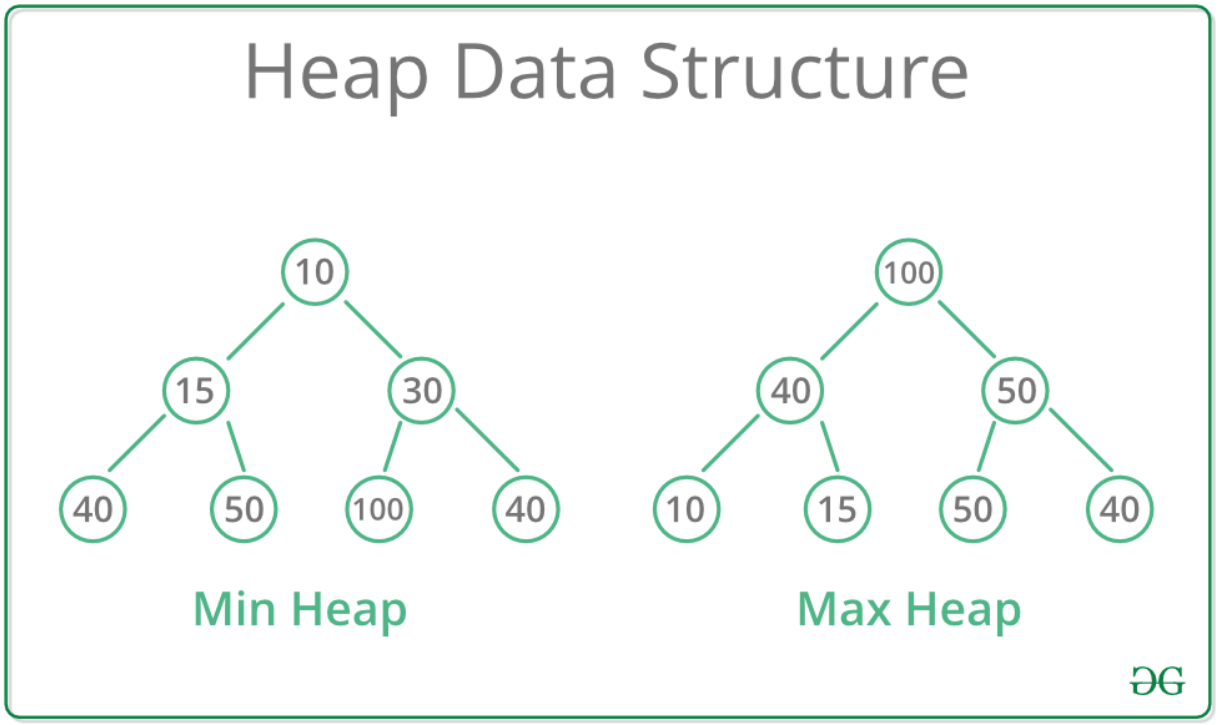
# Heap Data Structure




## Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the first (or can say the top) element of the heap.

## Types of Heap Data Structure

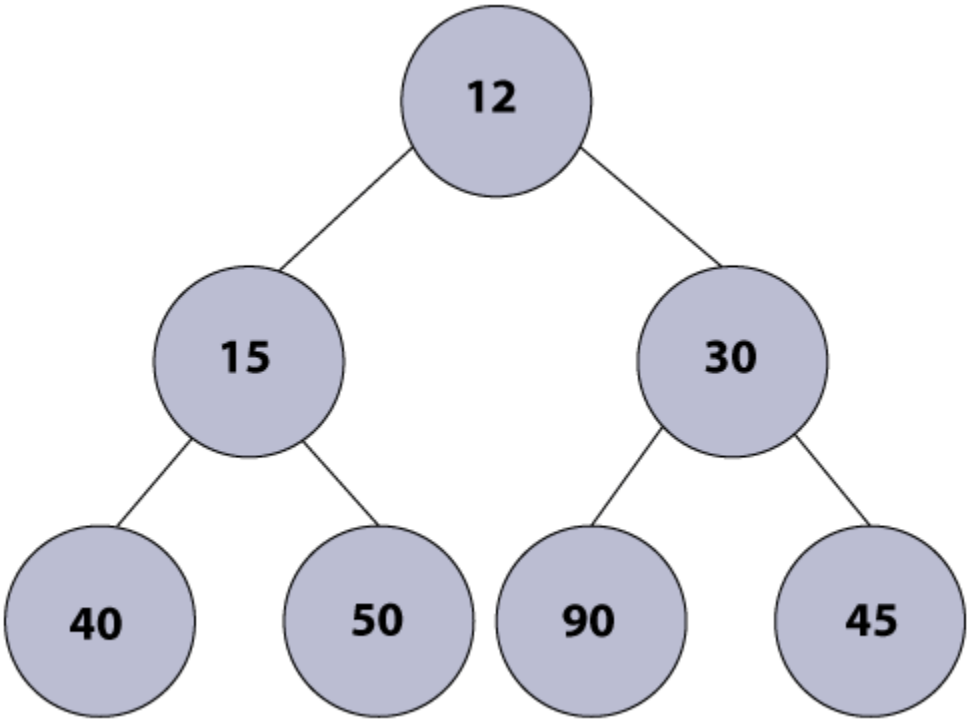
Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

## getMin()

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

**Example:**




**Min heap Algorithm**

```
1. proceduredesign_min_heap
2. Array arr: of size n => array of elements
3. // call min_heapify procedure for each element of the array to form min heap
4. repeat for (k = n/2 ; k >= 1 ; k--)
5.     call procedure min_heapify (arr, k);
6. proceduremin_heapify (vararr[ ] , var k, varn)
7. {
8. varleft_child  = 2*k;
9. varright_child = 2*k+1;
10. var smallest;
11. if(left_child<= n and arr[left_child ] <arr[ k ] )
12. smallest = left_child;
13. else
14. smallest = k;
15. if(right_child<= n and arr[right_child ] <arr[smallest] )
16. smallest = right_child;
17. if(smallest != k)
18.    {
19. swaparr[ k ] and  arr[ smallest ]);
20. callmin_heapify (arr, smallest, n);
21.    }
22. }
```

**MinHeapJavaImplementation.java**

```
1. // import required classes and packages
2. packagejavaTpoint.javacodes;
3.
4. importjava.util.Scanner;
5.
```

```
6.  // create class MinHeap to construct Min heap in Java
7.  classMinHeap {
8.      // declare array and variables
9.  privateint[] heapData;
10. privateintsizeOfHeap;
11. privateintheapMaxSize;
12.
13. private static final int FRONT = 1;
14.     //use constructor to initialize heapData array
15. publicMinHeap(intheapMaxSize)  {
16. this.heapMaxSize = heapMaxSize;
17. this.sizeOfHeap = 0;
18. heapData = new int[this.heapMaxSize + 1];
19. heapData[0] = Integer.MIN_VALUE;
20.     }
21.
22.     // create getParentPos() method that returns parent position for the node
23. privateintgetParentPosition(int position)  {
24. return position / 2;
25.     }
26.
27.     // create getLeftChildPosition() method that returns the position of left child
28. privateintgetLeftChildPosition(int position)  {
29. return (2 * position);
30.     }
31.
32.     // create getRightChildPosition() method that returns the position of right child
33. privateintgetRightChildPosition(int position)  {
34. return (2 * position) + 1;
```

```
35.    }
36.
37.    // checks whether the given node is leaf or not
38. privatebooleancheckLeaf(int position)  {
39. if (position >= (sizeOfHeap / 2) && position <= sizeOfHeap) {
40. return true;
41.        }
42. return false;
43.    }
44.
45.    // create swapNodes() method that perform swapping of the given nodes of
    the heap
46.    // firstNode and secondNode are the positions of the nodes
47. private void swap(intfirstNode, intsecondNode)  {
48. int temp;
49. temp = heapData[firstNode];
50. heapData[firstNode] = heapData[secondNode];
51. heapData[secondNode] = temp;
52.    }
53.
54.    // create minHeapify() method to heapify the node for maintaining the heap
    property
55. private void minHeapify(int position)  {
56.
57.        //check whether the given node is non-leaf and greater than its right and left
    child
58. if (!checkLeaf(position)) {
59. if (heapData[position] >heapData[getLeftChildPosition(position)] ||
    heapData[position] >heapData[getRightChildPosition(position)]) {
```

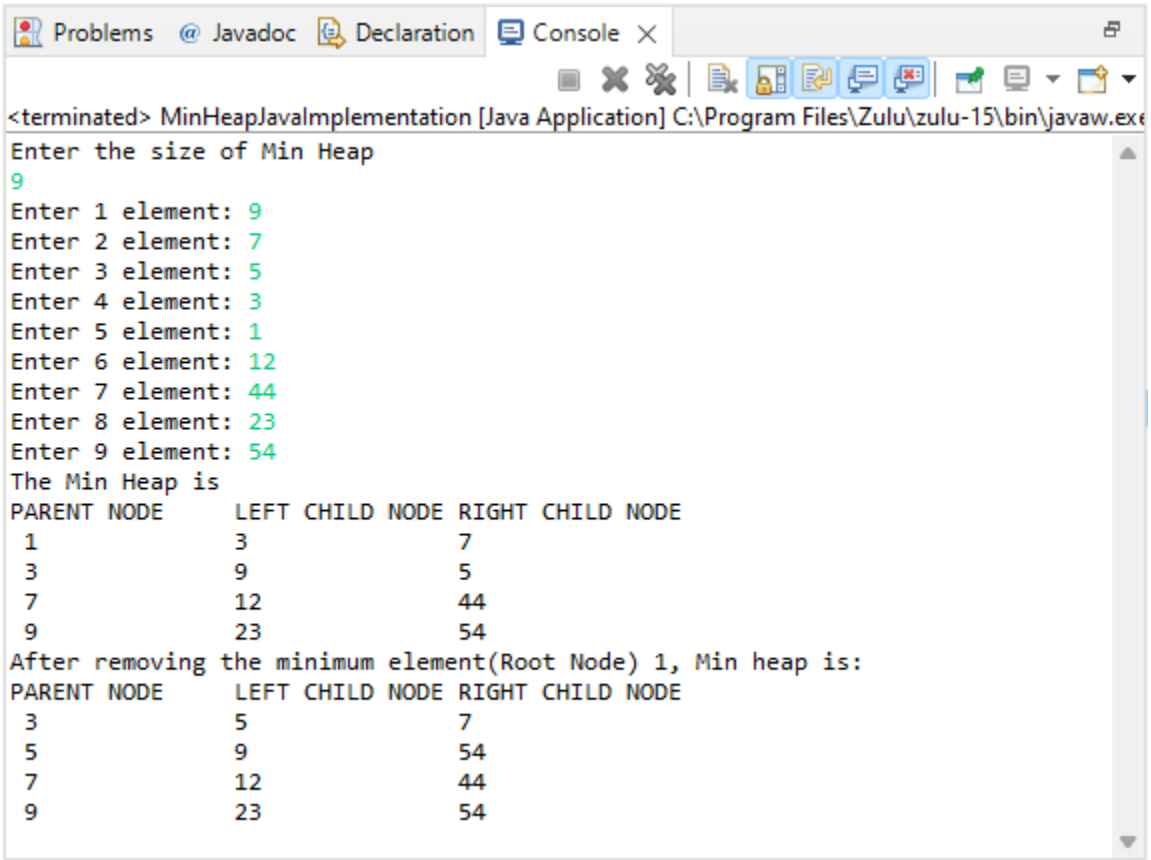
```
60.
61.             // swap with left child and then heapify the left child
62. if (heapData[getLeftChildPosition(position)]
    <heapData[getRightChildPosition(position)]) {
63. swap(position, getLeftChildPosition(position));
64. minHeapify(getLeftChildPosition(position));
65.             }
66.
67.             // Swap with the right child and heapify the right child
68. else {
69. swap(position, getRightChildPosition(position));
70. minHeapify(getRightChildPosition(position));
71.             }
72.         }
73.     }
74.   }
75.
76.   // create insertNode() method to insert element in the heap
77. public void insertNode(int data)  {
78. if (sizeOfHeap>= heapMaxSize) {
79. return;
80.     }
81. heapData[++sizeOfHeap] = data;
82. int current = sizeOfHeap;
83.
84. while (heapData[current] <heapData[getParentPosition(current)]) {
85. swap(current, getParentPosition(current));
86. current = getParentPosition(current);
87.     }
```

```
88.    }
89.
90.     // crreatedisplayHeap() method to print the data of the heap
91. public void displayHeap()  {
92. System.out.println("PARENT NODE" + "\t" + "LEFT CHILD NODE" + "\t" + "RIGHT
    CHILD NODE");
93. for (int k = 1; k <= sizeOfHeap / 2; k++) {
94. System.out.print(" " + heapData[k] + "\t\t" + heapData[2 * k] + "\t\t" + heapData[2 *
    k + 1]);
95. System.out.println();
96.        }
97.    }
98.
99.    // create designMinHeap() method to construct min heap
100.    public void designMinHeap()  {
101.    for (int position = (sizeOfHeap / 2); position >= 1; position--) {
102.    minHeapify(position);
103.          }
104.       }
105.
106.       // create removeRoot() method for removing minimum element from the
    heap
107.    publicintremoveRoot()  {
108.    intpopElement = heapData[FRONT];
109.    heapData[FRONT] = heapData[sizeOfHeap--];
110.    minHeapify(FRONT);
111.    returnpopElement;
112.       }
113.    }
```

```
114.
115.  // create MinHeapJavaImplementation class to create heap in Java
116.  classMinHeapJavaImplementation{
117.
118.     // main() method start
119.  public static void main(String[] arg)  {
120.     // declare variable
121.     intheapSize;
122.
123.     // create scanner class object
124.     Scanner sc = new Scanner(System.in);
125.
126.     System.out.println("Enter the size of Min Heap");
127.     heapSize = sc.nextInt();
128.
129.     MinHeapheapObj = new MinHeap(heapSize);
130.
131.     for(inti = 1; i<= heapSize; i++) {
132.        System.out.print("Enter "+i+" element: ");
133.        int data = sc.nextInt();
134.        heapObj.insertNode(data);
135.     }
136.
137.        // close scanner class obj
138.  sc.close();
139.
140.        //construct a min heap from given data
141.  heapObj.designMinHeap();
142.
```

```
143.          //display the min heap data
144.    System.out.println("The Min Heap is ");
145.    heapObj.displayHeap();
146.
147.          //removing the root node from the heap
148.    System.out.println("After removing the minimum element(Root Node)
        "+heapObj.removeRoot()+", Min heap is:");
149.    heapObj.displayHeap();
150.
151.       }
152.    }
```

**Output:**



# 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.

We can perform the following three operations in Max heap:

## insertNode()

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.

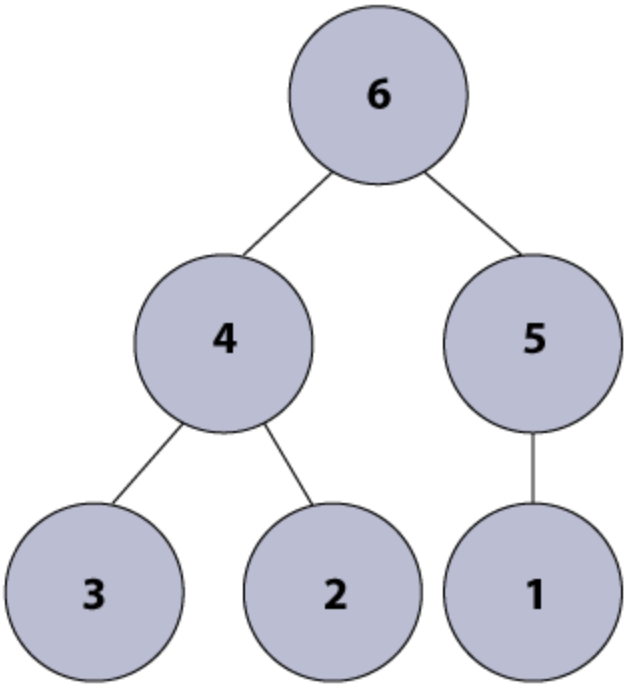
## extractMax()

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. The extractMax() operation takes O(Log n) time to remove the maximum element from the heap.

## getMax()

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

**Example:**




**Min heap Algorithm**

```
1.  proceduredesign_max_heap
2.  Array arr: of size n => array of elements
3.  // call min_heapify procedure for each element of the array to form max heap
4.  repeat for (k = n/2 ; k >= 1 ; k--)
5.      call procedure max_heapify (arr, k);
6.  proceduremin_heapify (vararr[ ] , var k, varn)
7.  {
8.  varleft_child  = 2*k + 1;
9.  varright_child = 2*k+ 2;
10. if(left_child<= n and arr[left_child] >arr[ largest ] )
11. largest = left_child;
12. else
13. largest = k;
14. if(right_child<= n and arr[right_child] >arr[largest] )
```

```
15. largest = right_child;
16. if(largest != k)
17.    {
18. swaparr[ k ] and  arr[ largest ]);
19. callmax_heapify (arr, largest, n);
20.    }
21. }
```

**MaxHeapJavaImplementation.java**

```
1.  //import required classes and packages
2.  packagejavaTpoint.javacodes;
3.
4.  importjava.util.Scanner;
5.
6.  //create class MinHeap to construct Min heap in Java
7.  classMaxHeap {
8.      // declare array and variables
9.      privateint[] heapData;
10.     privateintsizeOfHeap;
11.     privateintheapMaxSize;
12.
13.     private static final int FRONT = 1;
14.
15.     //use constructor to initialize heapData array
16.     publicMaxHeap(intheapMaxSize)  {
17.         this.heapMaxSize = heapMaxSize;
18.         this.sizeOfHeap = 0;
19.         heapData = new int[this.heapMaxSize];
20.     }
```

```
21.
22.    // create getParentPos() method that returns parent position for the node
23.    privateintgetParentPosition(int position)  {
24.        return (position - 1) / 2;
25.    }
26.
27.    // create getLeftChildPosition() method that returns the position of left child
28.    privateintgetLeftChildPosition(int position)  {
29.        return (2 * position);
30.    }
31.
32.    // create getRightChildPosition() method that returns the position of right child
33.    privateintgetRightChildPosition(int position)  {
34.        return (2 * position) + 1;
35.    }
36.
37.    // checks whether the given node is leaf or not
38.    privatebooleancheckLeaf(int position)  {
39.        if (position > (sizeOfHeap / 2) && position <= sizeOfHeap) {
40.            return true;
41.        }
42.        return false;
43.    }
44.
45.    // create swapNodes() method that perform swapping of the given nodes of
     the heap
46.    // firstNode and secondNode are the positions of the nodes
47.    private void swap(intfirstNode, intsecondNode)  {
48.        int temp;
```

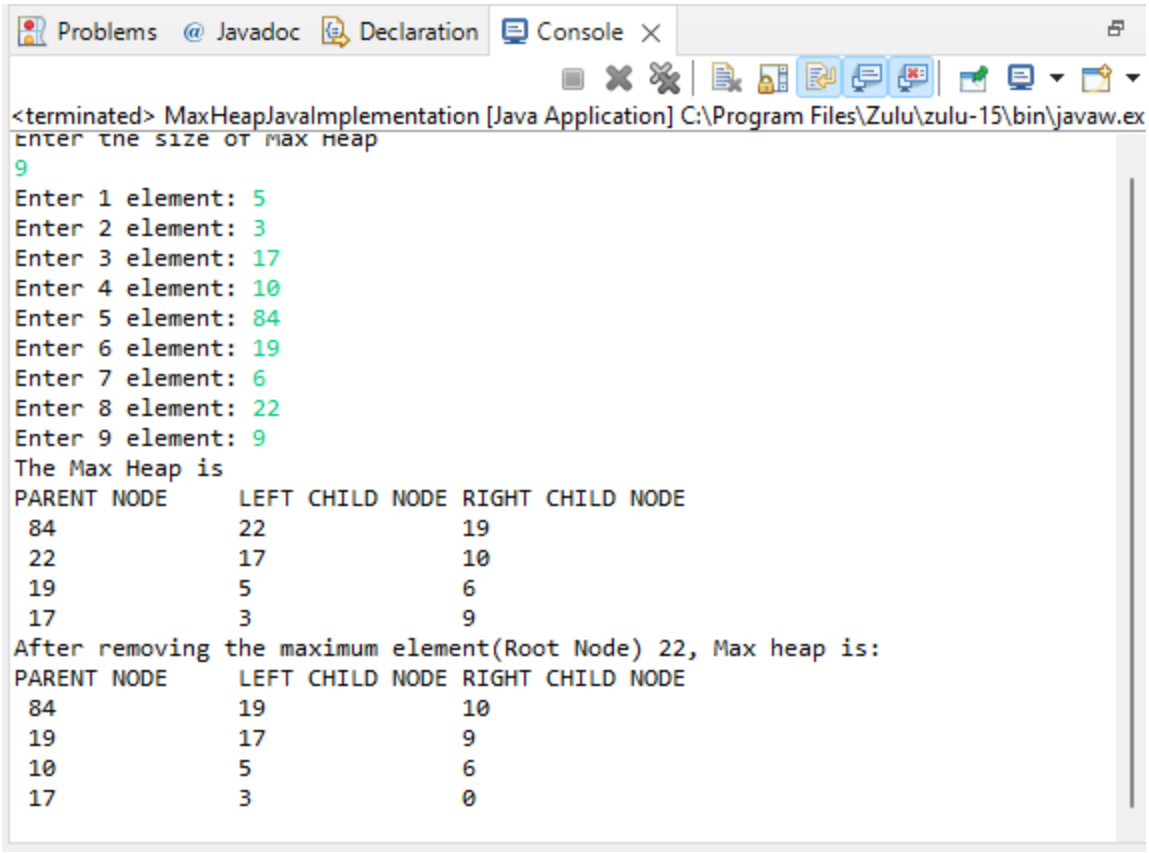
```
49.         temp = heapData[firstNode];
50.         heapData[firstNode] = heapData[secondNode];
51.         heapData[secondNode] = temp;
52.     }
53.
54.     // create maxHeapify() method to heapify the node for maintaining the heap
    property
55.     private void maxHeapify(int position)  {
56.
57.         //check whether the given node is non-leaf and greater than its right and left
    child
58.     if (!checkLeaf(position)) {
59.     if (heapData[position] <heapData[getLeftChildPosition(position)] ||
    heapData[position] <heapData[getRightChildPosition(position)]) {
60.
61.                 // swap with left child and then heapify the left child
62.     if (heapData[getLeftChildPosition(position)]
    >heapData[getRightChildPosition(position)]) {
63.     swap(position, getLeftChildPosition(position));
64.     maxHeapify(getLeftChildPosition(position));
65.                 }
66.
67.                 // Swap with the right child and heapify the right child
68.     else {
69.     swap(position, getRightChildPosition(position));
70.     maxHeapify(getRightChildPosition(position));
71.                 }
72.             }
73.         }
```

```
74.    }
75.
76.    // create insertNode() method to insert element in the heap
77.    public void insertNode(int data)  {
78.        heapData[sizeOfHeap] = data;
79.        int current = sizeOfHeap;
80.
81.        while (heapData[current] >heapData[getParentPosition(current)]) {
82.            swap(current, getParentPosition(current));
83.            current = getParentPosition(current);
84.        }
85.        sizeOfHeap++;
86.    }
87.
88.    // create displayHeap() method to print the data of the heap
89.    public void displayHeap()  {
90.        System.out.println("PARENT NODE" + "\t" + "LEFT CHILD NODE" + "\t" +
    "RIGHT CHILD NODE");
91.        for (int k = 0; k <sizeOfHeap / 2; k++) {
92.            System.out.print(" " + heapData[k] + "\t\t" + heapData[2 * k + 1] + "\t\t" +
    heapData[2 * k + 2]);
93.            System.out.println();
94.        }
95.    }
96.
97.    // create designMaxHeap() method to construct min heap
98.    public void designMaxHeap()  {
99.        for (int position = 0;  position < (sizeOfHeap / 2); position++) {
100.            maxHeapify(position);
```

```
101.          }
102.      }
103.
104.      // create removeRoot() method for removing maximum element from the
    heap
105.      publicintremoveRoot()  {
106.          intpopElement = heapData[FRONT];
107.          heapData[FRONT] = heapData[sizeOfHeap--];
108.          maxHeapify(FRONT);
109.          returnpopElement;
110.      }
111.  }
112.
113.  //create MinHeapJavaImplementation class to create heap in Java
114.  classMaxHeapJavaImplementation{
115.
116.      // main() method start
117.  public static void main(String[] arg)  {
118.      // declare variable
119.      intheapSize;
120.
121.      // create scanner class object
122.      Scanner sc = new Scanner(System.in);
123.
124.      System.out.println("Enter the size of Max Heap");
125.      heapSize = sc.nextInt();
126.
127.      MaxHeapheapObj = new MaxHeap(50);
128.
```

```
129.        for(inti = 1; i<= heapSize; i++) {
130.            System.out.print("Enter "+i+" element: ");
131.            int data = sc.nextInt();
132.            heapObj.insertNode(data);
133.        }
134.
135.        // close scanner class obj
136.    sc.close();
137.
138.        //construct a max heap from given data
139.    heapObj.designMaxHeap();
140.
141.        //display the max heap data
142.    System.out.println("The Max Heap is ");
143.    heapObj.displayHeap();
144.
145.        //removing the root node from the heap
146.    System.out.println("After removing the maximum element(Root Node)
      "+heapObj.removeRoot()+", Max heap is:");
147.    heapObj.displayHeap();
148.
149.     }
150.    }
```

**Output:**


Problems
Javadoc
Declaration
Console
<terminated> MaxHeapJavaImplementation [Java Application] C:\Program Files\Zulu\zulu-15\bin\javaw.ex
Enter the size of Max Heap
9
Enter 1 element: 5
Enter 2 element: 3
Enter 3 element: 17
Enter 4 element: 10
Enter 5 element: 84
Enter 6 element: 19
Enter 7 element: 6
Enter 8 element: 22
Enter 9 element: 9
The Max Heap is
PARENT NODE LEFT CHILD NODE RIGHT CHILD NODE
84 22 19
22 17 10
19 5 6
17 3 9
After removing the maximum element(Root Node) 22, Max heap is:
PARENT NODE LEFT CHILD NODE RIGHT CHILD NODE
84 19 10
19 17 9
10 5 6
17 3 0