


Trees, Heaps and Graphs



By 106119018, 106119094, 106119140

Trees



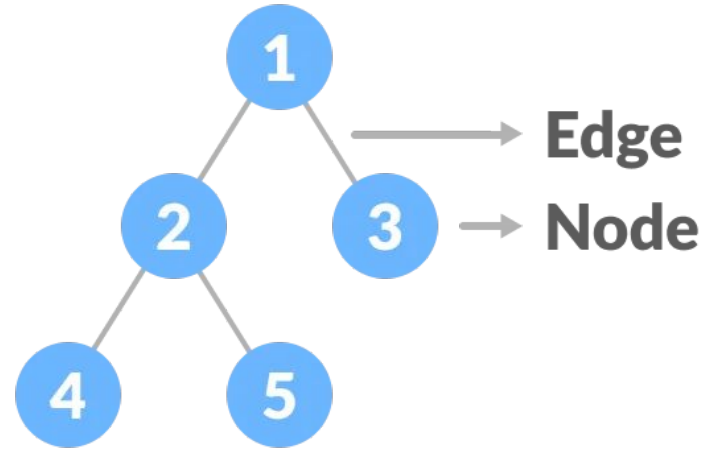
Terminologies

- **Node**

- A node is an entity that contains a key/value/pointer pointing to it
- Nodes are of two types:
- Leaf/external nodes, which are nodes that do not have any child nodes and are the last node in the respective branch of the tree
- Internal nodes, which are nodes that have at least one child node

- **Edge**

- An edge is a link that connects two nodes (parent node to child node)



Terminologies

- **Root**

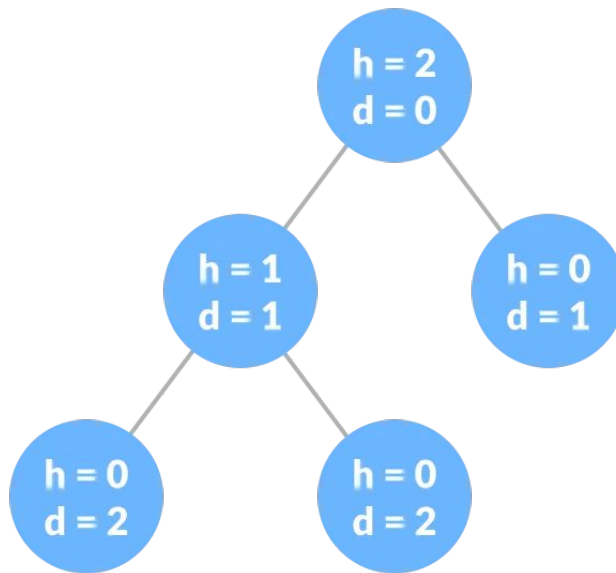
- Root is a node that is the topmost node of the tree, from which the tree starts and originates (i.e., it is the parent most node of the tree)

- **Height of a node**

- Height of a node is the number of from the node to the deepest leaf, i.e., longest path from the node to any leaf node

- **Depth of a node**

- Depth of a node is the number of edges from the root node to the current node



Terminologies

- **Height of a tree**

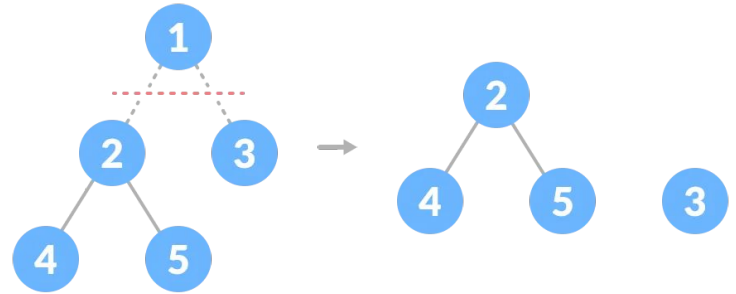
- Height of the tree is the maximum number of edges between the root node to any leaf node

- **Degree of a node**

- It is the total number of edges connected connecting to the node's children (i.e., the number of children)

- **Forest**

- A forest is a collection of disjoint trees



Tree traversal

Traversing a tree means to traverse through every node of a tree in a particular order

There are 4 common ways of tree traversal each with a different order of traversal

- Pre-order traversal
- Inorder traversal
- Post-order traversal
- Level order traversal

These methods are implemented with the help of different data structures like stacks, queues and arrays

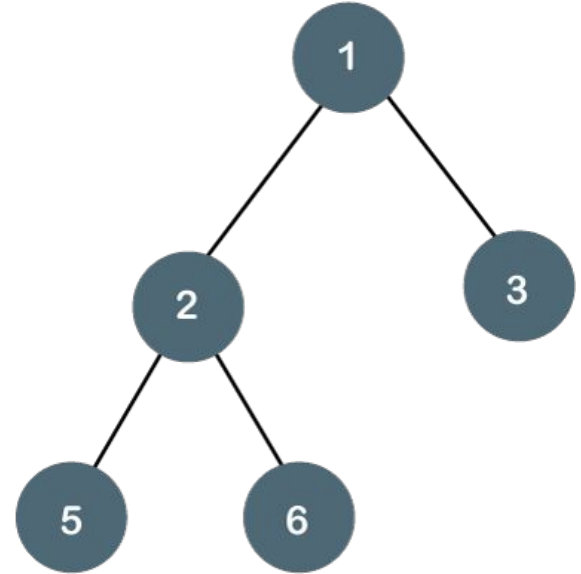
Pre-order tree traversal

In pre-order traversal, the parent node is visited first, then the left child and then the right child

Order: P L R

If we were to print every node in the following tree based on pre-order traversal, the order will be:

1 -> 2 -> 5 -> 6 -> 3



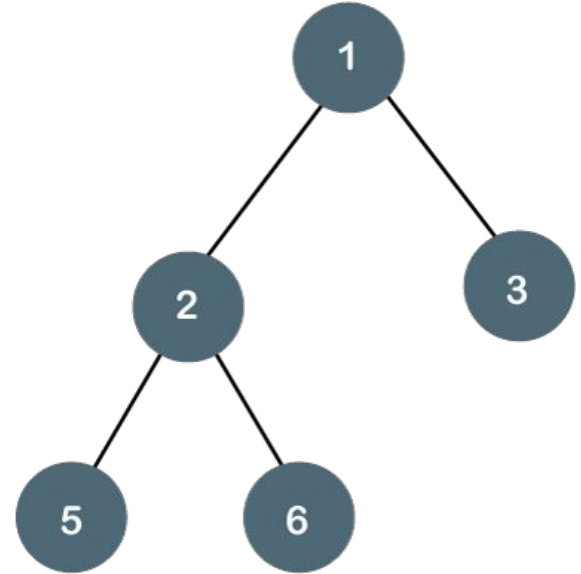
Inorder tree traversal

In inorder traversal, the left child is visited first, then the parent node and finally the right child

Order: L P R

If we were to print every node in the following tree based on inorder traversal, the order will be:

5 -> 2 -> 6 -> 1 -> 3



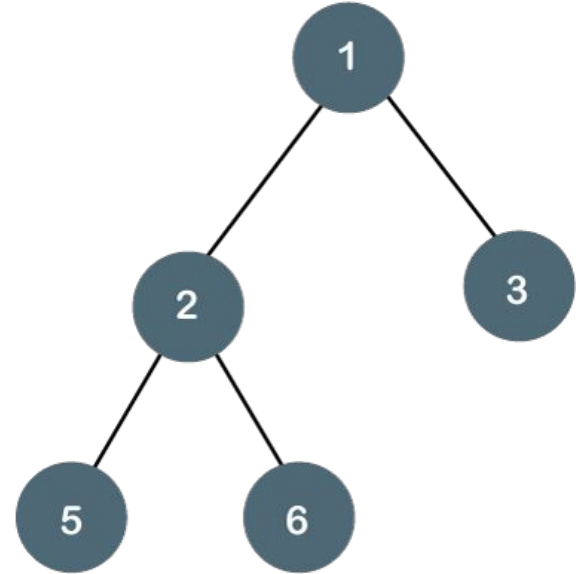
Post-order tree traversal

In post-order traversal, the left child is visited first, then the right child and then at the end the parent node

Order: L R P

If we were to print every node in the following tree based on post-order traversal, the order will be:

5 -> 6 -> 2 -> 3 -> 1



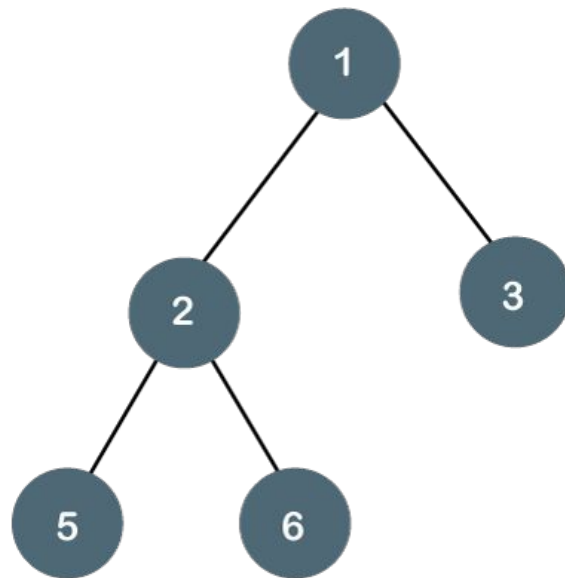
Level-order tree traversal

In level-order traversal, we traverse the node based on the depth

Nodes in the same depth are visited together, with the leftmost node being visited first

If we were to print every node in the following tree based on level-order traversal, the order will be:

1 -> 2 -> 3 -> 4 -> 5 -> 6



Different types of trees

Here are the following types of trees which will be explained more in the later slides

- Binary trees
- Binary search trees
- Self-balancing binary trees

Ideally a tree can have any number of children but we will be focusing on trees with nodes having at most 2 children

Binary tree

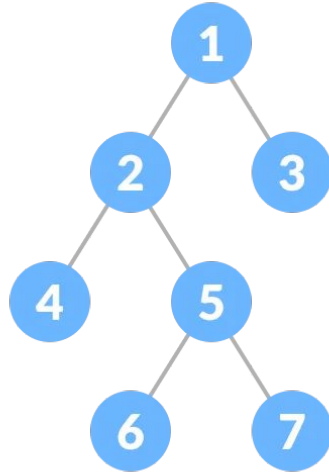
These trees are the type of trees that have at max 2 children

There are different types of binary trees

- Full binary tree
- Perfect binary tree
- Complete binary tree
- Skewed binary tree
- Balanced binary tree

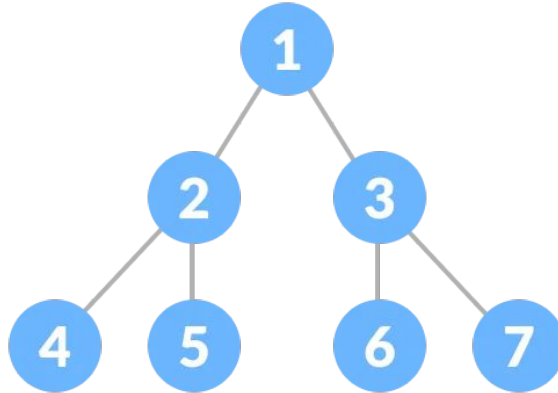
Full binary tree

Full binary trees are trees in which every node has either 0 or 2 child nodes



Perfect binary tree

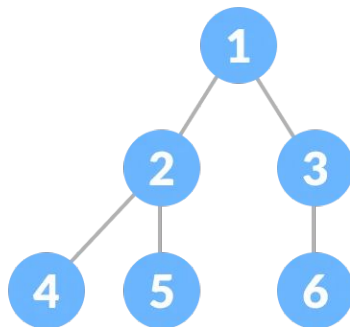
It is a type of binary tree where every internal node has exactly 2 child nodes and all the leaf nodes are at the same level/depth



Complete binary tree

A complete binary tree is one which:

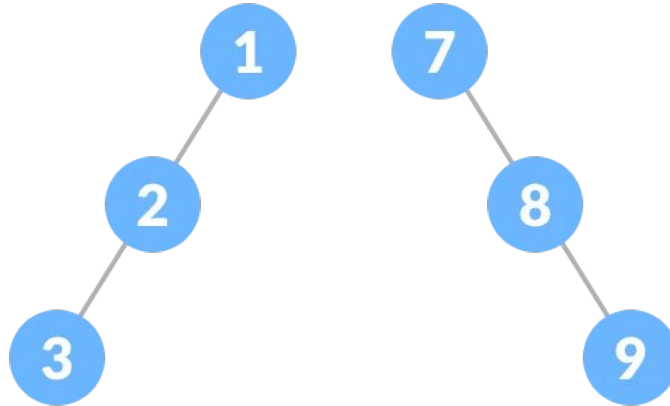
- Every level apart from the leaf node's level must be filled
- All leaf nodes must be filled from the left
- Last leaf node may not have a right sibling (it need not be full binary tree)



Skewed binary tree

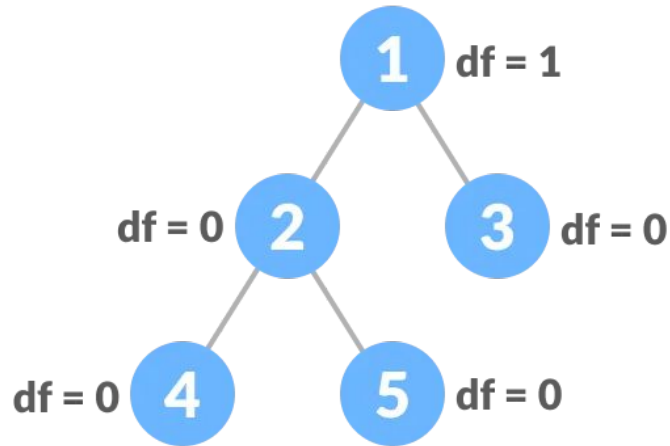
A skewed binary tree is one in which every node of tree has only left child nodes or right child nodes

There can be of two types, left skewed and right skewed



Balanced binary tree

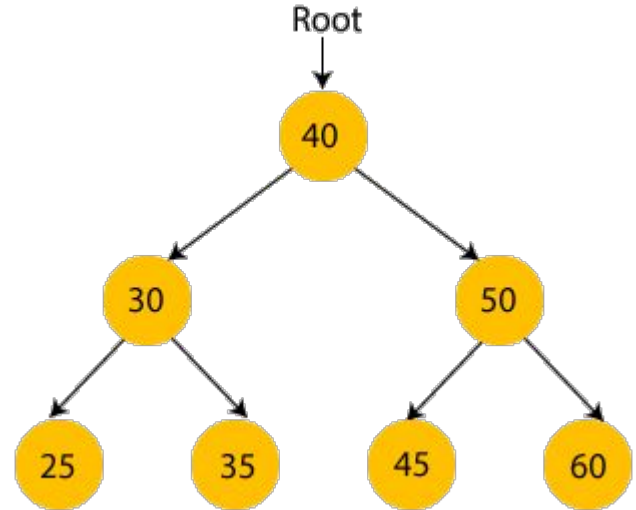
A balanced binary tree is one in which every node's left child (left subtree) and right child (right subtree) has a height difference of not more than 1



Binary search tree

Binary search tree is a specific instance of a binary tree which maintains the tree in a particular order, such that the nodes on the left subtree contain a value lesser than the parent node's value and the right subtree contains a value more than the parent node's value.

An inorder traversal of a BST always prints the values of a tree in ascending order



BST - search operation

Search operation is such that, given a value **X**:

```
if (X == node.value) return true;  
else if (X < node.value) searchInLeftSubtree();  
else searchInRightSubtree();
```

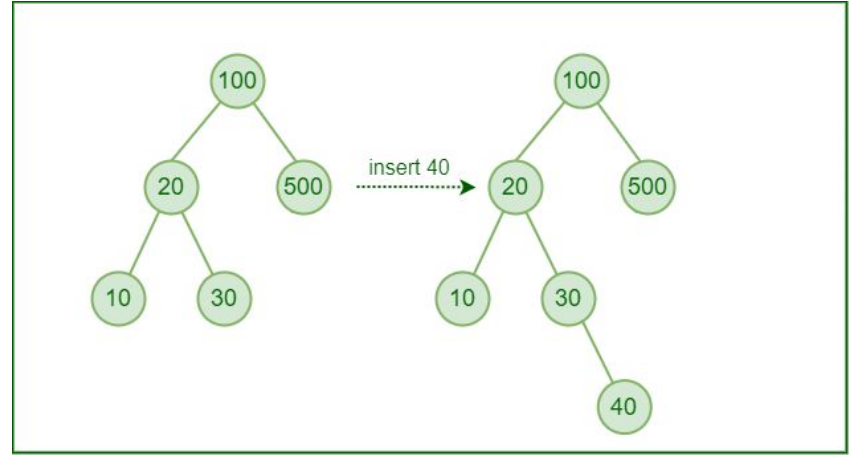
On average, this takes a $O(\log(n))$ time where n is the number of nodes

At worst case it can still take $O(n)$ time if the BST is skewed

BST - insertion

Given a node containing value **X**, we have to insert it in a BST tree such that we maintain the property of a binary search tree

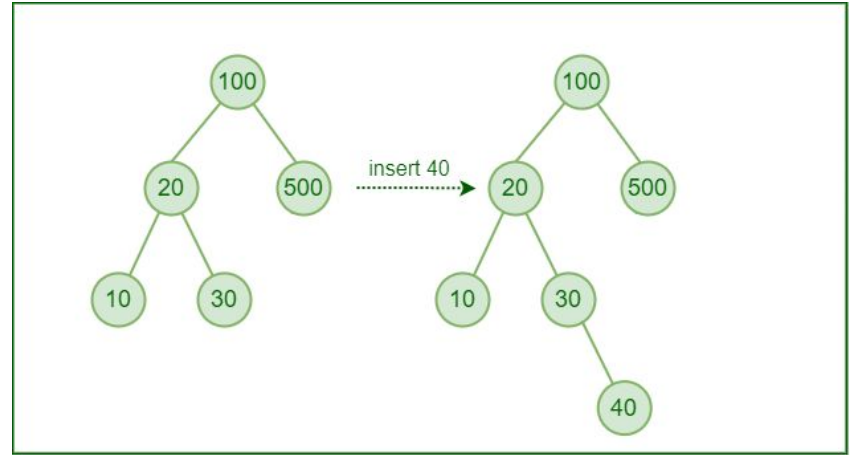
While inserting **X**, we compare the value with the current node's value, if it is more, we move it to the right subtree else move it to the left subtree if it is lesser



BST - insertion

We will finally come to a leaf node (let's say **A**), where we once again compare the value of the node with **A**. We insert the new node as a left child to **A** if the value of our node is lesser than **A**'s and right if it's value is more than **A**'s.

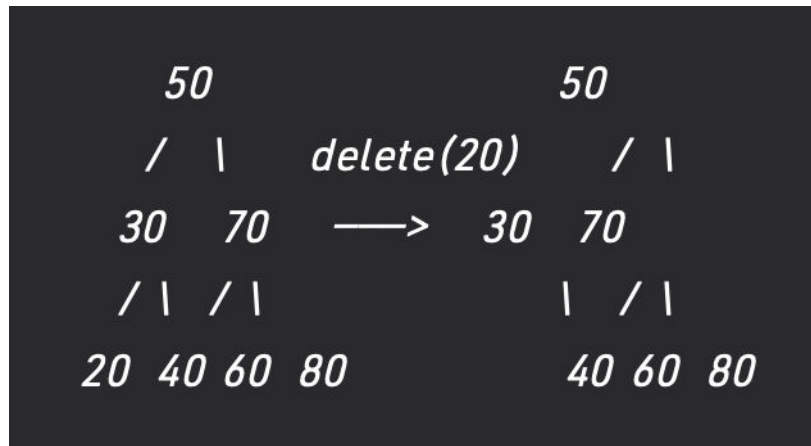
This on average takes $O(\log n)$ and at worst case can take $O(n)$ if the BST is skewed.



BST - deletion

Deletion of BST is a two step operation, one is performing the search operation to find the node and the other is the actual deletion of the node

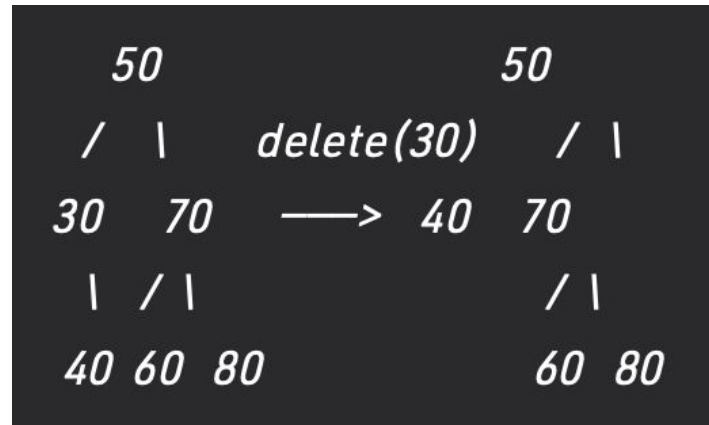
- Case 1: node is a leaf node (no. of children = 0): in this case, deletion is simple, the leaf node simply gets deleted



BST - deletion

- Case 2: no. of children of the node = 1

In this case the child node replaces the current node as a child to the current node's parent

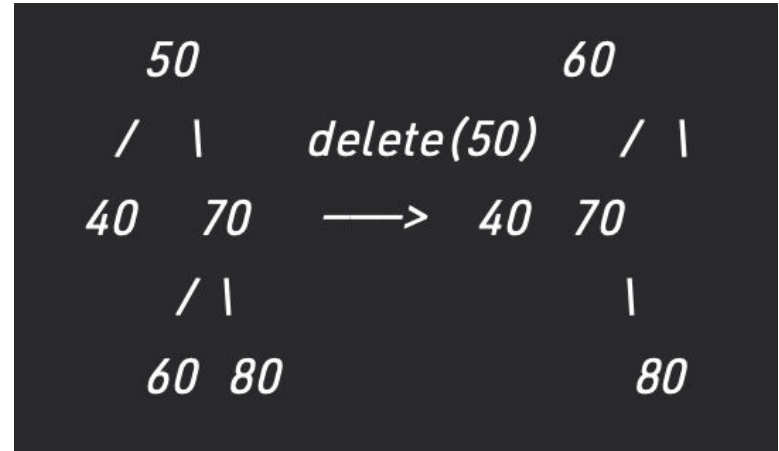


BST - deletion

- Case 3: no. of children of the node = 2

In this case we take the current node's inorder successor and copy its contents to the node to be deleted, and delete the selected node

On average, this operation takes $O(\log n)$ but on the worst case it can take $O(n)$ if the tree is skewed

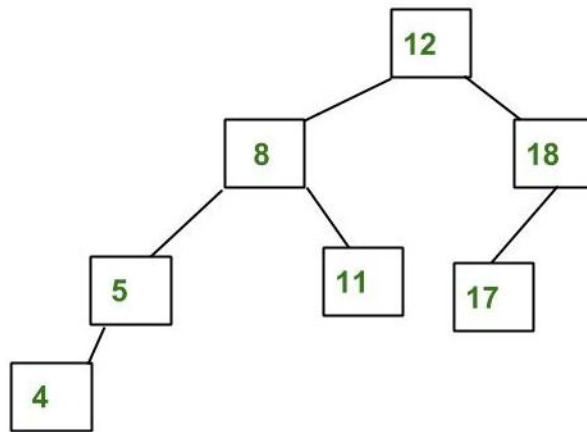


Self-balancing trees (AVL)

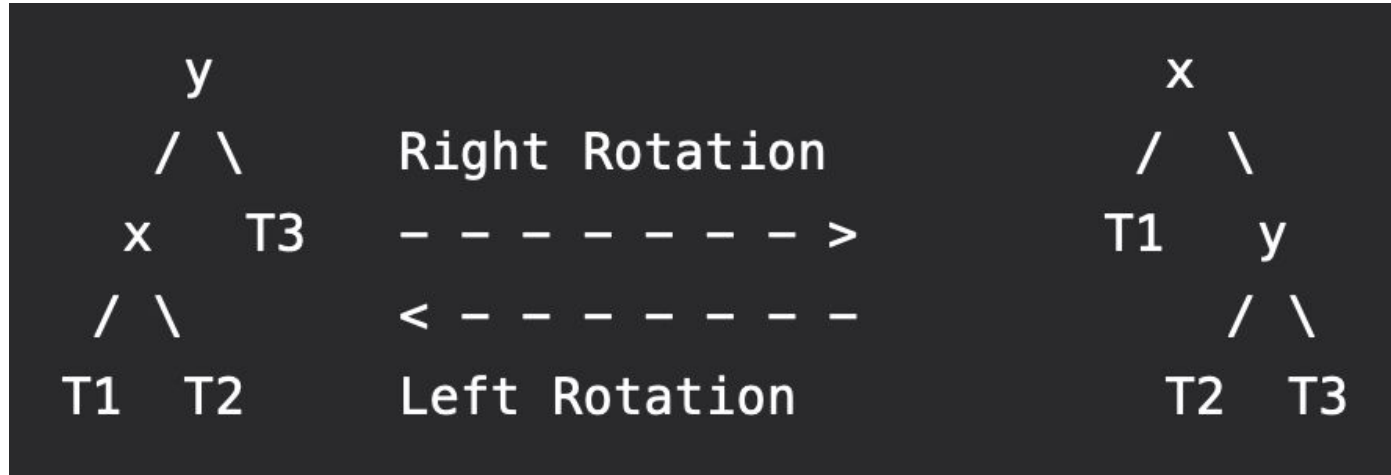
As seen before, operations on a simple BST can take $O(n)$, rendering the main property of a BST useless

We will now look at self balancing trees and focus on AVL tree

Self balancing trees are trees in which, on insertion and deletion, the tree will always remain a balanced binary search tree



Rotation in AVL trees



Here we see that the property of a BST is not violated

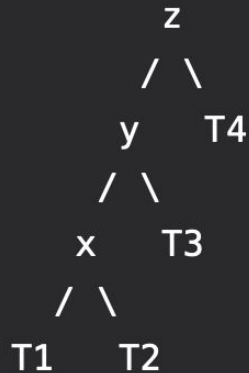
AVL - insertion

- First we perform insertion like BST, let this be **w**
- We then go up from **w** and find the first unbalanced node **z**
- Let **y** be the child of **z** that comes in the way till **y**
- We arrive at 4 cases to rebalance the tree

LL case

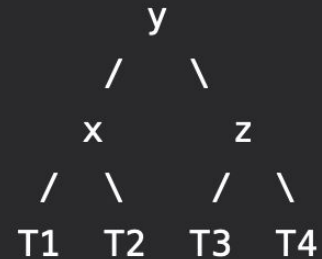
y is the left subchild of **z** and **x** is the left subchild of **y**

T1, T2, T3 and T4 are subtrees.



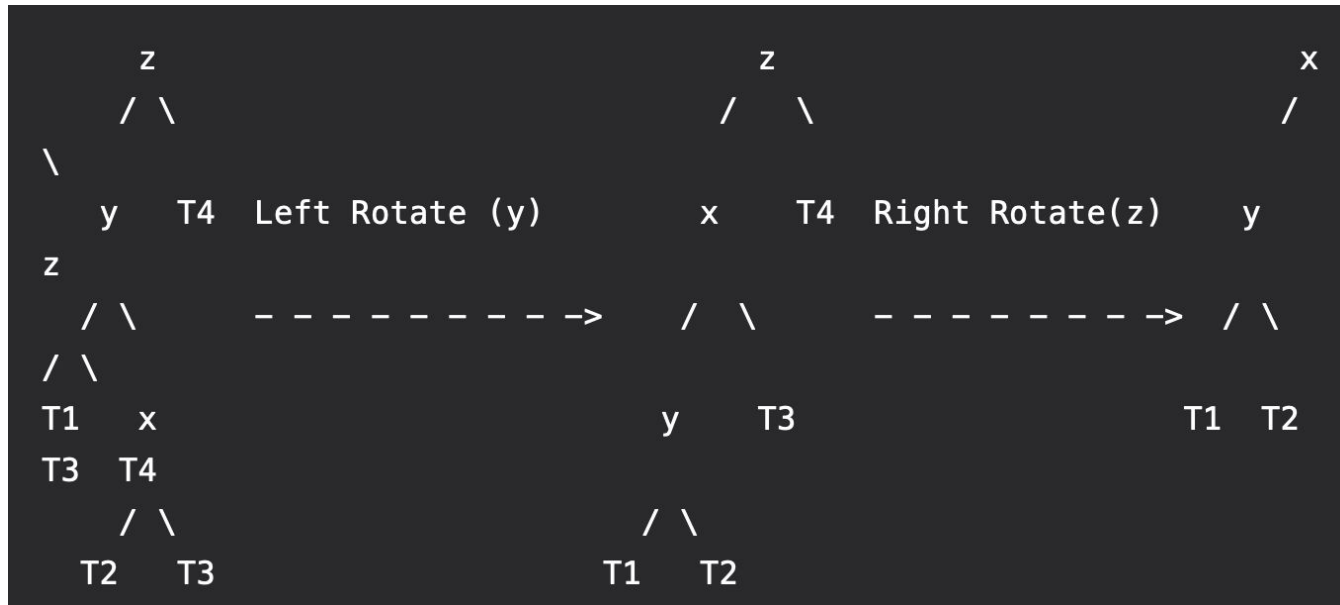
Right Rotate (z)

----->



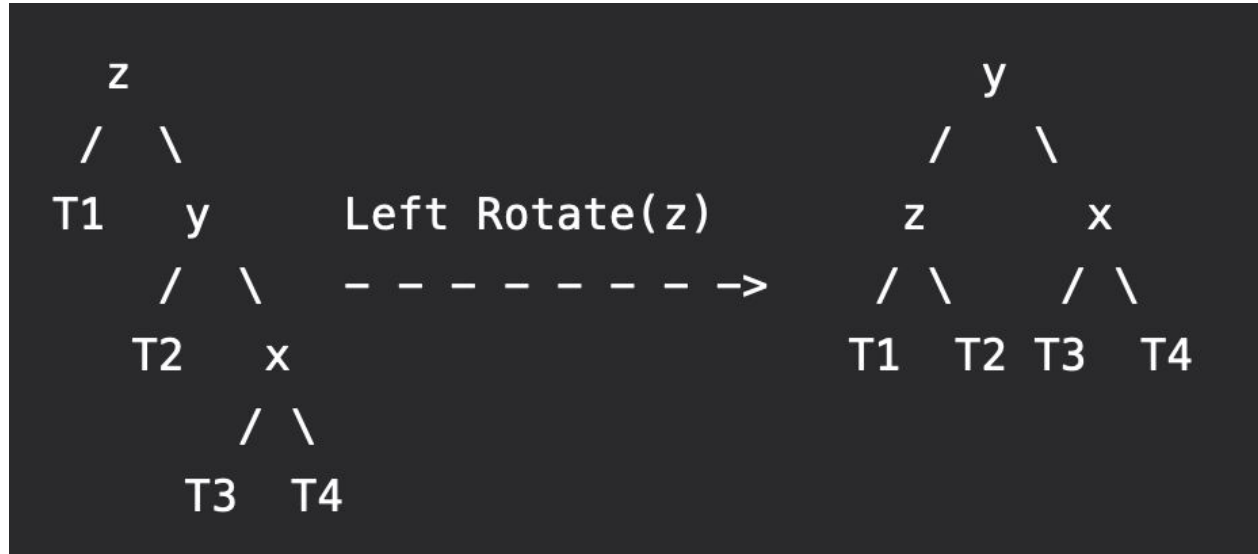
LR case

y is the left subchild of **z** and **x** is the right subchild of **y**



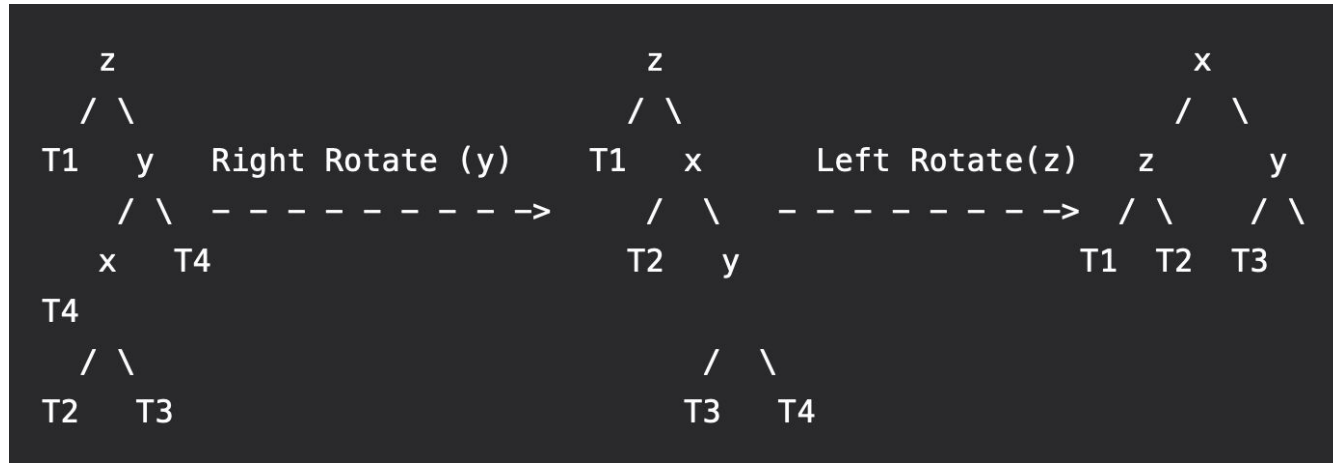
RR case

y is the right subchild of **z** and **x** is the right subchild of **y**



RL case

y is the right subchild of **z** and **x** is the left subchild of **y**



AVL - deletion

- Likewise, we perform normal BST deletion operation first
- We then try to find the first unbalanced node by getting balancing factor (height of left subtree - height of right subtree)
- If balancing factor > 1 , its a LL or LR case
 - If balancing factor of left subtree of unbalanced node is ≥ 0 , then its a LL case, or else it is LR
- If balancing factor < -1 , its a RR or RL case
 - If balancing factor of right subtree of unbalanced node is ≤ 0 , then its a RR case, else it is RL

AVL - search

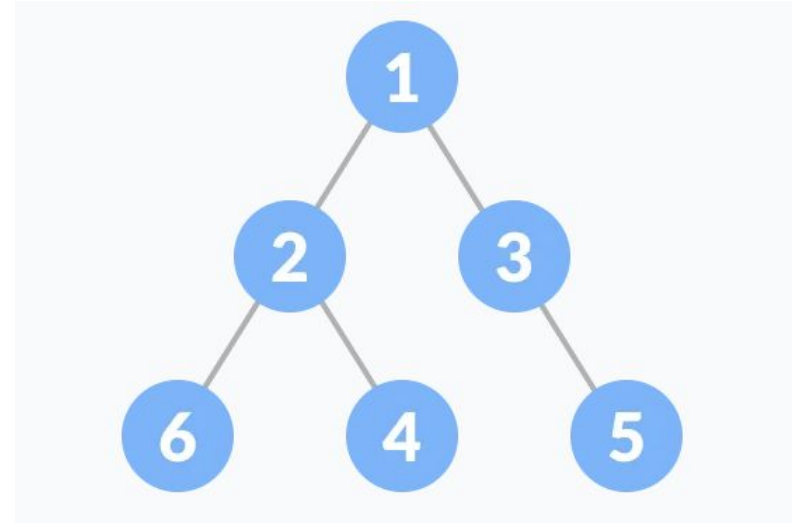
The search operation for an AVL tree is similar to a BST

The insertion, deletion and search operation in an AVL tree will always have a worst case time complexity of $O(\log n)$

Gate question

Which of the following matches the given tree?

- a) Not a full BT, not a complete BT
- b) Full BT, not a complete BT
- c) Not a full BT, is a complete BT
- d) Is a full BT and complete Bt



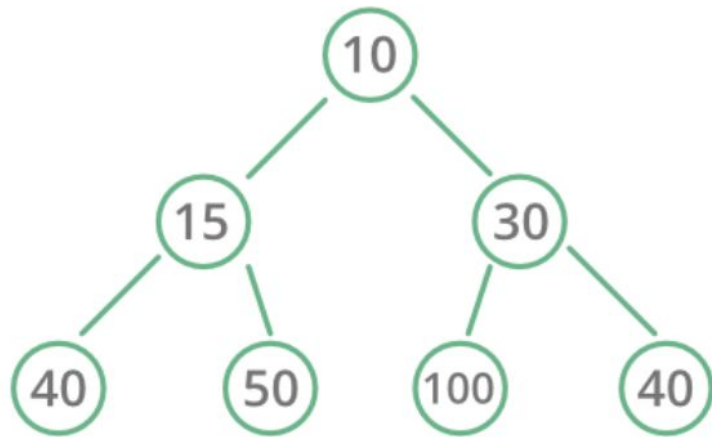
Binary Heaps



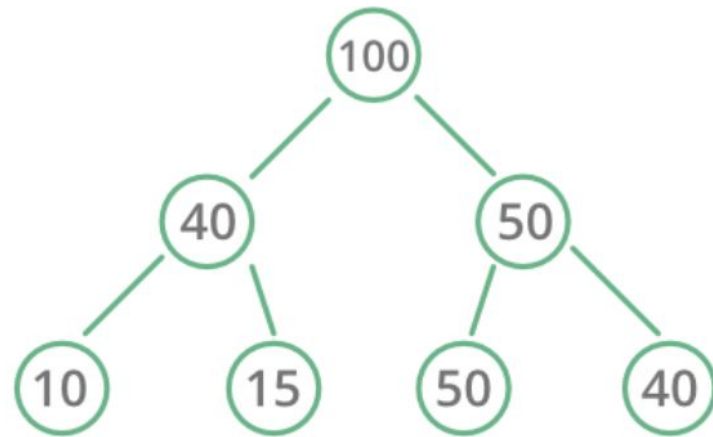
WHAT IS A BINARY HEAP?

- A binary heap is a **complete binary tree** which satisfies the heap ordering property.
- A complete binary tree is **a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.**
- This heap ordering property is either a max or min-heap property
- Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has $O(\log N)$ height.

Heap Data Structure



Min Heap

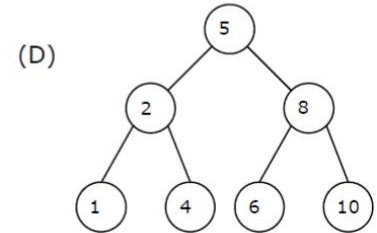
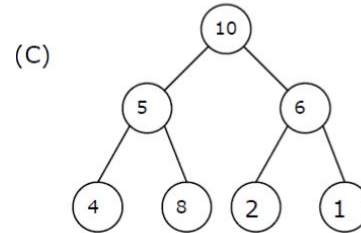
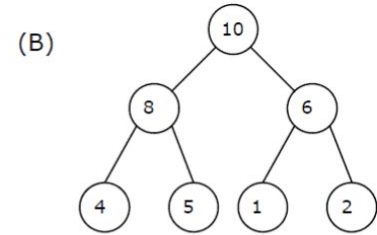
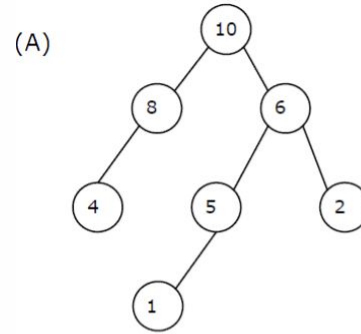


Max Heap

GATE 2011 QUESTION

Q. A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?

- (A) A
- (B) B
- (C) C
- (D) D

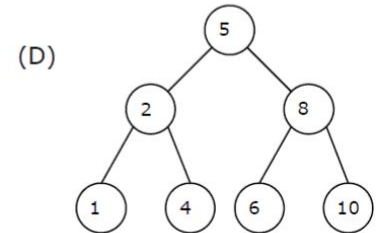
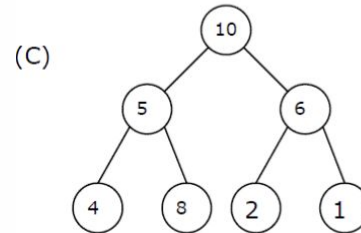
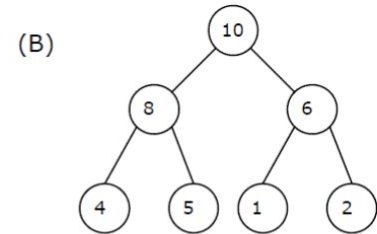
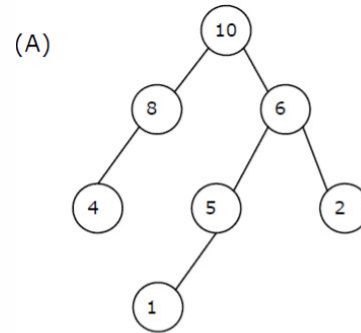


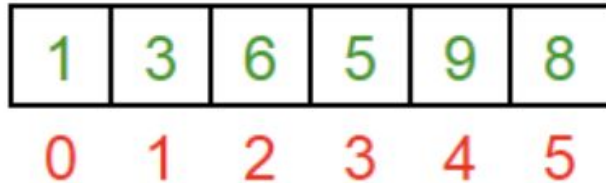
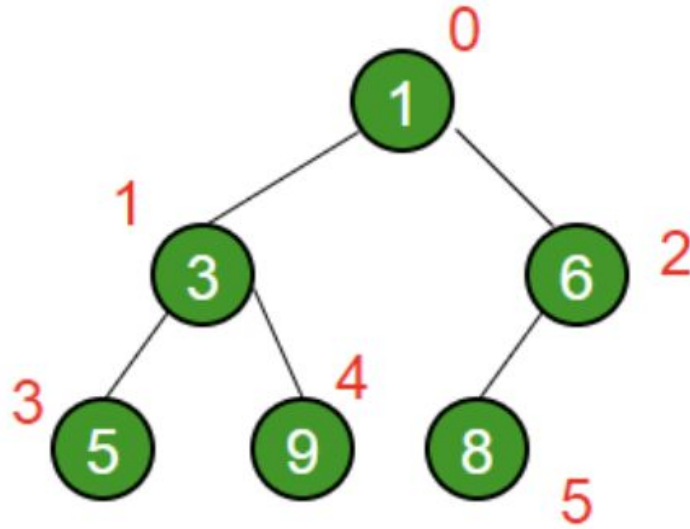
GATE 2011 QUESTION

Q. A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?

- (A) A
- (B) B
- (C) C
- (D) D

Answer (B)





REPRESENTING A BINARY HEAP

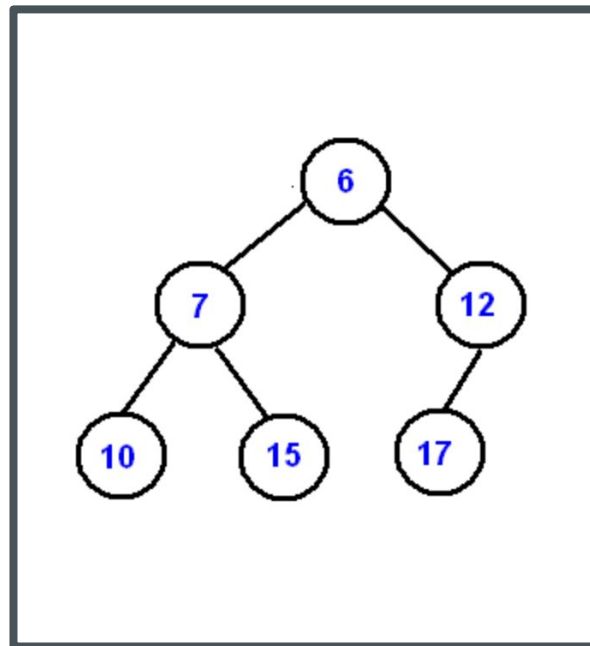
- It is represented as an array by uniquely storing its level order traversal
- Relation between nodes:
 - Left child : $(2*i)+1$
 - Right child : $(2*i)+2$
 - Parent : $(i-1)/2$

WEAKLY ORDERED

- We know how a binary search tree is developed – with lesser keys to the left; greater keys to the right as we descend
- But for nodes in a heap, **we don't have this strong ordering** - and this can cause us some difficulty.
- Hence No Convenient Search Mechanism
- Makes it less efficient for deletions

WEAKLY ORDERED

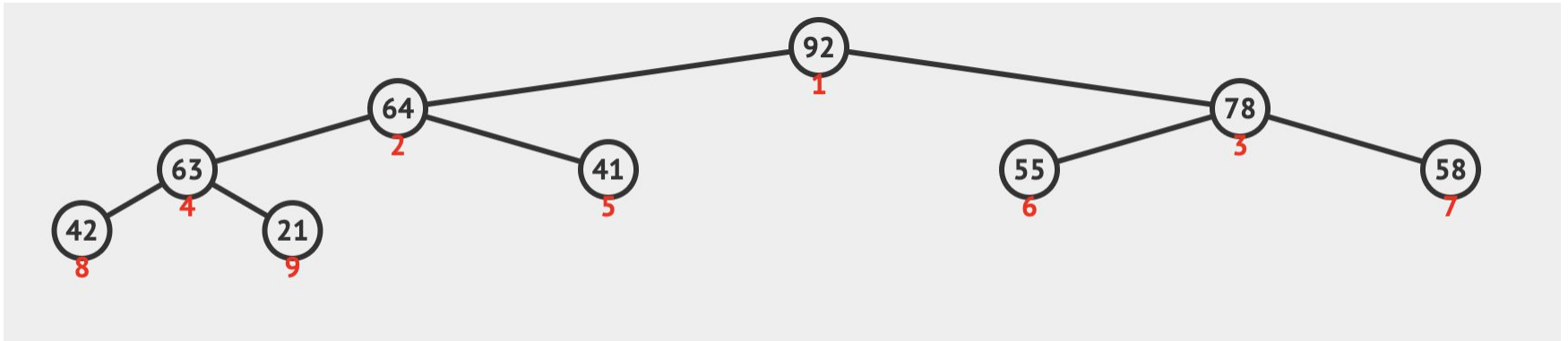
- ▶ As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.
- ▶ Sufficient Ordering: Yet, there is 'sufficient ordering' to allow
 - **quick removal** (yes, a delete) of the maximum node and
 - **fast insertion** of new nodes.
- These are the operations that are mostly used when dealing with this data structure



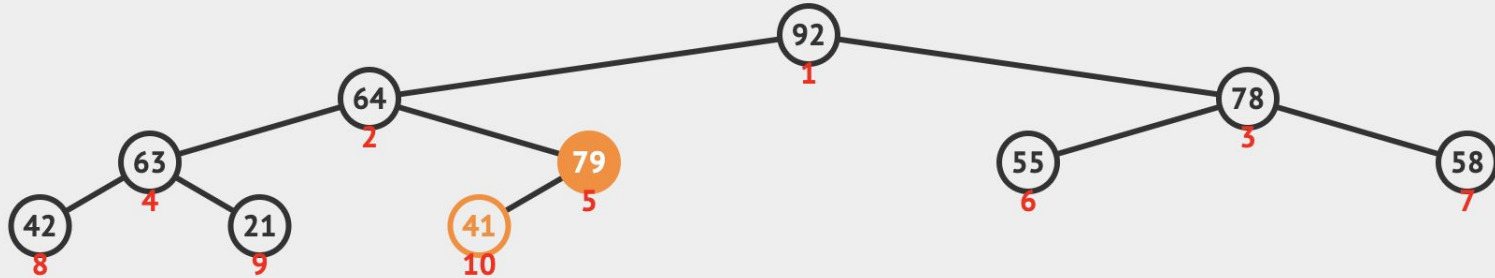
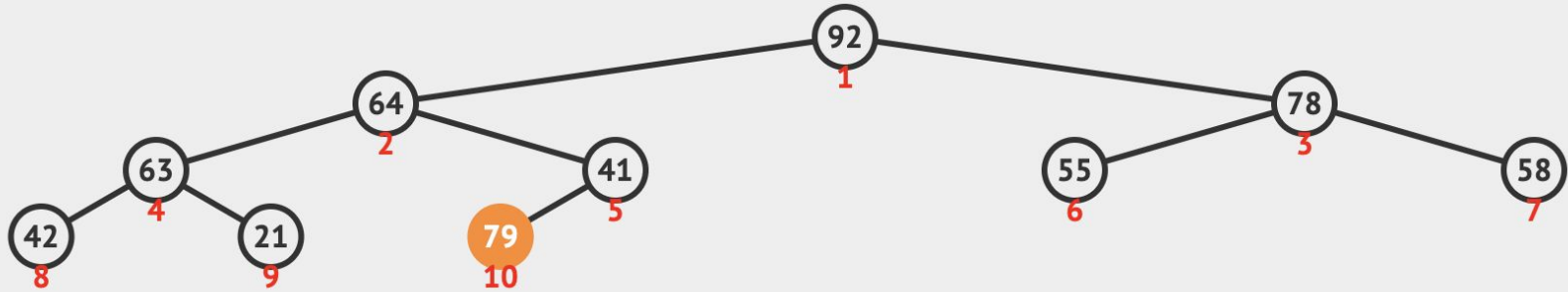
INSERT OPERATION

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify** this newly inserted element following a bottom-up approach.
- This process is called "percolation up".
- The comparison is repeated until the parent satisfies the heap property with percolating element.

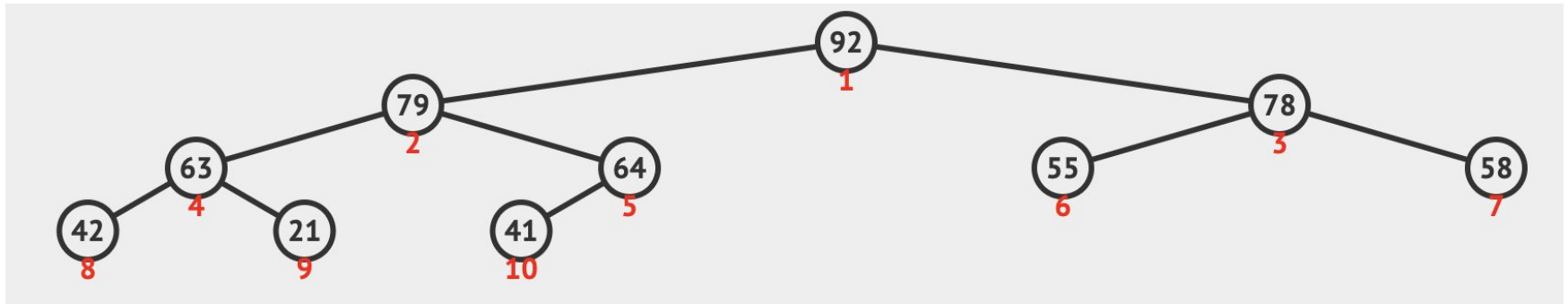
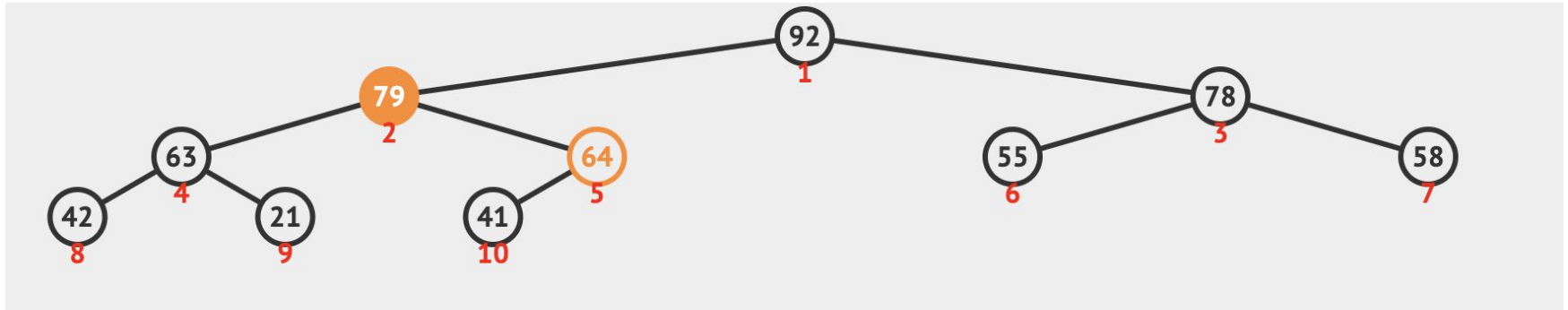
INSERT OPERATION – INITIAL HEAP



INSERT OPERATION – INSERTING 79 TO LAST POSITION



INSERT OPERATION – INSERTING 79 TO LAST POSITION



INSERT OPERATION CODE – $O(\log N)$ TIME

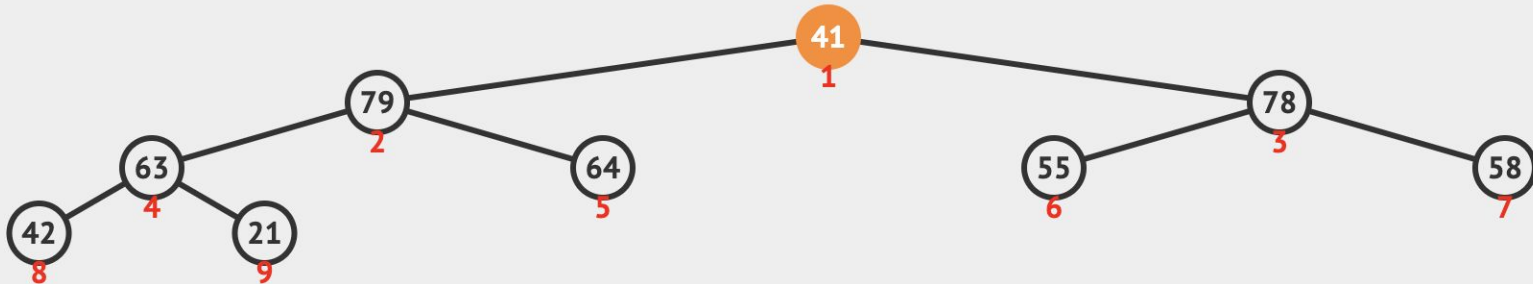
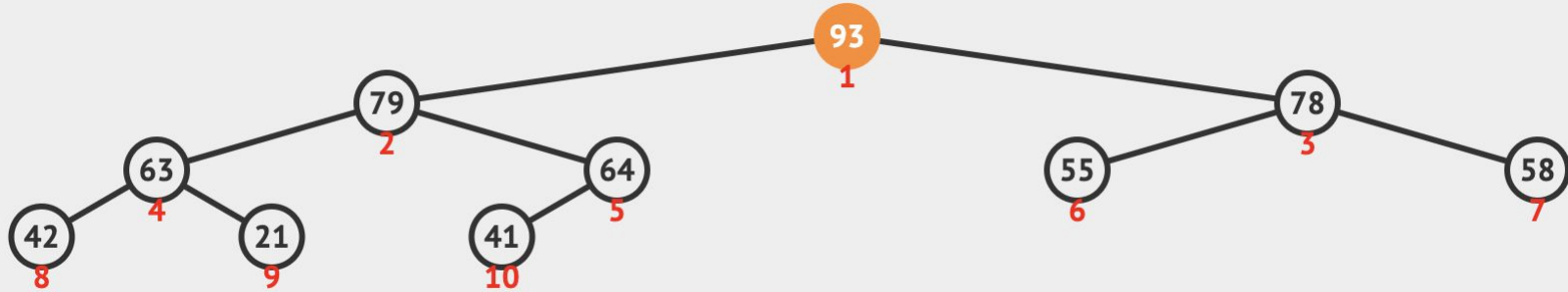
```
1 void insertNode(int arr[], int& n, int Key)
2 {
3     // Increase the size of Heap by 1
4     n = n + 1;
5
6     // Insert the element at end of Heap
7     arr[n - 1] = Key;
8
9     // Heapify the new node following a
10    // Bottom-up approach
11    heapify(arr, n, n - 1);
12 }
```

```
14 void heapify(int arr[], int n, int i)
15 {
16     // Find parent
17     int parent = (i - 1) / 2;
18
19     if (arr[parent] > 0) {
20         // For Max-Heap
21         // If current node is greater than its parent
22         // Swap both of them and call heapify again
23         // for the parent
24         if (arr[i] > arr[parent]) {
25             swap(arr[i], arr[parent]);
26
27             // Recursively heapify the parent node
28             heapify(arr, n, parent);
29         }
30     }
31 }
```

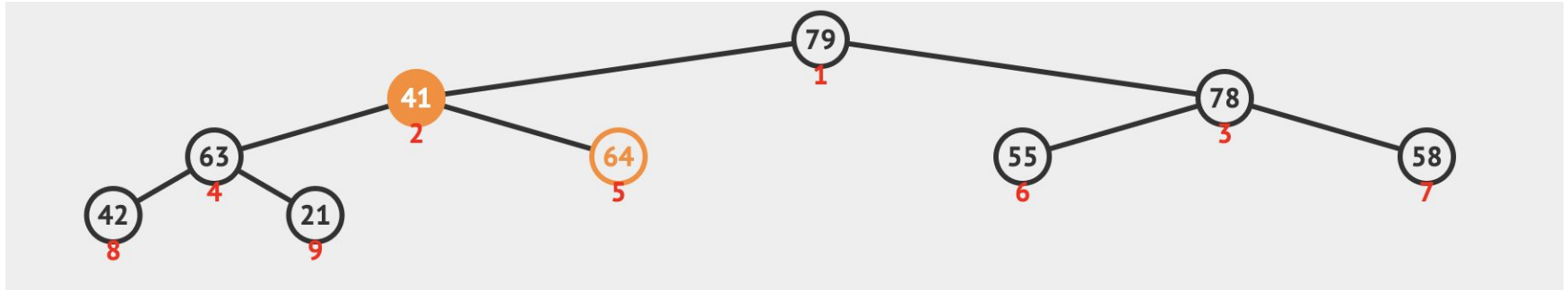
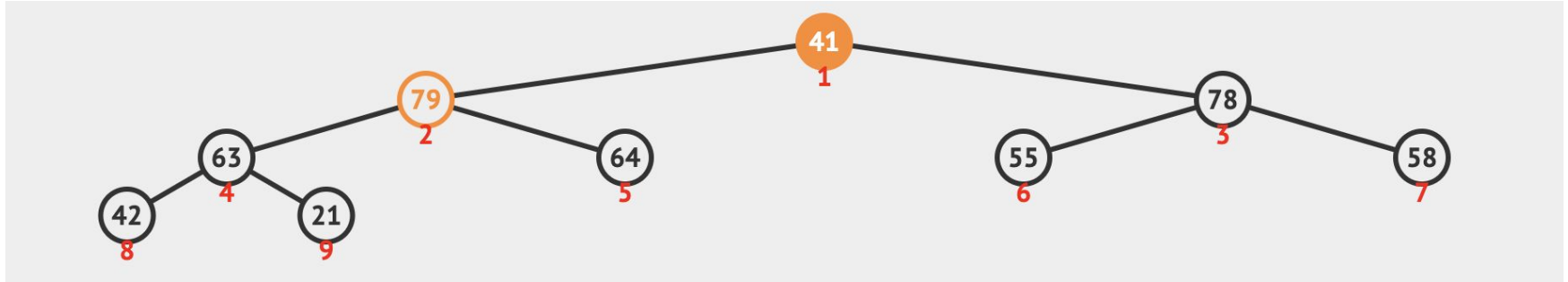
DELETE OPERATION

- If we delete the element from the heap, we consider the case where it deletes the root element of the tree as this is an important operation
- Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.
- We remove the root and replace it with the last element of the heap and then restore the heap property by *percolating down*. Similar to insertion, the worst-case runtime is $O(\log n)$.

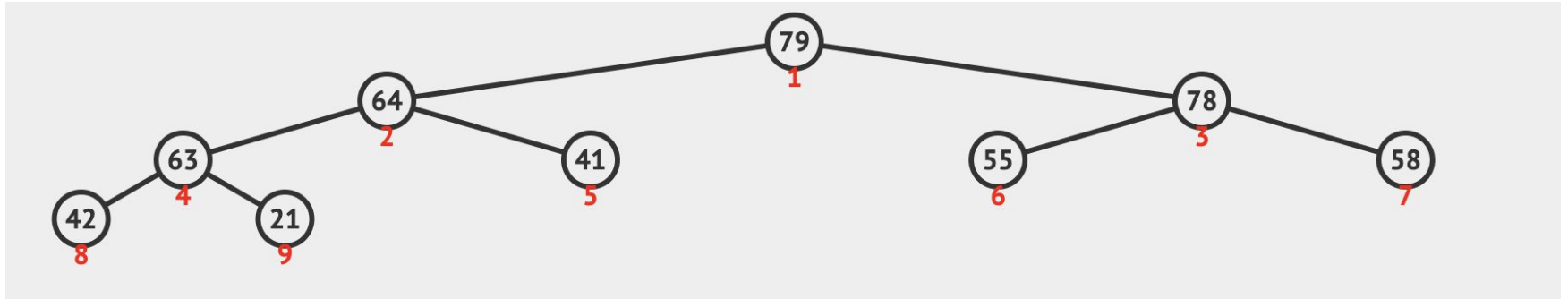
DELETE OPERATION – EXTRACTING MAX NODE



DELETE OPERATION – EXTRACTING MAX NODE



DELETE OPERATION – UPDATED HEAP



DELETE OPERATION CODE – O(LOGN) TIME

```
33 void deleteRoot(int arr[], int& n)
34 {
35     // Get the last element
36     int lastElement = arr[n - 1];
37
38     // Replace root with last element
39     arr[0] = lastElement;
40
41     // Decrease size of heap by 1
42     n = n - 1;
43
44     // heapify the root node
45     heapify(arr, n, 0);
46 }
```

```
48 void heapify(int arr[], int n, int i)
49 {
50     int largest = i; // Initialize largest as root
51     int l = 2 * i + 1; // left = 2*i + 1
52     int r = 2 * i + 2; // right = 2*i + 2
53
54     // If left child is larger than root
55     if (l < n && arr[l] > arr[largest])
56         largest = l;
57     // If right child is larger than largest so far
58     if (r < n && arr[r] > arr[largest])
59         largest = r;
60     // If largest is not root
61     if (largest != i) {
62         swap(arr[i], arr[largest]);
63         // Recursively heapify the affected sub-tree
64         heapify(arr, n, largest);
65     }
66 }
```

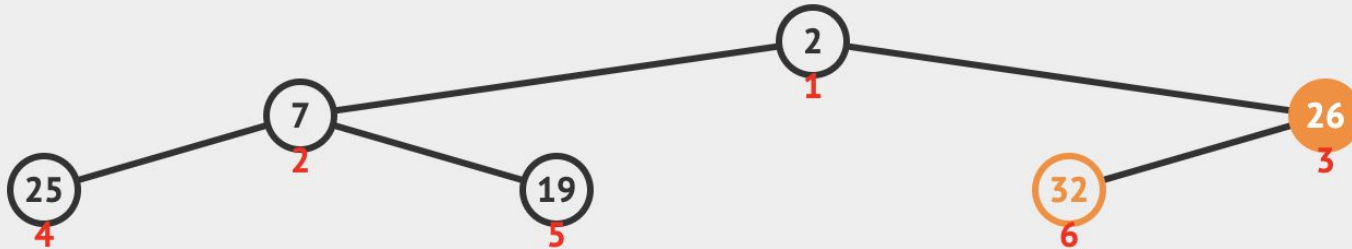
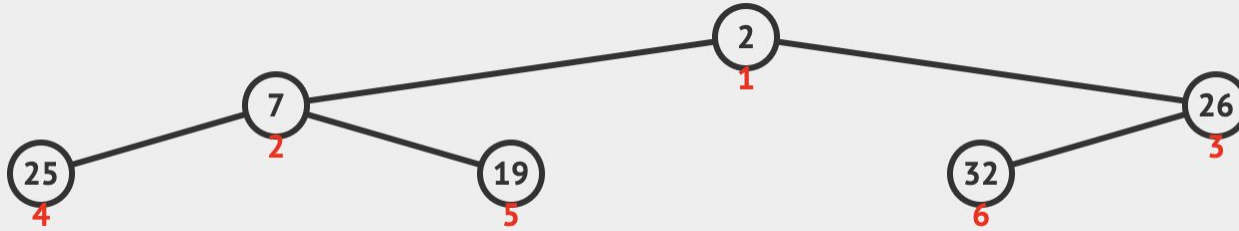
BUILD HEAP – USING HEAPIFY UP

- Start at the top of the heap (the beginning of the array) and heapify bottom-up.
- At each step, the previously sifted items (the items before the current item in the array) form a valid heap, and sifting the next item up places it into a valid position in the heap.
- After sifting up each node, all items satisfy the heap property.
- This approach is more expensive as deeper leveled nodes require more swaps moving upwards and no. of nodes increase at deeper levels
- Time complexity is $O(N * \log N)$ similar to repeatedly inserting elements

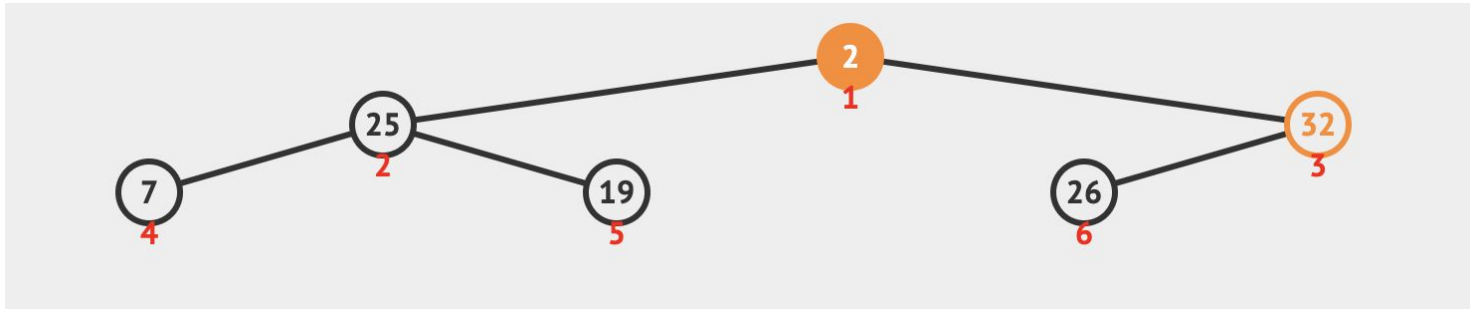
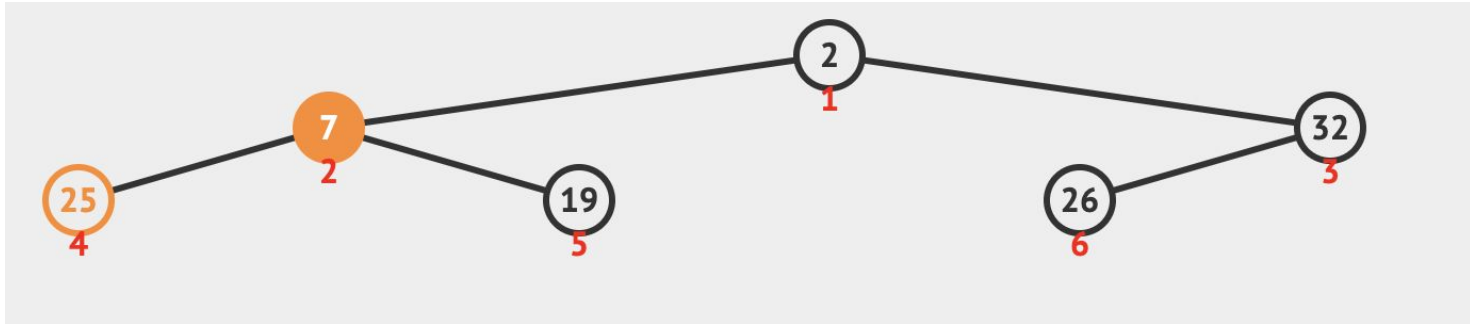
BUILD HEAP - EFFICIENT USING HEAPIFY DOWN

- This approach is optimized by observing the fact that the leaf nodes need not to be heapified as they already follow the heap property.
- Also, the array representation of the complete binary tree contains the level order traversal of the tree.
- So the idea is to find the position of the last non-leaf node and perform the heapify operation of each non-leaf node in reverse level order.
- Here each heapify operation swaps the element downwards till it satisfies the heap property
- This operation hence is more expensive at higher levels and is more efficient as at higher levels the no. of nodes greatly reduces
- Time complexity is $O(N)$

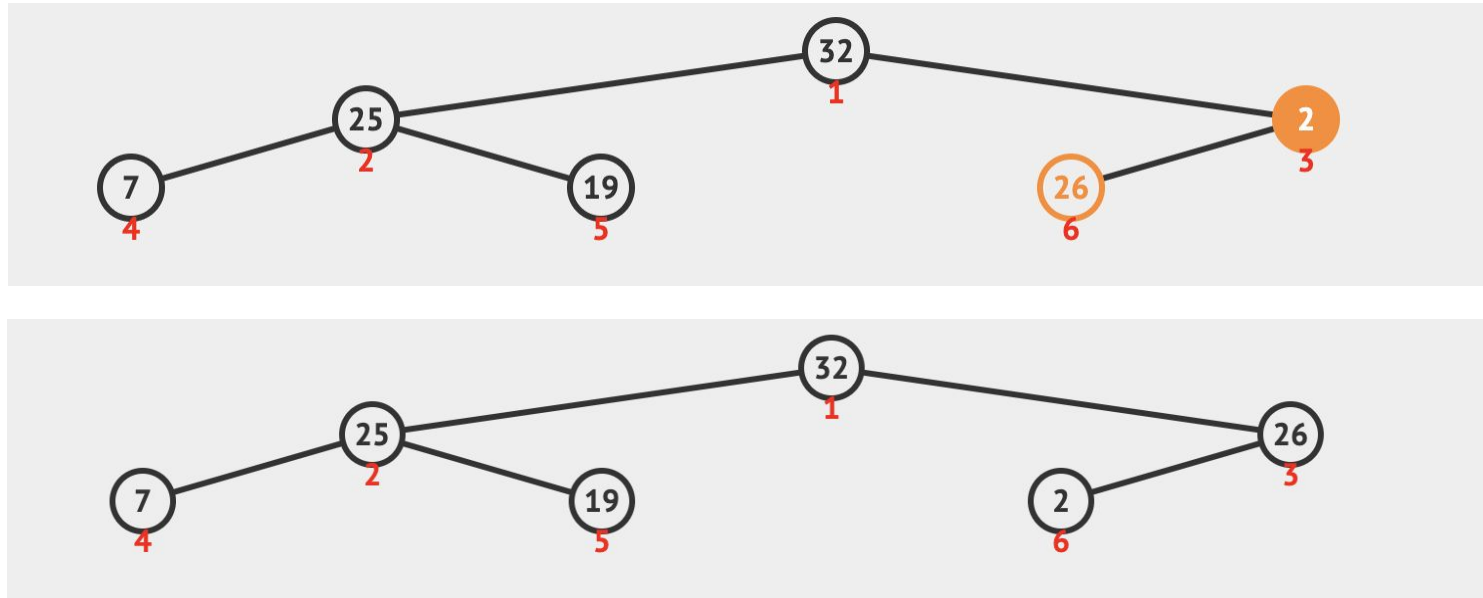
BUILD HEAP - 2,7,26,25,19,32



BUILD HEAP - 2,7,26,25,19,32



BUILD HEAP - 2,7,26,25,19,32



BUILD HEAP – CODE $O(N)$

```
void buildHeap(int arr[], int N)
{
    // Index of last non-leaf node
    int startIdx = (N / 2) - 1;

    // Perform reverse level order traversal
    // from last non-leaf node and heapify
    // each node
    for (int i = startIdx; i >= 0; i--) {
        heapify(arr, N, i);
    }
}
```

```
void heapify(int arr[], int N, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

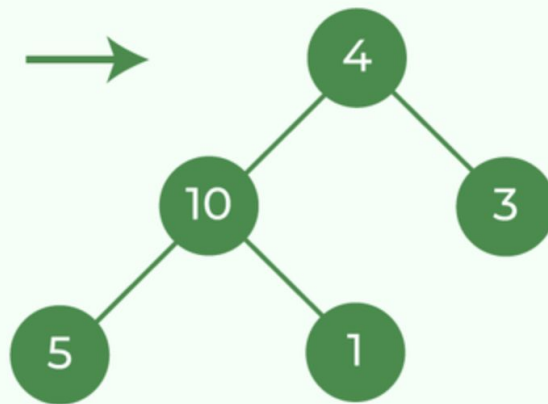
        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}
```

HEAP SORT

Step 1

Build complete Binary Tree

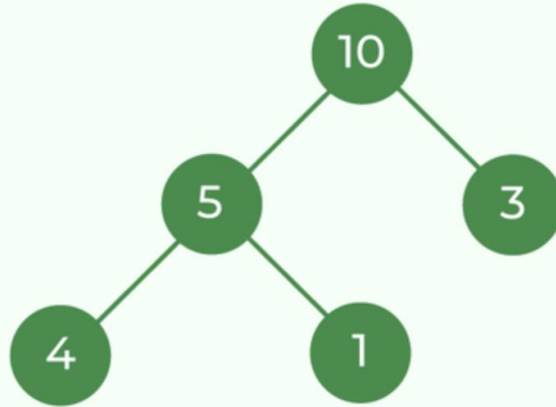
arr = {4, 10, 3, 5, 1}



HEAP SORT

Step 3

Make it a max heap (4 less than 5 & 5 is greater between the two children)

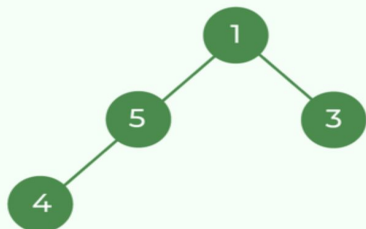


arr = {10, 5, 3, 4, 1}

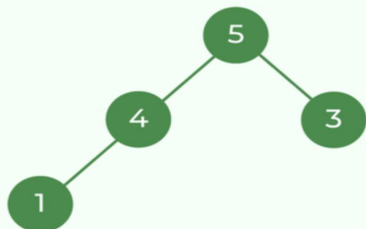
HEAP SORT

Step 4 Remove the max(10) & heapify

→ Remove the max (i.e., move it to the end)

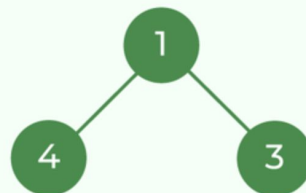


→ Heapify

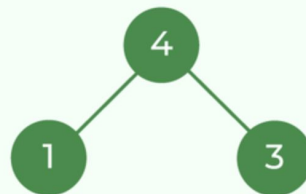


Step 5 Remove the current max(5) & heapify

→ Remove the max (i.e., move it to the end)



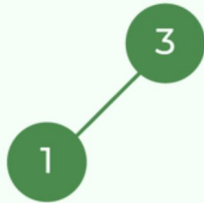
→ Heapify



HEAP SORT

Step 6 Remove the current max(4) & heapify

→ Remove the max (i.e., move it to the end)



arr = {3, 1, 4, 5, 10}

It is already in max heap form

Step 7 Remove the max(3)

arr = {1, 3, 4, 5, 10}

The array is now sorted

HEAP SORT - CODE $O(N \log N)$

```
void heapSort(int arr[], int N)
{
    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

```
void heapify(int arr[], int N, int i)
{
    // Initialize largest as root
    int largest = i;
    // left = 2*i + 1
    int l = 2 * i + 1;
    // right = 2*i + 2
    int r = 2 * i + 2;
    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;
    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}
```

GATE 2021 Question

Let H be a binary min-heap consisting of n elements implemented as an array. What is the worst case time complexity of an optimal algorithm to find the maximum element in H ?

- A. $\Theta(1)$
- B. $\Theta(\log n)$
- C. $\Theta(n)$
- D. $\Theta(n \log n)$

Answer

In a min heap, maximum element is present in one of the leaf nodes.

If index of heap starts with 1, indices of leaves are $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \lfloor n/2 \rfloor + 3 \dots n$.

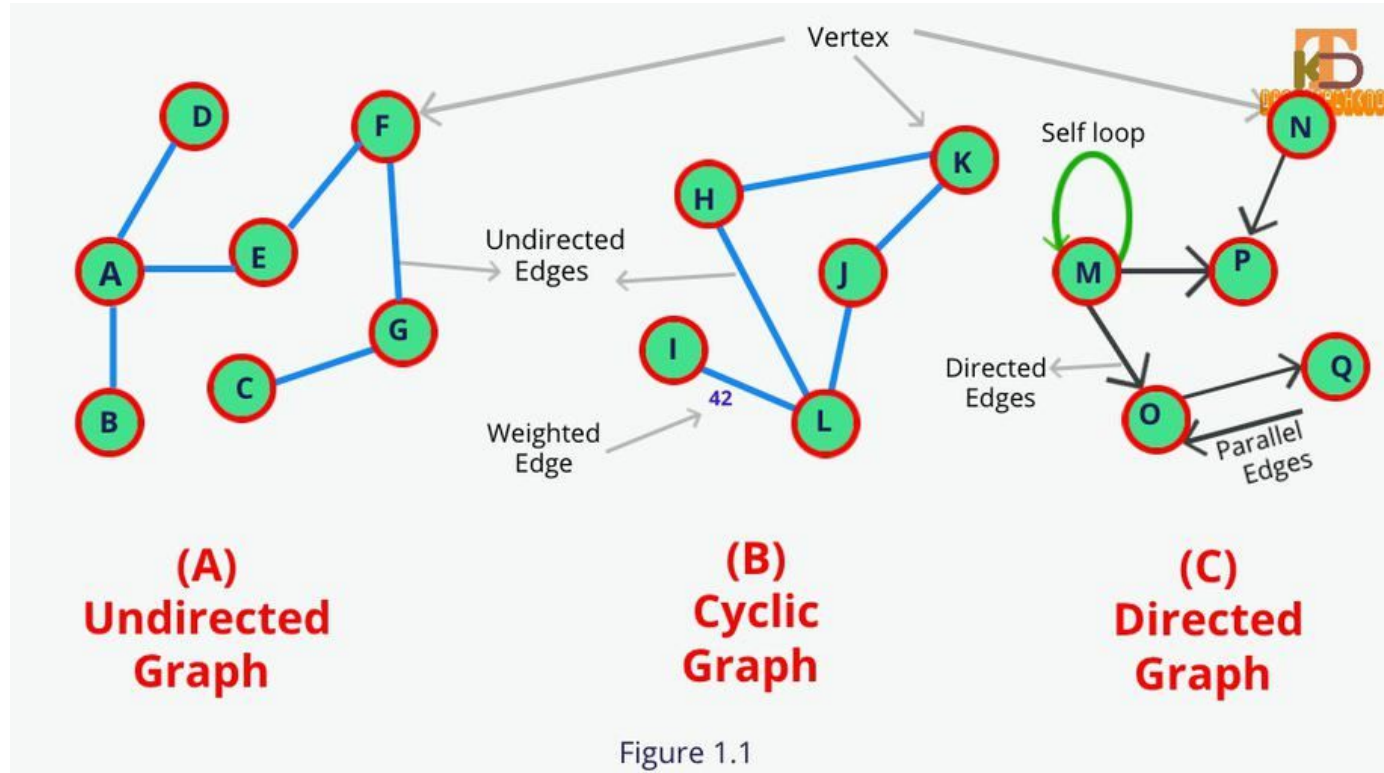
So, we have to perform linear search on at most $n/2 + 1$ elements to find the maximum element. (we cannot perform binary search since it is not guaranteed that leaves are in sorted order) and that multiplied by some constant c will be the time complexity of the optimal algorithm. (Here, c includes the cost of all operations which includes comparison, index shift etc. for a single maximum element compare)

In asymptotic terms, $c * (n/2 + 1)$ is $\Theta(n)$.

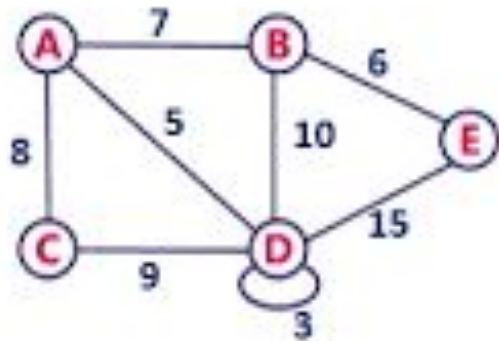
GRAPHS



COMMON GRAPH DATA STRUCTURES

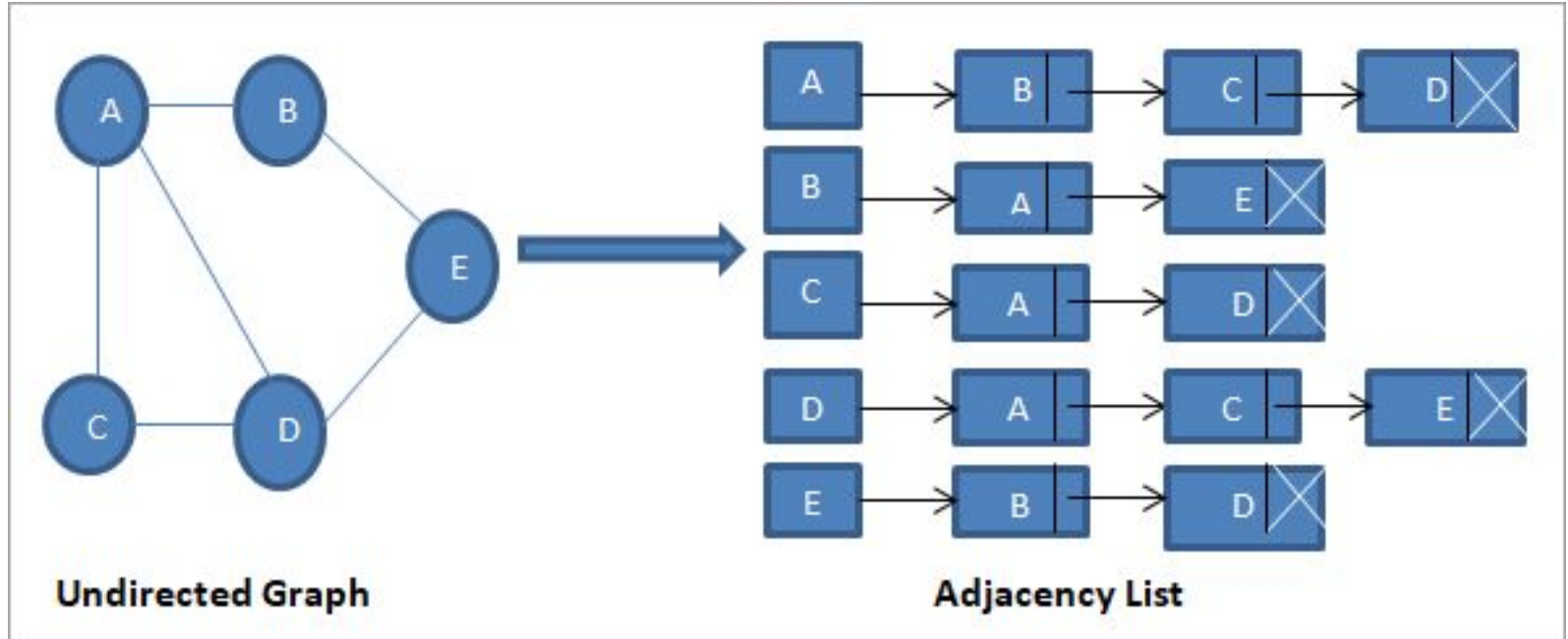


REPRESENTATION OF GRAPHS

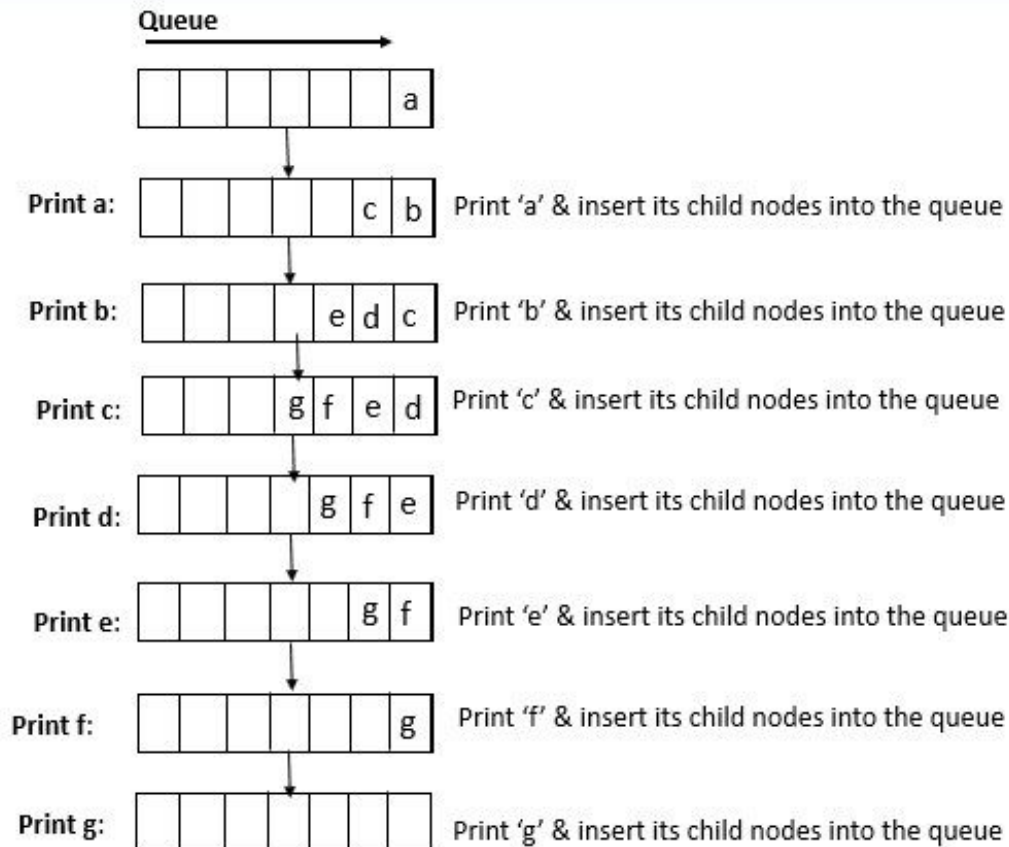
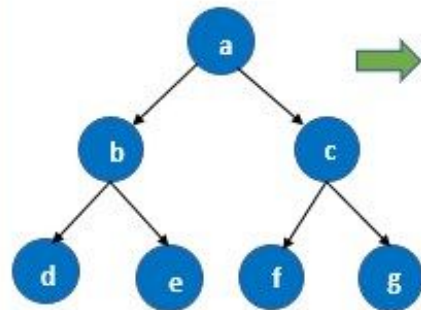


	A	B	C	D	E
A	0	7	8	5	0
B	7	0	0	10	6
C	8	0	0	9	0
D	5	10	9	3	15
E	0	6	0	15	0

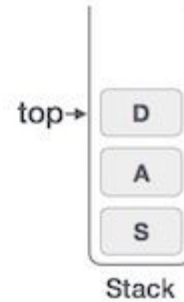
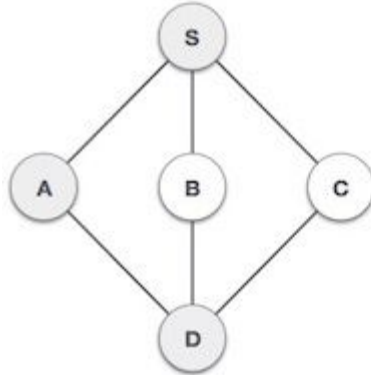
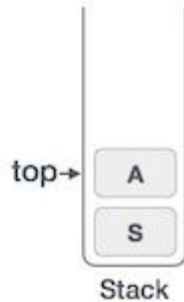
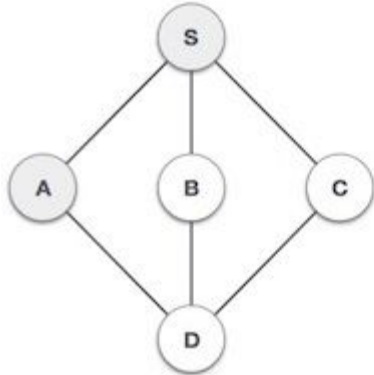
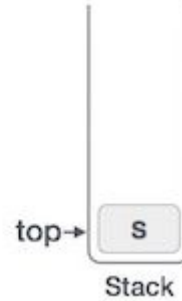
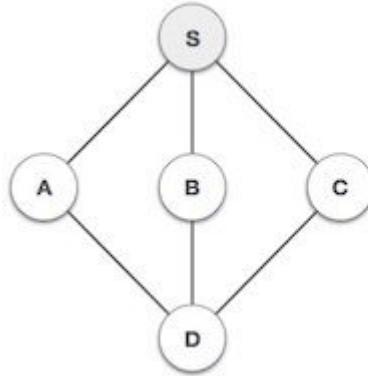
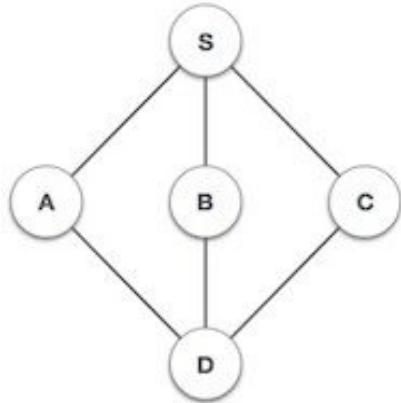
REPRESENTATION OF GRAPHS



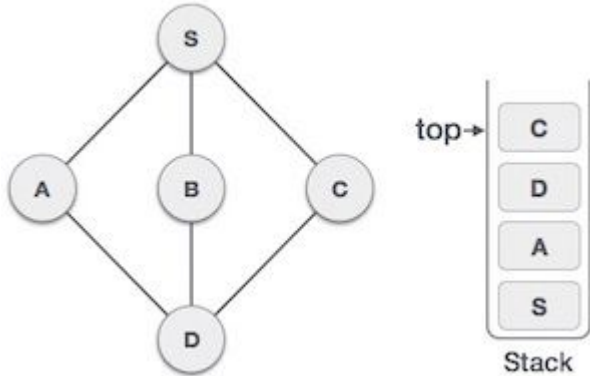
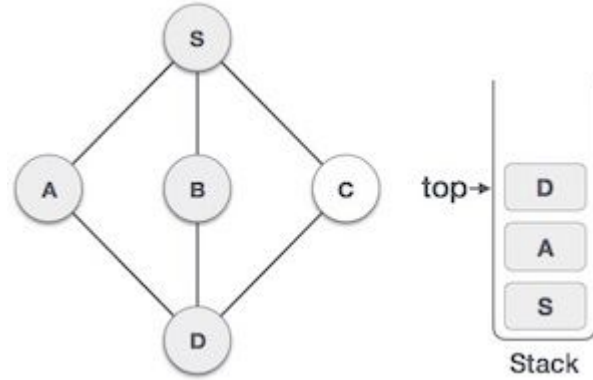
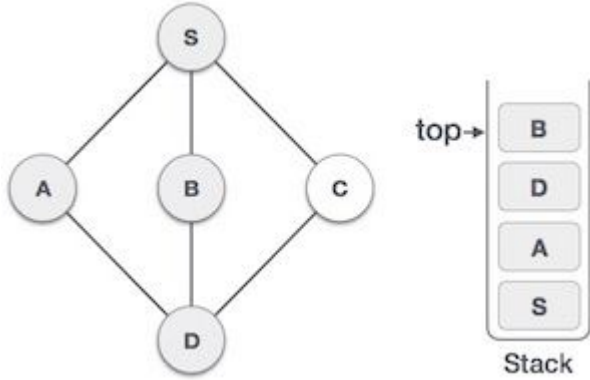
BREADTH FIRST SEARCH



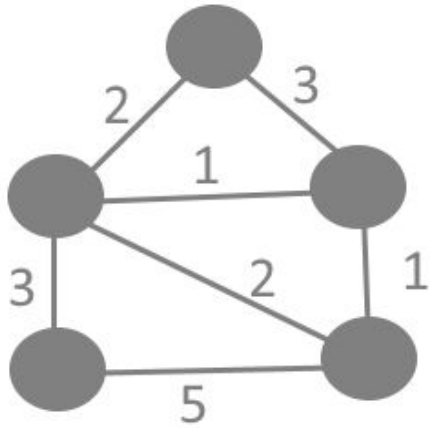
DEPTH FIRST SEARCH



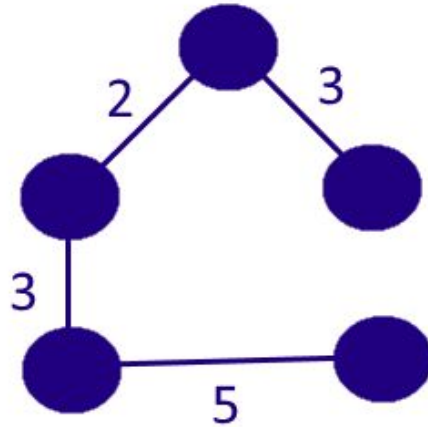
DEPTH FIRST SEARCH



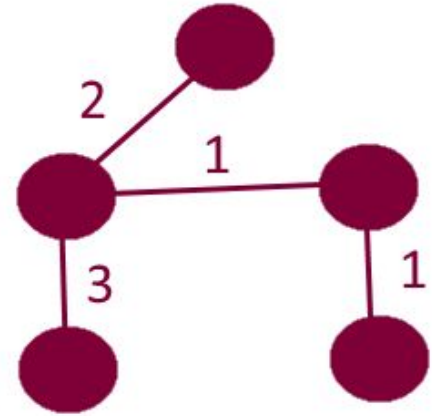
MINIMUM SPANNING TREE (MST)



Graph



Spanning Tree
Cost = 13

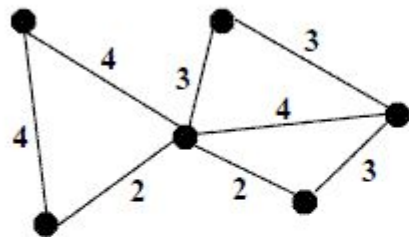


Minimum Spanning
Tree, Cost = 7

KRUSKAL'S ALGORITHM

Kruskal's Algorithm

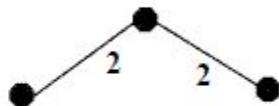
1 Given a network.....



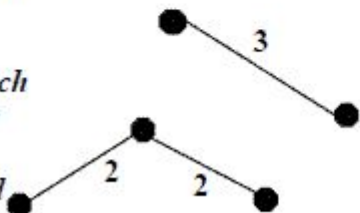
2 Choose the shortest edge (if there is more than one, choose any of the shortest).....



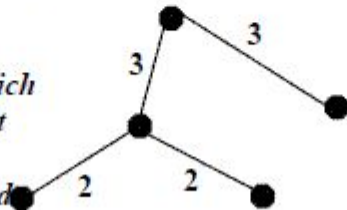
3 Choose the next shortest edge and add it.....



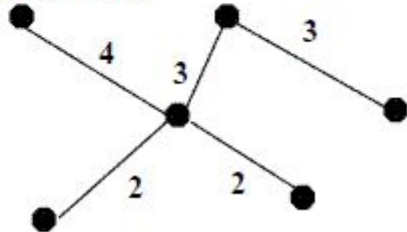
4 Choose the next shortest edge which wouldn't create a cycle and add it.



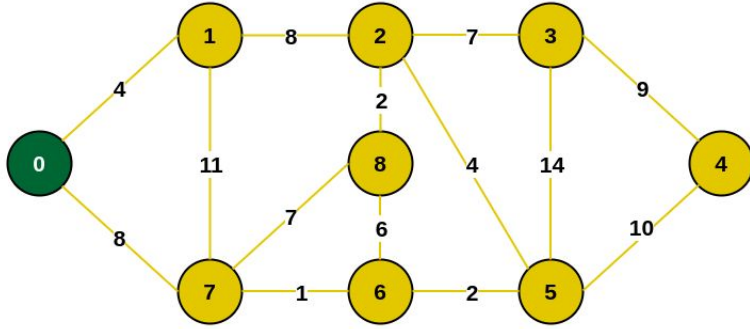
5 Choose the next shortest edge which wouldn't create a cycle and add it.



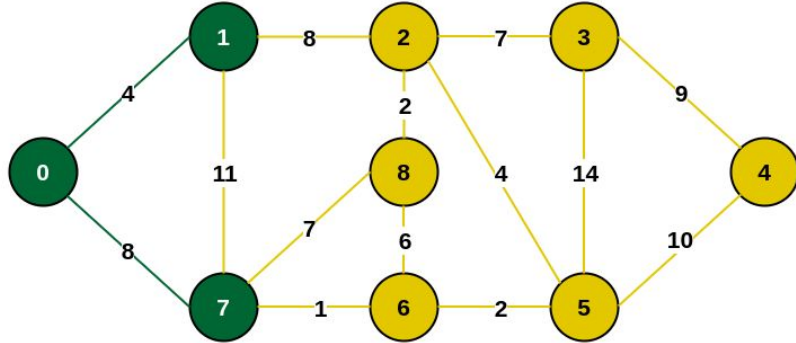
6 Repeat until you have a minimal spanning tree.



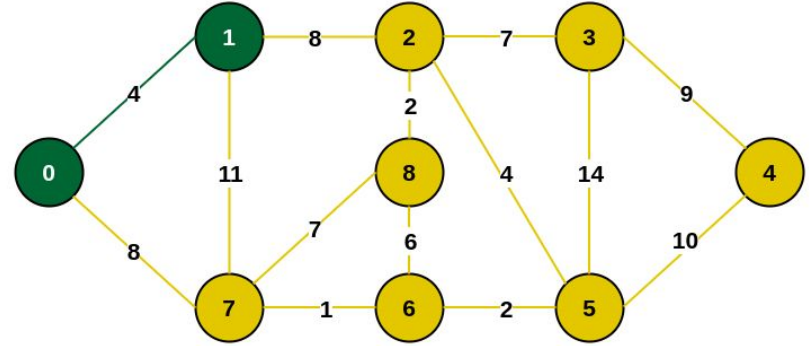
PRIM'S ALGORITHM



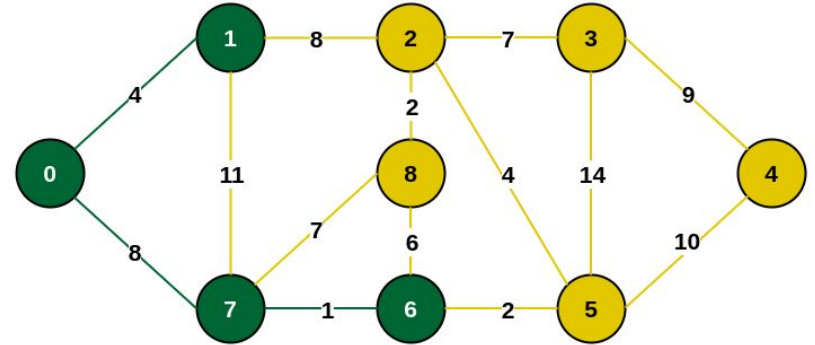
Select an arbitrary starting vertex. Here we have selected 0



Minimum weighted edge from MST to other vertices is 0-7 with weight 8

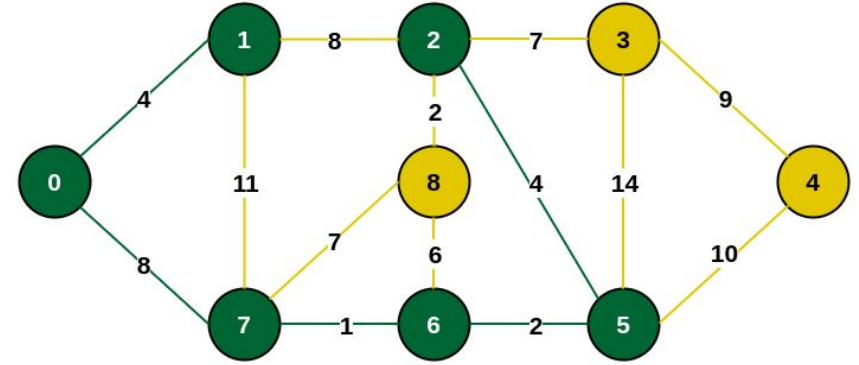
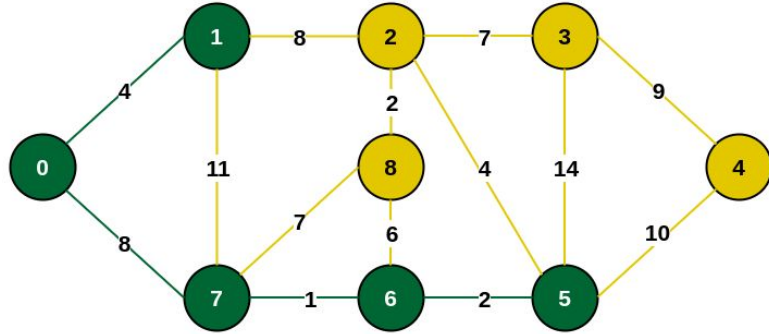


Minimum weighted edge from MST to other vertices is 0-1 with weight 4



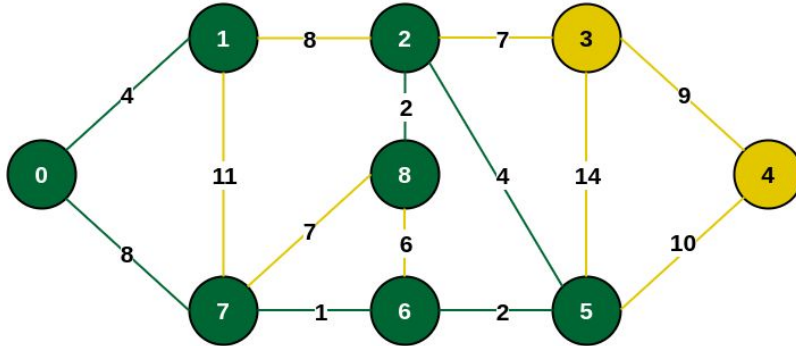
Minimum weighted edge from MST to other vertices is 7-6 with weight 1

PRIM'S ALGORITHM



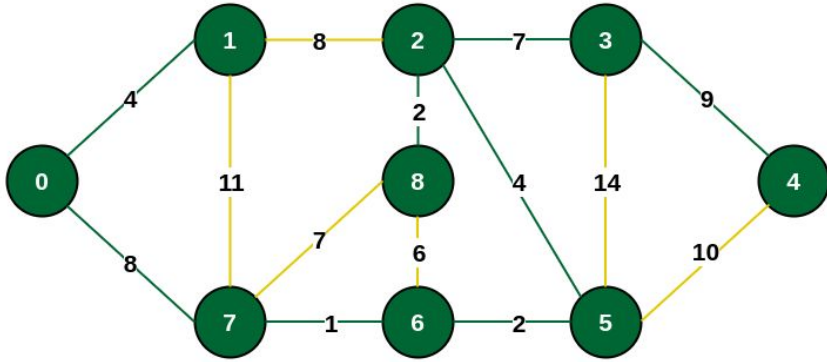
Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Minimum weighted edge from MST to other vertices is 5-2 with weight 4

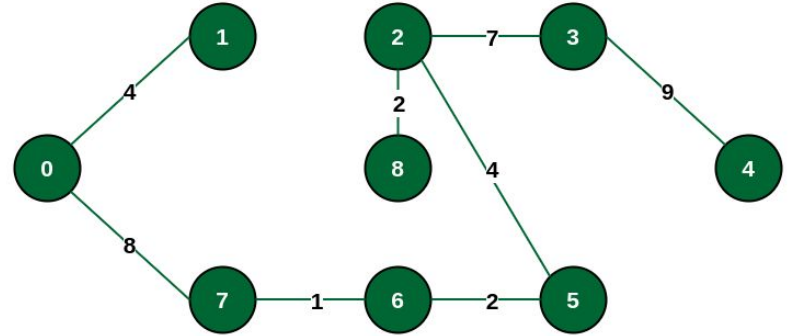


Minimum weighted edge from MST to other vertices is 2-8 with weight 2

PRIM'S ALGORITHM



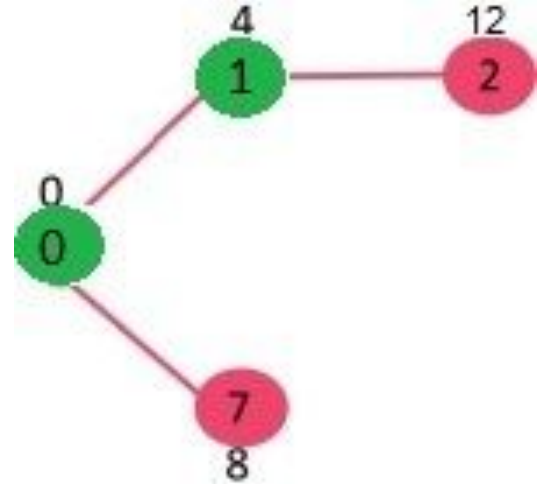
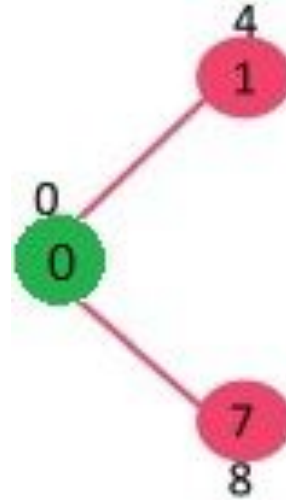
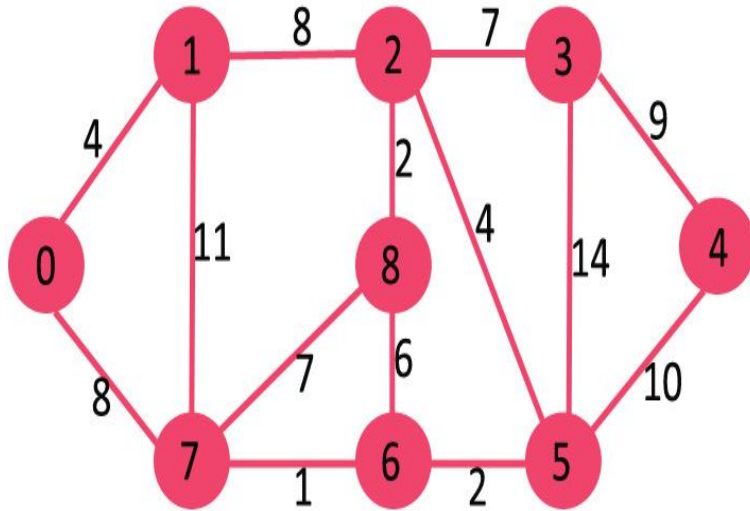
Minimum weighted edge from MST to other vertices is 3-4 with weight 9



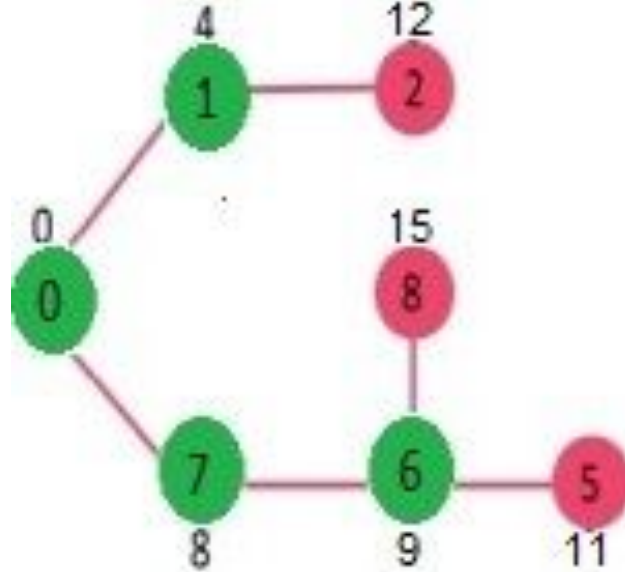
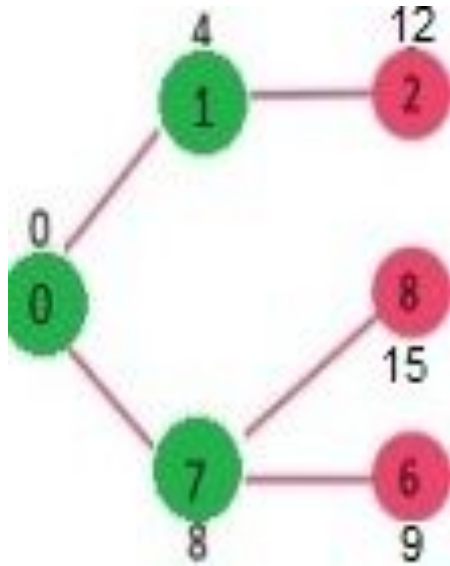
The final structure of MST

DIJKSTRA'S ALGORITHM

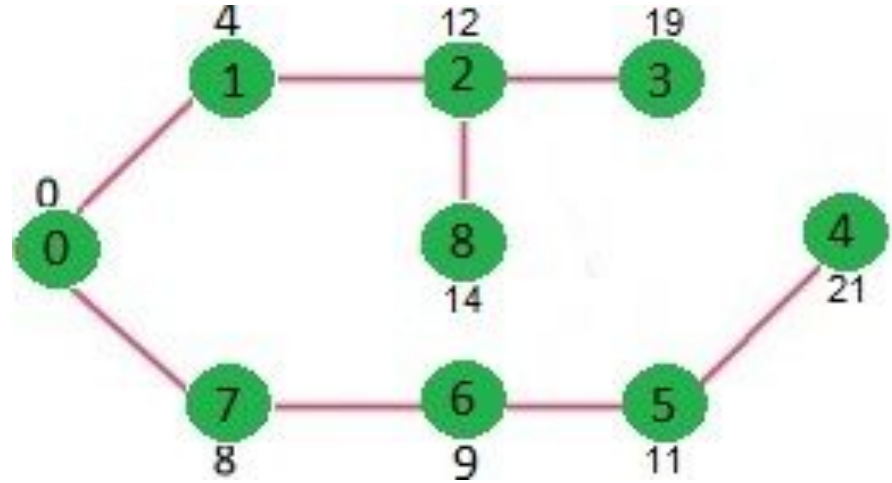
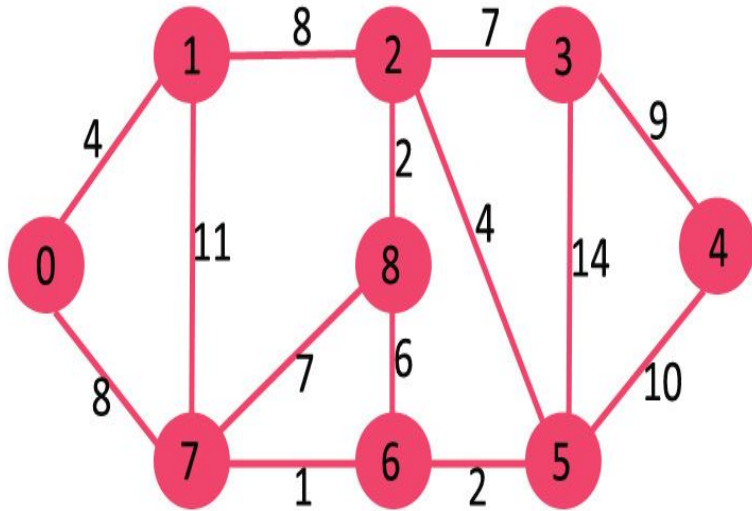
Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph. (In the below example lets take 0 as source node)



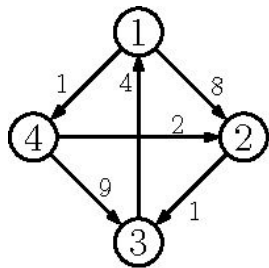
DIJKSTRA'S ALGORITHM



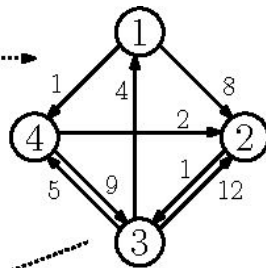
DIJKSTRA'S ALGORITHM



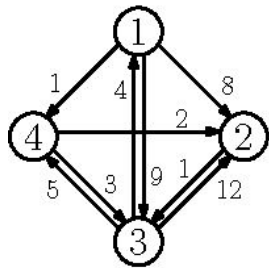
FLOYD WARSHALL ALGORITHM (All Pair Shortest Path)



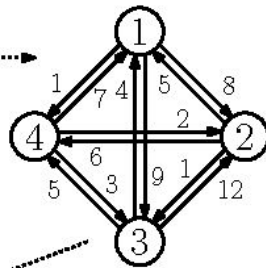
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



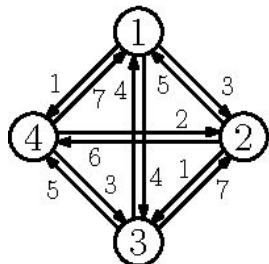
$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

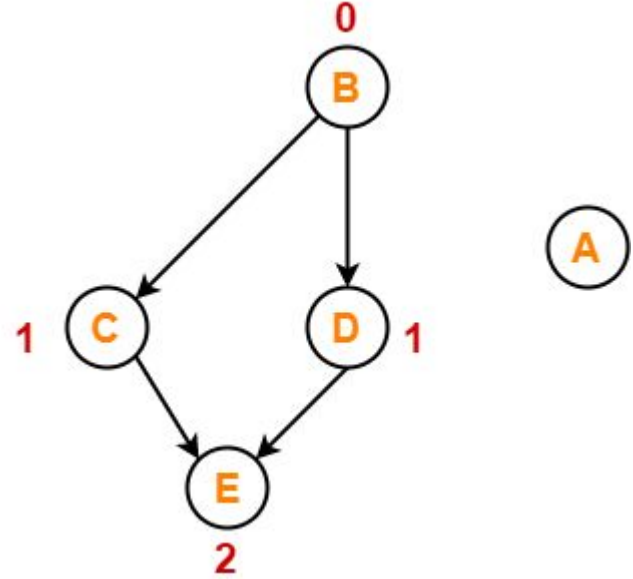
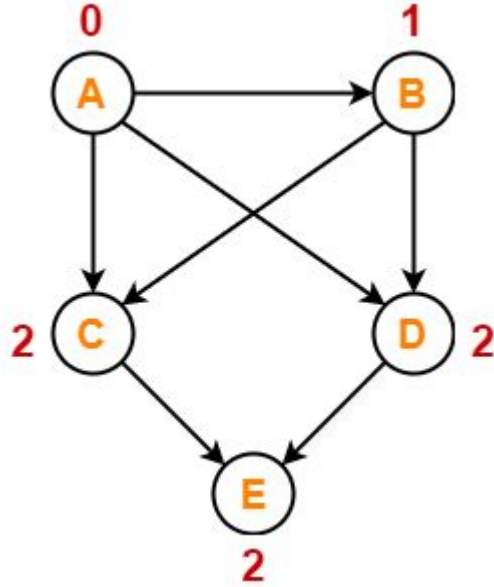
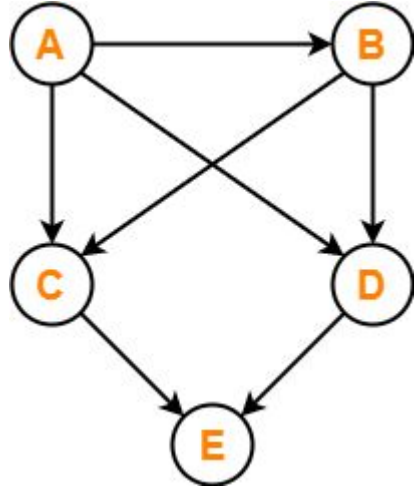
$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

TOPOLOGICAL SORTING

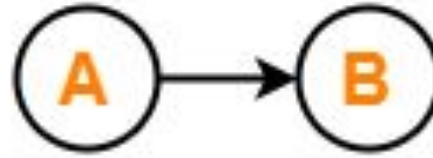
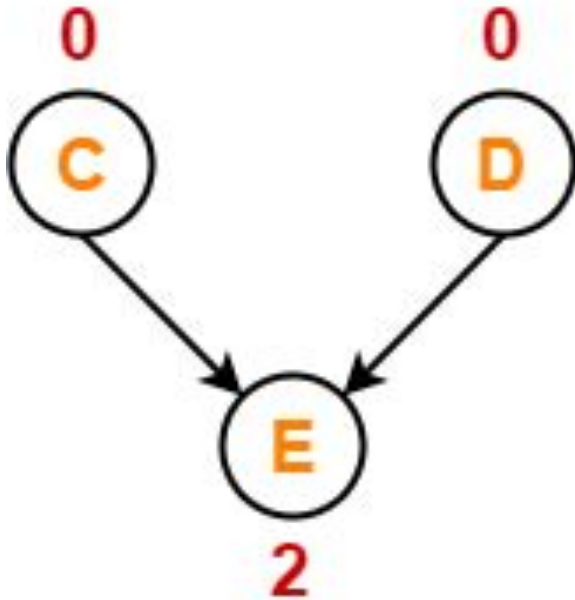
Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.

- Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**.
- There may exist multiple different topological orderings for a given directed acyclic graph.

TOPOLOGICAL SORTING

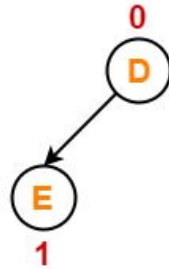


TOPOLOGICAL SORTING



TOPOLOGICAL SORTING

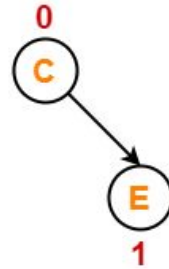
Case-01



Case-01



Case-02



Case-02

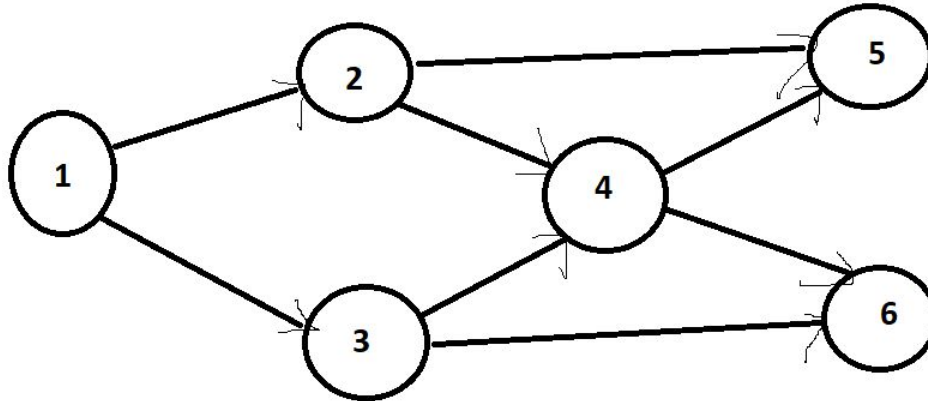


GATE QUESTION - 1

Consider the DAG with Consider $V = \{1, 2, 3, 4, 5, 6\}$, shown below. Which of the following options is NOT a topological ordering?

- (A) 1 2 3 4 5 6
- (B) 1 3 2 4 5 6
- (C) 1 3 2 4 6 5
- (D) 3 2 4 1 6 5

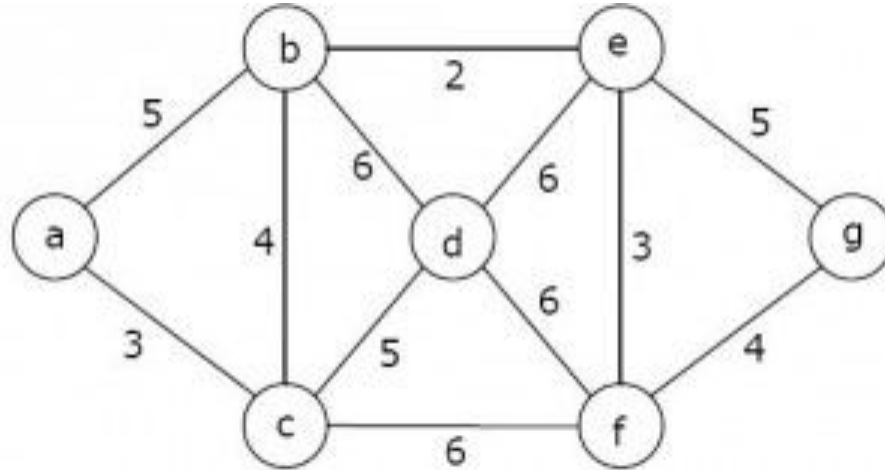
ANS : (D)



GATE QUESTION - 2

Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

- (A) (b,e)(e,f)(a,c)(b,c)(f,g)(c,d)
- (B) (b,e)(e,f)(a,c)(f,g)(b,c)(c,d)
- (C) (b,e)(a,c)(e,f)(b,c)(f,g)(c,d)
- (D) (b,e)(e,f)(b,c)(a,c)(f,g)(c,d)



Answer: (D)

GATE QUESTION - 3

Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree?

(A) (E, G), (C, F), (F, G), (A, D), (A, B), (A, C)

(B) (A, D), (A, B), (A, C), (C, F), (G, E), (F, G)

(C) (A, B), (A, D), (D, F), (F, G), (G, E), (F, C)

(D) (A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

Answer: (D)

