GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 4 Time Report

Muhammed Bedir ULUCAY

1901042697

Heap Class Time Complexity:

```java
/**Adding new element to heap*/
public int insert(E data) {

    int isExist = indexOf(data);
    if(isExist >= 0) {
        counter[isExist]++;
        return counter[isExist];
    }

    if(index == CAPACITY)
        return -1;

    heapData[index] = data;
    counter[index]++;
    int parent = (index-1) / 2 ,
        child = index;

    while( child != 0 && heapData[child].compareTo(heapData[parent]) > 0) {

        swap(child,parent);
        child = parent;
        parent = (child - 1) / 2;
    }
    return counter[index++];
}
```

We are use some algoritm to store data even we keep data in regular array but we are using an lineear search in indexOf = Θ(n) after that inseartion in array is Θ(logn)

T(n) = max(Θ(n) , Θ(log(n)) = Θ(n)

```java
/**Merge with another getting heap*/
public void merge(Heap<E> obj) {

    for(int i=0; i<obj.getSize();++i)
        this.insert(obj.get(i));
}
```

T(n) = Θ(o.n)

```java
/**Return the most occurrence element in heap*/
public E mod(int max) {

    for(int i=0; i<index; ++i)
        if(max == counter[i])
            return heapData[i];

    return null;
}
```

T(n) = Θ(n)

```
/**Return the most occurrence number*/
public int maxModNumber() {

    int max = counter[0];
    for(int i=0; i<index ; ++i) {
        if(counter[i] > max)
            max = counter[i];
    }
    return max;
}
```

T(n) = Θ(n)

```
/**Setting nth of array according to downward is necessary*/
private void moveToDownward(int idx) {

    int parent = idx,
        lChild = parent * 2 + 1,
        rChild = parent * 2 + 2;

    while(true) {

        if(lChild >= index-2)
            break;

        int maxChild = heapData[lChild].compareTo(heapData[rChild]) > 0
                ? lChild : rChild;

        if(heapData[maxChild].compareTo(heapData[parent]) > 0)
            swap(maxChild,parent);
        else
            break;

        parent = maxChild;
        lChild = parent * 2 + 1;
        rChild = parent * 2 + 2;

    }
}
```

Generally each line is constant time but the while true loop is moving some index to another index using heap rule it take logaritmic time

T(n) = Θ(log n)

```
/**Setting nth of array according to upward is necessary*/
private void moveToUpward(int idx) {

    int child = idx,
        parent = (idx - 1) / 2;

    while(child > 0 && heapData[child].compareTo(heapData[parent]) > 0) {

        swap(child,parent);
        child = parent;
        parent = (child - 1) / 2;
    }

}
```

Same as move downward

T(n) = Θ(log n)

```java
/**Finding nth max in the heap*/
public E findNthMax(int nth){

    Heap<E> h = new Heap<>(CAPACITY);
    for(int i=0; i<index; ++i)
        h.insert(heapData[i]);

    //Remove the nth time to find nth max element
    for(int i=0; i<nth; ++i)
        h.remove();

    return h.remove();
}
```

It has for in insert insert time complexity is as we sad Θ(logn) and for loop 0 to index = Θ(n)

After that loop there is another loop Θ (n) and in remove Θ (logn)

T(n) = Θ(2 * n logn) = Θ(nlogn)

```java
/**Removing the element which getting as parameter
 * if in the list remove it
 * return new occurrence otherwise return minus 1*/
public int removeData(E _data) {

    int index = indexOf(_data);
    if(index != -1) {
        int rest = counter[index];
        removeNthIndex(index);
        return rest == 1 ? 0 : counter[index];
    }
    return -1;
}
```

Index of = Θ(n)   removeNthIndex() = Θ(log n)

T(n) = max(Θ(n) , Θ(logn)) = Θ(n)

```java
/**Removing the root of heap and return it*/
public E remove() {

    if(isEmpty())
        return null;

    if(counter[0] > 1) {
        counter[0]--;
    }
    else if(counter[0] == 1) {

        E tmp = heapData[0];
        heapData[0] = heapData[index-1];
        heapData[index-1] = null;
        counter[0] = counter[index-1];
        counter[index-1] = 0;

        int parent = 0;
        int maxChild;

        while(true) {
            int lChild = 2 * parent + 1,
                rChild = 2 * parent + 2;

            if(lChild >= index-2)    break;

            maxChild = heapData[lChild].compareTo(heapData[rChild]) >= 0
                       ? lChild : rChild;
            if(heapData[parent].compareTo(heapData[maxChild]) < 0)
                swap(parent,maxChild);
            else
                break;
            parent = maxChild;
        }

        index--;
        return tmp;
    }
    return null;
}
```

Generally each line is constant time but in while loop there is while true loop to remove root of heap and it takes logaritmic time complexity.

T(n) = Θ(logn)

```java
/**Remove the nth of max value according to the heap rule*/
public E removeNthMax(int nth) {
    E data = findNthMax(nth);
    int idx = indexOf(data);
    return removeNthIndex(idx);
}
```

finfNthMax = Θ(n logn) indexOf = Θ(n) removeNthIndex = Θ(log n)

T(n) = max(nlogn, n, logn) = Θ(n logn)

```
/**Remove the nth of array according to the heap rule*/
public E removeNthIndex(int idx) {

    if(idx >= index || idx < 0)
        return null;

    if(counter[idx] > 1) {
        counter[idx]--;
        return heapData[idx];
    }

    E data = heapData[idx];
    if(idx == index-1) {
        heapData[idx] = null;
        counter[idx] = 0;
        index--;
        return data;
    }

    heapData[idx] = heapData[index-1];
    heapData[index-1] = null;
    counter[idx] = counter[index - 1];
    counter[index - 1] = 0;

    moveToUpward(idx);
    moveToDownward(idx);
    index--;
    return data;
}
```

Generally each lines are constant but moveUpWard and moveDownWard take logaritmic time.

T(n) = Θ(log n)

```
/**Wrapper Method
 * Searching element in heap
 * if heap has the element return true otherwise false */
public boolean search(E data) {
    return search(0, data);
}
/**Searching element in heap
 * if heap has the element return true otherwise false */
private boolean search(int location,E data) {

    if(location >= index || heapData[location].compareTo(data) < 0)
        return false;

    if(heapData[location].equals(data))
        return true;

    int lChild = location * 2 + 1,
        rChild = location * 2 + 2;

    return search(lChild,data) || search(rChild,data);
}
```

Search algoritm is look all node in heap using recursive approch we are look all node 1 time so that it takes like linnear time

T(n) = Θ(n)

```java
/**Setting element specific location*/
public E set(int pos,E data) {
    if(pos >= index)
        return null;

    E tmp = heapData[pos];
    heapData[pos] = data;

    int parent = pos,
        lChild = parent * 2 + 1,
        rChild = parent * 2 + 2;

    while(true) {

        if(lChild >= index-2)
            break;

        int maxChild = heapData[lChild].compareTo(heapData[rChild]) > 0
            ? lChild : rChild;
        if(heapData[maxChild].compareTo(heapData[parent]) > 0)
            swap(maxChild,parent);
        else
            break;

        parent = maxChild;
        lChild = parent * 2 + 1;
        rChild = parent * 2 + 2;
    }

    int child = parent;
        parent = (child - 1) / 2;

    while(child > 0 && heapData[parent].compareTo(heapData[child]) < 0) {
        swap(child,parent);
        child = parent;
        parent = (child - 1) / 2;
    }

    return tmp;
}
```

Set function has two while loop this loop like moveupward and movedownward each loop take log n time

T(n) = Θ(2 * logn) = Θ(log n)

```java
/**Wrapper method
 * Returning occurrence of getting data*/
public int find(E data) {
    return find(0, data);
}
/**Returning occurrence of getting data*/
private int find(Integer location,E data) {

    if(location >= index || heapData[location].compareTo(data) < 0)
        return 0;

    if(heapData[location].equals(data))
        return counter[location];

    Integer lChild = location * 2 + 1,
            rChild = location * 2 + 2;

    return find(lChild,data) + find(rChild,data);
}
```

Same as time complexity linear searh but we use recursive approch we need to look all node to absulute confirm.

T(n) = Θ(n)

```java
/**Return the index of data in list*/
public int indexOf(E data) {
    //Return the first same object index
    for(int i=0; i<index; ++i)
        if(heapData[i].equals(data))
            return i;
    return -1;
}
```

Linear search

T(n) = Θ(n)

BSTHeapTree Class Time Complexity:

We usually use O notations for the bad tree implementation generally time complexity is about the height of the tree if it is a worst tree that means h == n to it take linear time for the bad tree most of the time complexity I consider this stuation so I use O notation for time complexity.

```java
/**Wrapper Method for find_mode
 * Return the most occurrence element in the tree*/
public Integer find_mode() {
    Integer data = null;

    int max = find_mode_max(root,-1,data);
    if(max != -1)
        data = find_mode_num(root,max);

    return data;
}
```

both find mode method has **Θ(n)** time complexity so that this method takes linear time complexity

N = tree node number M = heap node number

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

```java
/**Return the most occurrence counter in the tree
 * Compare max current max occurrence with each node of heap*/
private Integer find_mode_max(Node local,Integer max,Integer data) {

    if(local == null)
        return max;
    if(local.h.maxModNumber() > max)
        max = local.h.maxModNumber();

    Integer right = find_mode_max(local.rBranch,max,data);
    Integer left = find_mode_max(local.lBranch,max,data);

    return left > right ? left : right;
}
```

This is classic tree recursive algorithm for searching an element to looking all node in tree.

Max mode Number is linear time for heap size

N = tree node number m = heap node number

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

```java
/**Return the most occurrence element in the tree */
private Integer find_mode_num(Node local,Integer max) {

    if(local == null)
        return null;

    if(local.h.maxModNumber() == max)
        return local.h.mod(max);

    Integer num2 = find_mode_num(local.rBranch,max);
    Integer num1 = find_mode_num(local.lBranch,max);

    return num1 != null ? num1 : num2;
}
```

This is classic tree recursive algorithm for searching an element to looking all node in tree

N = tree node number   m = heap node number

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

```java
/**Wrapper Method for remove element*/
public int remove(Integer _data) {
    return remove(root, _data);
}
/**Removing a number in tree and returning the occurance*/
private int remove(Node local,Integer _data) {

    if(local == null)
        return -1;

    if(local.h.search(_data)) {
        int result = local.h.removeData(_data);
        if(local.h.getSize() == 0) {
            local.h.insert(_data);
            root = removeNode(root,local.h);
        }
        return result;
    }
    if(local.compareTo(_data) > 0 )
        return remove(local.lBranch,_data);
    if(local.compareTo(_data) < 0)
        return remove(local.rBranch,_data);

    return -1;
}
```

searchin in bstree is logaritmic but search in heap node is linear

N = tree node number   m = heap node number

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

```java
/**If a node is empty then we remove the node*/
private Node removeNode(Node local,Heap<Integer> data) {

    if(local == null) return local;

    //Direction flow of control to find data
    if(data.compareTo(local.h) < 0)
        local.lBranch = removeNode(local.lBranch, data);

    else if(data.compareTo(local.h) > 0)
        local.rBranch = removeNode(local.rBranch, data);

    else {

        if (local.lBranch == null)      return local.rBranch;
        else if (local.rBranch == null) return local.lBranch;
        //Remove one by one until reach the leaf
        local.h = minValue(local.rBranch);
        local.rBranch = removeNode(local.rBranch, local.h);
    }

    return local;
}
```

Generally it takes time height of tree but in some specific case it could be take linear time unfortinatally if it is worst tree

T(n) = O(n)

```java
/**Returning minimum element of tree to remove method*/
private Heap<Integer> minValue(Node node) {

    if(node.lBranch != null)
        return minValue(node.lBranch);

    return node.h;
}
```

It takes logaritmic time because we need to find to minimum element moving always left direction

N = tree node number m = heap node number

T(n,m) = O(logn)

```java
/**Search an element in the tree using find method's return*/
public boolean search(Integer _data) {
    return find(_data) == 0 ? false : true;
}
```

Using find algorithm to return boolean value the data exist in the tree

N = tree node number m = heap node number

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

```java
/**Half Wrapper method
 * Adding new element to tree*/
public int add(Integer _data) {
    Integer occurance = 0;
    root = add(root,_data,occurance);
    return occurance;
}
/**Return the node to add new element*/
private Node add(Node local,Integer _data, Integer occurance){
    if(local == null) {
        Node newNode = new Node();
        occurance = newNode.add(_data);
        return newNode;
    }

    if(!local.h.isFull() || local.h.search(_data)) {
        occurance = local.add(_data);
        return local;
    }

    if(local.compareTo(_data) == 0)
        occurance = local.add(_data);
    if(local.compareTo(_data) > 0 )
        local.lBranch = add(local.lBranch,_data,occurance );
    if(local.compareTo(_data) < 0 )
        local.rBranch = add(local.rBranch,_data,occurance );

    return local;
}
```

The method use for binary tree comparision algorithm this is O(logn) complexith

And search is O(m) for heap add(logm) for heap

N = tree node number m = heap node number

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

```java
/**Wrapper Method for find method
 * Return the occurrence of getting data*/
public int find(Integer _data) {
    return find(root, _data);
}
/**Return the occurrence of getting data
 * Using recursive algorithm*/
private int find(Node local,Integer _data) {

    if(local == null)
        return 0;

    if(local.h.search(_data))
        return local.h.find(_data);

    return  find(local.lBranch,_data) + find(local.rBranch,_data);
}
```

It is using search for heap that takes linear time for heap and recursive approch is also check each node in tree this is also take linear time for the binary tree

N = tree node number m = heap node number

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