Binary Search Tree

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

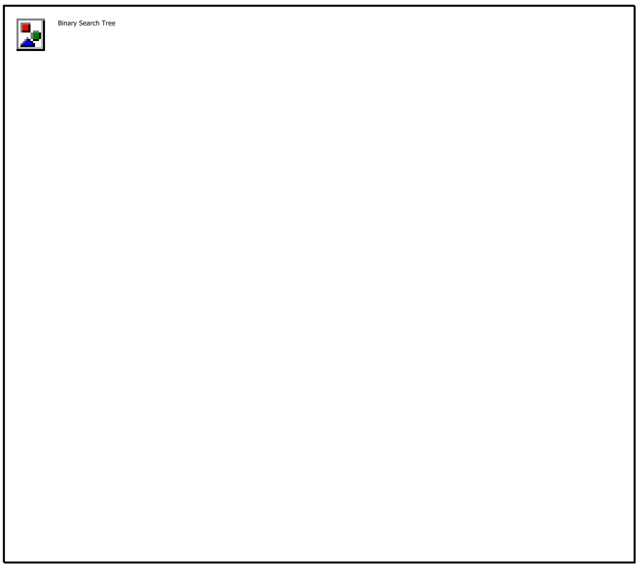
A binary tree has the following time complexities...

- **Search Operation - O(n)**
- **Insertion Operation - O(1)**
- **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows.

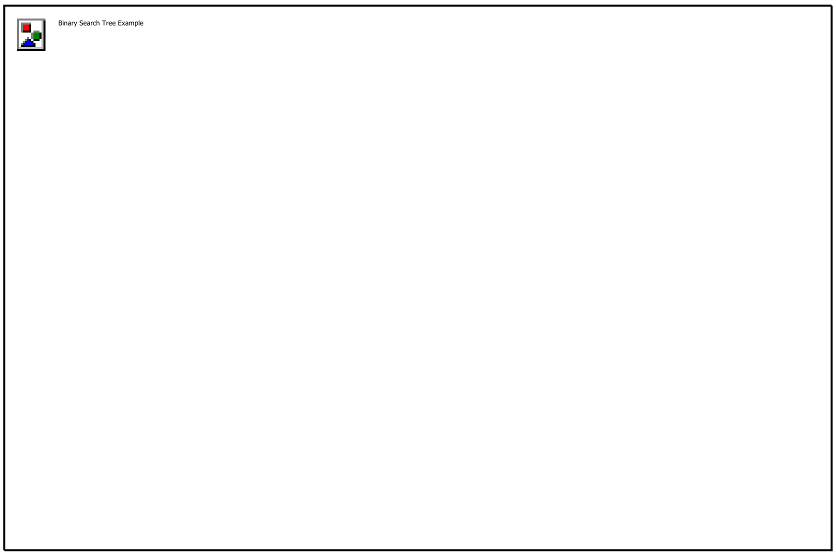
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure.



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

- **Search**
- **Insertion**
- **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Pseudo Code for Searching in BST:

```
search(element, root)
    if !root
        return -1
    if root.value == element
        return 1
    if root.value < element
        search(element, root.right)
    else
        search(element, root.left)
```

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its left and right to NULL.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is Empty, then set root to newNode.

Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Pseudocode for Inserting a Node in BST:

```
insert (element, root)
   Node x = root
   Node y = NULL
   while x:
        y = x
   if x.value < element.value
      x = x.right
   else
      x = x.left
   if y.value < element
        y.right = element
```

```
    else
        y.left = element
```

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.

Pseudo Code for Deleting a Node:

```
delete (value, root):
   Node x = root
   Node y = NULL
# searching the node
   while x:
         y = x
   if x.value < value
      x = x.right
   else if x.value > value
      x = x.left
   else if value == x
      break
# if the node is not null, then replace it with successor
   if y.left or y.right:
         newNode = GetInOrderSuccessor(y)
         root.value = newNode.value
# After copying the value of successor to the root #we're deleting the successor
         free(newNode)
```

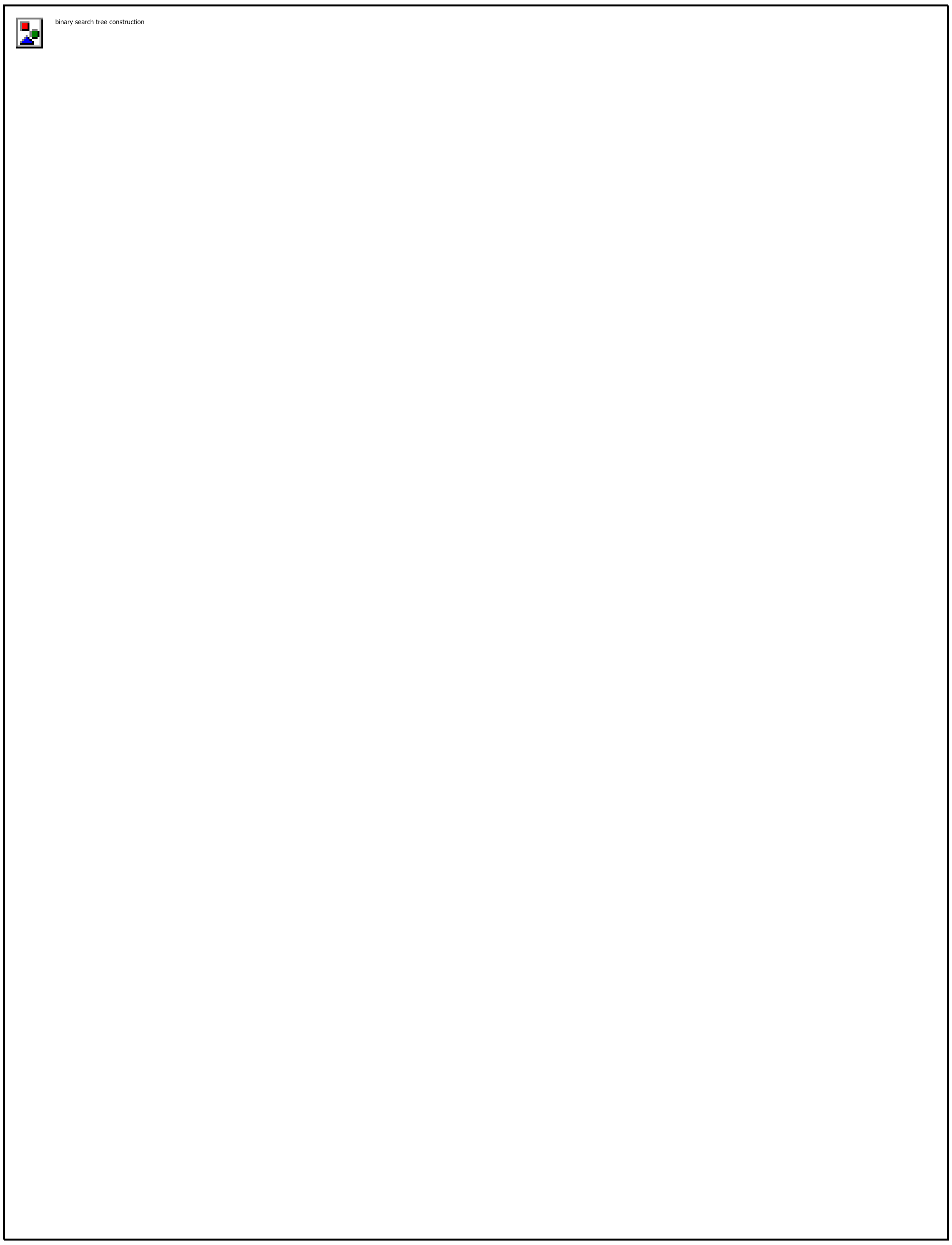
```
else
    free(y)
```

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...



B - Tree

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows...

B-Tree is a self-balanced search tree in which every node contains multiple keys and hasmore than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties...

Property #1 - All leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

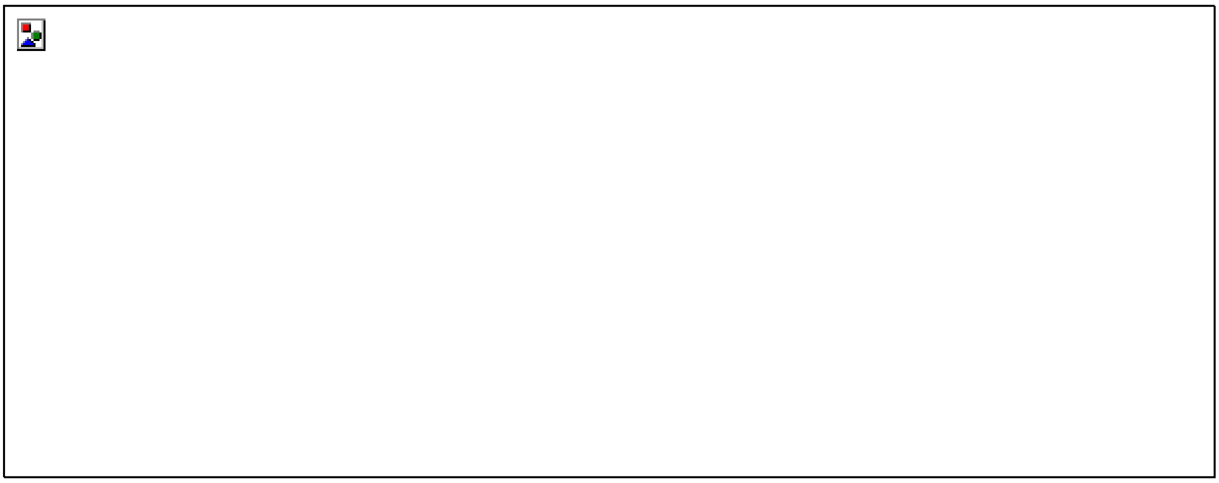
Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

Property #4 - If the root node is a non leaf node, then it must have atleast 2 children.

Property #5 - A non leaf node with n-1 keys must have n number of children.

Property #6 - All the key values in a node must be in Ascending Order.

For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.



Operations on a B-Tree

The following operations are performed on a B-Tree...

- **Search**
- **Insertion**
- **Deletion**

Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with O(log n) time complexity.

The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

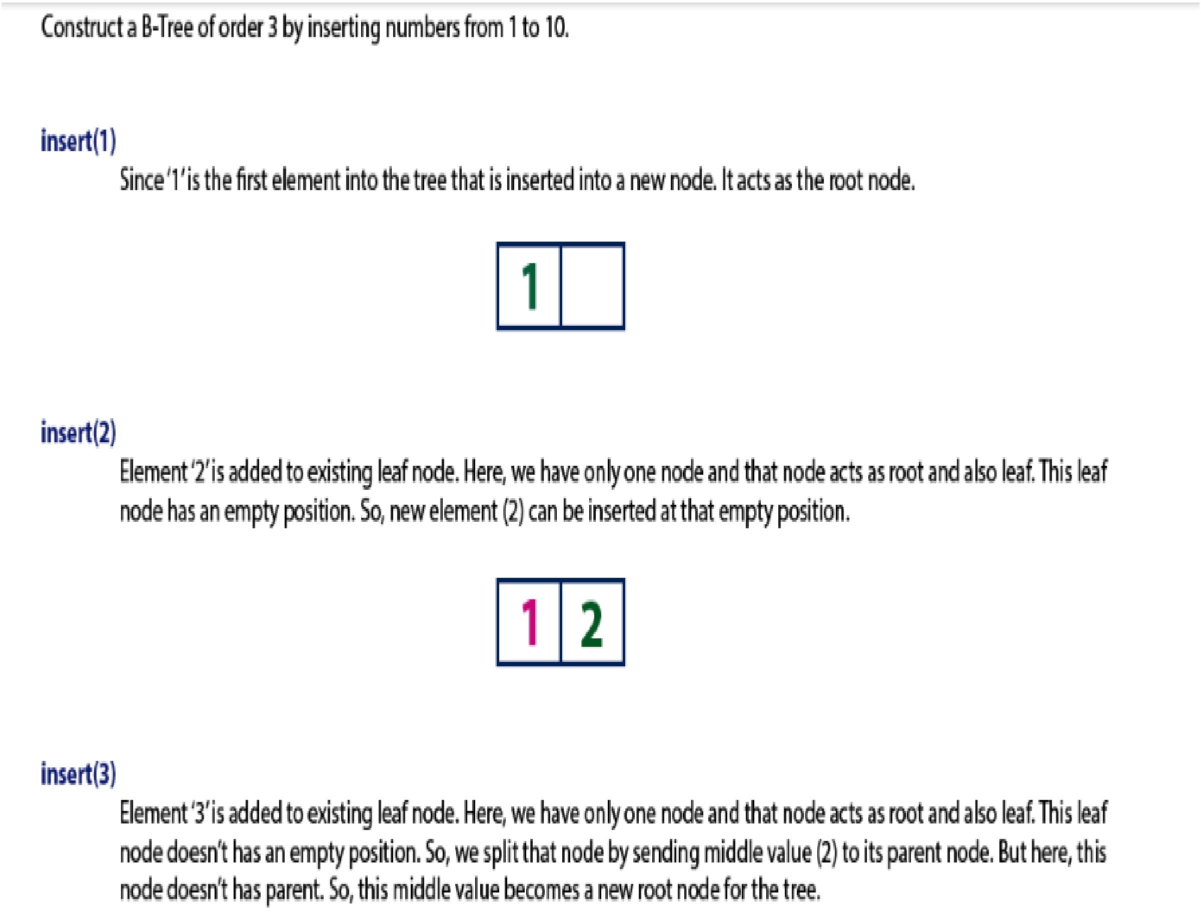
Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.



Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

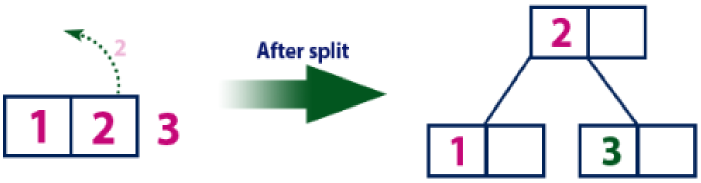
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

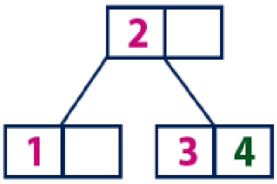
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



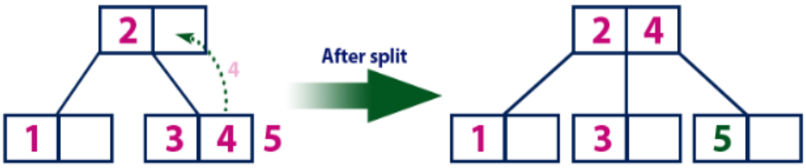
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



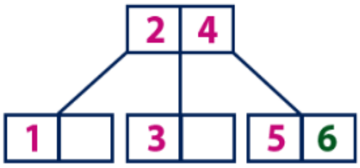
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



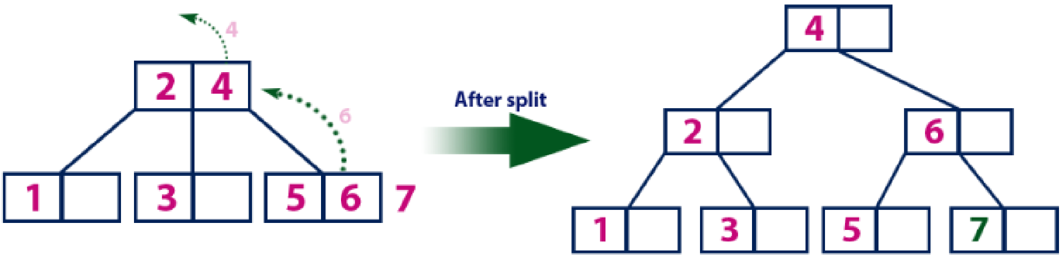
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



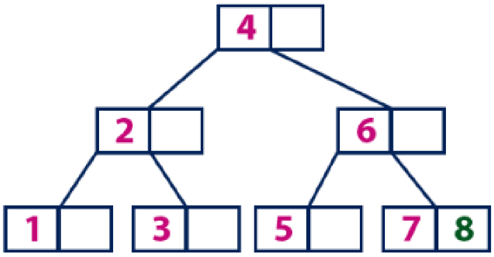
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



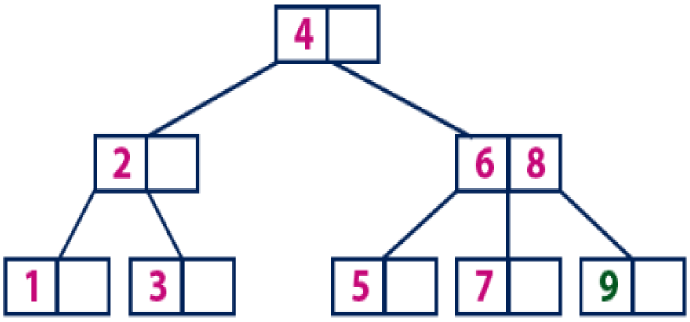
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



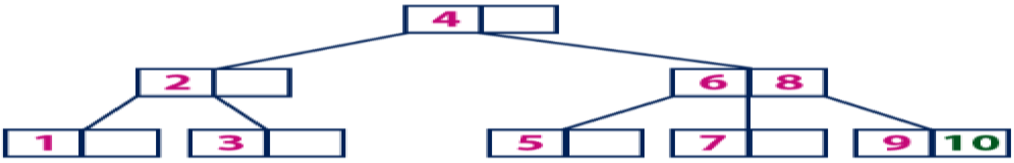
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Red - Black Tree

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

Property #1: Red - Black Tree must be a Binary Search Tree.

Property #2: The ROOT node must be colored BLACK.

Property #3: The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).

Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.

Property #5: Every new node must be inserted with RED color.

Property #6: Every leaf (e.i. NULL node) must be colored BLACK.

Example

Following is a Red-Black Tree which is created by inserting numbers from 1 to 9.

The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

- **Recolor**
- **Rotation**
- **Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

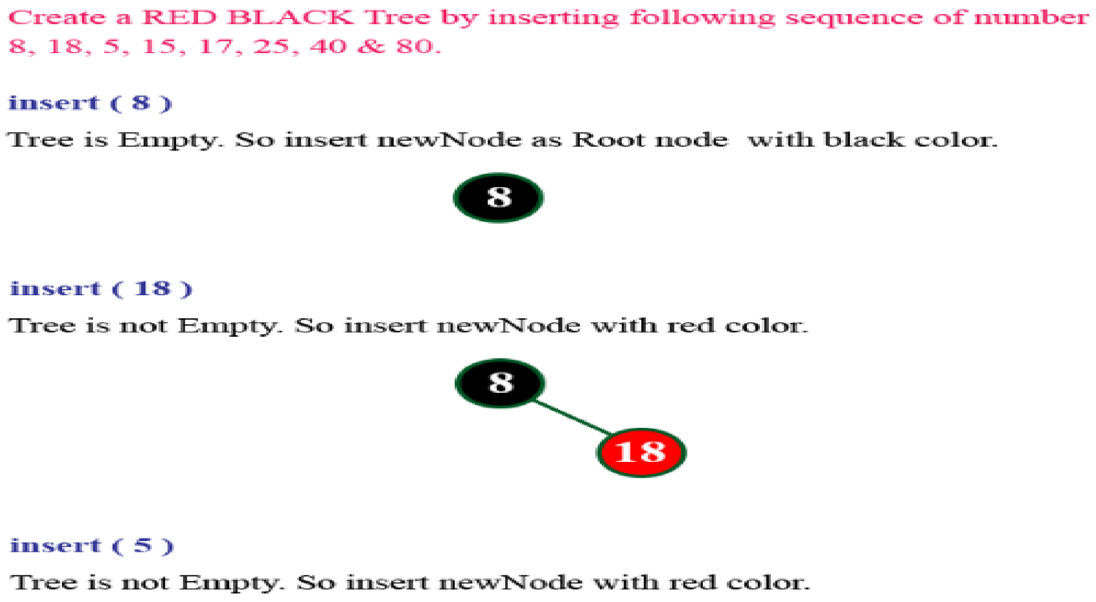
Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Example



Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

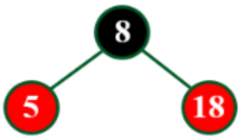
Tree is Empty. So insert newNode as Root node with black color.

insert (18)

Tree is not Empty. So insert newNode with red color.

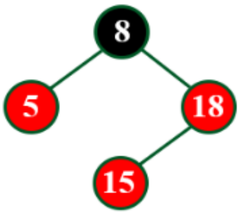
insert (5)

Tree is not Empty. So insert newNode with red color.



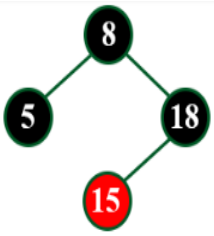
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

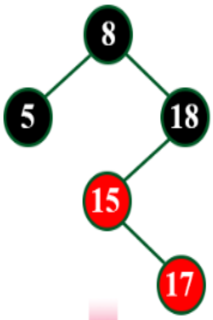
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

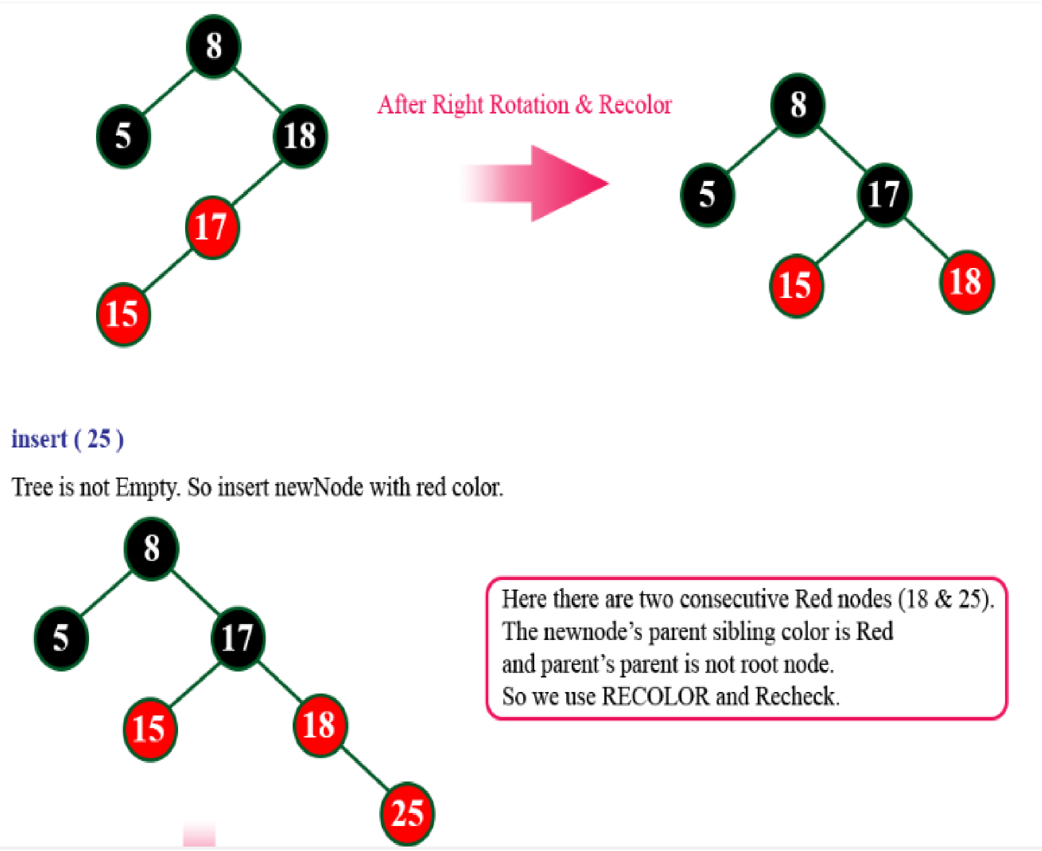
insert (17)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

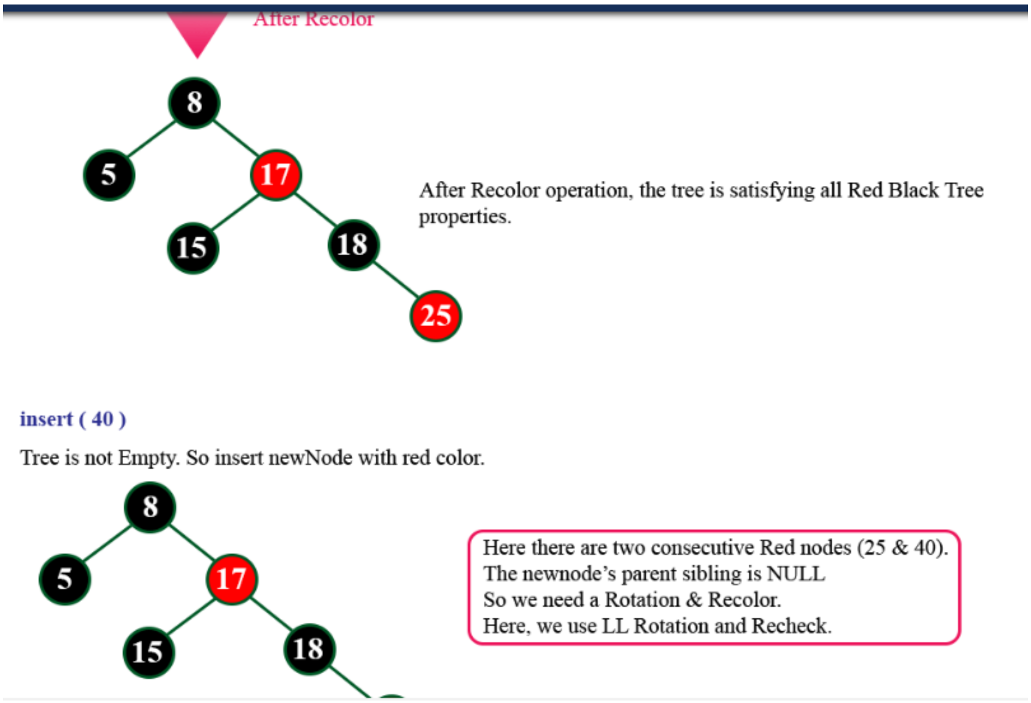
After Left Rotation



insert (25)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

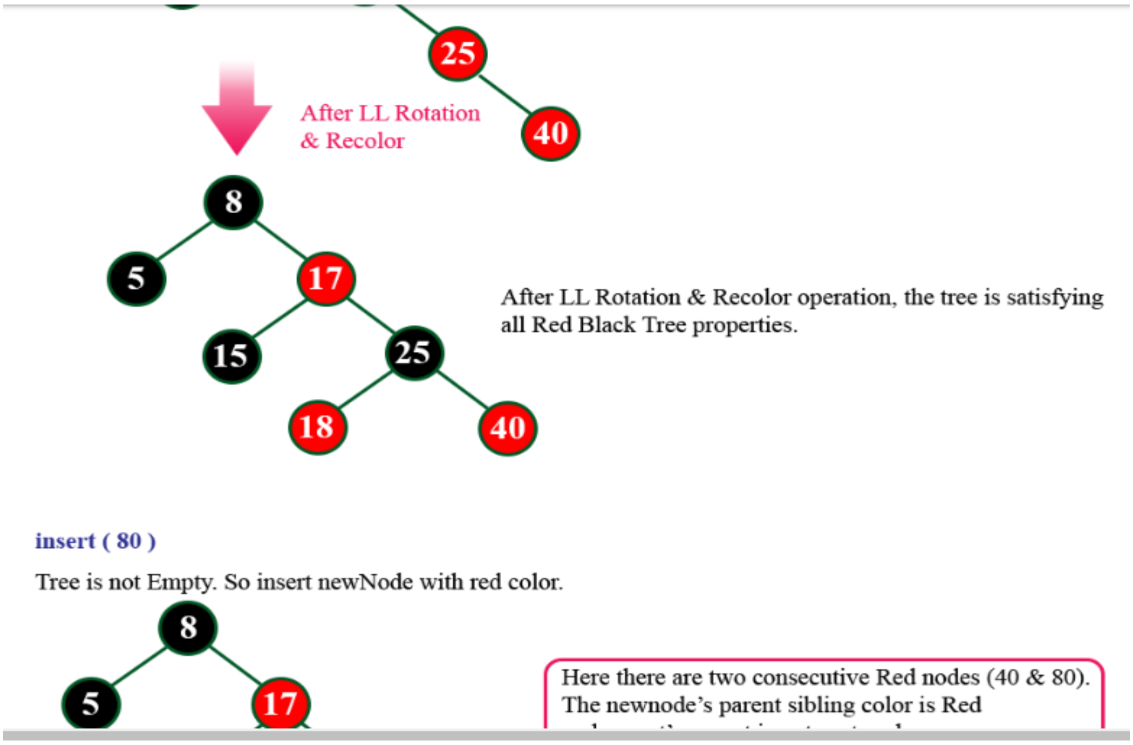


After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.



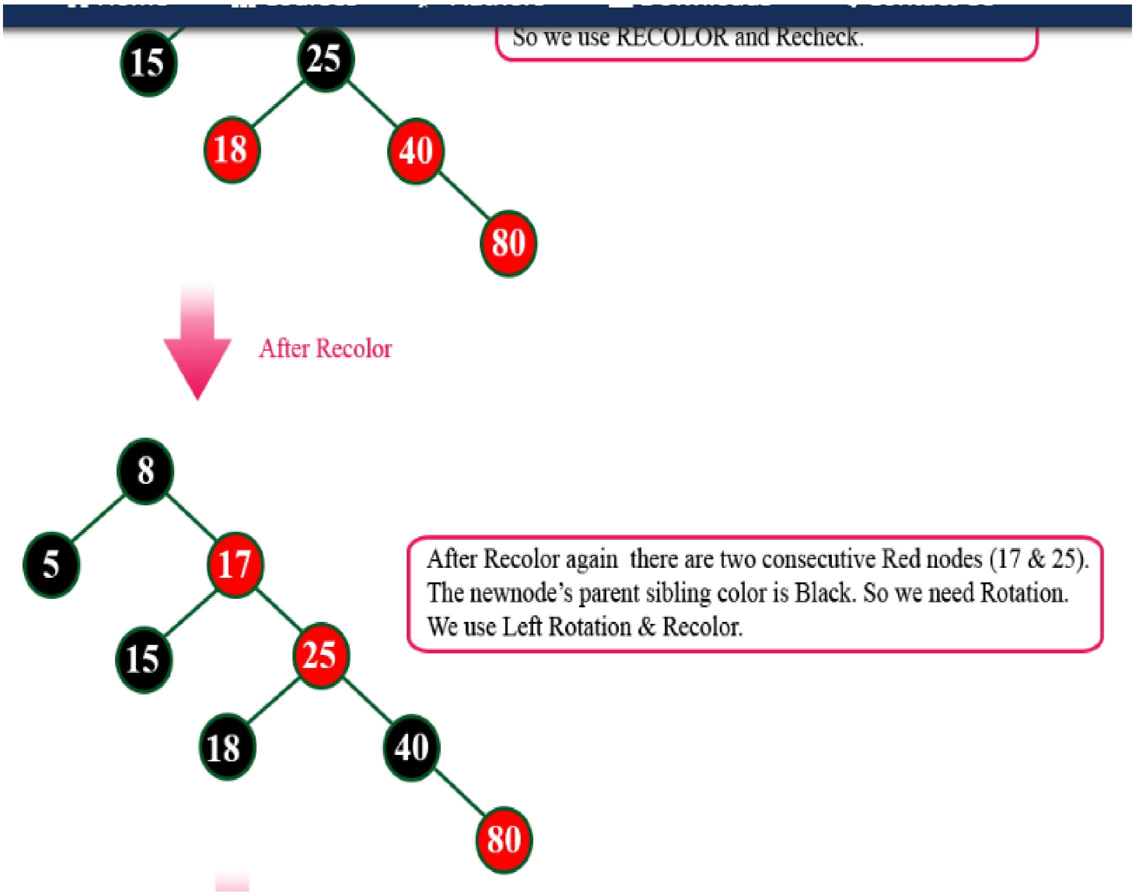
After LL Rotation & Recolor

After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

Tree is not Empty. So insert newNode with red color.

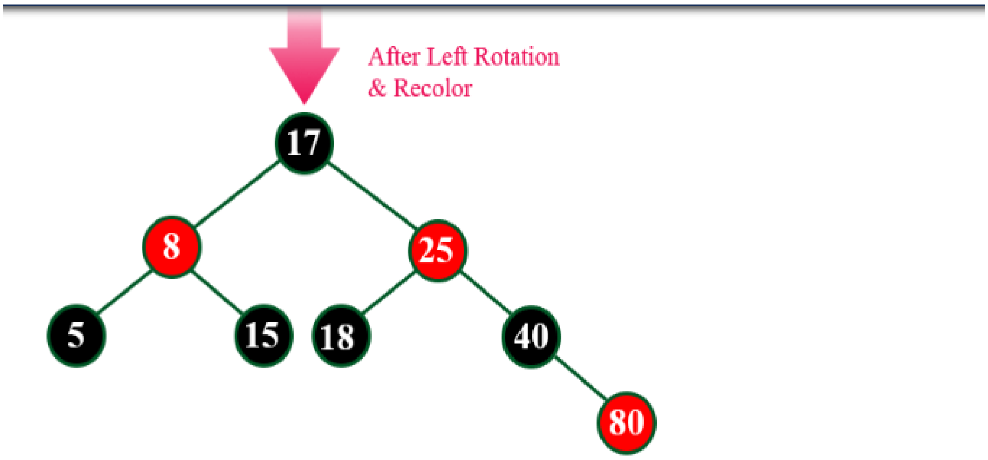
Here there are two consecutive Red nodes (40 & 80). The newnode's parent sibling color is Red



So we use RECOLOR and Recheck.

After Recolor

After Recolor again there are two consecutive Red nodes (17 & 25). The newnode's parent sibling color is Black. So we need Rotation. We use Left Rotation & Recolor.



Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Max Heap

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure some times also called as Binary Heap.

There are two types of heap data structures and they are as follows...

- **Max Heap**
- **Min Heap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

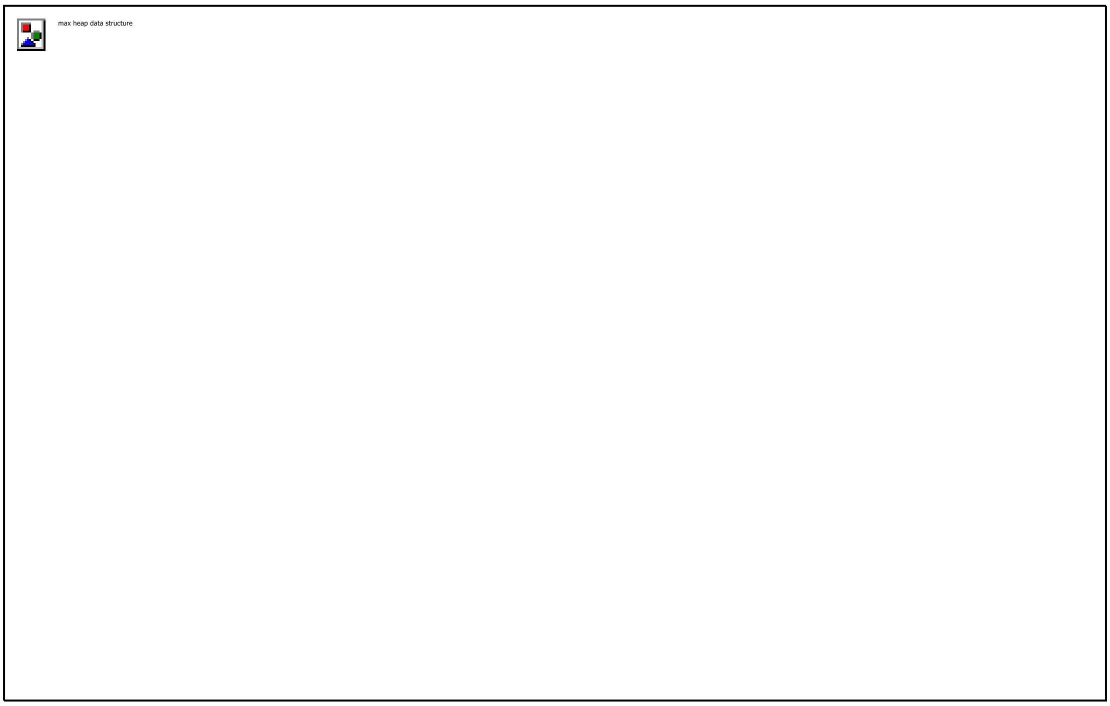
Max Heap

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

- **Finding Maximum**
- **Insertion**
- **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

Step 1 - Insert the newNode as last leaf from left to right.

Step 2 - Compare newNode value with its Parent node.

Step 3 - If newNode value is greater than its parent, then swap both of them.

Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

Example

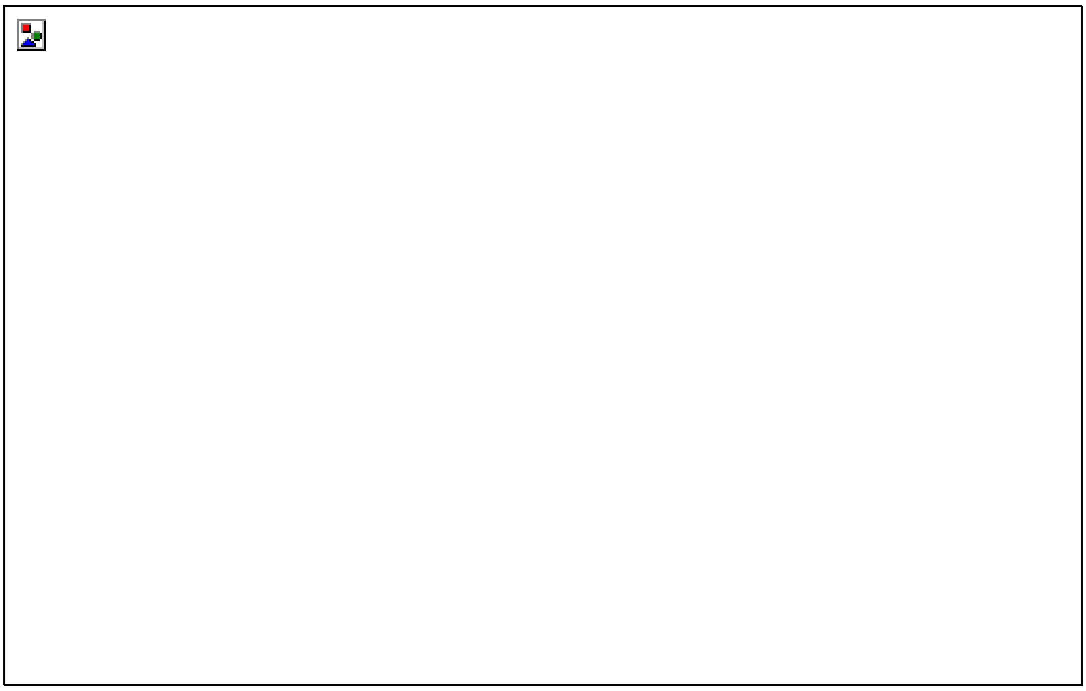
Consider the above max heap. Insert a new node with value 85.

Step 1 - Insert the newNode with value 85 as last leaf from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...

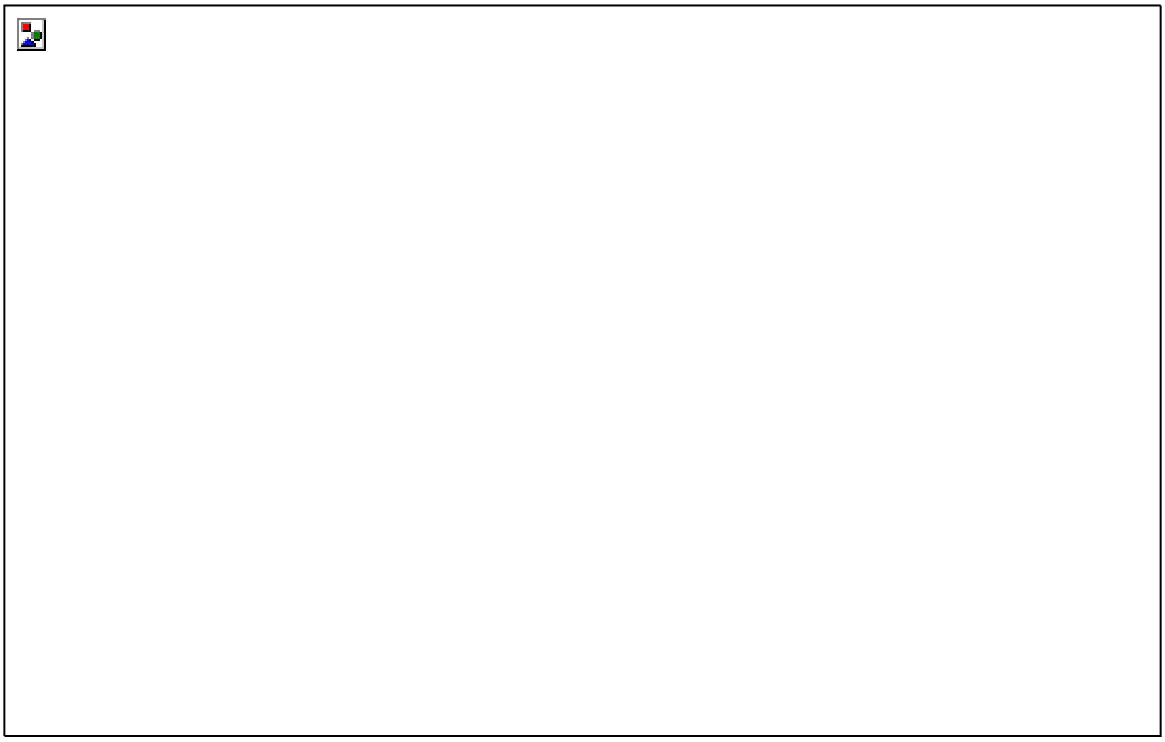
Step 2 - Compare newNode value (85) with its Parent node value (75). That means 85 > 75

Step 3 - Here newNode value (85) is greater than its parent value (75), then swap both of them. After swapping, max heap is as follows...

Step 4 - Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

Step 1 - Swap the root node with last node in max heap

Step 2 - Delete last node.

Step 3 - Now, compare root value with its left child value.

Step 4 - If root value is smaller than its left child, then compare left child with its right sibling. Else goto Step 6

Step 5 - If left child value is larger than its right sibling, then swap root with left child otherwise swap root with its right child.

Step 6 - If root value is larger than its left child, then compare root value with its right child value.

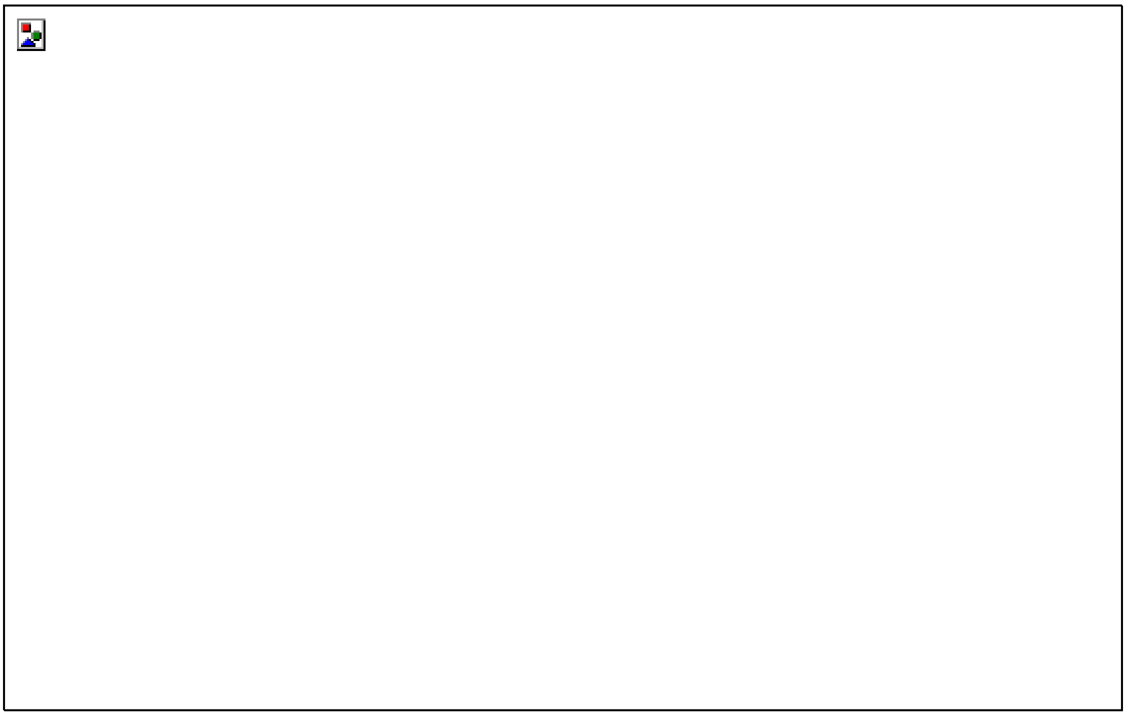
Step 7 - If root value is smaller than its right child, then swap root with right child otherwise stop the process.

Step 8 - Repeat the same until root node fixes at its exact position.

Example

Consider the above max heap. Delete root node (90) from the max heap.

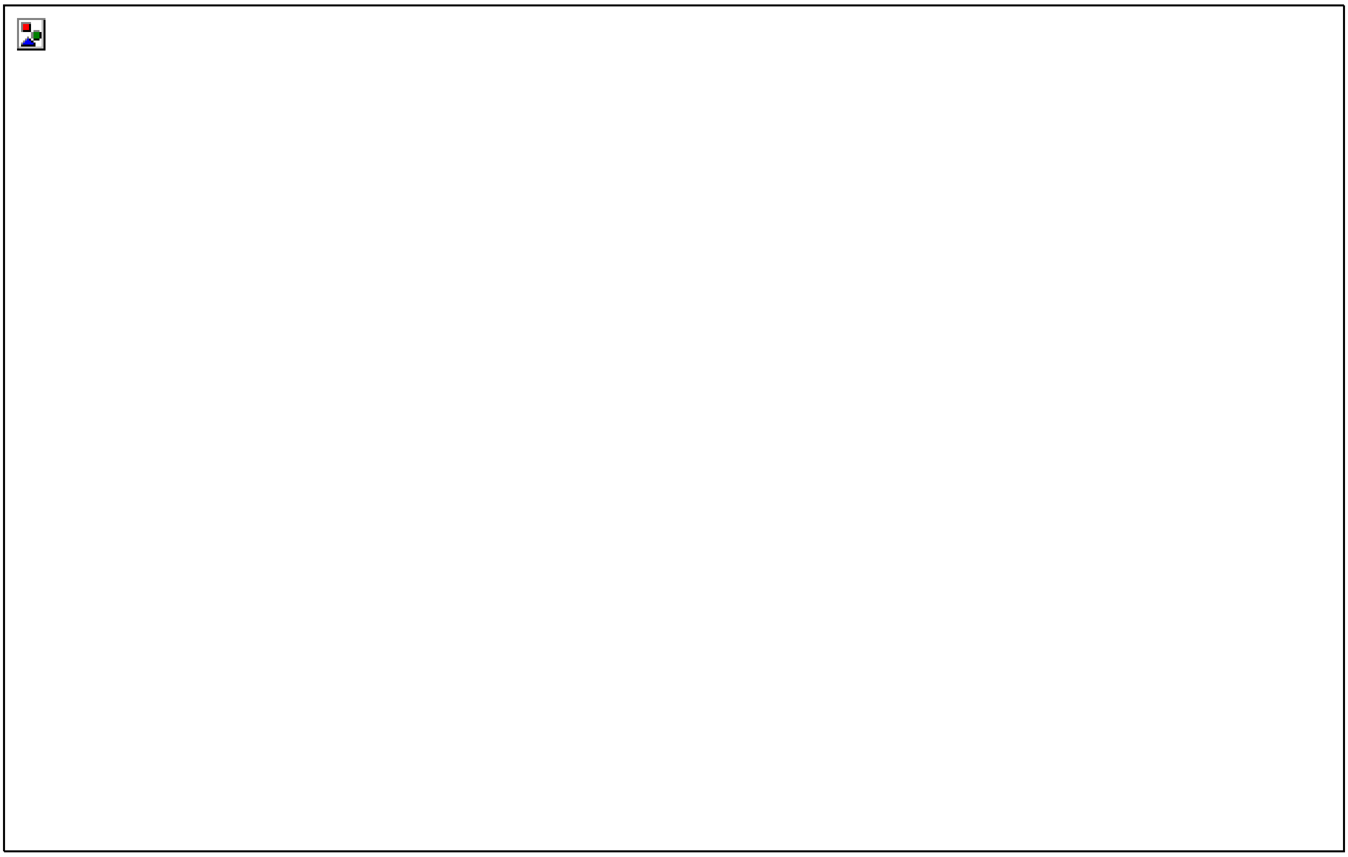
Step 1 - Swap the root node (90) with last node 75 in max heap. After swapping max heap is as follows...



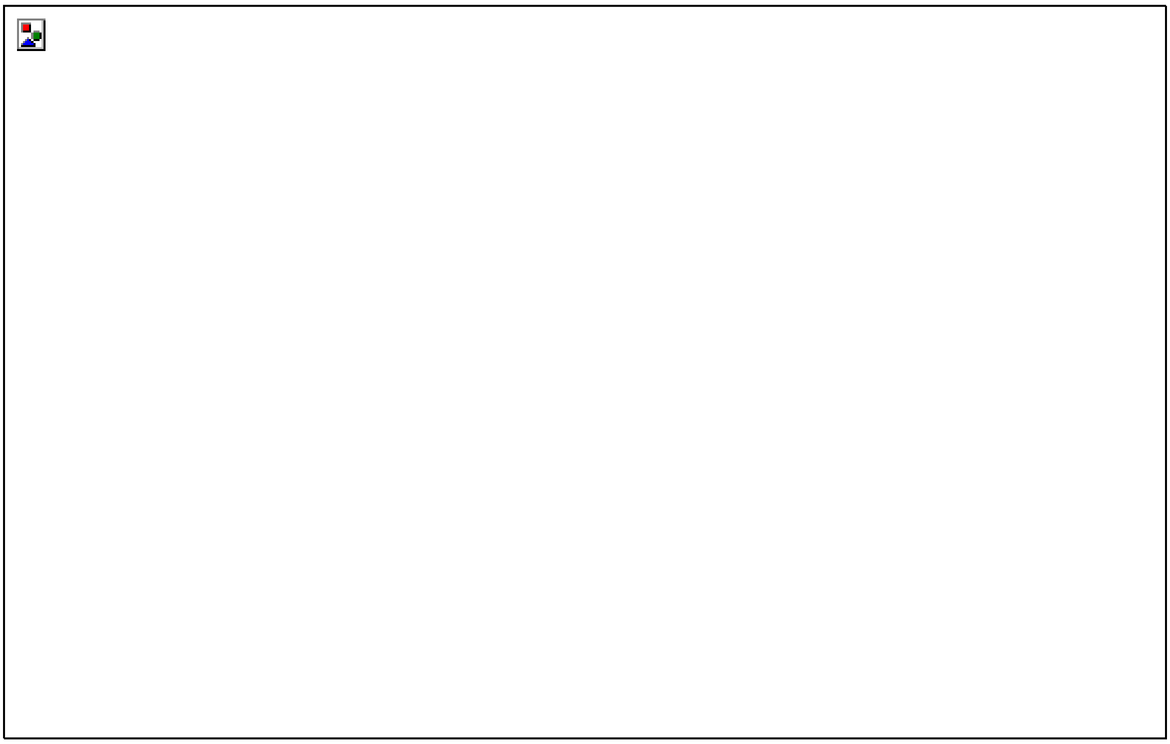
Step 2 - Delete last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...

Step 3 - Compare root node (75) with its left child (89).

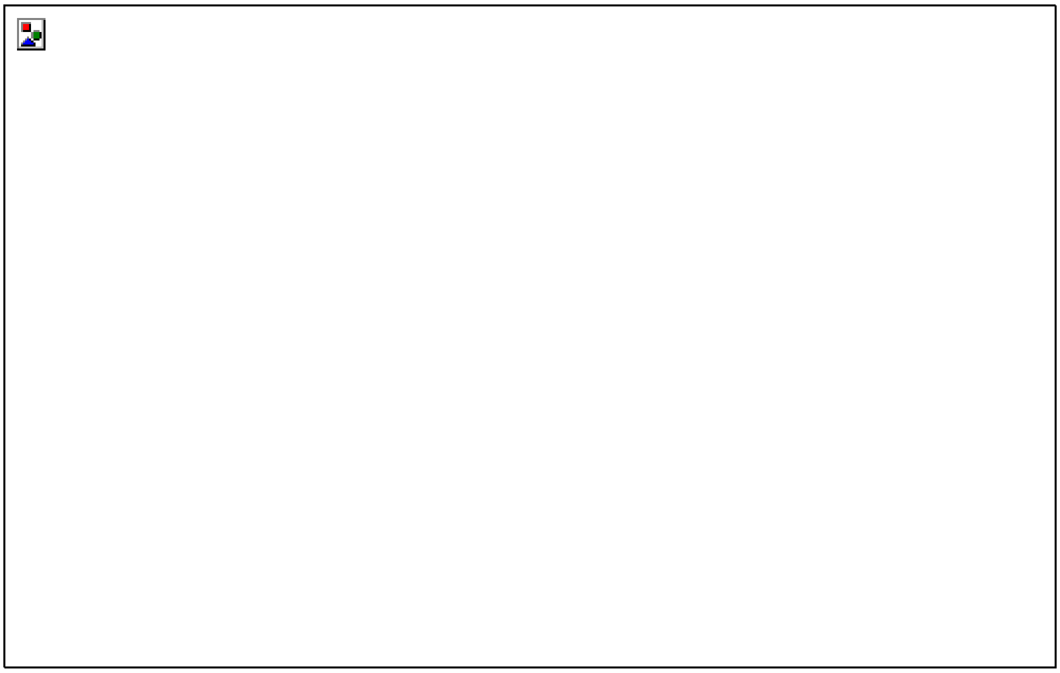
Here, root value (75) is smaller than its left child value (89). So, compare left child (89) with its right sibling (70).



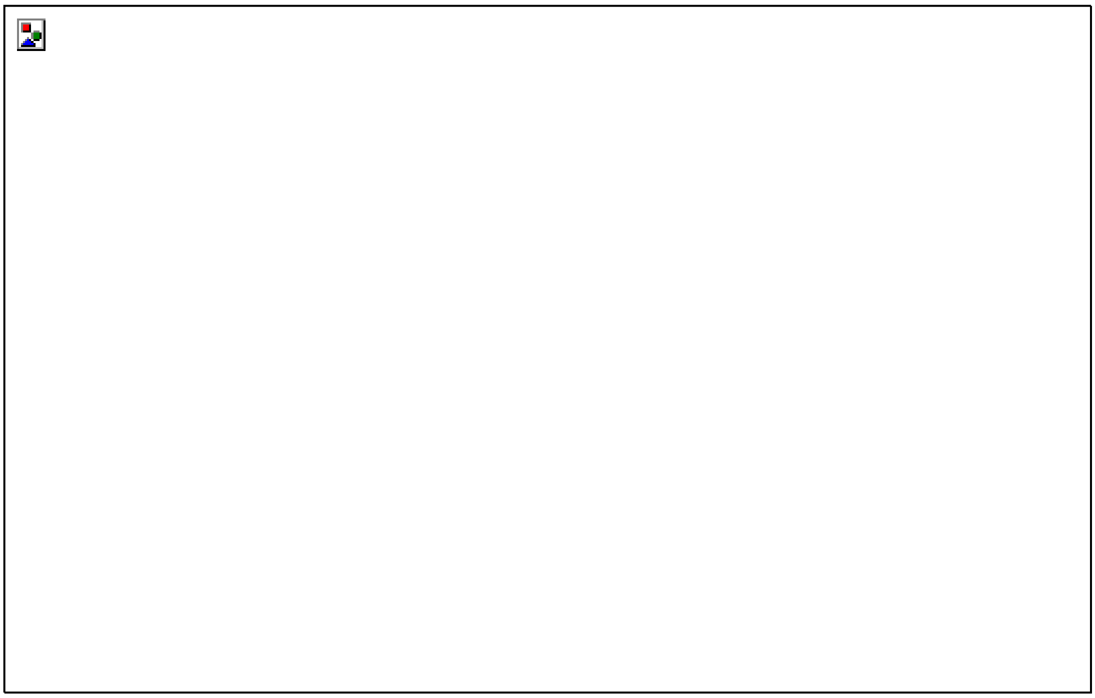
Step 4 - Here, left child value (89) is larger than its right sibling (70), So, swap root (75) with left child (89).



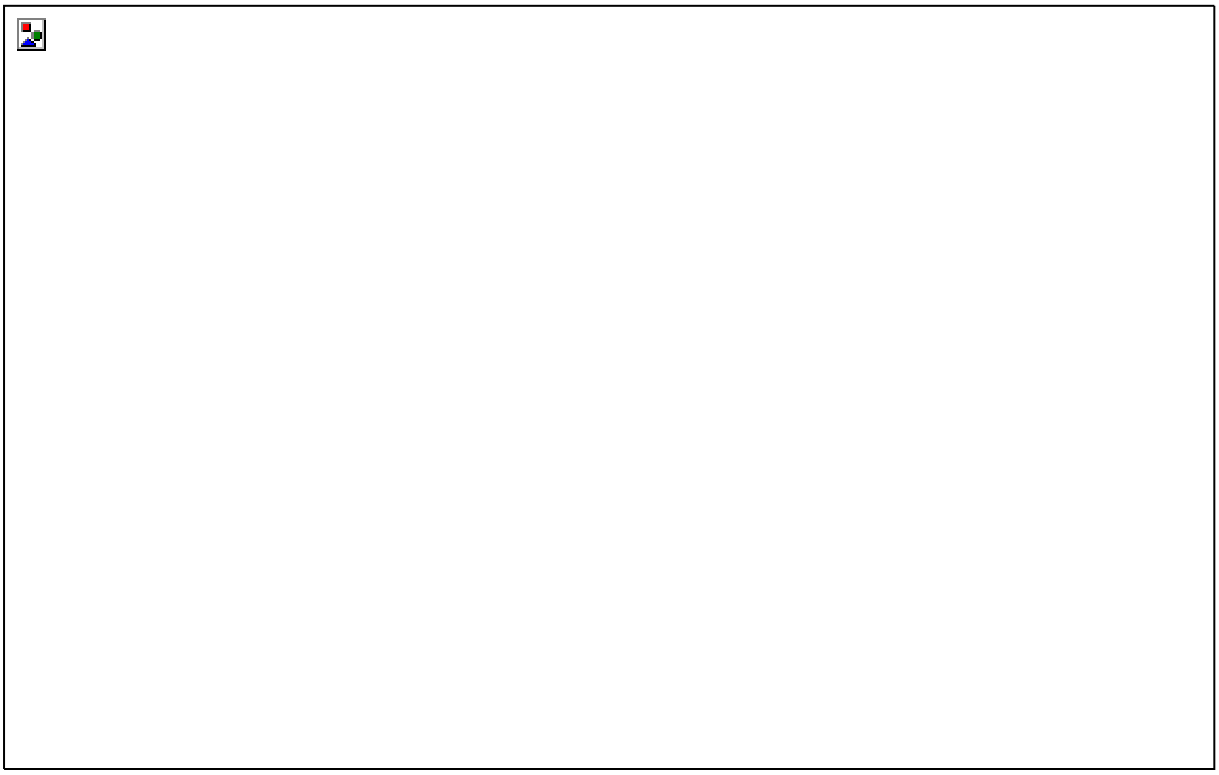
Step 5 - Now, again compare 75 with its left child (36).



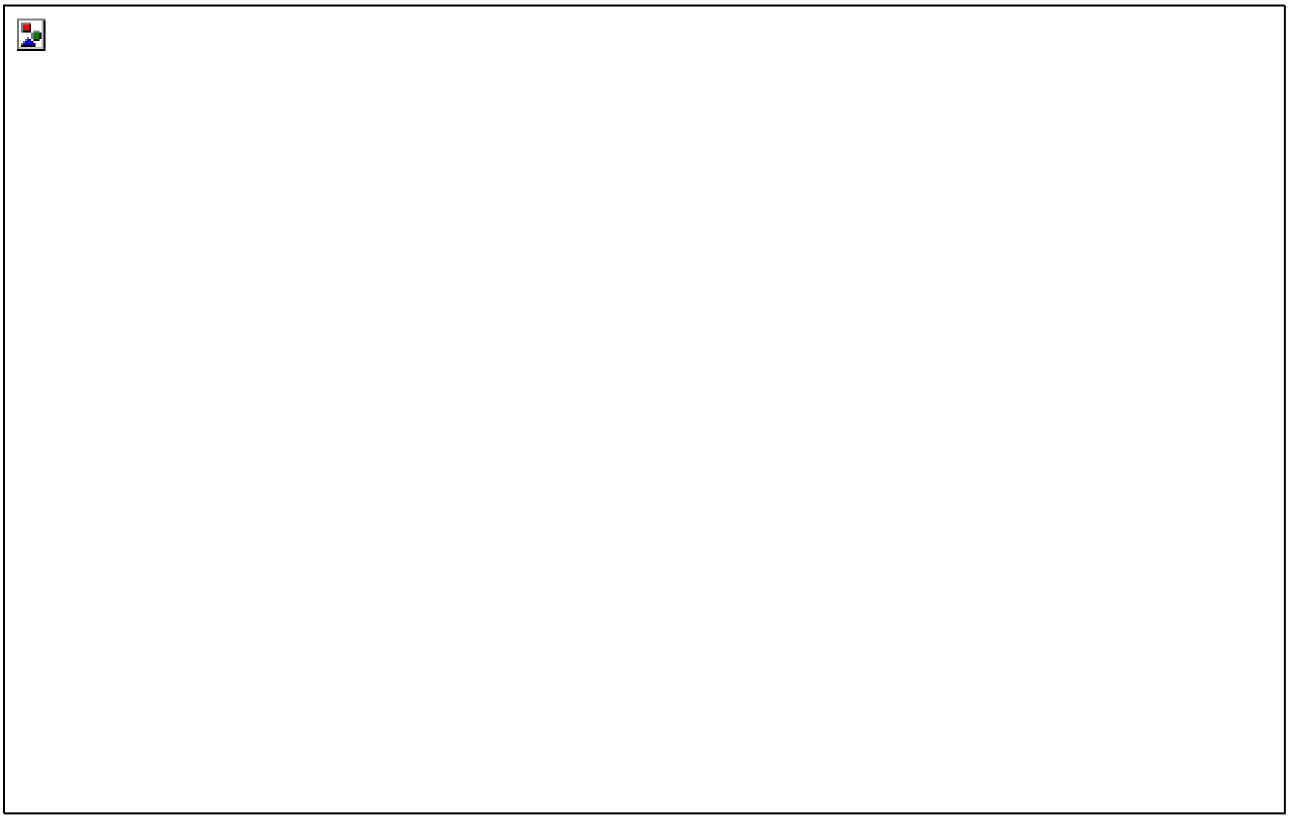
Here, node with value 75 is larger than its left child. So, we compare node 75 with its right child 85.



Step 6 - Here, node with value 75 is smaller than its right child (85). So, we swap both of them. After swapping max heap is as follows...



Step 7 - Now, compare node with value 75 with its left child (15).



Here, node with value 75 is larger than its left child (15) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (90) is as follows...