GIT

DEPARTMENT OF COMPUTER ENGINEERING

CSE 222/505 – Spring 2021

REPORT FOR HOMEWORK 4

SÜLEYMAN GÖLBOL

1801042656

1. ANALYZING TIME COMPLEXITY

When we are using T_{sum} and multiply the time cost with number of times executed, cost doesn't matter because we know it and it is constant. For example,

(for i=0 to n)

    sum = sum + 1;

return sum;

return sum; , *i++*, *i<n* and *sum = sum + 1;* normally takes constant time.

But for loop will work n+1 times so *sum = sum + 1;* will work n times. Θ(n) + Θ(1) + Θ(1)+ Θ(1) = O(n) because constant time doesn't effect when there is a bigger degree like Θ(n).

So when I see a for loop like for i=0 to n, I will directly use Θ(n) (to not mention same thing in every method implementation screenshot.)

Also the getters that just returns constant thing, and setters that equalize something will cost Θ(1) – constant time. So I won't include them in screenshots.

```
public void add(E e){
    heap.add(heap.size(), e); //Firstly adding element to the end of the list.
    largestList.add(e);
    Collections.sort(largestList, Collections.reverseOrder());

    setInitialTree();

    while( isBigger(heap.get(parentIndex), heap.get(childIndex))  && 0 <= parentIndex){
        swapChildAndParent();
    }
}
```

```
@Override
public void setInitialTree(){
    childIndex = heap.size();
    childIndex--;
    parentIndex = (childIndex-1)/(2);
}
```

```
public void swapChildAndParent(){
    // Changing heap values
    E temp = heap.get(parentIndex);
    heap.set(parentIndex, heap.get(childIndex) );
    heap.set(childIndex, temp);
    //Changing index values.
    int temp2 = childIndex;
    childIndex = parentIndex;
    parentIndex = (temp2-1)/2;
}
```

Add method is using arraylist's add method which is Θ(1), and Collections.sort() method takes Θ(n*log(n)) time says Oracle doc. setInitialTree() and swapChildrenAndParent() takes Θ(1) too and while loop will take Θ(n) time. So at the end , add method will take Θ(n*log(n)).

```
public boolean search(E e){
    if(heap.contains(e) == false)
        return false;
    return true;
}
```

Contains method's time complexity is Θ(n) so this method has Θ(n) complexity.

```
public void mergeWithAnotherHeap(MyMinHeap<E> heapToBeMerged){
    MyIterator<E> iterator2 = heapToBeMerged.myIterator();
    while(iterator2.hasNext() ){
        add( iterator2.next() );
    }

}
```

```
public MyIterator<E> myIterator(){
    MyIterator<E> myIterator = new MyIterator<E>(heap);
    return myIterator;
}
```

```
public boolean hasNext() {
    if(current == collect.size())
        return false;
    else
        return true;
}
```

```
public E next(){
    flag_illegalState = false;
    boolean flag = false;
    if(hasNext() == true){
        flag = true;
    }
    if(flag == false){
        flag_illegalState = true; //Because
        throw new NoSuchElementException();
    }
    valueToSet = (E)collect.get(current++);
    return valueToSet;
}
```

myIterator() hasNext() and next() is Θ(1) so because of while loop it will take Θ(n) time.

```java
public int findIndexOfLargest(int largestIndex){
    for(int i=0; i<heap.size(); i++){
        if(heap.get(i) == largestList.get(largestIndex) )
            return i;
    }
    throw new IllegalStateException("The item you wanted to delete couldn't find.");
}
```

Heap and largestList are arraylist so add method is Θ(1). Because of for loop it will take Θ(n) time.

```java
public void removeIthLargestElement(int i){
    //Checking if i is smaller than size.
    if(i > heap.size()){
        System.out.println("Error! Your index is
        return;
    }

    i = findIndexOfLargest(i-1);

    //Copying to a temp list.
    ArrayList<E> tempList = new ArrayList<E>();
    for(int j=0; j<heap.size(); j++){
        if(j < i)
            tempList.add(j, heap.get(j) );
        else if(j == i)
            continue;
        else
            tempList.add(j-1, heap.get(j));
    }

    heap.clear();
    for(int j=0; j<tempList.size(); j++)
        heap.add( tempList.get(j) );

}
```

For loops will take Θ(n) times. And findIndexOfLargest() is also Θ(n). So Θ(n)+ Θ(n)+ Θ(n) => Θ(3*n) is same with Θ(n).

```java
public void set(E value){
    if(flag_illegalState == true)
        throw new UnsupportedOperationException(
    collect.set(current-1, value);
}
```

Collect is an ArrayList and arraylist's set method takes constant time so this is constant time too. Θ(n).

```java
public int incrementNumberOfOccurences(){
    numberOfOccurences++;
    setNotation();
    return numberOfOccurences;
}

public int decrementNumberOfOccurences(){
    numberOfOccurences--;
    setNotation();
    return numberOfOccurences;
}

public void setNotation(){
    notation = value + "," + numberOfOccurences;
}
```

setNotation() just creates a string. If we used concatenate strings it would take big time because immutable. But because of we just create it is Θ(1). So all these 3 methods Θ(1).

```java
public int add(E e){
    return add(e, 1);
}
```

```java
public int add(E e, int numberOfOccurences){
    ListIterator<Data<E>> iterator = heap.listIterator();
    while(iterator.hasNext() == true){
        if(iterator.next().getValue().compareTo(e) == 0){
            int numberOfOccurences2 = iterator.previous().incrementNumberOfOccurences();
            return numberOfOccurences2;
        }
    }
    //If program reaches here that means there is no occurancy for that value before.

    heap.add(heap.size(), new Data<E>(e,numberOfOccurences) ); //Firstly adding element to the end of the list a

    setInitialTree();
    //int index = heap.size()-1;
    while( isSmaller( heap.get(parentIndex).getValue(), heap.get(childIndex).getValue() )  && 0 <= parentIndex){
        swapChildAndParent();
        swapIndexes();
    }
    return 1;
}
```

```java
public void swapIndexes(){
    //Changing index values.
    childIndex = parentIndex;
    parentIndex = (childIndex-1)/2;
}

/**
 * Swapping child and parent and their indexes to heapify.
 */
@Override
public void swapChildAndParent(){
    // Changing heap values
    Data<E> temp = heap.get(parentIndex);
    heap.set(parentIndex, heap.get(childIndex) );
    heap.set(childIndex, temp);
}
```

```java
public void setInitialTree(){
    childIndex = heap.size();
    childIndex--;
    parentIndex = (childIndex-1)/(2);
}
```

listIterator() , incrementNumberOfOccurences(), setInitialTree(), swapChildAndParent(), swapIndexes() takes constant time so because two while loops, Θ(n)+Θ(n) = Θ(2n) which is equals to Θ(n).

```java
public int remove(E item){
    int occurence = 0;
    //Checking if i is smaller than size.

    for(int i=0; i<heap.size(); i++){
        if( heap.get(i).getValue().compareTo(item) == 0){ //if item found
            if( heap.get(i).getNumberOfOccurences() != 1){ //if just there is just one of them, delete.
                occurence = heap.get(i).decrementNumberOfOccurences();
                return occurence;
            }
        }
    }

    // for find index and move to root.
    int i;
    for(i=0; i<heap.size(); i++)
        if( heap.get(i).getValue().compareTo(item) == 0 && heap.get(i).getNumberOfOccurences() == 1){ //
            break;
        }

    if(i == size())
        throw new IndexOutOfBoundsException();

    heap.set(i, heap.get( heap.size()-1 ) );
    heap.remove(heap.size()-1);
    heapify();

    return 1;
}
```

```java
public void heapify(){
    int leftIndex, rightIndex;
    parentIndex = 0;
    while(true){
        leftIndex  = 2*parentIndex+1;
        rightIndex = 2*parentIndex+2;
        if(leftIndex >= size())
            break;
        childIndex = leftIndex; //childIndex will hold the data of minimum child.
        if(rightIndex<size() && isSmaller( heap.get(leftIndex).getValue(), heap.get(rightIndex).getValue() ) )
            childIndex = rightIndex;
        if( isSmaller( heap.get(parentIndex).getValue(), heap.get(childIndex).getValue() ) ){
            swapChildAndParent();
            parentIndex = childIndex;
        }
        else
            break;

    }
}
```

Firstly, heapify() will take Θ(n) because of while loop. The methods iside it are just getters and setters.

Also heap variable is from arraylist. So heap's remove(index) method will take constant time when removing but because of also shifting it will take Θ(n/2) = Θ(n) time.

S because of for loop there are 2 Θ(n)'s. Θ(n)+ Θ(n)+ Θ(n)+ Θ(n) = Θ(4n) = Θ(n).

```java
public int search(E element){
    for(int i=0; i<heap.size(); i++){
        if(heap.get(i).getValue().compareTo(element) == 0)
            return heap.get(i).getNumberOfOccurences();
    }
    return 0;
}
```

Search method will take Θ(n) because of for loop.

```java
public void occurenceRecursive(Node<E> root, E item, boolean rootFlag){
    if(root == null)    return;
    if( root.heap.search(item) != 0){
        occurence(root, item, rootFlag);
        return;
    }

    occurenceRecursive(root.left, item, rootFlag);
    occurenceRecursive(root.right, item, rootFlag);
}
```

```java
public void occurence(Node<E> root, E item, boolean rootFlag){
    //Incrementing size and checking if this is mode.
    int tempOcc = root.heap.incrementNumberOfOccurences(item);
    if(mode.occurence < tempOcc && rootFlag == false){
        mode.occurence = tempOcc;
        mode.value = item;
    }
    else if(rootFlag == true){
        mode.occurence = root.heap.incrementRootOccurence();
        mode.value = root.heap.getRootValue();
    }
}
```

Occurrence method has $\Theta(1)$ complexity.

For recursive algorithm $T(n)=T(n/2) + \Theta(1)$. Because $T(1)=1$. So $T(n)$ is $\Theta(\log_2 n)$ from Master Theorem.

```java
public int add(E item){
    root = add(root,item);
    return find(item);
}
```

```java
public Node<E> add(Node<E> root, E item){

    if(root == null){
        root = new Node<E>( item );
        return root;
    }

    if(find(item) != 0){    //if element found increase occurence
        occurenceRecursive(root, item, false); //false means that this is not root occurence.
        return root;
    }

    if(root != null && root.heap.size() != 7){
        root.heap.add(item);
        return root;
    }

    //if root's depth is full create new nodes in bst.
    if(root.heap.size() == 7 && root.left == null && item.compareTo( root.heap.getRootValue() ) < 0){
        root.left  = new Node<E>(item);
        return root;
    }
    else if(root.heap.size() == 7 && root.right == null && item.compareTo( root.heap.getRootValue() ) > 0){
        root.right = new Node<E>(item);
        return root;
    }

    if(item.compareTo( root.heap.getRootValue() ) < 0){
        if(root.heap.search(item) != 0){
            occurenceRecursive(root, item, false);
            return root;
        }
        root.left = add(root.left, item);
    }
    else if( item.compareTo( root.heap.getRootValue() ) > 0)
        root.right = add(root.right, item);
    else{ //if element to be add equals to element in heap, just increment the occrence.
        //occurence(root, item, true); //true means that this is root element occurence
        if(root.heap.search(item) != 0){
            occurenceRecursive(root, item, false);
            return root;
        }
        return root;
    }

    return root;
```

This is a recursive method too. Inside add method, occurenceRecursive() method takes $\log_2 n$. So $T(n) = T(n/2) + \log_2 n$. If $\log_2 n$ was 1, it would be $T(n) = \Theta(\log_2 n)$. But because of $\log_2 n$, it will take $\Theta((\log_2 n)^2)$ time.

```java
public int remove(E item){
    remove(root, item);
    return root.heap.search(item);
}
```

```java
private Node<E> remove(Node<E> root, E item){
   if(root == null ) //If element couldn't find just return 0.
       return root;

   if(item.compareTo( root.heap.getRootValue() ) < 0){ //if in left side
       if(root.heap.search(item) != 0){
           root.heap.remove(item); //cc
           return root;
       }
       else if(root.left != null){
           root.left = remove(root.left, item);
           return root;
       }
   }
   else if(root.right != null && item.compareTo( root.heap.getRootValue() ) > 0){
       root.right = remove(root.right, item);
       return root;
   }
   else{ //then it means that our item is in root node.
       root.heap.remove(item);
       return root;
   }

   return root;
}
```

Search takes Θ(n) time. (It was calculated above).
Also root.heap.remove(E) method takes Θ(n).
Because of remove method is recursive, T(n)=T(n/2) + Θ(n).
T(n) = 3n-2 so ~ T(n) = Θ(n).

```java
public int find_mode(){
    System.out.println("Mode value is " + mode.value + " and occurence of that value is "+ mode.occurence);
    return mode.occurence;
}

/**
 * Most frequently added element
 * @return value of mode element.
 */
@Override
public E find_modeValue(){
    return mode.value;
}
```

No loop or no other method so Θ(1)

```java
public int find(E item){
    return find(root, item);
}
```

```java
private int find(Node<E> root, E item){
    if(root == null)    return 0;
    if( root.heap.search(item) != 0){
        return root.heap.search(item);
    }
    return find(root.left, item) + find(root.right, item);
}
```

Search gets Θ(n) time(Because it is belong to my rheap class).
And because of this method is recursive, T(n)=T(n/2) + Θ(n).
T(n) = 3n-2 so ~ T(n) = Θ(n).

2. SYSTEM REQUIREMENTS

Firstly, a min heap must be created in main.

```
MyMinHeap<Integer> heap = new MyMinHeap<Integer>();
heap.add(4);
```

After adding elements, for searching we should use search method.

```
System.out.print("Searching for element (Searching for 3 that isn't in array) : ");
System.out.println( heap.search(3) );
System.out.print("Searching for element (Searching for 2 that  is  in  array) : ");
System.out.println( heap.search(2) );
```

Because of search method returns true or false we can print it with System.out.println to check the condition.

```
heap.mergeWithAnotherHeap(heap2);
```

To merge we should create another heap, and then we can use mergeWithAnotherHeap(MyMinHeap<E>) method to merge them.

```
System.out.println("\nRemoving 5th largest element in heap: ");
heap.removeIthLargestElement(5);
```

To remove element we should use removeIthElement(E); method but this system requires to add argument to parameter a value that is not bigger than size of heap. Or it prints an error message.

Another requirement for system is BSTHeapTree.

```
BSTHeapTree<Integer> bst = new BSTHeapTree<Integer>();
int[] array = new int[3000];
```

We can add it like that. And then we can add elements to it with add method.

```
number = random.nextInt(5000);
bst.add(number);
```

For example the code above will add a random integer between 0-5000.

When finding an element we have to use `(bst.find( array[5*i] )` method. We add an E argument for example in the screenshot argument is an integer from an array.

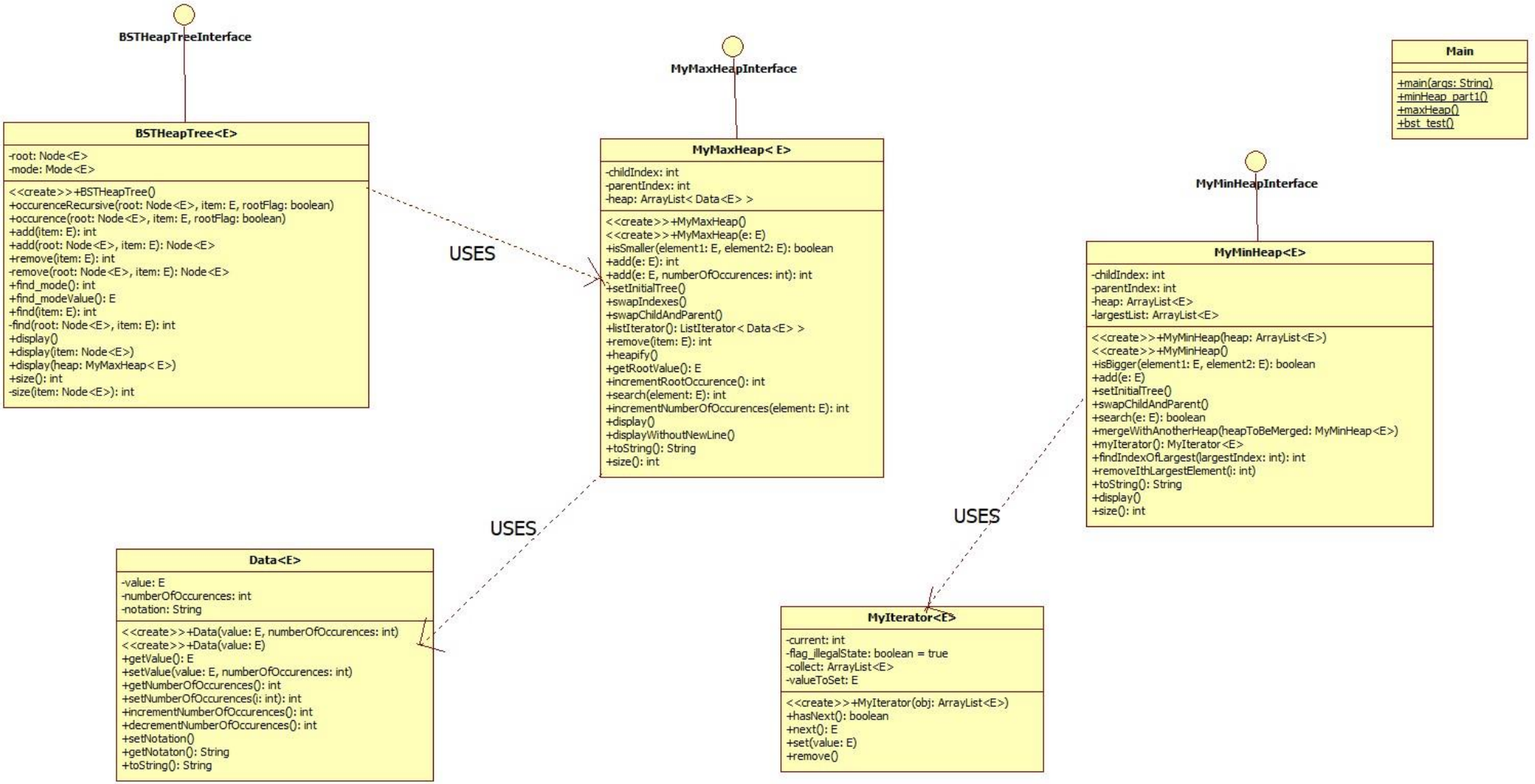
```
int modeOccurence = bst.find_mode();
int modeValue = bst.find_modeValue();
```

Another requirement for bst heap tree is find_mode() method. It returns the occurrence of the most frequent value. And the method below returns mode's value.

```
bst.remove( array[5*i] );
```

Lastly, to remove an element we should use remove method. It gets E as parameter. In example above it gets an int from and int array .

3. CLASS DIAGRAMS



4. PROBLEM SOLUTION APPROACH

My solution approach for Min Heap, I implemented a class called MyMinHeap. I used an ArrayList to hold the the heap tree values. Every time I add an element, I added to the end of the ArrayList and then I set index for child and parent. I used these indexes to swap child and parent if needed because in Min Heap, the root always has to be the smallest item. Also every subtree has to be heaps too. So I used a while loop to fix the heap until heap arraylist becomes a real heap.

Normally remove from heap is easy. All we have to do is removing root value, making the small child of it the new root. But to remove the i'th largest element I had to use another ArrayList called largestList. It hold my data sorted. So when I check the which value I am going to remove, all I have to do is looking to largestList.

Because of the BSTHeapTree is using Max Heap firstly I implemented a MyMaxHeap class. But MaxHeap needed to use both data value and data occurrence in to hold the most frequently added value. So I wrote another class called Data with using generic. It was private static class because I added them inside of another class as inner class.

Also, because of BSTHeapTree has MaxHeap nodes, I created a class called Node and inside of this generic class, I hold the node values as MyMaxHeap. Every time I add to binary search tree , firstly I check if node is full(which means this node has a heap with size = 7) . If it is full, I check if the value to be insert is smaller or bigger than the root of tree. If smaller, I add it to left, and if bigger I add it to right. But if there too many nodes, just an if condition is not enough so I

used recursive algorithm to achieve this. It goes until it finds an empty place to put this value in.
I used the same thing for remove(), size(), find() and display() too. Because without using recursive algorithm, it only finds or checks the current node as root, and its children. Not descendants.

5. TEST CASES

5.1) CREATING MIN HEAP AND ADDING ELEMENTS

```
System.out.println("-------PART1--------");
MyMinHeap<Integer> heap = new MyMinHeap<Integer>();
heap.add(4);
heap.add(2);
heap.add(1);
```

5.2) SEARCHING AN ELEMENT. BOTH SUCCESSFUL AND UNSUCCESSFUL SITUATIONS

```
System.out.print("Searching for element (Searching for 3 that isn't in array) : ");
System.out.println( heap.search(3) );
System.out.print("Searching for element (Searching for 2 that  is  in  array) : ");
System.out.println( heap.search(2) );
```

5.3) USING SET METHOD TO CHANGE LAST VALUE FROM OTERATOR'S NEXT() METHOD

```
MyIterator<Integer> iter = heap2.myIterator();
System.out.println("\nTrying set method without next: ");
try{
    iter.set(3);
}catch(UnsupportedOperationException e){
    System.out.println("Exception caught. Please first use next() for iterator before s
}
System.out.println("Trying set method with next: ");
iter.next();
iter.set(3);
```

5.4) MERGING 2 HEAPS.

```
System.out.println("\nMerged Heap is:");
heap.mergeWithAnotherHeap(heap2);
```

5.5) REMOVING ELEMENTS THAT HAS AND THAT DOESN'T HAVE

```
System.out.println("\nRemoving 5th largest element in heap: ");
heap.removeIthLargestElement(5);
heap.display();
//Removing that is not in heap
System.out.println("Removing 10th largest element in heap: (Heap doesn't have 10 elements
heap.removeIthLargestElement(10);
```

5.6) CREATING BSTHEAPTREE AND RANDOM

```
BSTHeapTree<Integer> bst = new BSTHeapTree<Integer>();
int[] array = new int[3000];
Random random = new Random();
```

5.7) ADDING ELEMENTS AND GENERATING RANDOM NUMBERS

```
for(int i=0; i<3000; i++){ //Adding
    number = random.nextInt(5000);
    bst.add(number);
    array[i] = number;
}
```

5.8) SEARCH ELEMENTS THAT IS IN ARRAY.

```
System.out.println("\nTest 2, searching for 100 numbers that in array:");
boolean errorFlag = false;
for(int i=0; i<100; i++){
    // Every time checking how many array[5*i] are there.
    int counter = 0;
    for(int j=0; j<3000; j++){
        if(array[5*i] == array[j])
            counter++;
    }
    if(bst.find( array[5*i] ) != counter){ //It means that occurences are not corr
        errorFlag = true;
        break;
    }
}
if(errorFlag == true) System.out.println("Error found! Occurences doesn't match.")
else                  System.out.println("Error not found for Q3.1. Occurences are
```

5.9) SEARCH ELEMENTS THAT ISN'T IN ARRAY.

```
System.out.println("\nTest 2, searching for 10 numbers that is not in array.");
errorFlag = false;
for(int i=0; i<10; i++){
    int counter = 0;
    for(int j=0; j<3000; j++){
        if(i+5000 == array[j]){
            counter++;
        }
    }
    if(bst.find(i+5000) == counter && counter != 0){ //i+5000 cannot be in bst so this
        errorFlag = true;
        break;
    }
}
if(errorFlag == true) System.out.println("Error found! Occurences for the numbers not in
else                  System.out.println("Error not found for Q3.2. Occurences for the n
```

5.10) CHECKING FOR MODE

```
System.out.println("\nTest 3, finding mode and checking if true.");
int modeOccurence = bst.find_mode();
int modeValue = bst.find_modeValue();
if( bst.find(modeValue) == modeOccurence )
    System.out.println("Mode value is correct\n");
else
    System.out.println("Mode value is not correct.\n");
```

5.10) REMOVING FROM BST HEAP TREE

```
errorFlag = false;
for(int i=0; i<100; i++){
    // Every time removing array[5*i].
    bst.remove( array[5*i] );
    int counter = 0;
    for(int j=0; j<3000; j++){
        if( array[5*i] == array[j] )
            counter++;
    }
    // Counter is counting occurences from array, and find returns occurence from bst.
    // If removal is successful if condition shouldn't be successful.
    if(bst.find(array[5*i]) != counter-1){  //Counter's number have to 1 bigger because
        System.out.println("Error found so removal is not successful.");
        errorFlag = true;
        break;
    }
}
if(errorFlag == true)
    System.out.println("Error found for Q4.1. After deleting occurences are not correct.
else
    System.out.println("Error not found for Q4.1. After deleting occurences are correct.
```

5.10) TRYING TO REMOVE FROM BST HEAP TREE THAT ISN'T IN.

```
System.out.println("Test 4, removing 10 numbers that aren't in the array.");
errorFlag = false;
for(int i=0; i<10; i++){
    // Every time removing 5000*i. Wont be removed becaouse there no such element.
    try{
        bst.remove( 5000+i );
    }catch(IndexOutOfBoundsException e){
        System.out.println("Exception caught. There are no such element for removing.")
    }
    int counter = 0;
    for(int j=0; j<3000; j++){
        if( 5000+i == array[j] )
            counter++;
    }
    // Counter is counting occurences from array, and find returns occurence from bst.
    if(bst.find(i+5000) == counter-1 && counter != 0){
        errorFlag = true;
        break;
    }
}
if(errorFlag == true)
    System.out.println("Error found for Q4.2. All after deleting occurences are not cor
else
    System.out.println("Error not found for Q4.2. All after, occurences are correct.");
```

6. RUNNING AND RESULTS

Creating heaps and searching if it has or not.

```
sglbl@SglblPC:/mnt/c/Araclar/GTU/2.Sınıf/2.Dönem/Cse222/hw4$ make
javac -d classfiles *.java
java -cp classfiles Main
-------PART1--------
Heap1 is:
[1, 4, 2]
Searching for element (Searching for 3 that isn't in array) : false
Searching for element (Searching for 2 that  is  in  array) : true
Heap2 is:
[5, 8, 6]
```

Setting a value without next and catching error. Setting a value of iterator.next() and using set method.

```
Heap2 is:
[5, 8, 6]

Trying set method without next:
Exception caught. Please first use next() for iterator before setting.
Trying set method with next:
Heap2 is:
[3, 8, 6]
```

Merging heaps.

```
Merged Heap is:
[1, 3, 2, 4, 8, 6]
```

Removing from merged heap firstly that contain, secondly that doesn’t contain.

```
Removing 5th largest element in heap:
[1, 3, 4, 8, 6]
Removing 10th largest element in heap: (Heap doesn't have 10 elements so should give error)
Error! Your index is bigger than size. There is no such element to remove!
[1, 3, 4, 8, 6]
```

Searching for an element in BST Heap Tree that is also in array and checking if occurrences of all these values are correct or not.

```
--------PART2---------

Test 1, Inserting 3000 elements and storing in array and BSTHeapTree.

Test 2, searching for 100 numbers that in array:
Error not found for Q3.1. Occurences are correct.
```

Searching for an element isn’t in BST Heap Tree and checking if occurrences(which is 0 because there is not) of all these values are correct or not.

```
Test 2, searching for 10 numbers that is not in array.
Error not found for Q3.2. Occurences for the nubmers not in array and occurence is true.
```

Trying to find mode and checking if it is true from array using a counter to count the how many times that number has been add.

```
Test 3, finding mode and checking if true.
Mode value is 2341 and occurence of that value is 5
Mode value is correct
```

Removing 100 numbers in the array and checking if occurrences are correct.

```
Test 4, removing 100 numbers in the array.
Error not found for Q4.1. After deleting occurences are correct.
```

Removing 10 numbers in array and checking if new occurrences are correct. (It has to be one smaller from array element because we remove from BSTHeapTree but not array. So the difference of occurrences should be 1 for same element)

```
Test 4, removing 10 numbers that aren't in the array.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Exception caught. There are no such element for removing.
Error not found for Q4.2. All after, occurences are correct.
sglbl@SglblPC:/mnt/c/Araclar/GTU/2.Sınıf/2.Dönem/Cse
```