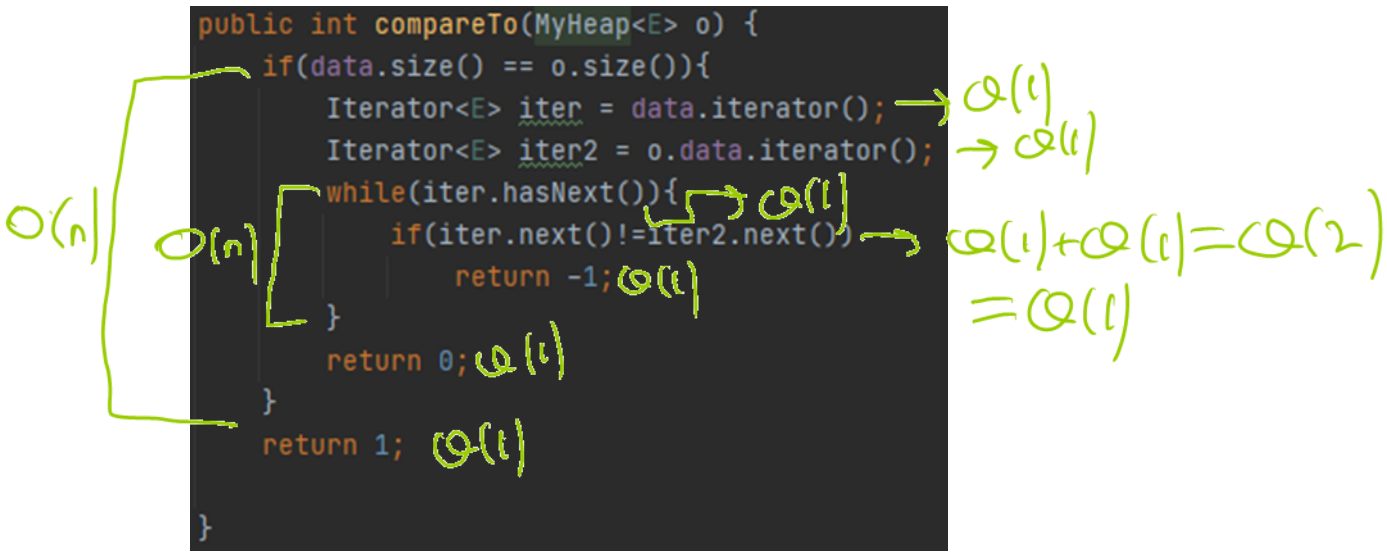
CSE222-HW#4
ANALYSIS OF METHODS

1) MyHeap Methods

n = size of this heap. Size of the o is not important because a) it is equal to n and b)if it is different then complexity is theta(1)



best case(data.size()!=o.size())

$T_B(n) = O(1)$

worst case(data.size()==o.size() and all elements of heaps are equal)

$T_W(n) = O(n)$

$T(n) = O(1) + O(n) = \underline{O(n)}$

```java
public void merge(MyHeap<E> other){
    PriorityQueue<E> temp;

    //for efficiency, smaller heap's elements should be appended to larger one.
    if(other.size() > size()) { //swap
        temp = this.data;
        this.data = other.data;
        other.data = temp;
    }
    while(!other.data.isEmpty()){
        data.add(other.data.poll());
    }
}
```

n = smaller heap's size.

O(1) = O(1) + O(1) + O(1) + O(1)

O(n) = O(n) · (O(1) + O(1))

$T(n) = O(n)$

```java
public E search(E element){

    for(E x : data){
        if(x.equals(element))
            return x;
    }
    return null;
}
```

n = size of the heap.

O(n) = O(n) · O(1) + O(1)

Worst case(element doesn't exist.):

$T_W(n) = O(n)$

Best case(element is in the first index.):

$T_B(n) = O(1)$

$T(n) = O(n) + O(1) = O(n)$

```java
public boolean removeIthLargest(int i){
    PriorityQueue<E> temp = new PriorityQueue<>( new Comparator<>() { // transform min heap to max heap.
        public int compare(E lhs, E rhs) {
            if (lhs.compareTo(rhs) < 0) return +1;
            if (lhs.equals(rhs)) return 0;
            return -1;
        }
    });

    int j;

    if(i<0 || i>= data.size())
        return false;
    for(j = 0;j<i;j++){
        temp.add(data.poll());
    }

    data.poll();

    for(j=0;j<data.size();j++){
        temp.add(data.poll());
    }

    data = temp;
    return true;
}
```

Annotations: compare method → sort the priority queue in descending order. Θ(1) for each comparator line and for the whole compare method; int j → Θ(1); if(i<0 || i>= data.size()) → Θ(1); return false → Θ(1); loops and poll/add operations → Θ(1) each; whole block → Θ(n); data = temp → Θ(1); return true → Θ(1).

n = size of the heap.

Create an empty temp heap(theta(1))

Poll all elements one by one and add all to temp except the given one (theta(n))

data = temp (theta(1))

$T(n) = \Theta(n)$

```java
public String toString(){
    StringBuilder returnString = new StringBuilder();

    for(E i : data){
        returnString.append(i).append("\n");
    }
    returnString.append("\n");
    return returnString.toString();

}
```

Annotations: n = size of heap. StringBuilder → Θ(1); for loop → Θ(n); append → Θ(1); returnString.append("\n") → Θ(1); return → Θ(1).

$T(n) = \Theta(n)$

Other methods of the MyHeap class and methods of the private inner class HeapIterator has theta(1) complexity.

2) BSTHeapTree methods

n = amount of Heaps
Amount of Nodes in a heap is at max 7. (const.)

```java
public int add(E data,int amount){
                                          → Θ(1)
    if(tree.isEmpty()){ //if BST is empty create a new heap and append to BST
        tree.add(new MyHeap<>()); → Θ(1)
    }

    MyHeap<Node<E> > availableHeap = null; → Θ(1)

    for(MyHeap<Node<E>> i : tree){
        MyIterator<Node<E>> iter = i.heapIter(); → Θ(1)
        while(iter.hasNext()){                   → Θ(1)
            Node<E> temp = iter.next();          → Θ(1)
            if(temp.getData().equals(data)) { //if the node already exists
                temp.insert(amount); → Θ(1)
                if(temp.getOccurrence() > mode.getOccurrence()) → Θ(1)
                    mode = temp; → Θ(1)
                return temp.getOccurrence(); → Θ(1)
            }
        }
        if(i.size()<7){ → Θ(1)
            availableHeap = i; → Θ(1)
        }
    }
    Node<E> newNode = new Node<>(data, amount); → Θ(1)
    if(availableHeap==null) { → Θ(1)
        availableHeap = new MyHeap<>(); → Θ(1)
        tree.add(availableHeap); → Θ(1)
    }

        availableHeap.add(newNode); → Θ(1)
        if(newNode.getOccurrence()>mode.getOccurrence()) → Θ(1)
            mode = newNode; → Θ(1)

        return newNode.getOccurrence(); → Θ(1)
}
```

Handwritten brackets in the left margin: the `while` loop is marked $O(7) = O(1)$; the `for` loop is marked $O(n)$.

```java
public int add(E data) { return add(data, amount: 1); }
```

Best Case: data is the first element of the first heap

$T_B(n) = \Theta(1)$

Worst Case: data doesn't exist

$T_w(n) = \Theta(n)$

$T(n) = \Theta(1) + \Theta(n) = O(n)$

```java
public int remove(E data) throws NoSuchElementException,IllegalStateException{
    if(tree.isEmpty()){ //if BST is empty create a new heap and append to BST
        throw new IllegalStateException("BSTHeapTree is empty.");
    }
    MyIterator<Node<E>> iter;
    for(MyHeap<Node<E>> i : tree){
        iter = i.heapIter();
        while(iter.hasNext()){
            Node<E> temp = iter.next();
            if(temp.getData().equals(data)) { //if the node exists
                temp.remove();
                if(temp.getOccurrence()==0){
                    i.remove(temp);

                    if(i.size()==0){
                        tree.remove(i);
                    }
                }
                if(mode.equals(temp))
                    update_mode();
                return temp.getOccurrence();
            }
        }
    }

    throw new NoSuchElementException("There is no such element in MyContainers.BSTHeapTree.");
}
```

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$

$\rightarrow O(1)$ → heap amount is at max 7.

$\rightarrow \Theta(1)$

$\rightarrow O(\log n)$

$O(1) + O(\log n) = O(n)$

$\rightarrow O(1)$

$O(n)$

$\rightarrow O(n)$

$\rightarrow O(1)$

$\rightarrow O(1)$

n = amount of Heaps
Amount of Nodes in a heap is at max 7. (const.)

Inner while will iterate 7 times at max. $(O(1))$
So: $O(n) \cdot (O(n) + O(\log n)) = O(n)$

Outer for loop:

$T_W(n)$ (element doesn't exist.):

$T_W(n) = O(n) \cdot O(n) = O(n^2)$

$T_B(n) = O(1) \cdot O(n) = O(n)$

$T(n) = O(n^2)$

```java
private void update_mode() {
    MyIterator<Node<E>> iter;
    Node<E> temp;
    for (MyHeap<Node<E>> i : tree) {
        iter = i.heapIter();
        while (iter.hasNext()) {
            temp = iter.next();
            if (temp.getOccurrence() > mode.getOccurrence()) {
                mode = temp;
            }
        }
    }
}
```

$O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(n)$ $O(1)$

$T(n) = O(n)$

n = amount of Heaps
Amount of Nodes in a heap is at max 7. (<const.)

```java
public int find(E item) throws NoSuchElementException{
    for(MyHeap<Node<E>> i : tree) {
        MyIterator<Node<E>> iter = i.heapIter();
        while (iter.hasNext()) {
            Node<E> temp = iter.next();
            if (temp.getData().equals(item)) { //if the node already exists
                return temp.getOccurrence();
            }
        }
    }
    throw new NoSuchElementException("There is no such element in MyContainers.BSTHeapTree");
}
```

$O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$

$O(1)$

$T_W(n) = O(n)$ (item doesn't exist.)

$T_B(n) = O(1)$ (item is in the first index of the first heap.)

$T(n) = O(n) + O(1) = \underline{O(n)}$

```java
public String toString(){
    StringBuilder returnString = new StringBuilder();
    int amount=0;
    for(MyHeap<Node<E>> x : tree){
        returnString.append(x.toString());
        amount++;
    }
    returnString.append("Mode:").append(mode()).append(" Occurrence:").append(mode.occurrence).append("\n").append("Amonut of heaps:").append(amount);
    return returnString.toString();
}
```

n = Amount of heaps

$O(n)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$

↳ 7 elements

$T(n) = O(n)$