PART 1 METHODS TIME COMPLEXITIY ANALYSIS :

```
protected void swap( int i , int  j ) {

    if ( i < 0 || j < 0 || (i > theData.size() - 1)
            || (j > theData.size() - 1) ) {
        throw new IndexOutOfBoundsException();
    }

    E tempData = theData.get(i);
    theData.set( i , theData.get(j) );
    theData.set( j , tempData );

}
```

Θ(1) Θ(1) Θ(1) Θ(1)

```
protected int compare( E left  , E right ) {

    if( comparator != null ) {
        return comparator.compare( left ,right );
    } else {
        return left.compareTo( right );
    }

}
```

O(n log n) O(1) O(n log n)

```
protected void heapifyUp() {

    int index = theData.size() - 1;

    while ( hasParent( index ) && ( compare( parent( index ) , theData.get( index ) ) < 0 ) ) {
        swap( getParentIndex( index ) , index );
        index = getParentIndex( index );
    }

}
```

Θ(1) Θ(1) Θ(1) Θ(1) Θ(n)

```java
protected void heapifyDown() { heapifyDown( index: 0 ); }

/**
 * Heapify down helper method.
 * @param index index
 */
protected void heapifyDown( int index ) {

    while ( hasLeftChild( index ) ) {
        int smallerChildIndex = getLeftChildIndex( index );

        if( hasRightChild( index ) && ( compare(rightChild( index ), leftChild( index )) > 0 )  ) {
            smallerChildIndex = getRightChildIndex( index );
        }

        if( compare( theData.get( index ), theData.get( smallerChildIndex )) > 0 ) {
            break;
        } else {
            swap( index , smallerChildIndex );
        }

        index = smallerChildIndex;
    }
}
```

Θ(1) Θ(1) Θ(1) Θ(1)

$T_b = \Theta(1)$

$T_w = \Theta(n)$

$\Theta(\log n)$ → generally

$O(n)$

```java
@Override
public E peek() {
    if( theData.size() == 0 ) {
        throw new IllegalStateException();
    }

    return theData.get( 0 );
}
```

Θ(1) Θ(1) Θ(1)

Θ(1)

```java
@Override
public E poll() {
    if( theData.size() == 0 ) {
        throw new IllegalStateException();
    }

    E item = theData.get( 0 );
    theData.set( 0 , theData.get( theData.size() - 1 ));
    theData.remove( theData.size() - 1 );
    heapifyDown();

    return item;
}
```

Θ(1)

Θ(1)

Θ(1)

O(n)

O(n)

```java
@Override
public E find( E  item ) {

    for (E theDatum : theData) {
        if ( theDatum.compareTo( item ) == 0 ) {
            return theDatum;
        }
    }

    return null;
}
```

Θ(1)

O(n)

```java
@Override
public void add( E item ) {
    theData.add( item );
    heapifyUp();
}
```

Amortized O(1)

O(n)

O(n)

```java
@Override
public boolean remove(E item) {

    if( theData.size() == 0 ) {
        throw new IllegalStateException();
    }

    int index = theData.indexOf( find( item ) );

    if( index == -1 ) {
        return false;
    }

    swap( index , theData.size() - 1 );
    theData.remove( theData.size() - 1 );
    heapifyDown( index );

    return true;
}
```

```java
public void maxHeapify(int[] arr, int n,int i)
{
    // Find largest of node and its children
    if (i >= n) {
        return;
    }
    int l = i * 2 + 1;
    int r = i * 2 + 2;
    int max;
    if (l < n && arr[l] > arr[i]) {
        max = l;
    }
    else
        max = i;
    if (r < n && arr[r] > arr[max]) {
        max = r;
    }

    // Put maximum value at root and
    // recur for the child with the
    // maximum value
    if (max != i) {
        int temp = arr[max];
        arr[max] = arr[i];
        arr[i] = temp;
        maxHeapify(arr, n, max);
    }
}
```

```java
// Merges max heaps a[] and b[] into merged[]
public void mergeHeaps(int[] arr, int[] a,int[] b, int n, int m)
{
    for (int i = 0; i < n; i++) {
        arr[i] = a[i];
    }
    for (int i = 0; i < m; i++) {
        arr[n + i] = b[i];
    }
    n = n + m;

    // Builds a max heap of given arr[0..n-1]
    for (int i = n / 2 - 1; i >= 0; i--) {
        maxHeapify(arr, n, i);
    }
}
```

Θ(n)

Θ(n)

Θ(n.logn)

Θ(n logn)

```java
public E kthLargestElement(int k){
    E temp=null;
    ArrayList<E> arrayList = new ArrayList<>();

    for(int i=0;i<k;i++){
        arrayList.add(poll());

    }

    temp = arrayList.remove(arrayList.size()-1);

    for(int i=0;i<arrayList.size();i++){
        add(arrayList.get(i));
    }

    return temp;
}
```

n

O(n)

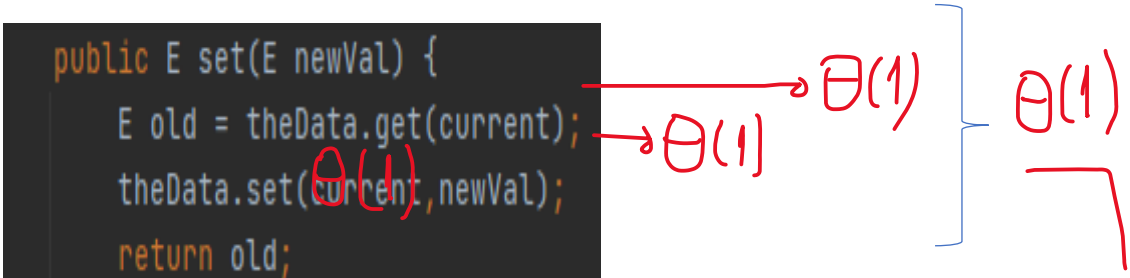
n

O(n)

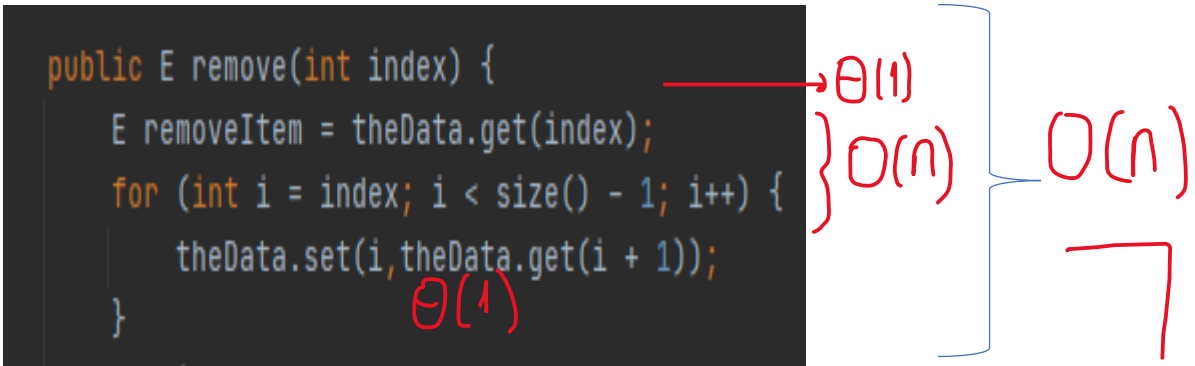
Θ(1)

O(n.logn)

```java
public E set(E newVal) {
    E old = theData.get(current);
    theData.set(current,newVal);
    return old;
}
```

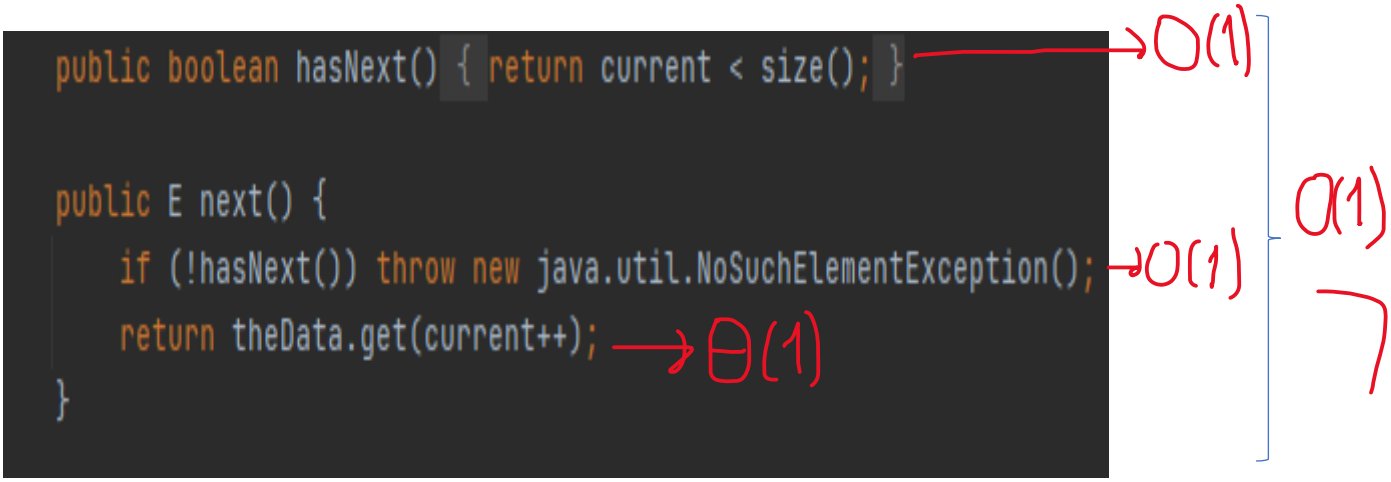



```java
public E remove(int index) {
    E removeItem = theData.get(index);
    for (int i = index; i < size() - 1; i++) {
        theData.set(i,theData.get(i + 1));
    }
    theSize--;
    return removeItem;
}
```

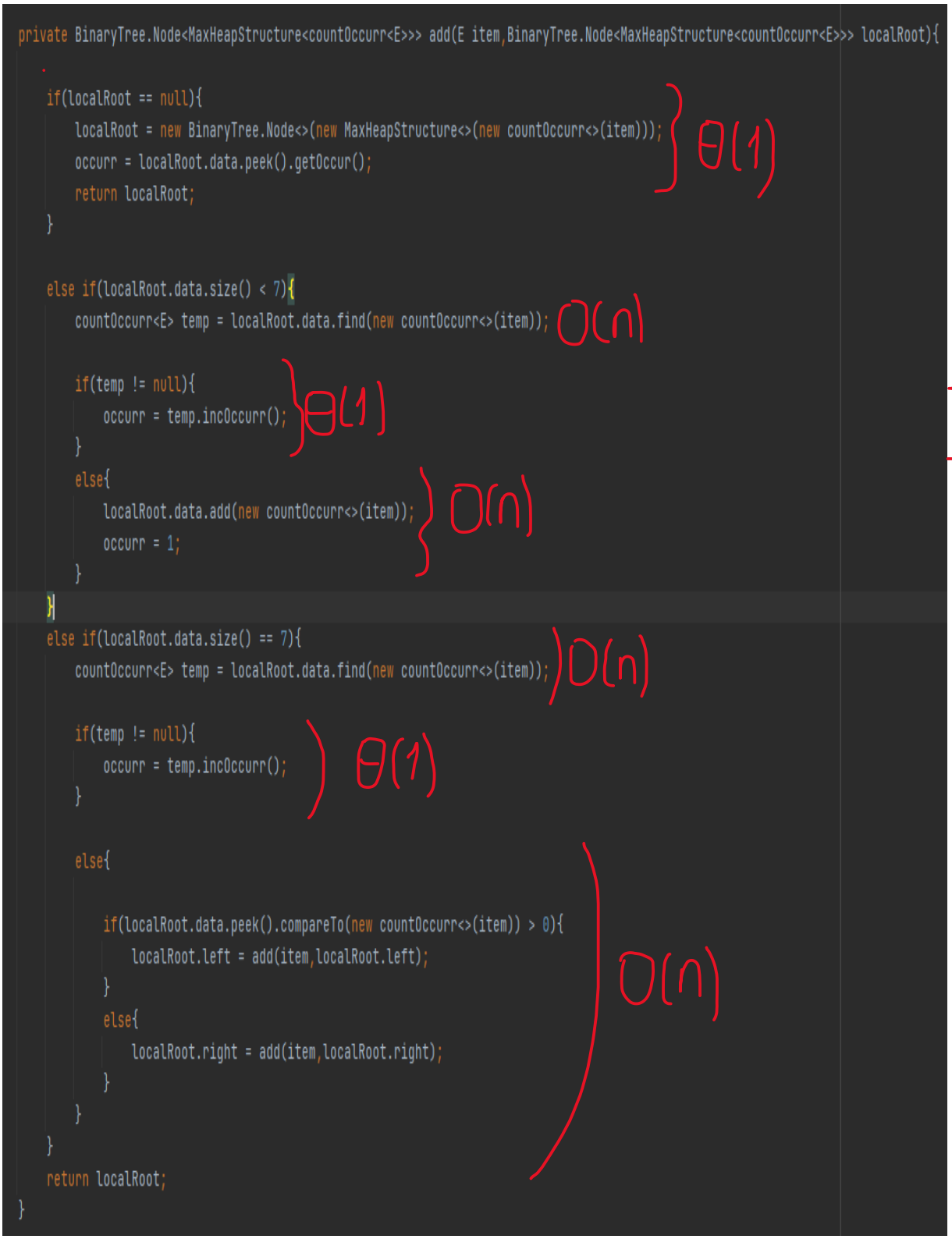



```java
public boolean hasNext() { return current < size(); }

public E next() {
    if (!hasNext()) throw new java.util.NoSuchElementException();
    return theData.get(current++);
}
```



PART2 METHODS TIME COMPLEXITIY:



```java
public int add (E item){

    theData.root = add(item,theData.root);

    return occurr;

}
```

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

```java
public int find(E target) {
    return find(theData.root,target);
}
```

O(n)

```java
private int find( BinaryTree.Node<MaxHeapStructure<countOccurr<E>>> localRoot, E target) {
    if (localRoot == null) throw new NoSuchElementException();

    countOccurr<E> temp = localRoot.data.find(new countOccurr<>(target));
    if (temp != null){
      return temp.getOccur();
    }

    else if (target.compareTo(localRoot.data.peek().getValue())< 0){
        return find(localRoot.left, target);
    }

    else
        return find(localRoot.right, target);
}

@Override
public String toString(){
    return theData.toString();
}
```

O(1) O(n) O(1) O(n) O(n) O(log n)

```java
@Override
public String toString(){
    return theData.toString();
}
```

O(n)

```java
@SuppressWarnings("unchecked")
private countOccurr<E> find_mode(BinaryTree.Node<MaxHeapStructure<countOccurr<E>>> localRoot, countOccurr<E> mode) {

    if (localRoot == null) {
        return new countOccurr<>(null,0);
    }
    else {
        countOccurr<E> temp;
        countOccurr<E> theOneToCompare = localRoot.data.peek();
        Iterator<E> iter = (Iterator<E>) localRoot.data.iterator();

        while(iter.hasNext()){
            temp = (countOccurr<E>) iter.next();
            if(temp.getOccur() > theOneToCompare.getOccur()){
                theOneToCompare = temp;

            }
        }

        countOccurr<E> left = find_mode(localRoot.left, mode);
        countOccurr<E> right = find_mode(localRoot.right, mode);
        mode = theOneToCompare;

        if(mode.getOccur() < left.getOccur()){
            mode = left;
        }
        if(mode.getOccur() < right.getOccur()){
            mode = right;
        }


        return mode;
    }
}
```

θ(1) θ(1) O(1) O(1) θ(1) θ(1) θ(1) θ(n)

```java
public E find_mode(){
    //System.out.println(find_mode(theData.root,new countOccurr<>(null,0)).getOccur());
    return find_mode(theData.root,new countOccurr<>(null,0)).getValue();

}
```

θ(n)

```java
public countOccurr<E> minimumElement(BinaryTree.Node<MaxHeapStructure<countOccurr<E>>> localRoot){
    if(localRoot.left == null){
        return localRoot.data.kthLargestElement(localRoot.data.size());
    }

    else{
        return minimumElement(localRoot.left);
    }

}
```

→ $O(n \log n)$

→ $\Theta(n)$

$O(n \log n)$

```java
private BinaryTree.Node<MaxHeapStructure<countOccurr<E>>> remove(E item,BinaryTree.Node<MaxHeapStructure<countOccurr<E>>> localRoot){

   if(localRoot == null){
      occurr = 0;
      return localRoot;
   }
      countOccurr<E> temp = localRoot.data.find(new countOccurr<>(item));

      if(temp == null){

            int comp = item.compareTo(localRoot.data.peek().getValue()); //try find to data
            if(comp < 0){
               localRoot.left = remove(item, localRoot.left);
               return localRoot;
            }
            else{
               localRoot.right = remove(item, localRoot.right);
               return localRoot;
            }
      }
      else{
         if(temp.getOccur() == 1){//data in this heap

            localRoot.data.remove(temp);
            countOccurr<E> min = minimumElement(localRoot);
            localRoot.data.add(min);
            occurr = 0;
         }
         else{
            occurr = temp.decOccurr();
         }
      }
      if(localRoot.data.size() == 1){

         theData.remove(localRoot.data);
      }
   return localRoot;

}
```

$T_b = \Omega(n)$

$T_w = \Theta(\log n)$

→ $T(n) = O(\log n)$

```java
public int remove (E item){

    theData.root = remove(item,theData.root);

    return occurr;

}
```

→ O(logn)

```java
public int compareTo(countOccurr<E> o){

    return this.value.compareTo(o.getValue());
}
```

→ O(1)

```java
public E getValue() { return value; }

public int getOccur() { return occurr; }
```

→ Θ(1)

```java
public int incOccurr(){

    return ++occurr;

}
```

) Θ(1)

```java
public int decOccurr(){

    return --occurr;

}
```

) Θ(1)

Θ(1)