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

Mustafa Gurler

171044034

System Requirements

Part 1 of assignment split up four group. Searching operation , merging operation, removing ith biggest number in the heap and setting the last value of the heap with new data. Data needs the be added to heap first than these operations can be done successfully.

Searching operation, shall search the all heap and finds the given data from heap.

Merging operation, shall merge two different heap and reheapify the current heap.

Removing ith biggest number, shall remove given ith biggest number in the heap and reheapify.

Setting the last value of heap with new data shall use iterator and set the last data of heap with given value and reheapify current heap.

Part 2 of assignment needs to be added some new features for requirements.

Each node of binary search tree shall keep maximum 7 number. These numbers are compatible with max heap data structures. Maximum number always has to be at the top.

Adding new element to BST, program shall add it at the end and swim it up in a loop until reaches to its place. Number's parent needs to bigger than it and child of number needs to be smaller than it. Same number occurrences, the program shall increment the value of occurrence.

Removing a element in the BST, program shall traverse the data until reaches it in the heap, also traverses the left and right child if there are exist. Two possibility exist in the remove operation, if occurrences are bigger than 1 for same number , program shall decrement the

occurrence. If occurrence is 1 for same number, program shall delete the value and swim it all the values for max heap data structures rule.

Finding method in BST, program shall take a number from user and show the number occurrence in the BST.

Finding mode method in BST, program shall give the biggest occurrences in the BST.

Test Drive of a program shows all the possibility of these features. All the features are tried by user and programmer step by step.

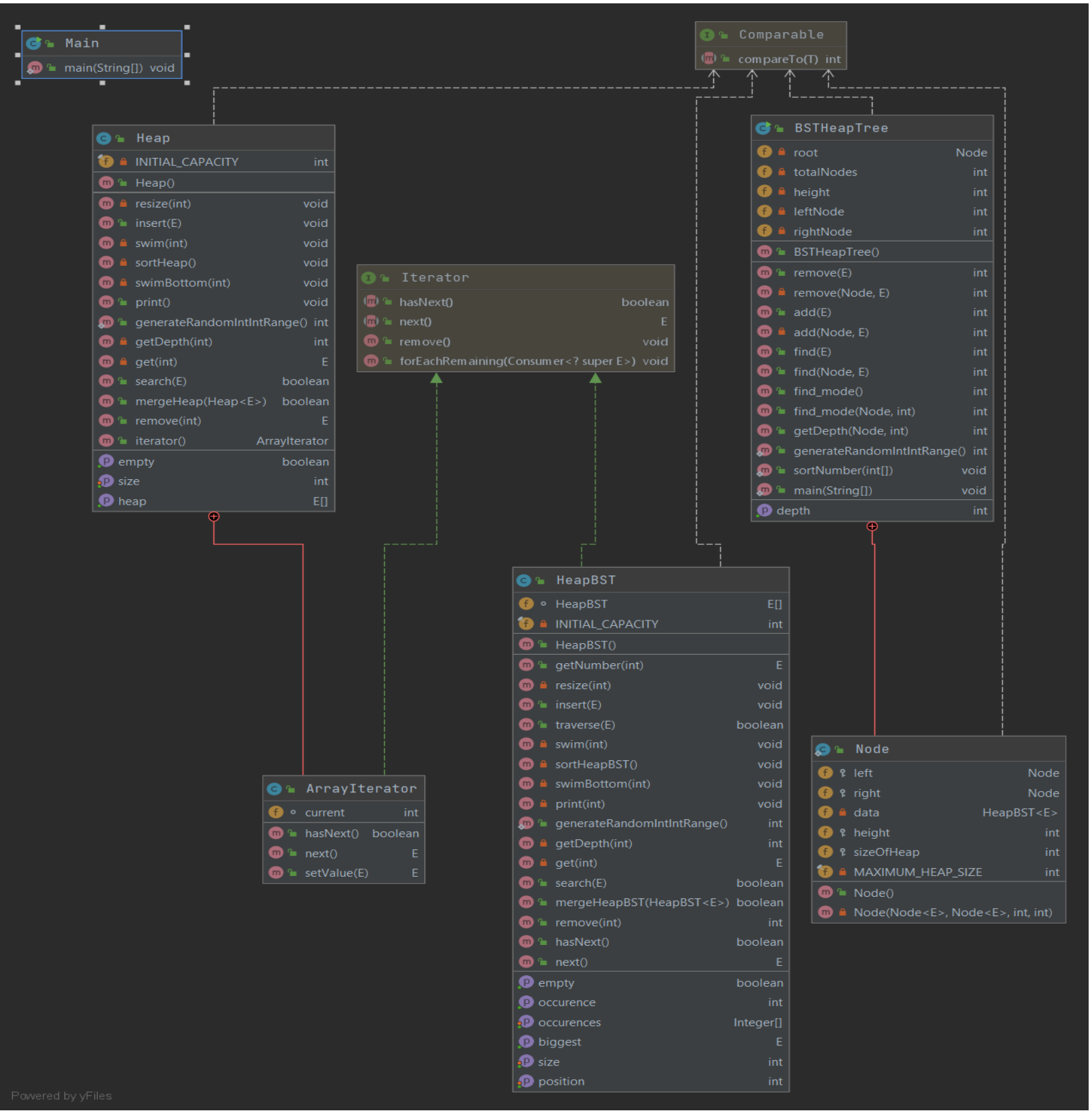
Environment of Tester PC:

Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

openjdk version "11.0.11-ea" 2021-04-20

OpenJDK Runtime Environment (build 11.0.11-ea+4-post-Debian-1)
OpenJDK 64-Bit Server VM (build 11.0.11-ea+4-post-Debian-1, mixed mode, sharing)

Class Diagram



Problem Solution Approach

I split up my Heap class and BST Heap class, I assume systematically

Requirements are more clear with two different class. Comparable interface has been implemented so two different numbers or variable easily can be compared by program.

Heap Class

I used MaxHeap implementation for my assignment. All numbers can be inserted up to down. Max number has to be in the top. Swim method carries all the inserted value to top if inserted values are bigger than their parents. SwimBottom method carries all the values to down. It is a helper class that does not have much responsibilities for the program and user.

Merging Heap implementation is very similar to insert method. Instead of one element , inserting all the other heap variable to current heap variable.

Removes method implementation is opposite of inserting element to heap. Data needs to change with last element in the heap. Size needs to be decrement just one time. After that changed last element needs go down if children of element bigger than our last element. SwimBottom method is used for this method.

Iterator method implemented from Iterable class and wrote down for ArrayIterator class which implements Iterator class. Instead of hasNext method and next method, setValue method has been added to ArrayIterator class and it can be used for sets the last value of heap and swim it up to top.

BST Heap Class:

BST Heap Class does the same thing with heap class. But I added some new features so same numbers can be added to the heap with occurrence variable. Same numbers can be added one times and occurrence will be incremented each time only one.

BST has a node which has a BST Heap class which numbers can be added maximum seven time for every each node. Left child of the root keeps smaller numbers than root, right child of the root keeps bigger numbers than root.

Inserting elements starts with root, if root has been filled before and the item is not in the root heap, left and right child's biggest data compared with item and item goes its way proper direction recursively. If item has been added before, occurrence of that item needs to be incremented one.

Removing item starts with root, opposite of inserting method. Program traverses the root's heap, compares the number with left and right child of current parent and tries the find item in the heap. If occurrence is bigger than 1 , occurrence has been decremented only one, if it is not , item needs to be deleted by the program and all the heap needs to be used by swim method.

Find method , implemented the find of the given item in the BST, traverses the root's heap, compares the number with left and right child of current parent and tries the find item in the heap. If it is in the heap , return the occurrence of item.

Find mode method , uses the find method , tries to find maximum occurrence in the heap.

Test Cases Running Command and Results

Part1

Insert Method:

Main

```
public static void main(String[] args){
    Heap<Integer> i = new Heap<>();
    for(int k = 0; k<15 ; k++){
        int y = generateRandomIntIntRange();
        System.out.print(y + " ");
        i.insert(y);
    }
    System.out.println();
    i.print(15);
}
```

Console

```
Insert Method:
658 4099 1558 1499 120 2474 2467 3233 929 2168 1578 1960 24 220 3505
4099 3233 3505 1499 2168 1960 2474 658 929 120 1578 1558 24 220 2467
```

First numbers are random numbers, Second one numbers with max heap data structure hierarchy

Search Method:

Main

```java
public static void main(String[] args){
    System.out.println("Search number:");
    Scanner input = new Scanner(System.in);
    Heap<Integer> j = new Heap<>();
    for(int i=0 ; i<20 ; i++){
        j.insert(generateRandomIntIntRange());
    }
    j.print(20);
    System.out.println();
    System.out.print("enter a number:");
    int sayi = input.nextInt();

    if(j.search(sayi)){
        System.out.println("number has been found");
    }else{
        System.out.println("number has not been found");
    }

}
```

Number has been added and searching one specific number in the heap. If number has been found returns true

True case

```
Search number:
4695 4418 4034 4236 4006 2932 2133 1579 3970 3280 624 941 1998 414 1239 1576 1496 2567 93 420
enter a number:420
5.depth 4.number
number has been found
```

420 is in the heap and shows the depth and place in the heap.

False Case

```
Search number:
4955 4905 4450 3996 4857 3068 3278 3260 2753 2803 519 200 683 2970 794 1269 2882 1713 2396 87
enter a number:42
number has not been found
```

Merge Heap Method:

Main

```
public static void main(String[] args){

    Heap<Integer> k = new Heap<>();
    Heap<Integer> j = new Heap<>();
    for(int i=0 ; i<5 ; i++){
        j.insert(generateRandomIntIntRange());
    }
    for(int i=0 ; i<5 ; i++){
        k.insert(generateRandomIntIntRange());
    }
    j.print(5);
    System.out.println();
    k.print(5);
    j.mergeHeap(k);
    System.out.println();
    j.print(10);
```

Two different heap has been created and random number has been inserted 10 times to each heap.

Console

```
Merging to heap
3049 2790 2045 920 663
4390 4078 3968 2208 3015
4390 3968 4078 2790 3015 2045 3049 920 2208 663
```

After heap's merged operation, all the datas has to be swim up and down again.

Iterator Method:

Main

```
Heap.ArrayIterator iter = j.iterator();
System.out.println();
System.out.println("Iterator in the Heap");
System.out.println("First item in the heap(iterator next method):" + iter.next());
System.out.println("Second item in the heap(iterator next method):" + iter.next());
System.out.println("iterator set last value in the heap:" + iter.setValue(2700));
j.print();
System.out.println();
```

Console

2700 number added to last item of the heap and it needs to go up for Heap data structure hierarchy

```
Iterator in the Heap
First item in the heap(iterator next method):4778
Second item in the heap(iterator next method):4633
4778 4633 3935 2947 3569 3791 3044 1903 2625 2914 1998 1271 1059 2973 1952 878 558 183 1187
iterator set last value in the heap:1187
4778 4633 3935 2947 3569 3791 3044 1903 2700 2914 1998 1271 1059 2973 1952 878 558 183 2625
```

Test Cases Running Command and Results

Part2

Add Method:

Main

```
public static void main(String[] args){
    BSTHeapTree<Integer> bst = new BSTHeapTree<>();
    int[] array = new int[3000];
    int random_number;
    for(int i=0 ; i<3000 ; i++){
        random_number = generateRandomIntIntRange();
        bst.add(random_number);
        array[i] = random_number;
    }
    sortNumber(array);
    for(int i=0 ; i<3000 ; i++){
        System.out.println(array[i] + " " + bst.find(array[i]));
    }
```

3000 random number generated and added to BST and array . Array has been sorted to see how many same number added to heap and show the occurrence for the same number

Console1 ***Console2***

As you see same number occurrences counted by the find method.

Remove Method:

Main

```
for(int i=0 ; i<100 ; i++){
    System.out.println("Number:" + array[i] + " Before remove:" +  (bst.remove(array[i])+1) + " After remove:" + bst.find
}
for(int i=0 ; i<10 ; i++){
    System.out.println((5000+i) + " " + bst.remove( item: 5000+i) + " " + bst.find( item: 5000+i));
    // after 5000 , numbers are not in the bst
}
```

Remove method has been tested with 100 numbers inside of the heap and 10 numbers not inside of heap.

Console1 for removing element in the heap

```
Number:33 Before remove:1 After remove:0
Number:39 Before remove:1 After remove:0
Number:40 Before remove:1 After remove:0
Number:42 Before remove:1 After remove:0
Number:43 Before remove:1 After remove:0
Number:44 Before remove:3 After remove:2
Number:44 Before remove:2 After remove:1
Number:44 Before remove:1 After remove:0
Number:45 Before remove:1 After remove:0
Number:47 Before remove:2 After remove:1
Number:47 Before remove:1 After remove:0
Number:48 Before remove:2 After remove:1
Number:48 Before remove:1 After remove:0
Number:53 Before remove:2 After remove:1
Number:56 Before remove:3 After remove:2
Number:56 Before remove:2 After remove:1
Number:56 Before remove:1 After remove:0
Number:58 Before remove:2 After remove:1
Number:59 Before remove:3 After remove:2
Number:59 Before remove:2 After remove:1
Number:59 Before remove:1 After remove:0
Number:61 Before remove:1 After remove:0
Number:62 Before remove:3 After remove:2
Number:62 Before remove:2 After remove:1
Number:62 Before remove:1 After remove:0
Number:64 Before remove:1 After remove:0
Number:65 Before remove:4 After remove:3
Number:65 Before remove:3 After remove:2
Number:65 Before remove:2 After remove:1
Number:65 Before remove:1 After remove:0
Number:66 Before remove:2 After remove:1
```

As you see numbers are removed one by one, When you look 56, its occurrence is 3, three times 56 has been removed.

Console2 for removing element not in the heap

```
Number:103 Before remove:2 After remove:1
Number:105 Before remove:1 After remove:0
Number:108 Before remove:2 After remove:1
Number:108 Before remove:1 After remove:0
Number:109 Before remove:1 After remove:0
Number:110 Before remove:2 After remove:1
Number:110 Before remove:3 After remove:2
Number:114 Before remove:1 After remove:0
Number:115 Before remove:1 After remove:0
Number:117 Before remove:1 After remove:0
Number:119 Before remove:1 After remove:0
Number:120 Before remove:1 After remove:0
5000 0 0
5001 0 0
5002 0 0
5003 0 0
5004 0 0
5005 0 0
5006 0 0
5007 0 0
5008 0 0
5009 0 0
```

After 5000 , numbers are not in the heap so it shows any occurrence in the heap.

Find Method:

Main

```
public static void main(String[] args){
    BSTHeapTree<Integer> bst = new BSTHeapTree<>();
    int[] array = new int[3000];
    int random_number;
    for(int i=0 ; i<3000 ; i++){
        random_number = generateRandomIntIntRange();
        bst.add(random_number);
        array[i] = random_number;
    }
    sortNumber(array);
/*    for(int i=0 ; i<3000 ; i++){
        System.out.println(array[i] + " " + bst.find(array[i]));
    }
*/
    for(int i=0 ; i<100 ; i++){
        System.out.println(array[i] + " " + bst.find(array[i]));
    }
    for(int i=0 ; i<10 ; i++){
        System.out.println((5000+i) + " " + bst.find( item: 5000+i));// after 5000 , numbers are not in the bst
    }
```

Finds the number occurrences in the heap, if number is in the heap program shows the occurrence in the heap

Console for find element occurrence in the heap

```
117 1
118 2
118 2
120 1
122 1
123 1
124 2
124 2
126 2
126 2
127 1
128 1
129 1
130 3
130 3
130 3
132 4
132 4
132 4
132 4
133 1
5000 0
5001 0
5002 0
5003 0
5004 0
5005 0
5006 0
5007 0
5008 0
5009 0
```

100 different numbers occurrence in the heap, and 10 different numbers occurrence in the heap,

Find Mod Method:

Main

```
public static void main(String[] args){
    BSTHeapTree<Integer> bst = new BSTHeapTree<>();
    int[] array = new int[3000];
    int random_number;
    for(int i=0 ; i<3000 ; i++){
        random_number = generateRandomIntIntRange();
        bst.add(random_number);
        array[i] = random_number;
    }
    sortNumber(array);
/*    for(int i=0 ; i<3000 ; i++){
        System.out.println(array[i] + " " + bst.find(array[i]));
    }
*/
    for(int i=0 ; i<3000 ; i++){
        System.out.print(array[i] + " " + bst.find(array[i]) + " ");
        if(bst.find(array[i]) == bst.find_mode()){
            System.out.print("Biggest Occurence in bst is this number!!!");
        }
        System.out.println();
```

Finds the all numbers occurrence in the heap and compares with biggest occurrence in the heap

Console

```
408 1
409 1
411 1
412 1
413 5 Biggest Occurence in bst is this number!!!
413 5 Biggest Occurence in bst is this number!!!
413 5 Biggest Occurence in bst is this number!!!
413 5 Biggest Occurence in bst is this number!!!
413 5 Biggest Occurence in bst is this number!!!
414 1
415 3
415 3
415 3
420 1
421 2
421 2
422 1
423 1
426 1
427 1
428 2
428 2
429 1
430 1
431 1
432 1
433 1
437 1
438 1
439 1
```

As you see 413 and 4944 have the biggest occurrence in the heap

Console2

```
4913 2
4913 2
4920 1
4923 1
4924 2
4924 2
4925 1
4926 1
4929 1
4933 1
4936 3
4936 3
4936 3
4937 1
4941 1
4942 1
4944 5 Biggest Occurence in bst is this number!!!
4944 5 Biggest Occurence in bst is this number!!!
4944 5 Biggest Occurence in bst is this number!!!
4944 5 Biggest Occurence in bst is this number!!!
4944 5 Biggest Occurence in bst is this number!!!
4946 1
4950 2
4950 2
4951 2
4951 2
4955 2
4955 2
4956 1
```

Time Complexity analysis For Heap Class

GetSize Method:

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

O(1)

setSize Method:

```
/**
 *
 * @param size new size of the heap
 */
public void setSize(int size) { this.n = size; }
```

O(1)

isEmpty Method:

```
/**
 * @return checks if heap is empty
 */
public boolean isEmpty() { return (n == 0); }
```

O(1)

resize Method:

```java
/**
 * extends the length of heap
 * @param capacity extended size of heap
 */
private void resize(int capacity) { heap = Arrays.copyOf(heap, capacity); }
```

Amortized O(1)

swim Method:

```java
/**
 * reheapify the heap bottom-up
 * @post (k/2) can not be 0.
 * @param k size of heap
 */
private void swim(int k){
    if(k/2 == 0){
        return;
    }
    int i = heap[k/2].compareTo(heap[k]);
    while(k > 1 && i < 0){
        E temp = heap[k/2];
        heap[k/2] = heap[k];
        heap[k] = temp;
        k = k/2;
        if(k/2 == 0){
            return;
        }
        i = heap[k/2].compareTo(heap[k]);
    }
}
```

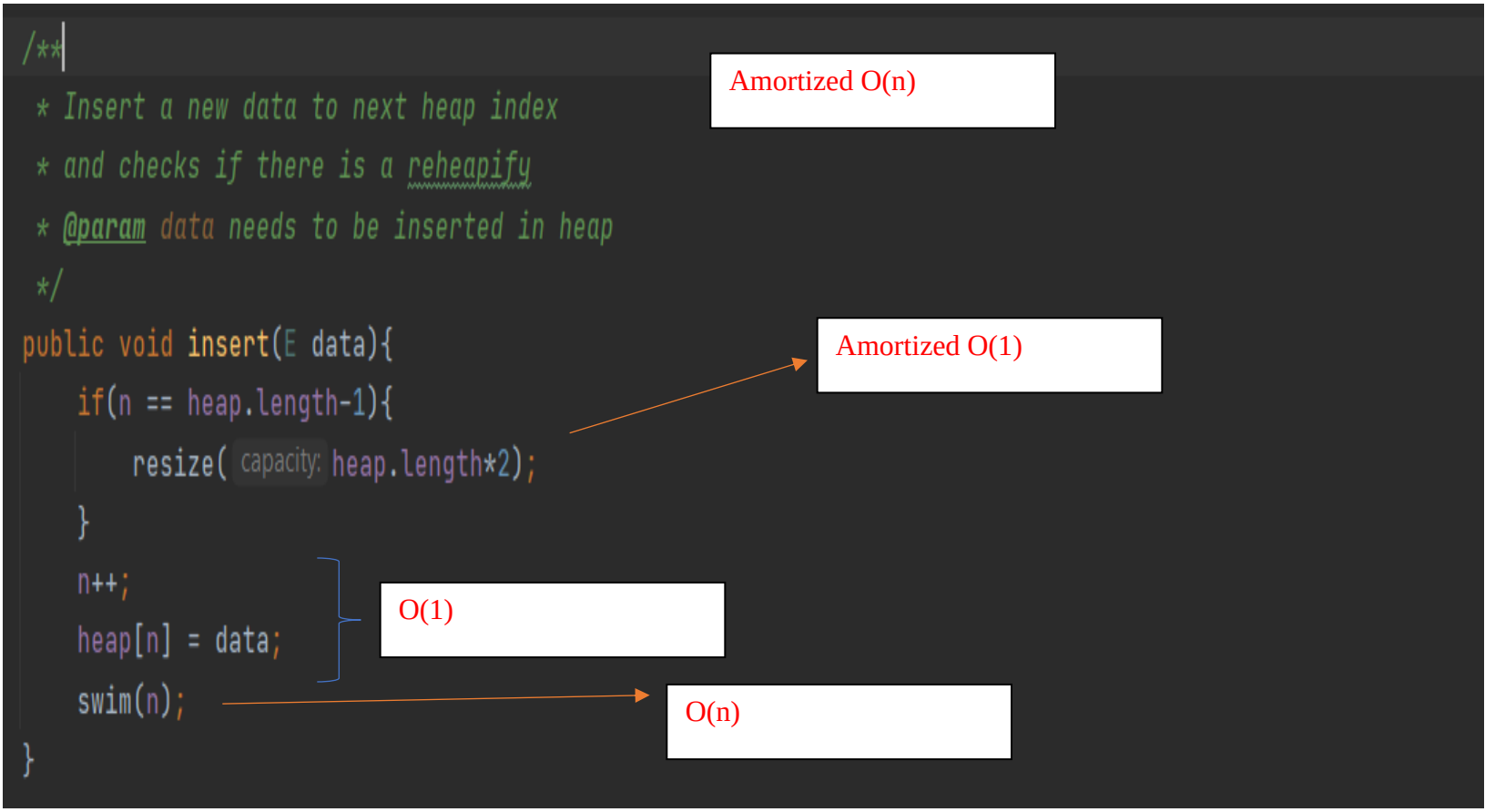
O(n)

O(1)

O(n)

O(1)

insert Method:



swimBottom Method:

```java
/**
 * carries the smaller element to bottom from top
 * if the element is not the right position
 * @param k the hill of the tree needs to be changed
 */
private void swimBottom(int k){
    if((k*2)+1 > n){
        return;
    }
    int j = heap[(k*2)].compareTo(heap[(k*2)+1]);// compares left and right child // 1 yada -1 gelcek
    int i = heap[(k*2)].compareTo(heap[k]); // compares left child and parent // 1 olursa gircek
    int m = heap[(k*2)+1].compareTo(heap[k]); //compares the right child and parent // 1 olursa gircek
    if(j == 1 && i == 1){ // if left child is bigger than parent switch them
        E temp = heap[k*2];
        heap[k*2] = heap[k];
        heap[k] = temp;
        swimBottom( k: k*2);
    }else if(j == -1 && i == 1){ // if right child is bigger than parent switch them.
        E temp = heap[(k*2)+1];
        heap[(k*2)+1] = heap[k];
        heap[k] = temp;
        swimBottom( k: (k*2)+1);
    }
}
```

Function runs O(1) time but inside of the swimBottom recursive function has been called. So time complexity is O(N)

O(1)

O(1)

O(1)

O(n)

print Method:

```java
/**
 * shows the heap top to bottom
 */
public void print(){
    for(int i=1 ; i <= getSize() && heap[i] != null ; i++){
        System.out.print(heap[i] + " ");
    }
}
```

O(n)

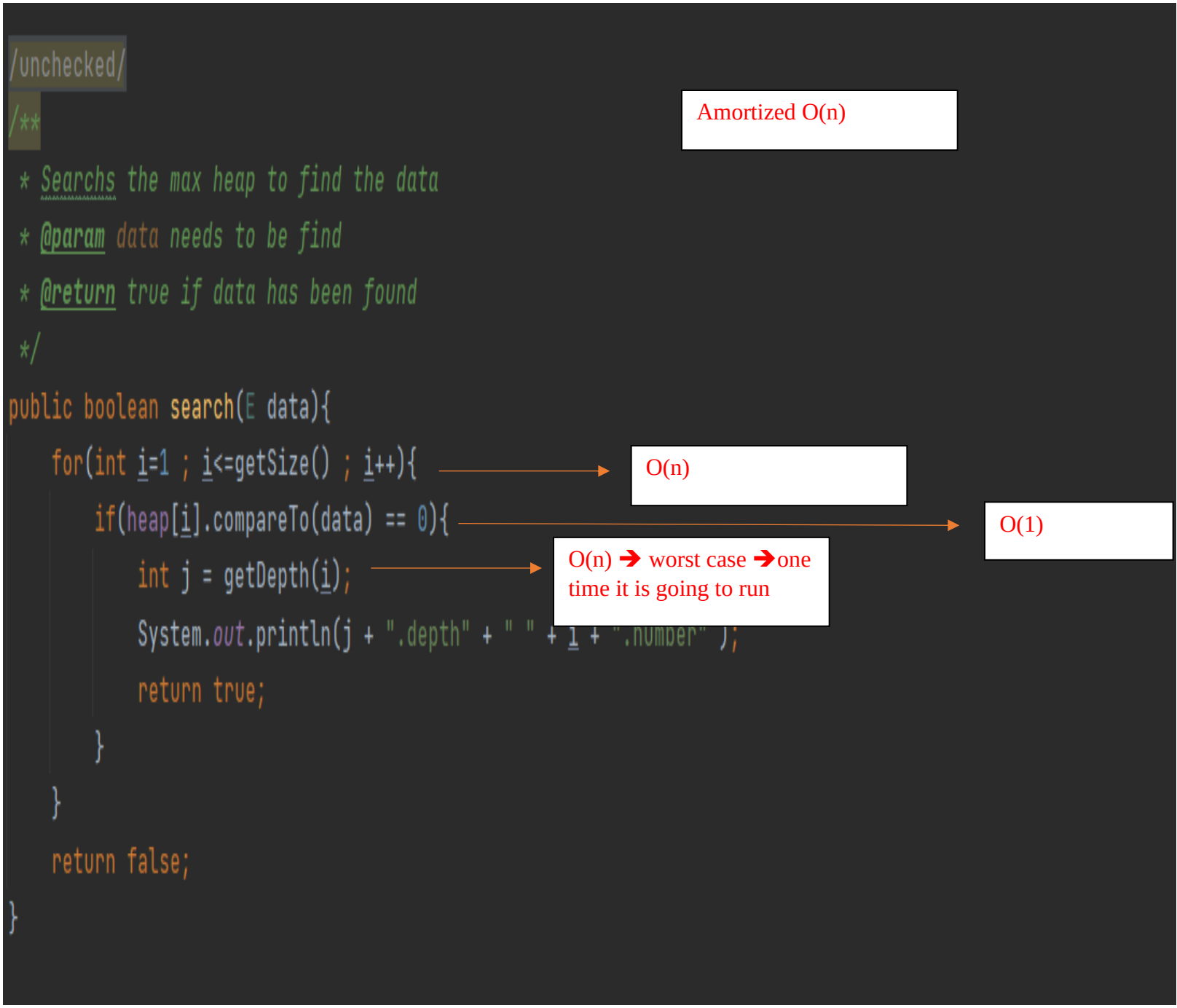
getDepth:

```java
/**
 * finds the depth of given heap's data
 * @param i given data
 * @return returns the depth of given data
 */
private int getDepth(int i){
    for(int j = 0 ; j < i ; j++){
        if((int)(i/Math.pow(2,j)) == 1){
            return j+1;
        }
    }
    return 0;
}
```

O(n) ➔ worst case

O(1)

search Method:



merge Heap:

```java
/**
 * Merges heap with another heap
 * @param other other heap
 * @return true if merge operation is successful
 */
public boolean mergeHeap(Heap<E> other){
    for(int i=1 ; i <= other.getSize() ; i++){
        this.insert(other.get(i));
    }
    return true;
}
```

O(n,m) ➔ O(N2)

O(m)

O(n)

get Heap:

```java
/**
 * gets heap array
 * @return
 */
public E[] getHeap() { return heap; }
```

O(1)

remove Method:

```java
/**
 * removes the ith largest element in the heap
 * @param i place of the element
 * @return data's of the elemet
 */
public E remove(int i){
    if(i < 0 || i > n){
        throw new NoSuchElementException("Index is wrong");
    }
    if(n == 0){
        throw new NoSuchElementException("Heap is empty");
    }
    E result = get(i);
    if(n == 1){
        setSize(0);
    }else{
        heap[i] = heap[n];
        setSize(getSize()-1);
        swimBottom(i);
    }
    return result;
}
```

Worst case ➔O(n)

Best Case ➔ O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(n)

O(n)

Array Iterator class:

```java
/**
 * Iterator class for Heap class
 */
public class ArrayIterator implements Iterator<E> {
    int current = 1;  // the current element we are looking at
    // starts with first array

    // return whether or not there are more elements in the array that
    public boolean hasNext() { return current < n; }

    // return the next element of the iteration and move the current
    public E next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return heap[current++];
    }

    /**
     * items need to be inserted to heap , starts from end and change the
     * last value of heap
     * @param item the data needs to be inserted
     * @return last value of heap
     */
    public E setValue(E item){
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        E result = heap[n];
        heap[n] = item;
        swim(n);
        return result;
```

O(1)

O(1)

O(n)

O(n)

Time Complexity analysis For Heap BST Class

****Just added the new features time complexity in here***

Traverse Method:

```
/**
 * Traversing the HeapBST to find if there is an equal item
 * @param data needs to be added
 * @return if data is in the HeapBST returns true
 */
public boolean traverse(E data){
   if(HeapBST == null){
       return false;
   }
   position = 1;
   for(int i=1 ; i <= getSize() ; i++){
       setPosition(i);
       if(HeapBST[i].compareTo(data) == 0 ){
           return true;
       }
   }
   return false;
}
```

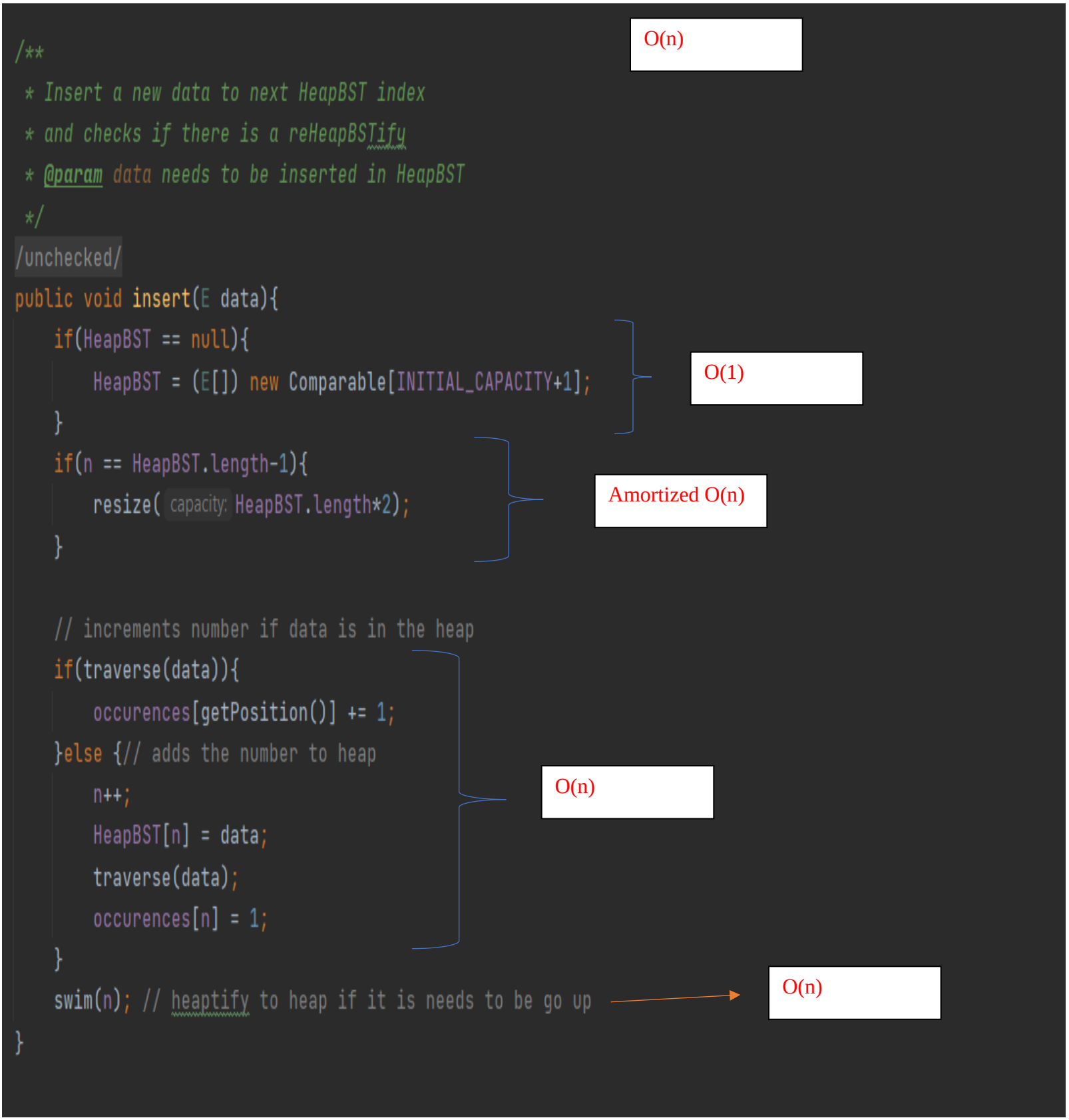
O(n)

O(1)

O(n) ➔worst case

O(1) ➔ best case

Insert Method:




```
/**
 * Insert a new data to next HeapBST index
 * and checks if there is a reHeapBSTify
 * @param data needs to be inserted in HeapBST
 */
/unchecked/
public void insert(E data){
    if(HeapBST == null){
        HeapBST = (E[]) new Comparable[INITIAL_CAPACITY+1];
    }
    if(n == HeapBST.length-1){
        resize( capacity: HeapBST.length*2);
    }

    // increments number if data is in the heap
    if(traverse(data)){
        occurences[getPosition()] += 1;
    }else {// adds the number to heap
        n++;
        HeapBST[n] = data;
        traverse(data);
        occurences[n] = 1;
    }
    swim(n); // heaptify to heap if it is needs to be go up
}
```

Time Complexity analysis For BST Class

Remove Method

Add Method:

```java
/**
 * adds item to heap
 * @param item needs to be inserted
 * @return the value of occurence of heap
 */
public int add(E item){
    Node current = root;
    return add(current, item);
}

/**
 * adds item to heap
 * @param current root or left and right child of root
 * @param item needs to be inserted
 * @return the value of occurence of heap
 */
private int add(Node current, E item) {
    // if heap is full there is no needed to be added of heap
    if(current.sizeOfHeap != (current.MAXIMUM_HEAP_SIZE)) {
        if(!(current.data.traverse(item))){
            current.sizeOfHeap++;
        }
        current.data.insert(item);
        return (Integer) current.data.getOccurence();
    }
    // if heap has same value with item increments to number of this item
    if(current.sizeOfHeap == (current.MAXIMUM_HEAP_SIZE) && (current.data.traverse(item))){
        current.data.insert(item);
        return (Integer) current.data.getOccurence();
    }
```

O(n)

O(n)

O(n)

O(n)

O(1)

O(n)

```
    // compares the items if item is smaller than heap
    if(current.left != null && current.data.getBiggest().compareTo(item) == 1){
        return add(current.left, item);
    }
    // compares the items if item is smaller than heap
    else if(current.right != null && current.data.getBiggest().compareTo(item) == -1){
        return add(current.right, item);
    }else if(current.data.getBiggest().compareTo(item) == 1){
        Node left = new Node();
        current.left = left;
        current = current.left;
        current.sizeOfHeap++;
        current.data.insert(item);
        return (Integer) current.data.getOccurence();
    }else if(current.data.getBiggest().compareTo(item) == -1){
        Node right = new Node();
        current.right = right;
        current = current.right;
        current.data.insert(item);
        current.sizeOfHeap++;
        return (Integer) current.data.getOccurence();
    }
    return 0;
}
```

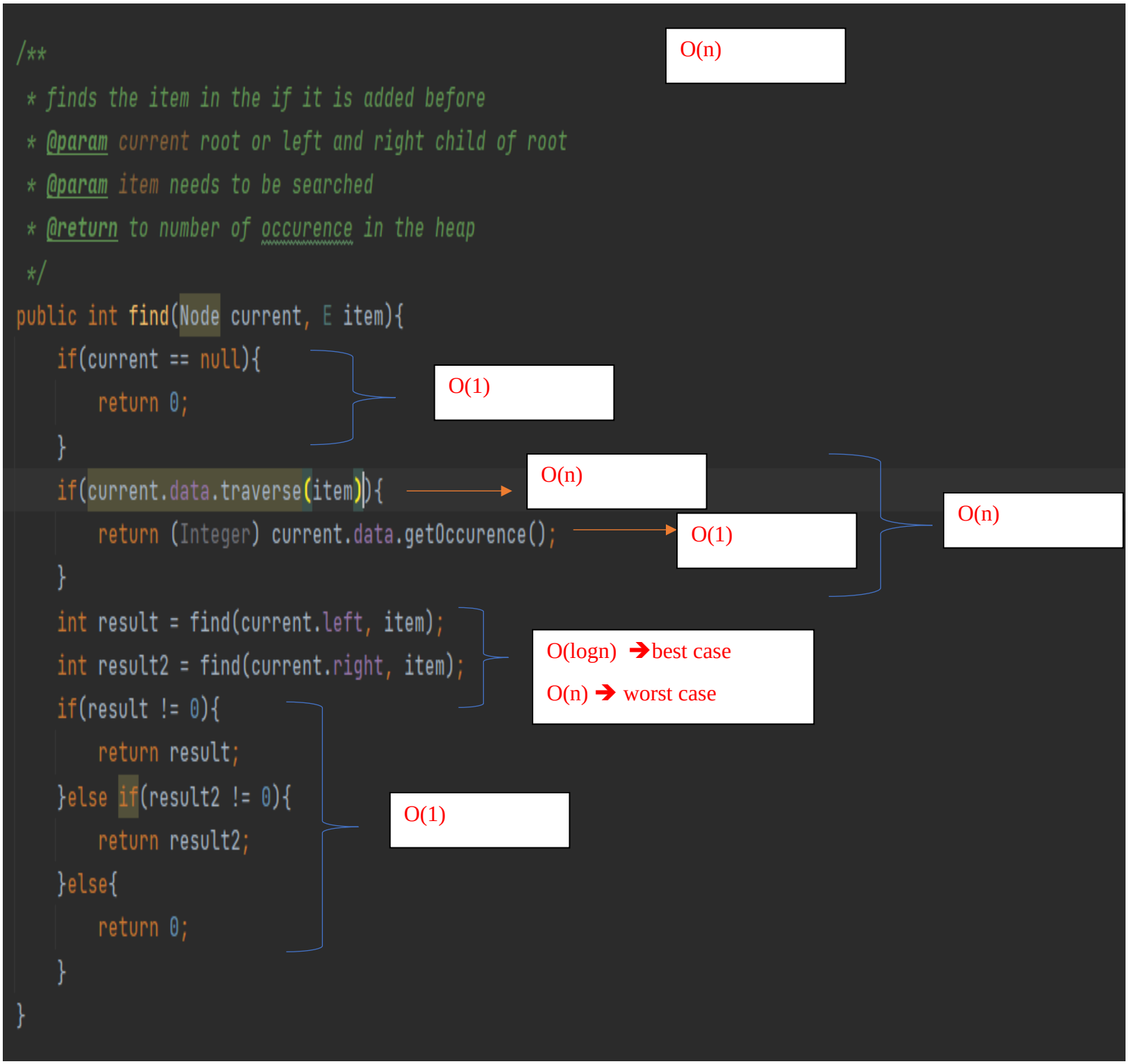
O(n)

O(n)

O(n)

O(n)

Find Method:


/**
* finds the item in the if it is added before
* @param current root or left and right child of root
* @param item needs to be searched
* @return to number of occurence in the heap
*/
public int find(Node current, E item){
if(current == null){
return 0;
}
if(current.data.traverse(item)){
return (Integer) current.data.getOccurence();
}
int result = find(current.left, item);
int result2 = find(current.right, item);
if(result != 0){
return result;
}else if(result2 != 0){
return result2;
}else{
return 0;
}
}
O(n)
O(1)
O(n)
O(1)
O(n)
O(logn) ➔best case
O(n) ➔ worst case
O(1)

Find Mode Method:

```java
/**
 * finds the biggest occurence in the heap
 * @param current root or left and right child of root
 * @param max max number of occurence. starts with zero
 * @return biggest occurence
 */
public int find_mode(Node current, int max){
    if(current == null){
        return max;
    }
    int maxHeapData = 0;
    //searches the heap if there is more biggest occurence than before
    for(int i=1 ; i <= current.sizeOfHeap ; i++){
        current.data.setPosition(i);
        maxHeapData = current.data.getOccurence();
        current.data.setPosition(i);
        if(maxHeapData > max){
            max = maxHeapData;
        }
    }
    //tries to reach left and right child if there is more biggest occurence than before
    int max1 = (int)find_mode(current.left, max);
    int max2 = (int)find_mode(current.right, max);
    if(max1 > max2){
        return max1;
    }else{
        return max2;
    }
}
```

O(nlogn)

O(1)

O(n)

O(n)

O(1)

O(1)

O(1)

O(nlogn)