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

Sena Nur Ulukaya
1901042622

1. SYSTEM REQUIREMENTS

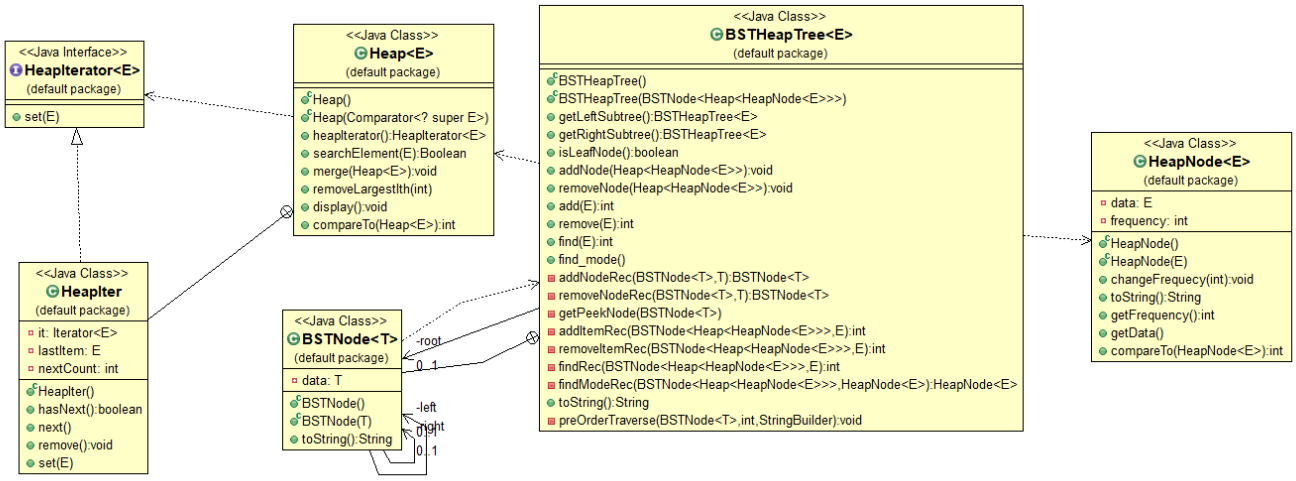
a. Non-Functional Requirements

- Java Runtime Environment to run code.
- Some memory to test the program. (50-100KB)
- Java as programming language.

b. Functional Requirements

- I used PriorityQueue represent Heap, since Heap extends PriorityQueue it has all the functionality of PriorityQueue. (add, remove, contains etc.)
- Searching element in Heap.
- Merging a heap with another heap.
- Removing largest element of the heap. 1st largest element of the heap is largest element, sizeth largest element of the heap is smallest element.
- Displaying heap.
- Heap is comparable.
- HeapIterator with extended set method to set heap elements with specified element while iterating.
- Adding item to the BSTHeapTree.
- Removing item from the BSTHeapTree.
- Finding frequency of each data of the BSTHeapTree.
- Finding mode of the BSTHeapTree.

2. USE CASE AND CLASS DIAGRAMS



3. PROBLEM SOLUTION APPROACH

Firstly, for part 1, I have used PriorityQueue to represent my Heap class. I have extended it. Also, I have created a interface, which extends Iterator, with a new set method. Then I used my interface to create inner class HeapIter and added my new feature there.

To merging I have simply used feature of the PriorityQueue addAll. You can't merge a Heap with itself, it will give exception.

To search element, I iterated with my iterator and search the whole heap.

To remove largest i^{th} element, I made an array from my Heap and then I sorted it. It helped me to easily find the index of the element. Because they're already in increasing order. My range is between 1-size. Because 0^{th} largest element means nothing. 1^{st} largest element is largest element and sizeth largest element is smallest element.

Secondly, for part 2, I have created BSTNode class to represent BST nodes and HeapNodes to represent Heap nodes with a data and a frequency. My Heaps are the same with part 1 but I make it maxheap with the comparator.

I have HeapNodes of Heaps of BSTNodes to my class to represent our structures. Basically, BST has BSTNodes in it and BSTNodes has Heap in it and Heap has HeapNodes with a data and frequency.

I have used classic BST methods to add and remove nodes. (The ones from the class.)

I compared the Heap's peek for traversing and building the structure.
I have used this algorithm to add item. Firstly, I tried to find if item exist before, if it is I just increased frequency. If it does not, I have tried to find a not-full node recursively for it within the BST structure suitable.

I have used this algorithm to remove item. Firstly, If there's more than one of that item I just decreased the frequency. But if there's only 1, then I find it recursively and then remove it. But since removing occurs emptiness in the Heap and I don't want it for the not-leaf Heaps, I removed the Heap and add its element one by one again. By doing this, I add the items to the proper places and prevent the emptiness. Exception if element does not exist.

For find method, again I traversed nodes recursively and search the inside of the heaps.
If I can find the item, I returned its frequency, else exception.

For mode method, I have find the most occurred element of the heap and take it. I have compare it with its right and left and I get the biggest frequency each time. After recursively traversing each heap, I returned mode if it does exist and null if it does not.

4. TEST CASES AND RUNNING AND RESULTS

PART1:

- I created 2 heaps and add some numbers to them.

```
Numbers from 50 to 1 added to heap1.
Heap1:
1 5 2 14 6 3 22 19 15 11 7 4 27 25 23 34 20 18 16 30 12 10 8 26 21 45 28 46 38 39 24 50 41 44 29 47 35 36 17 48 33 42 13 43 31 32 9 49 37 40

Numbers from 70 to 30 added to heap2.
Heap2:
30 31 41 34 32 46 42 39 35 33 51 57 47 45 43 54 40 38 36 50 62 63 52 69 60 65 48 66 58 59 44 70 61 64 49 67 55 56 37 68 53
```

- I search for some exist and some non-exist element.

```
Searching for 0 in heap 1.
Element does not exist.

Searching for 10 in heap 1.
Element does exist.

Searching for 20 in heap 1.
Element does exist.

Searching for 30 in heap 1.
Element does exist.

Searching for 40 in heap 1.
Element does exist.

Searching for 50 in heap 1.
Element does exist.

Searching for 60 in heap 1.
Element does not exist.

Searching for 70 in heap 1.
Element does not exist.

Searching for 80 in heap 1.
Element does not exist.

Searching for 90 in heap 1.
Element does not exist.
```

- I merged Heap2 in Heap1.

```
Heap1:
1 5 2 14 6 3 22 19 15 11 7 4 27 25 23 34 20 18 16 30 12 10 8 26 21 31 28 42 35 33 24 43 40 36 29 47 35 36 17 44 33 42 13 37 31 32 9 49 37 40 30 45 41 34 32 46 46 39 38 39 51 57 47 50 45 54 41 44 38 50 62 63
 52 69 60 65 48 66 58 59 48 70 61 64 49 67 55 56 43 68 53
Heap2:
30 31 41 34 32 46 42 39 35 33 51 57 47 45 43 54 40 38 36 50 62 63 52 69 60 65 48 66 58 59 44 70 61 64 49 67 55 56 37 68 53
```

- I merged heap2 with itself, it will give error.

```
Merging heap2 with itself.
Some property of an element of the specified Heap prevents it from being added to this queue, or if the specified Heap is this heap.
```

- I removed ith largest elements from Heap1.

```
Heap1 before removing:
1 5 2 14 6 3 22 19 15 11 7 4 27 25 23 34 20 18 16 30 12 10 8 26 21 31 28 42 35 33 24 43 40 36 29 47 35 36 17 44 33 42 13 37 31 32 9 49 37 40 30 45 41 34 32 46 46 39 38 39 51 57 47 50 45 54 41 44 38 50 62 63
 52 69 60 65 48 66 58 59 48 70 61 64 49 67 55 56 43 68 53

Removing smallest that is 91th largest element from heap1. Size: 91
1 removed from heap1. New size: 90

Removing largest that is 1th largest element from heap1. Size: 90
70 removed from heap1. New size: 89

Removing 12th largest element from heap1. Size: 89
58 removed from heap1.New size: 88

Removing 24th largest element from heap1. Size: 88
48 removed from heap1.New size: 87

Removing 36th largest element from heap1. Size: 87
41 removed from heap1.New size: 86

Removing 48th largest element from heap1. Size: 86
35 removed from heap1.New size: 85
```

```
Removing 84th largest element from heap1. Size: 83
There's no largest 84th element of this heap. The size is just 83

Removing 96th largest element from heap1. Size: 83
There's no largest 96th element of this heap. The size is just 83
```

```
Heap1 after removing:
2 5 3 15 6 4 22 19 16 11 7 21 28 25 23 34 20 18 17 30 12 10 8 26 30 31 32 42 38 33 24 43 40 36 29 47 35 36 43 44 33 42 13 37 31 32 9 49 37 40 53 45 55 34 49 46 46 39 67 39 51 57 47 50 45 54 41 44 38 50 62 6
3 52 69 60 65 56 66 64 59 48 68 61
```

- I used my iterator set method.

```
SET METHOD OF ITERATOR:

Heap2 before setting any element:
30 31 41 34 32 46 42 39 35 33 51 57 47 45 43 54 40 38 36 50 62 63 52 69 60 65 48 66 58 59 44 70 61 64 49 67 55 56 37 68 53

Setting 32 instead of 30
Setting 47 instead of 45
Setting 62 instead of 60


Heap2 after setting:
31 32 41 34 32 46 42 39 35 33 51 50 47 53 43 54 40 38 36 47 62 63 52 69 57 65 48 66 58 59 44 70 61 64 49 67 55 56 37 68 62
```

PART2:

- 3000 numbers within range 5000 added to the BSTHeapTree. (You can print them but I couldn't because it'll take a lot of place in terminal.)

- Frequency Test:

- Search for 100 numbers in the array, some of it is:

```
Number of occurrences of 106 in the BSTHeapTree: 2
Number of occurrences of 106 in the arrayList: 2


Number of occurrences of 107 in the BSTHeapTree: 1
Number of occurrences of 107 in the arrayList: 1


Number of occurrences of 109 in the BSTHeapTree: 1
Number of occurrences of 109 in the arrayList: 1
```

- 10 numbers not in the array:

```
Number of occurrences of 5001 in the BSTHeapTree: 0. Item does not exist.
Number of occurrences of 5001 in the arrayList: 0. Item does not exist.


Number of occurrences of 5002 in the BSTHeapTree: 0. Item does not exist.
Number of occurrences of 5002 in the arrayList: 0. Item does not exist.


Number of occurrences of 5003 in the BSTHeapTree: 0. Item does not exist.
Number of occurrences of 5003 in the arrayList: 0. Item does not exist.


Number of occurrences of 5004 in the BSTHeapTree: 0. Item does not exist.
Number of occurrences of 5004 in the arrayList: 0. Item does not exist.
```

- Find the mode of the BSTHeapTree: I have returned array of Modes in arrayList because there can be more than one and it can be different from BSTHeapTree's.

```
Mode of the BSTHeapTree: 2435 with frequency 5
Mode of the arrayList: [2435] with frequency 5
```

```
Mode of the BSTHeapTree: 1436 with frequency 5
Mode of the arrayList: [1436, 1802, 4505, 4706] with frequency 5
```

- Remove 100 numbers in the array:

```
Number of occurrences of 168 before removeing in the BSTHeapTree: 3
Number of occurrences of 168  after removing in the BSTHeapTree: 2


Number of occurrences of 170 before removeing in the BSTHeapTree: 1
Number of occurrences of 170  after removing in the BSTHeapTree: 0


Number of occurrences of 173 before removeing in the BSTHeapTree: 2
Number of occurrences of 173  after removing in the BSTHeapTree: 1
```

- 10 numbers not in the array:

```
5001 does not exist. You can't remove.
5002 does not exist. You can't remove.
5003 does not exist. You can't remove.
5004 does not exist. You can't remove.
5005 does not exist. You can't remove.
5006 does not exist. You can't remove.
5007 does not exist. You can't remove.
5008 does not exist. You can't remove.
5009 does not exist. You can't remove.
5010 does not exist. You can't remove.
```

5. PART3: TIME COMPLEXITY ANALYZE

- Heap Class

```
public Boolean searchElement(E element){
    HeapIterator<E> it = heapIterator();

    while(it.hasNext()){
        if(it.next().equals(element)) return true;
    }

    return false;
}
```

- $T_w(n) = \theta(n)$, because if element does not exist it will loop n times.
- $T_b(n) = \theta(1)$, it can be first element.
- $T(n) = O(n)$

```
public void merge(Heap<E> other)
    this.addAll(other);
}
```

- addAll's complexity is O(mlogn). m is the size of other Heap.
- T(n) = O(m.logn).

```
public E removeLargestIth(int index) throws IndexOutOfBoundsException{
    if(index <= 0 || index > this.size()) throw new IndexOutOfBoundsException();
    Object[] arr = this.toArray();
    Arrays.sort(arr);
    E element = (E) arr[this.size()-index];
    if(this.remove(element)) return element;
    return null;
}
```

toArray is θ(n) and sort θ(nlogn) and removing is O(logn) since finding the item takes n generally and deleting it takes logn times.

- T(n) = O(nlogn)

```
public int compareTo(Heap<E> o) {
    if(this.peek().compareTo(o.peek()) < 0) return -1;
    if(this.peek().compareTo(o.peek()) > 0) return 1;
    else return 0;
}
```

- Since finding peek takes constant time. It is constant time.
- T(n) = θ(1).

```
public E set(E value) throws IllegalStateException{
    if(lastItem == null) throw new IllegalStateException();
    it.remove();
    offer(value);

    it = iterator();
    for (int i = 0; i < nextCount; i++) {
       it.next();
    }

    E item = lastItem;
    lastItem = null;
    return item;
}
```

- Offer is O(logn) and there's loop until nextCount which is to prevent exception of the iterator and it runs n times in the worst case and 1 time in the best case.
- T(n) = O(n)

```
public boolean hasNext() {
    return(it.hasNext());
}

/** Move the iterator forward and return the next item.
 * @return The next item in the list
 * @throws NoSuchElementException if there is no such object
 */
@Override
public E next() throws NoSuchElementException{
    lastItem = it.next();
    nextCount++;
    return(lastItem);
}

/** Remove the last item returned. This can only be done once per call to next.
 *  @throws IllegalStateException if next was not called prior to calling this method.
 */
@Override
public void remove() throws IllegalStateException{
    it.remove();
}
```

- hasNext, next is θ(1) and remove is O(logn).

- BSTHeapIterator Class

```
public boolean isLeafNode() {
    return (root.left == null && root.right == null);
}
```

- T(n) = θ(1).

```
public void addNode(Heap<HeapNode<E>> data){
    root = addNodeRec(root, data);
}
```

```
private <T extends Comparable<T>> BSTNode<T> addNodeRec(BSTNode<T> curr, T data){
    if (curr == null) {
        return new BSTNode<>(data);
    }

    if (data.compareTo(curr.data) < 0) curr.left = addNodeRec(curr.left, data);
    else if (data.compareTo(curr.data) > 0) curr.right = addNodeRec(curr.right, data);
    return curr;
}
```

- h is the height of the tree and, T(h) = T(h-1) + θ(1) – rest of it is constant time. And h is logn because it's a complete tree.
- T(n) = O(logn)

```
private <T extends Comparable<T>> BSTNode<T> removeNodeRec(BSTNode<T> curr, T data){
    if(curr == null) return null;

    if (data.compareTo(curr.data) < 0) curr.left = removeNodeRec(curr.left, data);
    else if (data.compareTo(curr.data) > 0) curr.right = removeNodeRec(curr.right, data);

    else{

        if(curr.left == null) return curr.right;
        if(curr.right == null) return curr.left;
        else{
            if(curr.left.right == null){
                curr.data = curr.left.data;
                curr.left = curr.left.left;
            }

            else{
                curr.data = getPeekNode(curr.left);
            }
        }
    }

    return curr;
}
```

```
public void removeNode(Heap<HeapNode<E>> data){
    root = removeNodeRec(root,data);
}
```

- h is the height of the tree and, T(h) = T(h-1) + θ(1) – rest of it is constant time. And h is logn because it's a complete tree.
- T(n) = O(logn)

```java
private int addItemRec(BSTNode<Heap<HeapNode<E>>> curr,E item){
    if(curr == null){
        HeapNode<E> temp = new HeapNode<>(item);
        Heap<HeapNode<E>> newHeap = new Heap<>(Collections.reverseOrder());
        newHeap.add(temp);
        addNode(newHeap);
        return(1);
    }

    if(item.compareTo(curr.data.peek().getData()) <= 0){
        Iterator<HeapNode<E>> it = curr.data.iterator();
        while(it.hasNext()){
            HeapNode<E> temp = it.next();
            if(temp.getData().equals(item)){
                temp.changeFrequecy(temp.getFrequency()+1);
                return temp.getFrequency();
            }
        }
    }

    if(curr.data.size() < 7){
        curr.data.add(new HeapNode<>(item));
        return 1;
    }

    if(item.compareTo(curr.data.peek().getData()) > 0) return(addItemRec(curr.right, item));
    else return(addItemRec(curr.left,item));
}
```

```java
public int add(E item){
    return(addItemRec(root, item));
}
```

- Adding to heap is 0(logn) and adding node is O(logn) too.
- There's a loop and it can max loop 7 time we know that so constant.
- T(n) = O(logn)

```java
public int remove(E item){
    return(removeItemRec(root, item));
}
private int removeItemRec(BSTNode<Heap<HeapNode<E>>> curr,E item){
    if(curr == null){
        throw new NoSuchElementException();
    }

    if(item.compareTo(curr.data.peek().getData()) <= 0){
        E peek = curr.data.peek().getData();
        Boolean removed = false;
        int freq = 0;
        Iterator<HeapNode<E>> it = curr.data.iterator();
        while(it.hasNext()){
            HeapNode<E> temp = it.next();
            freq = temp.getFrequency();
            if(temp.getData().equals(item) && freq > 1){
                temp.changeFrequecy(--freq);
                removed = true;
            }

            else if(temp.getData().equals(item) && freq == 1){
                temp.changeFrequecy(--freq);
                curr.data.remove(temp);
                removed = true;
            }

            if(removed) break;
        }
```

```java
        if(curr.data.isEmpty()){
            curr.data.add(new HeapNode<>(peek));
            removeNode(curr.data);
            return(freq);
        }

        if(removed){
            Heap<HeapNode<E>> backup = curr.data;
            removeNode(curr.data);
            Iterator<HeapNode<E>> i = backup.iterator();
            while(i.hasNext()){
                HeapNode<E> hNode = i.next();
                for(int j = 0; j < hNode.getFrequency(); ++j){
                    add(hNode.getData());
                }
            }
            return(freq);
        }
    }

    if(item.compareTo(curr.data.peek().getData()) > 0) return(removeItemRec(curr.right, item));
    else return(removeItemRec(curr.left,item));
```

- Removing from heap is 0(logn) and removing node is O(logn) too.
- There's a loop and it can max loop 7 time we know that so constant and there's another loop in it and it can be m.
- T(n) = O(mlogn)

```
private int findRec(BSTNode<Heap<HeapNode<E>>> curr,E item){
    if(curr == null){
        throw new NoSuchElementException();
    }

    if(item.compareTo(curr.data.peek().getData()) <= 0){
        Iterator<HeapNode<E>> it = curr.data.iterator();
        while(it.hasNext()){
            HeapNode<E> temp = it.next();
            if(temp.getData().equals(item)){
                return temp.getFrequency();
            }
        }
    }

    if(item.compareTo(curr.data.peek().getData()) > 0) return(findRec(curr.right, item));
    else return(findRec(curr.left,item));
}
public int find(E item){
    return(findRec(root,item));
}
```

- There's a loop and it can max loop 7 time we know that so constant. It is traversing recursively that is O(logn).
- T(n) = O(logn)

```
private HeapNode<E> findModeRec(BSTNode<Heap<HeapNode<E>>> curr, HeapNode<E> largestFreq){
    Iterator<HeapNode<E>> it = curr.data.iterator();
    while(it.hasNext()){
        HeapNode<E> temp = it.next();
        if(temp.getFrequency() > largestFreq.getFrequency()) largestFreq = temp;
    }

    if(curr.left != null){
        HeapNode<E> leftFreq = findModeRec(curr.left, largestFreq);
        if(leftFreq.getFrequency() > largestFreq.getFrequency()) largestFreq = leftFreq;
    }

    if(curr.right != null){
        HeapNode<E> rightFreq = findModeRec(curr.right, largestFreq);
        if(rightFreq.getFrequency() > largestFreq.getFrequency()) largestFreq = rightFreq;
    }

    return largestFreq;
}
```

```
public E find_mode(){
    HeapNode<E> result = findModeRec(root, root.data.peek());
    if(result != null) return(result.getData());
    else return null;
}
```

- It is traversing recursively but this time with 2 direction right and left.
- So it is, T(h) = 2.T(h-1) + θ(1) this time and that is O(n).
- T(n) = O(n).

```java
private <T extends Comparable<T>> T getPeekNode(BSTNode<T> parent){
    if (parent.right.right == null) {
        T peek = parent.right.data;
        parent.right = parent.right.left;
        return peek;
    }

    else return getPeekNode(parent.right);
}
```

- It is traversing recursively in right direction.
- T(n) = O(logn).

```java
/**
 * Display the tree while traversing preorder.
 */
public String toString() {
    StringBuilder sb = new StringBuilder();
    preOrderTraverse(root, 1, sb);
    return sb.toString();
}

/** Perform a preorder traversal.
 * @param node The local root
 * @param depth The depth
 * @param sb The string buffer to save the output
*/
private <T extends Comparable<T>> void preOrderTraverse(BSTNode<T> node, int depth, StringBuilder sb){
    for (int i = 1; i < depth; i++) sb.append("  ");
    if (node == null) sb.append("null\n");
    else{
        sb.append(node.toString());
        sb.append("\n");
        preOrderTraverse(node.left, depth + 1, sb);
        preOrderTraverse(node.right, depth + 1, sb);
    }
}
```

- PreorderTraverse traversing each direction as I mentioned before so O(n).
- To string is O(n) too in this case.