (1,2)

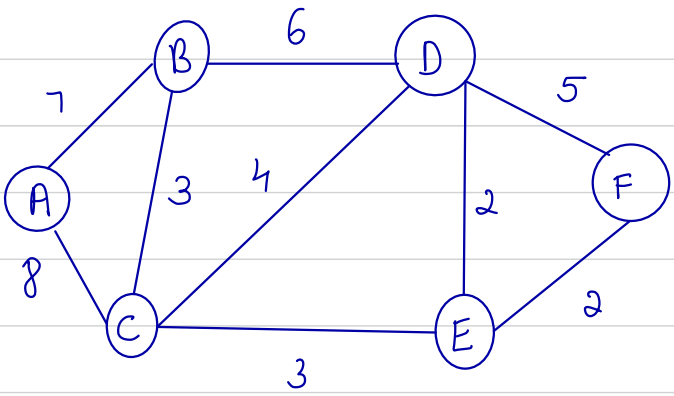
25 28

✓ ✗

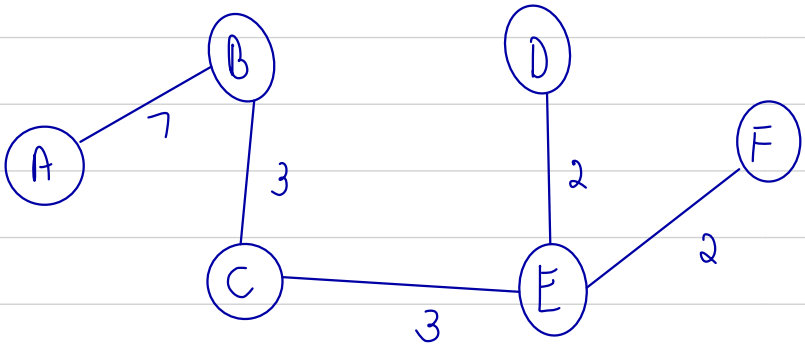
Step 2: Connect edges in ascending order and discard the connection if cycle is formed.



Eg 2:



Edge:	DE	EF	BC	CE	CD	DF	BD	AB	AC
cost:	2 ✓	2 ✓	3 ✓	3 ✓	4 ✗	5 ✗	6 ✗	7 ✓	8 ✗

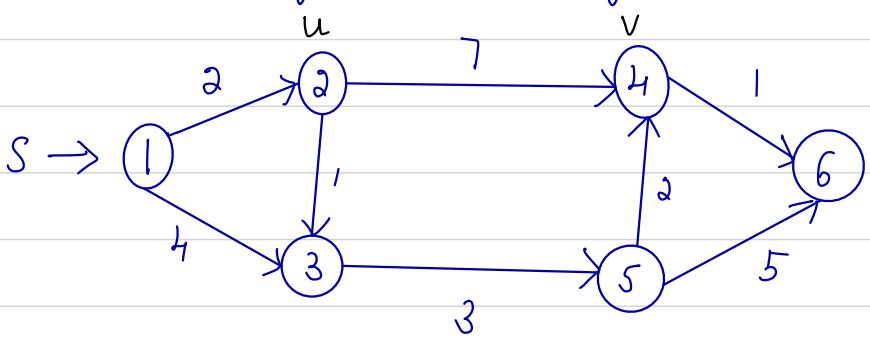


Shortest Path algorithm

1) Dijkstra's algorithm
2) Bellman - Ford algorithm

1) Dijkstra's algorithm: (Not applicable for graphs with Negative edges)

Eg 1:

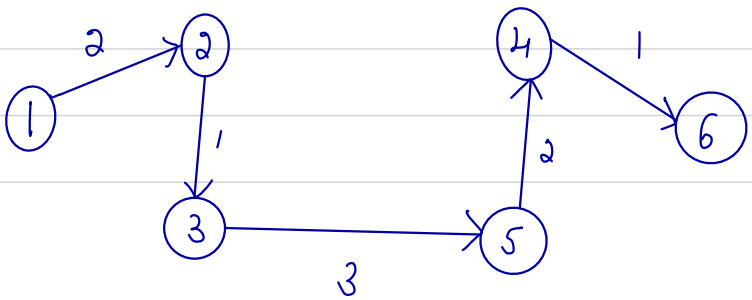


d → distance

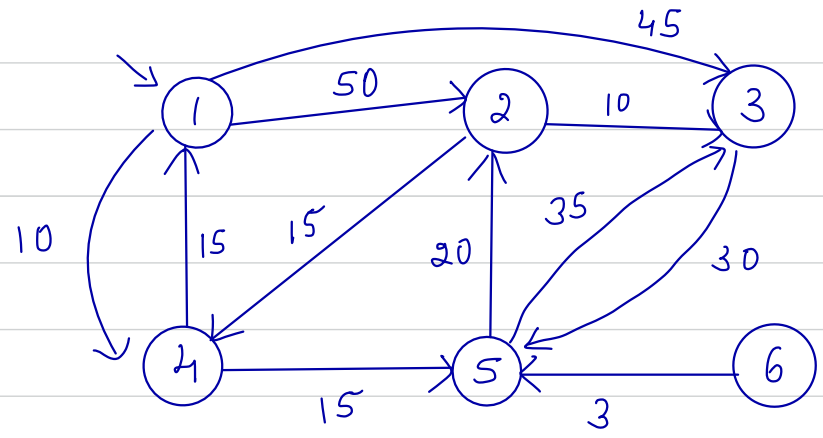
If $d[u] + cost(u \to v) < d[v]$
then $d[v] = d[u] + cost(u \to v)$

Selected Vertex	1	2	3	4	5	6
1	0	2	4	∞	∞	∞
2	0	2	3	9	∞	∞
3	0	2	3	9	6	∞
5	0	2	3	8	6	11
4	0	2	3	8	6	9

← Vertex



Eg 2:



Selected Vertex	1	2	3	4	5	6	← vertex
	0	∞	∞	∞	∞	∞	
1	0	50	45	10	∞	∞	
4	0	50	45	10	25	∞	
5	0	45	45	10	25	∞	
2	0	45	45	10	25	∞	
3	0	45	45	10	25	∞	

shortest path:

1 → 4 → 5 → 2 = 45

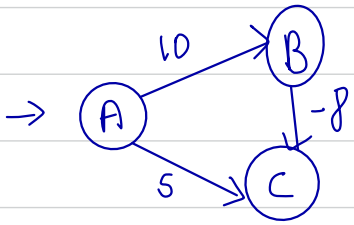
1 → 3 = 45

1 → 4 = 10

1 → 4 → 5 = 25

1 → 6 = ∞

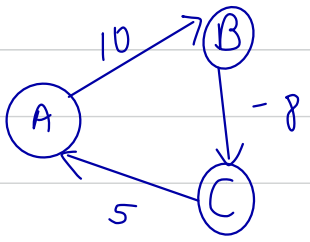
Eg 3:



Selected Vertex	A	B	C	vertex
	0	∞	∞	
A	0	10	5	
C	0	10	5	
B	0	10	5	

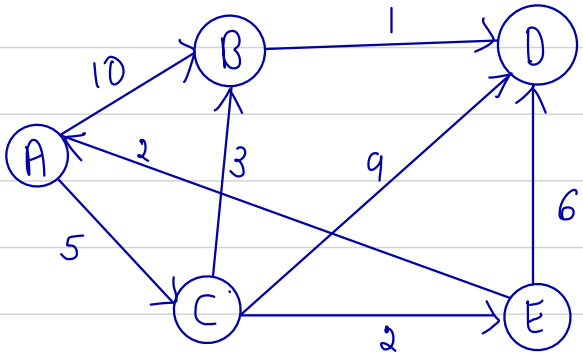
2) Bellman Ford algorithm:

- V-1 iterations are carried out, where V = no. of vertices
- And last iteration will be carried out to check whether there is an negative edged cycle. eg:



Time complexity (Dijkstra) < Time complexity (Bellman ford)

Eg:) Use Dijkstra's method:



Selected vertex	A	B	C	D	E
	0	∞	∞	∞	∞
A	0	10	5	∞	∞
C	0	8	5	14	7
E	0	8	5	13	7
B	0	8	5	9	7
D	0	8	5	9	7

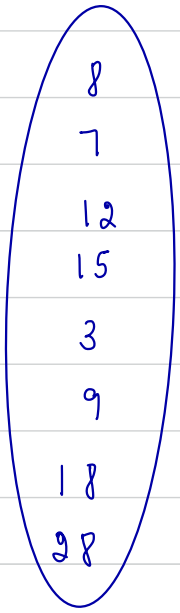
vertex

Hashing

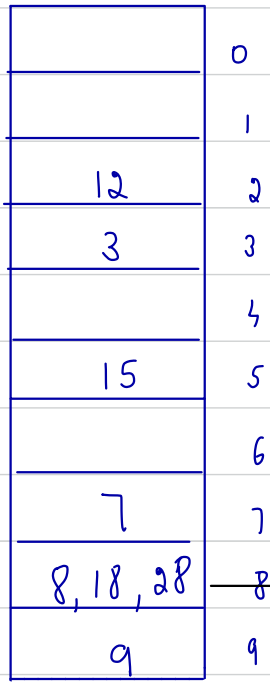
1> Open hashing → chaining
2> Closed hashing → linear probing
→ Quadratic probing

1> Open hashing

eg: Key space

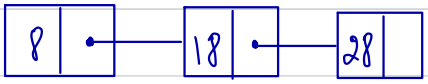


Hash table



size = 10

collision →



(chaining)

$h(x) = x \,\%\, size$

2> Closed hashing

i> linear probing; $h(x) = [h(x) + f(i)] \,\%\, size$

Key space

8
7
12
15
3
9
1
28
22

Hash table

9	0
1	1
12	2
3	3
22	4
15	5
	6
7	7
8	8
28	9

2) Quadrating probing: $h(x) = [h(x) + f[i]^2] \% size$

Eg:

Key space

32
42
52
19
26
72

Hash table

quadratic

	0
72	1
32	2
42	3
	4
	5
52	6
26	7
	8
19	9

Linear

	0
	1
32	2
42	3
52	4
72	5
26	6
	7
	8
19	9