GIT Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework #4 Report

Erdi Bayır
171044063

SYSTEM REQUIREMENTS

Functional Requirements

- **Heap Class;**

  Must add element.

  Must remove element.

  Must remove i’th largest element.

  Must set next value of Heap.

  Must Search An Element.

  Must merge with another Heap.

- **Binary Searh Tree Class;**

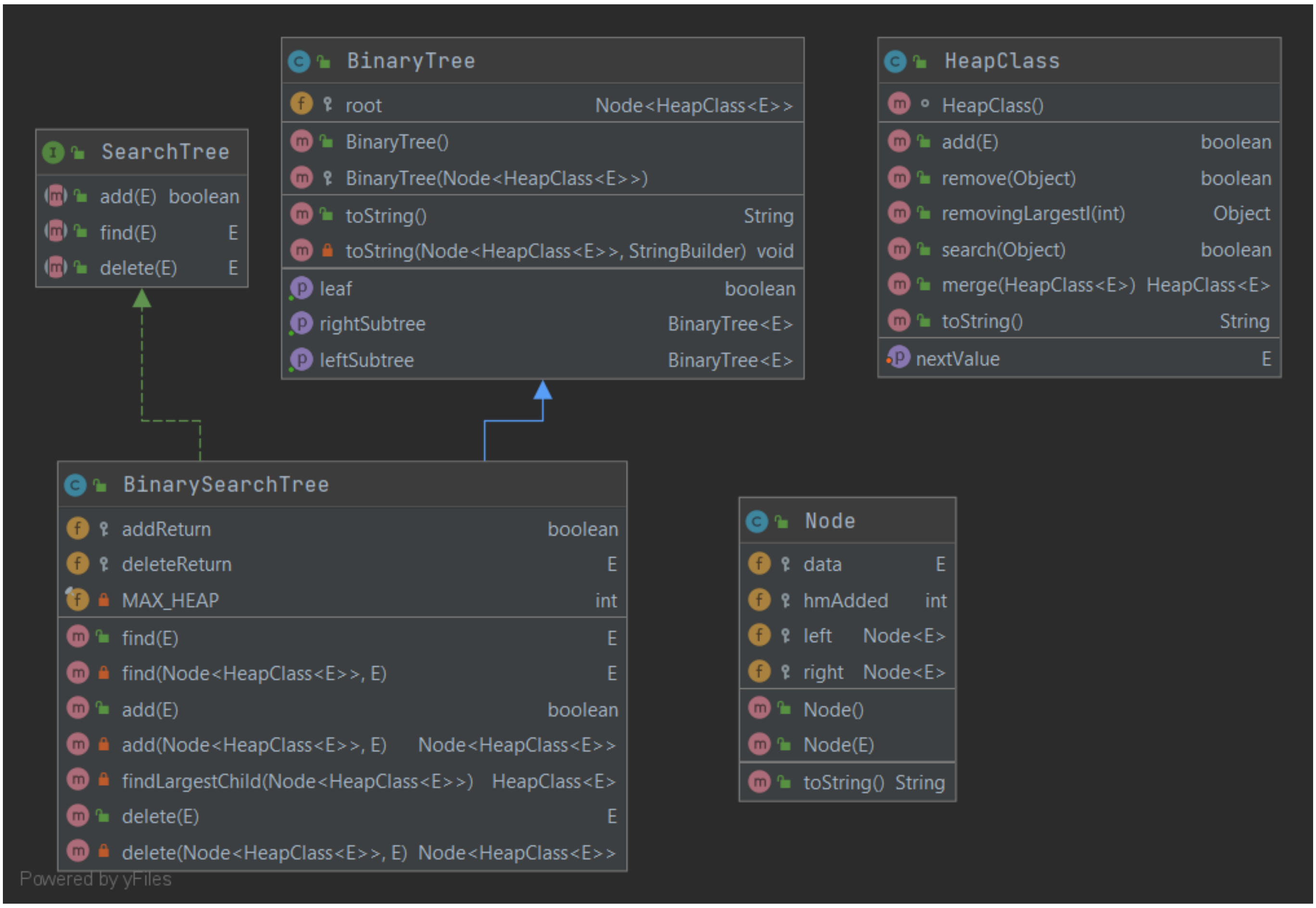
  Must add element.

  Must remove element.

  Must find mode of Binary Search Tree.

  Must Search An Element.

CLASS DIAGRAM



PROBLEM SOLUTION APPROACH

Note: I could not do the occurrence of the item part in the second question, other features work.

For Part 1:

What is required for this part is to write a Heap Class and implement the extra methods given. Since the Priority Queue class in Java is similar to the Heap class I want to create, I created the Heap class by extending this class and using the methods in it. I also wrote the properties using this class's methods.

For Part 2:

For this part, we will implement a Binary Search Tree in each Node with Heaps similar to the one we implemented in the first part. I did it using the implementation in the book. While the Node Class is a class that holds a variable data, Binary Tree is a class that holds Heap objects from this Node Class. And I implemented the desired methods in the Binary Search Tree class.

TEST CASES

For Part 1:

1) Create Heap
2) Add Element To Heap
3) Remove Element From Heap
4) Search For An Element
5) Merge With Another Heap
6) Removing I'th largest element from the Heap
7) Set next value with Iterator

For Part 1:

1) Create Binary Search Tree
2) Add Element Binary Search Tree
3) Remove Element From Binary Search Tree
4) Search For An Element

DRIVER TEST RESULTS

```
--- All Prints during the tests are Pre Order Traverse Form----
PART 1

Heap object Created...
Added some variables ...
Heap 1:[1, 4, 3, 6, 12, 19, 7, 15, 56]
Heap 2:[4, 15, 12, 23, 19, 34, 66]
Set the next value with 23!
After setNextValue  Heap 1:[3, 4, 7, 6, 12, 19, 56, 15, 23]
Remove 4 from Heap 1 is true
After remove call Heap 1:[3, 6, 7, 15, 12, 19, 56, 23]
Remove 34 from Heap 1 is false
After remove call Heap 1:[3, 6, 7, 15, 12, 19, 56, 23]
Remove 4'th largest element in Heap 1!
After remove 4'th largest element Heap 1:[3, 6, 7, 23, 12, 19, 56]
Search 56 in Heap 1 = true
Search 2 in Heap 1 = false
Search 19 in Heap 1 = true
Merge Heap 1 with Heap 2
After merge operation Heap 1:[3, 4, 7, 6, 12, 19, 56, 23, 15, 34, 66]
```

```
PART 2
Binary Search Tree Object Created...
Added some variables to BST 1...
Each line represents a bst node (heap)
[5, 9, 6, 19, 12, 16, 8]
[2, 4, 3]
[6, 7, 9, 16, 14, 13, 11]
[5]
[8, 10, 9, 18, 13, 15, 16]
[6, 7]
[8, 13, 10, 15, 14, 16, 17]
[8, 9, 12, 16, 14, 19, 18]
[10, 13, 11, 16, 18, 14, 12]
[9]
[10, 13]

BST 2 after add some elements
[1, 6, 2, 9, 7, 8, 5]
[3, 5, 7, 8, 9]

Find 6 in BST = 6
Find 3 in BST = 3
Find 14 in BST = null
Deleted Some Elements
[1, 6, 2, 9]
[3, 5, 7, 8, 9]

Last Element In Node
[2]
[3, 5, 7, 8, 9]

Node Removed...
[3, 5, 7, 8, 9]
```

TIME COMPLEXITY ANALYSIS

Heap Class

Search Method

```
/**
 * Search for an element
 * @param o The element to search in Heap.
 * @return Return true if search operation is successfully done otherwise false.
 */
public boolean search(Object o) {
    return super.contains(o);
}
```

n = size of Heap

super.contains => T(n) = O(n)

Time Complexity of Search method = > T(n) = O(n)

Add Method

```java
/**
 * Add element to Heap
 * @param e The element to add to Heap.
 * @return Return true if add operation is successfully done otherwise false.
 */
@Override
public boolean add(E e) {
    if(!search(e))
        return super.add(e);
    return false;
}
```

search = > T(n) = O(n)

super.add = > T(n) = O(logn)

Time Complexity of Search method => T(n) = O(logn) + O(n) = O(n)

Remove Method

```java
/**
 * Remove element from Heap
 * @param o The element to remove from Heap.
 * @return Return true if remove operation is successfully done otherwise false.
 */
@Override
public boolean remove(Object o) {
    return super.remove(o);
}
```

super.remove = > **T(n)** = O(logn)

Time Complexity of Remove method = > O(logn)

Removing i’th largest element Method

```java
/**
 * Removing ith largest element from the Heap
 * @param i ith largest element index
 * @return Return removed element if remove operation is successfully done otherwise null.
 */
public Object removingLargestI(int i){
    Object[] temp =super.toArray();
    Arrays.sort(temp);
    Object deleted = temp[size()-i];
    if(size() >= i) {
        remove(temp[size() - i]);
        return  deleted;
    }
     return null;
}
```

super.toArray() => **T(n)** = O(n)
Arrays.sort = > **T(n)** = O(n*logn)
remove() = > **T(n)** = O(logn)
Time Complexity of removingLargestI ()= O(n) + O(logn) + O(n*logn) = O(n*logn)

Merge Method

```java
/**
 * Merge with another heap
 * @param h1 The heap object to merge with current Heap
 * @return Return Heap Object.
 */
public HeapClass<E> merge(HeapClass<E> h1){
    this.addAll(h1);
    return this;

}
```

addAll() => **T(n)** = O(n)

Time Complexity of merge() => **T(n)** = O(n)

setNextValue Method

```
/**
 * Set the value of the last element returned by the next methods
 * @param newData New data for set element in Heap
 */
public void setNextValue(E newData){
    if(super.iterator().hasNext()) {
        remove(super.iterator().next());
        add(newData);
    }
}
```

Iterator.hasNext() => **T(n)** = O(1)
remove() = > **T(n)** = O(logn)
add() = > **T(n)** = O(n)

Time Complexity of setNextValue => **T(n)** = O(1) + O(logn) + O(n) = O(n)

Binary Search Tree Class

Find Method

```java
/**
 * Find method for Binary Search Tree.
 @param target The Comparable object being sought
 @return The object, if found, otherwise null
 */
public E find(E target) { return find(root, target); }

/** Recursive find method.
 @param localRoot The local subtree's root
 @param target The object being sought
 @return The object, if found, otherwise null
 */
private E find(Node<HeapClass<E>> localRoot, E target) {
    if (localRoot == null)
        return null;
    boolean search = localRoot.data.search(target);
    assert localRoot.data.peek() != null;
    int compResult = target.compareTo(localRoot.data.peek());
    if(search)
        return target;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Qbest = >T(n) = O(1) (If localRoot is null)

Search= O(n)

compareTo = O(1)

peek() = O(1)

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

Add Method

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

```java
/**
 * Recursive add method.
 * @param localRoot The local subtree's root
 * @param item The object being add
 * @return Return current Heap class node.
 */
private Node<HeapClass<E>> add(Node<HeapClass<E>> localRoot, E item) {
    if (localRoot == null) {
        addReturn = true;
        Node<HeapClass<E>> newN = new Node<HeapClass<E>>();
        newN.data = new HeapClass<>();
        newN.data.add(item);
        return newN;
    }
    else if (localRoot.data.size() < MAX_HEAP) {
        localRoot.data.add(item);
        addReturn = true;
        return localRoot;
    }
    else if (item.compareTo(localRoot.data.peek()) < 0) {
        localRoot.left = add(localRoot.left, item);
        return localRoot;
    }
    else {
        localRoot.right = add(localRoot.right, item);
        return localRoot;
    }
}
```

Data.add() = > O(n)
Peek() = > O(1)
compareTo = > O(1)
T(n) = > T(n-1) + O(n) + O(1)

FindLargestChild Method

```java
    /**
     * Find largest child in Binary Search Tree.
     * @param parent The parent node in Tree.
     * @return Return largest node in Binary Search Tree.
     */
    private HeapClass<E> findLargestChild(Node<HeapClass<E>> parent) {
        if (parent.right.right == null) {
            HeapClass<E> returnValue = parent.right.data;
            parent.right = parent.right.left;
            return returnValue;
        }
        else {
            return findLargestChild(parent.right);
        }
    }
```

Qbest() = > T(n) = O(1)
QWorst = T(n-1) + O(1)

Delete Method

```java
     */
    private Node<HeapClass<E>> delete(Node<HeapClass<E>> localRoot, E item) {
        if (localRoot == null) {
            deleteReturn = null;
            return localRoot;
        }
        boolean search = localRoot.data.search(item);
        assert localRoot.data.peek() != null;
        int compResult = item.compareTo(localRoot.data.peek());
        if(search){
            localRoot.data.remove(item);
            if(localRoot.data.size() == 0) {
                if (localRoot.left == null) {
                    return localRoot.right;
                } else if (localRoot.right == null) {
                    return localRoot.left;
                } else {
                    if (localRoot.left.right == null) {
                        localRoot.data = localRoot.left.data;
                        localRoot.left = localRoot.left.left;
                        return localRoot;
                    } else {
                        localRoot.data = (HeapClass<E>) findLargestChild(localRoot.left);
                        return localRoot;
                    }
                }
            }
            return localRoot;
        }
        if (compResult < 0) {
            localRoot.left = delete(localRoot.left, item);
            return localRoot;
        }
```

QBest = > T(n) = O(1) (If localroot is null)
Q(worst):

Search= Search= O(n)
compareTo = O(1)
peek() = O(1)
data.remove() = O(logn)
size() = O(1)
findLargestChild = T(n-1) + O(1)

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