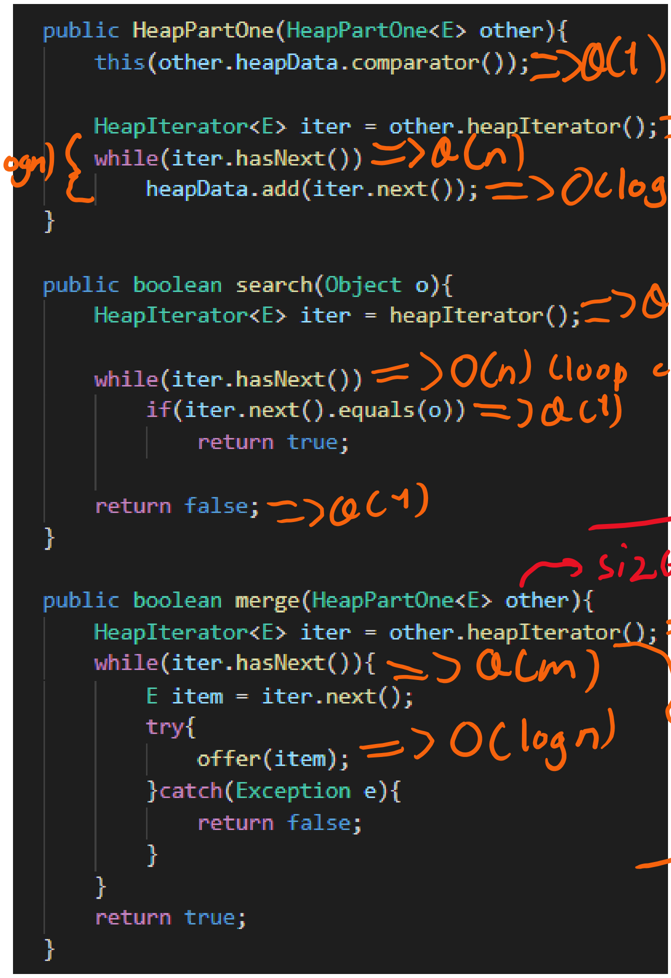
Yunus Emre
Öztürk
1901042619

n = input size

PART 3



```
public HeapPartOne(HeapPartOne<E> other){
    this(other.heapData.comparator()); => O(1)

    HeapIterator<E> iter = other.heapIterator(); => O(1)
O(nlogn) {  while(iter.hasNext()) => O(n)
        heapData.add(iter.next()); => O(logn)
}
```

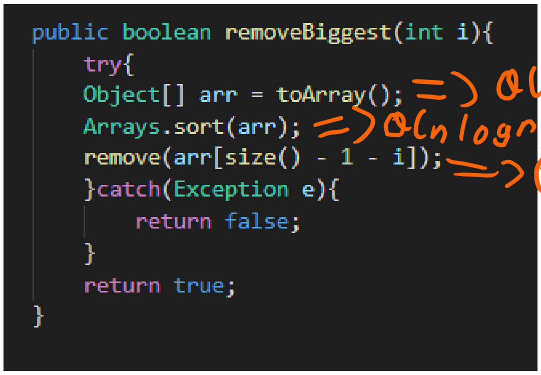
O(1)} copy const. O(nlogn)

```
public boolean search(Object o){
    HeapIterator<E> iter = heapIterator(); => O(1)

    while(iter.hasNext()) => O(n) (loop can break)
        if(iter.next().equals(o)) => O(1)
            return true;

    return false; => O(1)
}
```

O(n) Search: O(n)

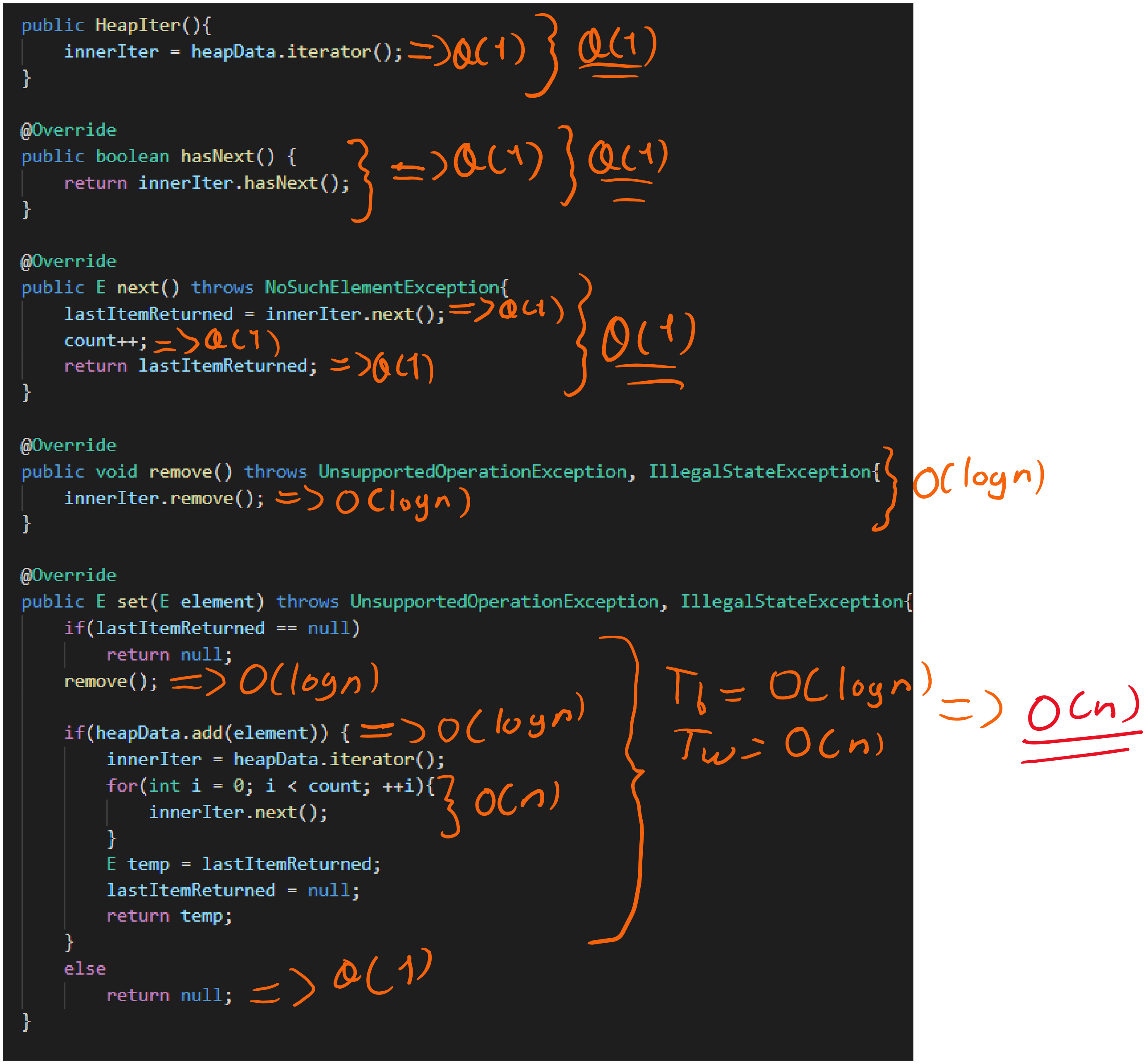
```
                                   → size == m
public boolean merge(HeapPartOne<E> other){
    HeapIterator<E> iter = other.heapIterator(); => O(1
    while(iter.hasNext()){ => O(m)
        E item = iter.next();
        try{
            offer(item); => O(logn)
        }catch(Exception e){
            return false;
        }
    }
    return true;
}
```

O(mlogn) Merge: O(m logn)



```
public boolean removeBiggest(int i){
    try{
        Object[] arr = toArray(); => O(n)
        Arrays.sort(arr); => O(nlogn)
        remove(arr[size() - 1 - i]); => O(logn)
    }catch(Exception e){
        return false;
    }
    return true;
}
```

Remove Biggest: O(nlogn)



```java
public HeapIter(){
    innerIter = heapData.iterator();
}

@Override
public boolean hasNext() {
    return innerIter.hasNext();
}

@Override
public E next() throws NoSuchElementException{
    lastItemReturned = innerIter.next();
    count++;
    return lastItemReturned;
}

@Override
public void remove() throws UnsupportedOperationException, IllegalStateException{
    innerIter.remove();
}

@Override
public E set(E element) throws UnsupportedOperationException, IllegalStateException{
    if(lastItemReturned == null)
        return null;
    remove();

    if(heapData.add(element)) {
        innerIter = heapData.iterator();
        for(int i = 0; i < count; ++i){
            innerIter.next();
        }
        E temp = lastItemReturned;
        lastItemReturned = null;
        return temp;
    }
    else
        return null;
}
```

innerIter = heapData.iterator(); => $O(1)$ } $\underline{O(1)}$

hasNext() } => $O(1)$ } $\underline{O(1)}$

lastItemReturned = innerIter.next(); => $O(1)$

count++; => $O(1)$

return lastItemReturned; => $O(1)$

next() } $\underline{O(1)}$

innerIter.remove(); => $O(\log n)$

remove() } $O(\log n)$

remove(); => $O(\log n)$

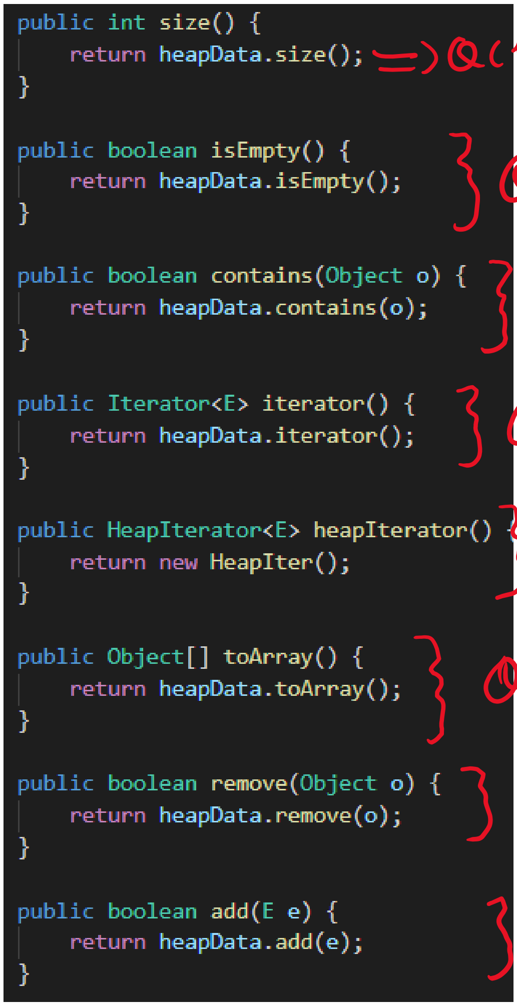
if(heapData.add(element)) { => $O(\log n)$

for loop } $O(n)$

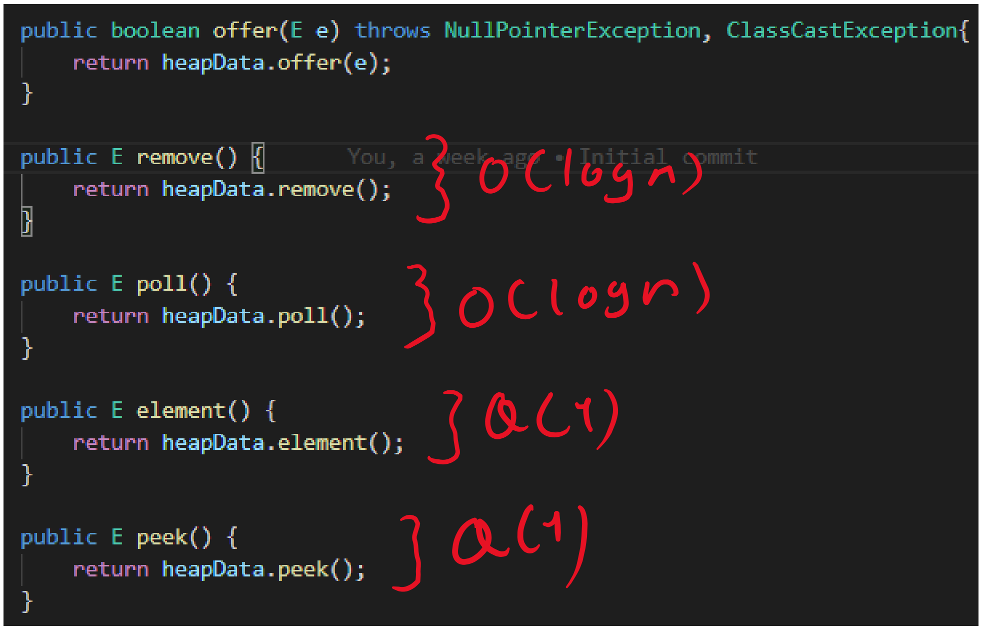
else return null; => $O(1)$

$T_b = O(\log n)$ => $\underline{O(n)}$

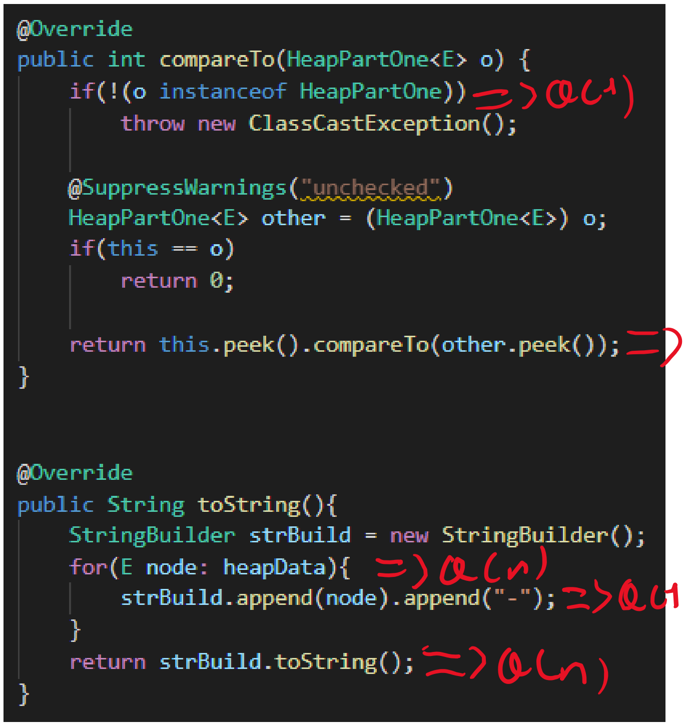
$T_w = O(n)$



```
public int size() {
    return heapData.size();       => O(1) } O(1)
}

public boolean isEmpty() {
    return heapData.isEmpty();    } O(1)
}

public boolean contains(Object o) {
    return heapData.contains(o);  } O(n)
}

public Iterator<E> iterator() {
    return heapData.iterator();   } O(1)
}

public HeapIterator<E> heapIterator() {
    return new HeapIter();        } O(1)
}

public Object[] toArray() {
    return heapData.toArray();    } O(n)
}

public boolean remove(Object o) {
    return heapData.remove(o);    } O(n log n)
}

public boolean add(E e) {
    return heapData.add(e);       } O(log n)
}
```



```
public boolean offer(E e) throws NullPointerException, ClassCastException{   } O(log n)
    return heapData.offer(e);
}

public E remove() {        You, a week ago • Initial commit
    return heapData.remove();   } O(log n)
}

public E poll() {
    return heapData.poll();     } O(log n)
}

public E element() {
    return heapData.element();  } O(1)
}

public E peek() {
    return heapData.peek();     } O(1)
}
```

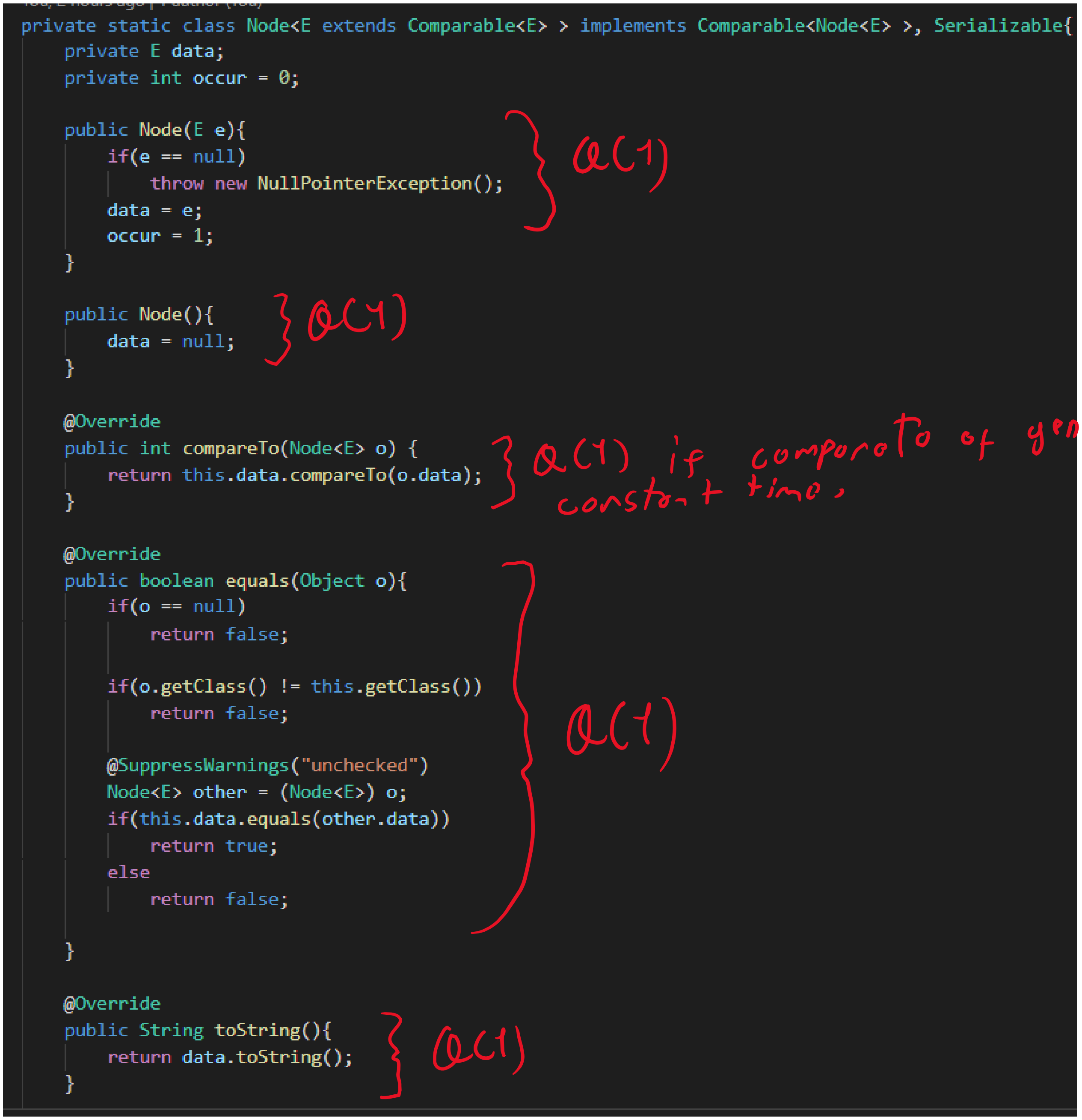


```
@Override
public int compareTo(HeapPartOne<E> o) {
    if(!(o instanceof HeapPartOne))  => Θ(1)
        throw new ClassCastException();

    @SuppressWarnings("unchecked")
    HeapPartOne<E> other = (HeapPartOne<E>) o;
    if(this == o)
        return 0;

    return this.peek().compareTo(other.peek());  => Θ(1)
}


@Override
public String toString(){
    StringBuilder strBuild = new StringBuilder();
    for(E node: heapData){  => Θ(n)
        strBuild.append(node).append("-");  => Θ(1
    }
    return strBuild.toString();  => Θ(n)
}
```

Θ(1) if compareTo of generic type is constant, otherwise O(m)

<u>Θ(n)</u>

BSTHeapTree



```
You, 2 hours ago | 1 author (You)
private static class Node<E extends Comparable<E> > implements Comparable<Node<E> >, Serializable{
    private E data;
    private int occur = 0;

    public Node(E e){
        if(e == null)
            throw new NullPointerException();
        data = e;
        occur = 1;
    }
```

$\Theta(1)$

```
    public Node(){
        data = null;
    }
```

$\Theta(1)$

```
    @Override
    public int compareTo(Node<E> o) {
        return this.data.compareTo(o.data);
    }
```

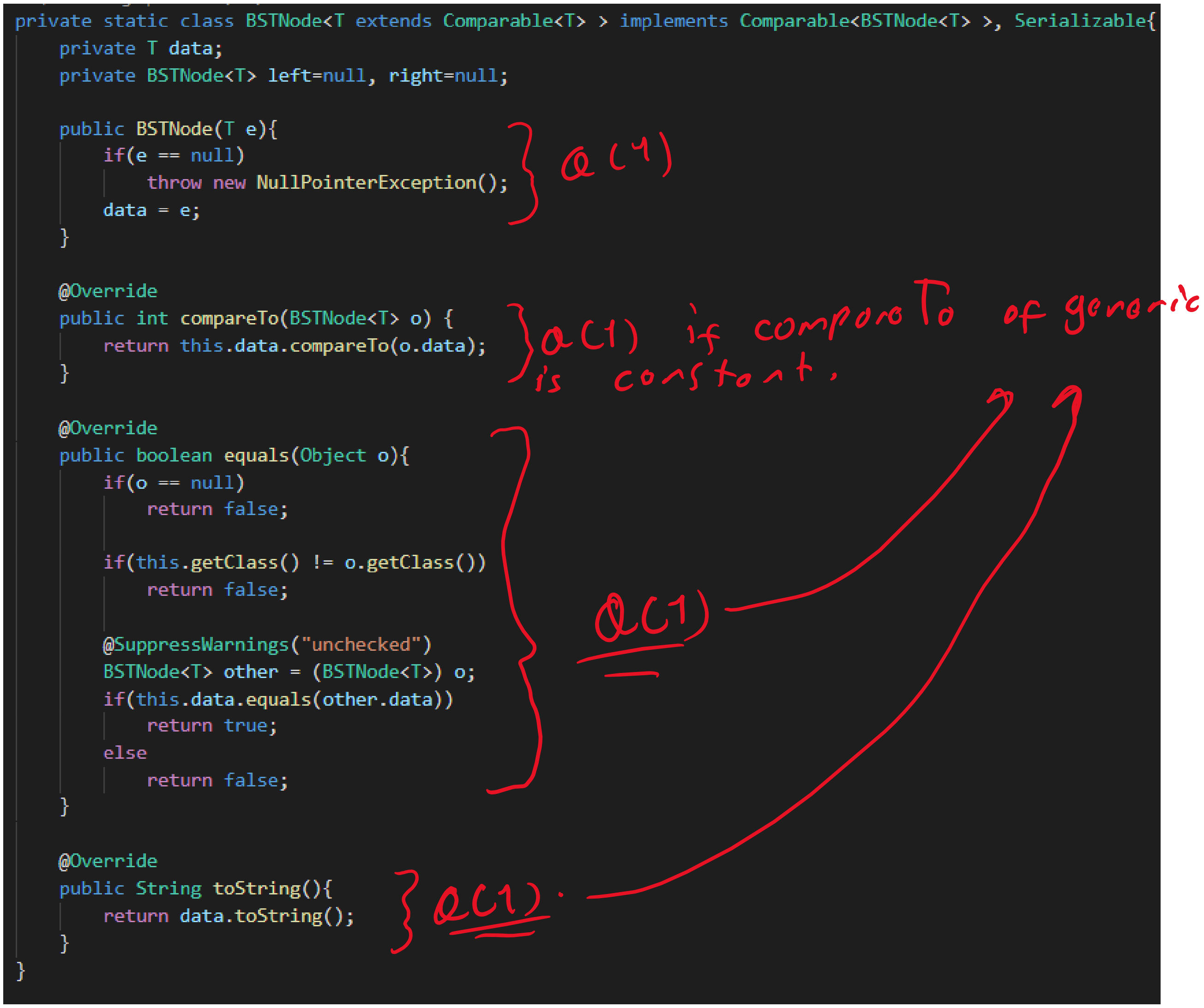
$\Theta(1)$ if compareTo of generic constant time,

```
    @Override
    public boolean equals(Object o){
        if(o == null)
            return false;

        if(o.getClass() != this.getClass())
            return false;

        @SuppressWarnings("unchecked")
        Node<E> other = (Node<E>) o;
        if(this.data.equals(other.data))
            return true;
        else
            return false;

    }
```

$\Theta(1)$

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

$\Theta(1)$



```java
private static class BSTNode<T extends Comparable<T> > implements Comparable<BSTNode<T> >, Serializable{
    private T data;
    private BSTNode<T> left=null, right=null;

    public BSTNode(T e){
        if(e == null)
            throw new NullPointerException();
        data = e;
    }
```

Θ(1)

```java
    @Override
    public int compareTo(BSTNode<T> o) {
        return this.data.compareTo(o.data);
    }
```

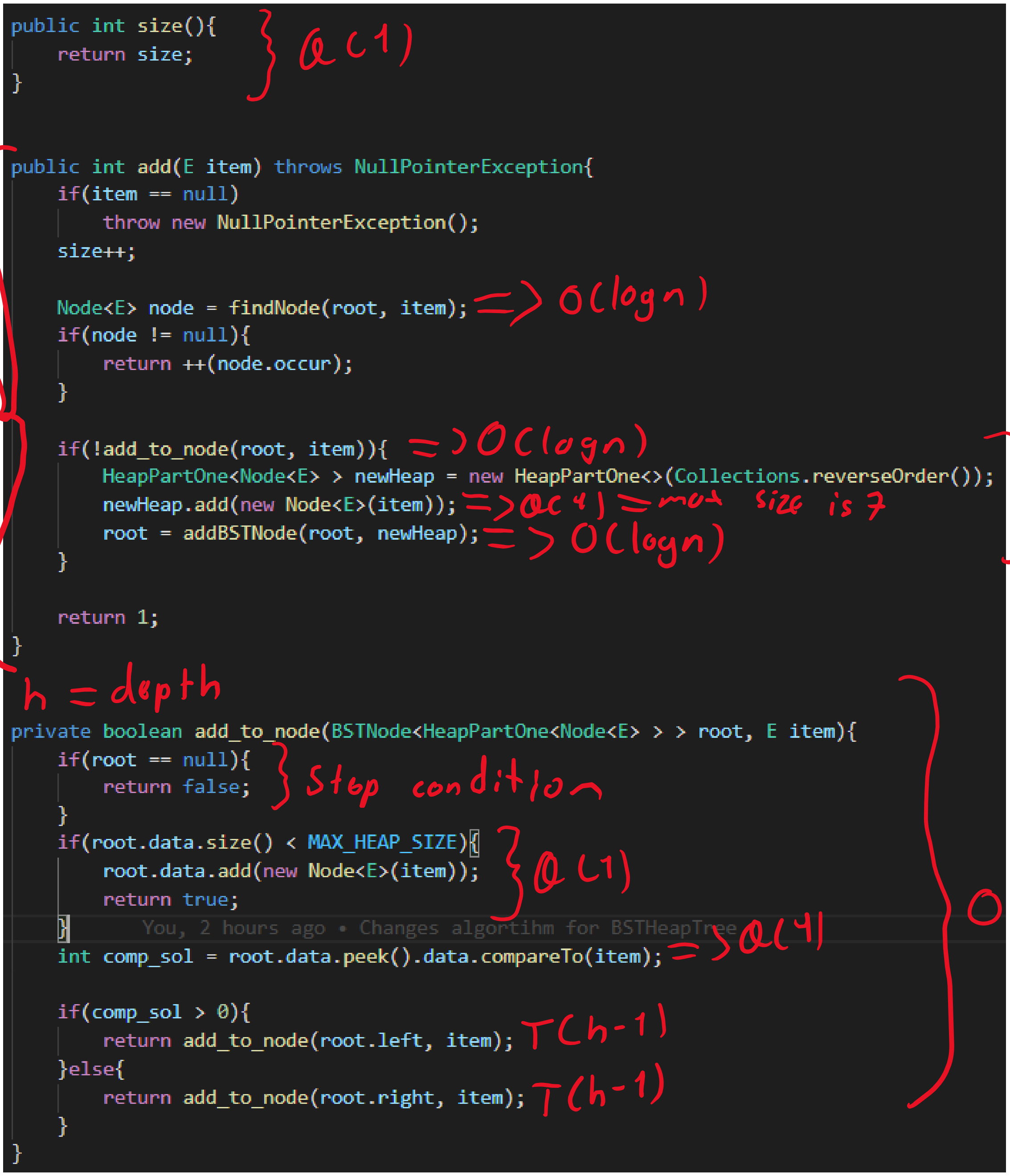
Θ(1) if compareTo of generic is constant.

```java
    @Override
    public boolean equals(Object o){
        if(o == null)
            return false;

        if(this.getClass() != o.getClass())
            return false;

        @SuppressWarnings("unchecked")
        BSTNode<T> other = (BSTNode<T>) o;
        if(this.data.equals(other.data))
            return true;
        else
            return false;
    }
```

Θ(1)

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

Θ(1).



```
public int size(){                                   } Θ(1)
    return size;
}


public int add(E item) throws NullPointerException{
    if(item == null)
        throw new NullPointerException();
    size++;

    Node<E> node = findNode(root, item);  => O(logn)
    if(node != null){
        return ++(node.occur);
    }

    if(!add_to_node(root, item)){  => O(logn)
        HeapPartOne<Node<E> > newHeap = new HeapPartOne<>(Collections.reverseOrder());
        newHeap.add(new Node<E>(item));  => Θ(1) = max size is 7
        root = addBSTNode(root, newHeap);  => O(logn)
    }

    return 1;
}
```

O(logn)

O(logn)

h = depth

```
private boolean add_to_node(BSTNode<HeapPartOne<Node<E> > > root, E item){
    if(root == null){
        return false;                 } Stop condition
    }
    if(root.data.size() < MAX_HEAP_SIZE){
        root.data.add(new Node<E>(item));   } Θ(1)
        return true;
    }        You, 2 hours ago • Changes algortihm for BSTHeapTree
    int comp_sol = root.data.peek().data.compareTo(item);  => Θ(1)

    if(comp_sol > 0){
        return add_to_node(root.left, item);  T(h-1)
    }else{
        return add_to_node(root.right, item);  T(h-1)
    }
}
```

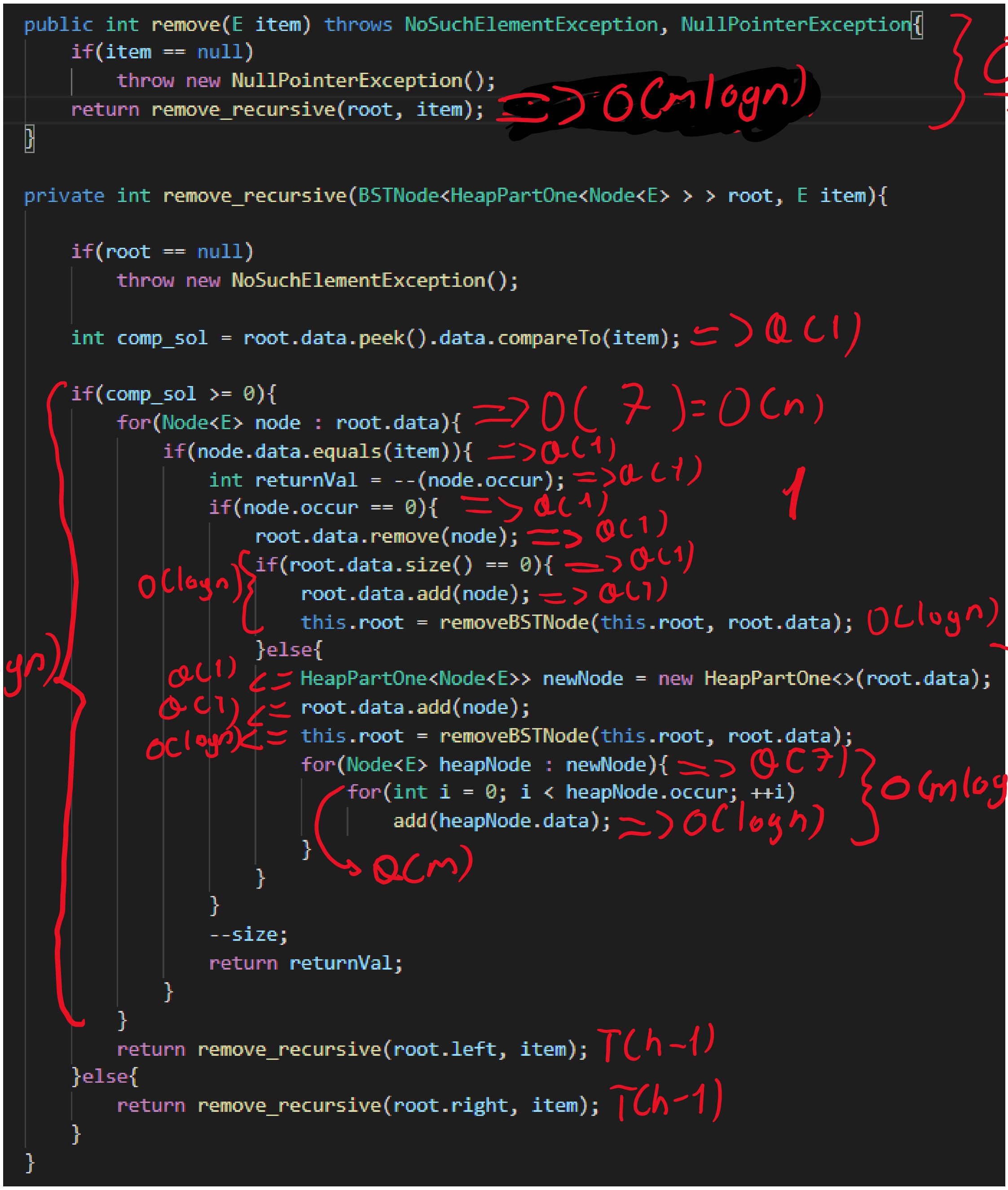
O(logn)

x $T(h) = T(h-1) + \Theta(1)$

$T(h) = h\,\Theta(1) = \Theta(h)$

h = logn if tree is complete

$$\underline{\underline{T(n) = O(\log n)}}$$



```java
public int remove(E item) throws NoSuchElementException, NullPointerException{
    if(item == null)
        throw new NullPointerException();
    return remove_recursive(root, item);
}

private int remove_recursive(BSTNode<HeapPartOne<Node<E> > > root, E item){

    if(root == null)
        throw new NoSuchElementException();

    int comp_sol = root.data.peek().data.compareTo(item);

    if(comp_sol >= 0){
        for(Node<E> node : root.data){
            if(node.data.equals(item)){
                int returnVal = --(node.occur);
                if(node.occur == 0){
                    root.data.remove(node);
                    if(root.data.size() == 0){
                        root.data.add(node);
                        this.root = removeBSTNode(this.root, root.data);
                    }else{
                        HeapPartOne<Node<E>> newNode = new HeapPartOne<>(root.data);
                        root.data.add(node);
                        this.root = removeBSTNode(this.root, root.data);
                        for(Node<E> heapNode : newNode){
                            for(int i = 0; i < heapNode.occur; ++i)
                                add(heapNode.data);
                        }
                    }
                }
                --size;
                return returnVal;
            }
        }
        return remove_recursive(root.left, item);
    }else{
        return remove_recursive(root.right, item);
    }
}
```

$O(m\log n)$

$\Rightarrow O(m\log n)$

$\Rightarrow O(1)$

$\Rightarrow O(\ 7\ ) = O(n)$

$\Rightarrow O(1)$

$\Rightarrow O(1)$

1

$\Rightarrow O(1)$

$\Rightarrow O(1)$

$O(\log n)$

$\Rightarrow O(1)$

$\Rightarrow O(1)$

$O(\log n)$

$O(m\ \log n)$

$O(1) \Leftarrow$

$O(1) \Leftarrow$

$O(\log n) \Leftarrow$

$\Rightarrow O(7)$

$O(m\log n)$

$O(m\log n)$

$\Rightarrow O(\log n)$

$O(m)$

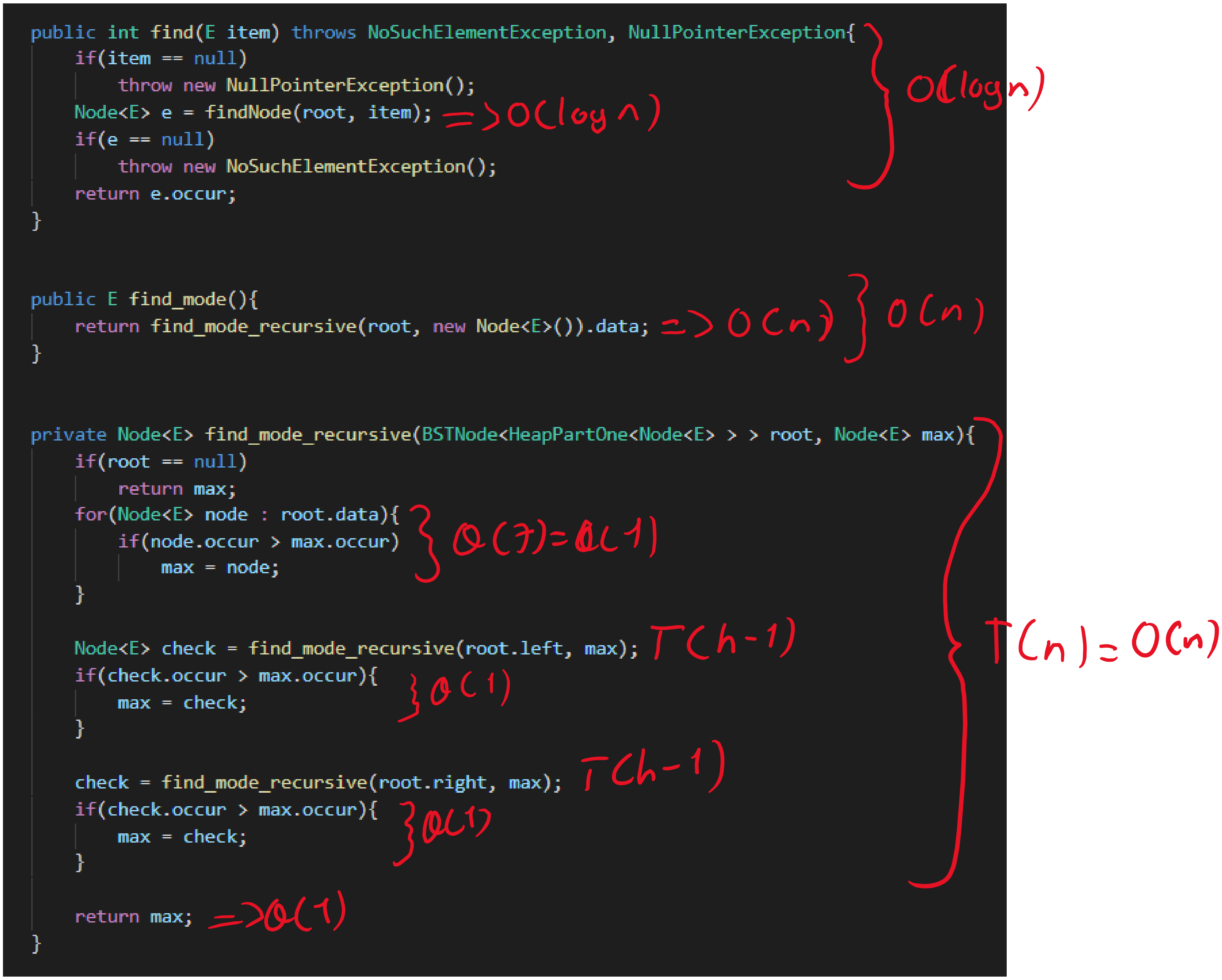
$T(h-1)$

$T(h-1)$

$T(h) = T(h-1) + O(1)$

$T(h) = h\,O(1) + O(m\log n) \simeq O(m\ \log n)$

h is logn if tree is complete

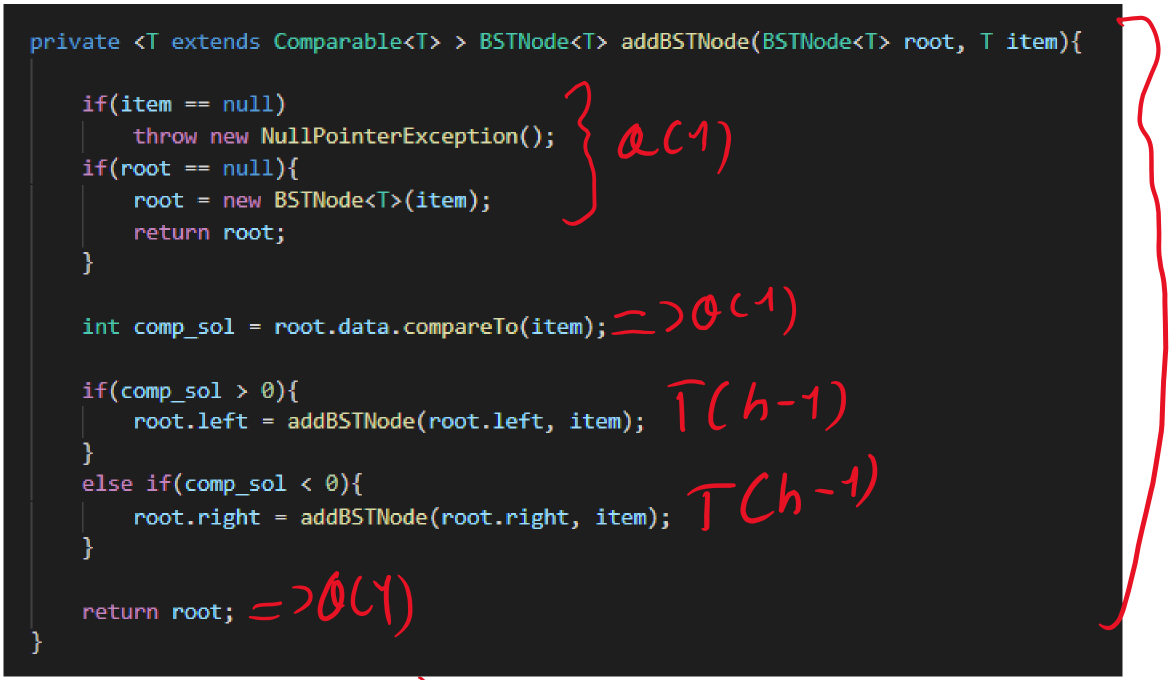


```
public int find(E item) throws NoSuchElementException, NullPointerException{
    if(item == null)
        throw new NullPointerException();
    Node<E> e = findNode(root, item);
    if(e == null)
        throw new NoSuchElementException();
    return e.occur;
}


public E find_mode(){
    return find_mode_recursive(root, new Node<E>()).data;
}


private Node<E> find_mode_recursive(BSTNode<HeapPartOne<Node<E> > > root, Node<E> max){
    if(root == null)
        return max;
    for(Node<E> node : root.data){
        if(node.occur > max.occur)
            max = node;
    }

    Node<E> check = find_mode_recursive(root.left, max);
    if(check.occur > max.occur){
        max = check;
    }

    check = find_mode_recursive(root.right, max);
    if(check.occur > max.occur){
        max = check;
    }

    return max;
}
```

$$T(h) = 2T(h-1) + \Theta(1)$$

$$T(h) = 2^h \cdot \Theta(1) = O(2^h) = O(2^{\log_2 n})$$

$$= O(n)$$



```
private <T extends Comparable<T> > BSTNode<T> addBSTNode(BSTNode<T> root, T item){

    if(item == null)
        throw new NullPointerException();
    if(root == null){
        root = new BSTNode<T>(item);
        return root;
    }

    int comp_sol = root.data.compareTo(item);

    if(comp_sol > 0){
        root.left = addBSTNode(root.left, item);
    }
    else if(comp_sol < 0){
        root.right = addBSTNode(root.right, item);
    }

    return root;
}
```

$\}\ \Theta(1)$

$\Rightarrow \Theta(1)$

$T(h-1)$

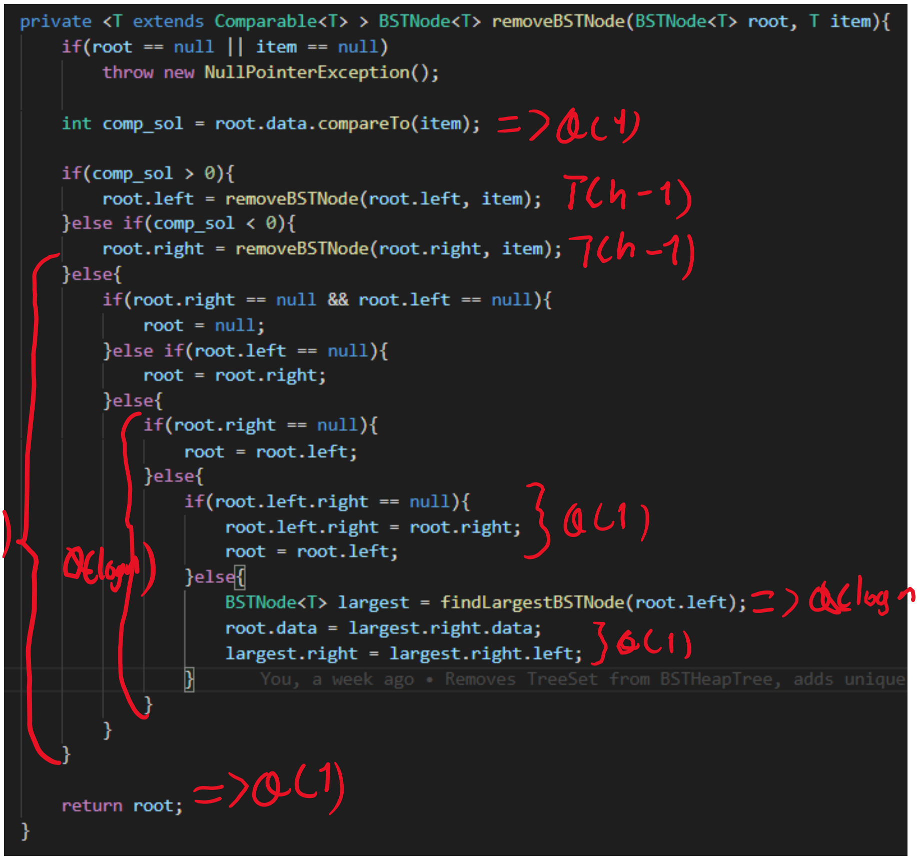
$T(h-1)$

$\Rightarrow \Theta(1)$

$O(\log n)$

$T(h) = T(h-1) + \Theta(1)$ If tree is complete.

$T(h) = h\Theta(1) \Rightarrow O(h) = O(\log n)$



```java
private <T extends Comparable<T> > BSTNode<T> removeBSTNode(BSTNode<T> root, T item){
    if(root == null || item == null)
        throw new NullPointerException();

    int comp_sol = root.data.compareTo(item);

    if(comp_sol > 0){
        root.left = removeBSTNode(root.left, item);
    }else if(comp_sol < 0){
        root.right = removeBSTNode(root.right, item);
    }else{
        if(root.right == null && root.left == null){
            root = null;
        }else if(root.left == null){
            root = root.right;
        }else{
            if(root.right == null){
                root = root.left;
            }else{
                if(root.left.right == null){
                    root.left.right = root.right;
                    root = root.left;
                }else{
                    BSTNode<T> largest = findLargestBSTNode(root.left);
                    root.data = largest.right.data;
                    largest.right = largest.right.left;
                }       You, a week ago • Removes TreeSet from BSTHeapTree, adds unique
            }
        }
    }

    return root;
}
```

$\Rightarrow \Theta(1)$

$T(h-1)$

$T(h-1)$

$\Theta(\log n)$

$\Theta(\log n)$

$\} \Theta(1)$

$\Rightarrow \Theta(\log n)$

$\} \Theta(1)$

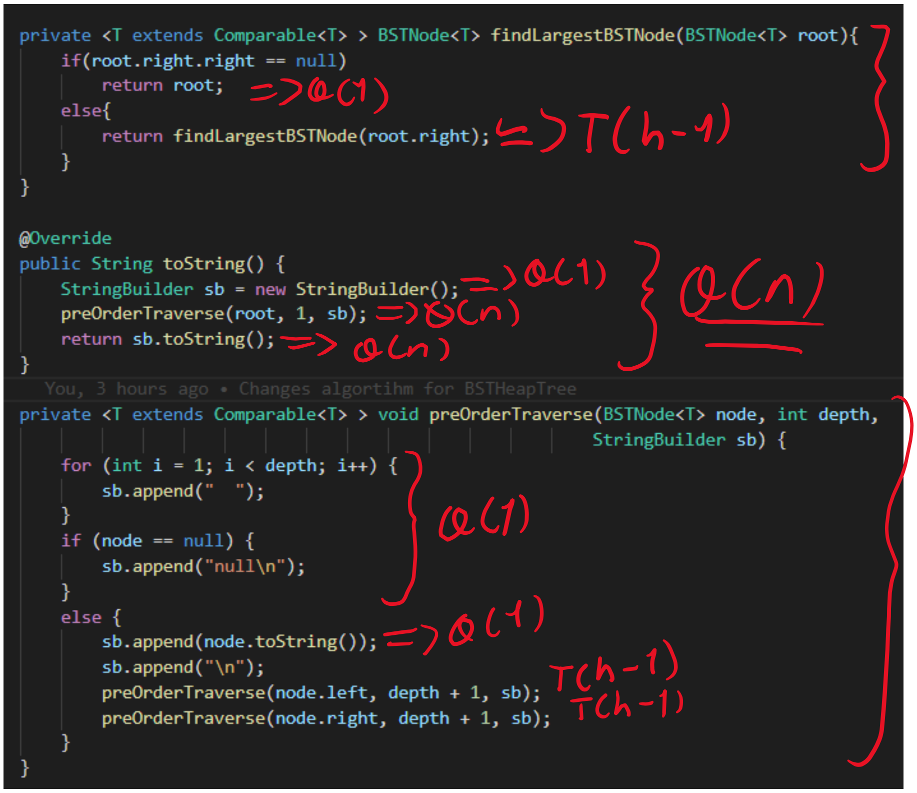
$\Rightarrow \Theta(1)$

$T(h) = T(h-1) + \Theta(1)$

$T(h) = h\,\Theta(1)$

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

If tree is complete



```java
private <T extends Comparable<T> > BSTNode<T> findLargestBSTNode(BSTNode<T> root){
    if(root.right.right == null)
        return root;
    else{
        return findLargestBSTNode(root.right);
    }
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    preOrderTraverse(root, 1, sb);
    return sb.toString();
}
  You, 3 hours ago • Changes algortihm for BSTHeapTree
private <T extends Comparable<T> > void preOrderTraverse(BSTNode<T> node, int depth,
                                                           StringBuilder sb) {
    for (int i = 1; i < depth; i++) {
        sb.append("  ");
    }
    if (node == null) {
        sb.append("null\n");
    }
    else {
        sb.append(node.toString());
        sb.append("\n");
        preOrderTraverse(node.left, depth + 1, sb);
        preOrderTraverse(node.right, depth + 1, sb);
    }
}
```

$\Rightarrow \Theta(1)$

$\Rightarrow T(h-1)$

$T(h) = T(h-1) + \Theta(1)$

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

$\Rightarrow \Theta(1)$

$\Rightarrow \Theta(n)$

$\Rightarrow \Theta(n)$

$\} \Theta(n)$

$\Theta(1)$

$\Rightarrow \Theta(1)$

$T(h-1)$

$T(h-1)$

$T(h) = 2T(h-1) + \Theta(1)$

$T(h) = 2^h\,\Theta(1)$

$= \Theta(2^{\log_2 n})$

$= \Theta(n)$