Part 1 ‘ s methods:

```java
public myHeap() {
    arr = new ArrayList<E>();
}
/**
 * constructs the heap
 * if it is a max Heap than isMax is true
 * @param isMax if it is true than heap is
 */
protected myHeap(boolean isMax) {
    if(isMax){
        multiplier = -1;
    }
    arr = new ArrayList<E>();
}
/**
 * returns the number of elements
 * @return the number of elements
 */
public int size() {
    return arr.size();
}
```

```java
protected boolean indexAvailable(int i) {
    return (i>= 0 && i<arr.size());
}
protected int getParent(int index) {
    return (index-1)/2;
}
E rootValue() {
    return (arr.size() == 0) ? null : arr.get(0);
}
```

These are all Teta(1) functons

Add function

```java
/**
 * adds the given element
 * @param e added element
 * @return always return true
 */
public boolean add(E e) {
    arr.add(e);
    int indexOfAdded = arr.size()-1;
    pushUp(indexOfAdded);
    return true;
}
```

Add functions = Amortized constont time arr is an ArrayList

```java
private void pushUp(int index) {
    int parentIndex = getParent(index);
    while(indexAvailable(parentIndex)) {
        if(arr.get(index).compareTo(arr.get(parentIndex))* multiplier < 0) {
            swap(index,parentIndex);
        }else {
            break;
        }
        index = parentIndex;
        parentIndex = getParent(index);
    }

}
```

GetParent =

IndexAvailable = arr.get =teta(1)

E.compareTo = changes according to the E ' s method I will assume that it is teta(1)

```java
protected void swap(int i,int j) {
    E e = arr.get(i);
    arr.set(i,arr.get(j));
    arr.set(j,e);
}
```

Swap = teta(1)

```java
protected int getParent(int index) {
    return (index-1)/2;
}
```

Because of getParent , in every step index is divided by 2 so while loop goes logn times in the worst case

Best case Teta(1)

So pushUp and add method's complexity is O(logn)

Remove function:

```java
public E remove() {
    if(arr.size() == 0) {
        return null;
    }
    E e = arr.get(0);
    arr.set(0,arr.get(arr.size()-1));
    arr.remove(arr.size()-1);

    putDown(0);

    return e;

}
```

Arr.Get = set = size = remove(size()-1) =teta(1)

```java
private void putDown(int index) {

    int left,right,curIndex;
    curIndex = index;
    int smallerChild;
    while(true) {
        left = getLeft(curIndex);
        right = getRight(curIndex);
        if(left >= arr.size()) {
            break;
        }

        smallerChild = left;
        if(right < arr.size() && arr.get(right).compareTo(arr.get(left)) * multiplier < 0) {
            smallerChild = right;
        }
        if(arr.get(curIndex).compareTo(arr.get(smallerChild))* multiplier > 0 ) {
            E x = arr.get(curIndex);
            arr.set(curIndex,arr.get(smallerChild));
            arr.set(smallerChild,x);
            curIndex = smallerChild;
        }else {
            break;
        }
    }

}
```

```java
protected static int getLeft(int i) {
    return i*2+1;
}
/**
 * returns the index of right child of g
 * @param i index whose right child's ir
 * @return index of right child
 */
protected static int getRight(int i) {
    return i*2+2;
}
```

GetLeft getRight = teta(1)

There is no function which has bigger complexity than Teta(1) in putDown function and in every turn index is multiplied by 2 so while loop goes log(index) times at most.

PutDown:

Best Case Teta(1)

Worst Case Teta(log n)

Average = O(logn)

Other functions of remove has teta(1) complexity so Remove methods complexity is O(logn) too.

Search Function

```java
public boolean search(E e) {
    if(arr == null || arr.size() == 0) {
        return false;
    }
    return search(0,e);

}
```

```java
protected boolean search(int index,E e) {
    int result;
    if(index >= arr.size() ||((result = arr.get(index).compareTo(e))*multiplier > 0)) {
        return false;
    } else if(result == 0) {
        return true;
    } else {
        return (search(index*2+1,e) || search(index*2+2,e));
    }
}
```

Best case is teta(1)

Worst case is teta(logn)

Average: O(logn)

```
protected E get(int i) {
    return arr.get(i);
}
/**
 * returns true if the heap
 * @return
 */
public boolean isEmpty() {
    return arr.size() == 0;
}
```

Get and isEmpty = Teta(1)

```
public void merge(myHeap<E> heap) {
    int size = heap.size();
    for(int i=0;i<size;i++) {
        add(heap.get(i));
    }
}
```

If n = heap.size() and m = our array's size :

Than it is Teta(n) * O(log(m)) = O(n*log(m))

```
public String toString() {
    return arr.toString();
}
```

It is asymptotic notation teta(n)

```
protected int findIndexOfIthLargest(int index) {
    if(index<1 || index > arr.size())
        throw new IndexOutOfBoundsException();
    ArrayList<E> temp = (ArrayList<E>)arr.clone();
    temp.sort(new myComporator());
    E e = temp.get(temp.size() - index);
    System.out.print("Index of "+e+" ");
    return arr.indexOf(e);
}
```

Arr.clone = teta(n)

Sort = O(nlogn)

So its O(nlogn)

```java
public E removeIthLargestElement(int order) throws IndexOutOfBoundsException {
    if(order > arr.size()) {
        throw new IndexOutOfBoundsException();
    }
    int index = findIndexOfIthLargest(order);
    System.out.println(index+" = index of " + order +" th largest" );
    return remove(index);
}
```

FindIndexOfIthLargest = O(nlogn)

Remove(index) :

```java
protected E remove(int index) {
    if(index == arr.size()-1) {
        return arr.remove(arr.size()-1);
    }
    E e = arr.get(index);
    arr.set(index,arr.get(arr.size()-1));
    arr.remove(arr.size()-1);
    setOrder(index);
    return e;
}
```

Best case teta(1)

```java
protected void setOrder(int index) {
    if(!isBiggerThanParent(index)) {
        pushUp(index);
    }
    else if(!isSmallerThanChild(index)) {
        putDown(index);
    }
}
```

```
private boolean isBiggerThanParent(int index) {
    if(!indexAvailable(index)) {
        throw new IndexOutOfBoundsException();
    }
    int parentIndex;
    if(index == 0) {
        return true;
    }else if(indexAvailable(parentIndex = getParent(index))) {
        if(arr.get(index).compareTo(arr.get(parentIndex))* multiplier >= 0) {
            return true;
        }else {
            return false;
        }
    }else {
        return false;
    }
}
```

IsBiggerThanParent is Teta(1)

PushUp is ,as we said earlier, O(logn)

```
private boolean isSmallerThanChild(int index) {
    if(!indexAvailable(index))
        throw new IndexOutOfBoundsException();
    if(!indexAvailable(getLeft(index))) {
        return true;
    }else if(arr.get(index).compareTo(arr.get(getLeft(index))) * multiplier>0) {
        return false;

    }else if(indexAvailable(getRight(index)) && arr.get(index).compareTo(arr.get(getRight(index))) * multiplier >0) {
        return false;
    }else {
        return true;
    }

}
```

Issmallerthanchild is Teta(1)

Putdown function is ,as we said earlier, O(logn)

So setOrder's and remove(int) 's complexity is O(logn)

```
protected int remove(E val) {
    int index = indexOf(val);
    if(index != -1)
        remove(index);
    return index;
}
```

```
protected int indexOf(E val) {
    return arr.indexOf(val);
}
```

IndexOf's asymptotic notation is O(n)

Remove(int) 's asymptotic notation is O(logn)

So remove(E val) 's asymptotic notation is O(n) + O(logn) = O(n)

```
E rootValue() {
    return (arr.size() == 0) ? null : arr.get(0);
}
```

RootValue = Teta(1)

```
class myComporator implements Comparator<E> {

    // override the compare() method
    public int compare(E s1, E s2)
    {
        return s1.compareTo(s2);
    }
}
```

I used this class to sort ArrayList

CompareTo method's complexity depends on the E , I will assume its compareTo method is teta(1)

```
 @Override
 public Iterator<E> iterator() {
     return new Iter();
 }
```

Iterator = Teta(1)

```
public boolean hasNext() {
    return index != arr.size();
}

@Override
public E next() {
    if(hasNext()) {
        lastReturned = index;
        return arr.get(index++);
    }

    throw new  NoSuchElementException();
}
```

HasNext and next = Teta(1)

```
public HeapIterator heapIterator() {
    return new HeapIterator();
}
```

HeapIterator = Teta(1)

```
public class HeapIterator extends Iter{
    /**
     * sets the last element returned to given value
     * @param e new value of the last returned element
     */
    public void set(E e) {
        if(lastReturned == -1) {
            throw new NoSuchElementException();
        }
        arr.set(lastReturned,e);
        setOrder(lastReturned);

    }
}
```

SetOrder = O(logn) ,as we said earlier

Other functions are Teta(1) so

Set = O(logn)

```java
public E find(E val) {
    return find(0,val);
}

/**
 * searches for the value
 * @param index index of the node who searched if it h
 * @param val searched value
 * @return found element
 */
protected E find(int index, E val) {
    if(index>=arr.size()) {
        return null;
    }
    E element = arr.get(index);
    if(element.compareTo(val) == 0) {
        return element;
    }else if(element.compareTo(val)*multiplier < 0) {
        E found = find(getLeft(index),val);
        if(found != null) {
            return found;
        }else {
            return find(getRight(index),val);
        }
    }else {
        return null;
    }

}
```

In every recursive call, the number of possible positions divided by 2

Because getLeft/right multiplies the index by 2

T(n) = T(n/2) + Teta(1)

It is O(logn)

```java
@Override
public int compareTo(myHeap<E> o) {
    if(arr.size() == 0 || o == null || o.size() == 0)
        throw new NoSuchElementException();
    return arr.get(0).compareTo(o.arr.get(0));
}
```

Teta(1)

```java
@Override
public boolean equals(Object o) {
    if(o == null)
        return false;
    myHeap<E> h = (myHeap<E>) o;
    return arr.get(0).equals(h.arr.get(0));
}
```

Teta(1)

MaxHeap:

```java
public maxHeap() {
    super(true);
}
```

Teta(1)

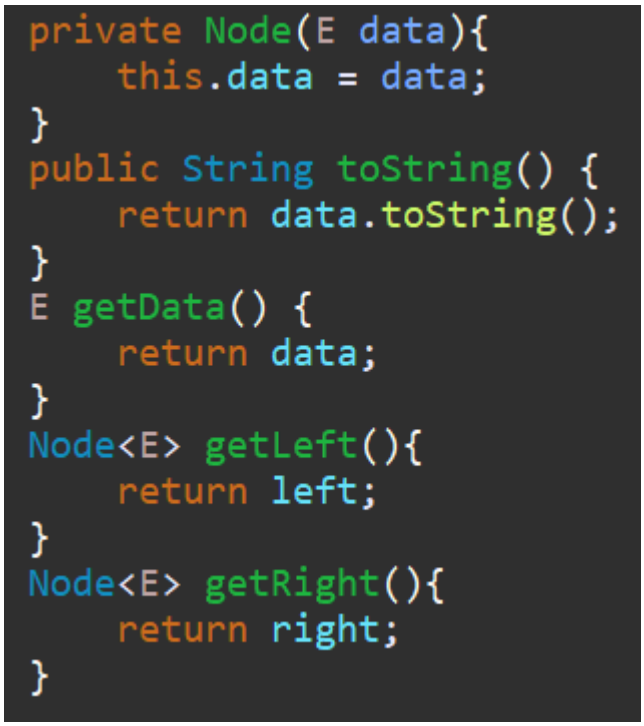
```java
E removeSmallest() {
    if(!arr.isEmpty()) {
        int minIndex = 0;
        for(int i=1;i<arr.size();i++) {
            if(arr.get(i).compareTo(arr.get(minIndex)) < 0) {
                minIndex = i;
            }
        }
        return remove(minIndex);
    }
    throw new NoSuchElementException();

}
```

Remove(int) is O(logn)

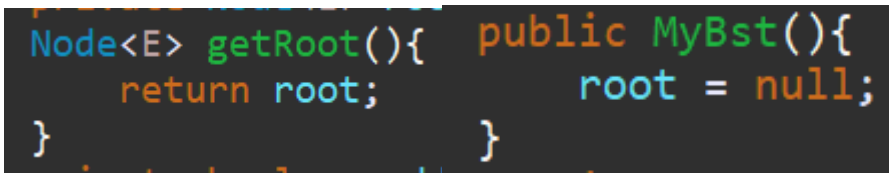
So its asymptotic notation is Teta(n) + O(logn) = Teta(n)

MyBst.Node:



All of them is Teta(1)

MyBst :



Teta(1)

```java
public boolean add(E item) {
    root = add(root,item);
    return addCompleted;
}
/**
 * helper add function
 * @param proot node which is checked if its po
 * @param data value to be added
 * @return true if the data is added
 */
private Node<E> add(Node<E> proot,E data) {
    if(proot == null) {
        proot = new Node<E>(data);
        addCompleted = true;
    }
    else if(data.compareTo(proot.data) == 0) {
        addCompleted = false;
    }
    else if(proot.data.compareTo(data)<0) {
        proot.right = add(proot.right,data);
    }
    else {
        proot.left = add(proot.left,data);
    }
    return proot;
}
```

Add(node,E) = O(logn)

Add(E) = O(logn)

```java
public void inorderTraverse() {
    inorderTraverse(root);
}
/**
 * inorder traverse's helper function
 * @param proot root of the node about to be p
 */
private void inorderTraverse(Node<E> proot) {
    if(proot == null)
        return;
    inorderTraverse(proot.left);
    System.out.print(proot.data+ " ");
    inorderTraverse(proot.right);
    if(proot == root) {
        System.out.println();
    }
}
```

Inorder traverse goes through every element so it Teta(n)

```java
public E find(E item) {
    return find(root,item);
}
/**
 * helper method of the find
 * @param proot root node which is started search
 * @param item searched item
 * @return found element , if it does not returns
 */
private E find(Node<E> proot,E item) {

    if(proot == null)
        return null;
    else if(proot.data.compareTo(item) == 0) {
        //System.out.println(proot.data+ " is equ
        return proot.data;
    }
    else if(proot.data.compareTo(item) < 0 ) {
        //System.out.println(proot.data+ " is sma
        return find(proot.right,item);
    }
    else {
        //System.out.println(proot.data+ " is gre
        return find(proot.left,item);
    }
}
```

Find :

Best case : Teta(1)

Worst = teta(logn)

Average = O(logn)

```java
public boolean contains(E item) {
    return find(root,item) != null;

}
```

Find is O(logn) so contaions is O(logn) too

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

Teta(1)

```java
public E delete(E target) {
    Node<E> temp = delete(root,target);

    if (temp == null)
        return null;

    return temp.data;
}
```

```java
private Node<E> delete(Node<E> proot,E item) {
    if(proot == null) {
        return null;

    }
    else if(proot.data.equals(item)) {
        if(isLeaf(proot))
        {
            isRemoved = true;
            if(proot == root)
                root = null;
            return null;
        }
        else if(numberOfChildren(proot) == 1) {
            Node<E> child = (proot.left!=null) ? proot.left : proot.right;
            if(proot == root) {
                root = child;
            }
            isRemoved = true;
            return child;
        }
        else { // has 2 child
            Node<E> leftChild = proot.left;
            if(leftChild.right == null) {
                proot.data = leftChild.data;
                proot.left = leftChild.left;
                return proot;
            }else {
                E biggestOfSmallers=  findBiggestEndDelete(leftChild);
                proot.data = biggestOfSmallers;
                return proot;
            }
        }
    }
    else if(proot.data.compareTo(item)<0) {
```

```java
private E findBiggestEndDelete(Node<E> proot) {
    if(proot.right.right == null) {
        E temp = proot.right.data;
        proot.right = proot.right.left;
        return temp;
    }
    return findBiggestEndDelete(proot.right);
}
```

```
else if(proot.data.compareTo(item)<0) {
    proot.right = delete(proot.right,item);
    return proot;
}
else {
    proot.left = delete(proot.left,item);
    return proot;
}
```

Find biggest and delete: BestCase = Teta(1) , Worst case = Teta(n) , Average = O(n)

It is called when the element is found,so deleting element when it is found is O(n)

If it is not found, in every step , the number of elements divided by 2 (hopefully) so

Finding an element is O(logn) and deleting it is O(n) so:

O(logn) + O(n) = O(n)

```
static boolean isLeaf(Node prroot) {
    return prroot != null && prroot.left == null && prroot.right == null;
}
```

Teta(1)

```
private void preOrderTraverse(Node<E> proot,int depth,
                              StringBuilder s) {
    for (int i = 0; i < depth; i++) {
        s.append("  ");
    }
    if(proot == null) {
        s.append("null\n");
    }
    else {
        s.append(proot.toString());
        s.append("\n");
        preOrderTraverse(proot.left,depth+1,s);
        preOrderTraverse(proot.right,depth+1,s);
    }
}
```

It goes through every element so it is Teta(n)

```java
public String toString() {
    StringBuilder s = new StringBuilder();
    preOrderTraverse(root,1,s);
    return s.toString();
}
```

ToString = Teta(n)

```java
public boolean isEmpty() {
    return (root == null);
}
/**
 * returns the value of root
 * @return
 */
public E rootValue() {
    return (root == null) ? null : root.data;
}
```

Both are Teta(1)

Cup:

```java
public Cup(E _val) {
    val = _val;
}
/**
 * increases the occurences of the value
 */
public void increaseOccurences() {
    occurences++;
}
/**
 * decreases the occurences of the value
 */
public void decreaseOccurences() {
    occurences--;
}
/**
 * returns the occurences of the value
 * @return the occurences of the value
 */
public int getOccurences() {
    return occurences;
}
/**
 * returns the value
 * @return returns the value
 */
public E getVal() {
    return val;
}
/**
 * compares this cup with the given cup
 *  @param other cup
 */
@Override
public int compareTo(Cup<E> o) {
    return val.compareTo(o.getVal());
}
/**
 * returns the value as a string
 */
public String toString() {
    return String.format(val + ","+occurences );
}
@Override
public boolean equals(Object o) {
    if(o == null)
        return false;
    Cup<E> c = (Cup<E>) o;
    return val.equals(c.val);
}
```

All cup functions are Teta(1)

BSTHeapTree Methods:

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

Teta(1)

```
public int add (E item) {
    size++;
    return add(tree.getRoot(),item);
}
```

```
private int add(MyBst.Node<myHeap<Cup<E>>> root ,E item) {
    if(root == null) {
        Cup<E> temp = new Cup<E>(item);
        maxHeap<Cup<E>> heapAdded = new maxHeap<Cup<E>>();
        heapAdded.add(temp);
        tree.add(heapAdded);
        return 1;
    }

    maxHeap<Cup<E>> heap = (maxHeap<Cup<E>>) root.getData();
    Cup<E> temp = new Cup<E>(item);
    Cup<E> found = heap.find(temp);
    if(found!= null) {
        found.increaseOccurences();
        return found.getOccurences();
    }
    if(heap.size() < MAXSIZEOFHEAP) {
        heap.add(temp);
        return 1;
    }
    else {
        if(heap.rootValue().compareTo(temp) > 0){
            return add(root.getLeft(),item);
        }else {
            return add(root.getRight(),item);
        }
    }
}
```

Best case = tree.add= O(logn)

I will assume that searching a element in a heap is constant time because every heap has 7 numbers of element and O(7) is equal to the O(1) = Teta(1)

Worst case Teta(logn) * Teta(MAXSIZEOFHEAP) , if we assume that MAXSIZEOFHEAP is constant than it is Teta(logn)

So Average is O(logn)

```
}
private int find(MyBst.Node<myHeap<Cup<E>>> r ,  Cup<E> val) {
    if(r == null) {
        return 0;
    }
    myHeap<Cup<E>> heap = r.getData();
    Cup<E> found = heap.find(val);

    if(found != null) {
        return found.getOccurences();
    }else if(heap.rootValue().compareTo(val) > 0) {
        return find(r.getLeft(),val);
    }else {
        return find(r.getRight(),val);
    }
}
public int find(E val) {
    return find(tree.getRoot(),new Cup<E>(val));
}
```

Best case Teta(1)

Worst case Teta(logn)

Average : O(logn)

```java
public E find_mode() {
    mode = null;
    if(tree.getRoot() == null)
        return null;
    find_mode(tree.getRoot());
    return mode.getVal();
}
/**
 * helper method of the find_mode method
 * @param node node may have the mode
 */
private void find_mode(MyBst.Node<myHeap<Cup<E>>> node){
    if(node == null||node.getData() == null) {
        return;
    }
    Cup<E> temp = find_mode_heap(node.getData());
    if(temp != null) {
        if(mode == null) {
            mode = temp;
        }else {
            if(mode.getOccurences() < temp.getOccurences()) {
                mode = temp;
            }
        }
    }
    find_mode(node.getLeft());
    find_mode(node.getRight());
}
```

It searches for every value , so it is Teta(n)

```java
private Cup<E> find_mode_heap(myHeap<Cup<E>> h) {
    if(h == null || h.size() == 0)
        return null;
    int maxOccurIndex = 0;
    for(int i=1;i<h.size();i++) {
        if(h.get(i).getOccurences()> h.get(maxOccurIndex).getOccurences() ) {
            maxOccurIndex = i;
        }
    }
    return h.get(maxOccurIndex);
}
/**
```

In this case max size of the heap is 7 so it is Teta(1) ,but generally speaking , it is Teta(h.size())

```java
public int remove (E item) {
    Cup<E> temp = remove(tree.getRoot(),new Cup<E>(item));
    if(temp == null) return -1;
    size--;
    return temp.getOccurences();
}
```

```java
private Cup<E> remove(MyBst.Node<myHeap<Cup<E>>> node,Cup<E> searched){
    if(node == null) {
        return null;
    }

    if(node.getData().size()==0) {
        throw new NullPointerException();
    }
    int result = node.getData().rootValue().compareTo(searched);
    if( result < 0 ) {
        return remove(node.getRight(),searched); // maxheap does not have
    } else {
        Cup <E> found = node.getData().find(searched);
        if(found != null) {
            if(found.getOccurences() == 1) {
                if(node.getData().size() == 1) {
                    tree.delete(node.getData());

                }
                else {
                    node.getData().remove(found);
                    if(!MyBst.isLeaf(node)) {
                        setProperly(node);
                    }
                }
            }
            found.decreaseOccurences();
            return found;

        } else {
            return remove(node.getLeft(),searched);
        }
    }

}
```

Tree.delete(node.getData()) = O(logn)

Node.getData().remove(found) = O(7) = O(1) = teta(1)

```
private void setProperly(MyBst.Node<myHeap<Cup<E>>>  node) {
    Cup<E> temp = removeBiggest(node.getLeft());
    if(temp == null) {
        temp = removeSmallest(node.getRight());
    }
    node.getData().add(temp);
}
```

```
private Cup<E> removeBiggest(MyBst.Node<myHeap<Cup<E>>>  node){
    if(node == null)
        return null;
    if(node.getRight() == null) {
        Cup<E> temp = node.getData().remove();
        if(node.getData().size() == 0) {
            tree.delete(node);
        }
        if(node.getLeft() != null)
            setProperly(node);
        return temp;
    }
    return removeBiggest(node.getRight());
}
```

Node.getData().remove() is O(logn) but n is at most 7 so Teta(1)

Tree.delete(node) = O(logn)

SetProperly is a undirect recursive call , because setProperly called this function

If S = setProperly and T = removeBiggest

T(n) = T(n/2) + S(n) + O(logn)

```
private Cup<E> removeSmallest(MyBst.Node<myHeap<Cup<E>>>  node){
    if(node == null)
        return null;
    if(node.getLeft() == null) {
        Cup<E> temp = ((maxHeap<Cup<E>>) node.getData()).removeSmallest();
        if(node.getData().size() == 0) {
            tree.delete(node);
        }
        if(node.getRight() != null) {
            setProperly(node);
        }
        return temp;
    }
    return removeSmallest(node.getLeft());
}
```

Node.getData().remove() is O(logn) but n is at most 7 so Teta(1)

Tree.delete(node) = O(logn)

SetProperly is a undirect recursive call , because setProperly called this function

If S = setProperly and T = removeSmallest

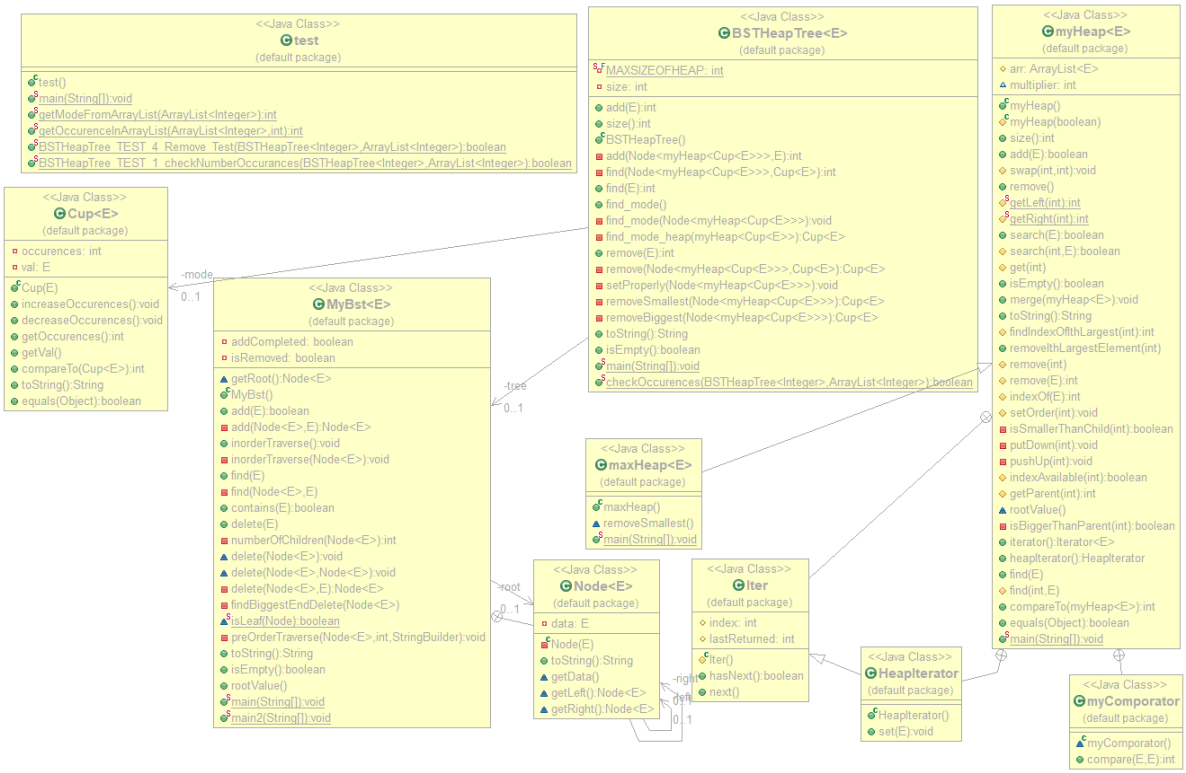
T(n) = T(n/2) + S(n) + O(logn)

Set properly calls only removeSmallest or removeBiggest , and there is no other function call which is greater than Teta(1) in setProperly, so setProperly's asymptotic notation is equal to the these functions's analysis.

But in every call setProperly calls the other function and it keeps getting deeper edges of the tree.

Every setProperly method calls the return(smallest/biggest) function which is not called most recently.

UML diagram


<<Java Class>>
test
(default package)
test()
main(String[]):void
getModeFromArrayList(ArrayList<Integer>):int
getOccurenceInArrayList(ArrayList<Integer>,int):int
BSTHeapTree_TEST_4_Remove_Test(BSTHeapTree<Integer>,ArrayList<Integer>):boolean
BSTHeapTree_TEST_1_checkNumberOccurances(BSTHeapTree<Integer>,ArrayList<Integer>):boolean
<<Java Class>>
BSTHeapTree<E>
(default package)
MAXSIZEOFHEAP: int
size: int
add(E):int
size():int
BSTHeapTree()
add(Node<myHeap<Cup<E>>>,E):int
find(Node<myHeap<Cup<E>>>,Cup<E>):int
find(E):int
find_mode()
find_mode(Node<myHeap<Cup<E>>>):void
find_mode_heap(myHeap<Cup<E>>):Cup<E>
remove(E):int
remove(Node<myHeap<Cup<E>>>,Cup<E>):Cup<E>
setProperly(Node<myHeap<Cup<E>>>):void
removeSmallest(Node<myHeap<Cup<E>>>):Cup<E>
removeBiggest(Node<myHeap<Cup<E>>>):Cup<E>
toString():String
isEmpty():boolean
main(String[]):void
checkOccurences(BSTHeapTree<Integer>,ArrayList<Integer>):boolean
<<Java Class>>
myHeap<E>
(default package)
arr: ArrayList<E>
multiplier: int
myHeap()
myHeap(boolean)
size():int
add(E):boolean
swap(int,int):void
remove()
getLeft(int):int
getRight(int):int
search(E):boolean
search(int,E):boolean
get(int)
isEmpty():boolean
merge(myHeap<E>):void
toString():String
findIndexOfIthLargest(int):int
removeIthLargestElement(int)
remove(int)
remove(E):int
indexOf(E):int
setOrder(int):void
isSmallerThanChild(int):boolean
putDown(int):void
pushUp(int):void
indexAvailable(int):boolean
getParent(int):int
rootValue()
isBiggerThanParent(int):boolean
iterator():Iterator<E>
heapIterator():HeapIterator
find(E)
find(int,E)
compareTo(myHeap<E>):int
equals(Object):boolean
main(String[]):void
<<Java Class>>
Cup<E>
(default package)
occurences: int
val: E
Cup(E)
increaseOccurences():void
decreaseOccurences():void
getOccurences():int
getVal()
compareTo(Cup<E>):int
toString():String
equals(Object):boolean
-mode
0..1
<<Java Class>>
MyBst<E>
(default package)
addCompleted: boolean
isRemoved: boolean
getRoot():Node<E>
MyBst()
add(E):boolean
add(Node<E>,E):Node<E>
inorderTraverse():void
inorderTraverse(Node<E>):void
find(E)
find(Node<E>,E)
contains(E):boolean
delete(E)
numberOfChildren(Node<E>):int
delete(Node<E>):void
delete(Node<E>,Node<E>):void
delete(Node<E>,E):Node<E>
findBiggestEndDelete(Node<E>)
isLeaf(Node):boolean
preOrderTraverse(Node<E>,int,StringBuilder):void
toString():String
isEmpty():boolean
rootValue()
main(String[]):void
main2(String[]):void
-tree
0..1
<<Java Class>>
maxHeap<E>
(default package)
maxHeap()
removeSmallest()
main(String[]):void
-root
0..1
<<Java Class>>
Node<E>
(default package)
data: E
Node(E)
toString():String
getData()
getLeft():Node<E>
getRight():Node<E>
-right
-left
0..1
0..1
<<Java Class>>
Iter
(default package)
index: int
lastReturned: int
Iter()
hasNext():boolean
next()
<<Java Class>>
HeapIterator
(default package)
HeapIterator()
set(E):void
<<Java Class>>
myComporator
(default package)
myComporator()
compare(E,E):int