GTU Department of Computer Engineering
CSE 222/505 – Spring 2020
Homework 4 Report

Ozan GEÇKİN
1801042103

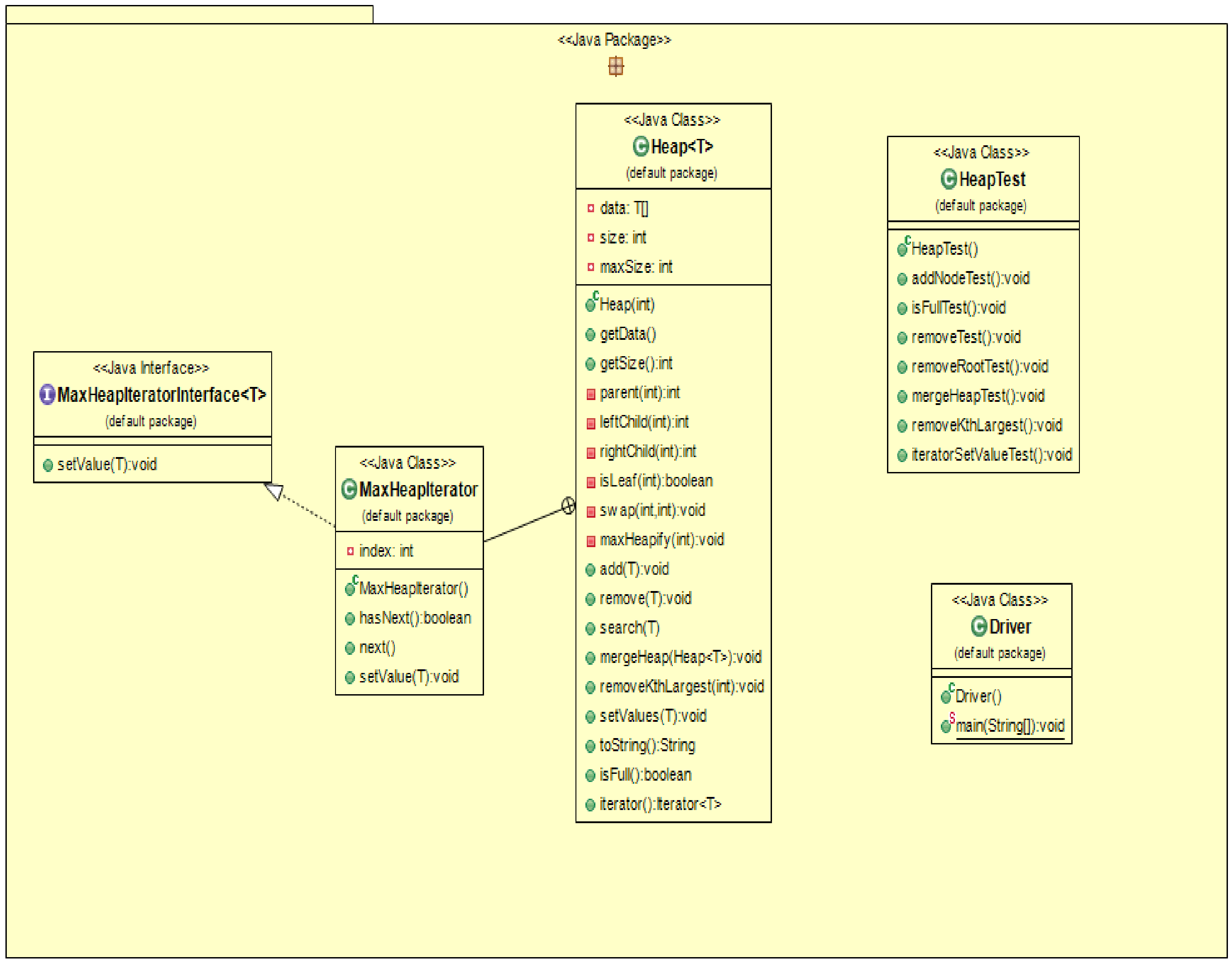
Part-1 Report

Problem:

You should implement the following features for the Heap structure.

i. Search for an element
ii. ii. Merge with another heap
iii. iii. Removing ith largest element from the Heap
iv. 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.

1.Class Diagrams



2.Problem Solution Approach

My solution steps are ;

-Specify the problem requirements
-Analyze the problem
-Design an algorithm and program
-Implement the algorithm
-Test and verify the program
-Maintaion and update the program.

a) **Specify the problem requirements :** I understand the problem.
b) **Analyze the problem :** I identify ; input data, output data additional requirements and constraints.
c) **Design an algorithm and program**: I divide the problem into sub-problems.I listed major steps(sub-problems). I break down each step into a more detailed list.To do these We have to divide this big project into small pieces.
d) **Implement the algorithm:**

A max-heap is a complete binary tree in which the value in each node is greater than or equal to the values in the children of that node.
In Heap.java implementation elements of heap is stored in an array. If a node is stored at index i, then its left child is stored at index 2i+1 and its right child at index 2i+2.

Methods:

add (): We add a new key at the end of the tree. If the new key is smaller than its parent, then we don't need to do anything. Otherwise, we need to traverse up recursively to fix the heap.

remove(): Removes the item from heap. We put the last item in the array into removed one's index and fix the heap.

mergeHeap(): If total size of heaps is less than the maximum size of first heap all items from second heap is added to first heap array in sequence. Then heap is fixed.

search() : Searches the given element in heap. It's linear search.

removeKthLargest(): k th largest means (size-k+1) smallest item. We used a second max-heap to store To store (size-k+1) item. We traverse the heap and add to the second heap. When second heap is full we remove the largest element (root) from second heap. So at the end we have the (size-k+1) smallest item of the first heap in second heap. Maximum of these is the k th largest element of first heap.

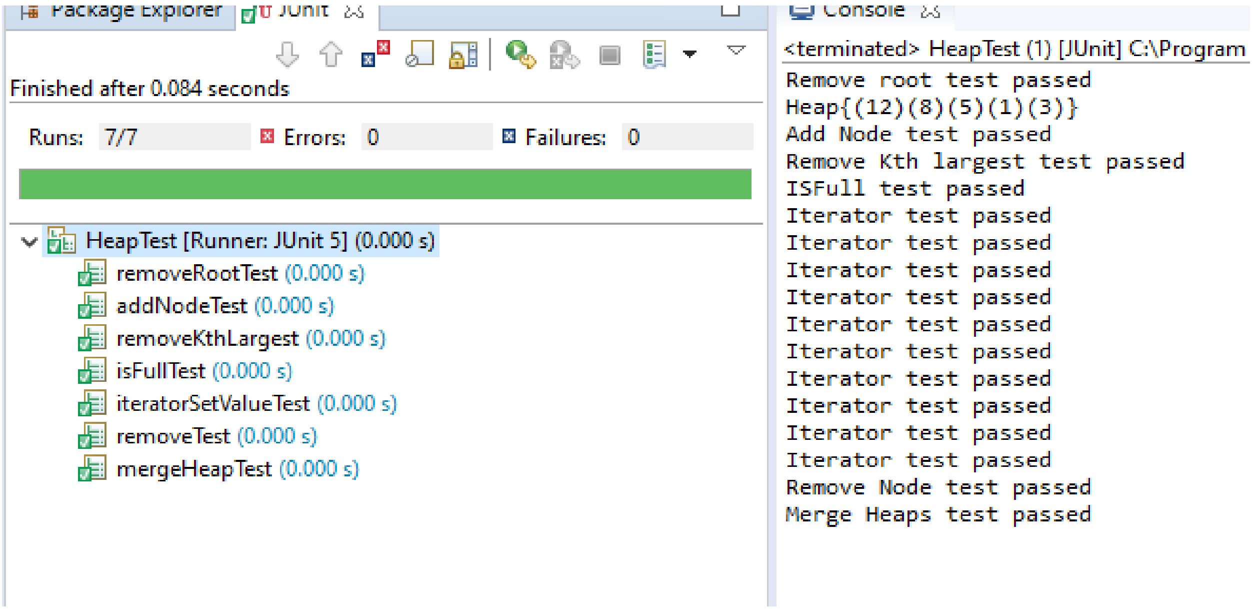
setValue():set the value of the last element returned by the next method to a parameter value.

3.Test Case

Test Case Id	**Test Scenario**	**Test Data**	**Expected Result**	**Actual Result**	**Pass/Fail**
1	Create an integer heap and add data into it	5,2,15,1,67,45,23	67,15,45,1,2,5,23	As expected	Pass
2	Heap get size	Heap<Integer> heap; contet 5,2,15,1,67,45,23	7	As expected	Pass
3	Heap remove integer, by giving both data and index	Heap<Integer> heap; contet 5,2,15,1,67,45,23 -Remove index 2 -"Remove 2" Remove 15	67,5,23,1	As expected	Pass
4	Heap in search	Search element “67”	Founded 67	As expected	Pass
5	Heaps merge	Two heap one heap content 67,5,23,1 other heap content 12,3,26	67,5,23,1,26,3,12	As expected	Pass
6	Create an double heap and add data	67,6-45,4-23,9-7,5-15,3-1,1	67,6-45,4-23,9-7,5-15,3-1,1	As expected	Pass
7	Double heap set value	67,6-45,4-23,9-7,5-15,3-1,1 set value 1,1	1,1-1,1-1,1-1,1-1,1-1,1	As expected	Pass
8	Create an string heap and add data	`Resul,Ozan,Oguzcan,Ayca,Nuray`	`Resul,Ozan, Oguzcan,Ayca,Nuray`	As expected	Pass
9	Heap remove string, by giving both data and index	Heap<String> heap; contet `Resul,Ozan,Oguzcan,Ayca,Nuray` -Remove index 3 -Remove “Ayca” -Remove “Resul”	`Oguzcan,nuray,ayca`	As expected	Pass
10	**Heap search**	Heap<String> heap; contet `Oguzcan,nuray,ayca (nuray)`	`Founded nuray`	As expected	Pass

4.Running and Result

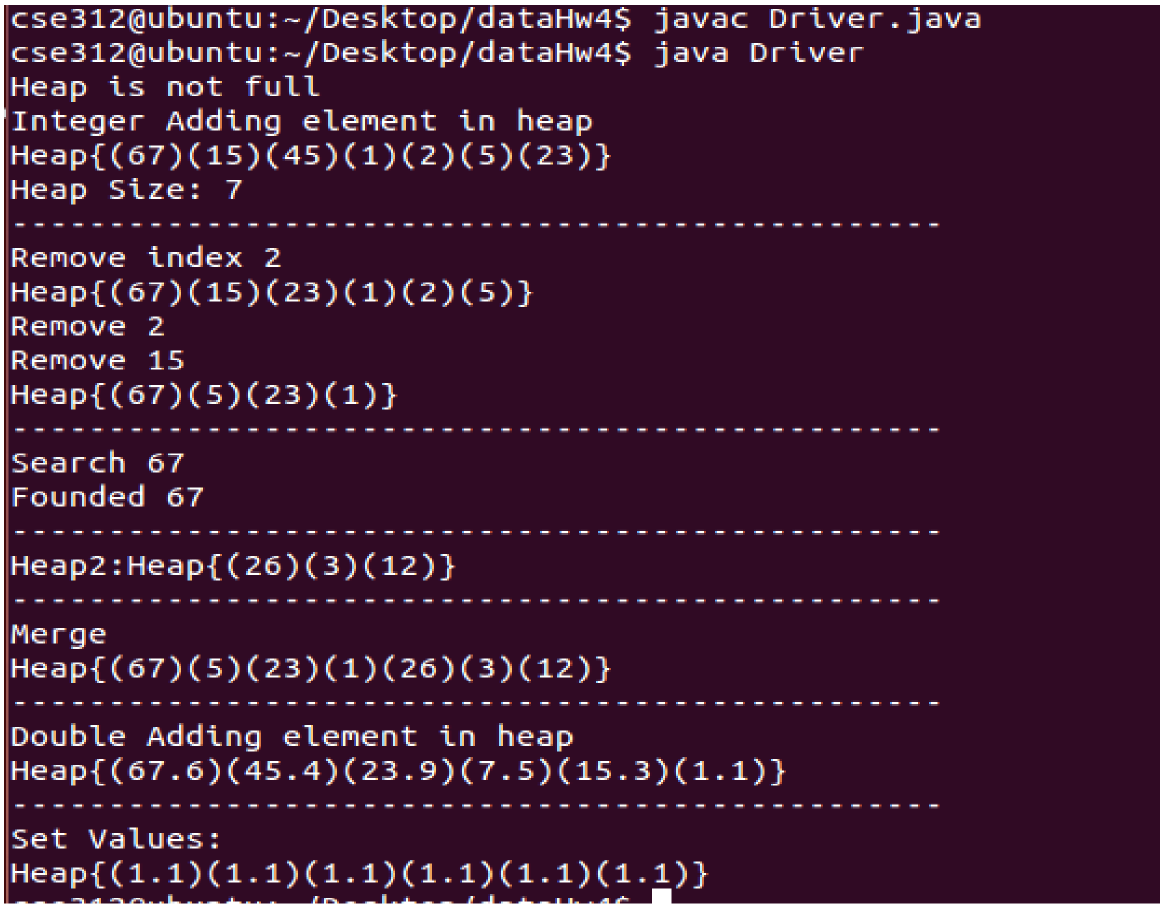
Junit test result:



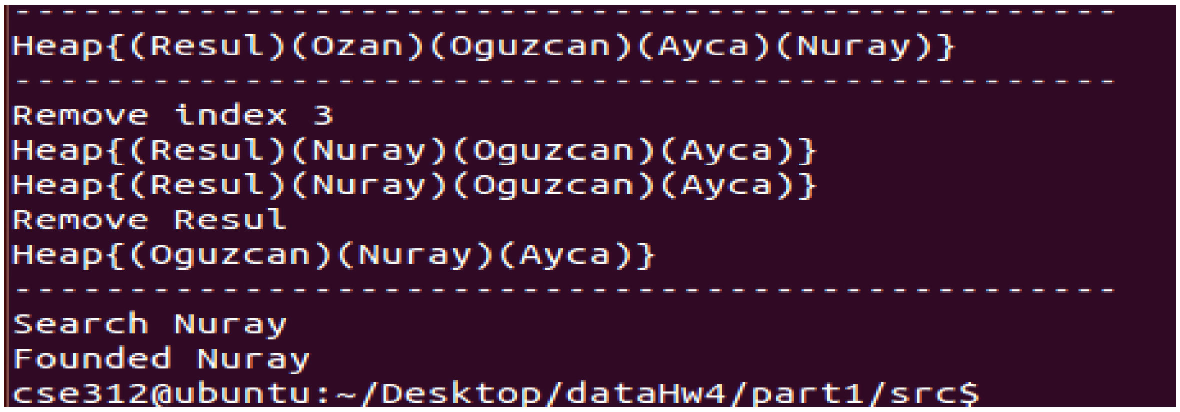
```
Finished after 0.084 seconds

Runs: 7/7    Errors: 0    Failures: 0

HeapTest [Runner: JUnit 5] (0.000 s)
    removeRootTest (0.000 s)
    addNodeTest (0.000 s)
    removeKthLargest (0.000 s)
    isFullTest (0.000 s)
    iteratorSetValueTest (0.000 s)
    removeTest (0.000 s)
    mergeHeapTest (0.000 s)
```

```
<terminated> HeapTest (1) [JUnit] C:\Program
Remove root test passed
Heap{(12)(8)(5)(1)(3)}
Add Node test passed
Remove Kth largest test passed
ISFull test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Iterator test passed
Remove Node test passed
Merge Heaps test passed
```

Test case is done below



```
cse312@ubuntu:~/Desktop/dataHw4$ javac Driver.java
cse312@ubuntu:~/Desktop/dataHw4$ java Driver
Heap is not full
Integer Adding element in heap
Heap{(67)(15)(45)(1)(2)(5)(23)}
Heap Size: 7
------------------------------------------------
Remove index 2
Heap{(67)(15)(23)(1)(2)(5)}
Remove 2
Remove 15
Heap{(67)(5)(23)(1)}
------------------------------------------------
Search 67
Founded 67
------------------------------------------------
Heap2:Heap{(26)(3)(12)}
------------------------------------------------
Merge
Heap{(67)(5)(23)(1)(26)(3)(12)}
------------------------------------------------
Double Adding element in heap
Heap{(67.6)(45.4)(23.9)(7.5)(15.3)(1.1)}
------------------------------------------------
Set Values:
Heap{(1.1)(1.1)(1.1)(1.1)(1.1)(1.1)}
```

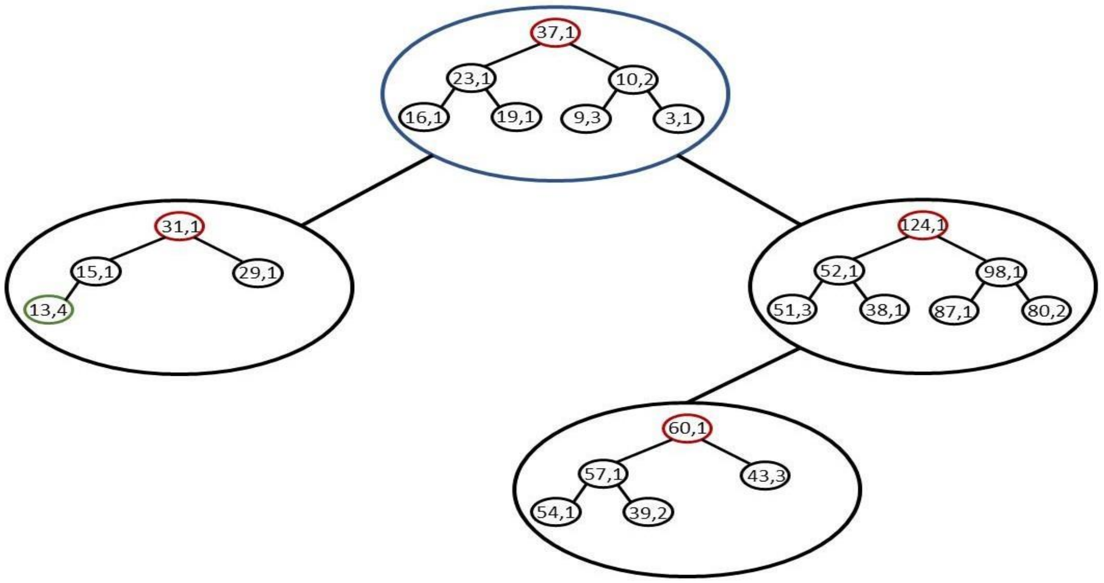


```
----------------------------------------------------
Heap{(Resul)(Ozan)(Oguzcan)(Ayca)(Nuray)}
----------------------------------------------------
Remove index 3
Heap{(Resul)(Nuray)(Oguzcan)(Ayca)}
Heap{(Resul)(Nuray)(Oguzcan)(Ayca)}
Remove Resul
Heap{(Oguzcan)(Nuray)(Ayca)}
----------------------------------------------------
Search Nuray
Founded Nuray
cse312@ubuntu:~/Desktop/dataHw4/part1/src$
```

Part-2 Report

Problem:

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).
An example of this class is below.



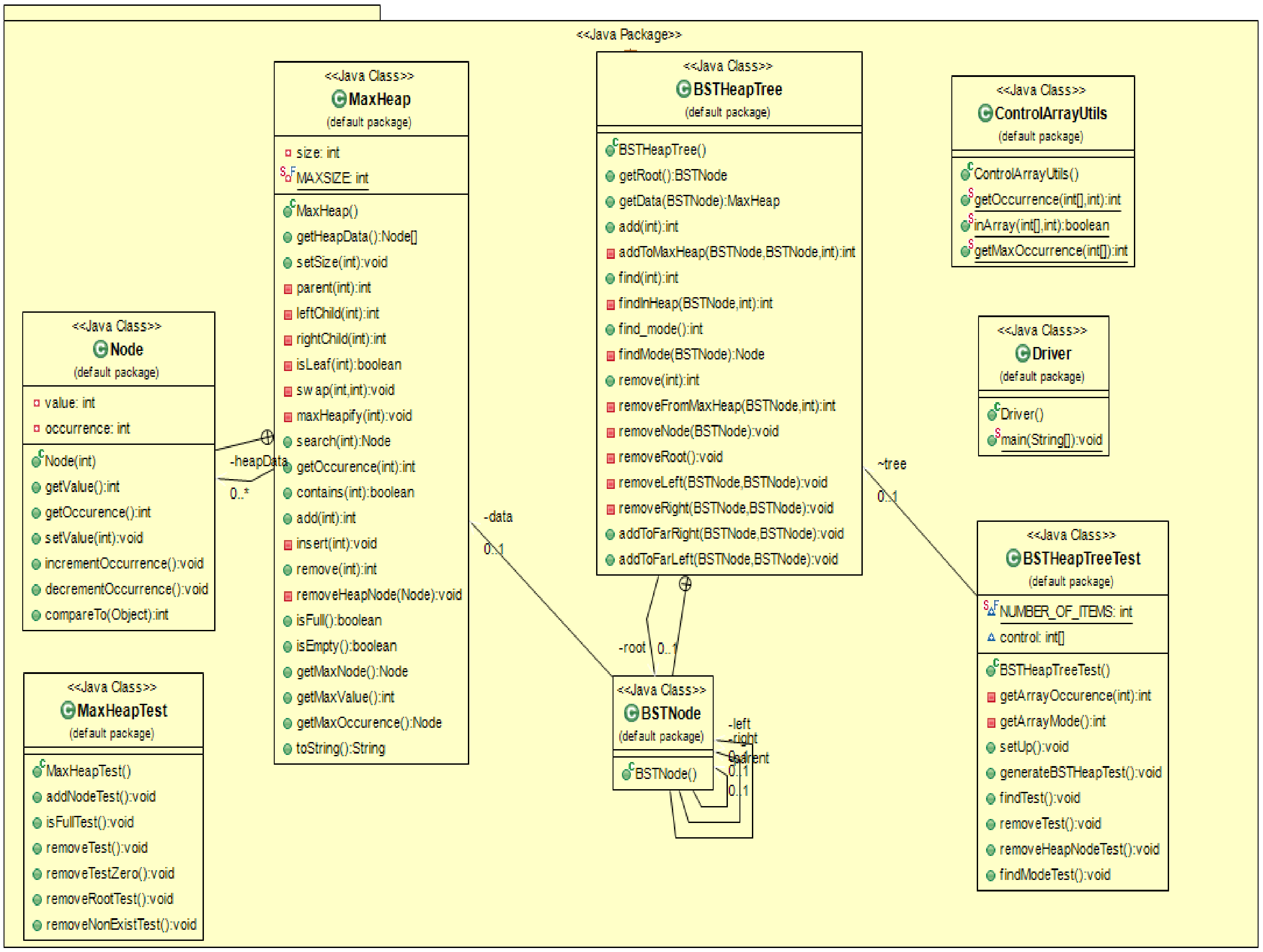
When implementing the class, you must adhere to the following features:
Each node in the heap holds two data: a value (an integer in the example above) and the number of occurrences of the value (how many that number is added to the tree). In the example, the notation 37.1 means that the number 37 is there is only one occurrence of the value 37 is available in the container.
Movement on BST is based on values at the root nodes of the Heap. In the example, the root nodes of Heaps are shown in red. If you want to add the number 50 to the tree, the movement will be done as follows:

Since the Heap with root 37 is full, since 50 is not an element of this Heap and 50 > 37, it moves to the right node in BST.
Since the Heap with root 124 is full, since 50 is not the element of this Heap and 50 < 124, it moves to the left node in BST.
Since the Heap with root 60 does not have the number 50 and there is space, it is inserted into the heap at this node as (50.1).
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. After the Heap in the root node of the BST in the example is filled, when a number smaller than 37 is inserted, the left child is created, and the number is inserted into the new Heap in that node. When we examine the example, there are 7 numbers smaller than 37 are inserted into the left child (one 31, one 15, one 29 and four 13's).
Remove operation removes only one occurrence of the value. If the number of occurrences becomes zero than the value should be removed.
During a remove operation, if the heap at a node becomes empty, the corresponding BST node should be removed as well.
The mode is the value in the BSTHeapTree that occurs most frequently. In the example the mode is 13 (node marked in green).
Your class should include the following methods:

int add (E item) – returns the number of occurrences of the item after insertion
int remove (E item) – returns the number of occurrences of the item after removal
int find (E) – returns the number of occurrences of the item in the BSTHeapTree
find_mode ()
Test your implementation as follows:

Insert the 3000 numbers that are randomly generated in the range 0-5000 into the BSTHeapTree. Store these numbers in an array as well. Sort the numbers to find the number occurrences of all the numbers.
Search for 100 numbers in the array and 10 numbers not in the array and make sure that the number of occurrences is correct.
Find the mode of the BSTHeapTree. Check whether the mode value is correct.
Remove 100 numbers in the array and 10 numbers not in the array and make sure that the number of occurrences after removal is correct.

1.Class Diagrams



2.Problem Solution Approach

My solution steps are ;

-Specify the problem requirements
-Analyze the problem
-Design an algorithm and program
-Implement the algorithm
-Test and verify the program
-Maintain and update the program.

a) **Specify the problem requirements :** I understand the problem.
b) **Analyze the problem :** I identify ; input data, output data additional requirements and constraints.

c) **Design an algorithm and program**: I divide the problem into sub-problems.I listed major steps(sub-problems). I break down each step into a more detailed list.To do these We have to divide this big project into small pieces.

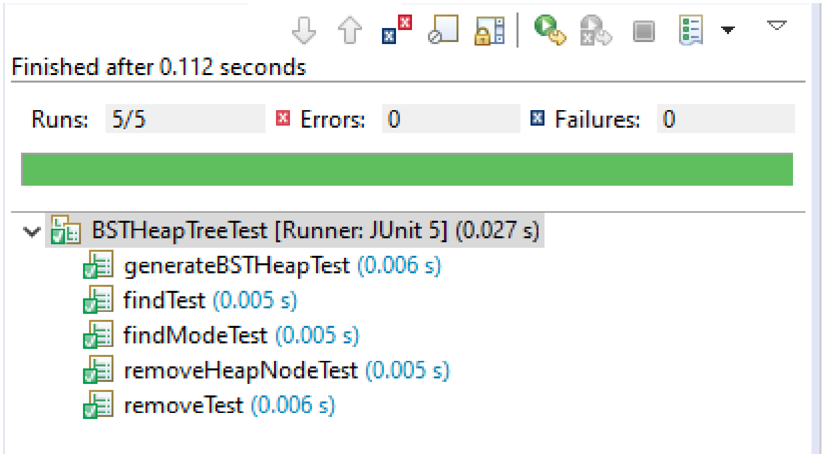
d) **Implement the algorithm:**

Binary Search Tree is a node-based binary tree data structure which has the following properties:
The left subtree of a node contains only nodes with keys lesser than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
The left and right subtree each must also be a binary search tree.
There must be no duplicate nodes.
We implemented the BST in the second part of the assignment. Nodes of BST are of type max-heap.
Each node in the heap holds two data: a value and the number of occurrences of the value (how many that number is added to the tree).
Methods:
add(E item): item is added to BST according to the BST and max-heap rules. We search the BST nodes until found the item in heap or a node which is not full. If heap contains item occerrence of item is incremented. If we found a node according to BST rules we add the item to heap as a new node. If we couldn't find a proper node we create a new as a left or right child of BST depending on item value.
remove(E item) : We find the BST node that contains the item. Occurrence of item is decremented. If occurrence of item becomes 0 we delete the item from max-heap. If heap becomes empty we also delete the BST node.
find(E item): We traverse the BST to find the item in max-heap.
find_mode(): We find the most frequent item recursively for each nodes of BST.

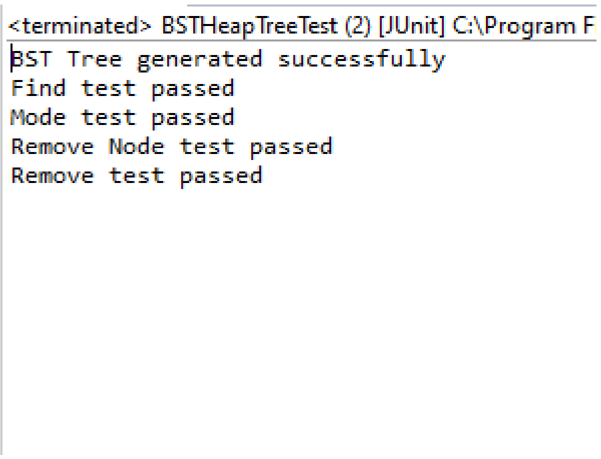
3.Test Case:

4.Running and Result

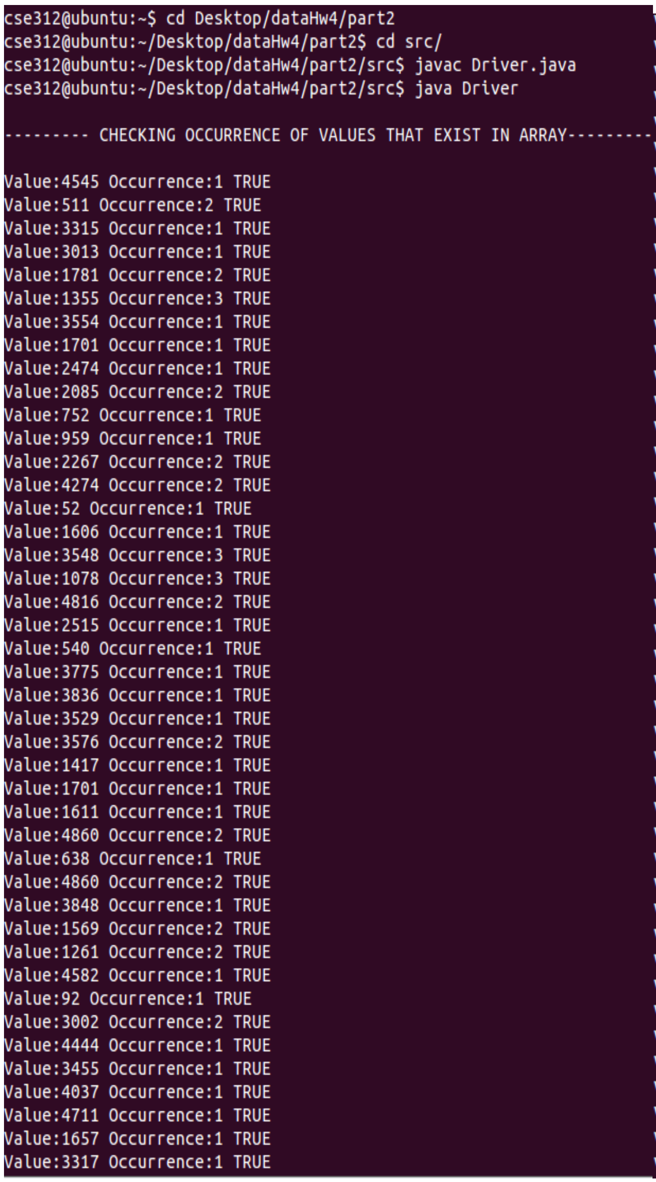
Junit test result:



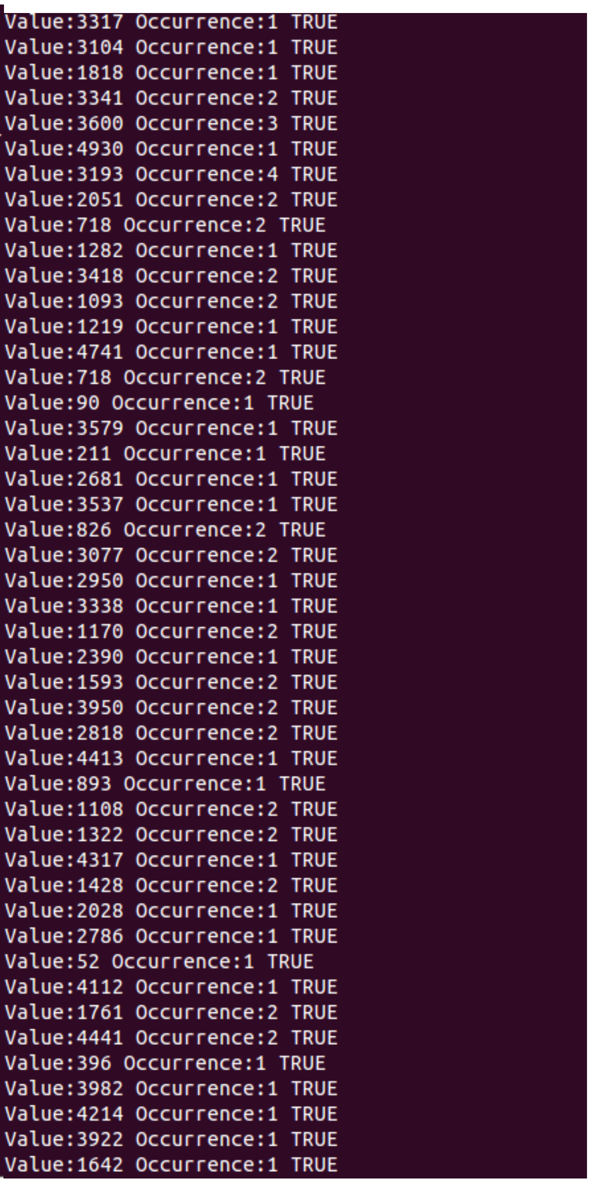
```
Finished after 0.112 seconds

Runs: 5/5    Errors: 0    Failures: 0

BSTHeapTreeTest [Runner: JUnit 5] (0.027 s)
    generateBSTHeapTest (0.006 s)
    findTest (0.005 s)
    findModeTest (0.005 s)
    removeHeapNodeTest (0.005 s)
    removeTest (0.006 s)
```



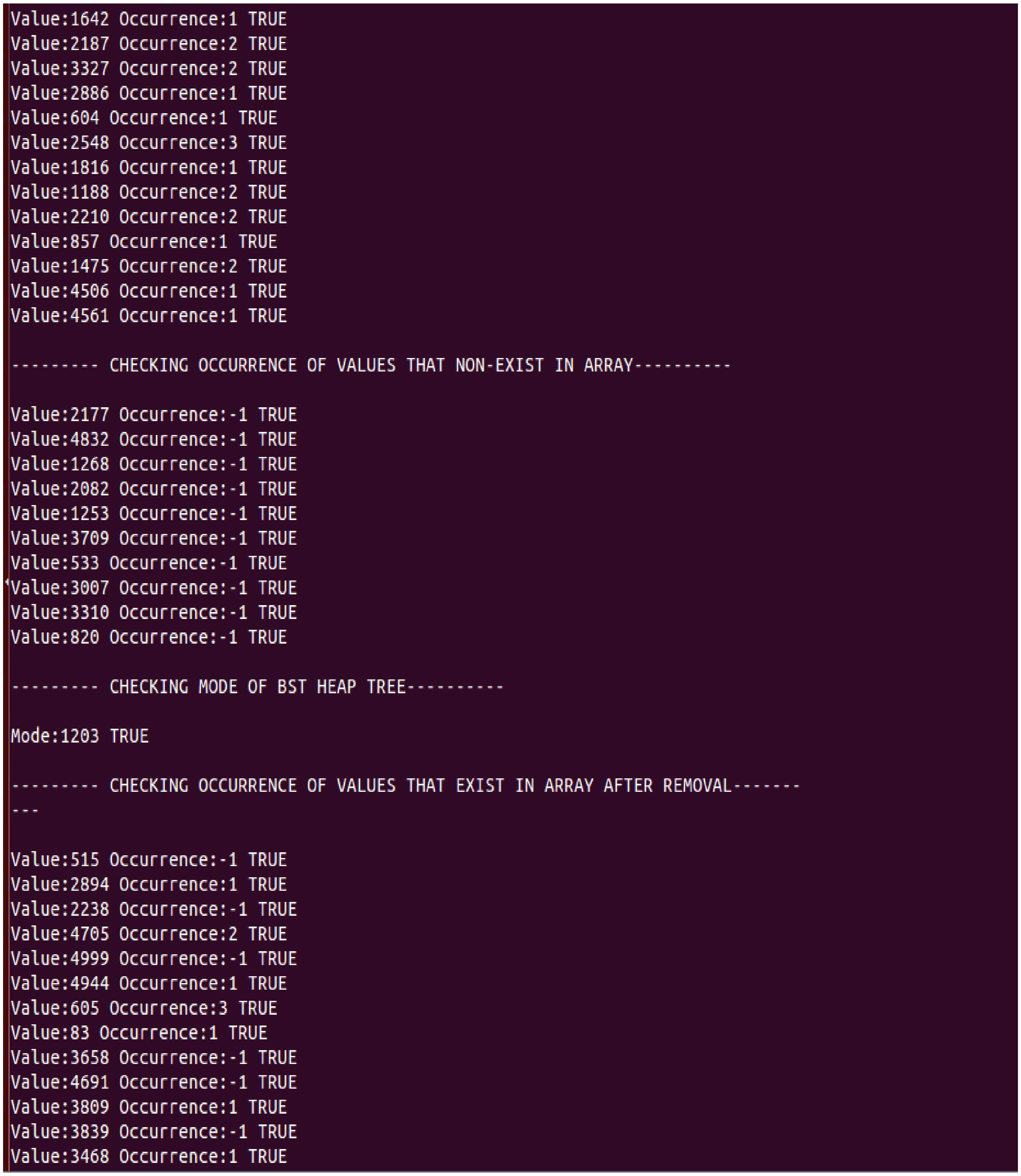
```
<terminated> BSTHeapTreeTest (2) [JUnit] C:\Program F
BST Tree generated successfully
Find test passed
Mode test passed
Remove Node test passed
Remove test passed
```



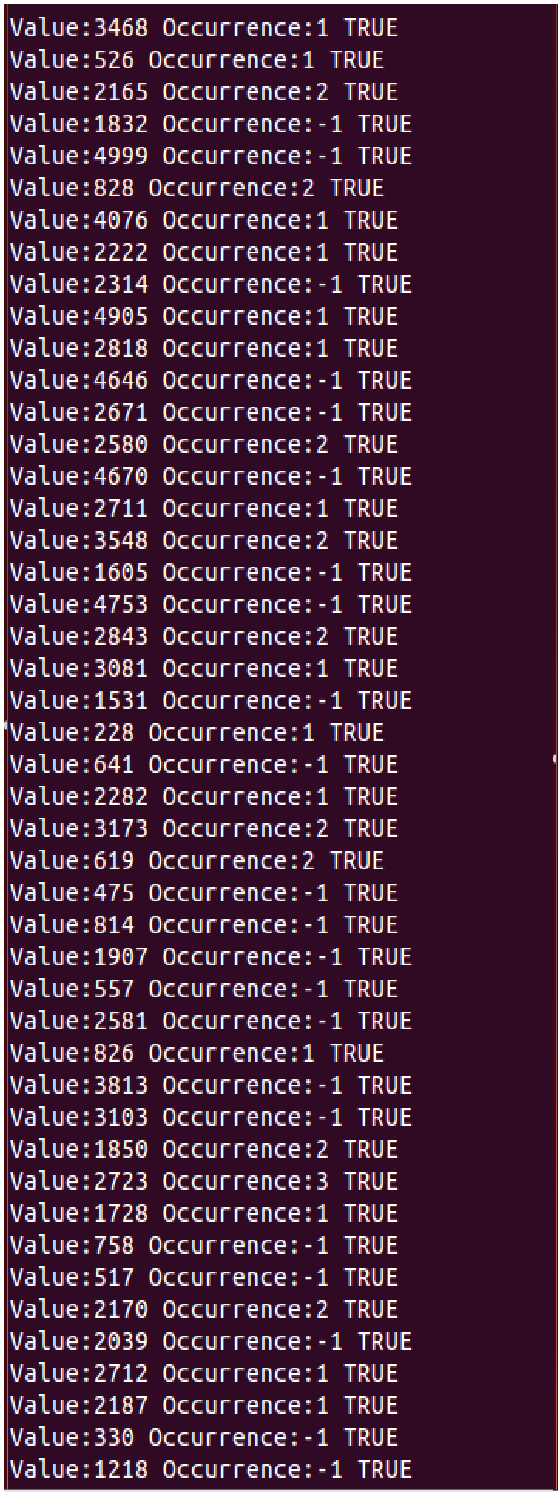
```
cse312@ubuntu:~$ cd Desktop/dataHw4/part2
cse312@ubuntu:~/Desktop/dataHw4/part2$ cd src/
cse312@ubuntu:~/Desktop/dataHw4/part2/src$ javac Driver.java
cse312@ubuntu:~/Desktop/dataHw4/part2/src$ java Driver

--------- CHECKING OCCURRENCE OF VALUES THAT EXIST IN ARRAY---------

Value:4545 Occurrence:1 TRUE
Value:511 Occurrence:2 TRUE
Value:3315 Occurrence:1 TRUE
Value:3013 Occurrence:1 TRUE
Value:1781 Occurrence:2 TRUE
Value:1355 Occurrence:3 TRUE
Value:3554 Occurrence:1 TRUE
Value:1701 Occurrence:1 TRUE
Value:2474 Occurrence:1 TRUE
Value:2085 Occurrence:2 TRUE
Value:752 Occurrence:1 TRUE
Value:959 Occurrence:1 TRUE
Value:2267 Occurrence:2 TRUE
Value:4274 Occurrence:2 TRUE
Value:52 Occurrence:1 TRUE
Value:1606 Occurrence:1 TRUE
Value:3548 Occurrence:3 TRUE
Value:1078 Occurrence:3 TRUE
Value:4816 Occurrence:2 TRUE
Value:2515 Occurrence:1 TRUE
Value:540 Occurrence:1 TRUE
Value:3775 Occurrence:1 TRUE
Value:3836 Occurrence:1 TRUE
Value:3529 Occurrence:1 TRUE
Value:3576 Occurrence:2 TRUE
Value:1417 Occurrence:1 TRUE
Value:1701 Occurrence:1 TRUE
Value:1611 Occurrence:1 TRUE
Value:4860 Occurrence:2 TRUE
Value:638 Occurrence:1 TRUE
Value:4860 Occurrence:2 TRUE
Value:3848 Occurrence:1 TRUE
Value:1569 Occurrence:2 TRUE
Value:1261 Occurrence:2 TRUE
Value:4582 Occurrence:1 TRUE
Value:92 Occurrence:1 TRUE
Value:3002 Occurrence:2 TRUE
Value:4444 Occurrence:1 TRUE
Value:3455 Occurrence:1 TRUE
Value:4037 Occurrence:1 TRUE
Value:4711 Occurrence:1 TRUE
Value:1657 Occurrence:1 TRUE
Value:3317 Occurrence:1 TRUE
```



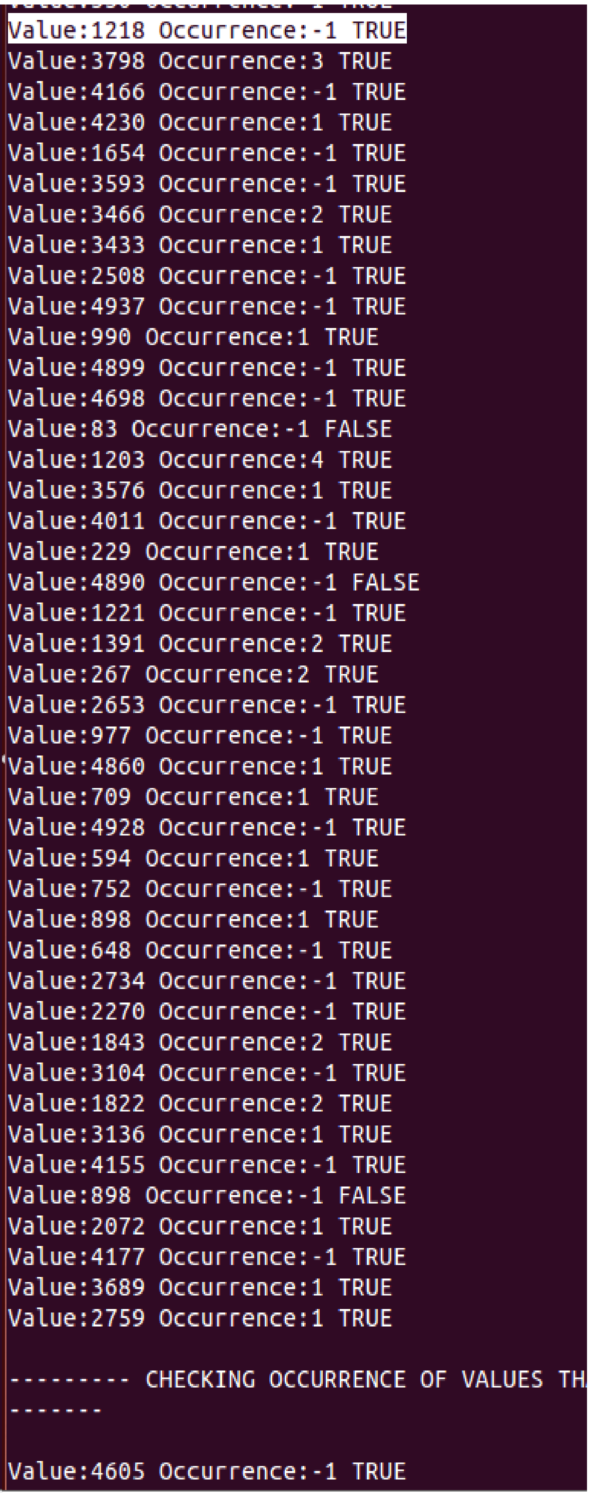
```
Value:3317 Occurrence:1 TRUE
Value:3104 Occurrence:1 TRUE
Value:1818 Occurrence:1 TRUE
Value:3341 Occurrence:2 TRUE
Value:3600 Occurrence:3 TRUE
Value:4930 Occurrence:1 TRUE
Value:3193 Occurrence:4 TRUE
Value:2051 Occurrence:2 TRUE
Value:718 Occurrence:2 TRUE
Value:1282 Occurrence:1 TRUE
Value:3418 Occurrence:2 TRUE
Value:1093 Occurrence:2 TRUE
Value:1219 Occurrence:1 TRUE
Value:4741 Occurrence:1 TRUE
Value:718 Occurrence:2 TRUE
Value:90 Occurrence:1 TRUE
Value:3579 Occurrence:1 TRUE
Value:211 Occurrence:1 TRUE
Value:2681 Occurrence:1 TRUE
Value:3537 Occurrence:1 TRUE
Value:826 Occurrence:2 TRUE
Value:3077 Occurrence:2 TRUE
Value:2950 Occurrence:1 TRUE
Value:3338 Occurrence:1 TRUE
Value:1170 Occurrence:2 TRUE
Value:2390 Occurrence:1 TRUE
Value:1593 Occurrence:2 TRUE
Value:3950 Occurrence:2 TRUE
Value:2818 Occurrence:2 TRUE
Value:4413 Occurrence:1 TRUE
Value:893 Occurrence:1 TRUE
Value:1108 Occurrence:2 TRUE
Value:1322 Occurrence:2 TRUE
Value:4317 Occurrence:1 TRUE
Value:1428 Occurrence:2 TRUE
Value:2028 Occurrence:1 TRUE
Value:2786 Occurrence:1 TRUE
Value:52 Occurrence:1 TRUE
Value:4112 Occurrence:1 TRUE
Value:1761 Occurrence:2 TRUE
Value:4441 Occurrence:2 TRUE
Value:396 Occurrence:1 TRUE
Value:3982 Occurrence:1 TRUE
Value:4214 Occurrence:1 TRUE
Value:3922 Occurrence:1 TRUE
Value:1642 Occurrence:1 TRUE
```



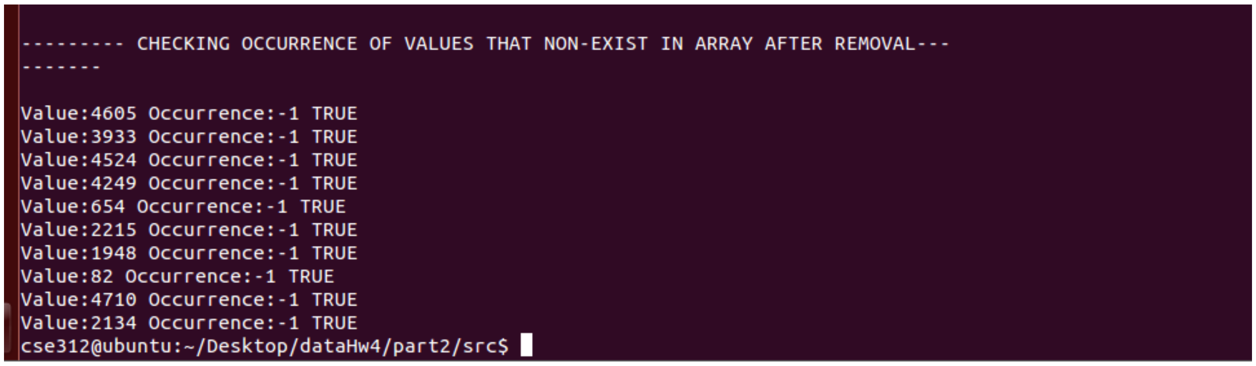
```
Value:1642 Occurrence:1 TRUE
Value:2187 Occurrence:2 TRUE
Value:3327 Occurrence:2 TRUE
Value:2886 Occurrence:1 TRUE
Value:604 Occurrence:1 TRUE
Value:2548 Occurrence:3 TRUE
Value:1816 Occurrence:1 TRUE
Value:1188 Occurrence:2 TRUE
Value:2210 Occurrence:2 TRUE
Value:857 Occurrence:1 TRUE
Value:1475 Occurrence:2 TRUE
Value:4506 Occurrence:1 TRUE
Value:4561 Occurrence:1 TRUE

--------- CHECKING OCCURRENCE OF VALUES THAT NON-EXIST IN ARRAY----------

Value:2177 Occurrence:-1 TRUE
Value:4832 Occurrence:-1 TRUE
Value:1268 Occurrence:-1 TRUE
Value:2082 Occurrence:-1 TRUE
Value:1253 Occurrence:-1 TRUE
Value:3709 Occurrence:-1 TRUE
Value:533 Occurrence:-1 TRUE
Value:3007 Occurrence:-1 TRUE
Value:3310 Occurrence:-1 TRUE
Value:820 Occurrence:-1 TRUE

--------- CHECKING MODE OF BST HEAP TREE----------

Mode:1203 TRUE

--------- CHECKING OCCURRENCE OF VALUES THAT EXIST IN ARRAY AFTER REMOVAL-------
---

Value:515 Occurrence:-1 TRUE
Value:2894 Occurrence:1 TRUE
Value:2238 Occurrence:-1 TRUE
Value:4705 Occurrence:2 TRUE
Value:4999 Occurrence:-1 TRUE
Value:4944 Occurrence:1 TRUE
Value:605 Occurrence:3 TRUE
Value:83 Occurrence:1 TRUE
Value:3658 Occurrence:-1 TRUE
Value:4691 Occurrence:-1 TRUE
Value:3809 Occurrence:1 TRUE
Value:3839 Occurrence:-1 TRUE
Value:3468 Occurrence:1 TRUE
```



```
Value:3468 Occurrence:1 TRUE
Value:526 Occurrence:1 TRUE
Value:2165 Occurrence:2 TRUE
Value:1832 Occurrence:-1 TRUE
Value:4999 Occurrence:-1 TRUE
Value:828 Occurrence:2 TRUE
Value:4076 Occurrence:1 TRUE
Value:2222 Occurrence:1 TRUE
Value:2314 Occurrence:-1 TRUE
Value:4905 Occurrence:1 TRUE
Value:2818 Occurrence:1 TRUE
Value:4646 Occurrence:-1 TRUE
Value:2671 Occurrence:-1 TRUE
Value:2580 Occurrence:2 TRUE
Value:4670 Occurrence:-1 TRUE
Value:2711 Occurrence:1 TRUE
Value:3548 Occurrence:2 TRUE
Value:1605 Occurrence:-1 TRUE
Value:4753 Occurrence:-1 TRUE
Value:2843 Occurrence:2 TRUE
Value:3081 Occurrence:1 TRUE
Value:1531 Occurrence:-1 TRUE
Value:228 Occurrence:1 TRUE
Value:641 Occurrence:-1 TRUE
Value:2282 Occurrence:1 TRUE
Value:3173 Occurrence:2 TRUE
Value:619 Occurrence:2 TRUE
Value:475 Occurrence:-1 TRUE
Value:814 Occurrence:-1 TRUE
Value:1907 Occurrence:-1 TRUE
Value:557 Occurrence:-1 TRUE
Value:2581 Occurrence:-1 TRUE
Value:826 Occurrence:1 TRUE
Value:3813 Occurrence:-1 TRUE
Value:3103 Occurrence:-1 TRUE
Value:1850 Occurrence:2 TRUE
Value:2723 Occurrence:3 TRUE
Value:1728 Occurrence:1 TRUE
Value:758 Occurrence:-1 TRUE
Value:517 Occurrence:-1 TRUE
Value:2170 Occurrence:2 TRUE
Value:2039 Occurrence:-1 TRUE
Value:2712 Occurrence:1 TRUE
Value:2187 Occurrence:1 TRUE
Value:330 Occurrence:-1 TRUE
Value:1218 Occurrence:-1 TRUE
```



```
Value:1218 Occurrence:-1 TRUE
Value:3798 Occurrence:3 TRUE
Value:4166 Occurrence:-1 TRUE
Value:4230 Occurrence:1 TRUE
Value:1654 Occurrence:-1 TRUE
Value:3593 Occurrence:-1 TRUE
Value:3466 Occurrence:2 TRUE
Value:3433 Occurrence:1 TRUE
Value:2508 Occurrence:-1 TRUE
Value:4937 Occurrence:-1 TRUE
Value:990 Occurrence:1 TRUE
Value:4899 Occurrence:-1 TRUE
Value:4698 Occurrence:-1 TRUE
Value:83 Occurrence:-1 FALSE
Value:1203 Occurrence:4 TRUE
Value:3576 Occurrence:1 TRUE
Value:4011 Occurrence:-1 TRUE
Value:229 Occurrence:1 TRUE
Value:4890 Occurrence:-1 FALSE
Value:1221 Occurrence:-1 TRUE
Value:1391 Occurrence:2 TRUE
Value:267 Occurrence:2 TRUE
Value:2653 Occurrence:-1 TRUE
Value:977 Occurrence:-1 TRUE
Value:4860 Occurrence:1 TRUE
Value:709 Occurrence:1 TRUE
Value:4928 Occurrence:-1 TRUE
Value:594 Occurrence:1 TRUE
Value:752 Occurrence:-1 TRUE
Value:898 Occurrence:1 TRUE
Value:648 Occurrence:-1 TRUE
Value:2734 Occurrence:-1 TRUE
Value:2270 Occurrence:-1 TRUE
Value:1843 Occurrence:2 TRUE
Value:3104 Occurrence:-1 TRUE
Value:1822 Occurrence:2 TRUE
Value:3136 Occurrence:1 TRUE
Value:4155 Occurrence:-1 TRUE
Value:898 Occurrence:-1 FALSE
Value:2072 Occurrence:1 TRUE
Value:4177 Occurrence:-1 TRUE
Value:3689 Occurrence:1 TRUE
Value:2759 Occurrence:1 TRUE

--------- CHECKING OCCURRENCE OF VALUES TH
-------

Value:4605 Occurrence:-1 TRUE
```



```
--------- CHECKING OCCURRENCE OF VALUES THAT NON-EXIST IN ARRAY AFTER REMOVAL---
-------

Value:4605 Occurrence:-1 TRUE
Value:3933 Occurrence:-1 TRUE
Value:4524 Occurrence:-1 TRUE
Value:4249 Occurrence:-1 TRUE
Value:654 Occurrence:-1 TRUE
Value:2215 Occurrence:-1 TRUE
Value:1948 Occurrence:-1 TRUE
Value:82 Occurrence:-1 TRUE
Value:4710 Occurrence:-1 TRUE
Value:2134 Occurrence:-1 TRUE
cse312@ubuntu:~/Desktop/dataHw4/part2/src$
```

Part-3 Report

Heap.java:

```
public T[] getData() {
        return data;                Analyze the time complexity=O(1)
}

public int getSize() {
        return size;                Analyze the time complexity=O(1)
}


private int parent(int pos) {
        return pos / 2;             Analyze the time complexity=O(1)
}


private int leftChild(int pos) {
        return (2 * pos);           Analyze the time complexity=O(1)
}


private int rightChild(int pos) {
        return (2 * pos) + 1;       Analyze the time complexity=O(1)
}

private boolean isLeaf(int pos) {
        if (pos > (size / 2) && pos <= size) {
                return true;
        }
        return false;               Analyze the time complexity=O(1)
}


private void swap(int fpos, int spos) {
        T tmp;
        tmp = data[fpos];
        data[fpos] = data[spos];
        data[spos] = tmp;                   Analyze the time complexity=O(1)
}
```

```
private void maxHeapify(int pos) {
        if (isLeaf(pos))
                return;

if ((data[leftChild(pos)] != null && data[pos].compareTo(data[leftChild(pos)]) < 0)
        || (data[rightChild(pos)] != null && data[pos].compareTo(data[rightChild(pos)]) < 0)) {
                if (data[leftChild(pos)].compareTo(data[rightChild(pos)]) > 0) {
                        swap(pos, leftChild(pos));
                        maxHeapify(leftChild(pos));
                } else {
                        swap(pos, rightChild(pos));
                        maxHeapify(rightChild(pos));
                }
        }
}
```

Recursive function calls up O(logn) yourself.

Analyze the time complexity=O(logn)

```
public void add(T element) {
        if (!isFull()) {
                data[++size] = element;

                // Traverse up and fix violated property
                int current = size;
        while (parent(current) != 0 && data[current].compareTo(data[parent(current)]) > 0) {
                        swap(current, parent(current));
                        current = parent(current);
                }
        }
}
```

The time complexity is logn because it goes all the way to the length of the tree. isFull method is time complexity O(1), swap method is time complexity O(1)

Analyze the time complexity=O(logn)

```
public void remove(T node) {
        for (int i = 1; i < size; i++) {
                if (node.equals(data[i])) {
                        T lastNode = data[size];
                        data[i] = lastNode;
                        data[size] = null;
                        size--;
                        if (i == 1) {
                                maxHeapify(1);
                        }
                        return;
                }
        }
        data[size] = null;
        if (this.size == 0) {
                data = null;
        }
}
```

Time complexity is O(nlogn) because maxHeapify recursive in how much the for loop returns as O(logn).
Analyze the time complexity=O(nlogn)

```
public T search(T item) {
        for (int i = 1; i <= size; i++) {
                if (data[i].compareTo(item) == 0) {
                        return data[i];
                }
        }
        return null;
}
```

Time complexity is O(n) .Because for the for loop returns size.
Analyze the time complexity=O(n)

```
public void mergeHeap(Heap<T> mergeHeap) {
        if (size + mergeHeap.size <= maxSize) {
                for (int i = 1; i <= mergeHeap.size; i++) {
                        data[++size] = (T) mergeHeap.data[i];
                }
                maxHeapify(1);
        }
}
```

Time complexity is O(n) .Because for the for loop returns size.
Analyze the time complexity=O(n)

public void removeKthLargest(int order) {

```
        Heap<T> newHeap = new Heap<T>(size - order + 1);
        for (int i = 1; i <= size; i++) { -> O(n)
            if (newHeap.search(data[i]) == null) {
                if (newHeap.isFull()) {
                    newHeap.remove(newHeap.data[1]); ->O(nlogn)
                }
                newHeap.add(data[i]); ->O(logn)
            }
        }
        T kLargest = newHeap.data[1];
        remove(kLargest); ->O(nlogn)

    }
```

The for loop returns O (n) times, if found, the remove function remove is O (nlogn). Since the append method searches through the tree, it is O (logn). By using the time complexities of these functions, the time complexity of the removeKthLargest function is O (n ^ 2).

Analyze the time complexity=O(n^2)

```
    public void setValues(T val) {

        MaxHeapIterator iterator = (Heap<T>.MaxHeapIterator) this.iterator();
        while (iterator.hasNext()) {
            iterator.setValue(val);
        }
    }
```

The Iterator while loop travels n times in the loop.

Analyze the time complexity=O(n)

public String toString() {

```
        String str = "";
        for (int i = 1; i <= this.size; i++) {
            str += "(" + data[i] + ")";
        }
        return "Heap{" + str + '}';
    }
```

Since the for loop returns to size

Analyze the time complexity=O(n)

public boolean isFull() {

```
        return size == maxSize;
    }
```

Analyze the time complexity=O(1)

```
public Iterator<T> iterator() {
        return (Iterator<T>) new MaxHeapIterator();      Analyze the time complexity=O(1)
}


public MaxHeapIterator() {
        super();
        this.index = -1;                     Analyze the time complexity=O(1)
}


public boolean hasNext() {
        return (index + 1) < size;           Analyze the time complexity=O(1)
}

public T next() {
        index++;
        return data[index];                  Analyze the time complexity=O(1)
}
public void setValue(T value) {
        index++;
        data[index] = value;                 Analyze the time complexity=O(1)
}
```

BSTHeapTree.java:

```
public int add(int item) {
        return addToMaxHeap(root, null, item);
}
```

Analyze the time complexity=O(n)

```
private int addToMaxHeap(BSTNode node, BSTNode parent, int item) {
        if (node == null) {
                node = new BSTNode();
                MaxHeap heap = new MaxHeap();
                heap.add(item);
                node.data = heap;
                node.parent = parent;
                if (parent != null) {
                    if (parent.data.getMaxNode().compareTo(node.data.getMaxNode()) > 0) {
                                parent.left = node;
                        } else {
                                parent.right = node;
                        }
                }
                return 1;
        }

        if (!node.data.isFull() || node.data.contains(item)) {
                return node.data.add(item);
        } else {
                if (node.data.getMaxValue() > item) {
                        return addToMaxHeap(node.left, node, item);
                } else {
                        return addToMaxHeap(node.right, node, item);
                }
        }

}
```

Analyze the time complexity=O(n)

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

Analyze the time complexity=O(logn)

```
private int findInHeap(BSTNode node, int item) {
        if (node == null) {
                return -1;
        }

        int occurence = node.data.getOccurence(item);
        if (occurence > -1) {
                return occurence;
        } else {
                if (node.data.getMaxNode().getValue() > item) {
                        return findInHeap(node.left, item);
                } else {
                        return findInHeap(node.right, item);
                }
        }
}
```

Analyze the time complexity=O(logn)

```
public int find_mode() {
        MaxHeap.Node heapNode = findMode(root);
        return heapNode.getValue();
}
```

Analyze the time complexity=O(logn)

```
private MaxHeap.Node findMode(BSTNode node) {

    if (node.left == null & node.right == null) {
        return node.data.getMaxOccurence();
    }
    if (node.left == null) {
        return findMode(node.right);
    } else {
        if (node.right == null) {
            return findMode(node.left);
        } else {
            MaxHeap.Node leftMax = findMode(node.left);
            MaxHeap.Node rightMax = findMode(node.right);
            if (leftMax.getOccurence() > rightMax.getOccurence()) {
                if (leftMax.getOccurence() >
node.data.getMaxOccurence().getOccurence()) {
                    return leftMax;
                } else {
                    return node.data.getMaxOccurence();
                }
            } else {
                if (rightMax.getOccurence() >
node.data.getMaxOccurence().getOccurence()) {
                    return rightMax;
                } else {
                    return node.data.getMaxOccurence();
```

Analyze the time complexity=O(logn)

```
public int remove(int item) {
    return removeFromMaxHeap(root, item);
}
```

Analyze the time complexity=O(logn)

```
private int removeFromMaxHeap(BSTNode node, int item) {

        if (node == null) {
            return -1;
        }
        if (node.data.contains(item)) {
            int occurence = node.data.remove(item);
            if (node.data.isEmpty()) {
                removeNode(node);
            }
            return occurence;
        } else {
            if (node.data.getMaxValue() > item) {
                return removeFromMaxHeap(node.left, item);
            } else {
                return removeFromMaxHeap(node.right, item);
            }
        }
    }
```

Analyze the time complexity=O(logn)

```
    private void removeNode(BSTNode node) {

        if (this.root.equals(node)) {
            removeRoot();
            return;
        }

        // left child of parent
        if (node.equals(node.parent.left)) {
            removeLeft(node.parent, node);
            return;
        }

        // right child of parent if
        if (node.equals(node.parent.right)) {
            removeRight(node.parent, node);

            return;
        }
```

Analyze the time complexity=O(logn)

private void removeRoot() {

```
        if (root.left != null && root.right != null) {
                addToFarRight(root.left, root.right);
                root.left.parent = null;
                root = root.left;
                return;
        }
        if (root.right != null) {
                root.right.parent = null;
                root = root.right;
                return;
        }
        if (root.left != null) {
                root.left.parent = null;
                root = root.left;
                return;
        }
        this.root = null;
```

Analyze the time complexity=O(logn)

private void removeLeft(BSTNode parent, BSTNode node) {

```
        if (node.left != null && node.right != null) {
                addToFarRight(node.left, node.right);
                node.left.parent = parent;
                parent.left = node.left;
                return;
        }
        if (node.right != null) {
                parent.left = node.right;
                node.right.parent = parent;
                return;
        }

        if (node.left != null) {
                parent.left = node.left;
                node.left.parent = parent;
                return;
        }

        parent.left = null;
```

Analyze the time complexity=O(logn)

```
private void removeRight(BSTNode parent, BSTNode node) {
        if (node.left != null && node.right != null) {
                addToFarLeft(node.right, node.left);
                node.right.parent = parent;
                parent.right = node.right;
                return;
        }
        if (node.right != null) {
                parent.right = node.right;
                node.right.parent = parent;
                return;
        }

        if (node.left != null) {
                parent.right = node.left;
                node.left.parent = parent;
                return;
        }

        parent.right = null;
```

Analyze the time complexity=O(logn)

```
public void addToFarRight(BSTNode left, BSTNode right) {
        while (left.right != null) {
                left = left.right;
        }
        left.right = right;
        right.parent = left;

}
```

Analyze the time complexity=O(logn)

```
public void addToFarLeft(BSTNode right, BSTNode left) {
        while (right.left != null) {
                right = right.left;
        }
        right.left = left;
        left.parent = right;
```

Analyze the time complexity=O(logn)