GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 5 Report

Muhammed Sinan Pehlivanoğlu
1901042664

1. SYSTEM REQUIREMENTS

Operating System need to jdk and jre for start the program
Stack capacity should not be less.
You need to enough space to store data's of program according to the how many you have objects.

2.PROBLEM SOLUTION APPROACH

- *For 1.Problem*

  Part A :The depth of the any level is equal to height of the that level plus number of nodes at that level. We need to find these variables.

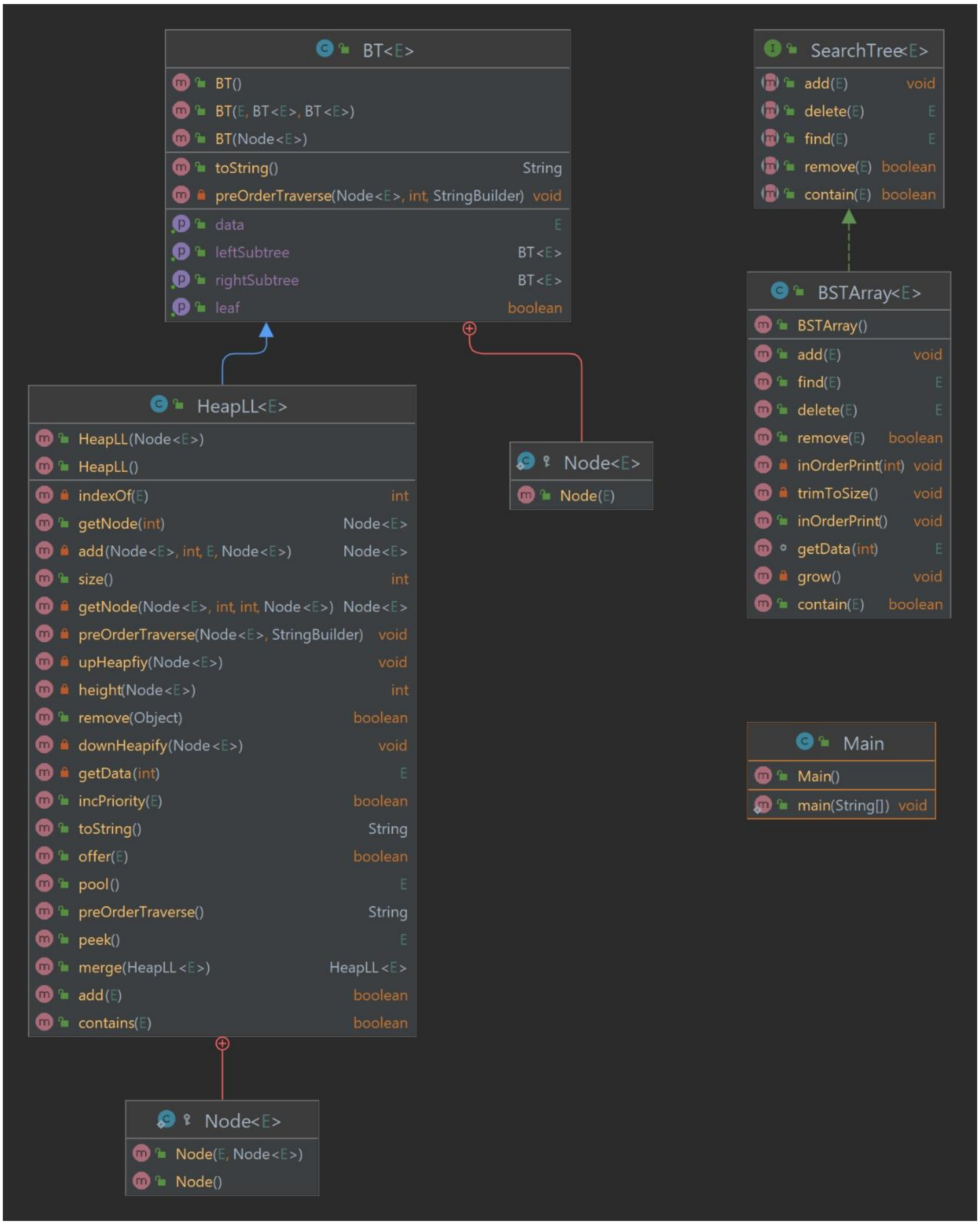
  Part B: Average comparison is total comparison of tree for each node divided by total node number of tree. We need to find these variables.

  Part C : The internal nodes is nodes that has child. The leaves are nodes with no child. So we need to find these numbers.

- *For 2.Problem , The structure of quad tree is that the quad tree has for child and each child has region like region of coordinate system and its border.The main purpose is that, if node to be added is located into region that has already node. Then the pre node in there divide into 4 node and these node to dived node according to its region again. İf the there is no pre node in that region, then locate the node.*

- *For 3.Problem The Heap is node structural. İnner Node class have tree subnode. Parent, left , right nodes. The element is accessed by level order like a array. Then the operation is done according to accessed node. For example adding the element to the heap is done by adding the desired data to tail of the node then the up heapifying is done until the added the find the exact location of itself. Another example is pool the data. Pooling is meaning that remove the element with higher priority. To pool the data, data of tail is inserted to head of the heap. Then the down heapify is done until the last element find exact position of itself.*
- *For 4. Problem,*

3.CLASS DIAGRAMS

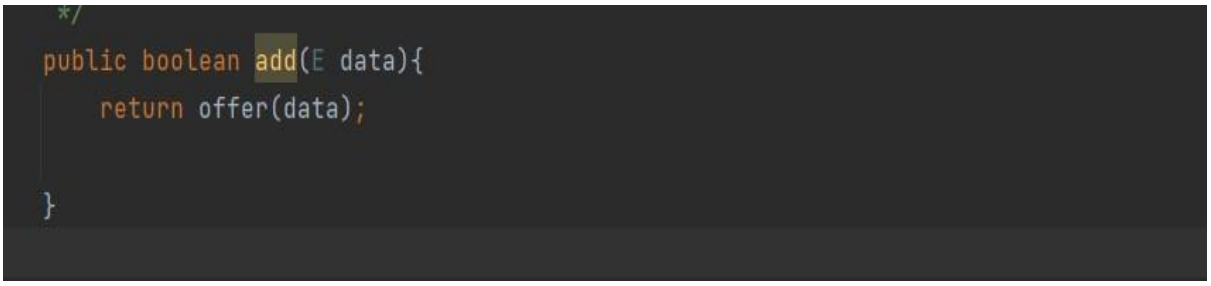


4.TEST CASES

- Add heap to element.
- Remove heap from element.
- Poll element from heap.
- Peek element from heap.
- Find the element is contained in the heap.
- Merge two heap.
- Increase the priority of desired element.
- Show the size of the heap.

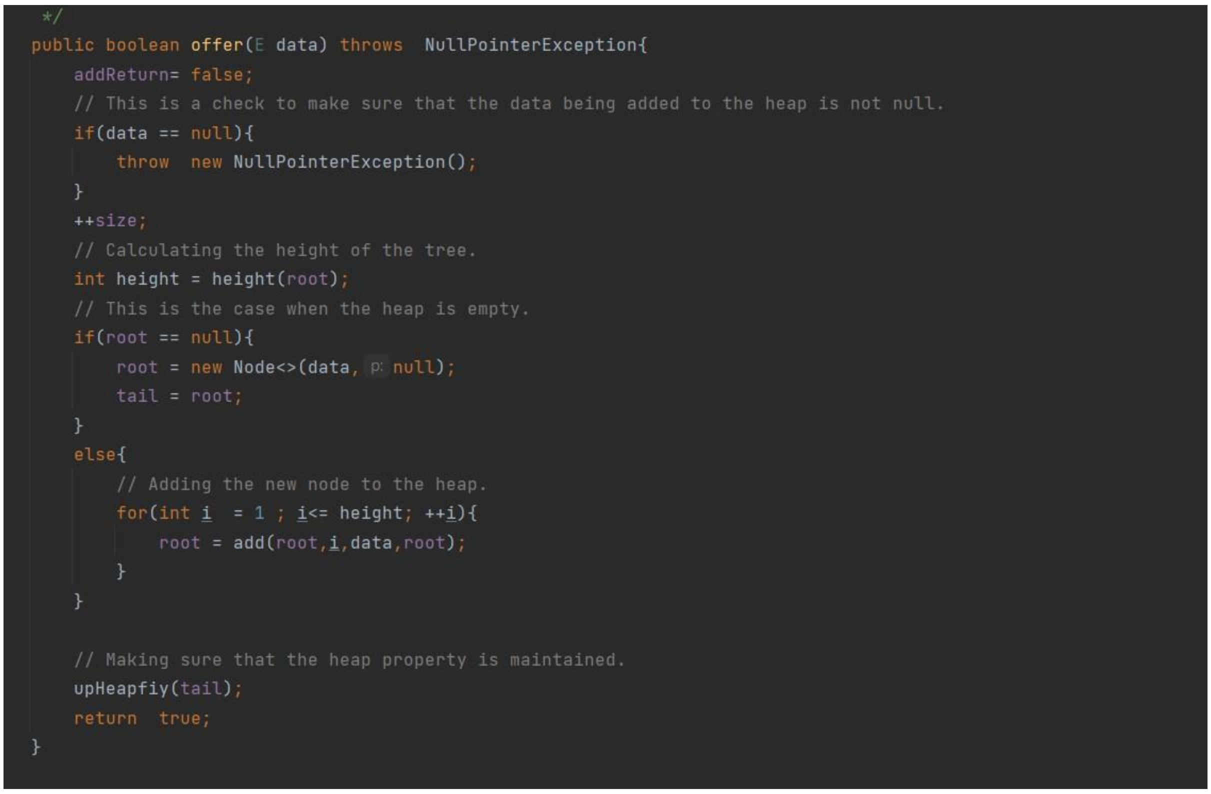
5.RUNNING COMMANDS AND RESULTS

- **Adding the element to the heap.**



```
 */
public boolean add(E data){
    return offer(data);

}
```

- **Offer method**

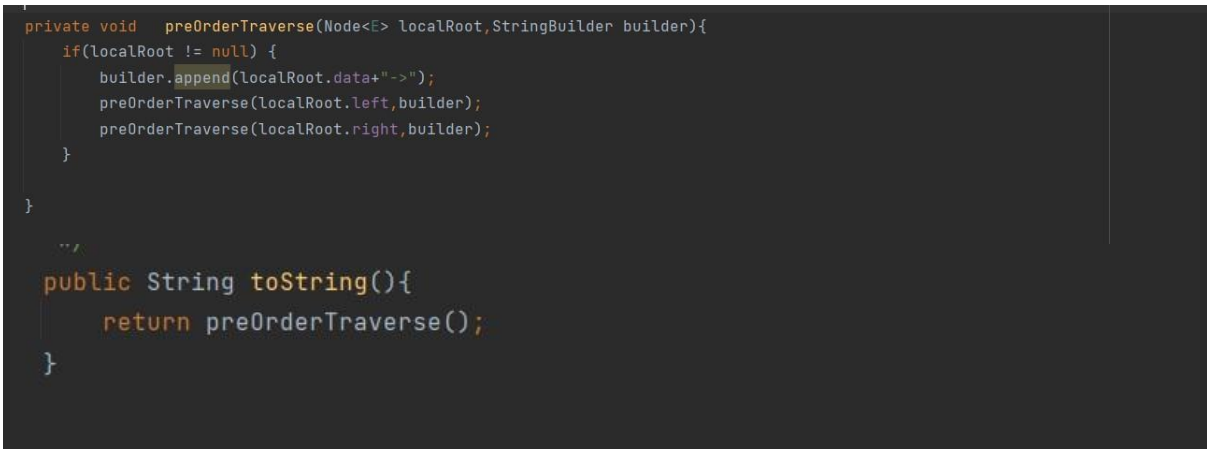


```
 */
public boolean offer(E data) throws  NullPointerException{
    addReturn= false;
    // This is a check to make sure that the data being added to the heap is not null.
    if(data == null){
        throw  new NullPointerException();
    }
    ++size;
    // Calculating the height of the tree.
    int height = height(root);
    // This is the case when the heap is empty.
    if(root == null){
        root = new Node<>(data, p: null);
        tail = root;
    }
    else{
        // Adding the new node to the heap.
        for(int i  = 1 ; i<= height; ++i){
            root = add(root,i,data,root);
        }
    }

    // Making sure that the heap property is maintained.
    upHeapfiy(tail);
    return  true;
}
```

Worst case is O(logn).

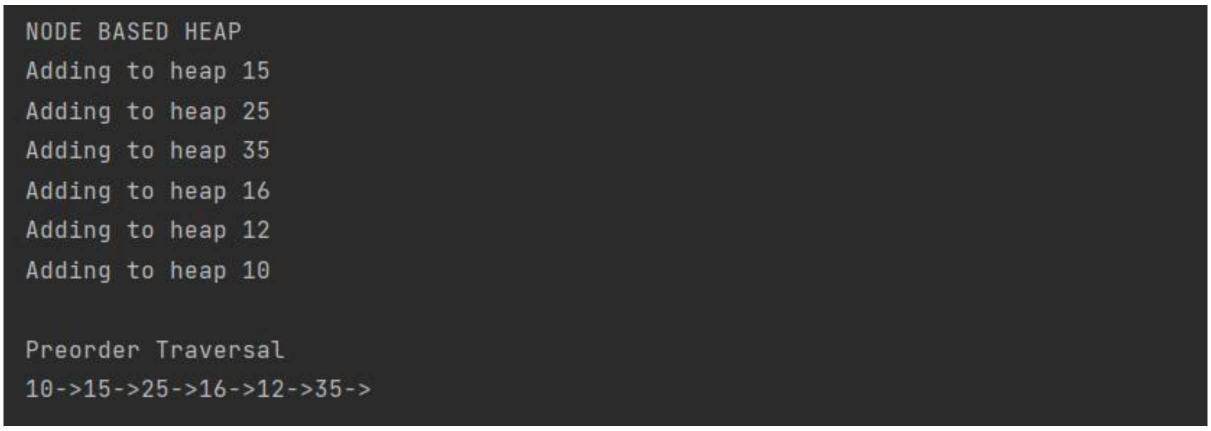
Best case is θ(1).

- preOrderTraversing



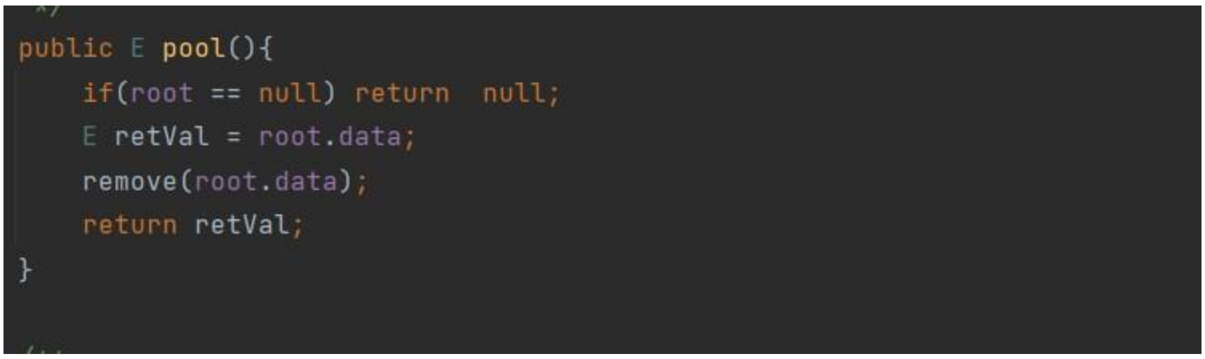
```
private void   preOrderTraverse(Node<E> localRoot,StringBuilder builder){
    if(localRoot != null) {
        builder.append(localRoot.data+"->");
        preOrderTraverse(localRoot.left,builder);
        preOrderTraverse(localRoot.right,builder);
    }

}

  public String toString(){
      return preOrderTraverse();
  }
```

Preorder traversing takes θ(n)



```
NODE BASED HEAP
Adding to heap 15
Adding to heap 25
Adding to heap 35
Adding to heap 16
Adding to heap 12
Adding to heap 10

Preorder Traversal
10->15->25->16->12->35->
```

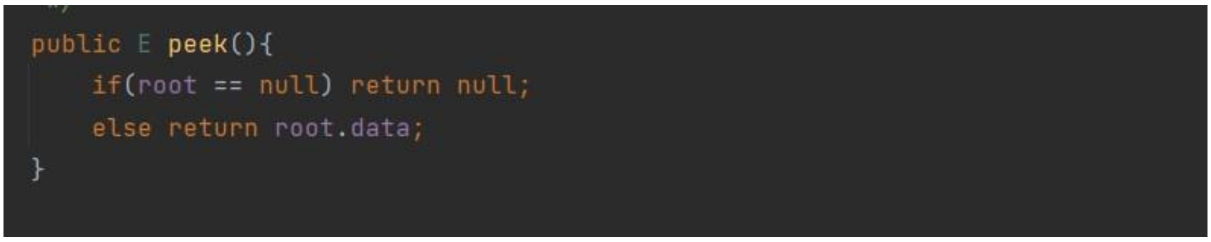
- **Poll the element from heap.**



```
public E pool(){
    if(root == null) return  null;
    E retVal = root.data;
    remove(root.data);
    return retVal;
}
```

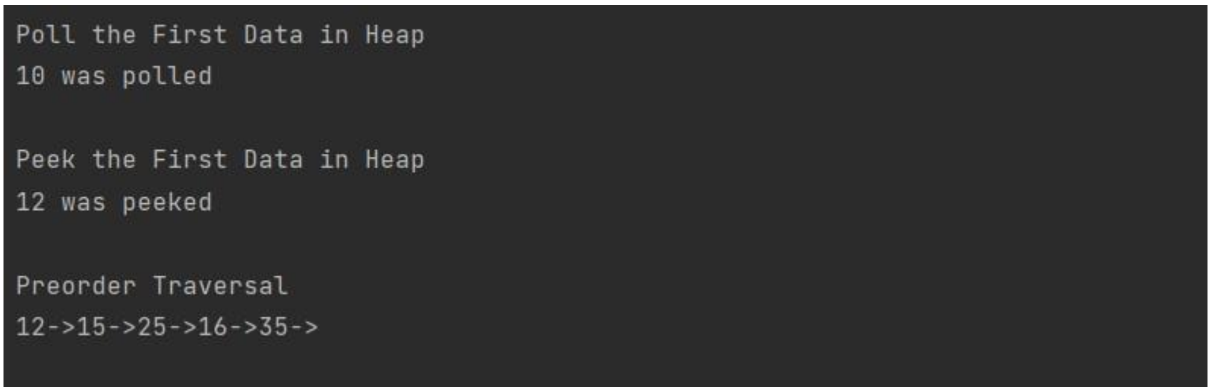
Takes O(logn).

- **Peek the element from heap .**



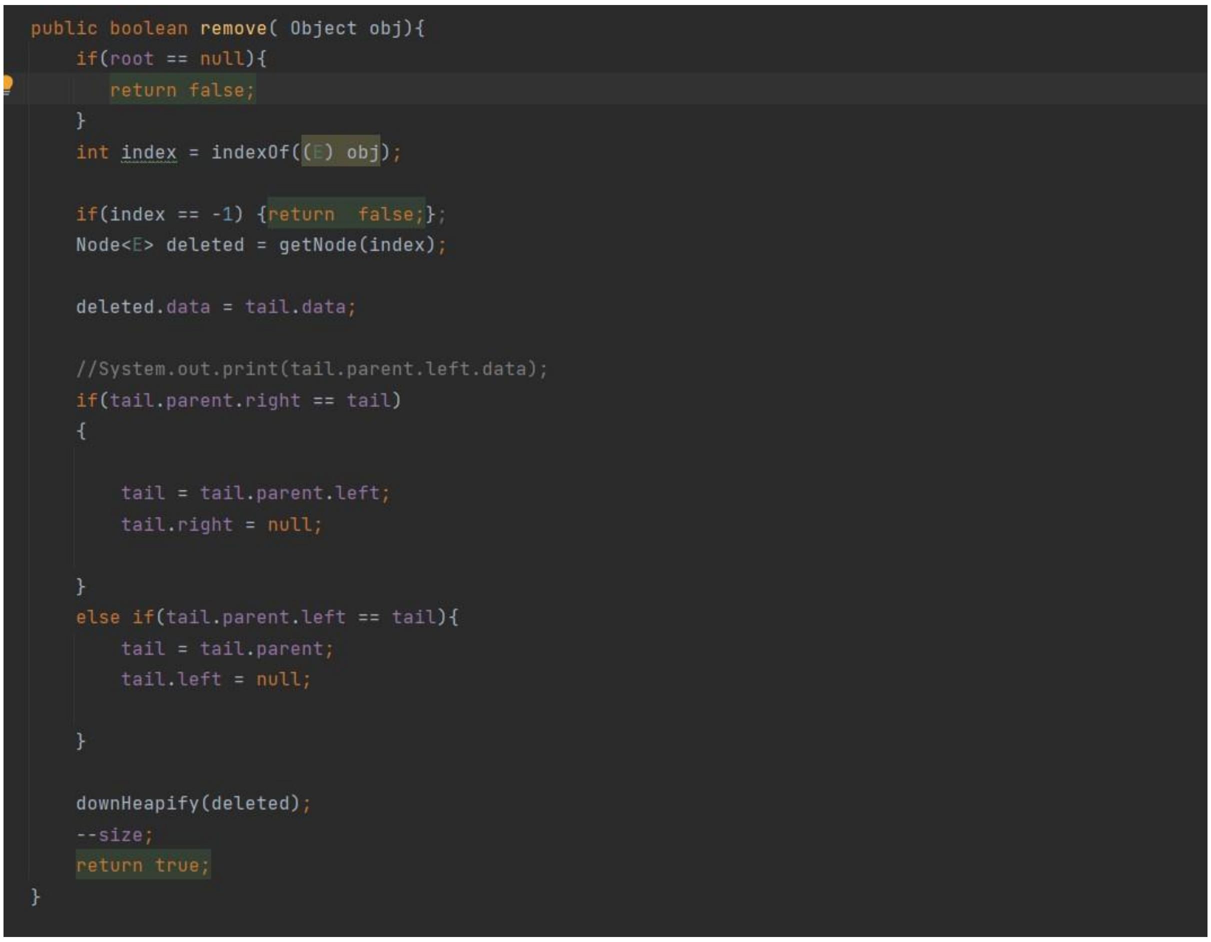
```
public E peek(){
    if(root == null) return null;
    else return root.data;
}
```

Takes θ(1).



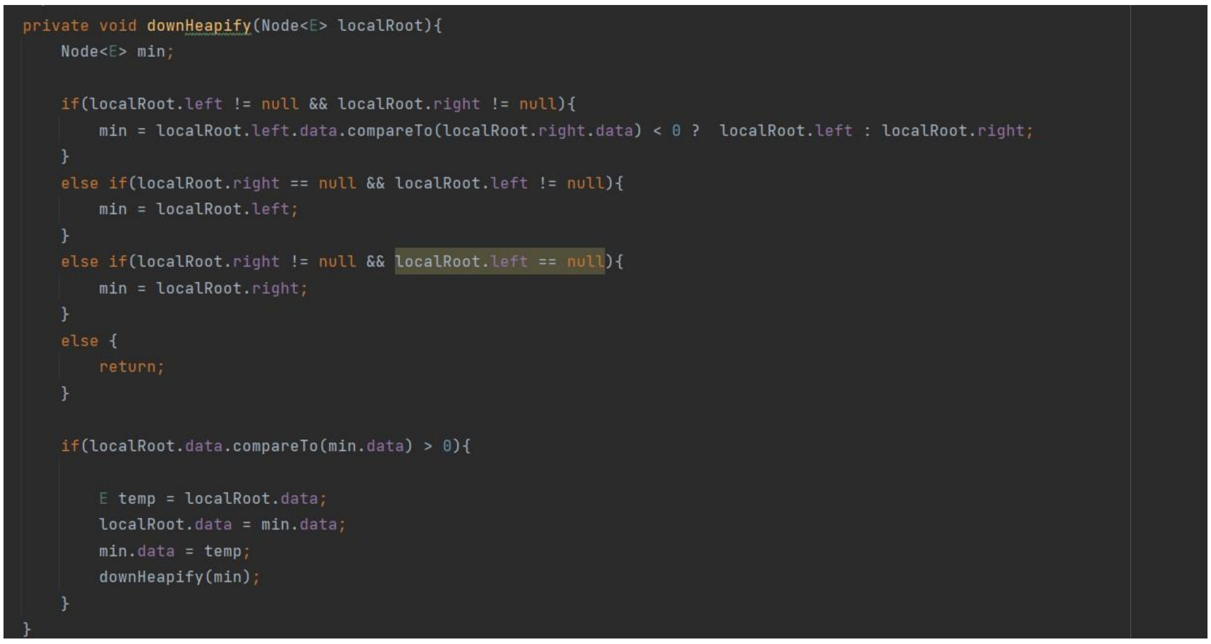
```
Poll the First Data in Heap
10 was polled

Peek the First Data in Heap
12 was peeked

Preorder Traversal
12->15->25->16->35->
```

- **Remove element from heap.**

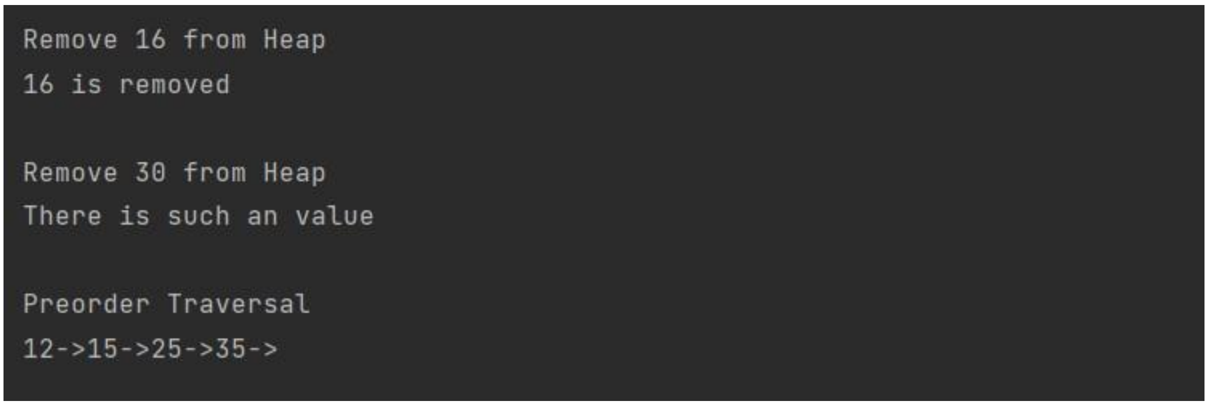


```
public boolean remove( Object obj){
    if(root == null){
        return false;
    }
    int index = indexOf((E) obj);

    if(index == -1) {return  false;};
    Node<E> deleted = getNode(index);

    deleted.data = tail.data;

    //System.out.print(tail.parent.left.data);
    if(tail.parent.right == tail)
    {

        tail = tail.parent.left;
        tail.right = null;

    }
    else if(tail.parent.left == tail){
        tail = tail.parent;
        tail.left = null;

    }

    downHeapify(deleted);
    --size;
    return true;
}
```

DownHeapify takes O(logn). So remove takes O(logn)

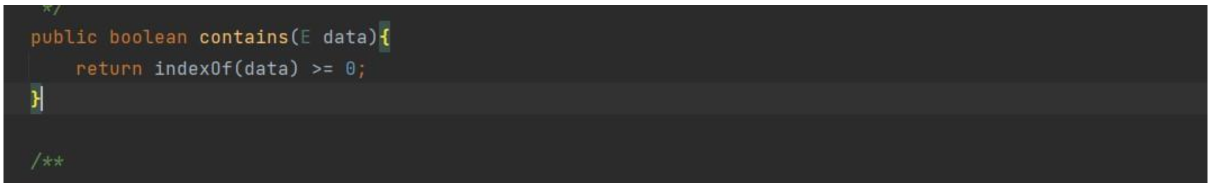


```
private void downHeapify(Node<E> localRoot){
    Node<E> min;

    if(localRoot.left != null && localRoot.right != null){
        min = localRoot.left.data.compareTo(localRoot.right.data) < 0 ?  localRoot.left : localRoot.right;
    }
    else if(localRoot.right == null && localRoot.left != null){
        min = localRoot.left;
    }
    else if(localRoot.right != null && localRoot.left == null){
        min = localRoot.right;
    }
    else {
        return;
    }

    if(localRoot.data.compareTo(min.data) > 0){

        E temp = localRoot.data;
        localRoot.data = min.data;
        min.data = temp;
        downHeapify(min);
    }
}
```

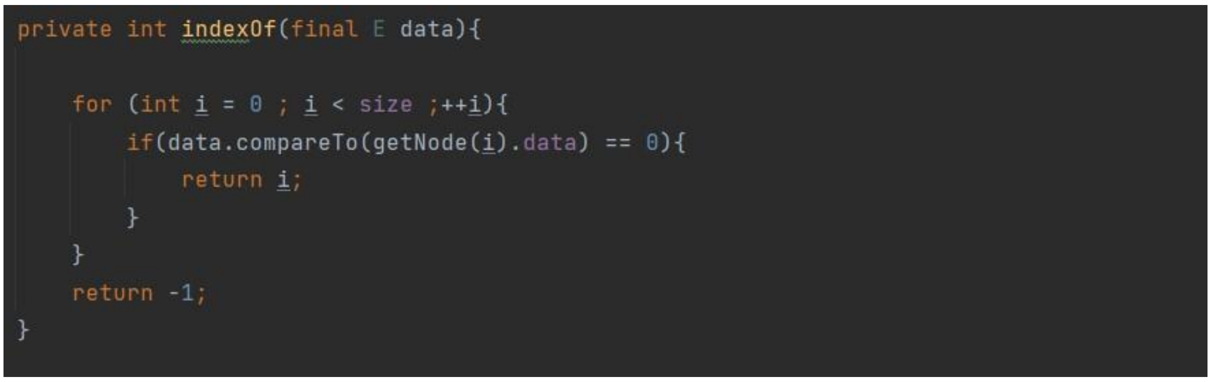


```
Remove 16 from Heap
16 is removed

Remove 30 from Heap
There is such an value

Preorder Traversal
12->15->25->35->
```

- **Check the element is contained in heap.**



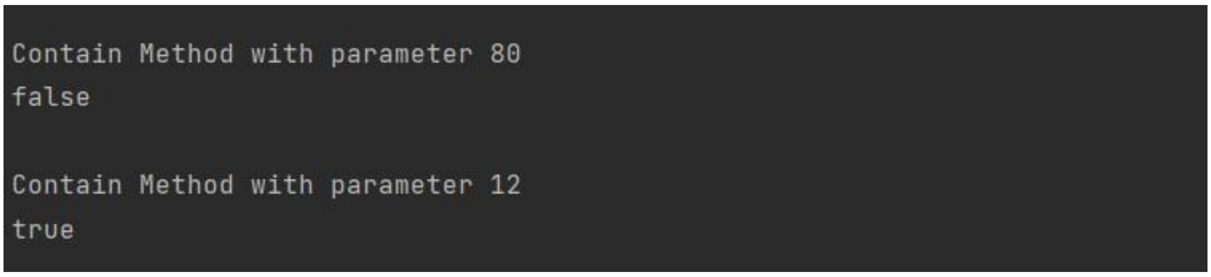
```
*/
public boolean contains(E data){
    return indexOf(data) >= 0;
}

/**
```



```
private int indexOf(final E data){

    for (int i = 0 ; i < size ;++i){
        if(data.compareTo(getNode(i).data) == 0){
            return i;
        }
    }
    return -1;
}
```

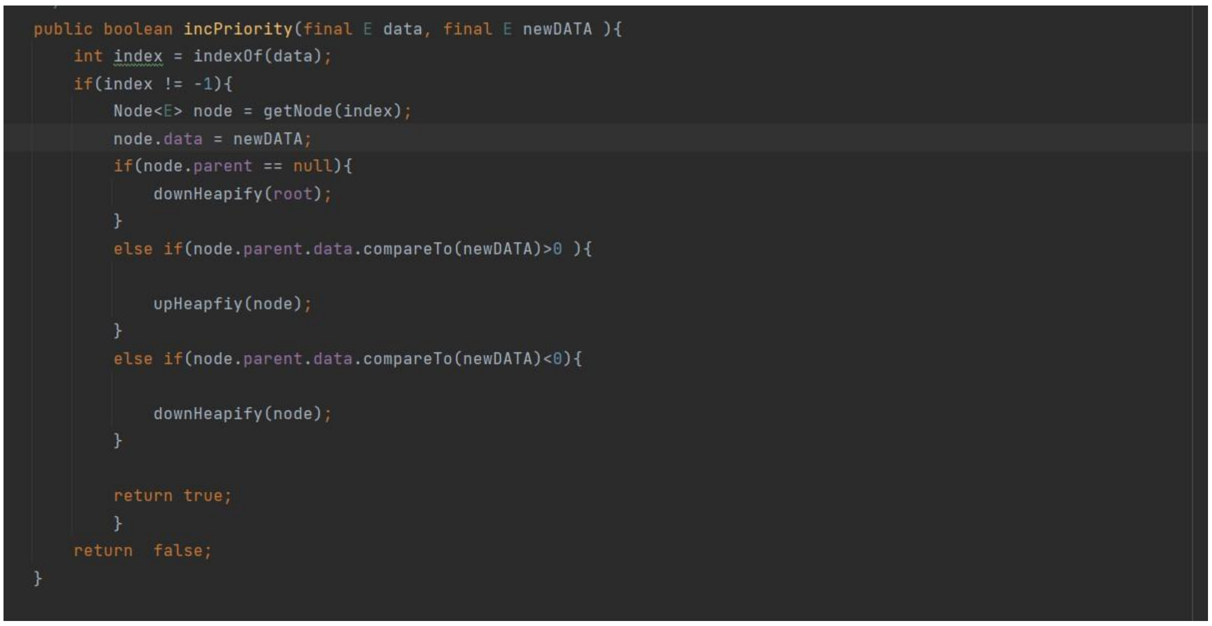
- **IndexOf method takes O(n^2). Due to getNode takes O(N).**

So Contains take O(n^2).



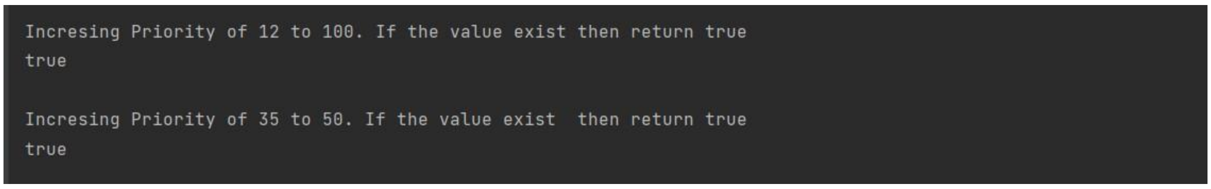
```
Contain Method with parameter 80
false

Contain Method with parameter 12
true
```

- **Increase the priority**



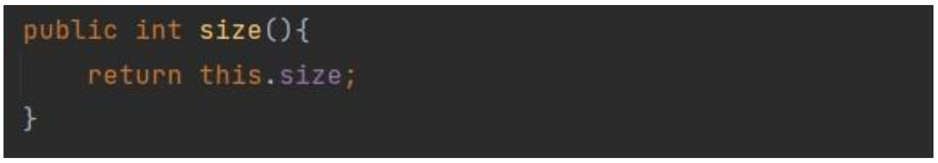
```
public boolean incPriority(final E data, final E newDATA ){
    int index = indexOf(data);
    if(index != -1){
        Node<E> node = getNode(index);
        node.data = newDATA;
        if(node.parent == null){
            downHeapify(root);
        }
        else if(node.parent.data.compareTo(newDATA)>0 ){

            upHeapfiy(node);
        }
        else if(node.parent.data.compareTo(newDATA)<0){

            downHeapify(node);
        }

        return true;
        }
    return  false;
}
```

It takes O (logn) due to downhepify and upheapiy takes O(logn).



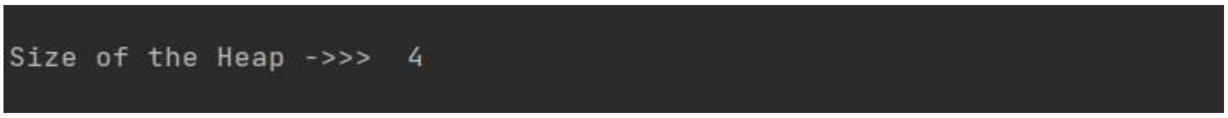
```
Incresing Priority of 12 to 100. If the value exist then return true
true

Incresing Priority of 35 to 50. If the value exist  then return true
true
```

- **Size of heap.**



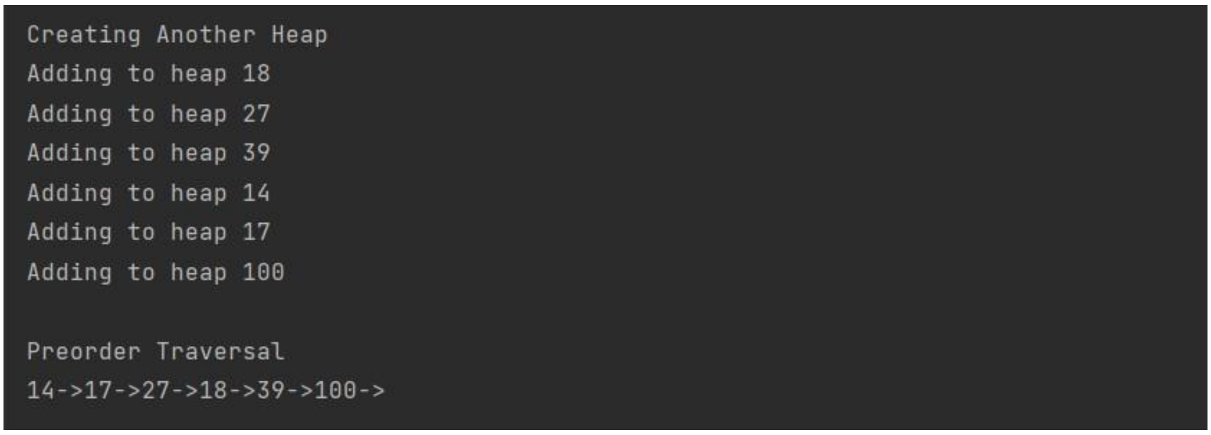
```
public int size(){
    return this.size;
}
```

İt takes Ɵ (1).



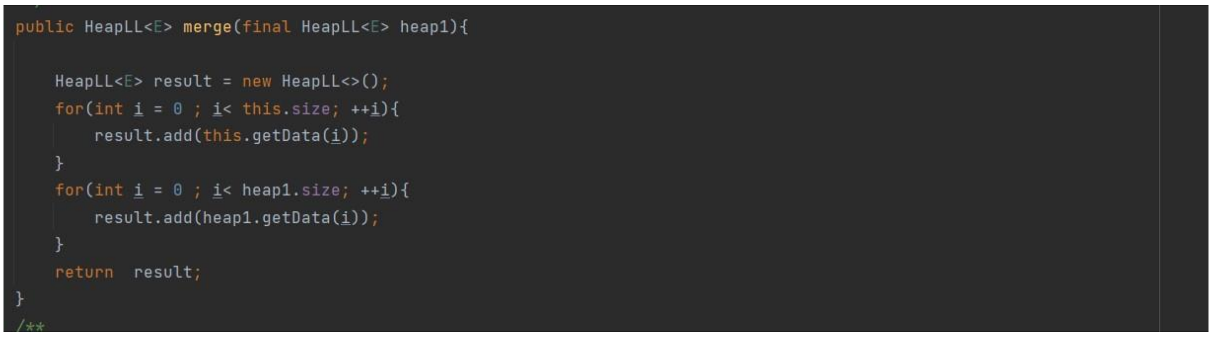
```
Size of the Heap ->>>  4
```

- **Creating another heap**



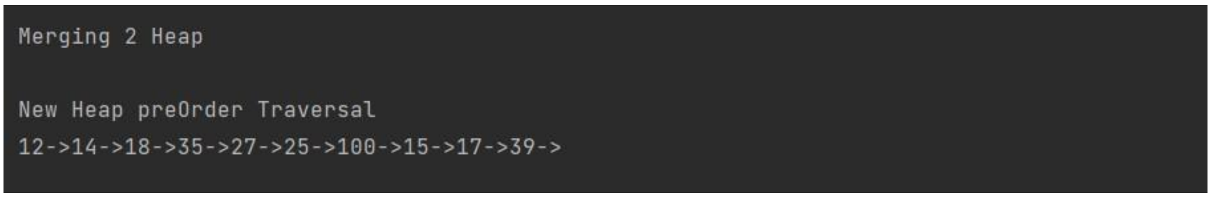
```
Creating Another Heap
Adding to heap 18
Adding to heap 27
Adding to heap 39
Adding to heap 14
Adding to heap 17
Adding to heap 100

Preorder Traversal
14->17->27->18->39->100->
```

- **Merging two Heap**



```
public HeapLL<E> merge(final HeapLL<E> heap1){

    HeapLL<E> result = new HeapLL<>();
    for(int i = 0 ; i< this.size; ++i){
        result.add(this.getData(i));
    }
    for(int i = 0 ; i< heap1.size; ++i){
        result.add(heap1.getData(i));
    }
    return  result;
}
/**
```

N size of first heap. M is size of second heap.
It takes O(nlogn+ mlogm).
If the size of first heap much greater than second then time complexity is O(nlogn).



```
Merging 2 Heap

New Heap preOrder Traversal
12->14->18->35->27->25->100->15->17->39->
```