Data Structures II and Algorithms

Lab1

Muhammad Salah Mahmoud Osman – 41

1. Binary Heaps

1.1 Introduction

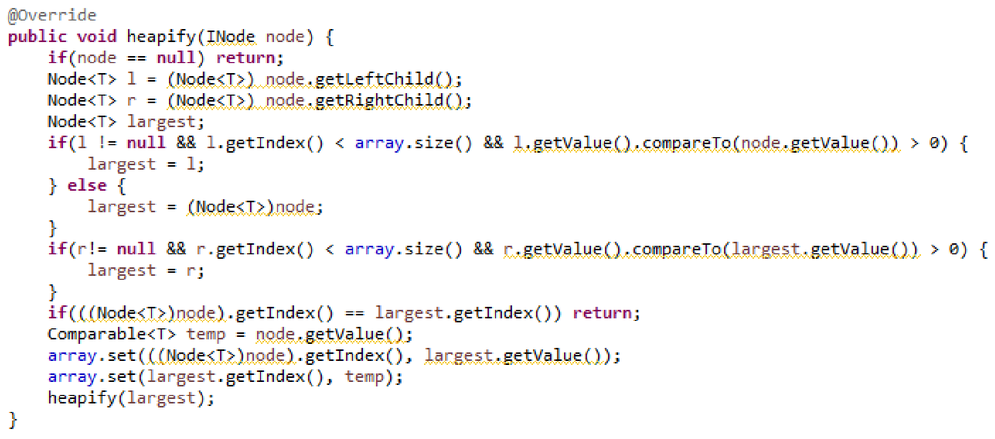
The (binary) heap data structure is an array object that we can view as a nearly complete binary tree as shown in 1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is _lled from the left up to a point. An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A. There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the speci_cs of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root, A[parent[i]] _ A[i] (1) that is, the value of a node is at most the value of its parent.

1.2 Requirements

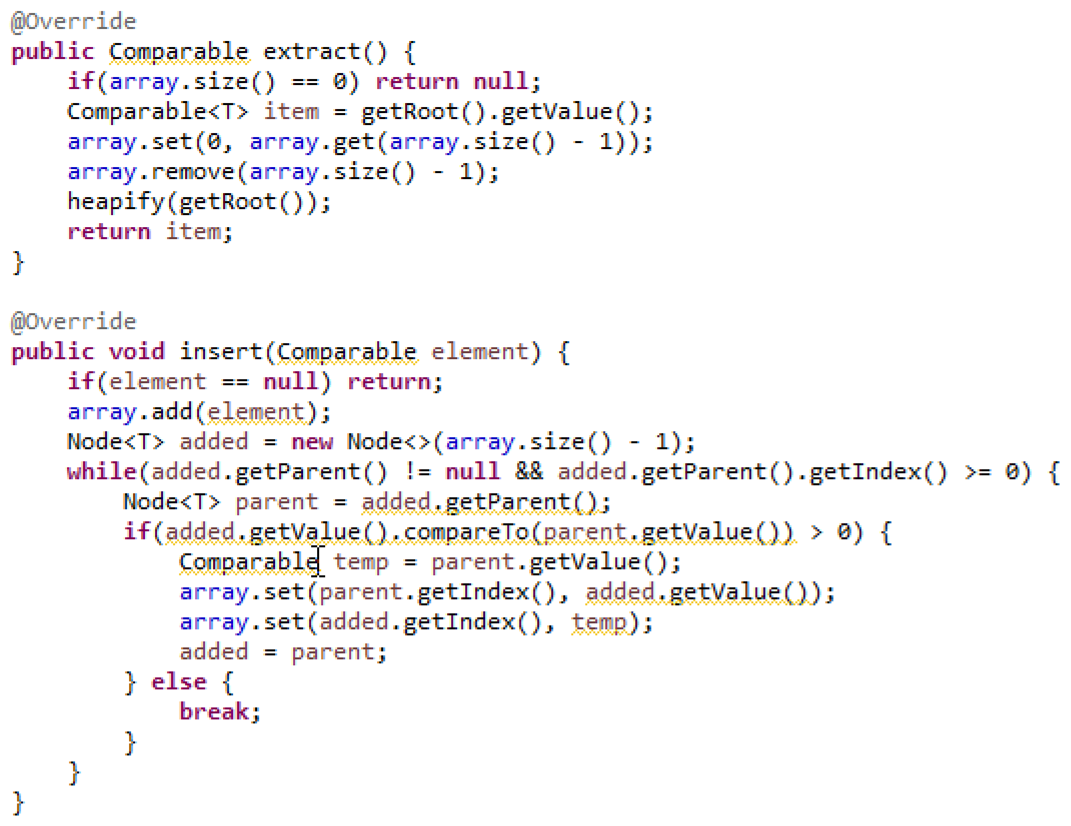
In this assignment, you are required to implement the following some basic procedures:

- The MAX-HEAPIFY procedure, which runs in O(lg n) time, is the key to maintaining the max-heap property. Its input is a root node. When it is called, it assumes that the binary trees rooted to the left and right of the given node are max-heaps, but that the element at the root node might be smaller than its children, thus violating the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an un-ordered input array.
- The HEAPSORT procedure, which runs in O(n lg n) time, sorts an array in place.
- The MAX-HEAP-INSERT, and HEAP-REMOVE-MAX procedures, which run in O(lg n) time, allow the heap data structure to implement a priority queue.

1.3 Code snippet

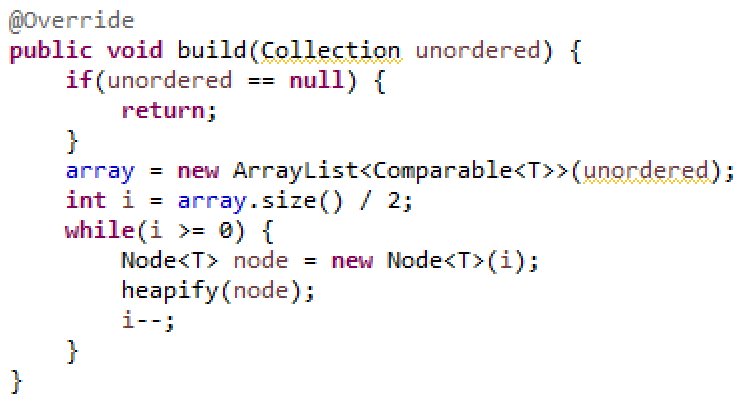


```java
@Override
public void heapify(INode node) {
    if(node == null) return;
    Node<T> l = (Node<T>) node.getLeftChild();
    Node<T> r = (Node<T>) node.getRightChild();
    Node<T> largest;
    if(l != null && l.getIndex() < array.size() && l.getValue().compareTo(node.getValue()) > 0) {
        largest = l;
    } else {
        largest = (Node<T>)node;
    }
    if(r!= null && r.getIndex() < array.size() && r.getValue().compareTo(largest.getValue()) > 0) {
        largest = r;
    }
    if(((Node<T>)node).getIndex() == largest.getIndex()) return;
    Comparable<T> temp = node.getValue();
    array.set(((Node<T>)node).getIndex(), largest.getValue());
    array.set(largest.getIndex(), temp);
    heapify(largest);
}
```



```
@Override
public Comparable extract() {
    if(array.size() == 0) return null;
    Comparable<T> item = getRoot().getValue();
    array.set(0, array.get(array.size() - 1));
    array.remove(array.size() - 1);
    heapify(getRoot());
    return item;
}

@Override
public void insert(Comparable element) {
    if(element == null) return;
    array.add(element);
    Node<T> added = new Node<>(array.size() - 1);
    while(added.getParent() != null && added.getParent().getIndex() >= 0) {
        Node<T> parent = added.getParent();
        if(added.getValue().compareTo(parent.getValue()) > 0) {
            Comparable temp = parent.getValue();
            array.set(parent.getIndex(), added.getValue());
            array.set(added.getIndex(), temp);
            added = parent;
        } else {
            break;
        }
    }
}
```

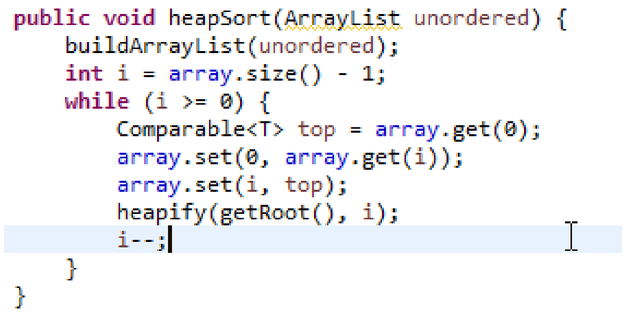


```
@Override
public void build(Collection unordered) {
    if(unordered == null) {
        return;
    }
    array = new ArrayList<Comparable<T>>(unordered);
    int i = array.size() / 2;
    while(i >= 0) {
        Node<T> node = new Node<T>(i);
        heapify(node);
        i--;
    }
}
```

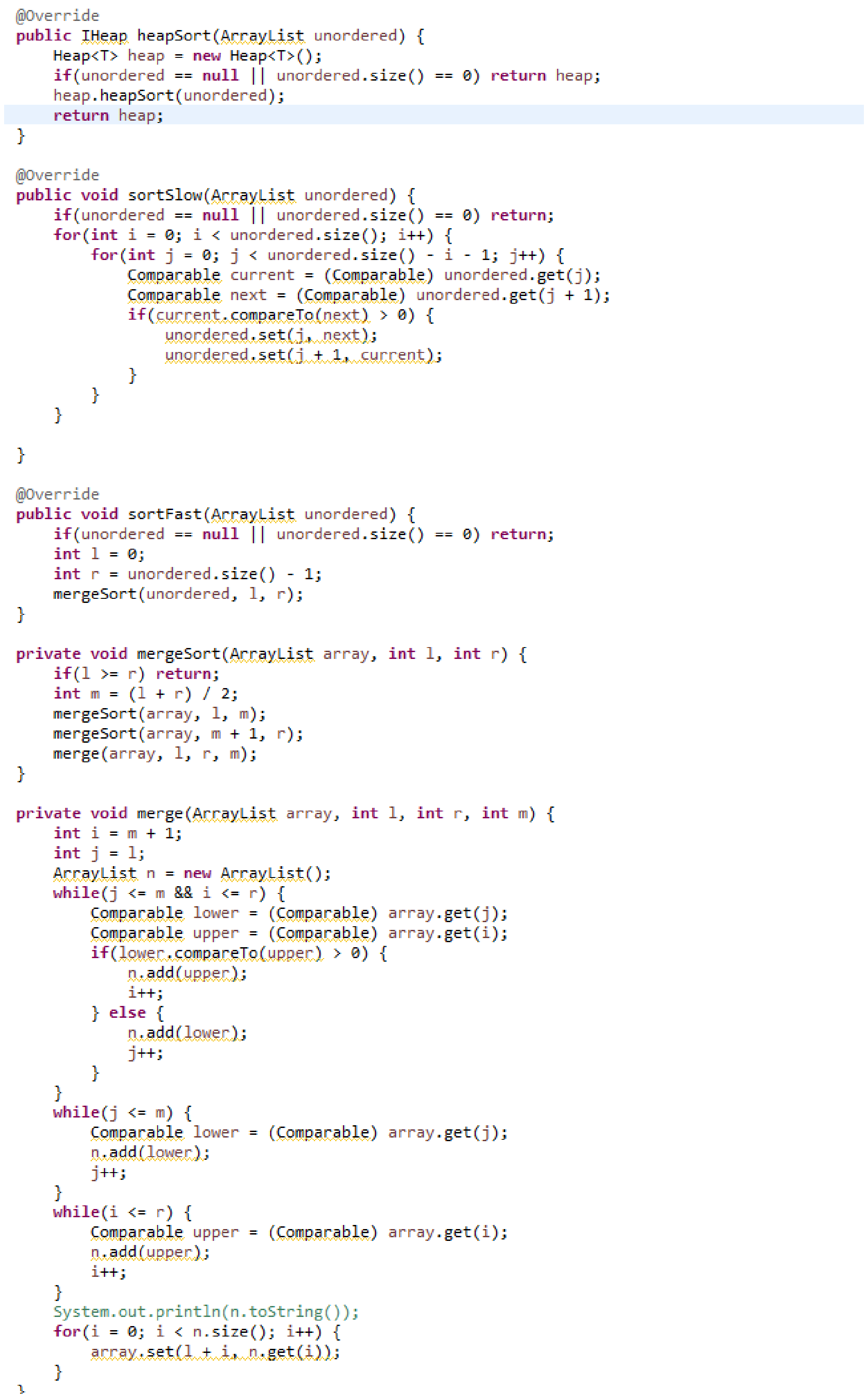
2 Sorting Techniques

- You are required to implement the heapsort algorithm as an application for binary heaps. You're advised to compare the running time of your implementation against: { An O(n2) sorting algorithm such as Selection Sort, Bubble Sort, or Insertion sort. { An O(n lg n) sorting algorithm such as Merge Sort or Quick sort algorithm in the average case.
- In addition to heapsort, implement any of the sorting algorithms from each class mentioned above, O(n lg n) and O(n2). For example, you can choose to implement Merge Sort and Bubble Sort.
- To test your implementation and analyze the running time performance, you are advised to generate a dataset of random numbers and plot the relationship between the execution time of the sorting algorithm versus the input size.

2.1 Code Snippet



```java
public void heapSort(ArrayList unordered) {
    buildArrayList(unordered);
    int i = array.size() - 1;
    while (i >= 0) {
        Comparable<T> top = array.get(0);
        array.set(0, array.get(i));
        array.set(i, top);
        heapify(getRoot(), i);
        i--;
    }
}
```



```java
@Override
public IHeap heapSort(ArrayList unordered) {
    Heap<T> heap = new Heap<T>();
    if(unordered == null || unordered.size() == 0) return heap;
    heap.heapSort(unordered);
    return heap;
}

@Override
public void sortSlow(ArrayList unordered) {
    if(unordered == null || unordered.size() == 0) return;
    for(int i = 0; i < unordered.size(); i++) {
        for(int j = 0; j < unordered.size() - i - 1; j++) {
            Comparable current = (Comparable) unordered.get(j);
            Comparable next = (Comparable) unordered.get(j + 1);
            if(current.compareTo(next) > 0) {
                unordered.set(j, next);
                unordered.set(j + 1, current);
            }
        }
    }

}

@Override
public void sortFast(ArrayList unordered) {
    if(unordered == null || unordered.size() == 0) return;
    int l = 0;
    int r = unordered.size() - 1;
    mergeSort(unordered, l, r);
}

private void mergeSort(ArrayList array, int l, int r) {
    if(l >= r) return;
    int m = (l + r) / 2;
    mergeSort(array, l, m);
    mergeSort(array, m + 1, r);
    merge(array, l, r, m);
}

private void merge(ArrayList array, int l, int r, int m) {
    int i = m + 1;
    int j = l;
    ArrayList n = new ArrayList();
    while(j <= m && i <= r) {
        Comparable lower = (Comparable) array.get(j);
        Comparable upper = (Comparable) array.get(i);
        if(lower.compareTo(upper) > 0) {
            n.add(upper);
            i++;
        } else {
            n.add(lower);
            j++;
        }
    }
    while(j <= m) {
        Comparable lower = (Comparable) array.get(j);
        n.add(lower);
        j++;
    }
    while(i <= r) {
        Comparable upper = (Comparable) array.get(i);
        n.add(upper);
        i++;
    }
    System.out.println(n.toString());
    for(i = 0; i < n.size(); i++) {
        array.set(l + i, n.get(i));
    }
}
```