Topics to cover:

1. Heap in Java
2. Heap sort

1. Heap

What is heap?

In Java, Heap is a special type of data structure where the root node or parent node is compared with its left and right children and arranged according to the order.

Based on the order of the parent and child nodes, Heap can be classified in two forms, i.e., Min heap and Max heap. Let's understand both of them one by one and implement the code in Java.

Min heap

Min heap is a special type of heap data structure that is a complete binary tree in itself . Min heap has the following properties:

1. Root node value is always smaller in comparison to the other nodes of the heap.
2. Each internal node has a key value that is always smaller or equal to its children.

The most three important operations in Min heap are as follows:

Inserting node:

We can perform insertion in the Min heap by adding a new key at the end of the tree. If the value of the inserted key is smaller than its parent node, we have to traverse the key upwards for fulfilling the heap property. The insertion process takes O(log n) time.

Extracting and deleting minimum value:

It is one of the most important operations which we perform to remove the minimum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained. This operation takes O(Logn) time to remove the minimum element from the heap.

Retrieving minimum value:

This operation is used to get the root node of the heap, i.e., minimum element in O(1) time.

Max heap

Max heap is another special type of heap data structure that is also a complete binary tree in itself in Java. Max heap has the following properties:

1. Root node value is always greater in comparison to the other nodes of the heap.
2. Each internal node has a key value that is always greater or equal to its children.

The most three important operations in Max heap are as follows:

Inserting node:

We can perform insertion in the Max heap by adding a new key at the end of the tree. If the value of the inserted key is greater than its parent node, we have to traverse the key upwards for fulfilling the heap property. The insertion process takes O(log n) time.

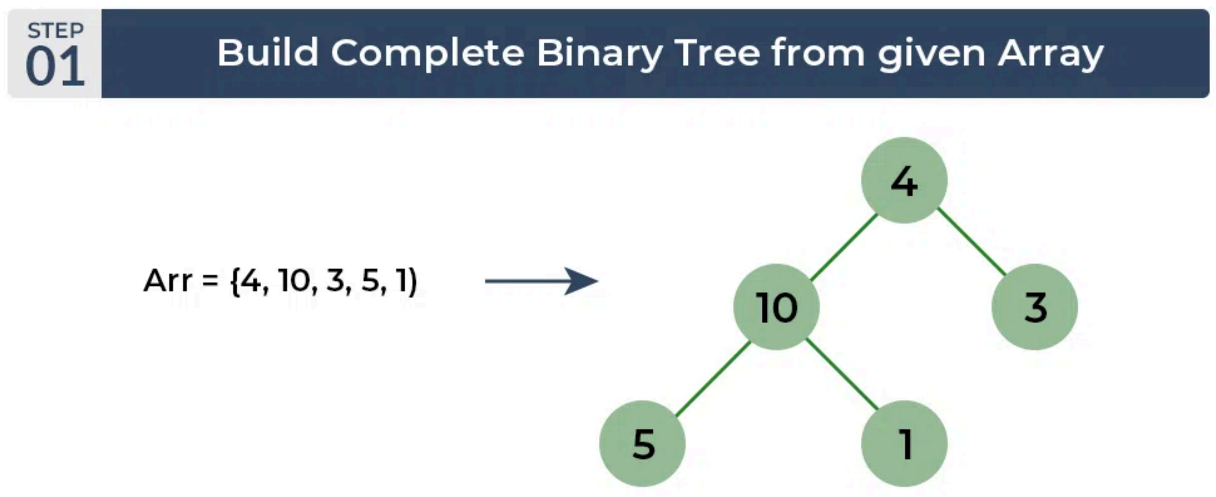
Extracting and deleting maximum value:

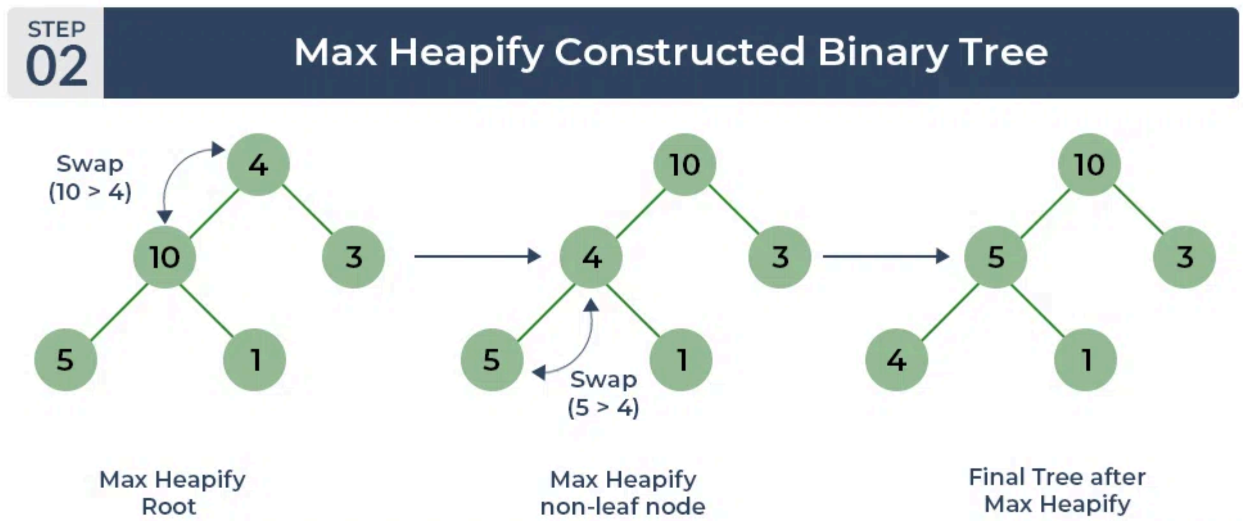
It is one of the most important operations which we perform to remove the maximum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained. This operation takes O(Log n) time to remove the maximum element from the heap.

Retrieving maximum value:

This operation is used to get the root node of the heap, i.e., maximum element in O(1) time.

Building Max Heap from an arrary:

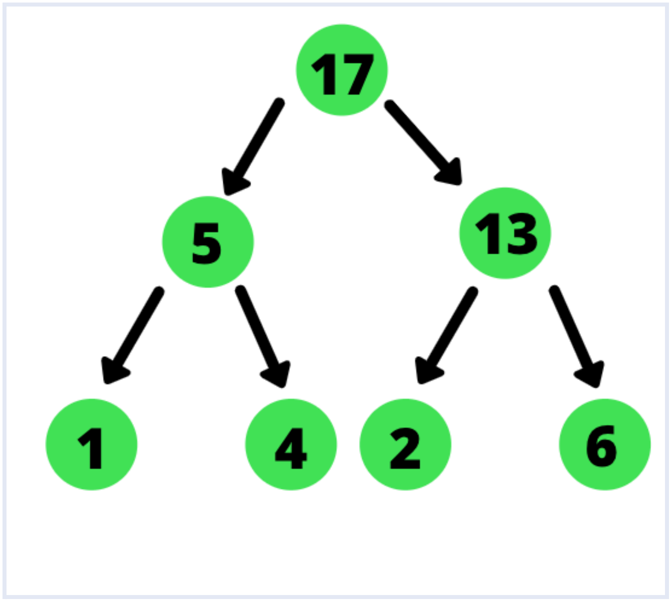



STEP 02
Max Heapify Constructed Binary Tree
Swap (10 > 4)
4
10
3
5
1
Max Heapify Root
10
4
3
5
1
Swap (5 > 4)
Max Heapify non-leaf node
10
5
3
4
1
Final Tree after Max Heapify

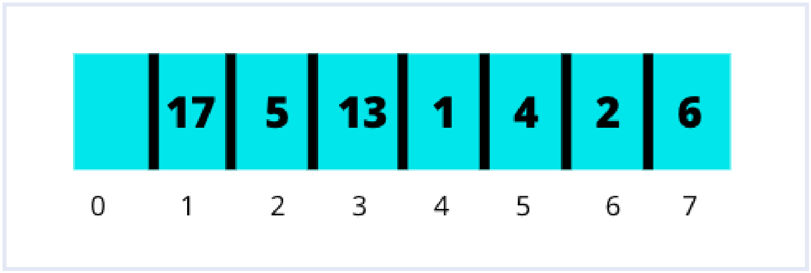
Implementation of a Max Heap Data Structure in Java

We represent a heap using an array. Since the heap is a complete binary tree, there is no wastage of space.

For example, let’s consider a heap as follows :



The array representation is:



The declaration of max heap is done as follows:

```
static class MaxHeap {
    private int[] Heap; // array
    private int size;
    private int maxsize;

    public MaxHeap(int size) {
      this.maxsize = size;
      this.size = 0;
      Heap = new int[this.maxsize + 1];
      Heap[0] = Integer.MAX_VALUE;
    }
```

Heap is the array that stores the max heap. The constructor takes the size and initializes the array with 0th element as infinity. We start our heap from index 1.

1. Getting parent of a node

Since we are storing the heap as an array, getting the parent for a node becomes easier.

For an element at position i, the position of its parent is given by :

(i)/2

During implementation we can get the parent using :

```
private int parent(int pos) {
        return pos / 2;
    }
```

2. Getting children for the node

For a node at position i, its children are given by the formula :

Left child :

(2*i)

Right child :

(2*i)+ 1

*Note : This is true when your heap is starting from index 1. If the heap is starting at position 0, the values are (2*i) +1 and (2*i) +2 for left and right child respectively.*

In code we implement this as follows :

```
private int leftChild(int pos) {
        return (2 * pos) ;
    }
```

```
 private int rightChild(int pos) {
          return (2 * pos) + 1;
      }
```

3. Heapify a newly inserted element

After inserting an element into the heap, it may not satisfy the heap property. In that case, we need to adjust the locations of the heap to make it heap again. This process is called Heapifying.

To heapify an element in a max heap we need to find the maximum of its children and swap it with the current element. We continue this process until the heap property is satisfied at each node.

In order to heapify we move down from the root to the leaves. Hence this is also known as Down Heapify.

Another interesting point to note is that we perform down heapify only on non-leaf nodes.

The code for down-heapify function is:

```
private void downHeapify(int pos) {

if((leftChild(pos)<=size || rightChild(pos)<=size) && size>0) {
          if (Heap[pos] < Heap[leftChild(pos)] ||
                 Heap[pos] < Heap[rightChild(pos)]) {
             if (Heap[leftChild(pos)] > Heap[rightChild(pos)]) {
                 swap(pos, leftChild(pos));
                 downHeapify(leftChild(pos));
             } else {
                 swap(pos, rightChild(pos)); //1,3
                 downHeapify(rightChild(pos)); //dhf=3
             }
          }
          }
```

```
            else {
                    return;
            }
        }
```

The swap function is as follows:

```
private void swap(int fpos, int spos) {
            int tmp;
            tmp = Heap[fpos];
            Heap[fpos] = Heap[spos];
            Heap[spos] = tmp;
        }
```

You can also write the same code using a while loop instead of recursion.

In down-heapify we were moving from parents towards its children. We can move in a bottom-up fashion as well. When we’re moving in a bottom-up fashion, we compare a node to its parents. This is called Up Heapify.

The code for up-heapify is:

```
private void heapifyUp(int pos) {
            int temp = Heap[pos];
            while(pos>0 && temp > Heap[parent(pos)]){
               Heap[pos] = Heap[parent(pos)];
               pos = parent(pos);
            }
            Heap[pos] = temp;
        }
```

We’ve written the code for up heapify using a while loop instead of recursion.

4. Insert new nodes

New element is added to the end of the array and swaps are performed to make sure that the heap property holds.

The algorithm for insertion is:

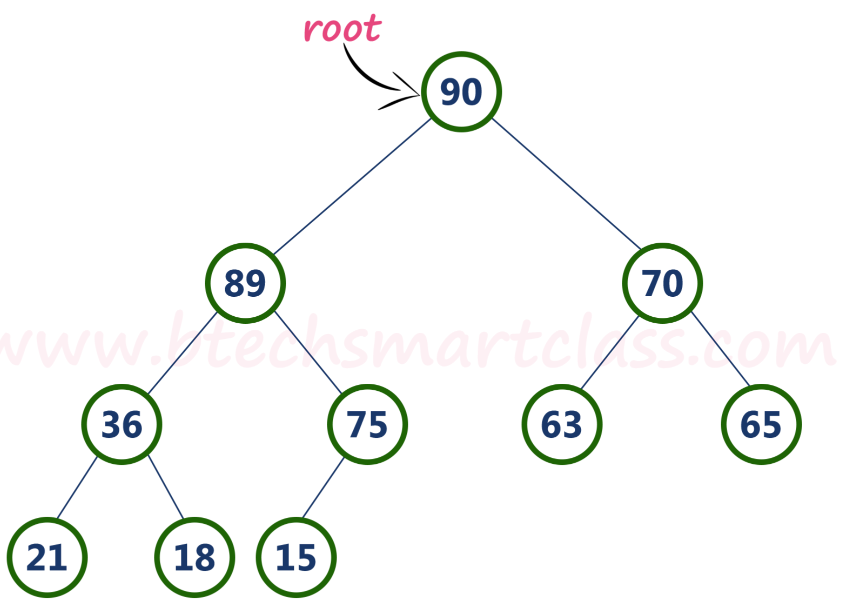
1. Increase the heap size
2. Keep the new element at the end of the heap (array)
3. Heapify from bottom to top

The code for insertion is:

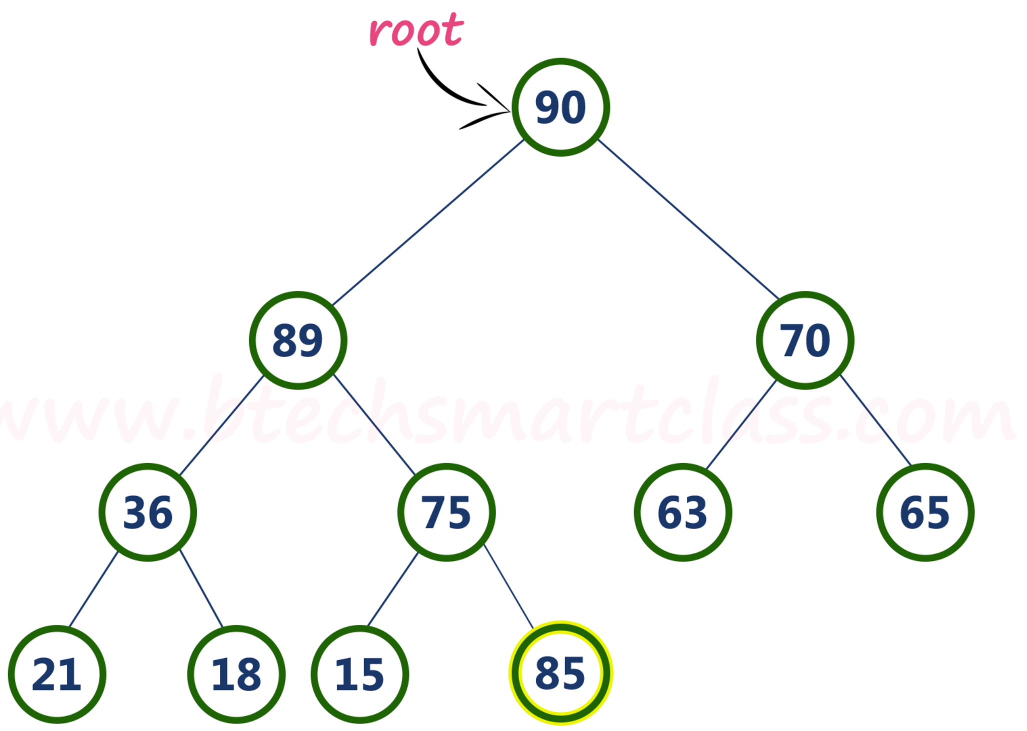
```
public void insert(int element) {     //size=0. element = 5
          Heap[++size] = element;  // heap[1]=5, size=1,  Heap[0 = 5, size=1
          int current = size;
          heapifyUp(current);
      }
```

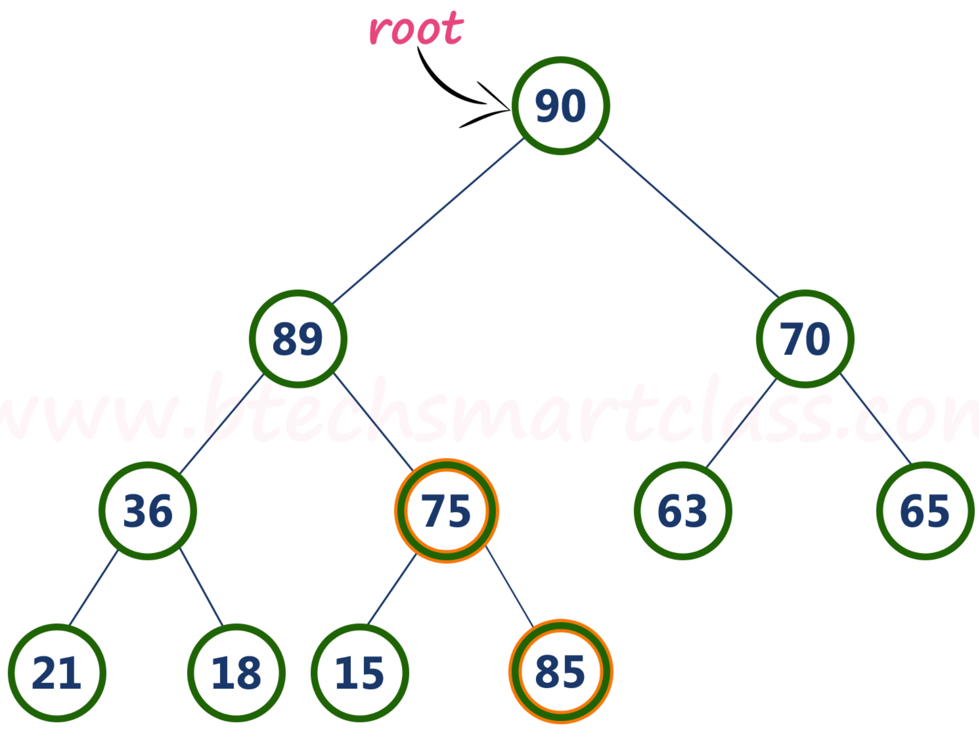
Example:

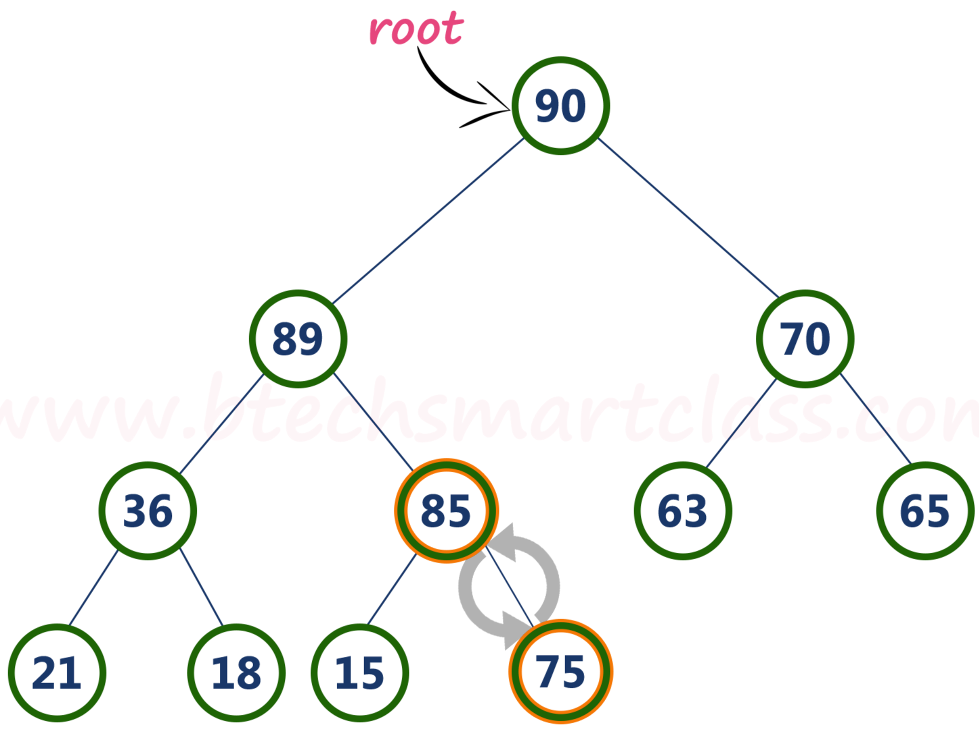
Lets us consider the following max heap

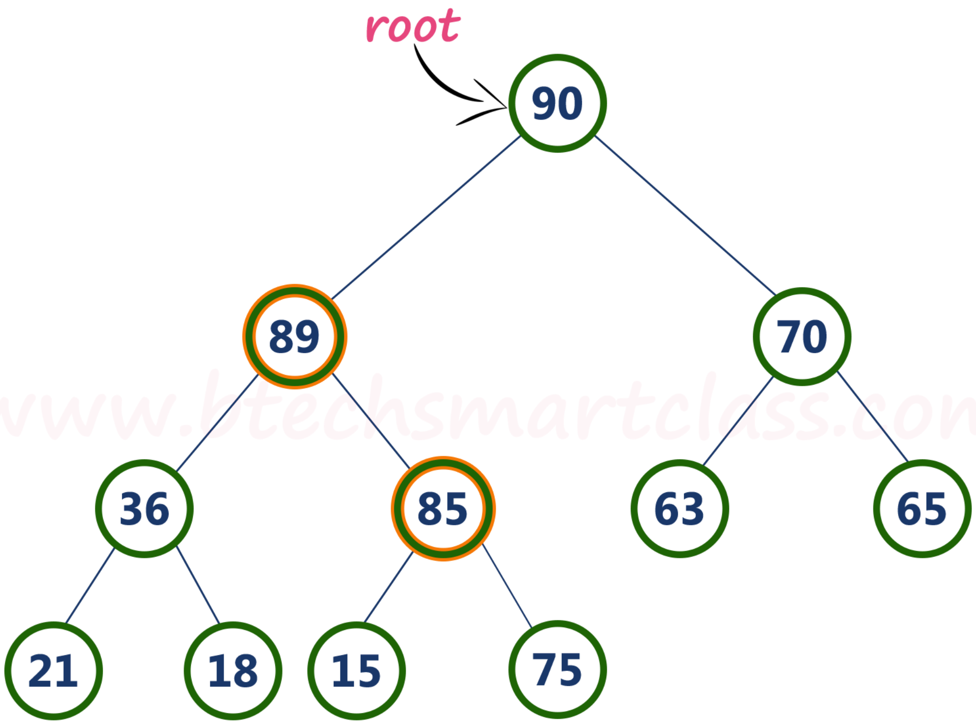


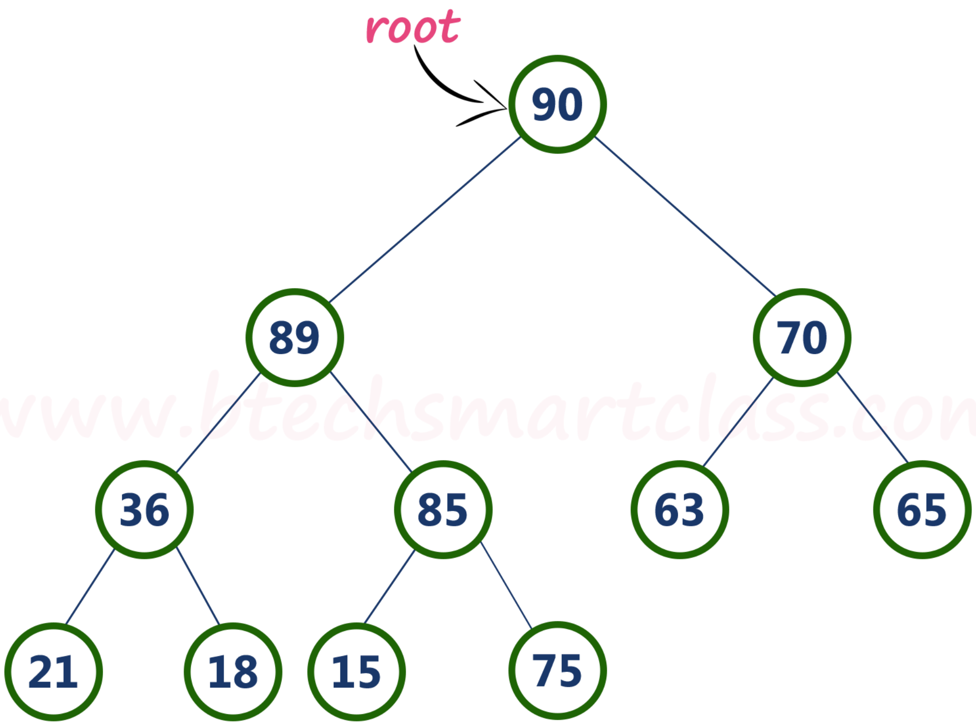
We want to insert 85.




root
90
89
70
36
75
63
65
21
18
15
85


root
90
89
70
36
85
63
65
21
18
15
75

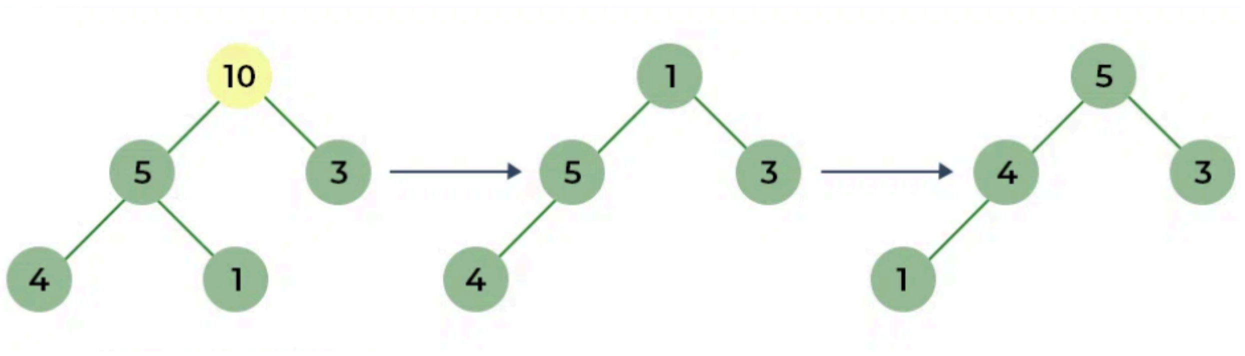

root
90
89
70
36
85
63
65
21
18
15
75


root
90
89
70
36
85
63
65
21
18
15
75

5. Deleting/extracting nodes

To delete/extract a node from the heap we delete the element from the root. The root always gives the maximum element.

Example:



The algorithm for deletion is as follows:

1. Copy the first element into a variable.
2. Copy the last element to the first position (root).
3. call downHeapify().

The code for deletion is :

```
public int extractMax() {
        int max = Heap[1];
        Heap[1] = Heap[size--];  //size 7 size 6
        downHeapify(1);
        return max;
    }
```

Here we use size-- to reduce the size of the heap.

Complete Implementation of Max Heap in Java

The complete java implementation of Max Heap is as follows.

```
public class Main {
    static class MaxHeap {
        private int[] Heap;
        private int size;
        private int maxsize;

        public MaxHeap(int size) {
            this.maxsize = size;
            this.size = 0;
            Heap = new int[this.maxsize + 1];
            Heap[0] = Integer.MAX_VALUE;
        }

        private int parent(int pos) {
            return pos / 2;
        }

        private int leftChild(int pos) {
            return (2 * pos)  ;
        }

        private int rightChild(int pos) {
            return (2 * pos) + 1;
        }


        private void swap(int fpos, int spos) {
            int tmp;
            tmp = Heap[fpos];
            Heap[fpos] = Heap[spos];
            Heap[spos] = tmp;
        }
```

```
      private void downHeapify(int pos) {

  if((leftChild(pos)<=size || rightChild(pos)<=size) && size>0) {
          if (Heap[pos] < Heap[leftChild(pos)] ||
                Heap[pos] < Heap[rightChild(pos)]) {
            if (Heap[leftChild(pos)] > Heap[rightChild(pos)]) {
                swap(pos, leftChild(pos));
                downHeapify(leftChild(pos));
            } else {
                swap(pos, rightChild(pos)); //1,3
                downHeapify(rightChild(pos)); //dhf=3
            }
          }
          }
          else {
                  return;
          }
      }

      private void heapifyUp(int pos) {
          int temp = Heap[pos];
          while(pos>0 && temp > Heap[parent(pos)]){
              Heap[pos] = Heap[parent(pos)];
              pos = parent(pos);
          }
          Heap[pos] = temp;
      }


      public void insert(int element) {
          Heap[++size] = element;


          int current = size;
          heapifyUp(current);

      }

      public void print() {
          for (int i = 1; i <= size / 2; i++) {
              System.out.print(+ Heap[i] + ": L- " +
                      Heap[2 * i] + " R- " + Heap[2 * i + 1]);
              System.out.println();
          }
      }

      public int extractMax() {
          int max = Heap[1];
          Heap[1] = Heap[size--];
          downHeapify(1);
          return max;
      }
  }
  public static void main(String[] arg)
  {

      MaxHeap maxHeap = new MaxHeap(15);
      maxHeap.insert(1);
      maxHeap.insert(4);
      maxHeap.insert(2);
      maxHeap.insert(5);
      maxHeap.insert(13);
      maxHeap.insert(6);
      maxHeap.insert(17);

      maxHeap.print();
      System.out.println("The max is " + maxHeap.extractMax());
  }

  }
```

Output :

```
17: L- 5 R- 13
5: L- 1 R- 4
13: L- 2 R- 6
The max is 17
```

Hometask:

Min Heap is similar to maxheap except that the root is the minimum among all nodes.
Implement min heap in java by yourself.

2. Heap sort

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

If we just add the following few lines of code to the above implementation of Max heap in Java, we can perform heap sort on the array.

You can add the following lines of code to the main method in the previous program.

```
int c=maxHeap.size;
     for(int i=1;i<=c;i++) {
        System.out.print(maxHeap.extractMax()+" ");
     }
```

The above loop simply goes through all the elements of the Max heap and repeatedly delete the root element of the heap. Since this is Max heap, it will sort the elements in descending order. We need to form a Min heap if we want to sort the array in ascending order.