GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CSE222/505 – Spring 2021

Homework 4 Report

Yakup Talha Yolcu

1801042609

PART 1

1. PROBLEM DEFINITION

Implementation of Heap and some extra properties.

i. Search for an element
ii. Merge with another heap
iii. Removing i^{th} largest element from the Heap
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.

Heap is data structure that we can remove just root of it and add element at the bottom left.

2. SYSTEM REQUIREMENTS

We need to have elements in the heap.

We need another heap to merge them.

We need to remove ith larges element of the heap.

We search for element in the heap.

```
THeap<Integer> th=new THeap<>();

th.offer( item: 5);
th.offer( item: 6);

THeap<Integer> tx=new THeap<>();

tx.offer( item: 10);

th.merge(tx);

System.out.println(th);

try {
    System.out.println(th.remove_ith( index: 1));
}
catch (IndexOutOfBoundsException | NullPointerException ne) {
    System.out.println(ne);
}
```

```
System.out.println(th.find( item: 5));
```

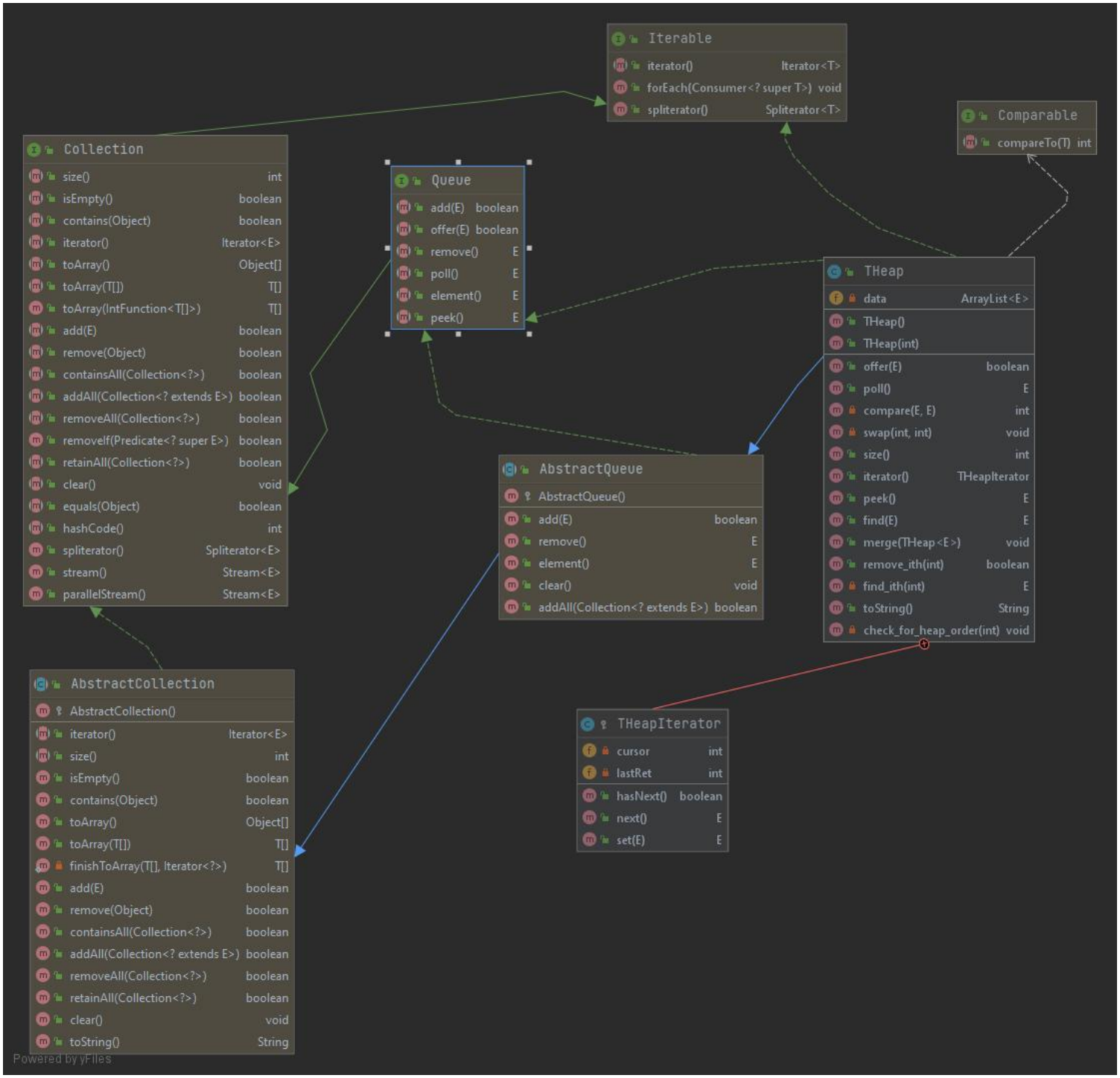
We need to have iterator to set an element

```
THeap.THeapIterator it=  th.iterator();
System.out.println(th);
it.next();
it.next();

try {
    it.set(60);
}
catch (IllegalArgumentException ne) {
    System.out.println(ne);
}


System.out.println(th);
```

3. CLASS DIAGRAM



4. PROBLEM SOLUTION APPROACH

```
public boolean offer(E item) {
    if(item==null) {
        return false;
    }
    data.add(item);
    int child=data.size()-1;
    int parent=(child-1)/2;
    /*
    We need to protect heap order property. If something is wrong, swap values.
     */
    while(child>0 && parent>=0 && data.get(parent).compareTo(data.get(child))>0) {
        swap(parent,child);
        child=parent;
        parent=(child-1)/2;
    }
    return true;
}
```

I've implemented the offer method like that:

I added elements the at the end of arraylist. Then I determined children as 2*x+1 and 2*x+2

Then I check whether heap order property is protected or not. If not, then I swap the values.

```
public E poll() {
    if(isEmpty()) {
        return null;
    }
    E result=data.get(0);
    if(data.size()==1) {
        data.remove( index: 0);
        return result;
    }
    /*
    After removal, heap order property should be protected.
     */
    data.set(0,data.remove( index: data.size()-1));
    int parent=0;
    while(true) {
        int leftchild=2*parent+1;
        if(leftchild>=data.size()) {
            break;
        }
        int rightchild=leftchild+1;
        int minchild=leftchild;
        if(rightchild<data.size() && compare(data.get(leftchild),data.get(rightchild))>0) {
            minchild=rightchild;
        }
        if(compare(data.get(parent),data.get(minchild))>0) {
            swap(parent,minchild);
            parent=minchild;
        }
        else {
            break;
        }
    }
    return result;
}
```

In poll method, I just take the root of the heap. Then I fix the heap order property issue.

```
    E[] temp=(E[])Array.newInstance(data.get(0).getClass(),size());
    int i=0;
    while(!value.equals(data.get(0))) {
        temp[i]=poll();
        i++;
    }
    poll();

    if(data.size()==0) {
        data=null;
        data=new ArrayList<>();
    }
    else {
        ArrayList<E> arr = new ArrayList<>(data);
        data=null;
        data=new ArrayList<>();
        for (E e : arr) {
            offer(e);
        }
    }
    for (E e : temp) {
        offer(e);
    }
    return true;
}
```

At remove ith index method, I removed all element of the heap until needed element is reached. Then I add other elements again.

5. TEST CASES

1)

```
[ 5 50 10 60 55 85]
```

```
Enter element to search
100
Heap does not contain this element
```

```
Enter element to search
10
Heap contains this element
```

2)

```
Enter elements of the new Heap, to end entering enter -1
70
80
100
7
-1
Before merging, first heap is :
[ 5 50 10 60 55 85]

Before merging other heap is :
[ 7 70 100 80]
After merging:
[ 5 50 7 60 55 85 10 70 100 80]
```

3)

```
Enter an index -starts with 1- to remove ith largest element
2
Before removing:
[ 5 50 10 60 55 85]
After removing:
[ 5 50 10 85 55]
1)SEARCH FOR AN ELEMENT
```

4)

```
4
To set a value, enter an index to make iterator next
4
Enter a new value to set
70
Before setting:
[ 5 50 10 60 55 85]
After setting:
[ 5 50 10 60 70 85]
1)SEARCH FOR AN ELEMENT
```

6. RUNNING COMMAND AND RESULTS

```
Enter element to search
10
Heap contains this element
```

```
Enter elements of the new Heap, to end entering enter -1
6
7
8
-1
Before merging, first heap is :
[ 5 50 10 60 55 85]

Before merging other heap is :
[ 6 7 8]
After merging:
[ 5 7 6 8 55 85 10 60 50]
```

```
Enter an index -starts with 1- to remove ith largest element
1
Before removing:
[ 5 7 6 8 55 85 10 60 50]
After removing:
[ 5 6 7 8 10 50 55 60]
1)SEARCH FOR AN ELEMENT
```

```
4
To set a value, enter an index to make iterator next
3
Enter a new value to set
11
Before setting:
[ 5 6 7 8 10 50 55 60]
After setting:
[ 5 6 7 11 10 50 55 60]
1)SEARCH FOR AN ELEMENT
2)MERGE WITH ANOTHER HEAP
3)REMOVE THE ith LARGEST ELEMENT
4)SET A VALUE
0)EXIT
Enter :
0
EXITING...
```

7. TIME COMPLEXITIES

```java
public boolean offer(E item) {
    if(item==null) {
        return false;
    }
    data.add(item);
    int child=data.size()-1;
    int parent=(child-1)/2;
    /*
    We need to protect heap order property. If something is wrong, swap values.
     */
    while(child>0 && parent>=0 && data.get(parent).compareTo(data.get(child))>0) {
        swap(parent,child);
        child=parent;
        parent=(child-1)/2;
    }
    return true;
}
```

Offer method:

ArrayList add method -> Θ(1) (Amortized)

ArrayList size method -> Θ(1)

While loop -> ArrayList get method-> Θ(1)

Swap method-> Θ(1)

While loop executed logn time. Time complexity is

O(logn)

```java
public E poll() {
    if(isEmpty()) {
        return null;
    }
    E result=data.get(0);
    if(data.size()==1) {
        data.remove( index: 0);
        return result;
    }
    /*
    After removal, heap order property should be protected.
     */
    data.set(0,data.remove( index: data.size()-1));
    int parent=0;
    while(true) {
        int leftchild=2*parent+1;
        if(leftchild>=data.size()) {
            break;
        }
        int rightchild=leftchild+1;
        int minchild=leftchild;
        if(rightchild<data.size() && compare(data.get(leftchild),data.get(rightchild))>0) {
            minchild=rightchild;
        }
        if(compare(data.get(parent),data.get(minchild))>0) {
            swap(parent,minchild);
            parent=minchild;
        }
        else {
            break;
        }
    }
    return result;
}
```

Poll method:

isEmpty method -> Θ(1)

ArrayList get -> Θ(1)

ArrayList size -> Θ(1)

ArrayList remove -> O(n) (worst case)

ArrayList set -> Θ(1)

While loop ->

Swap method -> Θ(1)

While loop will be executed worst case logn times.

Tw= theta(logn)

Tb=theta(1)

T(n) = O(logn)

```java
private int compare(E left,E right) {
    return (left).compareTo(right);
}
```

Compare method:

In this case I used Integer wrapper class' compareTo method, It is **Θ(1)**

```java
public int size() {
    return data.size();
}
```

```java
private void swap(int a,int b) {
    E temp=data.get(a);
    data.set(a,data.get(b));
    data.set(b,temp);
}
```

Swap method **Θ(1)**

Size method **Θ(1)**

Iterator method **Θ(1)**

Peek method **Θ(1)**

Find method **O(n)**

```java
public THeapIterator iterator() {
    return new THeapIterator();
}
```

```java
*/
public E peek() {
    return data.get(0);
}
```

```java
public E find(E item) {
    for (E val : data) {
        if (val.equals(item)) {
            return val;
        }
    }
    return null;
}
```

Merge method

Let's say other heap has m elements.

For loop will be executed m times.

Offer -> **O(logn)**

T(n,m)=O(m*logn)

```java
public void merge(THeap<E> other) {
    /*
     * Merge two heaps
     */
    for(E val:other.data) {
        offer(val);
    }
}
```

```java
public boolean remove_ith(int index) throws IndexOutOfBoundsException,NullPointerException{

    /*
    If index is less than 1 or greater than size
     */
    if(index<1 || index>=size()) {
        throw new IndexOutOfBoundsException();
    }
    else if(data == null || data.size()==0) {
        throw new NullPointerException();
    }
    /*
     * find the value
     */
    E value=find_ith(index);
    /*
    If it is root
     */
    if(value.equals(data.get(0))) {
        poll();
        return true;
    }
    /*
     * Until it is find in the heap, remove all elements,then add them again without wanted value
     */
    E[] temp=(E[])Array.newInstance(data.get(0).getClass(),size());
    int i=0;
    while(!value.equals(data.get(0))) {
        temp[i]=poll();
        i++;
    }
    poll();
```

Remove ith method

İf else if statement -> constant

Find ith method -> **O(nlogn)**

Equals method -> constant (Integer assumed)

Poll -> **O(logn)** (Worst case)

Creation of array -> constant

While loop -> **O(n)**

Poll -> -> O(logn)

Continue at next page …

```
    if(data.size()==0) {
        data=null;
        data=new ArrayList<>();
    }
    else {
        ArrayList<E> arr = new ArrayList<>(data);
        data=null;
        data=new ArrayList<>();
        for (E e : arr) {
            offer(e);
        }
    }
    for (E e : temp) {
        offer(e);
    }
    return true;
}
```

Remove ith continued.

Size method -> constant

For loop will be executed number of remained elements: **O(n)**

Other For loop will be executed again number of remained elements: **O(n)**

T(n) = O(nlogn)

```
@SuppressWarnings("unchecked")
private E find_ith(int index) {
    /*
     * Sort the arraylist,return value
     */
    ArrayList<E> hold=(ArrayList<E>) data.clone();
    hold.sort(Comparable::compareTo);
    return hold.get(size()-index);
}
```

Private find ith method

ArrayList clone -> **Θ(n)**

Sort method -> O(nlogn)

Get method -> constant

T(n) = O(nlogn)

```
@Override
public boolean hasNext() {
    return cursor!=data.size();
}
```

Iterator's has next method :**Θ(1)**

Iterator's next method: **Θ(1)**

```
@Override
public E next() {
    if(hasNext()) {
        lastRet=cursor;
        return data.get(cursor++);
    }
    return null;
}
```

```
public E set(E item) throws IllegalArgumentException{
    if(cursor==0 || lastRet<0) {
        throw new IllegalArgumentException();
    }
    if(item==null) {
        return null;
    }
    data.set(lastRet,item);
    check_for_heap_order(lastRet);
    return THeap.this.data.get(cursor);
}
```

Iterator's set method:

İf statements -> constant time

Set method -> constant time

Check for heap order method -> **O(logn)**

T(n) = **O(logn)**

```java
public String toString() {
    StringBuilder str=new StringBuilder();
    str.append("[");
    for(E val:data) {
        str.append(" ");
        str.append(val);
    }
    str.append("]");
    return str.toString();
}
```

toString method:

It is **Θ(n)**

```java
private void check_for_heap_order(int index) {
    /*
     * Check parent
     */
    if((index-1)/2>=0) {
        if(data.get((index-1)/2).compareTo(data.get(index))>0) {
            swap( a: (index-1)/2,index);
            check_for_heap_order(index);
        }
    }
    /*
     * Check child -> leftchild -> 2*i+1 rightchild -> 2*i+2
     */
    if(2*index+1>=size()) {
        if(2*index+2<size()) {
            if(data.get(index).compareTo(data.get(2*index+2))>0) {
                swap(index, b: 2*index+2);
                check_for_heap_order(index);
            }
        }
    }
    else {
        if(2*index+2>=size()) {
            if(data.get(index).compareTo(data.get(2*index+1))>0) {
                swap(index, b: 2*index+1);
                check_for_heap_order(index);
            }
        }
        else {
            if(data.get(index).compareTo(data.get(2*index+1))>0) {
                swap(index, b: 2*index+1);
                check_for_heap_order(index);
            }
            if(data.get(index).compareTo(data.get(2*index+2))>0) {
                swap(index, b: 2*index+2);
                check_for_heap_order(index);
            }
        }
    }
}
```

Recursive check for heap order method

O(logn)

PART 2

1.PROBLEM DEFINITON

Elements are need to be stored in Binary Search Heap Tree. Each node of the Binary Search Tree contains a Max Heap. Binary Search Tree's order depends on the roots of the heap. Max heap is the heap that root value is the maximum value of the heap. Each heap should have maximum 7 elements. Each node of the heap have Value class that holds the value and the occurrency of the value as an integer.

2.SYSTEM REQUIREMENTS

We need MaxHeap class and BSTHeap class to fix the problem.

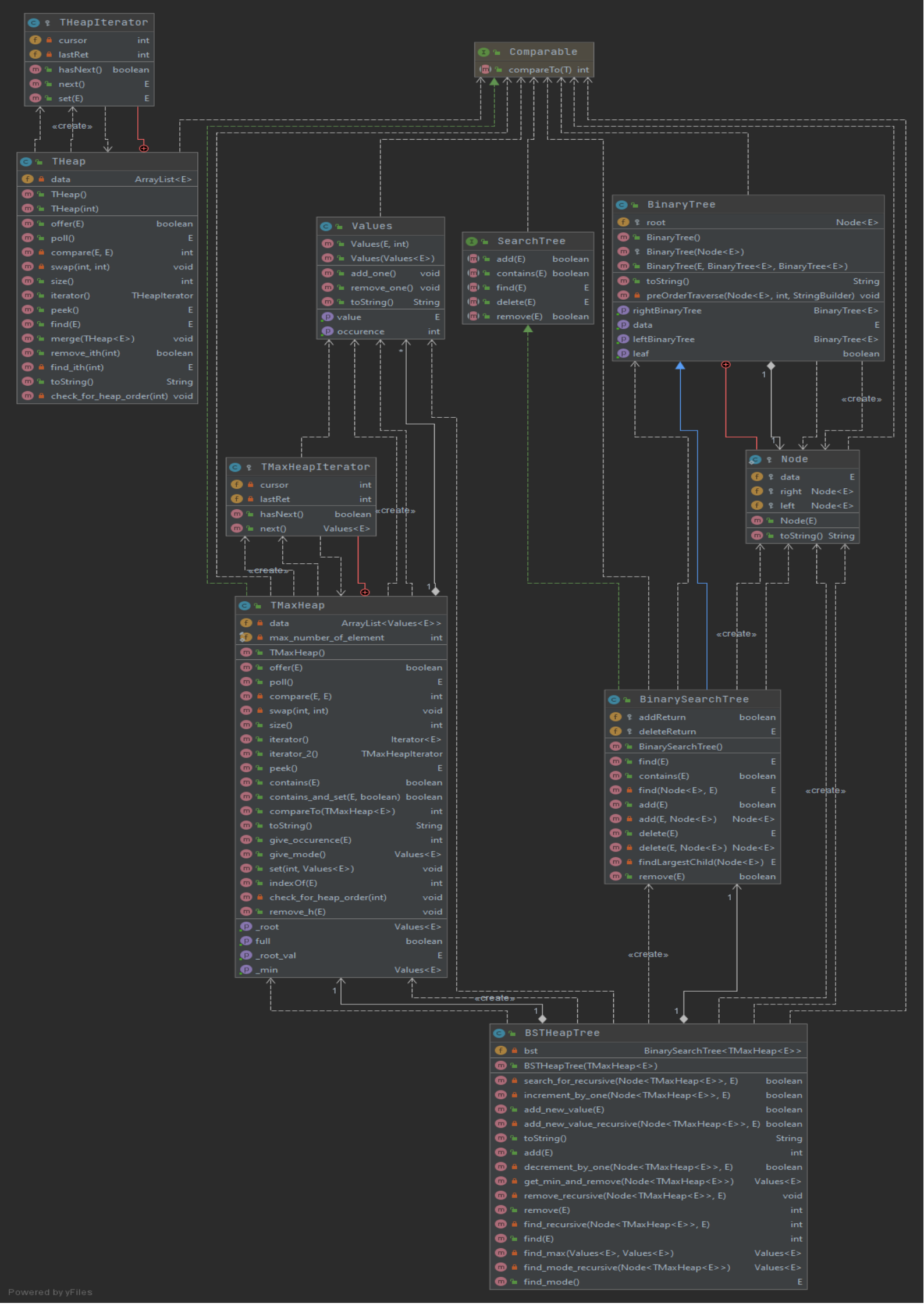
MaxHeap class have arraylist to hold data. Working principle is nearly same with Heap structure. We need to change < to >. BSTHeap class is a Binary Search Tree class which contains a MaxHeap as a component. Binary Search Tree is a Binary Tree class also.

```
TMaxHeap<Integer> tMaxHeap=new TMaxHeap<>();
tMaxHeap.offer( e: 37);
tMaxHeap.offer( e: 23);
tMaxHeap.offer( e: 10);
System.out.println(tMaxHeap);


BSTHeapTree<Integer> bst_heap=new BSTHeapTree<>(tMaxHeap);
```

3.CLASS DIAGRAM

Class diagram is at the next page. It couldn't fit this page.


THeapIterator
cursor int
lastRet int
hasNext() boolean
next() E
set(E) E
«create»
Comparable
compareTo(T) int
THeap
data ArrayList<E>
THeap()
THeap(int)
offer(E) boolean
poll() E
compare(E, E) int
swap(int, int) void
size() int
iterator() THeapIterator
peek() E
find(E) E
merge(THeap<E>) void
remove_ith(int) boolean
find_ith(int) E
toString() String
check_for_heap_order(int) void
Values
Values(E, int)
Values(Values<E>)
add_one() void
remove_one() void
toString() String
value E
occurence int
SearchTree
add(E) boolean
contains(E) boolean
find(E) E
delete(E) E
remove(E) boolean
BinaryTree
root Node<E>
BinaryTree()
BinaryTree(Node<E>)
BinaryTree(E, BinaryTree<E>, BinaryTree<E>)
toString() String
preOrderTraverse(Node<E>, int, StringBuilder) void
rightBinaryTree BinaryTree<E>
data E
leftBinaryTree BinaryTree<E>
leaf boolean
Node
data E
right Node<E>
left Node<E>
Node(E)
toString() String
TMaxHeapIterator
cursor int
lastRet int
hasNext() boolean
next() Values<E>
TMaxHeap
data ArrayList<Values<E>>
max_number_of_element int
TMaxHeap()
offer(E) boolean
poll() E
compare(E, E) int
swap(int, int) void
size() int
iterator() Iterator<E>
iterator_2() TMaxHeapIterator
peek() E
contains(E) boolean
contains_and_set(E, boolean) boolean
compareTo(TMaxHeap<E>) int
toString() String
give_occurence(E) int
give_mode() Values<E>
set(int, Values<E>) void
indexOf(E) int
check_for_heap_order(int) void
remove_h(E) void
_root Values<E>
full boolean
_root_val E
_min Values<E>
BinarySearchTree
addReturn boolean
deleteReturn E
BinarySearchTree()
find(E) E
contains(E) boolean
find(Node<E>, E) E
add(E) boolean
add(E, Node<E>) Node<E>
delete(E) E
delete(E, Node<E>) Node<E>
findLargestChild(Node<E>) E
remove(E) boolean
BSTHeapTree
bst BinarySearchTree<TMaxHeap<E>>
BSTHeapTree(TMaxHeap<E>)
search_for_recursive(Node<TMaxHeap<E>>, E) boolean
increment_by_one(Node<TMaxHeap<E>>, E) boolean
add_new_value(E) boolean
add_new_value_recursive(Node<TMaxHeap<E>>, E) boolean
toString() String
add(E) int
decrement_by_one(Node<TMaxHeap<E>>, E) boolean
get_min_and_remove(Node<TMaxHeap<E>>) Values<E>
remove_recursive(Node<TMaxHeap<E>>, E) void
remove(E) int
find_recursive(Node<TMaxHeap<E>>, E) int
find(E) int
find_max(Values<E>, Values<E>) Values<E>
find_mode_recursive(Node<TMaxHeap<E>>) Values<E>
find_mode() E
Powered by yFiles

4.PROBLEM SOLUTION APPROACH

- I solved keeping occurence problem with using Value class inside the ArrayList. In Value class there are item and the occurence of the item.
- In TMaxHeap, I used ArrayList to hold the values and I always tried to keep heap order structure.
- In BSTHeapTree , I always look for the current root and then left and then right.

5.TEST CASES

I used the given tests and when I test the program it passed.

- I inserted 3000 numbers that are randomly generated in the range 0-500. I also stored them in the ArrayList in the main. I compared the occurencesin test2_1 function and it passed, I went to the test2_2
- In test2_2 I searched for 100 numbers that are in the array, I searched 10 numbers that are not in the array, I compared the occurences and test is passed. I went to the test2_3
- In test2_3 I find the mode of the bstheap and arraylists There were more than 1 mode. But I found the right mode. Then I went to the test2_4
- In test2_4 I removed 100 elements from the arraylist and bstheap. I compared the occurrences of them and test is passed.

```
Successful,going to test test2_2
TEST2_2 IS SUCCESSFUL GOING TO THE TEST2_3
TEST2_3 IS SUCCESSFUL, GOING TO THE TEST2_4
ALL TESTS ARE PASSED
```

6.RUNNING COMMAND AND RESULTS

7.TIME COMPLEXITIES

1. MaxHeap class

a) Constructor **Θ(1)**

```
public TMaxHeap() { data=new ArrayList<>(); }
```

b) Offer

ArrayList add -> amortized constant time

While loop -> Tb= **Θ(1)**

Tw -> **O(logn)**

T(n) = O(logn)

```
@Override
public boolean offer(E e) {
    /*
    If this heap is full, return false
    If item is null,return false
    Otherwise add the item
    Check the heap order
     */
    if(isFull()) {
        return false;
    }
    else {
        if(e==null) {
            return false;
        }
        data.add(new Values<>(e, occur: 1));
        int child=data.size()-1;
        int parent=(child-1)/2;
    /*
    We need to protect heap order property. If something is wrong, swap values.
     */
        while(child>0 && parent>=0 && data.get(parent).getValue().compareTo(data.get(child).getValue())<0) {
            swap(parent,child);
            child=parent;
            parent=(child-1)/2;
        }

        return true;
    }
}
```

c) Poll method **Θ(1)**

```
@Override
public E poll() { return null; }
```

d) Compare method **Θ(1)**

```
private int compare(E left,E right) { return (left).compareTo(right); }
```

```
private void swap(int a,int b) {
    Values<E> temp=data.get(a);
    data.set(a,data.get(b));
    data.set(b,temp);
}
```

e) Swap method **Θ(1)**

f) Size method **Θ(1)**

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

g) İterator_2 method

Θ(1)

```
public TMaxHeapIterator iterator_2() { return new TMaxHeapIterator(); }
```

```
public boolean hasNext() { return cursor!=data.size(); }
```

h) TMaxHeapIterator class -> hasNext **Θ(1)**

i) TMaxHeapIterator -> next **Θ(1)**

```
public Values<E> next() {
    if(hasNext()) {
        lastRet=cursor;
        return data.get(cursor++);
    }
    return null;
}
```

j) Peek method **Θ(1)**

```
public E peek() { return data.get(0).getValue(); }
```

k) Contains method **O(n)**

```
public boolean contains(E item) {
    /*
    If item is null,return false
    If heap is empty, return false
    If arraylist is empty, return false
    Otherwise find it
     */
    if(item==null) {
        return false;
    }
    else if(isEmpty()) {
        return false;
    }
    else if(data==null) {
        return false;
    }
    for (Values<E> val : data) {
        if(val==null) {

        }
        else if (val.getValue().equals(item)) {
            return true;
        }
    }
    return false;
}
```

l) Contains and set method **O(n)**

m) Give occurence method **O(n)**

```
public int give_occurence(E val) {
    /*
    If item is null,return 0
    If heap is empty,return null
    If arraylist is empty, return null
    Otherwise find item and return occurence
     */
    if(val==null) {
        return 0;
    }
    else if(isEmpty()) {
        return 0;
    }
    else if(data==null) {
        return 0;
    }
    int counter=0;
    for (Values<E> temp:data) {
        if (temp==null) {

        }
        else if(temp.getValue().equals(val)) {
            counter=temp.getOccurence();
            break;
        }
    }
    return counter;
}
```

```
public boolean contains_and_set(E item,boolean sit) {
    TMaxHeapIterator it=iterator_2();
    /*
    If item is nul return null
    If heap is empty, return null
    If arraylist is null,return null
    Otherwise find item and increment-decrement
     */
    if(item==null) {
        return false;
    }
    else if(isEmpty()) {
        return false;
    }
    else if(data==null) {
        return false;
    }
    while(it.hasNext()) {
        Values<E> x=it.next();
        if(x==null) {

        }
        else if(x.getValue().equals(item)) {
            if(sit) {
                x.add_one();
            }
            else {
                x.remove_one();
            }
            return true;
        }
    }
    return false;
}
```

n) İsfull method **Θ(1)**

```
public boolean isFull() { return size()==max_number_of_element; }
```

o) Get rool val method

Θ(1)

```
public E get_root_val() { return data.get(0).getValue(); }
```

p) Give mode method

Θ(n)

```
public Values<E> give_mode() {
    /*
    Find the item that have the highest occurrence
     */
    int index=0;
    int max=-1;
    int i=0;

    if(isEmpty()) {
        return null;
    }
    for(Values<E> values:data) {
        if(values.getOccurence()>max) {
            max=values.getOccurence();
            index=i;
        }
        i++;
    }
    return data.get(index);
}
```

q) Get root method

Θ(1)

```
public Values<E> get_root() {
    if(isEmpty()) {
        return null;
    }
    return data.get(0);
}
```

```java
public void set(int index,Values<E> val) throws IndexOutOfBoundsException{
    if(index<0 || index>=size()) {
        throw new IndexOutOfBoundsException();
    }
    data.set(index,val);
    check_for_heap_order( index: 0);
}
```

r) Set method

Θ(1) + check for heap order ->

T(n) =

```java
public int indexOf(E item) throws NullPointerException {
    if(item==null) {
        throw new NullPointerException();
    }
    else if(isEmpty()) {
        throw new NullPointerException();
    }
    int i=0;
    for (Values<E> val:data) {
        if(val.getValue().equals(item)) {
            break;
        }
        i++;
    }
    return i;
}
```

s) indexOf method **O(n)**

t) get min method **Θ(n)**

```java
public Values<E> get_min() {

    if(isEmpty()) {
        return null;
    }
    Values<E> temp=new Values<E>(data.get(0).getValue(), occur: 0);
    for (Values<E> val:data) {
        if(val.getValue().compareTo(temp.getValue())<0) {
            temp=val;
        }
    }
    return temp;
}
```

u) Check for heap order method **O(logn)**

```java
private void check_for_heap_order(int index) {
    /*
     * Check parent
     */
    if(isEmpty()) {
        return;
    }
    else if(data==null) {
        return;
    }
    if((index-1)/2>=0) {
        if(data.get((index-1)/2).getValue().compareTo(data.get(index).getValue())<0) {
            swap( a: (index-1)/2,index);
            check_for_heap_order(index);
        }
    }
    /*
     * Check child -> leftchild -> 2*i+1 rightchild -> 2*i+2
     */
    else if(2*index+1>=size()) {
        if(2*index+2<size()) {
            if(data.get(index).getValue().compareTo(data.get(2*index+2).getValue())<0) {
                swap(index, b: 2*index+2);
                check_for_heap_order(index);
            }
        }
    }
    else {
        if(2*index+2>=size()) {
            if(data.get(index).getValue().compareTo(data.get(2*index+1).getValue())<0) {
                swap(index, b: 2*index+1);
                check_for_heap_order(index);
            }
        }
        else {
            if(data.get(index).getValue().compareTo(data.get(2*index+1).getValue())<0) {
                swap(index, b: 2*index+1);
                check_for_heap_order(index);
            }
            if(data.get(index).getValue().compareTo(data.get(2*index+2).getValue())<0) {
                swap(index, b: 2*index+2);
                check_for_heap_order(index);
            }
        }
    }
```

v) Remove_h method : arraylist-> remove O(n)

w) Check for heap order method : **O(logn)**

T(n) : **O(n)**

```java
public void remove_h(E item) {
    if(isEmpty()) {
        return;
    }
    data.remove(indexOf(item));
    check_for_heap_order( index: 0);
}
```

Binary Search Tree class' methods

Find method -> **O(logn)**

Contains method -> **O(logn)**

Find(recursive) method -> **O(logn)**

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

/**
 * Checks the item is in the tree
 * @param target item
 * @return if it contains, return true
 */
public boolean contains(E target) { return find(target)!=null; }

/**
 * Recursive find method
 * @param localRoot localroot that item is queried in
 * @param target item
 * @return item
 */
private E find(Node<E> localRoot,E target) {
    /*
    Look local root,left and right
     */
    if(localRoot==null) {
        return null;
    }
    int compResult=target.compareTo(localRoot.data);
    if(compResult==0) {
        return localRoot.data;
    }
    else if(compResult<0){
        return find(localRoot.left, target);
    }
    else {
        return find(localRoot.right, target);
    }
}
```

```java
public boolean add(E value) {
    root=add(value, root);
    return addReturn;
}
```

```java
private Node<E> add(E value,Node<E> node) {
    if(node==null) {
        addReturn=true;
        return new Node<E>(value);
    }
    else if(value.compareTo(node.data)==0) {
        addReturn=false;
        return node;
    }
    else if(value.compareTo(node.data)<0) {
        node.left=add(value, node.left);
        return node;
    }
    else {
        node.right=add(value,node.right);
        return node;
    }
}
//O(logn)-O(n)
```

add method -> **O(logn)**

add(recursive) method-> **O(logn)**

```java
private Node<E> delete(E item,Node<E> node) {
    if(node==null) {
        deleteReturn=null;
        return node;
    }
    int compResult=item.compareTo(node.data);
    if(compResult<0) {
        node.left=delete(item, node.left);
        return node;
    }
    else if(compResult>0) {
        node.right=delete(item, node.right);
        return node;
    }
    else {
        deleteReturn=node.data;
        if(node.left==null) {
            return node.right;
        }
        else if(node.right==null) {
            return node.left;
        }
        else {
            if(node.left.right==null) {
                node.data=node.left.data;
                node.left=node.left.left;
                return node;
            }
            else {
                node.data=findLargestChild(node.left);
                return node;
            }
        }

    }
}
```

delete method -> **O(logn)**

```java
private E findLargestChild(Node<E> parent) {
    if(parent.right.right==null) {
        E returnValue=parent.right.data;
        parent.right=parent.right.left;
        return returnValue;
    }
    else {
        return findLargestChild(parent.right);
    }
}
```

Find largest child method -> **O(logn)**

BSTHeapTree class' methods:

```java
*/
private boolean search_for_recursive(BinaryTree.Node<TMaxHeap<E>> cur_root,E val) {
    /*
    If current root is null,return false Otherwise look for item , if current
    root does not contain it look left and right
     */
    if(cur_root==null) {
        return false;
    }
    if(cur_root.data.contains(val)) {
        return true;
    }
    return search_for_recursive(cur_root.left,val) || search_for_recursive(cur_root.right,val);
}
```

Search for recursive method -> **O(logn)**

ArrayList contains-> constant

T(n)=T(n-1)+T(n-1) + constant

T(n)=O(logn)

```java
private boolean increment_by_one(BinaryTree.Node<TMaxHeap<E>> cur_root, E val) {
    /*
    If current root is null,return false.
    If item is at the current root, add (true) and return true
    Otherwise look left and right
     */
    if(cur_root==null) {
        return false;
    }
    if(cur_root.data.contains_and_set(val,true)) {
        return true;
    }
    return increment_by_one(cur_root.left,val) || increment_by_one(cur_root.right,val);
}
```

İncrement by one method -> **O(logn)**

Contains and set-> O(n)

T(n)=T(n-1)+T(n-1)+O(n)

T(n)=O(logn)

```java
*/
private boolean add_new_value_recursive(BinaryTree.Node<TMaxHeap<E>> cur_root, E val) {
    /*
    If current root is null, create a heap.Add item to this heap
    If current root is full and right is null, create right heap
    If current root is full and left is null, create left heap
    Compare the value of current root and the item, decide where to put the item
    If there current root has a space, add this item to the current heap.
     */
    if(cur_root==null) {
        cur_root=new BinaryTree.Node<>(new TMaxHeap<>());
        return cur_root.data.offer(val);
    }
    else if(cur_root.data.isFull()) {
        if(cur_root.right==null) {
            cur_root.right=new BinaryTree.Node<>(new TMaxHeap<>());
        }
        if(cur_root.left==null) {
            cur_root.left=new BinaryTree.Node<>(new TMaxHeap<>());
        }
        if(cur_root.data.get_root_val().compareTo(val)<0) {
            return add_new_value_recursive(cur_root.right,val);
        }
        else if(cur_root.data.get_root_val().compareTo(val)>0) {
            return add_new_value_recursive(cur_root.left,val);
        }
        return false;
    }
    else {
        return cur_root.data.offer(val);
    }
}
```

Add new value recursive method -> **O(nlogn)**

T(n)=T(n-1)+O(logn)

T(n)=O(nlogn)

```java
private boolean decrement_by_one(BinaryTree.Node<TMaxHeap<E>> cur_root, E val) {
    /*
    If current root is null, return false
    If current root's heap is null, return false
    If current root contains the item,return true
    Otherwise look left and right
     */
    if(cur_root==null) {
        return false;
    }
    else if(cur_root.data==null) {
        return false;
    }
    else if(cur_root.data.contains_and_set(val, sit: false)) {
        return true;
    }
    return decrement_by_one(cur_root.left,val) || decrement_by_one(cur_root.right,val);
}
```

decrement by one method -> **O(logn)**

Contains and set-> O(n)

T(n)=T(n-1)+T(n-1)+O(n)

T(n)=O(logn)

```java
private Values<E> get_min_and_remove(BinaryTree.Node<TMaxHeap<E>> cur_root) {
    /*
    If current root is null,return null
    If current root's left is not null,go to the left
    If current heap is null, return null
    Otherwise find the minimum of the heap.
    If you couldn't find,return null
    Otherwise remove this item from the current place, and return this item.
     */
    if(cur_root==null) {
        return null;
    }
    if(cur_root.left!=null) {
        return get_min_and_remove(cur_root.left);
    }
    if(cur_root.data==null) {
        return null;
    }
    Values<E> ret=cur_root.data.get_min();
    if(ret==null) {
        return null;
    }
    remove_recursive(cur_root,ret.getValue());
    return ret;
}
```

Get min and remove method -> **O(log ^{2} n)**

Get min -> **Θ(n)**

```java
public int remove(E item) throws NullPointerException{
    /*
    If item is null,throw exception
    Find occurrence of the item.
    If it is 1,remove item
    Otherwise decrement it is occurrence
     */
    if(item==null) {
        throw new NullPointerException();
    }
    int occur=find(item);
    if(occur==1) {
        remove_recursive(bst.root,item);
    }
    else if(occur>1) {
        decrement_by_one(bst.root,item);
    }
    return occur;
}

/**
```

remove method -> **O(log ^{2} n)**

```
 */
if(cur_root==null) {

}
else if(cur_root.data==null) {

}
//if we find the item
else if(cur_root.data.contains(item)) {
    //if maxheap is not full
    if(cur_root.left==null && cur_root.right==null) {
        cur_root.data.remove_h(item);
        if(cur_root.data.isEmpty()) {
            cur_root.data=null;
            cur_root=null;
        }
    }
    //if left is empty,right has element
    else if(cur_root.left==null) {
        Values<E> temp=get_min_and_remove(cur_root.right);
        try {
            cur_root.data.set(cur_root.data.indexOf(item),temp);
        }
        catch (NullPointerException ne) {
            System.out.println(ne);
        }

    }
    //if left and right has element
    else {
        if(cur_root.left.data==null) {
            Values<E> temp=get_min_and_remove(cur_root.right);
            try {
                cur_root.data.set(cur_root.data.indexOf(item),temp);
            }
            catch (NullPointerException ne) {
                System.out.println(ne);
            }
        }
        else {
            Values<E> temp2=cur_root.left.data.get_root();
            if(temp2==null) {
                if(cur_root.right==null || cur_root.right.data==null) {
      else {
          Values<E> temp2=cur_root.left.data.get_root();
          if(temp2==null) {
              if(cur_root.right==null || cur_root.right.data==null) {

              }
              else {
                  Values<E> temp=get_min_and_remove(cur_root.right);
                  try {
                      cur_root.data.set(cur_root.data.indexOf(item),temp);
                  }
                  catch (NullPointerException ne) {
                      System.out.println(ne);
                  }
              }
          }
          else {
              cur_root.left.data.remove_h(temp2.getValue());
              try {
                  cur_root.data.set(cur_root.data.indexOf(item),temp2);
              }
              catch (NullPointerException ne) {
                  System.out.println(ne);
              }
          }
      }
    }
}
//if we couldn't find the element
else {
    remove_recursive(cur_root.left,item);
    remove_recursive(cur_root.right,item);
}
```

Remove recursive method -> **O(log ^{2} n)**

```java
*/
private int find_recursive(BinaryTree.Node<TMaxHeap<E>> cur_root,E val) {
    /*
    If heap node is null,return null
    If inside of the heap is null, return null
    If cur root is empty, returnu 0, it doesn't contains it.
    Otherwise look this heap and right and left
     */
    if(cur_root==null) {
        return 0;
    }
    else if(cur_root.data==null) {
        return 0;
    }
    else if(cur_root.data.isEmpty()) {
        return 0;
    }
    return cur_root.data.give_occurence(val) + find_recursive(cur_root.left,val) + find_recursive(cur_root.right,val);
}
```

find recursive method -> **O(logn)**

```java
*/
private Values<E> find_max(Values<E> val1,Values<E> val2) {
    if(val1==null) {
        if(val2==null) {
            return null;
        }
        return val2;
    }
    else if(val2==null) {
        return val1;
    }
    else {
        if(val1.getOccurence()>val2.getOccurence()) {
            return val1;
        }
        return val2;
    }
}
*/
```

find max method -> **constant time**

```java
private Values<E> find_mode_recursive(BinaryTree.Node<TMaxHeap<E>> cur_root) {
    /*
    If current root is null,return null
    If current root data is null, return null
    Find mode of the heap.
    If it is null, return null
    Otherwise look to the left and right,compare with them and return the max of them.
     */
    if(cur_root==null) {
        return null;
    }
    else if(cur_root.data==null) {
        return null;
    }
    Values<E> val=cur_root.data.give_mode();
    if(val==null) {
        return null;
    }
    else {
        if(cur_root.left==null) {
            if(cur_root.right==null) {
                return val;
            }
            else {
                Values<E> val3=find_mode_recursive(cur_root.right);
                return find_max(val,val3);
            }
        }
        else {
            Values<E> val2=find_mode_recursive(cur_root.left);
            if(cur_root.right==null) {
                return find_max(val,val2);
            }
            else {
                Values<E> val3=find_mode_recursive(cur_root.right);
                return find_max(val,find_max(val2,val3));
            }
        }
    }
```

find mode recursive method -> **O(logn)**