# GIT Department of Computer Engineering
# CSE 222/505 - Spring 2021
# Homework4 # Report

**Can Duyar**
**171044075**

Explanation of the PDF Requirements:

* I used completely OOP Design and applied inheritance, I checked all of the possible errors and adding error handling parts for my homework

*My design works properly for Part-1 and Part-2. I also explained the Part-3(it is also done) on the following pages.

## 1-DETAILED SYSTEM REQUIREMENTS:

### For PART-1: (Heap Implementation with Array)

***Functional Requirements:***

**Heap<E extends Comparable<E>> implements Iterable<E> class:** This class implement several methods with using arrays for heap operations that are explained in the pdf.

**Methods:**

**boolean searchElement(E item):** Search for an element in the heap. It uses isEmpty() method to check that the heap is empty or not.

**boolean mergeHeap(Heap<E> ) :** Merge with another heap

**boolean isEmpty():** It returns true if the heap is empty, otherwise it returns false.

**boolean isFull():** it returns true if heap is full otherwise returns false

**void nthLargestRemove(int ):** Removing i th largest element from the Heap. This method uses helper method called as “delete”.

**E delete(int):** it is used for deleting operation.

**void heapUp(int):** This method used to protect properly after when we add an element.It uses a helper method called as "findMax".

**void heapDown(int):** This method used to protect properly after when we remove an element.

**void showHeap():** This method prints all of the elements of the heap

**void add(E):** It is used for adding the element to heap

**int findMax(int ):** This method used to find the child that has maximum value.

**int indexParent(int):** to find the index of parent

**int indexChild(int,int):** to find the index of a child

**public class HeapIterator implements Iterator<E> class:**

*I extended the Iterator class by adding a method to set the value (value passed as parameter) of the last element returned by the next methods.

*I added a method called as "setLast" to do this operation on heap class. I also used next(), hasNext() to have more control on heap class' arrays. My new iterator class' name is HeapIterator but actually it works on arrays for heap class.

**Running the program(main function):**

Creating two objects of Heap and I implemented all of the tests on them, I used two objects to show that my merge method works.

For PART-2: (BSTHeapTree Implementation with Array)

***Functional Requirements:***

**Node class:** Node class contain two int. The value and occurrences

**BSTHeapTree<E extends Comparable> class:** This class implement several methods to perform all require methods and variables. It contains a array of Node class with length 7. And also contains some final int. Those are used to identify the node within the array.

**Methods:**
**Int add(E):** It take an element and add to BSTHeapTree. It returns the occurrence of that value after added. For adding the at right position it usages a private method called addingData.

**Int addingData(E, int):** This method takes two arguments. The value and a node index to add value to that index. It checks the node and if it capable to took this number then it put in this node or it finds a new node to add that value. If all nodes are full then it creates a new BSTHeapTree object and link to main object. I use two helper method two find the node where it cat put the value. After adding value to a position it returns the occurrences of that number.

**Int getSmall(int), int getBig(int):** Those two methods take a int argument that is actually a index of nodes inside BSTHeapTree object and then returns the next node.

**Int find(E):** This method take an element and search through the entire tree and returns the occurrence of that value.

**Int find_mode():** This method doesn't take any argument and it returns the value that occurrences maximum number.

**Int remove(E):** This method takes a value and find this number in tree object if it founds the number then if occurrences is more than one then it decreases the occurrences and then returns the existed occurrences of that number. If occurrence isn't more than 1 then it simply delete that node and returns 0. And also if the number isn't in that object then also returns 0. This method is uses a private method named as isEmpty.

**Boolean isEmpty():** This method doesn't takes any arguments and returns if it is empty or not. This method is called from the parent tree object and if the link object is empty then set it to null.

**Node[] getArrayOfNodes():** This method doesn't takes any arguments and return a array of existed all nodes in a single array. To count the array size this method uses a method called getNumberOfNodes.

**Int getNumberOfNodes():** This method also doesn't takes any arguments and return the total number of nodes in this object.

**Int[] getRandomNumbers(int,int,int):** This method takes three arguments how may numbers to be generated, minimum range, maximum range. Then it returns an array of random numbers.

**Void sort(int[]):** This method takes an array of int and sort that array

**Running the program(main function):**

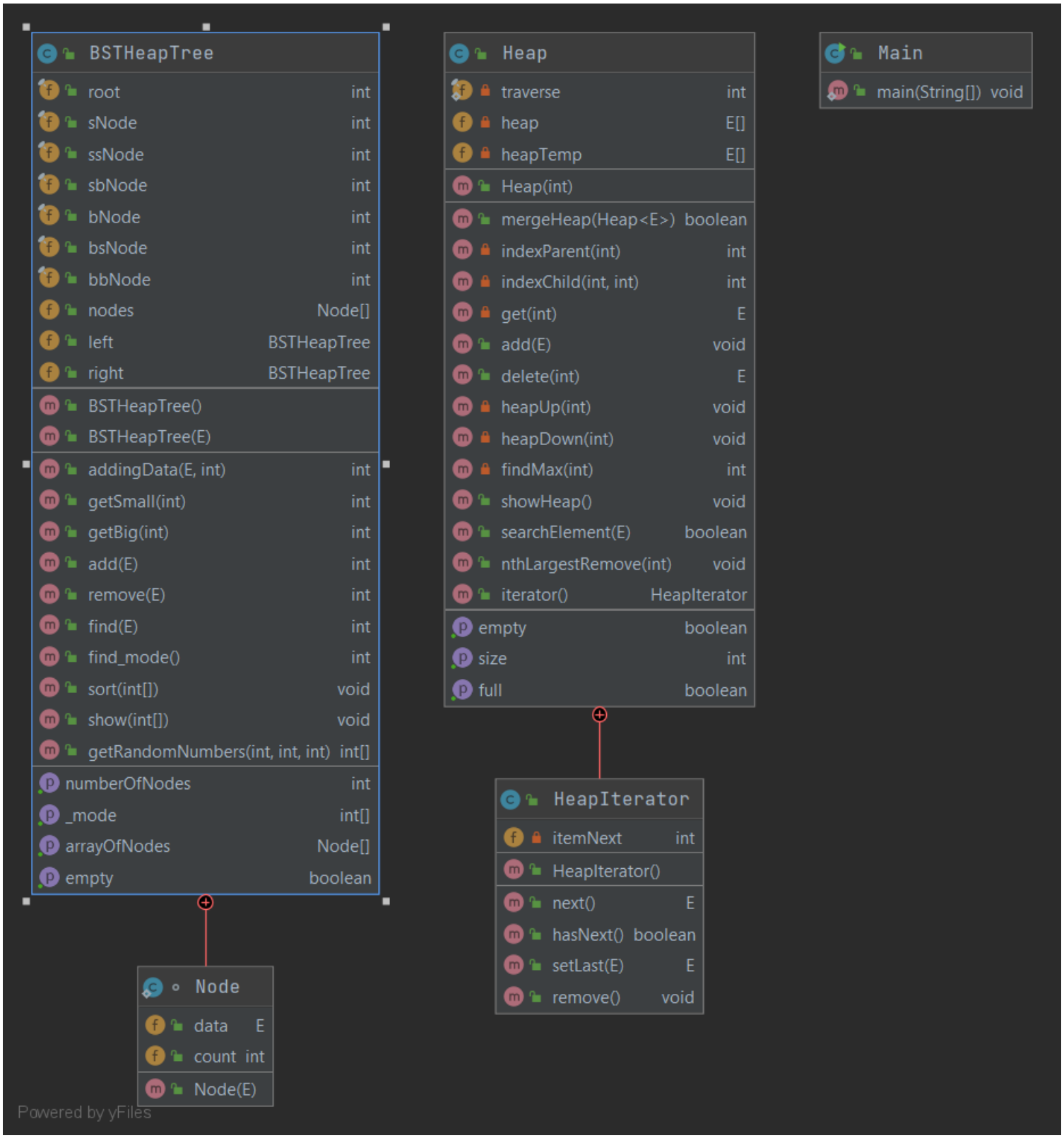
Creating the object of BSTHeapTree. And get an array of 3000 random numbers within 0 to 5000. After that using sort function to sort the array. Finding 100 numbers in the array and 10 numbers not in the array and

printing the return results.Removing 100 numbers in the array and 10 numbers not in the array and then printing the return results.

### *Non-Functional Requirements for part-1 and part-2:*

*I used Ubuntu 20.04-Focal Fossa and compile the Main.java with javac 11.0.11 for my Heap and BSTHeapTree implementation.

# 2-CLASS DIAGRAMS:



## 3-PROBLEM SOLUTIONS APPROACH:

My solution approach is completely based on OOP. I used two different classes to implement Heap(part-1) and BSTHeapTree(part-2). Both of these classes that extends Comparable class to use compareTo. I used “compareTo” to solve problem with converting generic. I used my tests on each of the types and showed the integer tests on my report because example in the pdf was related with integers. Implementing Heap Class(Part-1) uses maxHeap implementation based on array. I used two heap type arrays. I kept other array because I needed it on my “nthLargestRemove” method to copy other array’s information. I also kept size value on the heap class to handle size-based operations. I already explained all of my methods in the “detailed system requirements” part.I extended the Iterator class by adding a method to set the value (value passed as parameter) of the last element returned by the next methods to achieve pdf instruction which is given in the last part of the Part-1.

I added a method called as “setLast” to do this operation on heap class. I also used next(), hasNext() to have more control on heap class’ arrays. My new iterator class’ name is HeapIterator but actually it works on arrays for heap class.

To implement BSTHeapTree(Part-2), I created an inner node class to keep information of the nodes. I kept data and count variables to keep nodes’ data and occurrence variables. I also kept second node class in the node class that’s why I created Node classes connected to each other. It was my main problem solution method when I was dealing with implementation of BSTHeapTree class. I also kept Node array for each of the nodes(max limit was 7 according to PDF instructions) and object from BSTHeapTree type to keep left and right. I traverse the entire tree with using them.

# 4-TEST CASES / RUNNING COMMAND AND RESULTS:

```
(base) can@can-ThinkPad-L13:~/Desktop/veri_yapilari_son_odev$ java Main

HEAP(MaxHeap) TEST

Heap:
192 58 13 34 14 11 1 8

TEST - Search for an element (99)
false

TEST - Search for an element (13)
true

TEST - Merge with another heap

Heap-1:
192 58 13 34 14 11 1 8

Heap-2:
111 73 81 3 19 17 33

Merged Heap:
192 111 33 58 81 13 19 8 34 14 73 3 11 1 17

After removing the 0. index largest element
111 81 33 58 73 13 19 8 34 14 17 3 11 1

After removing the 3. index largest element
111 81 33 34 73 13 19 8 1 14 17 3 11
TEST-Iterator's next() method (  it.next()  )
111
TEST-Iterator's next() method (  it.next()  )
81

After 'setLast(313)'  (extended iterator's method)
11
313 81 111 34 73 33 19 8 1 14 17 3 13
```

99 is not element of the Heap so program couldn't find it and returns false

13 is the element of the heap and program found it so I returns true in this case.

192 was the biggest element in the heap so it was removed when we want to remove 0. index largest element

54 was the 3. index largest element in the heap so it was removed succesfully!!!

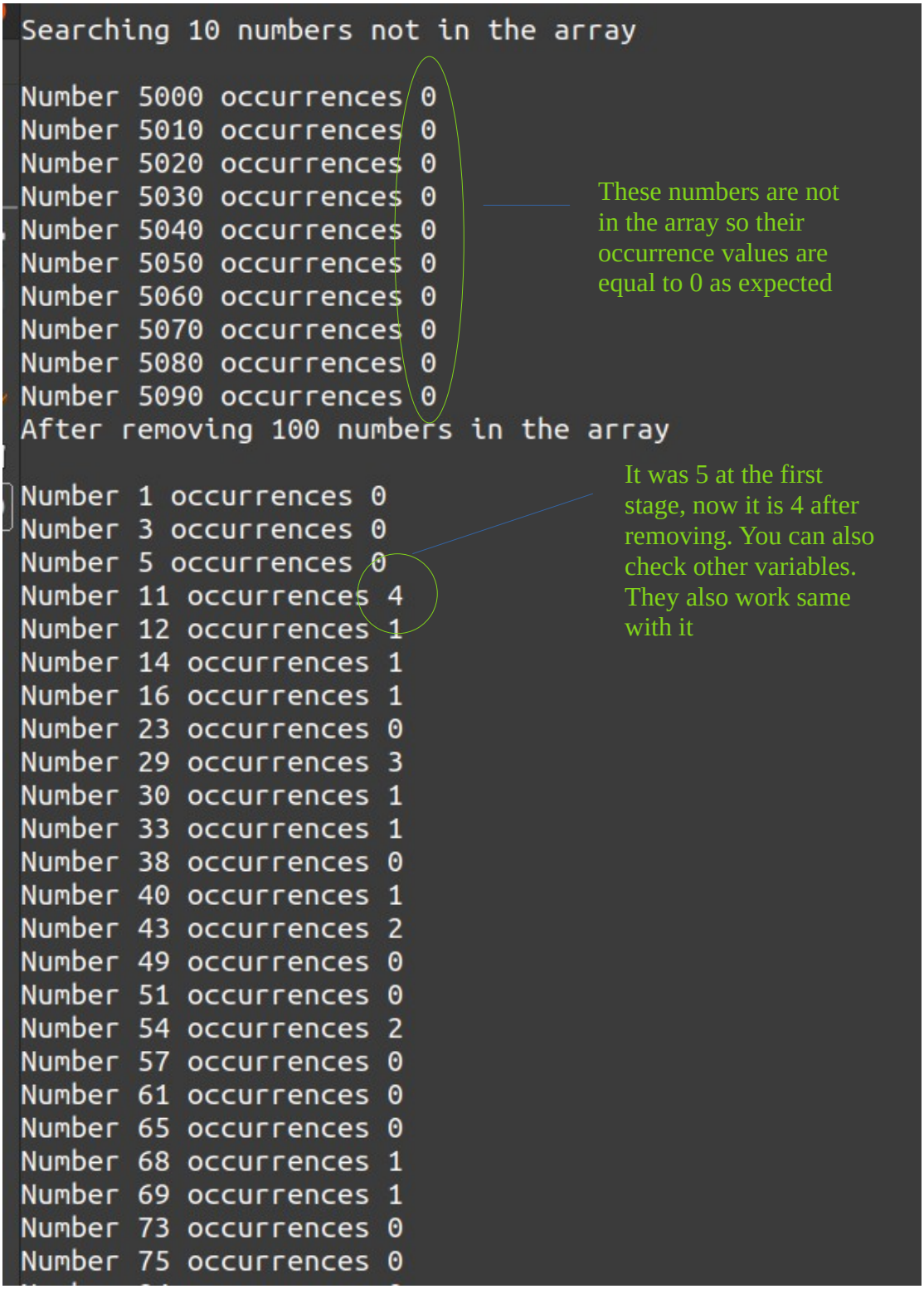
Last element was 11 and when we apply setLast(313) then 11 was returned and removed from the heap, after that 313 was added the heap and positions are arranged again!

```
After 'setLast(41)'  (extended iterator's method)
13
313 81 111 34 73 41 19 8 1 14 17 3 33

**************************************************

BSTHeapTree TEST


find mode:11

Searching 100 numbers in the array

Number 1 occurrences 1
Number 3 occurrences 1
Number 5 occurrences 1
Number 11 occurrences 5
Number 12 occurrences 2
Number 14 occurrences 2
Number 16 occurrences 2
Number 23 occurrences 1
Number 29 occurrences 4
Number 30 occurrences 2
Number 33 occurrences 2
Number 38 occurrences 1
Number 40 occurrences 2
Number 43 occurrences 3
Number 49 occurrences 1
Number 51 occurrences 1
Number 54 occurrences 3
Number 57 occurrences 1
Number 61 occurrences 1
Number 65 occurrences 1
Number 68 occurrences 2
Number 69 occurrences 2
Number 73 occurrences 1
Number 75 occurrences 1
Number 84 occurrences 1
```

11 is the mode of the BSTHeapTree because it is most frequently value as you see

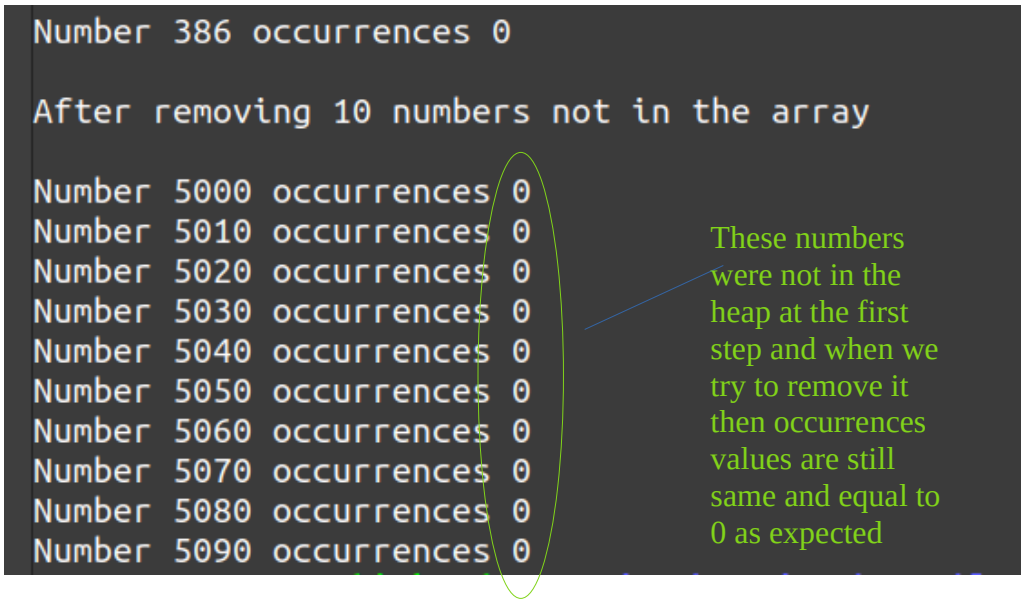
```
Number 91 occurrences 2
Number 93 occurrences 1
Number 98 occurrences 2
Number 106 occurrences 1
Number 110 occurrences 2
Number 117 occurrences 1
Number 124 occurrences 2
Number 125 occurrences 2
Number 130 occurrences 0
Number 135 occurrences 0
Number 137 occurrences 0
Number 138 occurrences 0
Number 144 occurrences 0
Number 147 occurrences 0
Number 149 occurrences 0
Number 153 occurrences 0
Number 157 occurrences 0
Number 162 occurrences 0
Number 164 occurrences 0
Number 165 occurrences 0
Number 166 occurrences 0
Number 167 occurrences 0
Number 169 occurrences 0
Number 171 occurrences 0
Number 173 occurrences 0
Number 178 occurrences 0
Number 184 occurrences 0
Number 189 occurrences 0
Number 200 occurrences 0
Number 201 occurrences 0
Number 204 occurrences 0
Number 207 occurrences 0
Number 212 occurrences 0
Number 216 occurrences 0
Number 220 occurrences 0
Number 223 occurrences 0
Number 230 occurrences 0
Number 235 occurrences 0
```

```
Number 238 occurrences 0
Number 239 occurrences 0
Number 244 occurrences 0
Number 246 occurrences 0
Number 247 occurrences 0
Number 252 occurrences 0
Number 260 occurrences 0
Number 263 occurrences 0
Number 265 occurrences 0
Number 270 occurrences 0
Number 275 occurrences 0
Number 279 occurrences 0
Number 284 occurrences 0
Number 287 occurrences 0
Number 290 occurrences 0
Number 294 occurrences 0
Number 299 occurrences 0
Number 308 occurrences 0
Number 312 occurrences 0
Number 313 occurrences 0
Number 316 occurrences 0
Number 319 occurrences 0
Number 325 occurrences 0
Number 330 occurrences 0
Number 331 occurrences 0
Number 337 occurrences 0
Number 342 occurrences 0
Number 349 occurrences 0
Number 353 occurrences 0
Number 358 occurrences 0
Number 359 occurrences 0
Number 369 occurrences 0
Number 372 occurrences 0
Number 374 occurrences 0
Number 380 occurrences 0
Number 383 occurrences 0
Number 386 occurrences 0

Searching 10 numbers not in the array
```


Searching 10 numbers not in the array
Number 5000 occurrences 0
Number 5010 occurrences 0
Number 5020 occurrences 0
Number 5030 occurrences 0
Number 5040 occurrences 0
Number 5050 occurrences 0
Number 5060 occurrences 0
Number 5070 occurrences 0
Number 5080 occurrences 0
Number 5090 occurrences 0
After removing 100 numbers in the array
Number 1 occurrences 0
Number 3 occurrences 0
Number 5 occurrences 0
Number 11 occurrences 4
Number 12 occurrences 1
Number 14 occurrences 1
Number 16 occurrences 1
Number 23 occurrences 0
Number 29 occurrences 3
Number 30 occurrences 1
Number 33 occurrences 1
Number 38 occurrences 0
Number 40 occurrences 1
Number 43 occurrences 2
Number 49 occurrences 0
Number 51 occurrences 0
Number 54 occurrences 2
Number 57 occurrences 0
Number 61 occurrences 0
Number 65 occurrences 0
Number 68 occurrences 1
Number 69 occurrences 1
Number 73 occurrences 0
Number 75 occurrences 0
These numbers are not in the array so their occurrence values are equal to 0 as expected
It was 5 at the first stage, now it is 4 after removing. You can also check other variables. They also work same with it

```
Number 84 occurrences 0
Number 91 occurrences 1
Number 93 occurrences 0
Number 98 occurrences 1
Number 106 occurrences 0
Number 110 occurrences 1
Number 117 occurrences 0
Number 124 occurrences 1
Number 125 occurrences 1
Number 130 occurrences 0
Number 135 occurrences 0
Number 137 occurrences 0
Number 138 occurrences 0
Number 144 occurrences 0
Number 147 occurrences 0
Number 149 occurrences 0
Number 153 occurrences 0
Number 157 occurrences 0
Number 162 occurrences 0
Number 164 occurrences 0
Number 165 occurrences 0
Number 166 occurrences 0
Number 167 occurrences 0
Number 169 occurrences 0
Number 171 occurrences 0
Number 173 occurrences 0
Number 178 occurrences 0
Number 184 occurrences 0
Number 189 occurrences 0
Number 200 occurrences 0
Number 201 occurrences 0
Number 204 occurrences 0
Number 207 occurrences 0
Number 212 occurrences 0
Number 216 occurrences 0
Number 220 occurrences 0
Number 223 occurrences 0
Number 230 occurrences 0
```

```
Number 235 occurrences 0
Number 238 occurrences 0
Number 239 occurrences 0
Number 244 occurrences 0
Number 246 occurrences 0
Number 247 occurrences 0
Number 252 occurrences 0
Number 260 occurrences 0
Number 263 occurrences 0
Number 265 occurrences 0
Number 270 occurrences 0
Number 275 occurrences 0
Number 279 occurrences 0
Number 284 occurrences 0
Number 287 occurrences 0
Number 290 occurrences 0
Number 294 occurrences 0
Number 299 occurrences 0
Number 308 occurrences 0
Number 312 occurrences 0
Number 313 occurrences 0
Number 316 occurrences 0
Number 319 occurrences 0
Number 325 occurrences 0
Number 330 occurrences 0
Number 331 occurrences 0
Number 337 occurrences 0
Number 342 occurrences 0
Number 349 occurrences 0
Number 353 occurrences 0
Number 358 occurrences 0
Number 359 occurrences 0
Number 369 occurrences 0
Number 372 occurrences 0
Number 374 occurrences 0
Number 380 occurrences 0
Number 383 occurrences 0
```




*I applied all of the test scenarios from the PDF. And program passed from all of the tests!!!

PART-3:

*ANALYZE THE TIME COMPLEXITY:*

For Part-1(Heap implementation):

```
/**
 * Heap constructor with one parameter
 * @param total_size - to initialize the total size of the heap
 */
@SuppressWarnings("unchecked")
public Heap(int total_size){
    size = 0;
    heap = (E[])new Comparable[total_size+1];
    heapTemp = (E[])new Comparable[total_size+1];
}
```

Theta(1) constant time

*There is no loop etc. so time complexity is constant time(Theta(1)).

```
/**
 * Merge with another heap
 * @param mergedHeap - another heap to merge
 * @return returns true when merge operation is done
 */
public boolean mergeHeap(Heap<E> mergedHeap){
    for(int t=0 ; t < mergedHeap.getSize(); t++){
        add(mergedHeap.get(t));
    }

    return true;
}
```

Theta(n)

Theta(1)

Theta(1)

T(n) = theta(n)*theta(1) + theta(1)
T(n) = theta(n)

```java
/**
 *  @return true or false - if heap is full then it returns true otherwise returns false
 */
public boolean isFull(){
    return size == heap.length;
}

/**
 * @return true or false - if heap is empty then it returns true otherwise returns false
 */
public boolean isEmpty(){
    return size==0;
}


/**
* @param parent - to find the index it
* @return it returns index of the parent
*/
private int indexParent(int parent){
    return (parent-1)/traverse;
}
```

Theta(1)
constant time

Theta(1)
Constant time

Theta(1)
Constant time

*There is no loop etc. so all of them has Theta(1) constant time as time complexity.

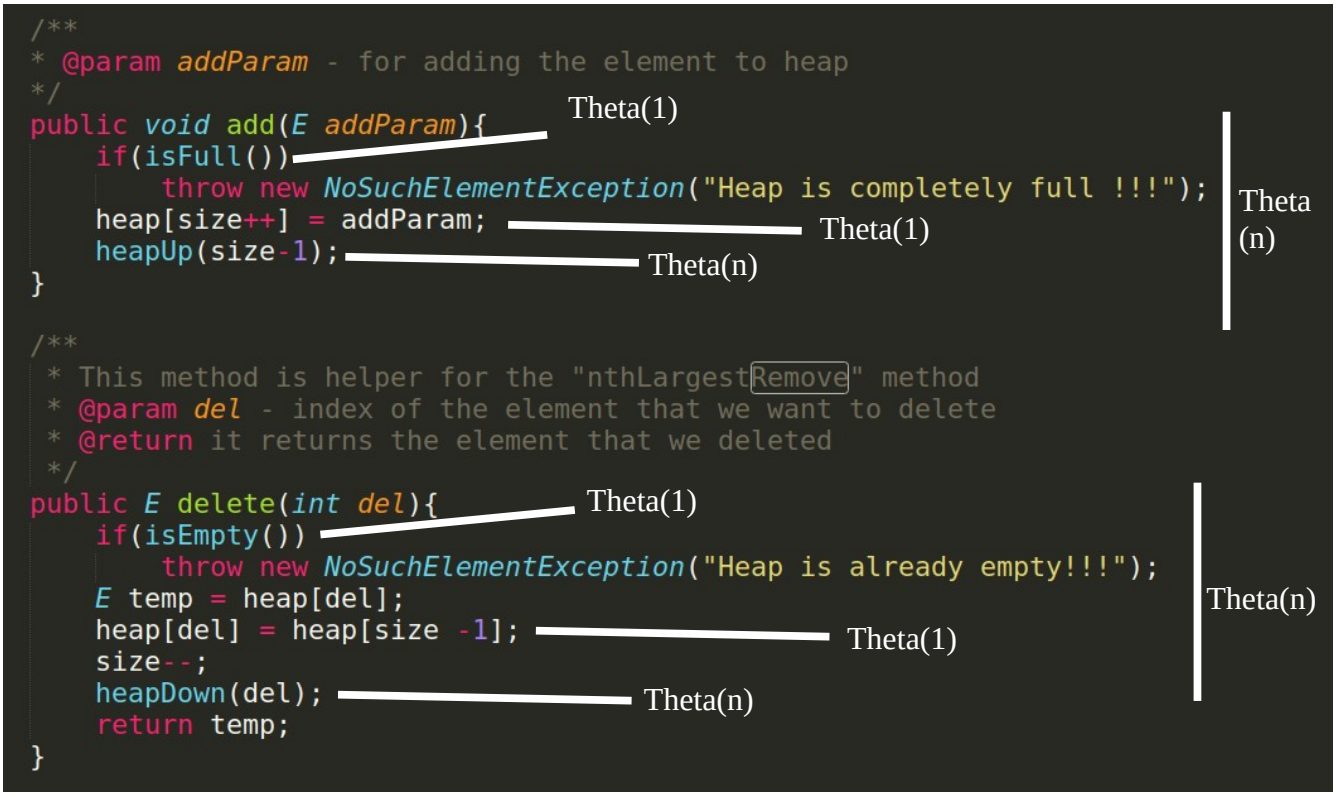
```java
* @param child - to find the index of the child
* @param param - helper parameter to find the index of the child properly
* @return it returns index of the child
*/

private int indexChild(int child,int param){
    return traverse*child + param;
}



/**
* @param ind - to get the element which is in given index
* @return specific data which is in given index in the heap
*/

private E get(int ind){
    return heap[ind];
}


/**
* @return size of the heap
*/
public int getSize(){
    return size;
}
```

Theta(1)
Constant time

Theta(1)
Constant time

Theta(1)
Constant time

*There is no loop etc. so all of them has Theta(1) constant time as time complexity.




Add method has Theta(n) time complexity.
Delete method has also Theta(n) time complexity.
Theta(n) = Theta(1)*Theta(1) + Theta(1) + Theta(n) = Theta(n)

```java
/**
 *  This method used to protect properly after when we add an element.
 * @param up - index of the added element in the heap
 */
private void heapUp(int up) {
    E keep = heap[up];
    while(up>0 && keep.compareTo(heap[indexParent(up)]) == 1){
        heap[up] = heap[indexParent(up)];
        up = indexParent(up);
    }
    heap[up] = keep;
}

/**
 *  This method used to protect properly after when we remove an element.
 * @param down - index of the removed element in the heap
 */
private void heapDown(int down){
    int ch;
    E keep = heap[down];
    while(indexChild(down, 1) < size){
        ch = findMax(down);
        if(keep.compareTo(heap[ch]) < 0){
            heap[down] = heap[ch];
        }else
            break;

        down = ch;
    }
    heap[down] = keep;
}
```

Theta(1)

Theta(1)

Theta(n) Because of while loop

Theta(1)

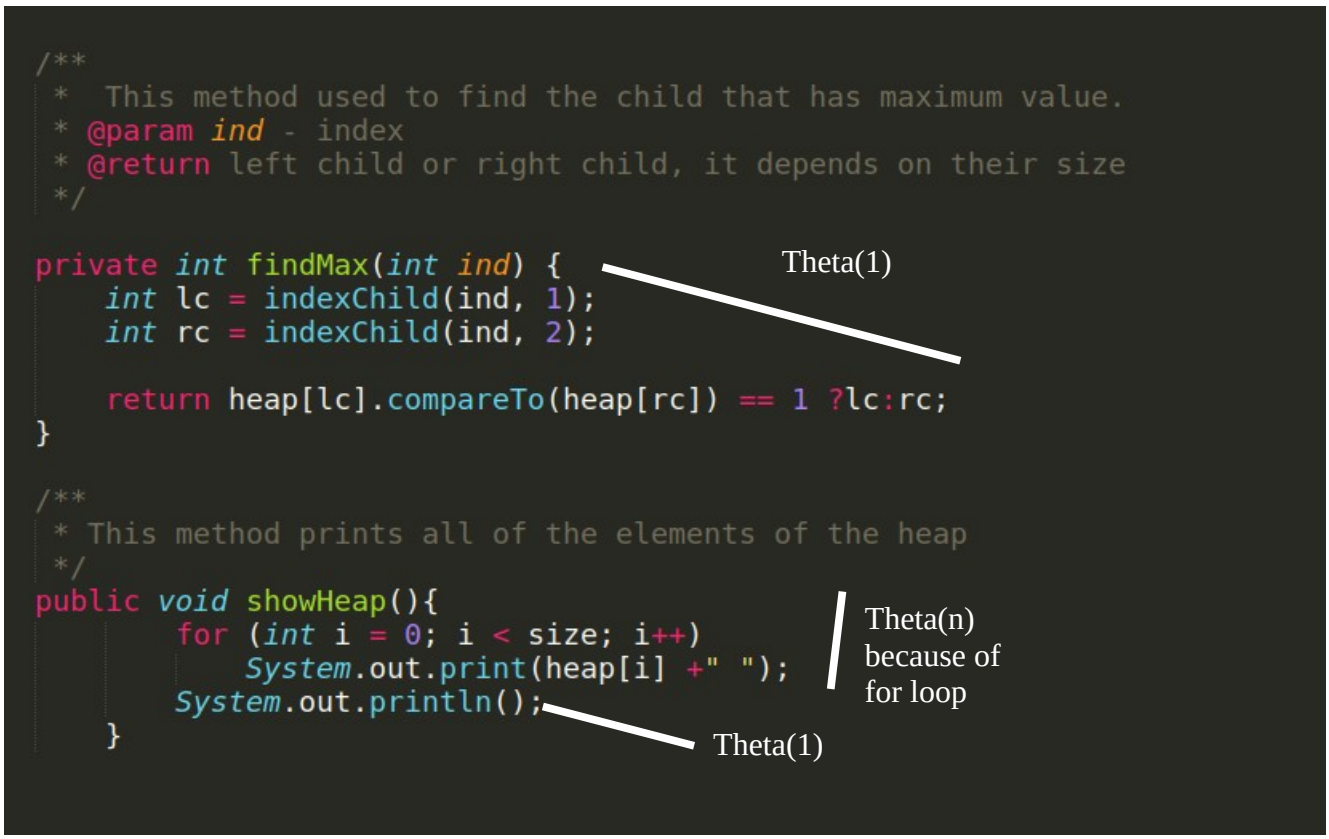
Theta(1)

Theta(1)

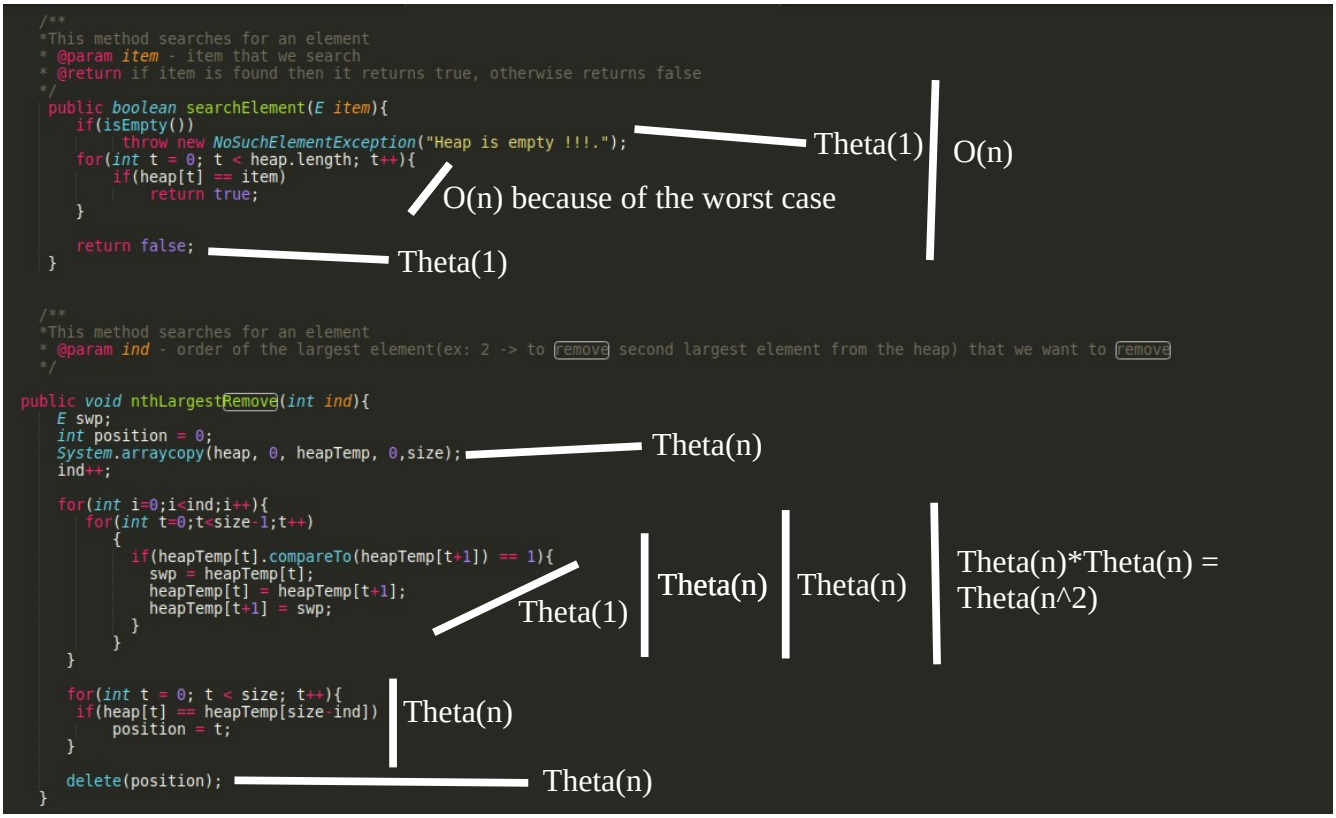
O(n) – because of worst case

Theta(1)

heapUp method has T(n) = Theta(1)*Theta(n) + Theta(1) = Theta(n)
T(n) = Theta(n)
heapDown method has T(n) = Theta(1) + Theta(1)*O(n) + Theta(1) = O(n)
T(n) = O(n)

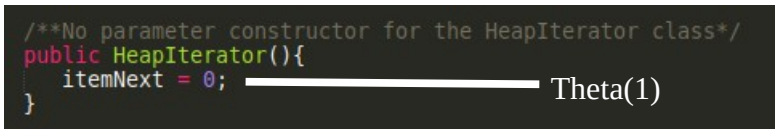


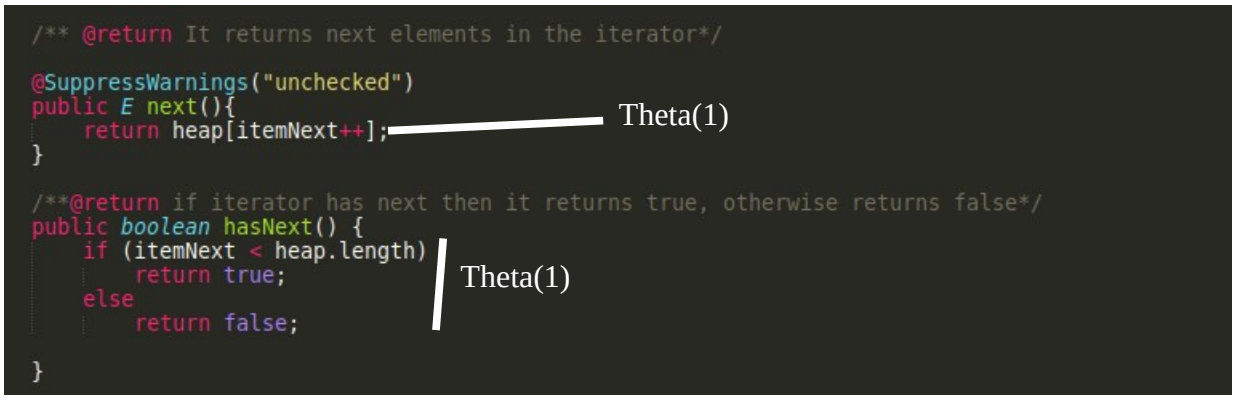

findMax() method has no loop etc. so it has Theta(1) time complexity.
ShowHeap method has a for loop and it has to return until the value of size in every situation that's why it has T(n) = Theta(n) + Theta(1) = Theta(n)
T(n) = Theta(n)




“searchElement” method has O(n) time complexity.
“NthLargestRemove” method has T(n) = Theta(n) + Theta(n)*Theta(n) + Theta(n) + Theta(n)
T(n) = Theta(n^2) as time complexity.

Heap Iterator methods:




* next() and hasNext() methods have Theta(1) as time complexity, because they don’t have any loop etc.

```java
    /**
    *method to set the value (value passed as parameter) of the last element returned by the next methods.
    *@param item - value for setting
    *@return it returns the value of last element according to pdf instructions
    */
    public E setLast(E item){
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        E last_value = heap[size-1];
        heap[size-1] = item;
        heapUp(size-1);
        return last_value;
        }


    /**remove method is not allowed in this class*/
    public void remove(){
        System.out.println("Remove is not allowed!!!");
    }

}


    /**We can reach the HeapIterator class from the outer classes with the help of this method
    * @return HeapIterator class object
    */
    @Override
    public HeapIterator iterator() {
        return new HeapIterator();
    }

}
```

Theta(1)

Theta(n) because of heapUp method

Theta(n)

Theta(1)

Theta(1)

For Part-2 (BSTHeapTree implementation)

```java
    /**
    *@param init  - to initialize data for the node
    */
    public Node(E init)
    {
        data = init;
        count = 1;
    }
```

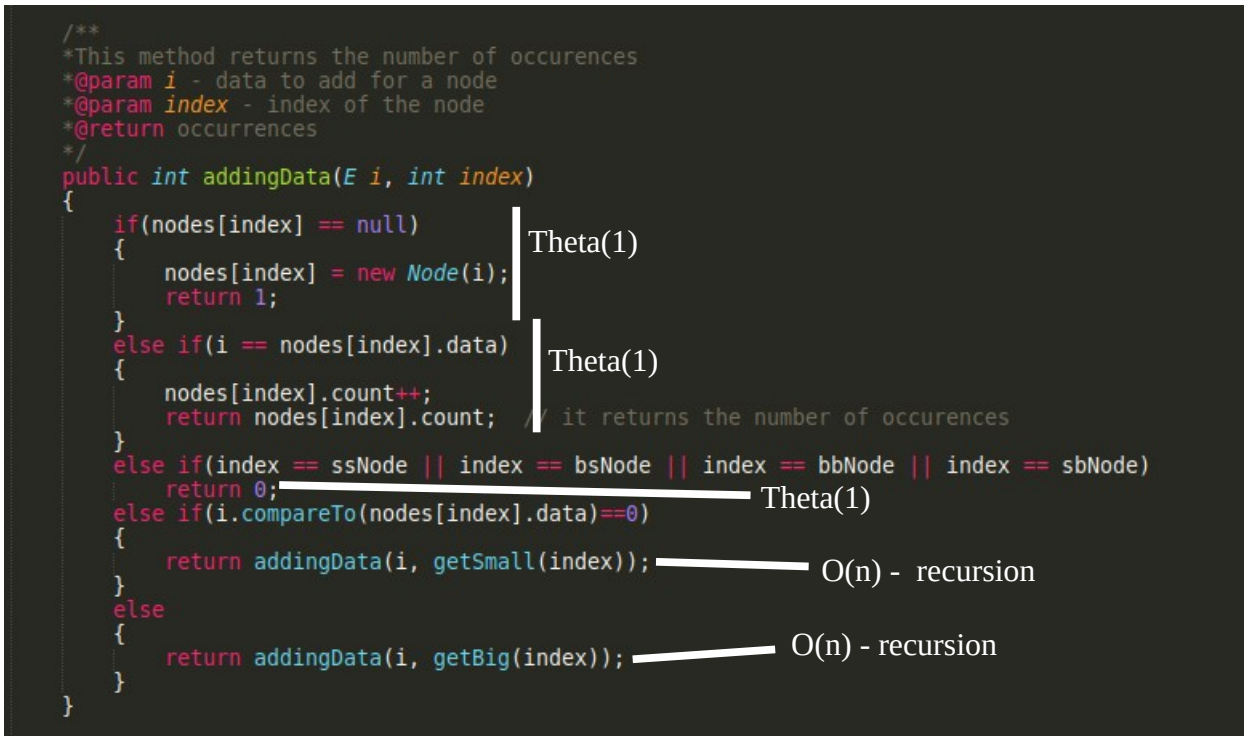
Theta(1)

```java
public BSTHeapTree(E init)
{
    nodes[root] = new Node(init);
}
```

Theta(1)

```java
/**
* This method returns the index of the small node
*@param Node i
*@return next Node index
*/
public int getSmall(int i)
{
    if(i == root)
        return sNode;
    else if(i == sNode)
        return ssNode;
    else if(i == bNode)
        return bsNode;
    else return 0;
}
```

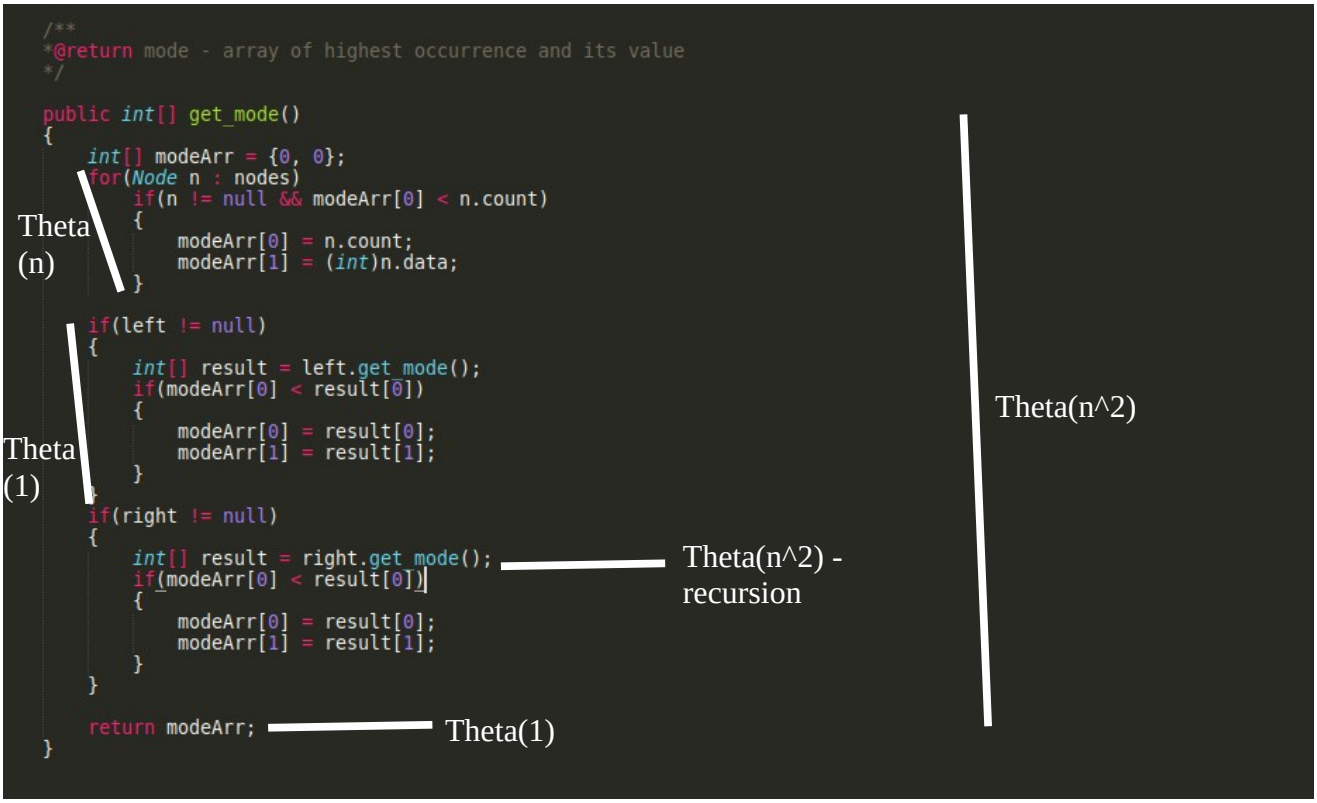
Theta(1)

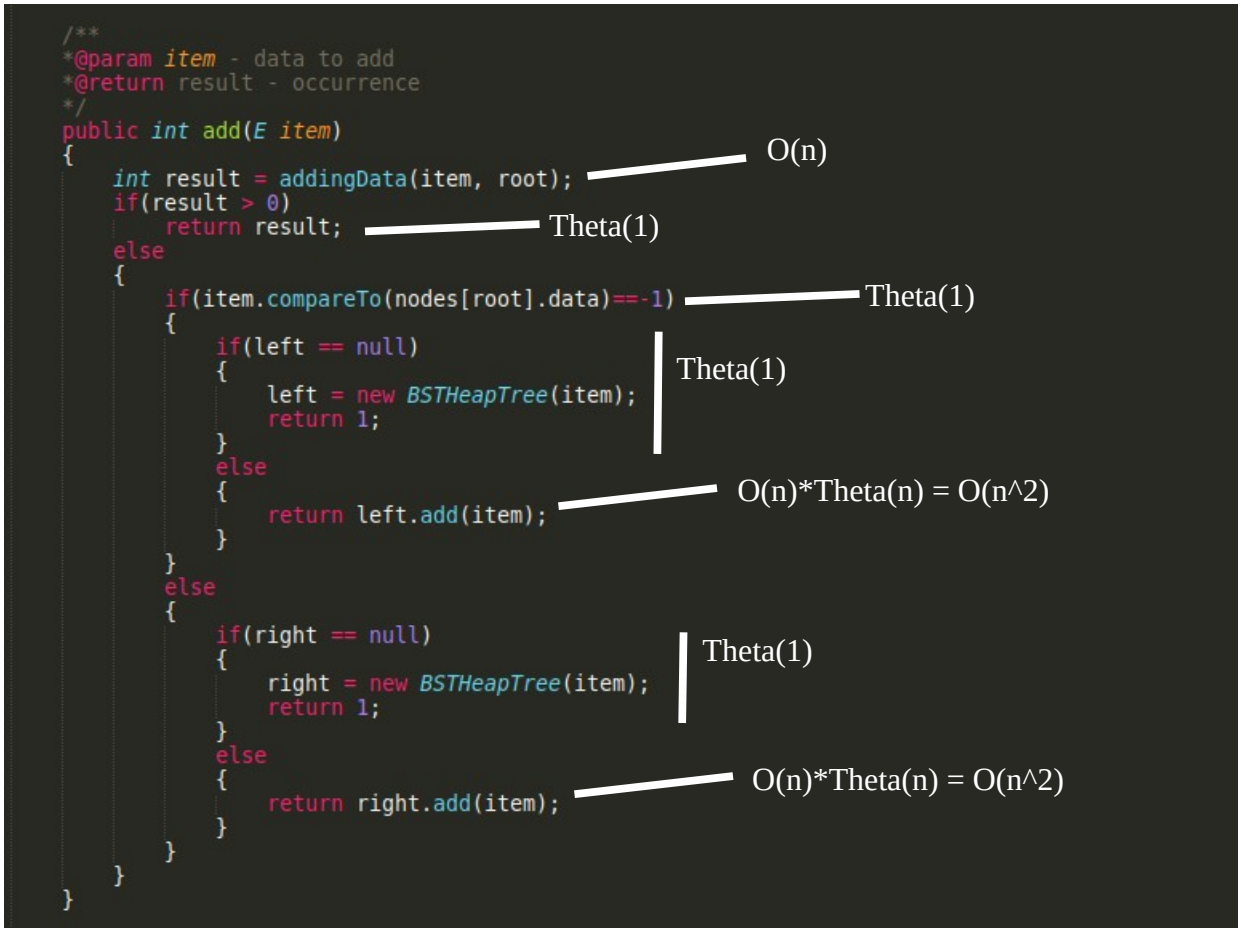



T(n)_best case = Theta(1)
T(n)_worst case = O(n)
T(n) = O(n)

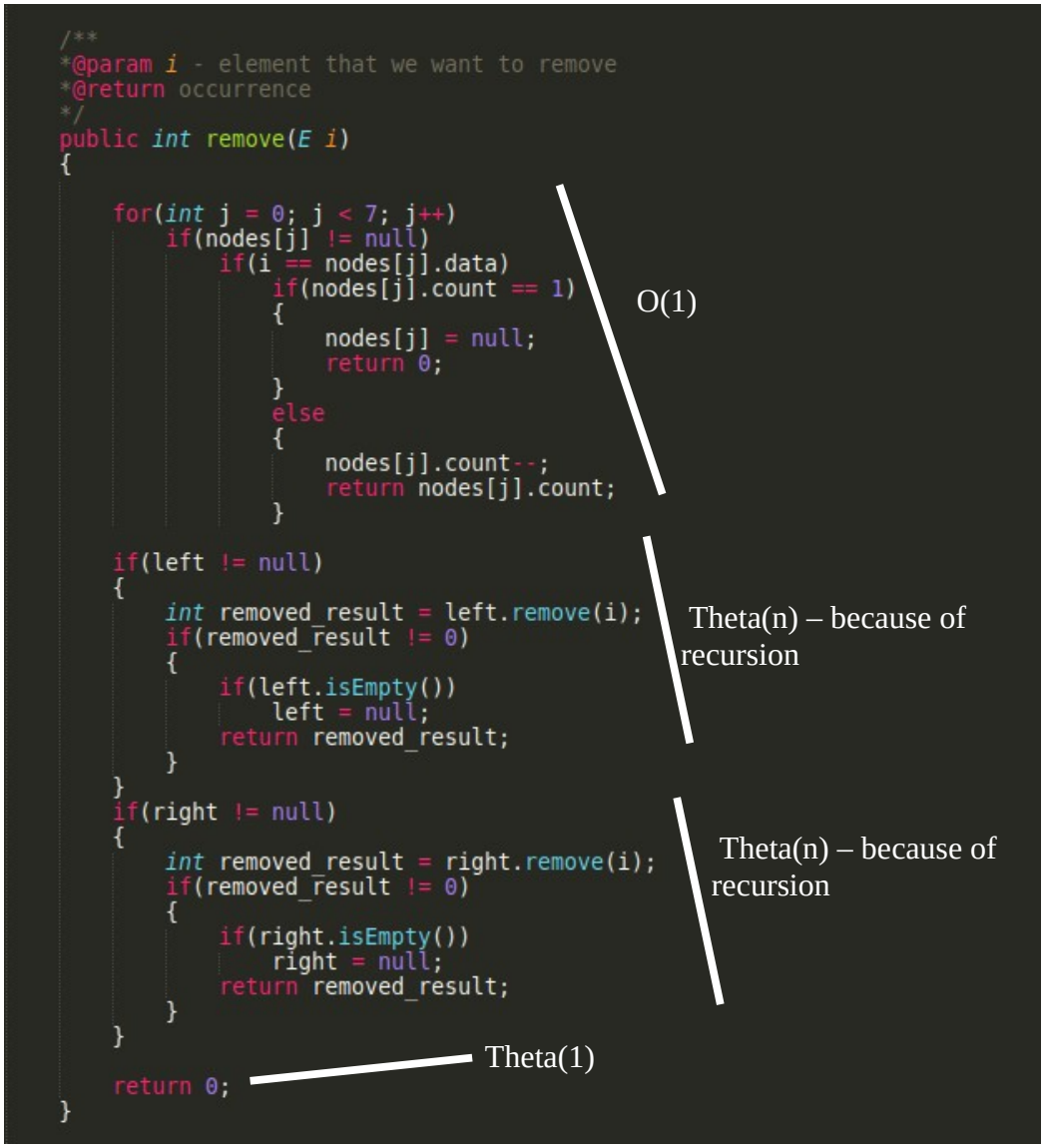

/**
* This method returns the index of the big node
*@param Node i
*@return next Node index
*/
public int getBig(int i)
{
if(i == root)
return bNode;
else if(i == bNode)
return bbNode;
else if(i == sNode)
return sbNode;
else return 0;
}
Theta(1)
/**
*@return true if it is empty else returns false
*/
public boolean isEmpty()
{
boolean val = nodes[root] == null;
boolean val2 = left == null ? true : left.isEmpty();
boolean val3 = right == null ? true : right.isEmpty();
return val && val2 && val3;
}
Theta(1)
/**
*@return c - number of total Nodes
*/
public int getNumberOfNodes()
{
int c = 0;
for(Node n : nodes)
if(n != null)
c++;
Tetha(n) – because of for loop
O(n) → time complexity
if(left != null)
c += left.getNumberOfNodes();
if(right != null)
c += right.getNumberOfNodes();
O(n) - recursion
return c;
}


/**
* @return array - all Nodes
*/
public Node[] getArrayOfNodes()
{
int size = getNumberOfNodes();
if(size == 0)
return null;
Node[] array = new Node[size];
int i = 0;
for(Node n : nodes)
if(n != null)
array[i++] = n;
if(left != null)
{
Node[] array2 = left.getArrayOfNodes();
for(Node n : array2)
array[i++] = n;
}
if(right != null)
{
Node[] array2 = right.getArrayOfNodes();
for(Node n : array2)
array[i++] = n;
}
return array;
}
O(n)
Theta(1)
Theta(n)
O(n^2) - recursion
O(n^2) recursion
Theta(1)
O(n^2) → time complexity

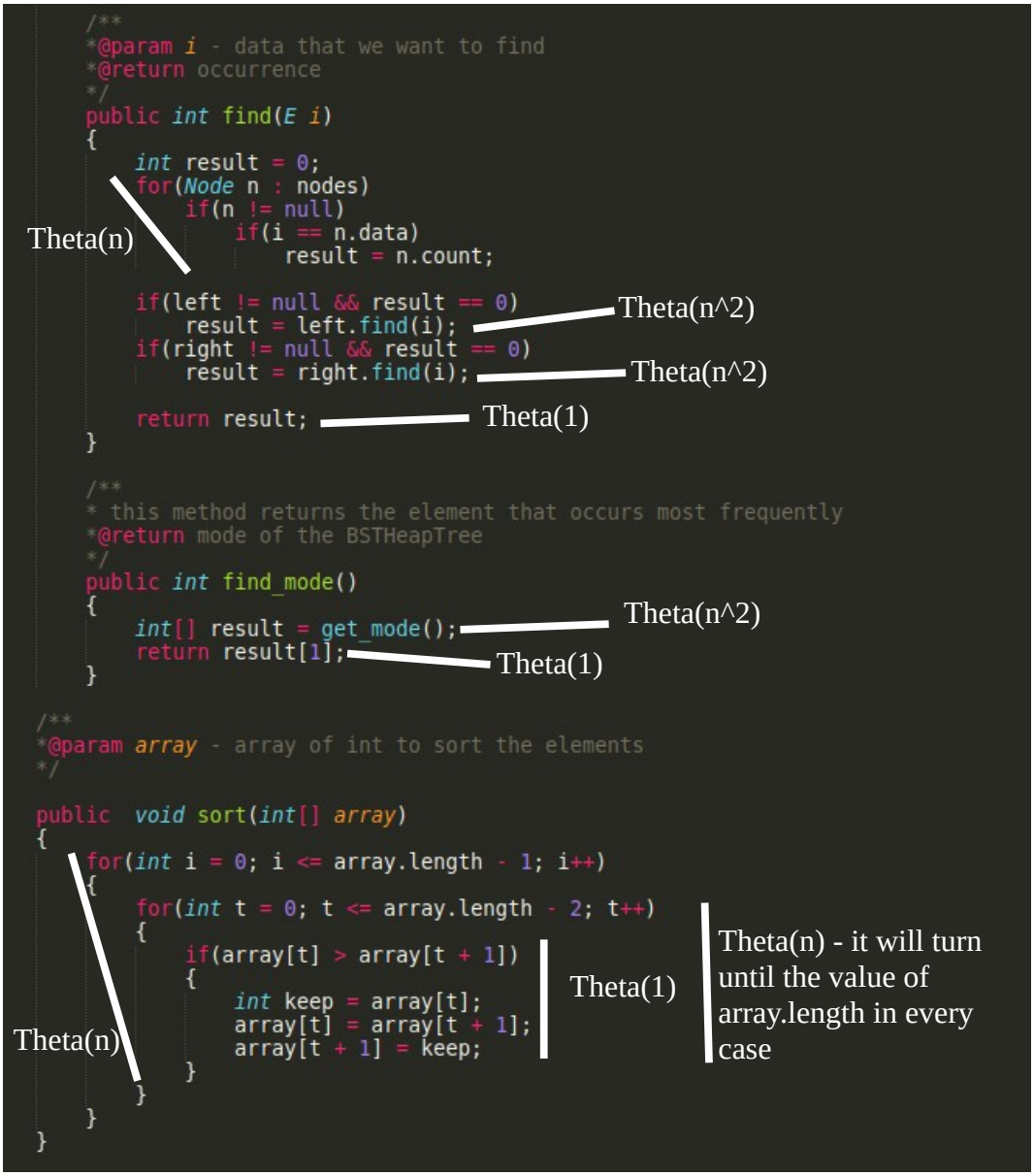

/**
*@return mode - array of highest occurrence and its value
*/
public int[] get_mode()
{
int[] modeArr = {0, 0};
for(Node n : nodes)
if(n != null && modeArr[0] < n.count)
{
modeArr[0] = n.count;
modeArr[1] = (int)n.data;
}
if(left != null)
{
int[] result = left.get_mode();
if(modeArr[0] < result[0])
{
modeArr[0] = result[0];
modeArr[1] = result[1];
}
}
if(right != null)
{
int[] result = right.get_mode();
if(modeArr[0] < result[0])
{
modeArr[0] = result[0];
modeArr[1] = result[1];
}
}
return modeArr;
}
Theta (n)
Theta (1)
Theta(n^2) - recursion
Theta(1)
Theta(n^2)



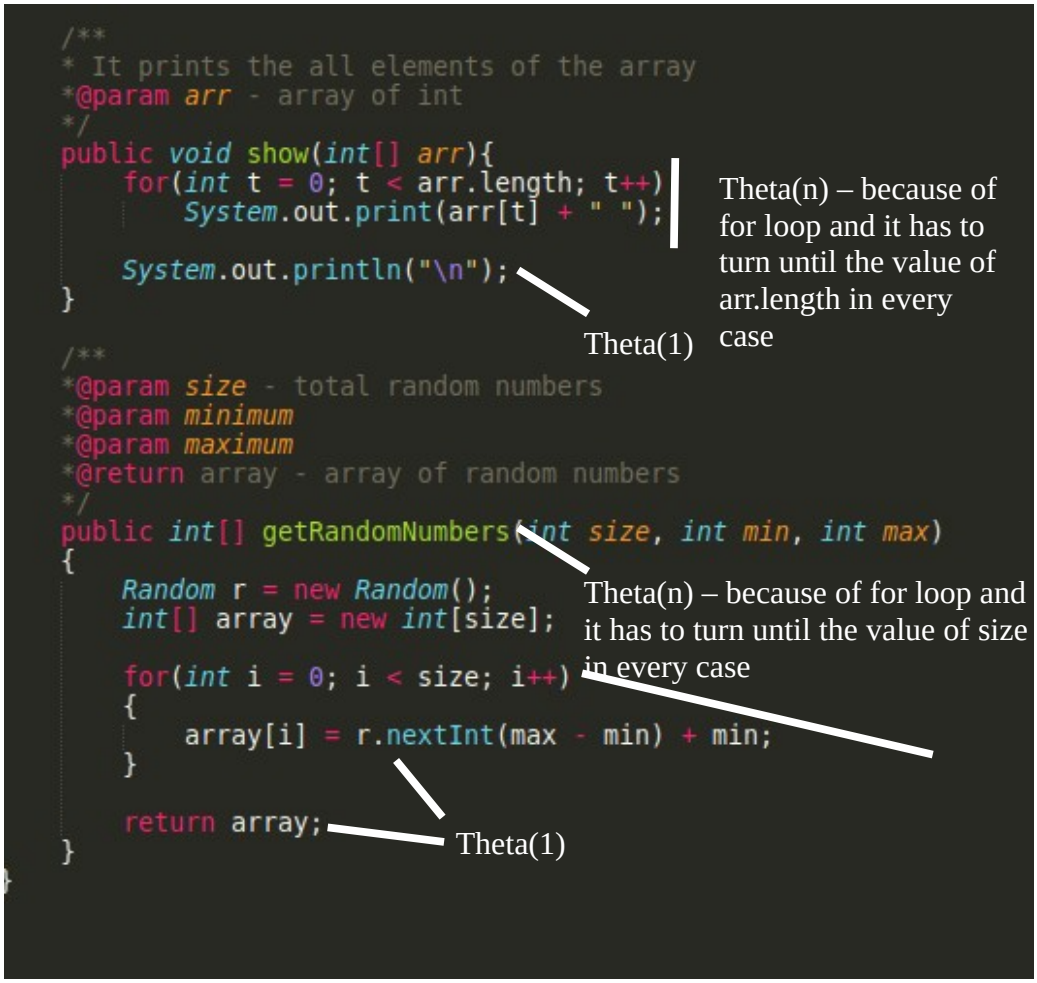

T(n)_best case = Theta(1)
T(n)_worst case = O(n^2)




T(n)_worst case = Theta(n)
T(n)_best case = O(1)
T(n) = Theta(n)




“find” method has T(n) = Theta(n^2)
“find_mode” method has T(n) = Theta(n^2) + Theta(1) = Theta(n^2)
“sort” method has T(n) = Theta(n)*Theta(n) = Theta(n^2)




“show ” method has T(n) = Theta(n) + Theta(1) = Theta(n)
“getRandomNumbers” T(n) = Theta(n) + Theta(1) = theta(n)