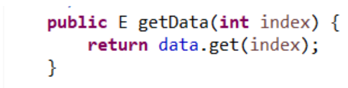
TIME COMPLEXITY

PART1:

- **GetData method:**

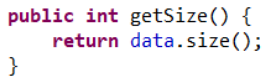
T(n) = O(1)



```java
public E getData(int index) {
    return data.get(index);
}
```

- **getSize method:**

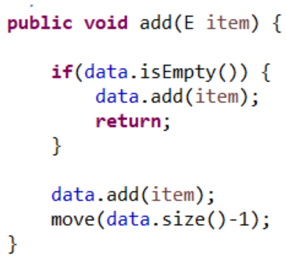
T(n) = O(1)



```java
public int getSize() {
    return data.size();
}
```

- **Add method:**

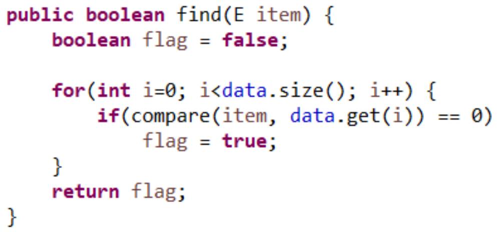
T(n) = O(1) + O(logn) = O(logn)   n: size of the heap



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

    if(data.isEmpty()) {
        data.add(item);
        return;
    }

    data.add(item);
    move(data.size()-1);
}
```

- **find method:**

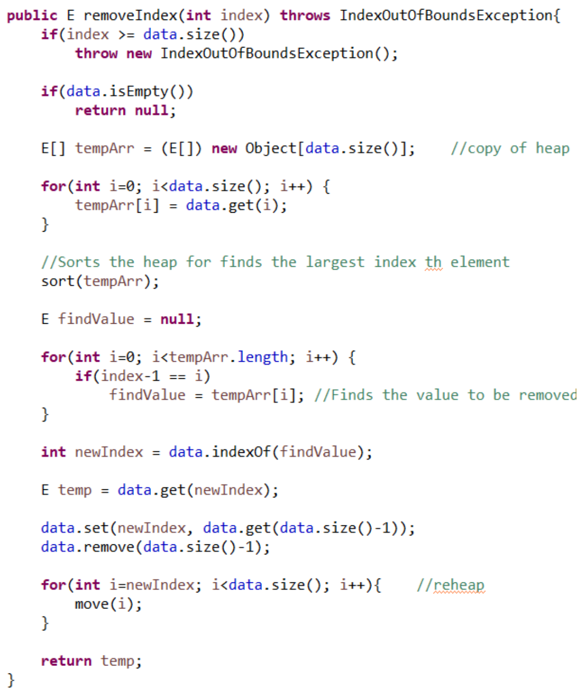
T(n) = O(n)   n: size of the heap



```java
public boolean find(E item) {
    boolean flag = false;

    for(int i=0; i<data.size(); i++) {
        if(compare(item, data.get(i)) == 0)
            flag = true;
    }
    return flag;
}
```

- **removeIndex method:**

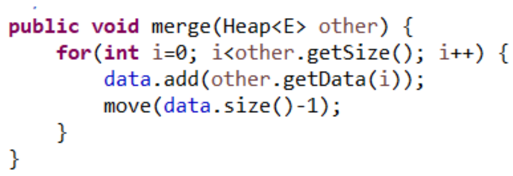
T(n) = O(n^2) n: size of the heap



```
public E removeIndex(int index) throws IndexOutOfBoundsException{
    if(index >= data.size())
        throw new IndexOutOfBoundsException();

    if(data.isEmpty())
        return null;

    E[] tempArr = (E[]) new Object[data.size()];    //copy of heap

    for(int i=0; i<data.size(); i++) {
        tempArr[i] = data.get(i);
    }

    //Sorts the heap for finds the largest index th element
    sort(tempArr);

    E findValue = null;

    for(int i=0; i<tempArr.length; i++) {
        if(index-1 == i)
            findValue = tempArr[i]; //Finds the value to be removed
    }

    int newIndex = data.indexOf(findValue);

    E temp = data.get(newIndex);

    data.set(newIndex, data.get(data.size()-1));
    data.remove(data.size()-1);

    for(int i=newIndex; i<data.size(); i++){    //reheap
        move(i);
    }

    return temp;
}
```

- **merge method:**

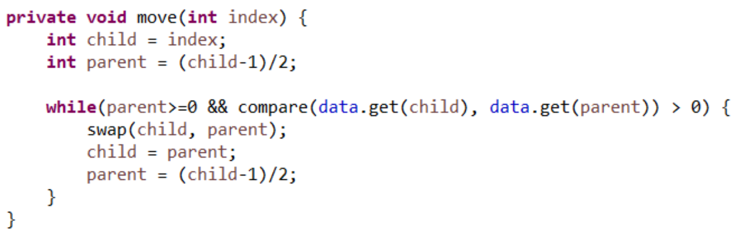
T(n) = O(m x logn) n: size of the heap, m: size of the other heap



```
public void merge(Heap<E> other) {
    for(int i=0; i<other.getSize(); i++) {
        data.add(other.getData(i));
        move(data.size()-1);
    }
}
```

- **Move method:**

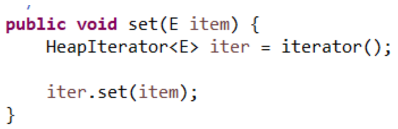
T(n) = O(logn) n: size of the heap



```
private void move(int index) {
    int child = index;
    int parent = (child-1)/2;

    while(parent>=0 && compare(data.get(child), data.get(parent)) > 0) {
        swap(child, parent);
        child = parent;
        parent = (child-1)/2;
    }
}
```

- **Set method:**

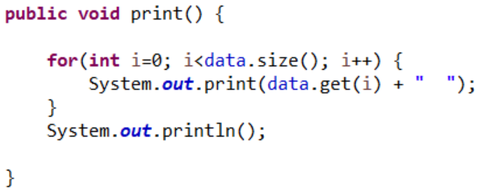
T(n) = O(n) n: size of the heap



```
public void set(E item) {
    HeapIterator<E> iter = iterator();

    iter.set(item);
}
```

- **Print method:**

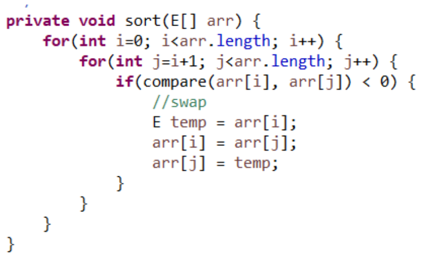
T(n) = O(n) n: size of the heap



```
public void print() {

    for(int i=0; i<data.size(); i++) {
        System.out.print(data.get(i) + "  ");
    }
    System.out.println();

}
```

- **Sort method:**

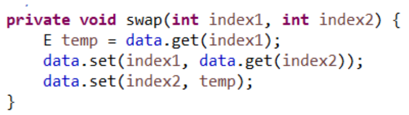
T(n) = O(n^2) n: size of the array (actually heap)



```
private void sort(E[] arr) {
    for(int i=0; i<arr.length; i++) {
        for(int j=i+1; j<arr.length; j++) {
            if(compare(arr[i], arr[j]) < 0) {
                //swap
                E temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

- **Swap method:**

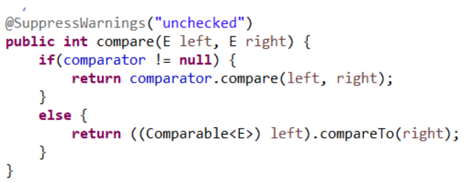
T(n) = O(1)



```
private void swap(int index1, int index2) {
    E temp = data.get(index1);
    data.set(index1, data.get(index2));
    data.set(index2, temp);
}
```

- **Compare method:**

T(n) = O(1)



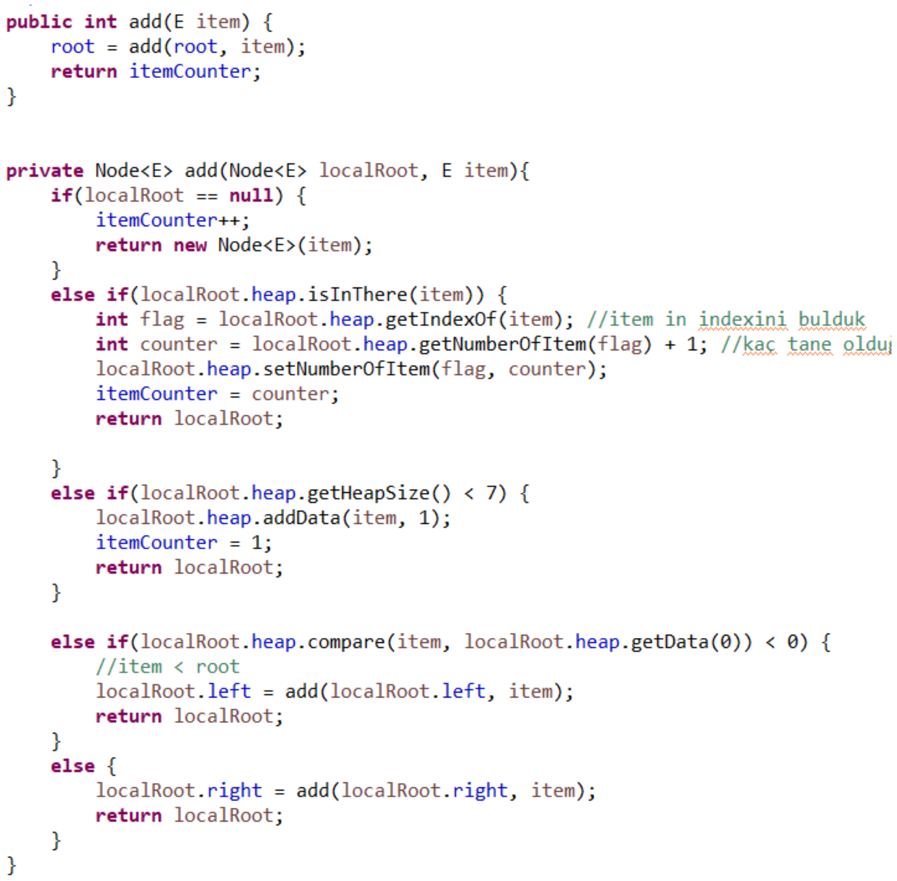
```java
@SuppressWarnings("unchecked")
public int compare(E left, E right) {
    if(comparator != null) {
        return comparator.compare(left, right);
    }
    else {
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

PART2:

1. Add:

T(n) = O(log(n) x log(m)) n: size of the heap in node, m: number of the node

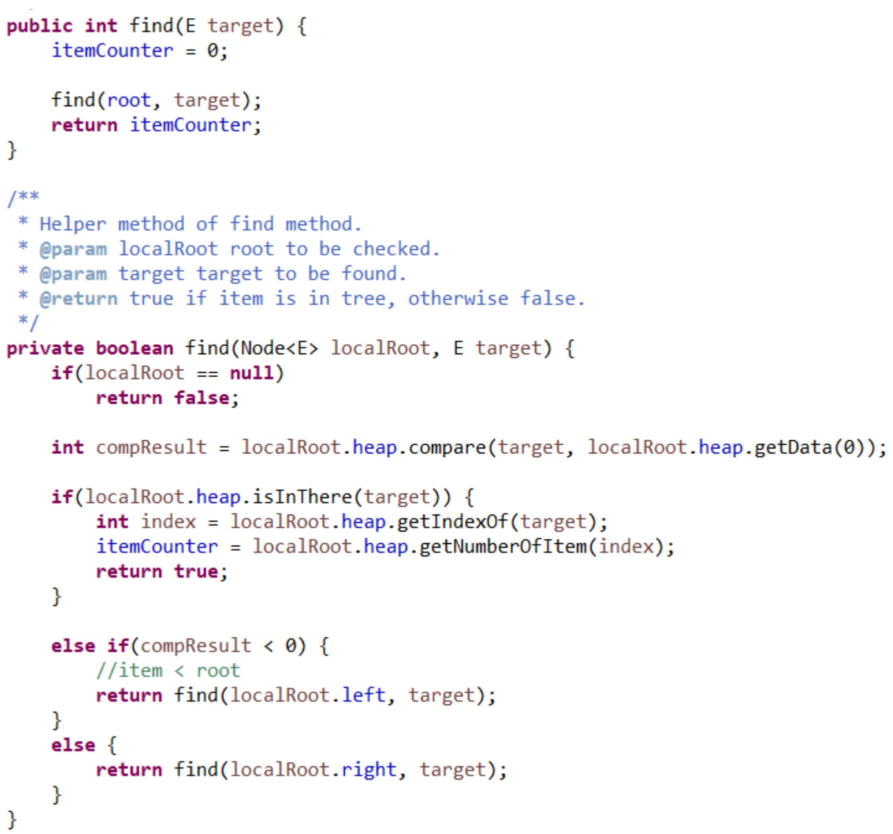
addData -> O(logn)



```
public int add(E item) {
    root = add(root, item);
    return itemCounter;
}


private Node<E> add(Node<E> localRoot, E item){
    if(localRoot == null) {
        itemCounter++;
        return new Node<E>(item);
    }
    else if(localRoot.heap.isInThere(item)) {
        int flag = localRoot.heap.getIndexOf(item); //item in indexini bulduk
        int counter = localRoot.heap.getNumberOfItem(flag) + 1; //kac tane oldu
        localRoot.heap.setNumberOfItem(flag, counter);
        itemCounter = counter;
        return localRoot;

    }
    else if(localRoot.heap.getHeapSize() < 7) {
        localRoot.heap.addData(item, 1);
        itemCounter = 1;
        return localRoot;
    }

    else if(localRoot.heap.compare(item, localRoot.heap.getData(0)) < 0) {
        //item < root
        localRoot.left = add(localRoot.left, item);
        return localRoot;
    }
    else {
        localRoot.right = add(localRoot.right, item);
        return localRoot;
    }
}
```

2. Find:

T(n) = O(n) (amortised)   n: number of the node

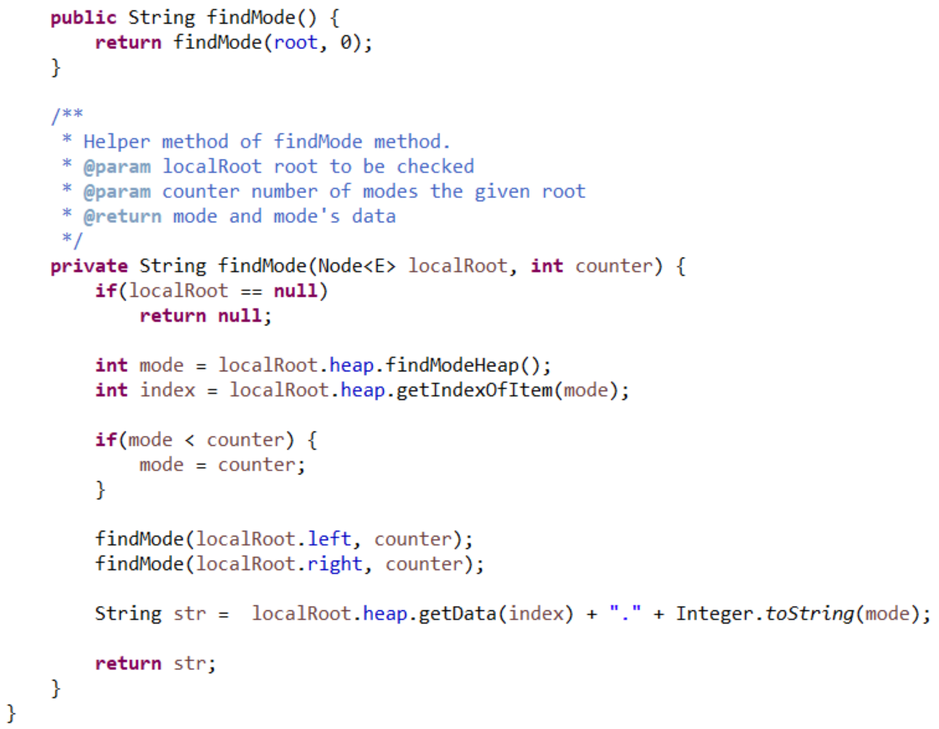


```
public int find(E target) {
    itemCounter = 0;

    find(root, target);
    return itemCounter;
}

/**
 * Helper method of find method.
 * @param localRoot root to be checked.
 * @param target target to be found.
 * @return true if item is in tree, otherwise false.
 */
private boolean find(Node<E> localRoot, E target) {
    if(localRoot == null)
        return false;

    int compResult = localRoot.heap.compare(target, localRoot.heap.getData(0));

    if(localRoot.heap.isInThere(target)) {
        int index = localRoot.heap.getIndexOf(target);
        itemCounter = localRoot.heap.getNumberOfItem(index);
        return true;
    }

    else if(compResult < 0) {
        //item < root
        return find(localRoot.left, target);
    }
    else {
        return find(localRoot.right, target);
    }
}
```

3. findMode

T(n) = O(n x m)   n: number of the node, m: size of the heap in node

findModeHeap -> O(n)



```
    public String findMode() {
        return findMode(root, 0);
    }

    /**
     * Helper method of findMode method.
     * @param localRoot root to be checked
     * @param counter number of modes the given root
     * @return mode and mode's data
     */
    private String findMode(Node<E> localRoot, int counter) {
        if(localRoot == null)
            return null;

        int mode = localRoot.heap.findModeHeap();
        int index = localRoot.heap.getIndexOfItem(mode);

        if(mode < counter) {
            mode = counter;
        }

        findMode(localRoot.left, counter);
        findMode(localRoot.right, counter);

        String str =  localRoot.heap.getData(index) + "." + Integer.toString(mode);

        return str;
    }
}
```