Name Member 1 : Febronia Ashraf Tawfik ***ID : 29***

Name Member 2 : Neveen Samir Nagy ***ID : 58***

Binary Heap & Sorting Techniques

1-Problem Statement :

a) Binary Heap :

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. 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 filled 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 specifics 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,

$$A[parent[i]] >= A[i]$$

that is, the value of a node is at most the value of its parent.

B) Sorting Techniques :

_Heap sort algorithm as an application for binary heaps.
-Bubble Sort : O(n^2)
-Merge Sort: O(n log n)

2-Snipping :

INode :

```
115     private class Node<T extends Comparable<T>> implements INode{
116         int index ;
117         Comparable<T> value ;
118         public Node(int i) {
119             index = i+1;
120         }
121         public void setIndex(int i) {
122             index = i+1;
123         }
124
125         @Override
126         public INode getLeftChild() {
127             if(2*index - 1 >= heap.size()) {
128                 return null;
129             }
130             return heap.get(2*index - 1);
131         }
132
133         @Override
134         public INode getRightChild() {
135             if(2*index+1 - 1 >= heap.size()) {
136                 return null;
137             }
138             return heap.get(2*index+1 - 1);
139         }
140
141         @Override
142         public INode getParent() {
143             if(index/2 == 0) {
144                 return null;
145             }
146             return heap.get((index/2) - 1);
147         }
148
149         @Override
150         public Comparable<T> getValue() {
151
152             return (Comparable<T>) value;
```

IHeap :

```
public class Heap<T extends Comparable<T>> implements IHeap{

    ArrayList<INode<T>> heap = new ArrayList<INode<T>>();
    @Override
    public INode getRoot() {
        // TODO Auto-generated method stub
        if(heap.size()==0) {
            return null;
        }
        return heap.get(0);
    }

    @Override
    public int size() {
        // TODO Auto-generated method stub
        return heap.size();
    }

    @Override
    public void heapify(INode node) {
        // TODO Auto-generated method stub
        if(node == null) {
            return;
        }
        INode left = node.getLeftChild();
        INode right = node.getRightChild();
        INode largest = node;
        if(left!=null && ((Comparable<T>) left.getValue()).compareTo((T) node.getValue()) > 0) {
            largest = left;
        }else {
            largest = node;
        }
        if(right != null && ((Comparable<T>) right.getValue()).compareTo((T) largest.getValue()) > 0){
            largest = right;
        }
        if(largest != node) {
            Comparable temp = node.getValue();
            node.setValue(largest.getValue());
```

```java
@Override
public void heapify(INode node) {
    // TODO Auto-generated method stub
    if(node == null) {
        return;
    }
    INode left = node.getLeftChild();
    INode right = node.getRightChild();
    INode largest = node;
    if(left!=null && ((Comparable<T>) left.getValue()).compareTo((T) node.getValue()) > 0) {
        largest = left;
    }else {
        largest = node;
    }
    if(right != null && ((Comparable<T>) right.getValue()).compareTo((T) largest.getValue()) > 0){
        largest = right;
    }
    if(largest != node) {
        Comparable temp = node.getValue();
        node.setValue(largest.getValue());
        largest.setValue(temp);
        if(node != getRoot()) {
        heapify(node.getParent());
        }
        heapify(largest);
    }
}

@Override
public Comparable extract() {
    // TODO Auto-generated method stub
    Comparable root;
    if(heap.size()==0) {
        root = null;
    }else  {
     root = (Comparable)(heap.get(0)).getValue();
    }
    if(root == null) {
        return null;
    }
    ((INode)(heap.get(0))).setValue(((INode)(heap.get(heap.size()-1))).getValue());
    ((Node)(heap.get(0))).setIndex(0);
    heap.remove(heap.size()-1);
    if(heap.size()!=0) {
    heapify(heap.get(0));
    }
    return root;
}

@Override
public void insert(Comparable element) {
    if(element == null) {
        return;
    }
    Node n= new Node<>(heap.size());
    n.setValue(element);
    heap.add(n);
    if(n.getParent() != null) {
    heapify(n.getParent());
    }
}
```

```java
public ArrayList<INode<T>> getHeap(){
    return heap;
}
public void setHeap(INode[] tempo){
    heap.clear();
    for(int i=0;i<tempo.length;i++) {
        heap.add(tempo[i]);
    }
}

@Override
public void build(Collection unordered) {
    if(unordered==null ) {
        throw new RuntimeErrorException(null);
    }
    for(int i=0;i<unordered.size();i++) {
        Node n = new Node<>(i);
        n.setValue((((ArrayList<T>) unordered).get(i)));
        heap.add(n);
    }
    for(int i=(unordered.size()/2)-1;i>=0;i--) {
        heapify(heap.get(i));
    }
    for(int i = 0; i<heap.size();i++) {
        ((ArrayList<T>) unordered).set(i, heap.get(i).getValue());
    }
}
```

ISort :

```java
@Override
public IHeap heapSort(ArrayList unordered) {
    // TODO Auto-generated method stub
    if(unordered==null) {
        throw new RuntimeErrorException(null);
    }
    Heap heap = new Heap();
    heap.build(unordered);
    INode[] tempo = new INode[heap.getHeap().size()];

    for(int i= (heap.getHeap().size())-1; i>0; i--) {
        Comparable temp =  ((INode)heap.getHeap().get(0)).getValue();
        ((INode)(heap.getHeap().get(0))).setValue(((INode)(heap.getHeap().get(i))).getValue());
        ((INode)(heap.getHeap().get(i))).setValue(temp);
        ((ArrayList<T>) unordered).set(i, (T) (((INode)heap.getHeap().get(i)).getValue()));
        tempo[i]=(INode) heap.getHeap().get(i);
        heap.getHeap().remove(i);
        heap.heapify((INode) heap.getHeap().get(0));

    }
    if(unordered.size()!=0) {
    tempo[0]=(INode) heap.getHeap().get(0);
    ((ArrayList<T>) unordered).set(0, (T) (((INode)heap.getHeap().get(0)).getValue()));
    }
    heap.setHeap(tempo);
    return heap;
}
```

```java
@Override
public void sortSlow(ArrayList unordered) {
    // TODO Auto-generated method stub
    // BuBBle_Sort
    if(unordered==null) {
        throw new RuntimeErrorException(null);
    }
    for (int i = 0; i < unordered.size(); i++) {
        for (int j = 0; j < unordered.size() - 1; j++) {
            if (((Comparable<T>) unordered.get(j)).compareTo((T) unordered.get(j + 1)) > 0) {
                T temp = (T) unordered.get(j);
                unordered.set(j, unordered.get(j + 1));
                unordered.set(j + 1, temp);
            }
        }
    }
}
```

```java
@Override
public void sortFast(ArrayList unordered) {
    // TODO Auto-generated method stub
    if(unordered==null) {
        throw new RuntimeErrorException(null);
    }
    int first =0;
    int last = unordered.size()-1;
    MergeSort m = new MergeSort<>();
    m.mergeSort(unordered, first, last);
}
```

```java
public void mergeSort(ArrayList  unordered, int first, int last) {

    if(first < last) {
    int mid = (last+first)/2;
    mergeSort(unordered, first, mid);
    mergeSort(unordered, mid+1, last);
    merge(unordered, first, mid, last);

    }
}
```

```
    private void merge(ArrayList array, int first, int mid, int last) {
        int first1 = first;
        int last1 =mid;
        int first2 = mid+1;
        int last2= last;
        int index =0;
        ArrayList  temp = new ArrayList<>();
        while(first1<=last1 && first2<=last2 ) {
            if(((Comparable<T>)array.get(first1)).compareTo((T)array.get(first2)) > 0) {
                temp.add(array.get(first2));
                first2++;
            }
            else {
                temp.add(array.get(first1));
                first1++;
            }
        }
        while(first1<=last1) {
            temp.add(array.get(first1));
            first1++;
        }
        while(first2<=last2) {
            temp.add(array.get(first2));
            first2++;
        }

        for(int i=first;i<=last;i++) {
            array.set(i, temp.get(index));
            index++;

        }
    }
}
```

3-Assumption :

-In INode implementation:

-In functions that return left child, right child or parent of the node we check if the index of them out of the size of the array or the node is the root so it hasn't a parent.

-In IHeap implementation :

-In function getRoot : when the array is empty then the function will return null.

-In function heapify : the program check first if the node isn't in the array or is null.

-In function extract : we check if the array is empty so there isn't max element and if the size of the array is one so we won't do recursion.

-In insert : we check if the element is null so we won't add it and if the array is at first empty so the element is the root of the tree and we won't call heapify as we call it for the parent of the node and the root don't have parent.

-In function build : we check if the array is null we throw exception for that.

-**In ISort implementation** :

-In all functions for heap, slow and fast sort we check if the array is null so the program will throw exception for that.

4-Pseudo code :

a) Heap Implementation :

MAX_HEAP :

```
heapify(INode node)
   if(node = null)
       return;
   endIF;
   Node left = node.getLeftChild();
   Node right = node.getRightChild();
   Node largest = node;

if(left!=null & (left.getValue()).compareTo(node.getValue()) > 0)

largest = left;
endIF;
else
largest = node;

endIF;

if(right != null & (right.getValue()).compareTo(largest.getValue()) > 0)

   largest = right;

endIF;

 if(largest != node)
   temp = node.getValue();
   node.setValue(largest.getValue());
   largest.setValue(temp);
   if(node != getRootOfTree())
   heapify(node.getParent());
   endIF;
   heapify(largest);
   endIF;

END;
```

INSERT_ELEMENT :

```
insert(Comparable element)
        if(element = null)
                return;
        endIF;
        Node n;
        n.setValue(element);
        heap.add(n);
        if(n isNot GetRoot())
        heapify(n.getParent());
        endIF;
  END;
```

EXTRACT_MAXELEMENT :

```
extract()

        root;

        if(heap.size()=0)
           root = null;
        endIF;
       else
         root = (heap.get(0)).getValue();
        end;
        if(root = null) {
           return null;
        endIF;
        ((heap.get(0))).setValue((heap.get(heap.size()-1)).getValue());
        heap.remove(heap.size()-1);
        if(heap.size()!=0)
        heapify(heap.get(0));
        endIF;
        return root;
  END;
```

BUILD_MAXHEAP :

```
build(Collection unordered)
        if(unordered=null )
                throw new RuntimeErrorException(null);
```

```
            endIF;
            for(I from 0 → unordered.size())
              Node n;
               n.setValue(((unordered).get(i)));
              heap.add(n);
            endFOR;
            for(I from (unordered.size()/2)-1 → 0)
               heapify(heap.get(i));
            endFOR;
            for(i from 0 → heap.size())
              (unordered).set(i, heap.get(i).getValue());
            endFOR;
  END;
```

---

b) Sorting Implementation :

HEAP_SORT :

```
heapSort(ArrayList unordered)
   if(unordered=null)
        throw new RuntimeErrorException(null);
   endIF;
 heap.build(unordered);
 for(I from (heap.getHeap().size())-1→0)
   temp =  (heap.getHeap().get(0)).getValue();
   ((heap.getHeap().get(0))).setValue(((heap.getHeap().get(i))).getValue());
   ((heap.getHeap().get(i))).setValue(temp);
    (unordered).set(i, ((heap.getHeap().get(i)).getValue()));
     heap.getHeap().remove(i);
     heap.heapify(heap.getHeap().get(0));
   endFOR;
   if(unordered.size()!=0)
   (unordered).set(0,  ((heap.getHeap().get(0)).getValue()));
   endIF;
   END;
```

SLOW_SORT : //O(N^2)

```
sortSlow(ArrayList unordered)
   if(unordered=null) {
       throw new RuntimeErrorException(null);
   endIF;
   for (i from 0 → unordered.size())
        for (j from 0 → unordered.size() - 1)
    if ((unordered.get(j)).compareTo(unordered.get(j + 1)) > 0)
            temp = unordered.get(j);
            unordered.set(j, unordered.get(j + 1));
            unordered.set(j + 1, temp);
   endIF;
   endFOR;
   endFOR;
   END;
```

FAST_SORT : //O(N LOG N)

```
mergeSort(ArrayList  unordered, int first, int last)

    if(first < last)
    mid = (last +first)/2;
    mergeSort(unordered, first, mid);
    mergeSort(unordered, mid+1, last);
    merge(unordered, first, mid, last);
    endIF;

END;

  merge(ArrayList array, int first, int mid, int last)

            first1 = first;
            last1 =mid;
            first2 = mid+1;
            last2= last;
            index =0;
            ArrayList  temp;
            while(first1<=last1 & first2<=last2 )
                if((array.get(first1)).compareTo(array.get(first2)) > 0)
                    temp.add(array.get(first2));
                    first2++;
                endIF;
                else
                      temp.add(array.get(first1));
                      first1++;
```

```
            end;
        endWhile;
        while(first1<=last1)
            temp.add(array.get(first1));
            first1++;
        endWhile;
        while(first2<=last2)
                temp.add(array.get(first2));
                first2++;
        endWhile;
        for(I from first → last)
            array.set(i, temp.get(index));
            index++;
        endFOR;
END;
```
