GTU Department of Computer Engineering CSE 222/505 - Spring 2021

Homework 4 Method Analyzes

BAHADIR ETKA KILINÇ
1901042701

Part1

1. Search for an element:

```
public E search(E data){
    for(int i=0; i<theData.size(); ++i){
        if(theData.get(i).equals(data))
            return theData.get(i);
    }
    return null;
}
```

Items kept on java Array List and its get method takes constant time.
For loops takes Θ(n) time.
So the total running time is Θ(n)

2. Merge with another heap

```
public void merge(BKPriorityQueue<E> other){
    ArrayList<E> temp = new ArrayList<>();
    BKIterator<E> iter = other.iterator();
    while(iter.hasNext()){
        temp.add(iter.next());
    }
    for(int i=0; i<temp.size(); ++i){
        this.offer(temp.get(i));
    }
}
```

The while loop takes Θ(n) time.
For loop takes Θ(n) time.
If we add them the total running time will be Θ(n).

3. Removing ith largest element from the Heap

```java
public E removeLargestOf(int index){
    if(index < 1 || index > size())
        throw new ArrayIndexOutOfBoundsException();

    if(size() > 0){
        ArrayList<E> temp = new ArrayList<>();
        for(int i=0; i<size(); i++){
            temp.add(theData.get(i));
        }
        temp.sort(comparator);
        int parent = theData.indexOf(temp.get(size()-index));
        parent = takeDownData(parent);
        swap(parent, minChild: size()-1);
        heapUp(parent);
        E res = theData.get(size()-1);
        theData.remove( index: size()-1);
        return res;
    }
    return null;
}
```

For loop takes Θ(n) time and the others takes amortized constant time.
So the total running time will be Θ(n).

4. Iterator's set method

```java
@Override
public E set(E data) {
    int tempIndex = theData.indexOf(lastReturned);
    theData.set(tempIndex,data);
    int child = tempIndex;
    heapUp(child);
    int parent = child;
    while (true) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= theData.size()) {
            break;
        }
        int rightChild = leftChild + 1;
        int maxChild = leftChild;

        if (rightChild < theData.size()
                && compare(theData.get(rightChild),
                theData.get(leftChild)) > 0) {
            maxChild = rightChild;
        }
        if (compare(theData.get(parent),
                theData.get(maxChild)) < 0) {
            swap(parent, maxChild);
            parent = maxChild;
        } else {
            break;
        }
    }
    return lastReturned;
}
```

While loop takes O(n) time.
So the total running time is O(n).

Part 2

1. int add (E item) – returns the number of occurrences of the item after insertion

```java
public int add(E data){
    if(root == null){
        root = new BSTNode<>(new BKMaxBSTHeapNode<E>());
        return root.theData.add(data);
    }
    int num = searchTree(root,data);
    if(num == -1)
        return add(root,root,data);
    return num;
}
```

Best case of this method is O(1).

```java
private int searchTree(BSTNode<E> root, E data){
    if(root == null)
        return -1;
    if(root.theData.contains(data))
        return root.theData.add(data);
    int res = root.theData.peek().compareTo(data);
    if(res > 0)
        return searchTree(root.left,data);
    return searchTree(root.right,data);
}
```

If root is not null then add method calls search tree method. That methods searches element in tree by comparing node's peek element. Node's contains method takes O(1)(the max node size is fixed with 7) and every call takes O(log(n)). If we multiply these total running time will be O(log(n)).

```java
private int add(BSTNode<E> current, BSTNode<E> last, E data){
    if(current == null){
        int res = last.theData.peek().compareTo(data);
        if(res > 0){
            last.left = new BSTNode<>(new BKMaxBSTHeapNode<E>());
            return last.left.theData.add(data);
        }
        else{
            last.right = new BSTNode<>(new BKMaxBSTHeapNode<E>());
            return last.right.theData.add(data);
        }
    }
    if(current.theData.size() < 7){
        return current.theData.add(data);
    }
    else {
        int com = current.theData.peek().compareTo(data);
        if (com > 0) {
            return add(current.left, current, data);
        }
        else {
            return add(current.right, current, data);
        }
    }
}
```

If the element is not contained in the tree this method executed.
And this method takes O(log(n))

So if we add these three methods running time O(1) + O(log(n)) + O(log(n)) the total running time will be O(log(n)).

2. int remove (E item) – returns the number of occurrences of the item after removal

```java
public int remove(E data){
    if(root == null){
        return -1;
    }
    return remove(root,root, STATUS.INVALID,data);
}
```

Best case of remove is O(1).

```java
private int remove(BSTNode<E> current, BSTNode<E> last, STATUS sts, E data){
    if(current == null){
        return -1;
    }
    if(current.theData.contains(data)){

        int res = current.theData.remove(data);
        if(current.theData.size() == 0){
            if(current.right != null) {
                if(current.right.left == null){
                    if(sts == STATUS.RIGHT) {
                        last.right = current.right;
                        last.right.left = current.left;
                    }
                    if(sts == STATUS.LEFT){
                        last.left = current.right;
                        last.left.left = current.left;
                    }
                    return 0;
                }
                current.theData = helper(current.right, current.right);
            }
            else{
                if(current.left != null){
                    if(sts == STATUS.LEFT)
                        last.left = current.left;
                    if(sts == STATUS.RIGHT)
                        last.right = current.left;
                    return 0;
                }
                if(sts == STATUS.LEFT)
                    last.left = null;
                if(sts == STATUS.RIGHT)
                    last.right = null;
            }
            return 0;
        }
        return res;
    }
    int res = current.theData.peek().compareTo(data);
    if(res > 0){
        return remove(current.left,current, STATUS.LEFT,data);
    }
    else{
        return remove(current.right,current, STATUS.RIGHT,data);
    }
}
```

Searching an element in each node takes constant time because the node size is fixed with 7.
If current node(will be deleted one) has right's left child helper method executes and that will takes O(n) time.
Traveling on tree with comparing MaxHeap's peek takes O(log(n)).

```java
private BKMaxBSTHeapNode<E> helper(BSTNode<E> current, BSTNode<E> last){
    if(current.left == null){
        last.left = null;
        return current.theData;
    }
    return helper(current.left,current);
}
```

So the worst case of every call takes O(n) and that call

executed O(log(n)) time. If we multiply these two the total running time of remove method will be O(nlog(n))