# *Unit – V – Advanced Concepts Of Trees*

Binary Search Trees, Definition, Searching a Binary Search Tree, Insertion into a Binary Search Tree, Deletion from a Binary Search Tree, Height of Binary Search Tree. Heaps, Priority Queues, Definition of a Max/Min Heap, Insertion into a Max/Min Heap, Deletion from a Max/Min Heap

Introduction to Binary Search Trees

**Binary Search Tree: Definition:** A binary search tree, also known as an ordinary binary search tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left-subtree have a value less than that of the root node. Correspondingly, all the nodes in the right subtree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.

To summarize, a binary search tree is a binary tree with the following properties:

- The left sub tree of a node N contains values are less than N's value
- The right sub tree of a node N contains values that are greater than N's value
- Both the left and the right binary trees also verify these properties and thus, are binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

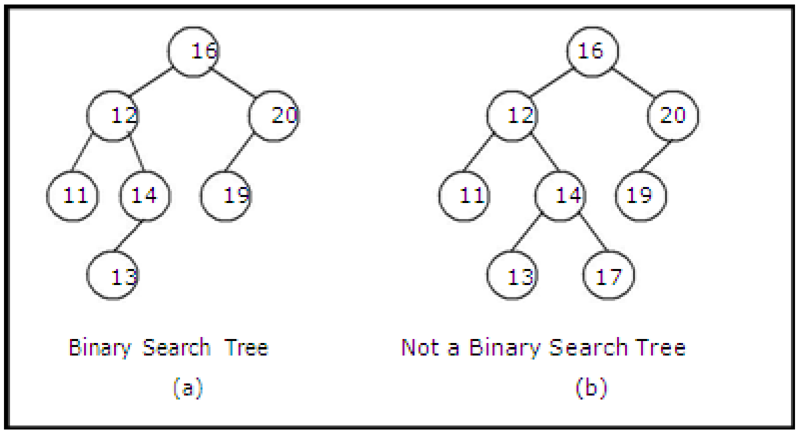


Figure 5.2.5. Examples of binary search trees

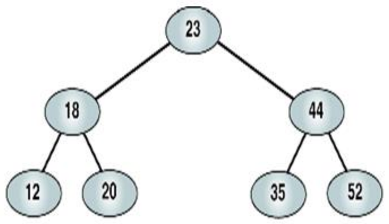
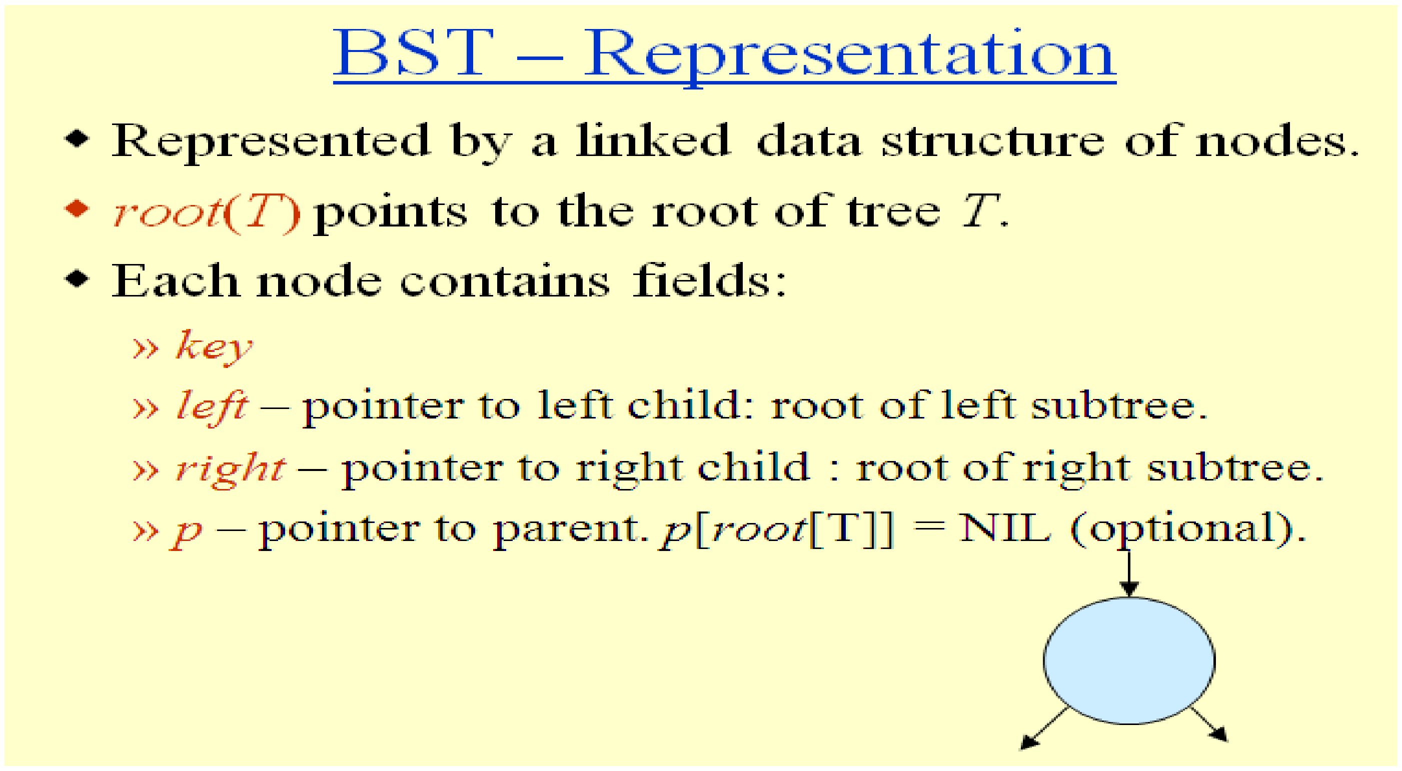


FIGURE 7-4 Example of a Binary Search Tree



## BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.
- Each node contains fields:
  - » *key*
  - » *left* – pointer to left child: root of left subtree.
  - » *right* – pointer to right child : root of right subtree.
  - » *p* – pointer to parent. *p*[*root*[T]] = NIL (optional).

A binary tree is a binary search tree (BST) if and only if an inorder traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node

4. Duplicate node keys are not allowed.

**Operations on Binary Search Trees:** The following are the basic operations on binary search tree:

- Searching for a Node in a Binary Search Tree
- Inserting a New node in a Binary Search Tree
- Deleting a Node from a Binary Search Tree
    - Deleting a node that has no child
    - Deleting a node with one child
    - Deleting a node with two children

Searching a Binary Search Tree

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message. However, if these are nodes in the tree, then the search function checks if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

**Algorithm to Search for a node in Binary Search Tree:**

```
Searchelement(TREE, VAL)

Step1:      if tree->data=val or tree=NULLs

    Return tree

    Else if val<tree->data

            Return searchelement(TREE->LEFT,VAL)

    Else

            Return seachelement(TREE->RIGHT, VAL)

    End if

Step 2: Stop
```

Insertion into a Binary Search Tree

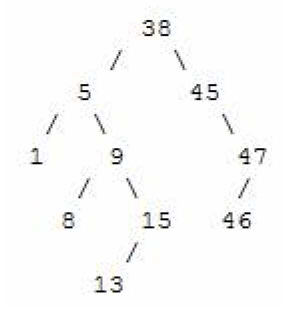
The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

**Algorithm to insert a new node in Binary Search Tree:**

```
insert (TREE, VAL)

Step1:  if  tree=NULL

        Allocate memory for TREE

        SET TREE->DATA = VAL

        SET TREE->LEFT = TREE->RIGHT=NULL

        Else if val<tree->data

                insert(TREE->LEFT, VAL)

        Else

                insert (TREE->RIGHT, VAL)

        End if

Step 2: Stop
```

Deletion from a Binary Search Tree

A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree



could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The later case is the hardest to resolve. But we will find a way to handle this situation as well.

Case 1: Deleting a Node that has no children
We can simply remove the deleted node without any issue.

Case 2: Deleting a Node with one Child
To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

Case 3: Deleting a Node with Two Children
To handle this case, replace the node's value with its in-order predecessor (largest value in the left subtree) or in order successor (smallest value in the right subtree). The in-order

Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:

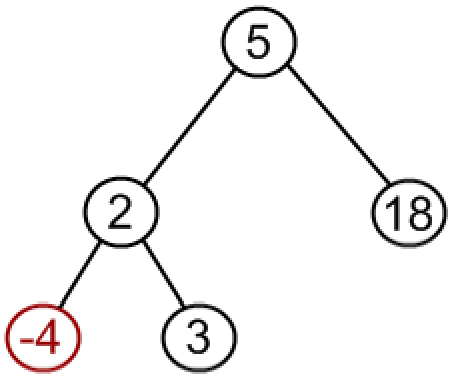
- search for a node to remove;
- if the node is found, run remove algorithm.

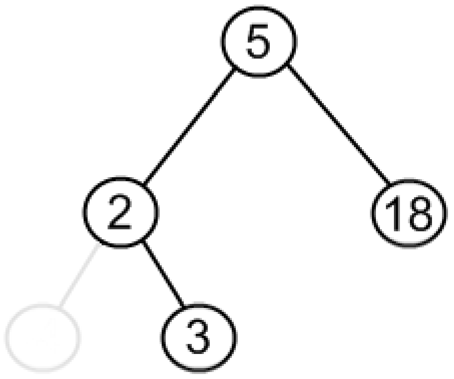
## Remove algorithm in detail

1. Node to be removed has no children.

   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

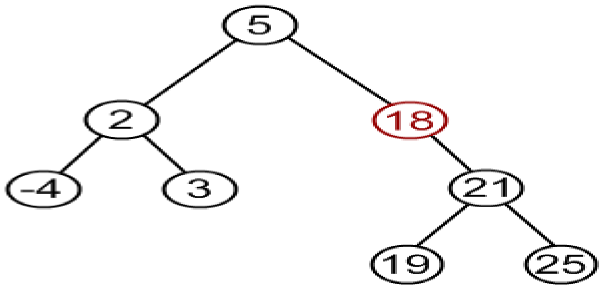
**Example.** Remove -4 from a BST.

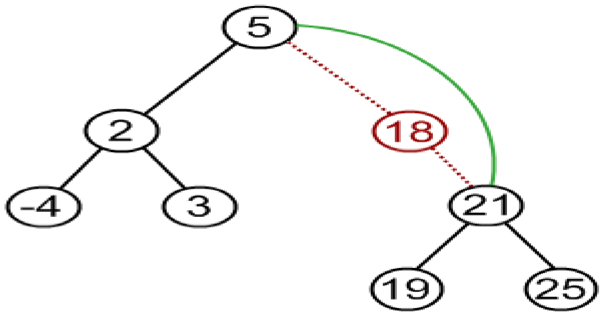


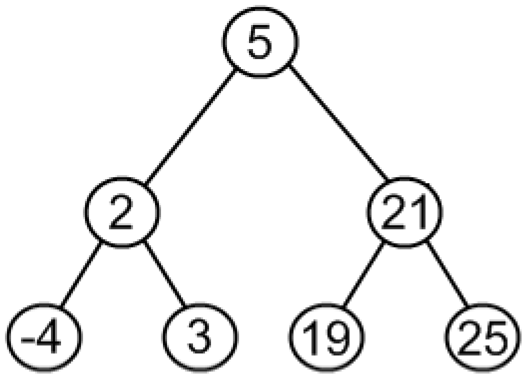


2. Node to be removed has one child.

   It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.**Example.** Remove 18 from a BST.

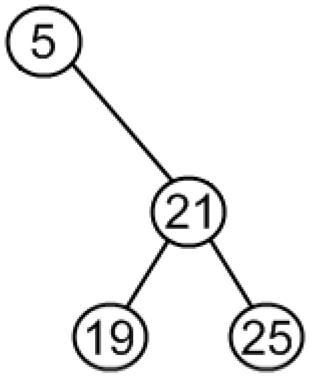


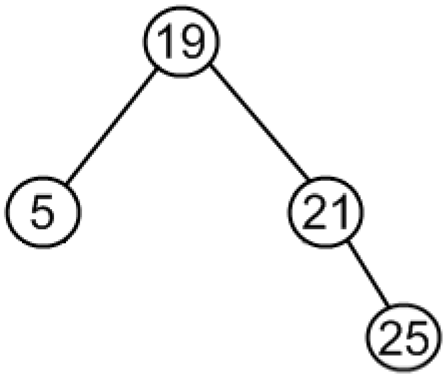




3. Node to be removed has two children.

   This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:





   contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

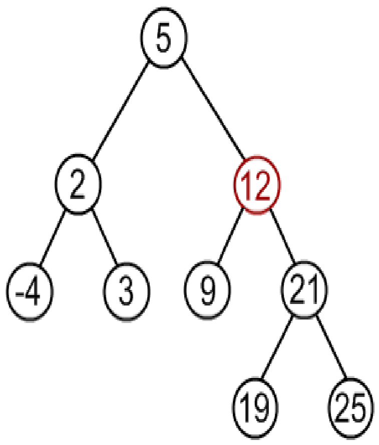
   - choose minimum element from the right subtree (19 in the example);
   - replace 5 by 19;
   - hang 5 as a left child.

   The same approach can be utilized to remove a node, which has two children:

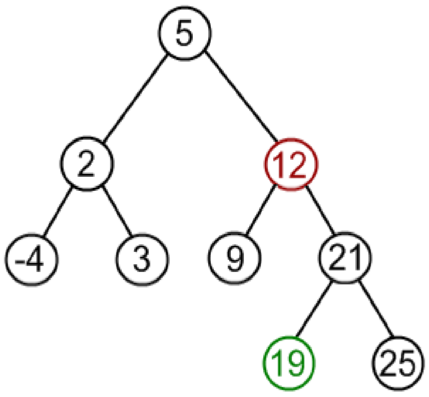
   - find a minimum value in the right subtree;
   - replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
   - apply remove to the right subtree to remove a duplicate.

   Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

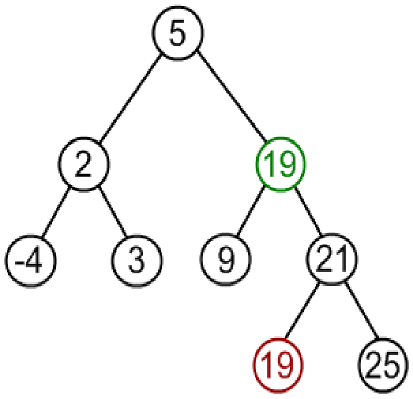
   **Example.** Remove 12 from a BST.



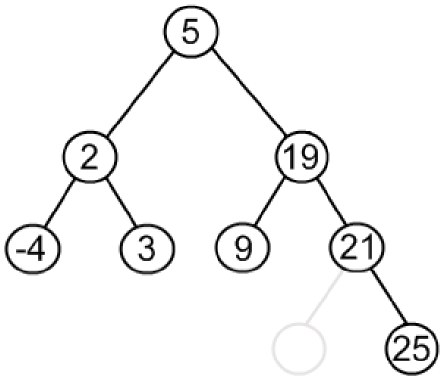
Find minimum element in the right subtree of the node to be removed. In current example it is 19.



Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Remove 19 from the left subtree.

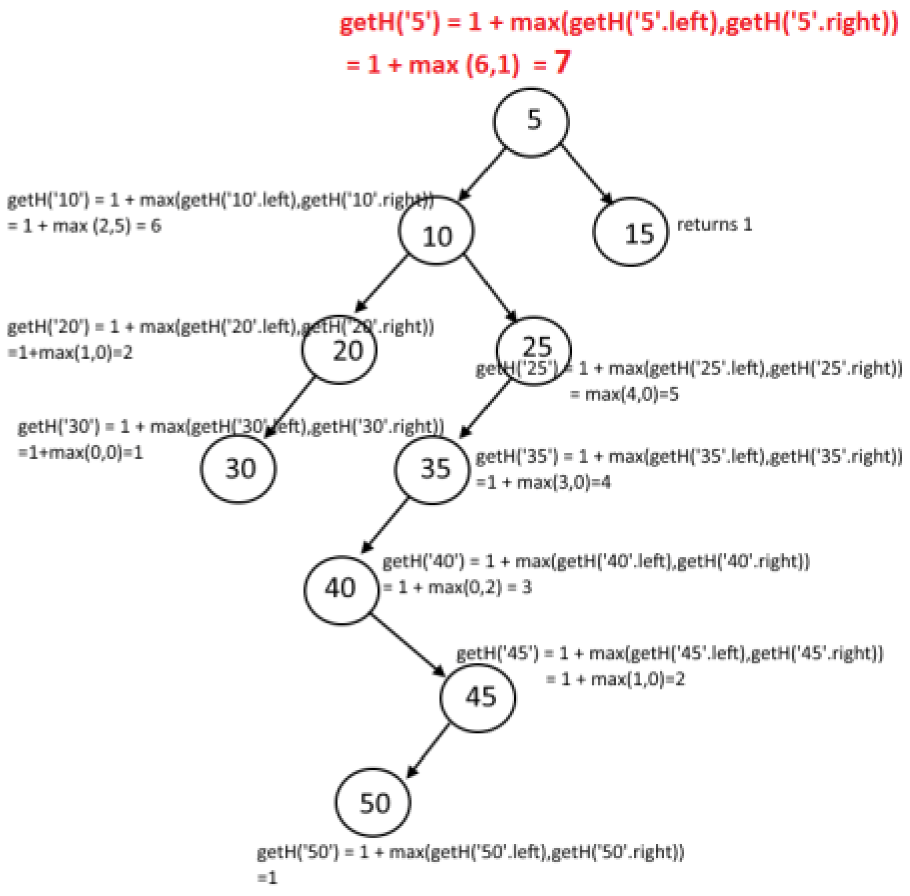


### 4.1.1.1 **Lecture-62:** Height of Binary Search Tree, Heaps

**Find the Maximum Depth OR Height of a Binary Tree**

Algorithm to find maximum depth or height of a binary tree as following:

- Get the height of left sub tree, say left Height
- Get the height of right sub tree, say right Height
- Take the Max(left Height, right Height) and add 1 for the root and return
- Call recursively.


getH('5') = 1 + max(getH('5'.left),getH('5'.right))
= 1 + max (6,1) = 7
getH('10') = 1 + max(getH('10'.left),getH('10'.right))
= 1 + max (2,5) = 6
returns 1
getH('20') = 1 + max(getH('20'.left),getH('20'.right))
=1+max(1,0)=2
getH('25') = 1 + max(getH('25'.left),getH('25'.right))
= max(4,0)=5
getH('30') = 1 + max(getH('30'.left),getH('30'.right))
=1+max(0,0)=1
getH('35') = 1 + max(getH('35'.left),getH('35'.right))
=1 + max(3,0)=4
getH('40') = 1 + max(getH('40'.left),getH('40'.right))
= 1 + max(0,2) = 3
getH('45') = 1 + max(getH('45'.left),getH('45'.right))
= 1 + max(1,0)=2
getH('50') = 1 + max(getH('50'.left),getH('50'.right))
=1

## *Priority Queues*

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

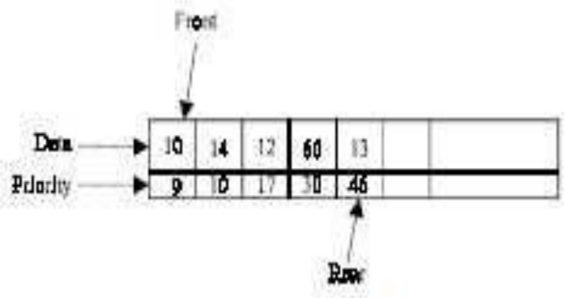
1.An element of higher priority is processed before any element of lower priority.

2.Two elements with the same priority are processed according to the order in which they were inserted to the queue.

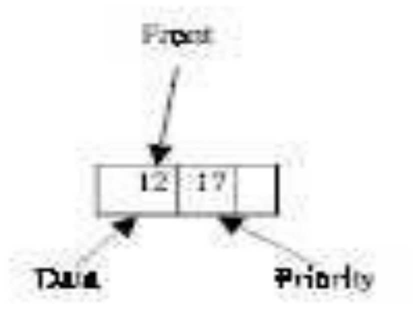
For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

Below Fig. gives a pictorial representation of priority queue using arrays after adding 5

elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the

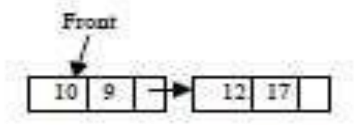


Priority queue representation using arrays priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is O(n), which is much higher than the other queue (ie; other queues takes only O(1) ) for inserting an element. So it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list - which is discussed in this section. A node in the priority queue will contain DATA, PRIORITY and NEXT field. DATA field will store the actual information; PRIORITY field will store its corresponding priority of the DATA and NEXT will store the address of the next node. Fig below shows the linked list representation of the node when a DATA (i.e., 12) and PRIORITY (i.e., 17) is inserted in a priority queue.

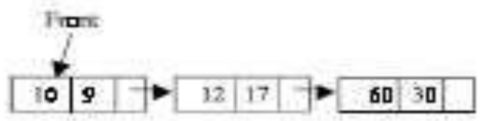


Linked list representation of priority queue

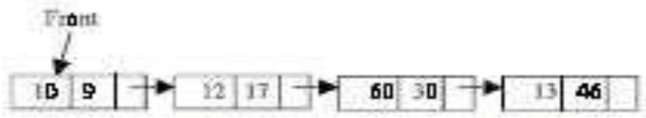
When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end. Following figures will illustrate the push and pop operation of priority queue using linked list.



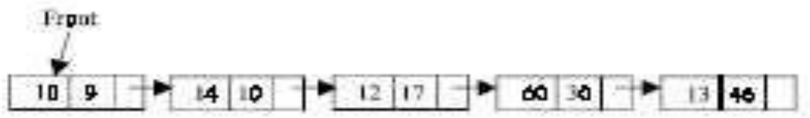
push(DATA =10, PRIORITY = 9)



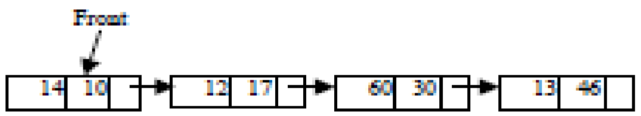
push(DATA = 60, PRIORITY = 30)



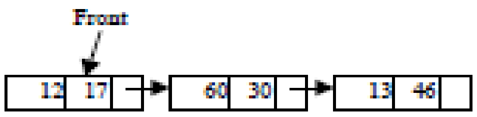
push(DATA = 13, PRIORITY = 46)



push(DATA = 14, PRIORITY = 10)



x = pop() (i.e., 10)



x = pop() (i.e., 14)

**Binary Heap:**

A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

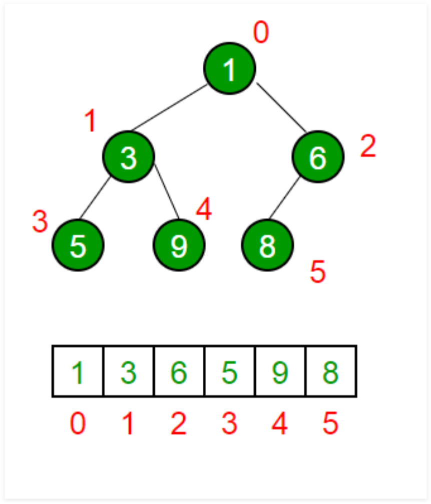
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

**Examples of Min Heap:**

```
          10                      10
       /      \               /       \
     20        100          15          30
    /                     /   \       /   \
  30                     40    50   100    40
```

**How is Binary Heap represented?**
A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array.

The traversal method use to achieve Array representation is **Level Order**



**Applications of Heaps:**
**1)** Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
**2)** Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
**3)** Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.
**4)** Many problems can be efficiently solved using Heaps. See following for example.
a) K'th Largest Element in an array.
b) Sort an almost sorted array/
c) Merge K Sorted Arrays.

**Operations on Min Heap:**
**1)** getMini(): It returns the root element of Min Heap. Time Complexity of this operation is O(1).
**2)** extractMin(): Removes the minimum element from Min Heap. Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.

**3)** insert(): Inserting a new key takes O(Logn) time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
**4)** delete(): Deleting a key also takes O(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove key.

## Insertion into a Max/Min Heap

The root element is the smallest element in the **min-max heap**. Let be any node in a **min-max heap**. If is on a **min** (or even) level, then is the minimum key among all keys in the subtree with root . If is on a **max**(or odd) level, then is the **maximum** key among all keys in the subtree with root

## Insertion into Min/Max heap

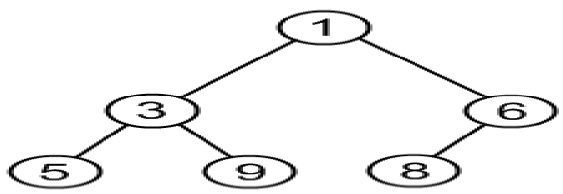
We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

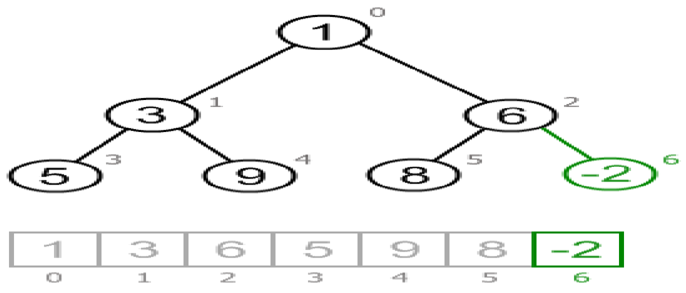
**Note** − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

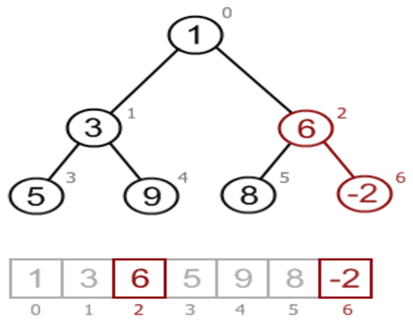
In this article we examine the idea laying in the foundation of the heap data structure. We call it sifting, but you also may meet another terms, like "trickle", "heapify", "bubble" or "percolate".



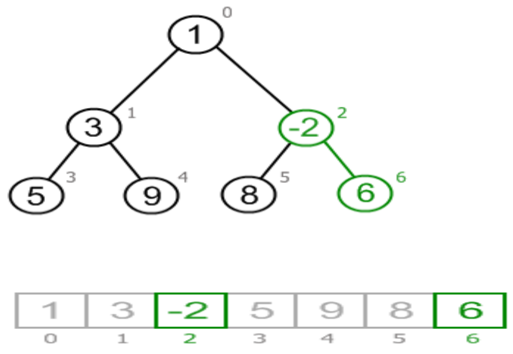
Insert a new element to the end of the array:



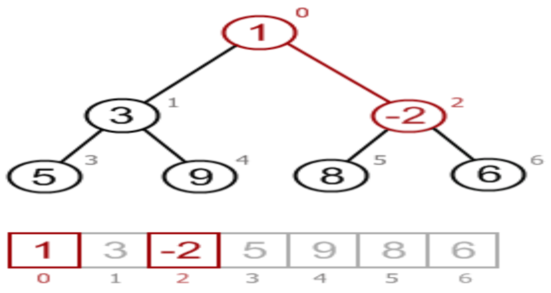
In the general case, after insertion, heap property near the new node is broken:



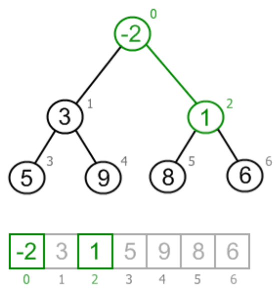
To restore heap property, algorithm *sifts up* the new element, by swapping it with its parent:



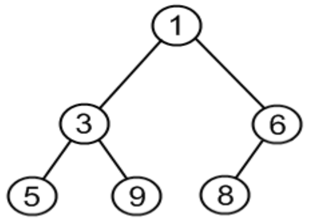
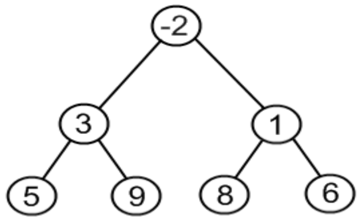
Now heap property is broken at the root node:



Keep sifting:



Heap property is fulfilled, sifting is over.

| Source heap | After -2 insertion |
|---|---|
|  |  |

### 4.1.1.2 **Lecture-66:** Deletion from a Max/Min Heap

## Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

## **Removing the minimum from a heap**

Removal operation uses the same idea as was used for insertion. Root's value, which is minimal by the heap property, is replaced by the last array's value. Then new value is *sifted down*, until it takes right position.

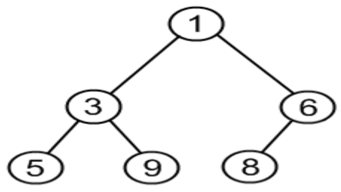
### **Removal algorithm**

1. Copy the last value in the array to the root;
2. Decrease heap's size by 1;

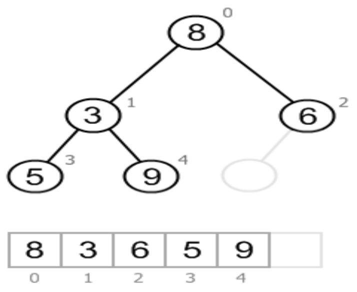
3. Sift down root's value. Sifting is done as following:
    - if current node has no children, sifting is over;
    - if current node has one child: check, if heap property is broken, then swap current node's value and child value; sift down the child;
    - if current node has two children: find the smallest of them. If heap property is broken, then swap current node's value and selected child value; sift down the child.

**Example:**

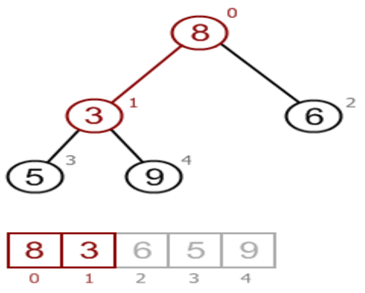
Remove the minimum from a following heap:



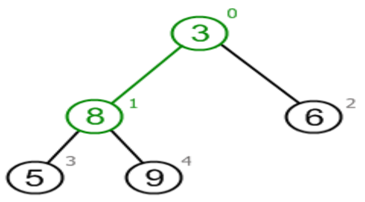
Copy the last value in the array to the root and decrease heap's size by 1:

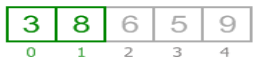


Now heap property is broken at root:

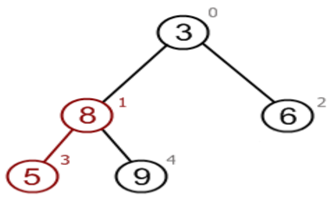


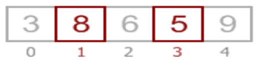
Root has two children. Swap root's value with the smallest:



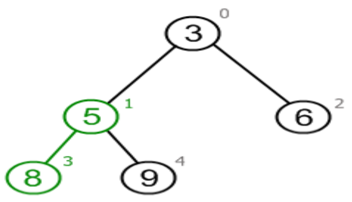


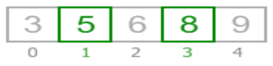
Heap property is broken in node 1:



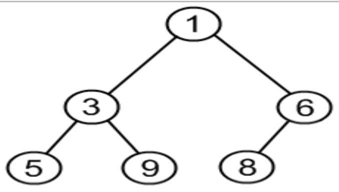
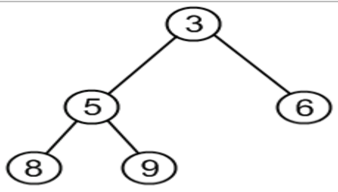


Recover heap property:





Node 3 has no children. Sifting is complete.

| Source heap | After minimum removal |
|---|---|
|  |  |

## Complexity analysis

Complexity of the removal operation is O(h) = O(log n), where **h** is heap's height, **n** is number of elements in a heap.