[Year]

Binary heap assignment

NAME:MAHMOUD HAMED SHARSHAR
ID:46

Description:

- Implementation of max heap data structure.
- Implementation of heap sort algorithm.
- Implementation of merge sort algorithm as fast sorting algorithm.
- Implementation of selection sort algorithm as slow sorting algorithm.

Data Structure used:

- Array list to store nodes and manipulate it to keep max heap property.

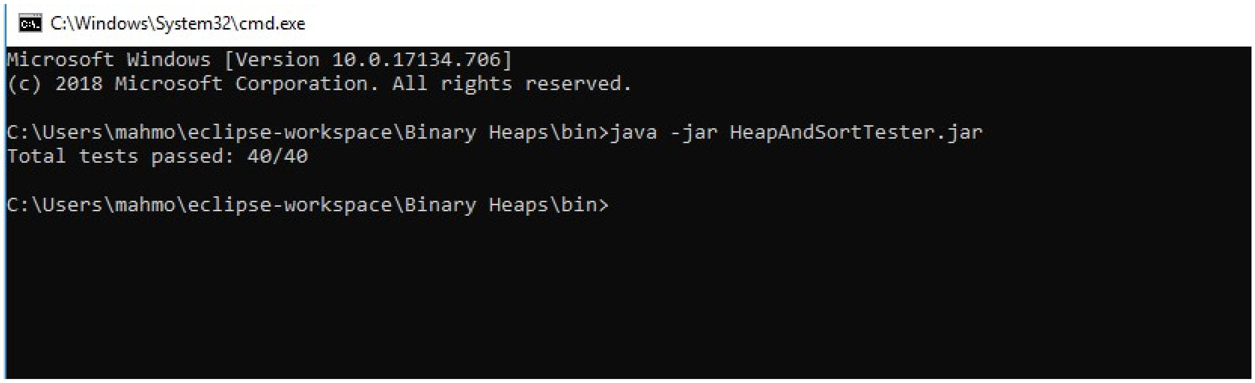
Main Modules:

- TreeNode class: data structure to represent each node in Heap.
- MaxHeap class:module to manipulate Heap and perform set of operations like insert ,extract,buildHeap,heapify.

- Sorting class: contain tree implementation of sorting algorithms to show differences between them .
  - Heap sort algorithm:run in O(nlogn)
  - Merge sort algorithm:run in O(nlogn)
  - Selection sort algorithm:run in O(n^2)

Test cases :

Test cases exist in bin folder as jar file and all of them passed.

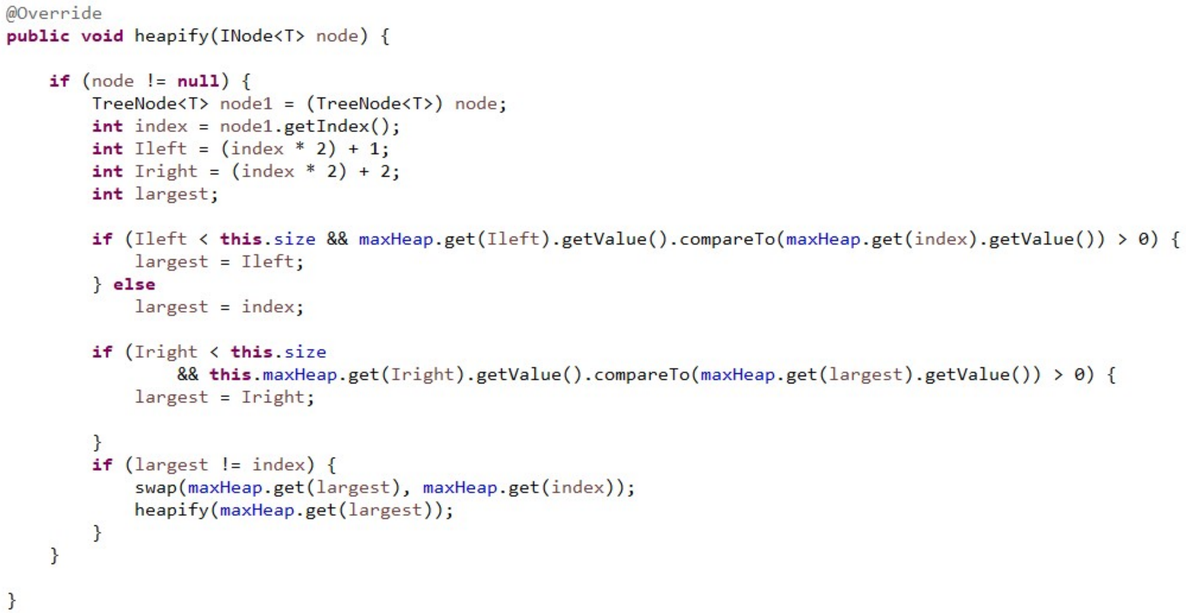


```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\mahmo\eclipse-workspace\Binary Heaps\bin>java -jar HeapAndSortTester.jar
Total tests passed: 40/40

C:\Users\mahmo\eclipse-workspace\Binary Heaps\bin>
```

Code snapshots:

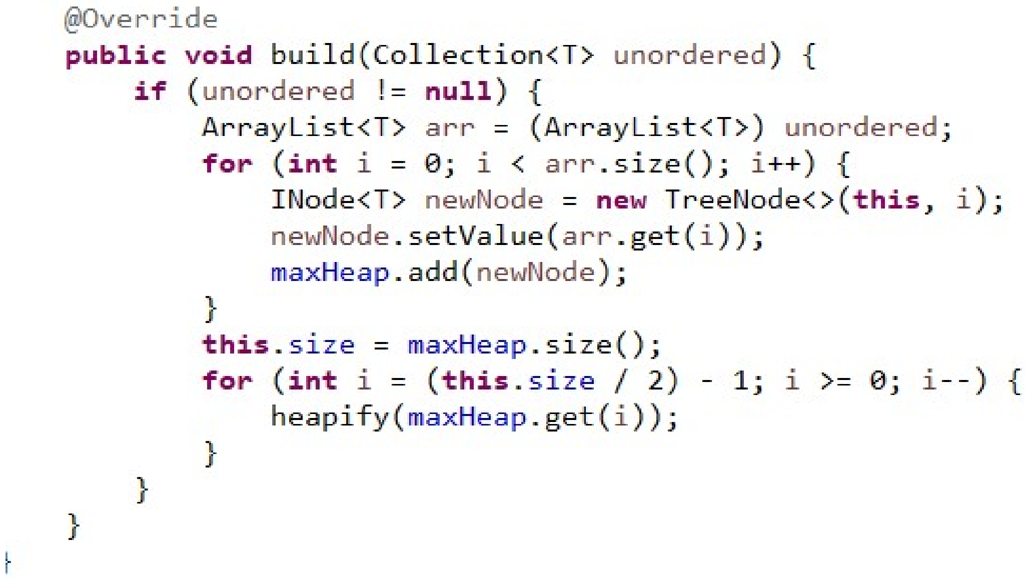
Basic methods in MaxHaep class:

- heapify:



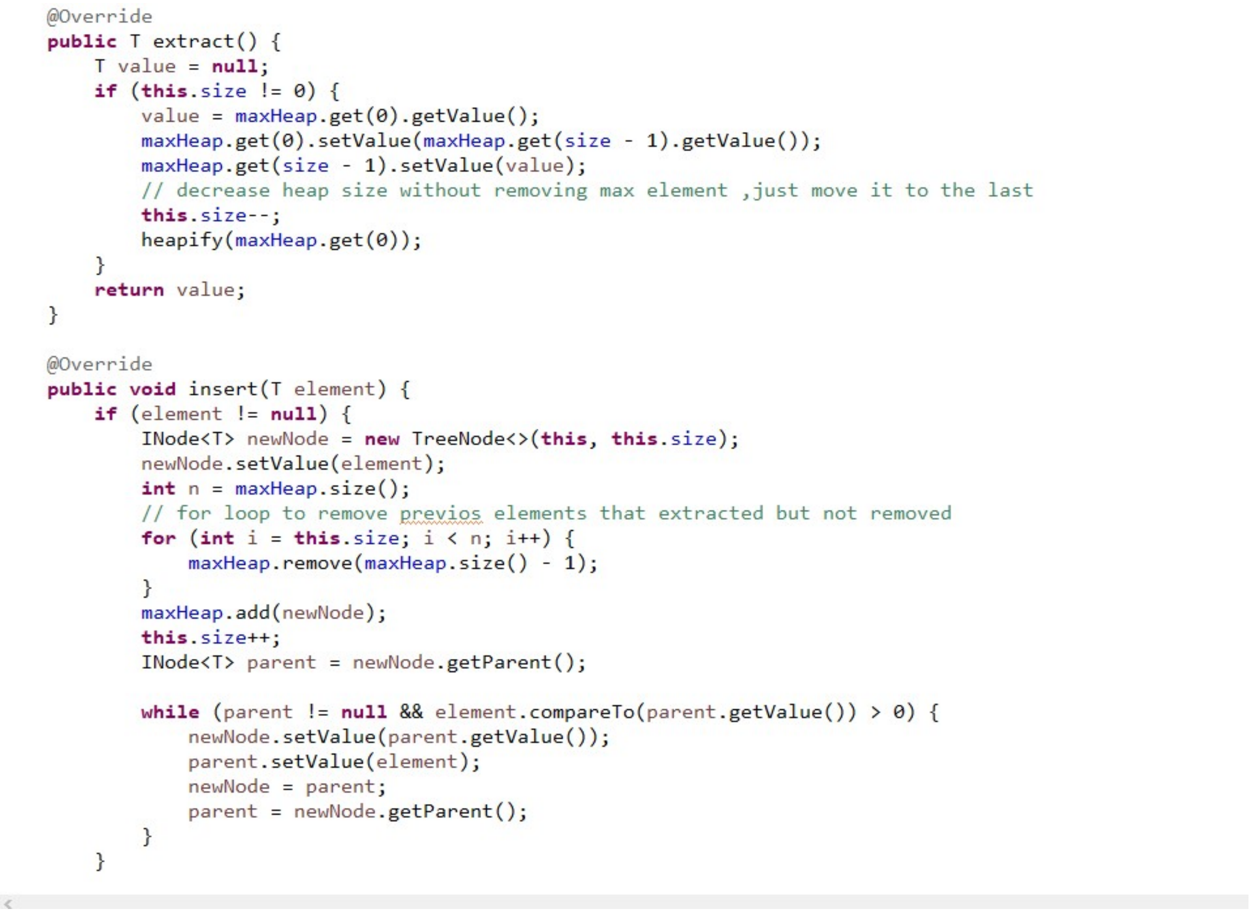
```
@Override
public void heapify(INode<T> node) {

    if (node != null) {
        TreeNode<T> node1 = (TreeNode<T>) node;
        int index = node1.getIndex();
        int Ileft = (index * 2) + 1;
        int Iright = (index * 2) + 2;
        int largest;

        if (Ileft < this.size && maxHeap.get(Ileft).getValue().compareTo(maxHeap.get(index).getValue()) > 0) {
            largest = Ileft;
        } else
            largest = index;

        if (Iright < this.size
                && this.maxHeap.get(Iright).getValue().compareTo(maxHeap.get(largest).getValue()) > 0) {
            largest = Iright;

        }
        if (largest != index) {
            swap(maxHeap.get(largest), maxHeap.get(index));
            heapify(maxHeap.get(largest));
        }
    }

}
```

- build:



```
@Override
public void build(Collection<T> unordered) {
    if (unordered != null) {
        ArrayList<T> arr = (ArrayList<T>) unordered;
        for (int i = 0; i < arr.size(); i++) {
            INode<T> newNode = new TreeNode<>(this, i);
            newNode.setValue(arr.get(i));
            maxHeap.add(newNode);
        }
        this.size = maxHeap.size();
        for (int i = (this.size / 2) - 1; i >= 0; i--) {
            heapify(maxHeap.get(i));
        }
    }
}
```

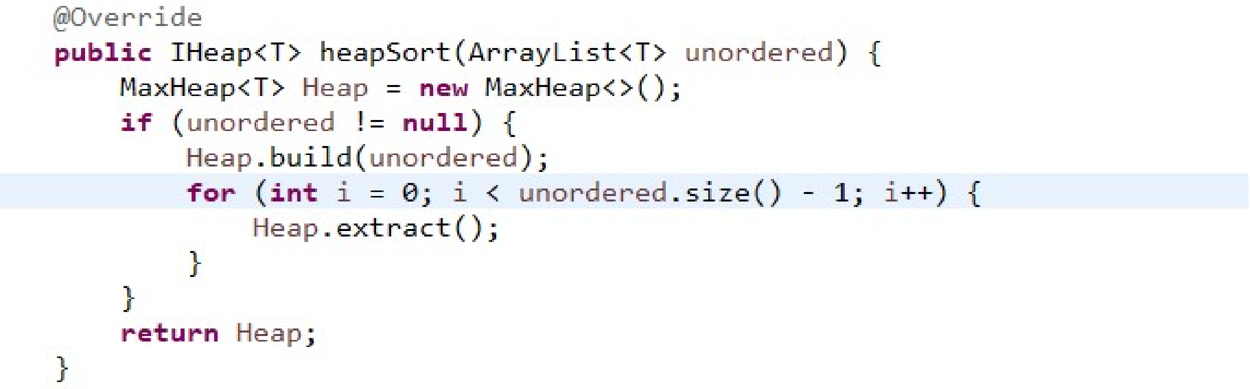
- extract and insert:



```
@Override
public T extract() {
    T value = null;
    if (this.size != 0) {
        value = maxHeap.get(0).getValue();
        maxHeap.get(0).setValue(maxHeap.get(size - 1).getValue());
        maxHeap.get(size - 1).setValue(value);
        // decrease heap size without removing max element ,just move it to the last
        this.size--;
        heapify(maxHeap.get(0));
    }
    return value;
}

@Override
public void insert(T element) {
    if (element != null) {
        INode<T> newNode = new TreeNode<>(this, this.size);
        newNode.setValue(element);
        int n = maxHeap.size();
        // for loop to remove previos elements that extracted but not removed
        for (int i = this.size; i < n; i++) {
            maxHeap.remove(maxHeap.size() - 1);
        }
        maxHeap.add(newNode);
        this.size++;
        INode<T> parent = newNode.getParent();

        while (parent != null && element.compareTo(parent.getValue()) > 0) {
            newNode.setValue(parent.getValue());
            parent.setValue(element);
            newNode = parent;
            parent = newNode.getParent();
        }
    }
```

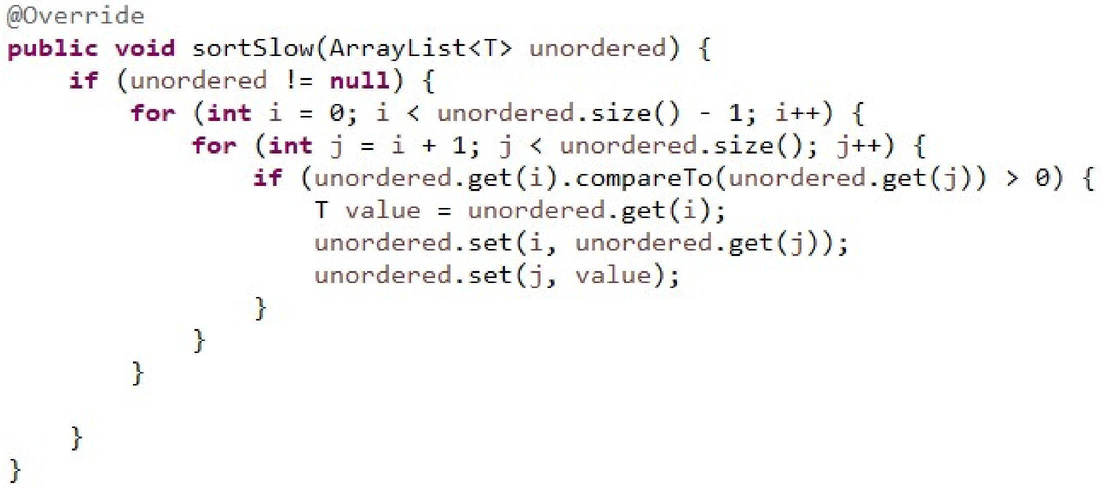
Methods of sorting:

- HeapSort:



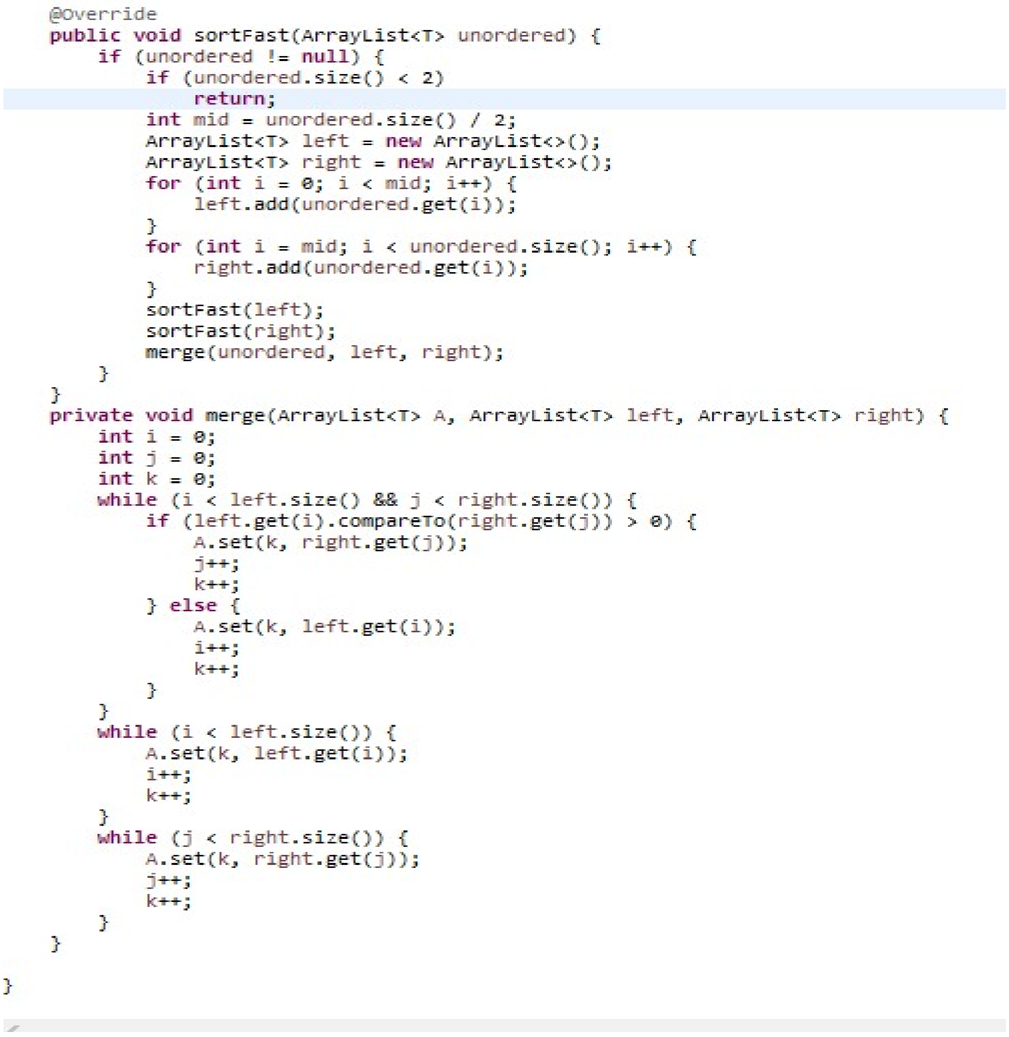
```
@Override
public IHeap<T> heapSort(ArrayList<T> unordered) {
    MaxHeap<T> Heap = new MaxHeap<>();
    if (unordered != null) {
        Heap.build(unordered);
        for (int i = 0; i < unordered.size() - 1; i++) {
            Heap.extract();
        }
    }
    return Heap;
}
```

- Slow sort==➔selection sort:



```
 @Override
 public void sortSlow(ArrayList<T> unordered) {
     if (unordered != null) {
         for (int i = 0; i < unordered.size() - 1; i++) {
             for (int j = i + 1; j < unordered.size(); j++) {
                 if (unordered.get(i).compareTo(unordered.get(j)) > 0) {
                     T value = unordered.get(i);
                     unordered.set(i, unordered.get(j));
                     unordered.set(j, value);
                 }
             }
         }

     }
 }
```

- fastSort➔merge sort:



```
    @Override
    public void sortFast(ArrayList<T> unordered) {
        if (unordered != null) {
            if (unordered.size() < 2)
                return;
            int mid = unordered.size() / 2;
            ArrayList<T> left = new ArrayList<>();
            ArrayList<T> right = new ArrayList<>();
            for (int i = 0; i < mid; i++) {
                left.add(unordered.get(i));
            }
            for (int i = mid; i < unordered.size(); i++) {
                right.add(unordered.get(i));
            }
            sortFast(left);
            sortFast(right);
            merge(unordered, left, right);
        }
    }
    private void merge(ArrayList<T> A, ArrayList<T> left, ArrayList<T> right) {
        int i = 0;
        int j = 0;
        int k = 0;
        while (i < left.size() && j < right.size()) {
            if (left.get(i).compareTo(right.get(j)) > 0) {
                A.set(k, right.get(j));
                j++;
                k++;
            } else {
                A.set(k, left.get(i));
                i++;
                k++;
            }
        }
        while (i < left.size()) {
            A.set(k, left.get(i));
            i++;
            k++;
        }
        while (j < right.size()) {
            A.set(k, right.get(j));
            j++;
            k++;
        }
    }

}
```