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

PART 3

Refik Orkun Arslan
151044063

PART 1

offer ->O(n*log(n))

```
public boolean offer(E item) {
    theData.add(item);

    int child = theData.size() - 1;
    int parent = (child - 1)/2;

    while(parent >= 0 && compare(theData.get(parent), theData.get(child)) > 0)
{O(n*log(n))
        swap(parent, child); O(1)
        child = parent;O(1)
        parent = (child - 1)/2;O(1)
    }
    return true;
}

private void swap(int i, int j){ O(1)
    E first = theData.get(i); //get first reference
    theData.set(i, theData.get(j)); //get second reference and set it to first
    theData.set(j, first);  //set second reference to first reference
}
private int compare(E left, E right){CompareTo ->O(n*log(n))
    if(comparator != null){ //use comparator if defined
        return comparator.compare(left, right);
    } else { // use left's compareTo method
        return ((Comparable<E>) left).compareTo(right);-
    }
}
```

SearchFor -> O(n^2)

```
public int searchFor(E target)
{
    for(int i = 0; i < theData.size(); i++)  O(n)
    {
        if(target.equals(theData.get(i))) equals->O(n)
            return i;
    }
    return -1;
}
```

Merge ->O((a.size+b.size)*logn/2)

```
public  void mergeHeaps(ArrayList<Integer> arr, ArrayList<Integer> a,ArrayList<Integer>
b)
{
    for (int i = 0; i < a.size(); i++) {  O(a.size);
        arr.add(i,a.get(i));
    }
    for (int i = 0; i < b.size(); i++) {  O(b.size);
        arr.add((a.size()+i) ,b.get(i));
    }
    int n;
    n = a.size() + b.size();

    for (int i = arr.size() / 2 - 1; i >= 0; i--) { O(a.size+b.size/2);
        minHeapify(i,arr);->log(n)
    }
}

private  void minHeapify(int i,ArrayList<Integer> list)->log(n)
{
 recursion 2 * i so log(n)
    int left = 2 * i + 1;
    int right =  2 * i + 2;
    int smallest = -1;
    if (left <= list.size() - 1 && list.get(2*i+1) < list.get(i)) {
        smallest = left;
    } else {
        smallest = i;
    }

    if (right <= list.size() - 1 && list.get(2 * i + 2) < list.get(smallest)) {
        smallest = right;
    }

    if (smallest != i) {

        swap(i, smallest);
        minHeapify(smallest,list);
    }
}
```

RemoveLargest ->

```
public void removeThLargest(int a) ->O(n^2)
{
        E m;
        m=kthLargest(theData,a);
        theData.remove(m); O(n)

}


public E kthLargest( ArrayList<E> arr,int k) ->O(n^2)
{
    ArrayList<E> a;
    a=arr;
    bubbleSort(a); ->O(n^2)
    return a.get(a.size()-k-1);
}


public void bubbleSort(ArrayList<E> array) { ->O(n^2)
    boolean swapped = true;
    int j = 0;
    E tmp;
    while (swapped) {
        swapped = false;
        j++;
        for (int i = 0; i < array.size() - j; i++) { O(n)
            if (compare(array.get(i),array.get(i+1))>0) { compare ->O(n)
                tmp = array.get(i);
                array.set(i,array.get(i+1));
                array.set(i+1,tmp);
                swapped = true;
            }
        }
    }
}
```

PART 2

BSTHeapTree add-> O(n^2)

```
private Node<E> add(Node<E> localRoot,  E item){  O(n^2)



    if(localRoot == null){  O(1)

        addReturn = true; O(1)
        return new Node<E>(new MaxHeap<E>()); O(1)
    } else {
        if(localRoot.data.getTheData().size()<=7) O(1)
        {
            localRoot.data.add(item);   O(n)

            return localRoot;O(1)
        }
        else
        {
            int compare = item.compareTo(localRoot.data.getTheData().get(0)); O(n)
            if ( compare==0){ O(1)

                addReturn = false;O(1)
                return localRoot;O(1)
            } else if (compare < 0){
                    localRoot.left = add(localRoot.left, item); O(n)
                    return localRoot;
            } else {
                    localRoot.right = add(localRoot.right, item); O(n)



                return localRoot;
        }
```

BSTHeapTree find-> O(n*2)

```
private Node<E> find(Node<E> localRoot, E target){
    if(localRoot == null)O(1)
        return null;
    for(int i =0;i<localRoot.data.getTheData().size();i++) O(1)
    {
        if( (target.compareTo(localRoot.data.getTheData().get(i))==0)) O(n)
        {



            return localRoot;O(1)
        }
    }
    //Compare target with the data field at the root
    int compResult = target.compareTo(localRoot.data.getTheData().get(0));O(n)
    if(compResult == 0)
```

```
        return localRoot;
    else if (compResult < 0)
        return find(localRoot.left, target);O(n)
    else
        return find(localRoot.right, target);O(n)
}
```

*BSTHeapTree delete> O(n*3)*

```
private Node<E> delete(Node<E> localRoot, E item){
    if(localRoot == null){O(1)
        //item is not in the tree
        deleteReturn = null;O(1)
        return localRoot;O(1)
    }



    //search for the item to delete
    for(int i=0; i<localRoot.data.getTheData().size();++i) O(n)
    {



        if((item.compareTo(localRoot.data.getTheData().get(i)))==0)O(n)
        {



            localRoot.data.getTheData().remove(i);O(n)
            return localRoot;
        }
    }



    int compResult = item.compareTo(localRoot.data.getTheData().get(0)); O(n)
    if(compResult < 0){



        localRoot.left = delete(localRoot.left, item); O(n)
        return localRoot;
    } else if (compResult > 0){



        localRoot.right = delete(localRoot.right, item); O(n)
        return localRoot;
    } else {
        //item is at local root
      //  deleteReturn = localRoot.data;
        if(localRoot.left == null){
```

```
            //if there is no left child, return right child which can also be null
            if(localRoot.right.data.getTheData().size() !=1)
            {
                localRoot.right.data.getTheData().remove(0); O(n)
                return localRoot.right;O(1)
            }
            else
            {
                return localRoot.right;O(1)
            }



        } else if (localRoot.right == null){
            if(localRoot.left.data.getTheData().size() !=1)
            {
                localRoot.left.data.getTheData().remove(0); O(n)
                return localRoot.left;O(1)
            }
            else
            {
                return localRoot.left;O(1)
            }
        } else {
            //Node being deleted has 2 children, replace the data with inorder
predecessor
            if(localRoot.left.right == null){
                //the left child has no right child. Replace the data with the data in
the left child
                localRoot.data = localRoot.left.data;O(1)
                localRoot.left = localRoot.left.left; // replace the left child
withO(1) its left child
                if(localRoot.data.getTheData().size() !=1)
                {
                    localRoot.data.getTheData().remove(0); O(n)
                    return localRoot;O(1)
                }
                else
                {
                    return localRoot;O(1)
                }
            } else {
                //Search for the inorder predecessor and replace deleted node's data
with it
                localRoot.data = findLargestChild(localRoot.left);
                if(localRoot.data.getTheData().size() !=1)
                {
                    localRoot.data.getTheData().remove(0); O(n)
                    return localRoot;O(1)
                }
                else
                {
                    return localRoot;O(1)
                }
            }
        }
    }
}

private MaxHeap<E> findLargestChild(Node<E> parent){O(n/2)
    //if the right child has no right child, it is the inorder predecessor
```

```
    if(parent.right.right == null){O(1)
        MaxHeap<E> returnValue = parent.right.data;O(1)
        parent.right = parent.right.left;O(1)
        return returnValue;
    } else {
        return findLargestChild(parent.right);O(n/2)
    }
}
```

```
        }
    }
}
```