GEBZE TECHNICAL UNIVERSITY

COMPUTER ENGINEERING

CSE222-2021

HOMEWORK 04 REPORT

AHMET FURKAN KURBAN

1801042674

1 INTRODUCTION
1.1 Problem Definition

You should implement the following features for the Heap structure.
i. Search for an element
ii. Merge with another heap
iii. Removing i_{th} largest element from the Heap
iv. Extend the Iterator class by adding a method to set the value (value passed as parameter) of the last element returned by the next methods.

You should implement a BSTHeapTree class that keeps the elements in a Binary Search Tree where the nodes store max-Heap with a maximum depth of 2 (maximum number of elements included in a node is 7).

System Requirements

BST CLASS

```
public boolean add (E element) {

	rootNode=addElement(rootNode,element);

	return addCompleted;

}

private Node<E> addElement(Node<E> roNode,E element)

Add element if element is bigger than current node , node will change with right node otherwise it change with left node so every element will be add suitable place.

public void inOrderTraverse() {

	inOrderTraverse(rootNode);

}

private void inOrderTraverse(Node<E> rNode)

Traverse the tree controlling under in order rule.

public E find(E element) {

	return find(rootNode,element);

}

private E find(Node<E> rNode,E element)
```

Find the given element in tree if given element is bigger than current node data ,node will be changed with right node of this node otherwise it will changed with left node of this node.

private int numberOfChildren(Node<E> rNode)

To find the children of given element

```
public E delete (E target) {

	Node<E> tempNode=delete(rootNode,target);

	return (tempNode==null) ? null : tempNode.data;

}
```

private Node<E> delete(Node<E> rNode,E target)

Delete the given element if element is bigger than current node , node will change with right node otherwise it change with left node so every element will be deleting from suitable place.

private E findBiggestEndDelete(Node<E> rNode)

Delete biggest element of given node.

public void preOrderTraverse(Node<E> rNode ,int depth , StringBuilder s)

Traverse the tree controlling under pre order rule.

HEAP CLASS (Max Heap)

public boolean add(E element)

Add element to the last and , After adding element if element is bigger than parent , they are swapping then biggest element is at 0.index of array.

private void swap (int parent,int indexOfAdded)

Swap parent with given index element in array.

public int getStorage(int index)

return occurrences of given index

public int setStorage(int index)

find occurrences of given index and change it

public E remove()

Remove the element that in the first element of tree.First we change it with last element and remove it After Removing arrange tree suitable way of heap.

public E SearchElement(E element)

Search element in array.

```
public void MergeAnotherHeap(HeapTree<E> heap)
```

```
given heap elements are adding to the this heap.
```

```
public E removeithLargestElement(int index)
```

Firstly i create a new arraylist then i copy current arraylist to this then i sort the copied arraylist then i find ith largest element easily then i found this element current array and i change it last element and i delete it after deleting i arranged the tree rule of Maxheap.

BSTHEAP CLASS TREE

```
public int add(E element) {

	return add(dataBst.root(),element);

}

private int add(Node<HeapTree<E>> node1,E element)
```

Movement on BST is based on values at the root nodes of the Heap.
If the Heap at a node of BST is full and a new number still needs to be added, a new BST node should be created as the left or the right child of the BST node.

```
public int remove(E element) {

	return remove(dataBst.root(),element);

}

private int remove(Node<HeapTree<E>> node1,E element)
```

Remove operation removes only one occurrence of the value. If the number of occurrences becomes zero than the value will be removed. During a remove operation, if the heap at a node becomes empty, the corresponding BST node will be removed as well. I remove an element that has children and has 1 occurrences if an heap element is removed with deleterec function if left element is available i reach left then i reach last right element then i remove first element of this node or i reach right then i reach last left element then i remove first element of this node it is working until reach leaf element.

```
public void sortNumber()

Sort the array.

public int find(E element) {

	return find(dataBst.root(),element);

}
```

`private int find(Node<HeapTree<E>> node1,E element)`

Find the given element traverse all node and all heap and return number of occurrences.

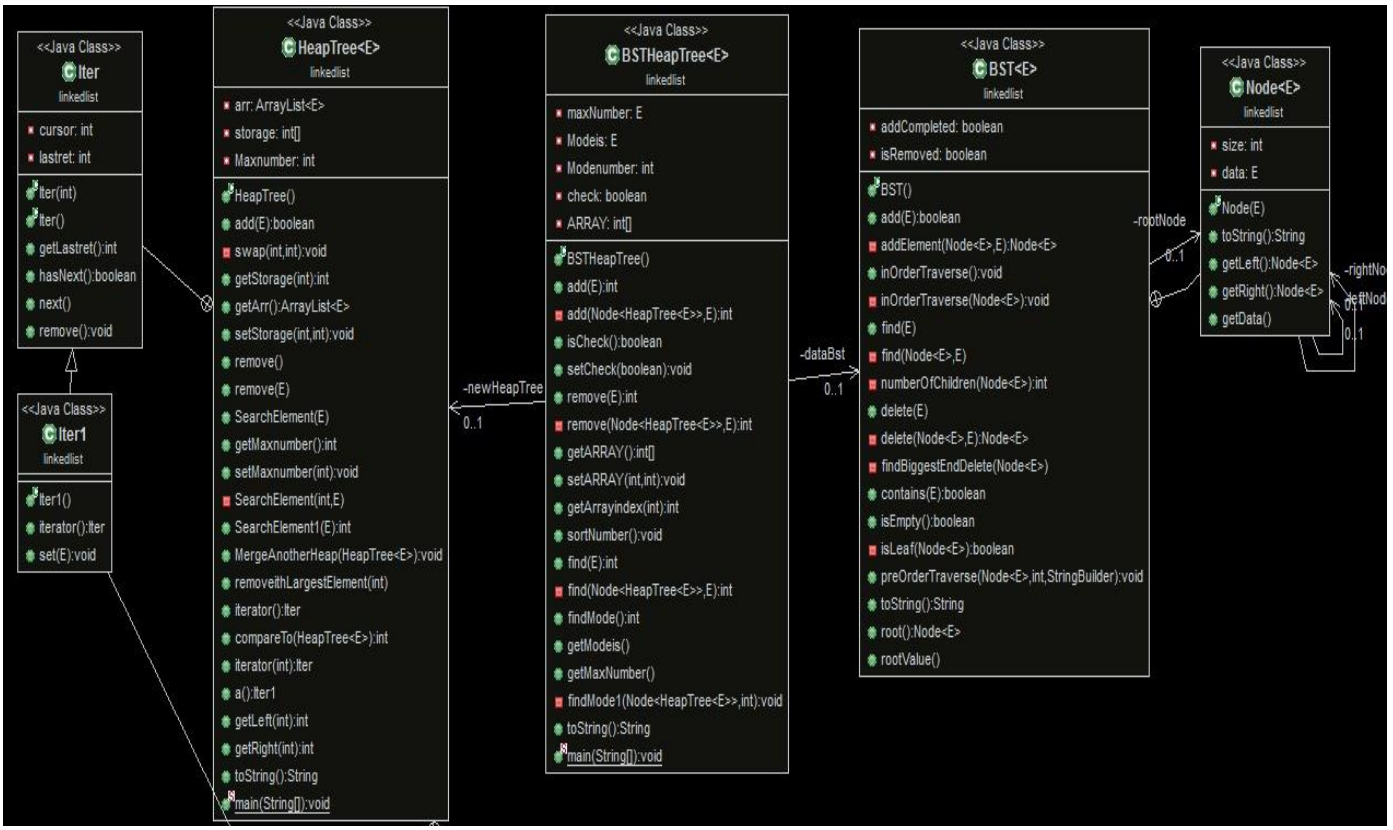
`private void findMode1(Node<HeapTree<E>> node1,int lastMode)`

Fınd the mode of tree and return.

2 METHOD:

2.1 Class Diagrams

UML DİAGRAM



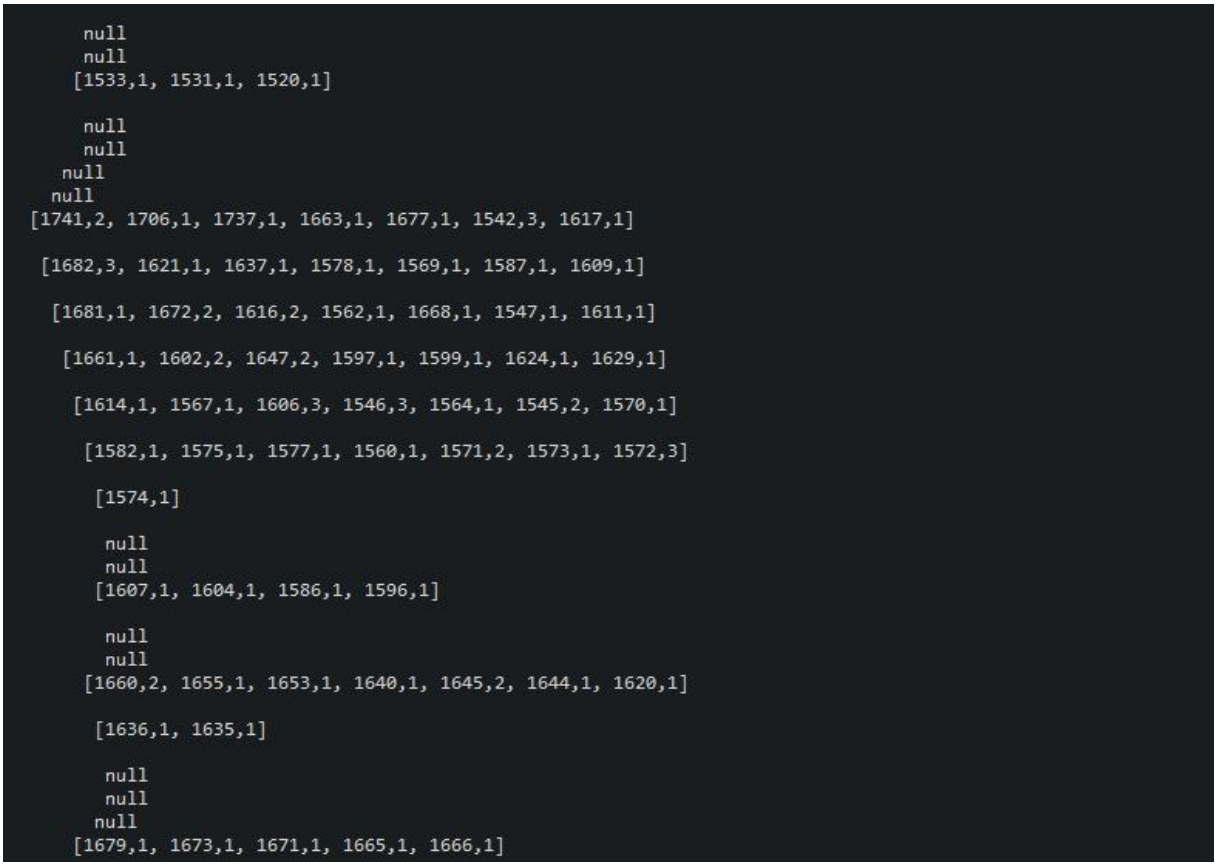
2.3 Problem Solution Approach

Firstly, I wrote binary search tree and maxheap classes, then I created objects using these classes in the binary search tree heap class, which is the class where our main work will be carried out, using those classes, and I performed the desired functions using the functions of these classes and applied 4 tests.

3 RESULT

3.1 Test Cases

Add element:



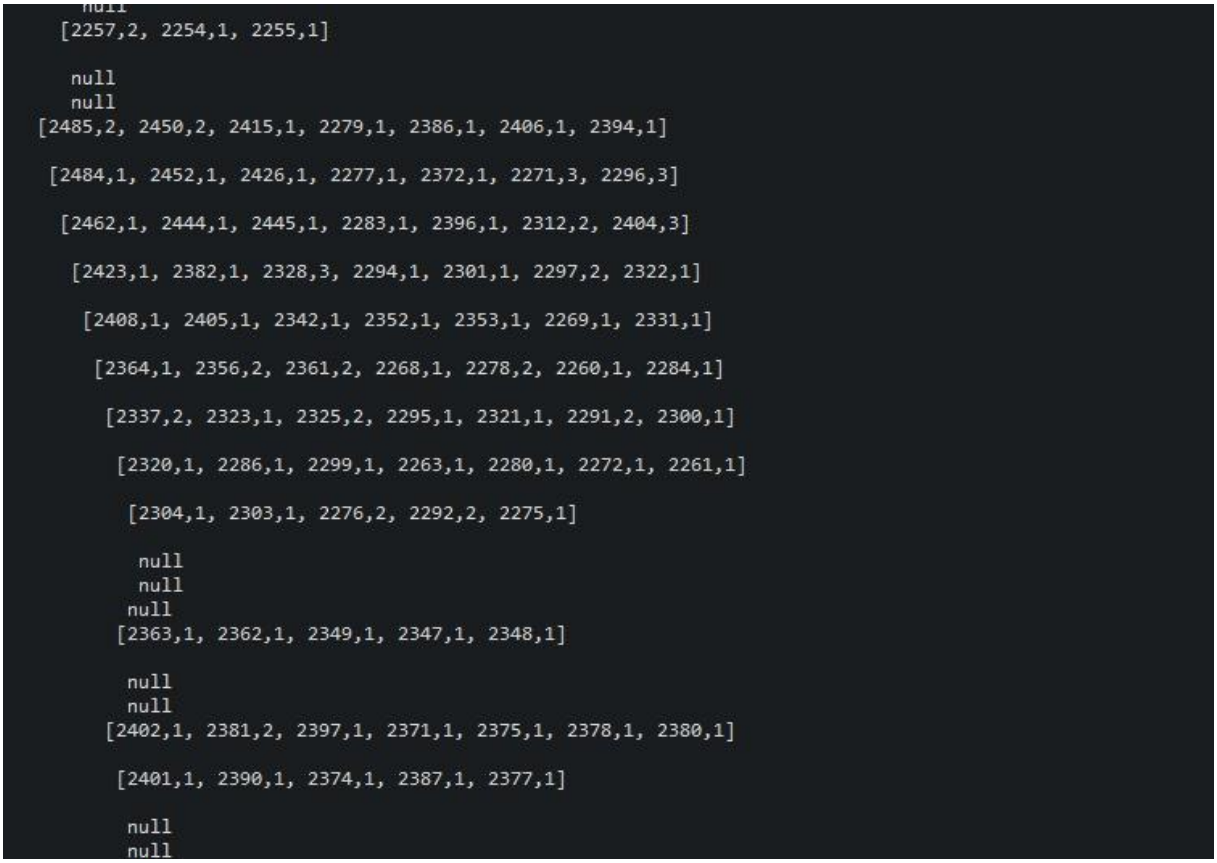
Find element:

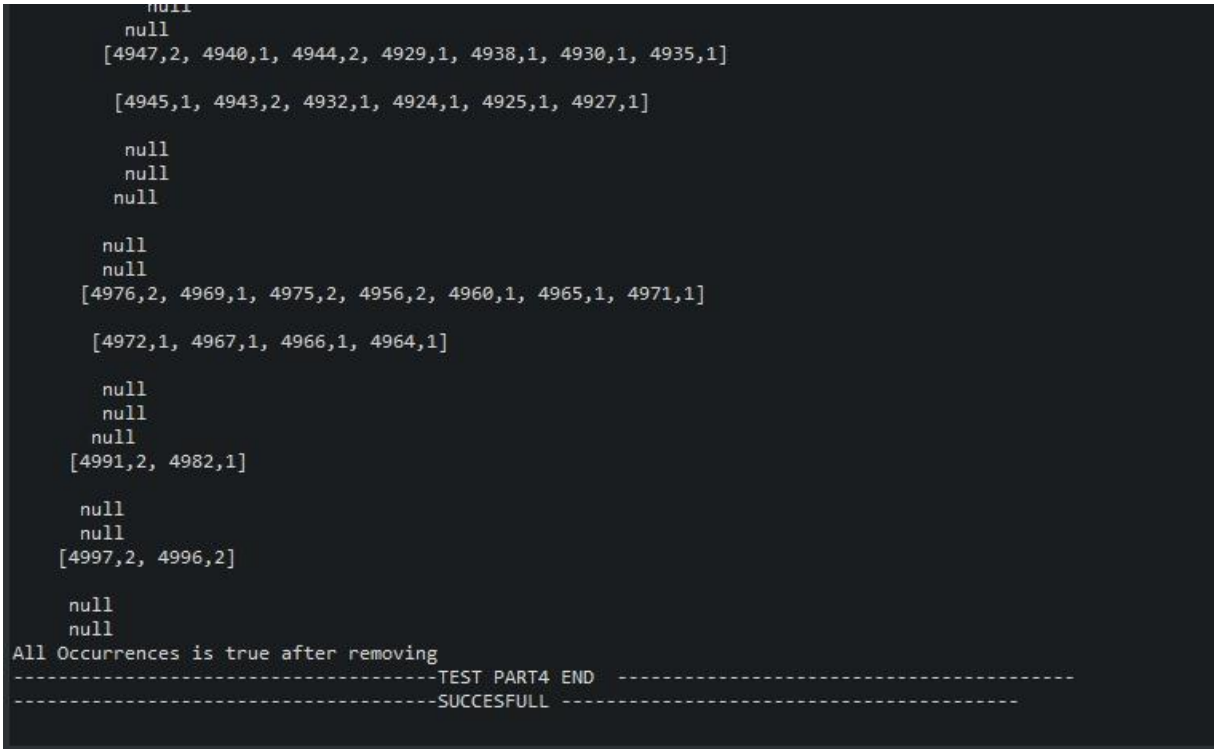
```
All Occurrences is true after adding
---------------------------------------TEST PART1 END  ---------------------------------------
---------------------------------------TEST PART2 START ---------------------------------------
The element is not founded! -> Element: 3567
The element is founded! -> Element: 4476 and element has occurrences :1
The element is not founded! -> Element: 2534
The element is not founded! -> Element: 3178
The element is not founded! -> Element: 3292
The element is not founded! -> Element: 845
The element is not founded! -> Element: 1460
The element is not founded! -> Element: 4468
The element is not founded! -> Element: 4049
The element is not founded! -> Element: 250
The element is not founded! -> Element: 4503
The element is not founded! -> Element: 250
The element is not founded! -> Element: 4177
The element is founded! -> Element: 2043 and element has occurrences :1
The element is not founded! -> Element: 4052
The element is not founded! -> Element: 2436
The element is founded! -> Element: 2183 and element has occurrences :1
The element is not founded! -> Element: 1443
The element is not founded! -> Element: 3212
The element is not founded! -> Element: 552
The element is not founded! -> Element: 1946
The element is not founded! -> Element: 3936
The element is not founded! -> Element: 4667
The element is not founded! -> Element: 2471
The element is founded! -> Element: 2244 and element has occurrences :1
The element is not founded! -> Element: 364
The element is founded! -> Element: 3890 and element has occurrences :1
The element is not founded! -> Element: 3667
The element is not founded! -> Element: 4375
The element is not founded! -> Element: 4295
The element is not founded! -> Element: 823
The element is not founded! -> Element: 3953
```

Mode element:(Part 4 is related with remove)

```
The element is not founded! -> Element: 3102
---------------------------------------TEST PART2 END  ---------------------------------------
---------------------------------------TEST PART3 START  ---------------------------------------
Mode is 3488 Occurrences is -6-
Mode is true!
---------------------------------------TEST PART3 END  ---------------------------------------
---------------------------------------TEST PART4 START  ---------------------------------------
[2684 2669 2671 ]
[1496 1491 1494 1473 1485 ]
[109 101 86 20 81 39 ]
[109 101 86 20 81 39 10 ]
[252 217 233 211 218 221 ]
[229 215 210 196 190 208 ]
[252 217 247 211 218 221 233 ]
[207 198 192 197 ]
[229 215 219 196 190 208 210 ]
[143 47 40 15 0 22 ]
[135 133 ]
[143 47 136 15 0 22 40 ]
[3771 ]
[1426 1414 ]
```

Remove element:

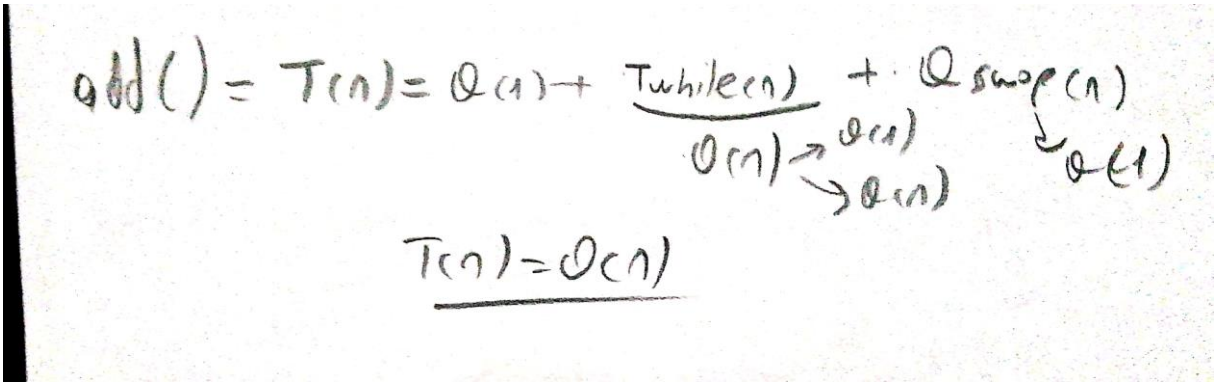




3.2 Asymptotic Notation Results

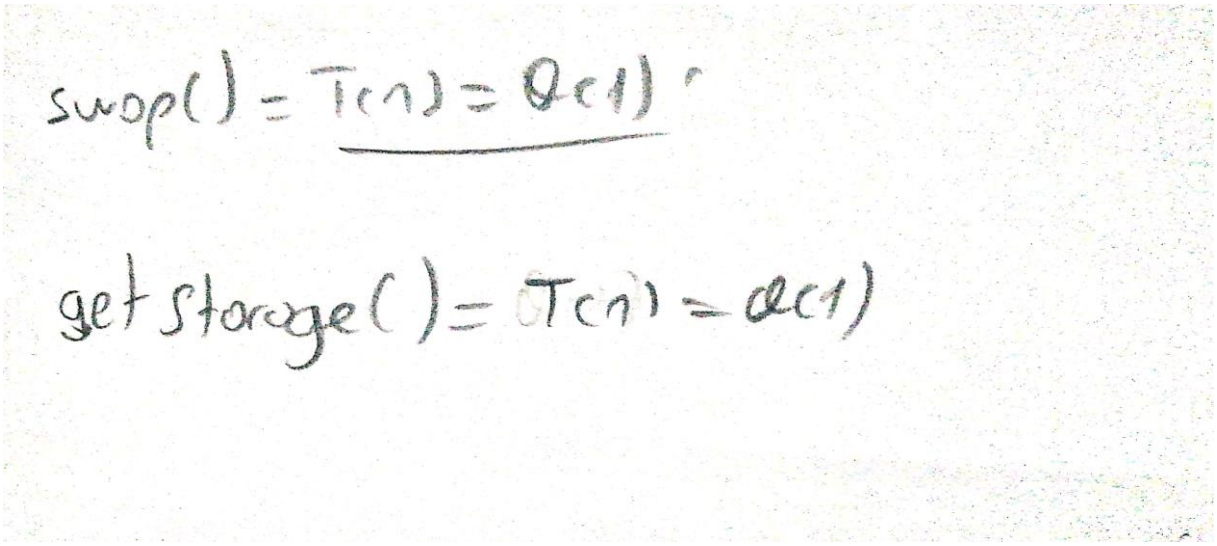
Heap Tree:

```java
public boolean add(E element) {
    arr.add(element);
    int indexOfAdded =arr.size()-1;
    int parent=(indexOfAdded-1)/2;
    while(indexOfAdded>0 && parent>=0 && arr.get(parent).compareTo(arr.get(indexOfAdded))<0) {
        swap(parent,indexOfAdded);
        indexOfAdded=parent;
        parent=(indexOfAdded-1)/2;
    }
    System.out.print(arr);
    return true;
}
```



```
private void swap (int parent,int indexOfAdded) {
    E e =arr.get(parent);
    arr.set(parent,arr.get(indexOfAdded));
    arr.set(indexOfAdded, e);
}
/**
 *
 *
 * @param find occurrences of given index and return it
 * @return return occurrences of given index
 */
public int getStorage(int index) {
    if(index<0 || index>arr.size()) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return storage[index];
}

public ArrayList<E> getArr() {
    return arr;
}
/**
 *
 *
 * @param find occurrences of given index and change it
 *
 */
public void setStorage(int index ,int newValue) {
    if(index<0 || index>arr.size()) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    storage[index]=newValue;
}
```



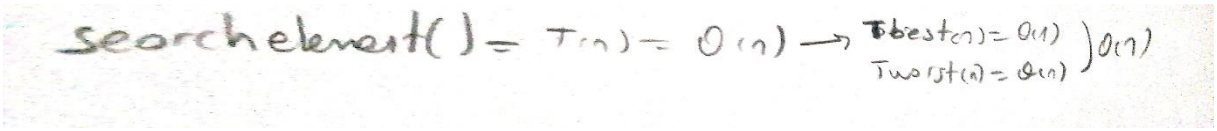
```java
public E remove() {
    int left,right,currentIndex=0,smallerChild;
    if(arr.size()==0) {
        return null;
    }
    if(arr.size()==1) {
        E e =arr.get(0);
        arr.remove(0);
        return e;
    }
    E e =arr.get(0);
    arr.set(0,arr.get(arr.size()-1));
    arr.remove(arr.size()-1);
    while(true) {
        left=getLeft(currentIndex);
        right=getRight(currentIndex);
        if(left >=arr.size()) {
            break;
        }
        smallerChild=left;
        if(right<arr.size() && arr.get(right).compareTo(arr.get(left)) > 0) {
            smallerChild=right;
        }
        if(arr.get(currentIndex).compareTo(arr.get(smallerChild))<0) {
            E tempE =arr.get(currentIndex);
            arr.set(currentIndex, arr.get(smallerChild));
            arr.set(smallerChild,tempE);
            currentIndex=smallerChild;
        }
        else {
            break;
        }
    }
    System.out.print(arr);
    return e;
}
```

setstorage() = $T(n) = O(1)$

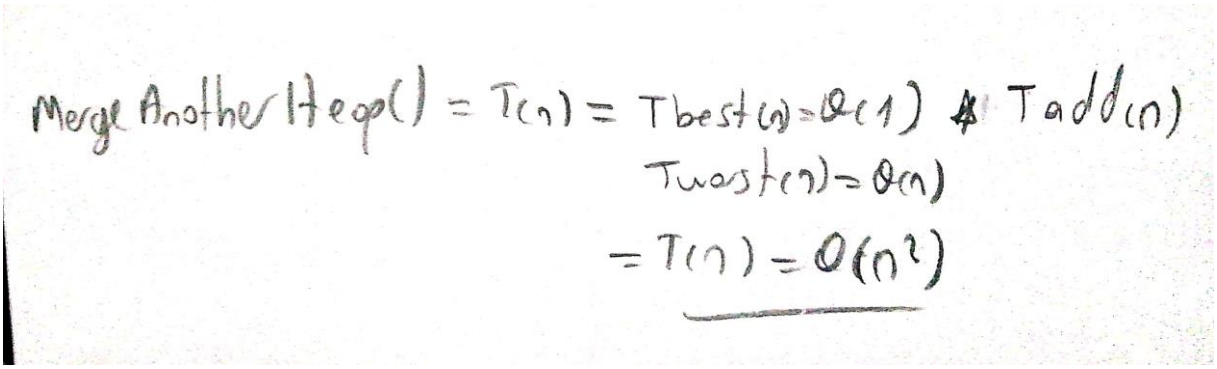
remove() = $T(n) = \underbrace{T_{arr.remove(n)}}_{O(n)} + O(1) + \underbrace{T_{while(n)}}_{O(n)}$

```java
public E SearchElement(E element) {
    E e=SearchElement(0,element);
    if(e==null) {
        System.out.println("Element is not founded! ");
        return null;
    }
    else {
        System.out.println("Element founded! " +e);
        return e;
    }
}

public int getMaxnumber() {
    return Maxnumber;
}
public void setMaxnumber(int maxnumber) {
    Maxnumber = maxnumber;
}
private E SearchElement(int index, E element) {
    if(arr.size()==0) {
        return null;
    }
    for(int i=0;i<arr.size();i++) {
        if(arr.get(i).equals(element)) {
            return arr.get(i);
        }
    }
    return null;
}
public int SearchElement1(E element) {
    if(arr.size()==0) {
        return -1;
    }
    for(int i=0;i<arr.size();i++) {
        if(arr.get(i).equals(element)) {
            return i;
        }
    }
    return -1;
}
```




```java
public void MergeAnotherHeap(HeapTree<E> heap) {
    if(heap.arr.size()==0) {
        return;
    }
    for(int i=0;i<heap.arr.size();i++) {
        this.add(heap.arr.get(i));
    }
}
```



```java
public E removeithLargestElement(int index) {
    int left,right,currentIndex=0,smallerChild;
    ArrayList<E> newList=new ArrayList<>();
    for(int i=0;i<arr.size();i++) {
        newList.add(arr.get(i));
    }
    for(int i=0;i<newList.size();i++) {
        for(int j=i+1;j<newList.size();j++) {
            if(newList.get(i).compareTo(newList.get(j))<0){
                E e=newList.get(i);
                newList.set(i, newList.get(j));
                newList.set(j, e);
            }
        }
    }
    System.out.println(newList);
    int index1=this.SearchElement1(newList.get(index-1));
    if(index1==arr.size()-1) {
        E e =arr.get(arr.size()-1);
        arr.remove(arr.size()-1);
        return e;
    }
    E e =arr.get(index1);
    arr.set(index1,arr.get(arr.size()-1));
    arr.remove(arr.size()-1);
    int parent=(index1-1)/2;
    if(arr.get(index1).compareTo(arr.get(parent))>0) {
        while(index1>0 && parent>=0 && arr.get(parent).compareTo(arr.get(index1))<0) {
            swap(parent,index1);
            index1=parent;
            parent=(index1-1)/2;
        }
    }
```

```java
else {
    currentIndex=index1;
    while(true) {
        left=getLeft(currentIndex);
        right=getRight(currentIndex);
        if(left >=arr.size()) {
            break;
        }
        smallerChild=left;
        if(right<arr.size() && arr.get(right).compareTo(arr.get(left)) > 0) {
            smallerChild=right;
        }
        if(arr.get(currentIndex).compareTo(arr.get(smallerChild))<0) {
            E tempE =arr.get(currentIndex);
            arr.set(currentIndex, arr.get(smallerChild));
            arr.set(smallerChild,tempE);
            currentIndex=smallerChild;
        }
        else {
            break;
        }
    }
}
return e;
```

removeithlargestelement() = $T(n) = \underbrace{T_{add}(n)}_{O(n)} + \underbrace{T_{for}(n)}_{O(n^2)} + \underbrace{T_{search}(n)}_{O(n)} + \underbrace{T_{while}(n)}_{O(n)}$

$$T(n) = O(n^2)$$

```java
public int getLeft(int i) {
    return i*2+1;
}
public int getRight(int i) {
    return i*2+2;
}
public String toString() {
    StringBuilder stringBuilder=new StringBuilder();
    stringBuilder.append("[");
    for (int i = 0; i <arr.size(); i++) {
        if(arr.size()-1==i) {
            stringBuilder.append(arr.get(i)+","+storage[i]);
        }
        else {
            stringBuilder.append(arr.get(i)+","+storage[i]+","+" ");
        }
    }
    stringBuilder.append("]\n");
    return stringBuilder.toString();
}
```

Getleft=O(1)

GetRight=O(1)

toStrıng=O(N)

```java
public void remove()throws ConcurrentModificationException {
    int left,right,currentIndex,smallerChild;
    if (lastret < 0)
        throw new IllegalStateException();
    try {
        E e = arr.get(lastret);
        arr.set(lastret,arr.get(arr.size()-1));
        arr.remove(arr.size()-1);
        currentIndex=lastret;
        while(true) {
            left=getLeft(currentIndex);
            right=getRight(currentIndex);
            if(left >=arr.size()) {
                break;
            }
            smallerChild=left;
            if(right<arr.size() && arr.get(right).compareTo(arr.get(left)) > 0) {
                smallerChild=right;
            }
            if(arr.get(currentIndex).compareTo(arr.get(smallerChild))<0) {
                E tempE =(E) arr.get(currentIndex);
                arr.set(currentIndex, arr.get(smallerChild));
                arr.set(smallerChild,(E) tempE);
                currentIndex=smallerChild;
            }
            else {
                break;
            }
        }
        cursor = lastret;
        lastret = -1;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
```

```java
public void set(E element) {
    int currentIndex,left,right,smallerChild;
    if (getLastret() < 0)
        throw new IllegalStateException();

    arr.set(getLastret(), (E) element);
    int index1=getLastret();
    int parent=(index1-1)/2;
    if(arr.get(index1).compareTo(arr.get(parent))>0) {
        while(index1>0 && parent>=0 && arr.get(parent).compareTo(arr.get(index1))<0) {
            swap(parent,index1);
            index1=parent;
            parent=(index1-1)/2;
        }
    }
    else {
        currentIndex=index1;
        while(true) {
            left=getLeft(currentIndex);
            right=getRight(currentIndex);
            if(left >=arr.size()) {
                break;
            }
            smallerChild=left;
            if(right<arr.size() && arr.get(right).compareTo(arr.get(left)) > 0) {
                smallerChild=right;
            }
            if(arr.get(currentIndex).compareTo(arr.get(smallerChild))<0) {
                E tempE =(E) arr.get(currentIndex);
                arr.set(currentIndex, arr.get(smallerChild));
                arr.set(smallerChild,tempE);
                currentIndex=smallerChild;
            }
            else {
                break;
            }
        }
    }
}
```

compareTo = $\Theta(1)$

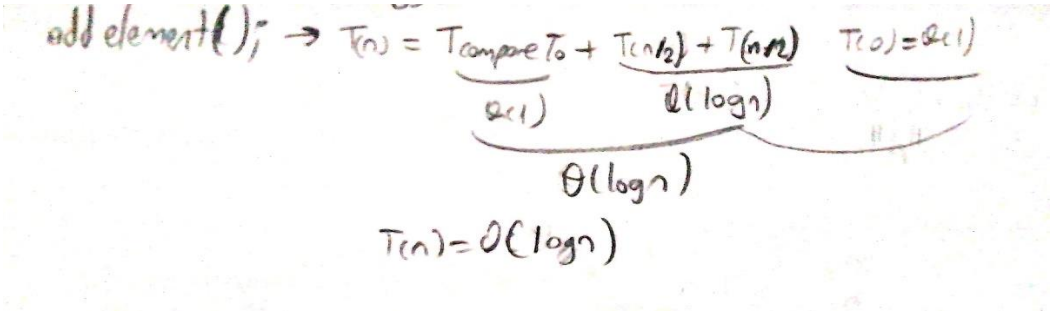
getLeft, getRight() = $\Theta(1)$, tostring() = $\Theta(n)$

Titerator remove() = $\Theta(n)$

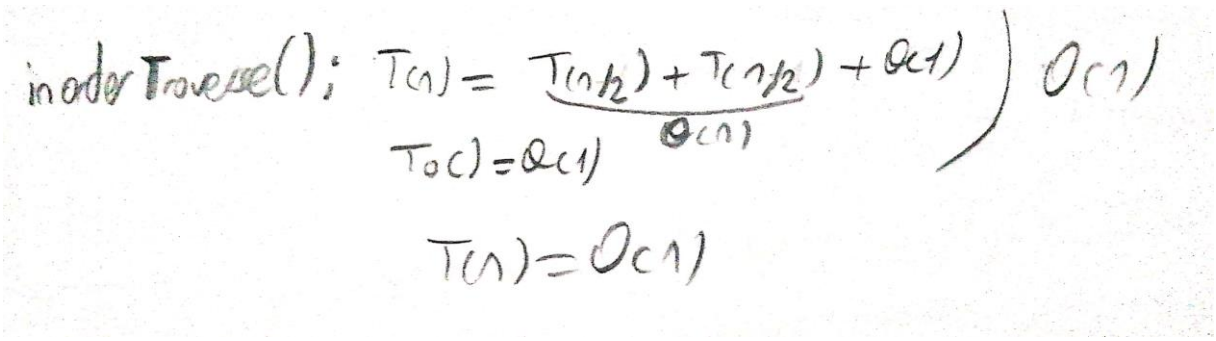
Titerator set() = $\Theta(n)$

BinarySearchTree:

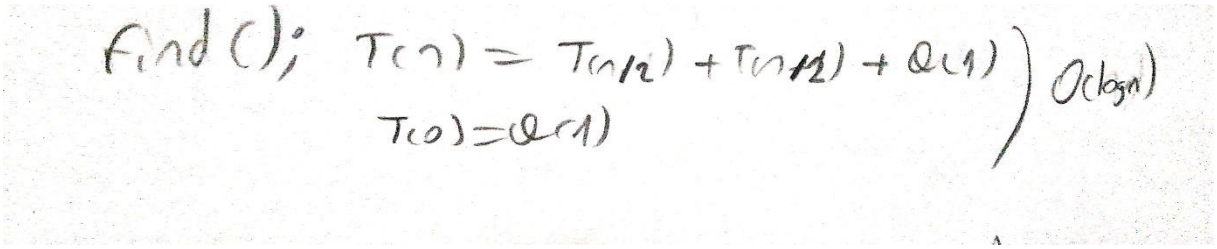
```java
public boolean add (E element) {
    rootNode=addElement(rootNode,element);
    return addCompleted;
}
private Node<E> addElement(Node<E> roNode,E element){
    if (roNode==null) {
        roNode=new Node<E>(element);
        addCompleted=true;
    }
    else if(element.compareTo(roNode.data)==0) {
        addCompleted=false;
    }
    else if(element.compareTo(roNode.data)<0) {
        roNode.leftNode=addElement(roNode.leftNode, element);
    }
    else {
        roNode.rightNode=addElement(roNode.rightNode, element);
    }
    return roNode;
}
```



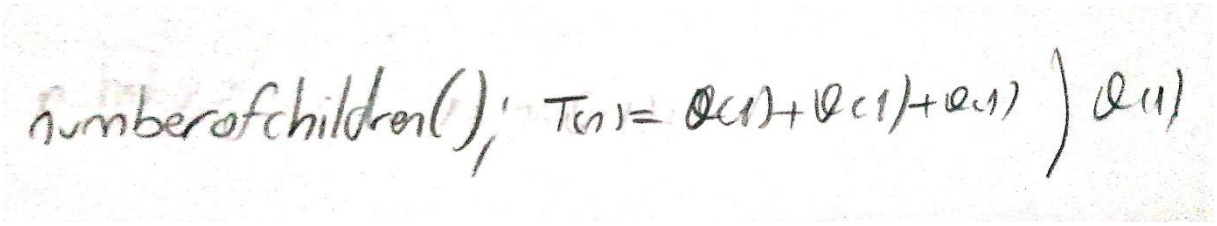

```java
public void inOrderTraverse() {
    inOrderTraverse(rootNode);
}
private void inOrderTraverse(Node<E> rNode) {
    if(rNode==null) return;
    inOrderTraverse(rNode.leftNode);
    System.out.print(rootNode.data+ " ");
    inOrderTraverse(rNode.rightNode);
    if (rNode==rootNode) {
        System.out.println("Reaching The Ancestor Rooté!!");
    }
}
```



```java
public E find(E element) {
   return find(rootNode,element);
}
private E find(Node<E> rNode,E element) {
    if(rNode==null) {
        return null;
    }
    else if(element.compareTo(rNode.data)>0) {
        System.out.println(rNode.data+ " is smaller than " + element + " going to the right");
        return  find(rNode.rightNode,element);
    }
    else if(element.compareTo(rNode.data)==0) {
        System.out.println(rNode.data+ " is equal to the " + element);
        return rNode.data;
    }
    else {
        System.out.println(rNode.data+ " is greater than " + element + " going to the left");
        return find(rNode.leftNode,element);
    }
}
```

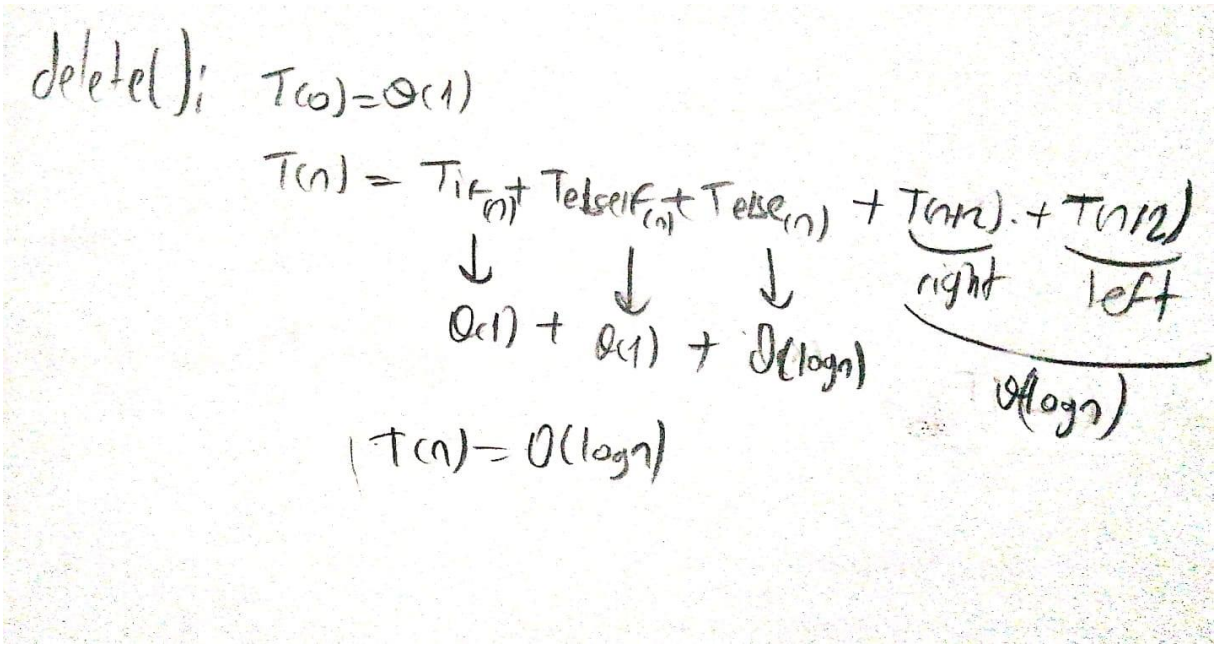



```java
private int numberOfChildren(Node<E> rNode) {
    if(rNode==null) return -1;
    int x=(rNode.rightNode==null) ? 0:1;
    int y=(rNode.leftNode==null) ? 0:1;
    return x+y;
}
```



```java
public E delete (E target) {
    Node<E> tempNode=delete(rootNode,target);
    return (tempNode==null) ? null : tempNode.data;
}
```

```java
private Node<E> delete(Node<E> rNode,E target){
    if(rNode==null) return null;
    else if (target.compareTo(rNode.data)==0) {
        if(isLeaf(rNode))
        {
            isRemoved = true;
            if(rNode == rootNode)
                rootNode = null;
            return null;
        }
        else if(numberOfChildren(rNode) == 1) {
            Node<E> child = (rNode.leftNode!=null) ? rNode.leftNode : rNode.rightNode;
            if(rNode == rootNode) {
                rootNode = child;
            }
            isRemoved = true;
            return child;
        }
        else {
            Node<E> leftChild = rNode.leftNode;
            if(leftChild.rightNode == null) {
                rNode.data = leftChild.data;
                rNode.leftNode = leftChild.leftNode;
                return rNode;
            }else {
                E biggestOfSmallers=  findBiggestEndDelete(leftChild);
                rNode.data = biggestOfSmallers;
                return rNode;
            }
        }
    }
    else if(target.compareTo(rNode.data)>0) {
        rNode.rightNode=delete(rNode.rightNode, target);
        return rNode;
    }
    else {
        rNode.leftNode=delete(rNode.leftNode, target);
        return rNode;
    }
}
```



```java
private E findBiggestEndDelete(Node<E> rNode) {
        if(rNode.rightNode.rightNode == null) {
            E temp = rNode.rightNode.data;
            rNode.rightNode = rNode.rightNode.leftNode;
            return temp;
        }
        return findBiggestEndDelete(rNode.rightNode);
    }
public boolean contains(E element) {
    return find(element)!=null;
}
public boolean isEmpty() {
    return rootNode==null;
}
private boolean isLeaf(Node<E> rNode) {
    return (rNode!=null &&rNode.leftNode==null && rNode.rightNode==null);
}
public void preOrderTraverse(Node<E> rNode ,int depth , StringBuilder s) {
    for(int i=0;i<depth;i++) {
        s.append(" ");
    }
    if(rNode==null) {
        s.append("null\n");
    }
    else {
        s.append(rNode.toString());
        s.append("\n");
        preOrderTraverse(rNode.leftNode, depth+1, s);
        preOrderTraverse(rNode.rightNode, depth+1, s);
    }
}
public String toString() {
    StringBuilder aBuilder=new StringBuilder();
    preOrderTraverse(rootNode,1,aBuilder);
    return aBuilder.toString();
}
```

contains = find = $O(n)$

isempty = $O(1)$

isleaf() = $O(1)$

findbiggestandDelete(): $T(n) = T(n/2) + O(1)$, $T(0) = O(1)$) $O(\log n)$

preordertraverse(): $T(0) = O(1)$

$T(n) = O(1) + T(n/2) + T(n/2)$, $O(1)$) $T(n) = O(n)$

root() = $O(1)$

rootvalue() = $O(1)$

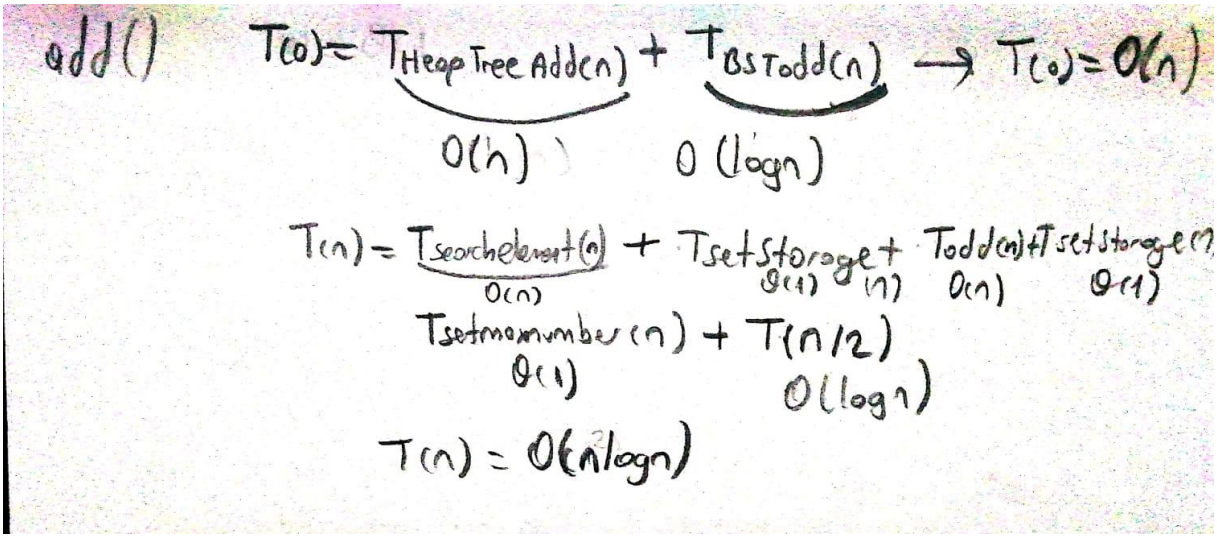
toString(): T(Preordertraverse) $+ O(1) = T(n) = O(n)$

BinarySearchTreeHeap:

```java
public int add(E element) {
    return add(dataBst.root(),element);
}
private int add(Node<HeapTree<E>> node1,E element) throws NoSuchElementException {
    if(element==null) {
        throw new NoSuchElementException();
    }
    if(dataBst.rootValue()==null || node1==null) {
        newHeapTree=new HeapTree<>();
        newHeapTree.add(element);
        newHeapTree.setMaxnumber(1);
        newHeapTree.setStorage(0,1);
        dataBst.add(newHeapTree);
        return 1;
    }
    int index=node1.getData().SearchElement1(element);
    if(index!=-1) {
        node1.getData().setStorage(index,node1.getData().getStorage(index)+1);
        return node1.getData().getStorage(index);
    }
    else if(index==-1 && node1.getData().getMaxnumber()!=7){
        node1.getData().add(element);
        node1.getData().setStorage(node1.getData().getArr().size()-1,1);
        node1.getData().setMaxnumber(node1.getData().getMaxnumber()+1);
        return 1;
    }
    else {
        if(element.compareTo(node1.getData().getArr().get(0))>0) {

            add(node1.getRight(),element);
        }
        else {

            add(node1.getLeft(),element);
        }
    }
    return -1;
}
```



```
private int remove(Node<HeapTree<E>> node1,E element) throws NoSuchElementException {
    if(element==null) {
        throw new NoSuchElementException();
    }
    if(node1!=null) {
        int index=node1.getData().SearchElement1(element);
        if(index==0) {
            if(node1.getData().getArr().size()==1) {
                if(node1.getData().getStorage(index)!=1) {
                    node1.getData().setStorage(index,node1.getData().getStorage(index)-1);
                    return node1.getData().getStorage(index);
                }
                else {
                    node1.getData().remove();
                    dataBst.delete(node1.getData());
                    return 0;
                }
            }
            else {
                if(node1.getData().getStorage(index)!=1) {
                    node1.getData().setStorage(index,node1.getData().getStorage(index)-1);
                    return node1.getData().getStorage(index);
                }
                else {
                    node1.getData().remove();
                    deleterec(node1);
                    return 0;
                }
            }
        }
```

```
        }
        else if(index==-1) {
            if(element.compareTo(node1.getData().getArr().get(0))>0) {

                remove(node1.getRight(),element);
            }
            else {

                remove(node1.getLeft(),element);
            }
        }
        else{
            if(node1.getData().getStorage(index)!=1) {
                node1.getData().setStorage(index,node1.getData().getStorage(index)-1);
                return node1.getData().getStorage(index);
            }
            else if(node1.getData().getStorage(index)==1) {
                if(node1.getData().getLeft(index) > node1.getData().getArr().size() && node1.getData().getRight(index)>node1.getData().getArr().size()) {
                    node1.getData().getArr().remove(index);
                    if(node1.getData().getArr().size()==0) {
                        dataBst.delete(node1.getData());
                        return 0;
                    }
                }
                else {
                    node1.getData().remove(element);
                    deleterec(node1);
                    return 0;
                }
            }
        }
    }
    return -1;
}
```

```
private void deleterec(Node<HeapTree<E>> node1) {
    if(node1==null) {
        return;
    }
    if(node1.getLeft()!=null) {
        Node<HeapTree<E>> teNode=node1;
        teNode=teNode.getLeft();
        while (teNode.getRight()!=null) {
            teNode=teNode.getRight();
        }
        E e=teNode.getData().getArr().get(0);
        teNode.getData().remove();
        node1.getData().add(e);
        node1=teNode;
        deleterec(node1);
    }
    else if(node1.getRight()!=null){
        Node<HeapTree<E>> teNode=node1;
        teNode=teNode.getRight();
        while (teNode.getLeft()!=null) {
            teNode=teNode.getLeft();
        }
        E e=teNode.getData().getArr().get(0);
        node1.getData().add(e);
        teNode.getData().remove();
        node1=teNode;
        deleterec(node1);
    }
}
```

remove() = $T_{searchelement}(n)$ + $T_{theoremove}(n)$ + $T_{BSTdelete}(n)$ +
$T_{deleterec}(n)$ + $T(n/2)$

$T(n/2)$ + add remove $O(\log n)$

$\Theta(\log n)$ $O(n)$

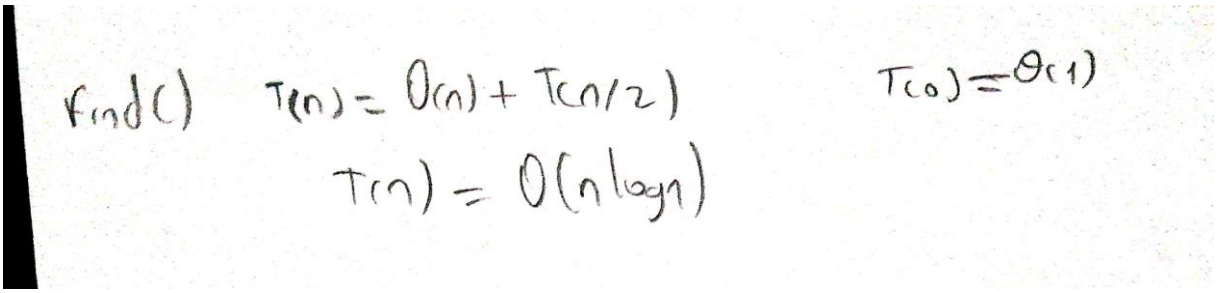
$T(n) = O(n)$

```
public void sortNumber() {
    for(int i=0;i<3000;i++) {
        for(int j=i+1;j<3000;j++) {
            if(getArrayindex(i)<getArrayindex(j)){
                int e=getArrayindex(i);
                setARRAY(i, getArrayindex(j));
                setARRAY(j, e);
            }
        }
    }
}
```

sortnumber() = $T(n) = \Theta(n^2) + \Theta(1)$

$T(n) = \Theta(n^2)$

```java
public int find(E element) {
    return  find(dataBst.root(),element);
}
private int find(Node<HeapTree<E>> node1,E element) {
    if(node1==null) {
        return -1;
    }
    else if(element.compareTo(node1.getData().getArr().get(0))>0) {
        return  find(node1.getRight(),element);
    }
    else if(element.compareTo(node1.getData().getArr().get(0))<0) {
        int index=node1.getData().SearchElement1(element);
        if(index!=-1) {
            return node1.getData().getStorage(index);
        }
        return  find(node1.getLeft(),element);
    }
    else {
        return node1.getData().getStorage(0);
    }
}
```



```java
public int findMode(){
    Modenumber=0;
    findMode1(dataBst.root(),0);
    return Modenumber;
}

public E getModeis() {
    return Modeis;
}
public E getMaxNumber() {
    return maxNumber;
}
private void findMode1(Node<HeapTree<E>> node1,int lastMode) {
    if(node1==null) {
        if(lastMode>Modenumber)
            Modenumber=lastMode;
            Modeis=maxNumber;
    }
    else {
        for(int i=0;i<node1.getData().getArr().size();i++) {
            if(node1.getData().getStorage(i)>lastMode) {
                lastMode=node1.getData().getStorage(i);
                maxNumber=node1.getData().getArr().get(i);
            }
        }
        findMode1(node1.getRight(),lastMode);
        findMode1(node1.getLeft(),lastMode);
    }
}
public String toString() {
    StringBuilder aBuilder=new StringBuilder();
    dataBst.preOrderTraverse(dataBst.root(),1,aBuilder);
    return aBuilder.toString();
}
```

findMode() = $T(n) = O(\frac{n}{2}) + O(n)$ = $T(n) = O(n \log n)$

toString() = preorderTraverse = $T(n) = O(n)$