Names:

Nada Maher Mahmoud (56)

Nada Mostafa Rashad (57)

Data Structures 2 - Lab 1

Implementing Binary Heaps & Sorting Techniques

Problem Statement:

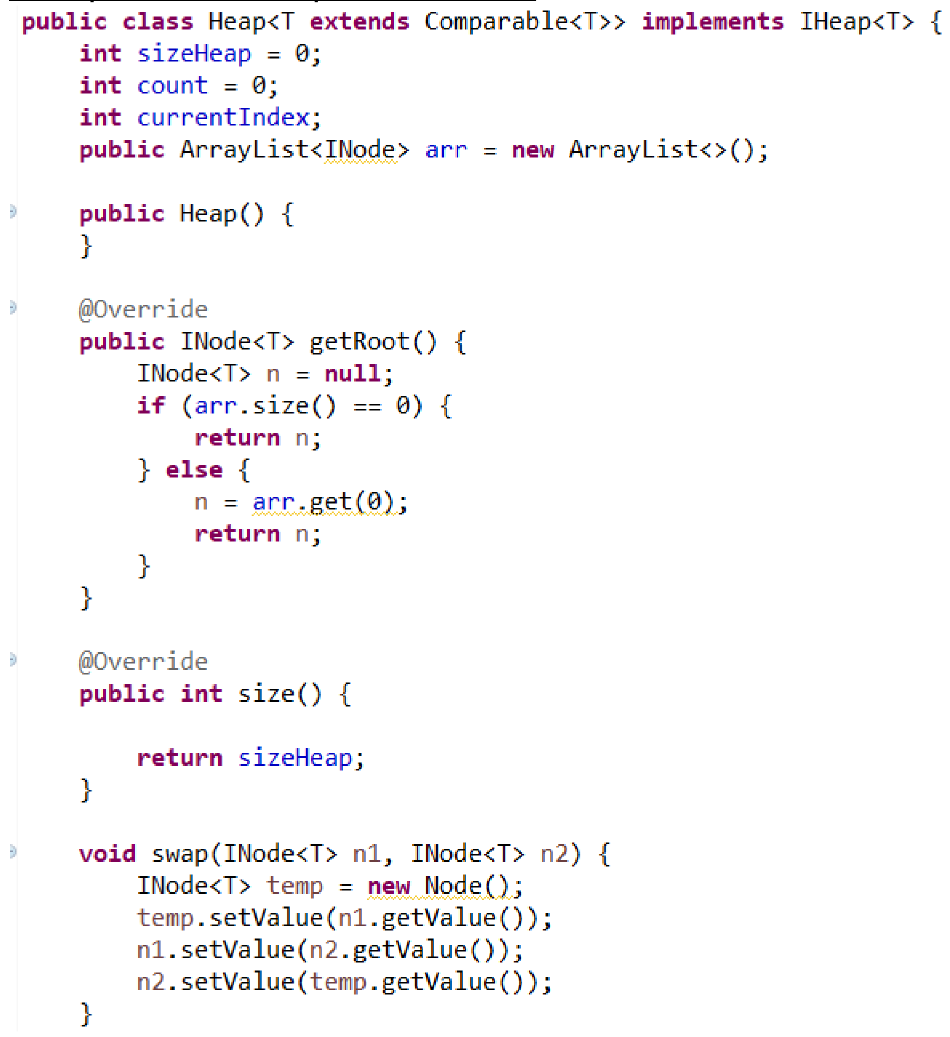
1) Implementing The MAX-HEAPIFY procedure, which runs in O(lg n) time,
2) Implementing The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
3) Implementing The HEAPSORT procedure, which runs in O(n lg n) time, sorts an array in place.
4) Implementing 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.
5) implementing the heapsort algorithm as an application for binary heaps.
6) implementing any of the sorting algorithms from each class, O(n lg n) and O(n2).

Assumptions and decisions:

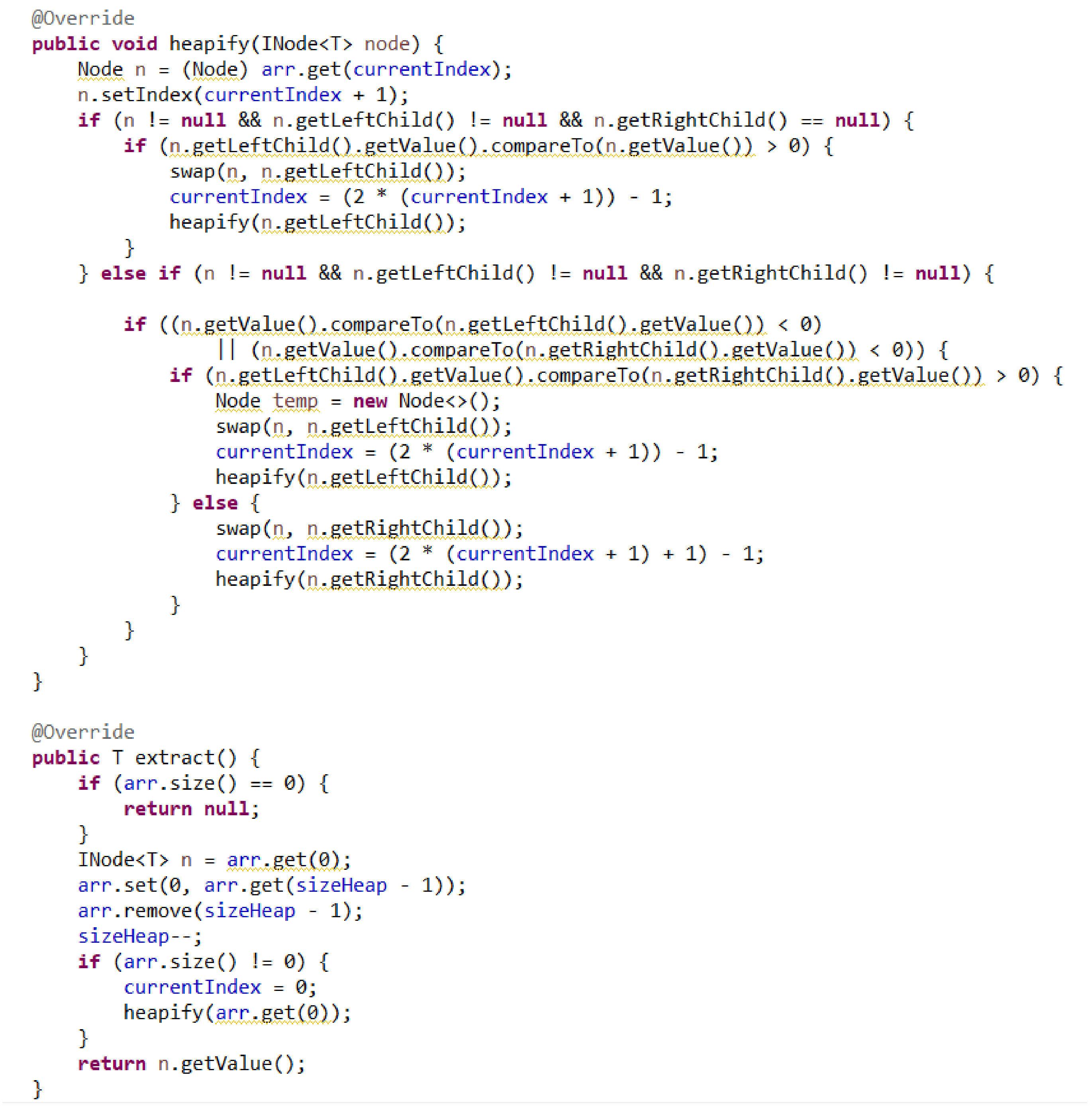
1) we decided to implement the bubble sort sorting technique as an O(n2) algorithm.
2) We decided to implement the merge sort sorting technique as an O(n lg n) algorithm.

Code Snippets:

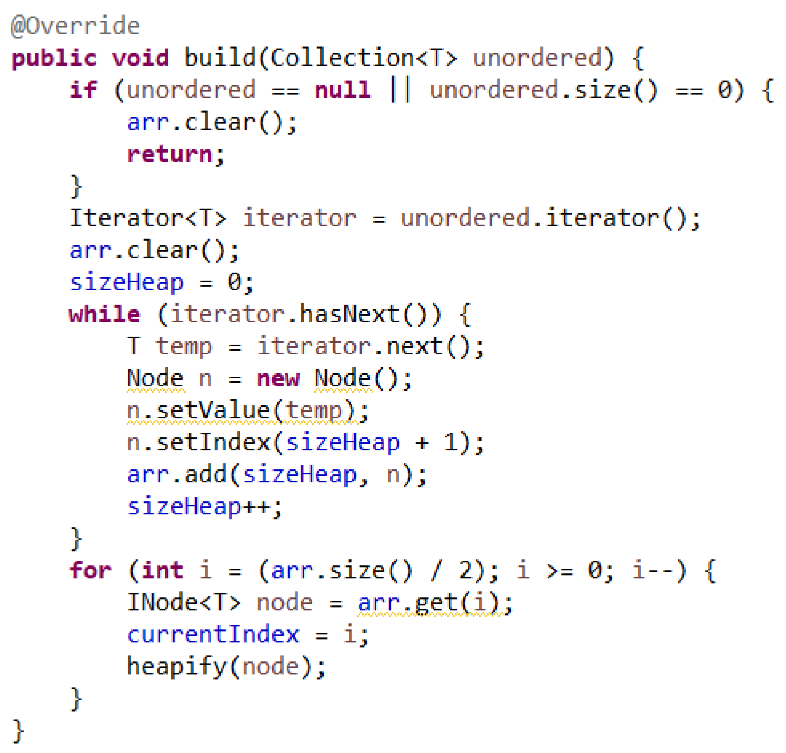
IHeap interface implementation:



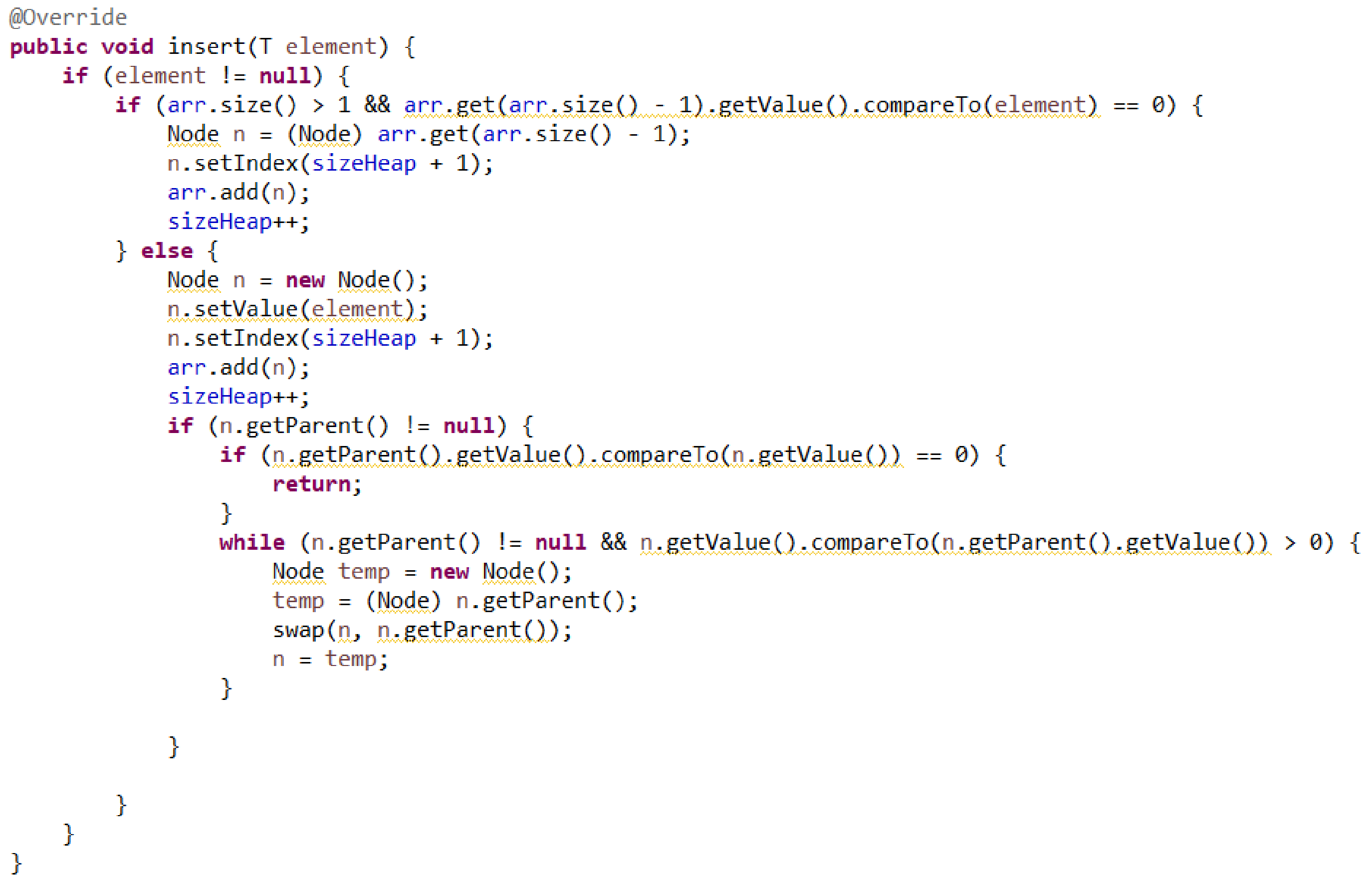
```
public class Heap<T extends Comparable<T>> implements IHeap<T> {
    int sizeHeap = 0;
    int count = 0;
    int currentIndex;
    public ArrayList<INode> arr = new ArrayList<>();

    public Heap() {
    }

    @Override
    public INode<T> getRoot() {
        INode<T> n = null;
        if (arr.size() == 0) {
            return n;
        } else {
            n = arr.get(0);
            return n;
        }
    }

    @Override
    public int size() {

        return sizeHeap;
    }

    void swap(INode<T> n1, INode<T> n2) {
        INode<T> temp = new Node();
        temp.setValue(n1.getValue());
        n1.setValue(n2.getValue());
        n2.setValue(temp.getValue());
    }
```



```java
@Override
public void heapify(INode<T> node) {
    Node n = (Node) arr.get(currentIndex);
    n.setIndex(currentIndex + 1);
    if (n != null && n.getLeftChild() != null && n.getRightChild() == null) {
        if (n.getLeftChild().getValue().compareTo(n.getValue()) > 0) {
            swap(n, n.getLeftChild());
            currentIndex = (2 * (currentIndex + 1)) - 1;
            heapify(n.getLeftChild());
        }
    } else if (n != null && n.getLeftChild() != null && n.getRightChild() != null) {

        if ((n.getValue().compareTo(n.getLeftChild().getValue()) < 0)
                || (n.getValue().compareTo(n.getRightChild().getValue()) < 0)) {
            if (n.getLeftChild().getValue().compareTo(n.getRightChild().getValue()) > 0) {
                Node temp = new Node<>();
                swap(n, n.getLeftChild());
                currentIndex = (2 * (currentIndex + 1)) - 1;
                heapify(n.getLeftChild());
            } else {
                swap(n, n.getRightChild());
                currentIndex = (2 * (currentIndex + 1) + 1) - 1;
                heapify(n.getRightChild());
            }
        }
    }
}

@Override
public T extract() {
    if (arr.size() == 0) {
        return null;
    }
    INode<T> n = arr.get(0);
    arr.set(0, arr.get(sizeHeap - 1));
    arr.remove(sizeHeap - 1);
    sizeHeap--;
    if (arr.size() != 0) {
        currentIndex = 0;
        heapify(arr.get(0));
    }
    return n.getValue();
}
```



```
@Override
public void build(Collection<T> unordered) {
    if (unordered == null || unordered.size() == 0) {
        arr.clear();
        return;
    }
    Iterator<T> iterator = unordered.iterator();
    arr.clear();
    sizeHeap = 0;
    while (iterator.hasNext()) {
        T temp = iterator.next();
        Node n = new Node();
        n.setValue(temp);
        n.setIndex(sizeHeap + 1);
        arr.add(sizeHeap, n);
        sizeHeap++;
    }
    for (int i = (arr.size() / 2); i >= 0; i--) {
        INode<T> node = arr.get(i);
        currentIndex = i;
        heapify(node);
    }
}
```



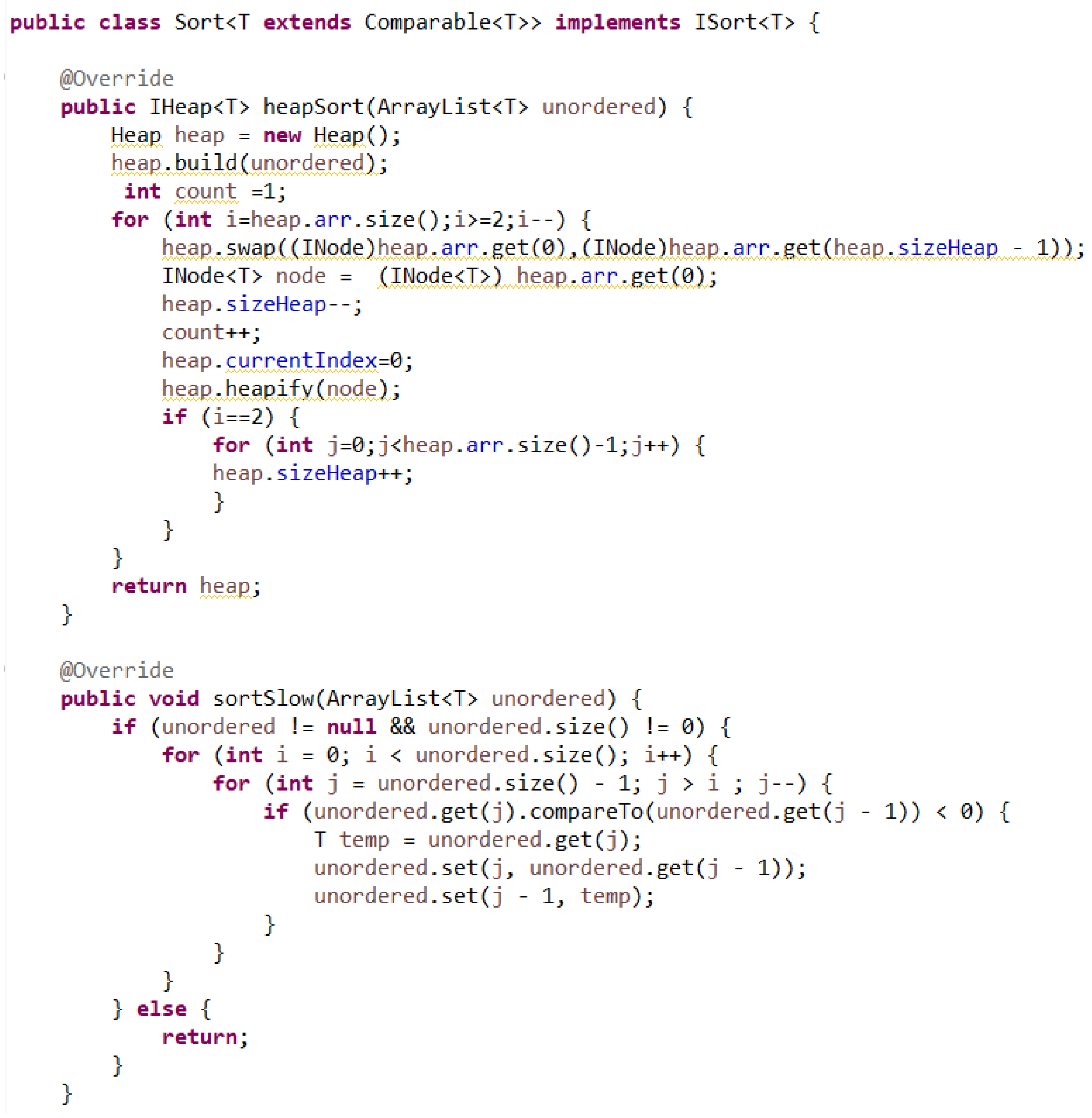
```
@Override
public void insert(T element) {
    if (element != null) {
        if (arr.size() > 1 && arr.get(arr.size() - 1).getValue().compareTo(element) == 0) {
            Node n = (Node) arr.get(arr.size() - 1);
            n.setIndex(sizeHeap + 1);
            arr.add(n);
            sizeHeap++;
        } else {
            Node n = new Node();
            n.setValue(element);
            n.setIndex(sizeHeap + 1);
            arr.add(n);
            sizeHeap++;
            if (n.getParent() != null) {
                if (n.getParent().getValue().compareTo(n.getValue()) == 0) {
                    return;
                }
                while (n.getParent() != null && n.getValue().compareTo(n.getParent().getValue()) > 0) {
                    Node temp = new Node();
                    temp = (Node) n.getParent();
                    swap(n, n.getParent());
                    n = temp;
                }

            }

        }
    }
}
```

INode interface implementation:

```
class Node<T extends Comparable<T>> implements INode<T> {

    private T value = null;
    private int index = 0;

    public void setIndex(int index) {
        this.index = index;
    }

    @Override
    public INode<T> getLeftChild() {
        int leftChildIndex;
        if (index < arr.size()) {
            leftChildIndex = 2 * index;
            if (leftChildIndex <= sizeHeap) {
                return (INode<T>) arr.get(leftChildIndex - 1);
            } else if (leftChildIndex > sizeHeap) {
                return null;
            }
        }

        return null;
    }

    @Override
    public INode<T> getRightChild() {
        int rightChildIndex;
        if (index < arr.size()) {
            rightChildIndex = 2 * index + 1;
            if (rightChildIndex <= sizeHeap) {
                return (INode<T>) arr.get(rightChildIndex - 1);
            } else if (rightChildIndex > sizeHeap) {
                return null;
            }
        }

        return null;
    }
```

```
@Override
public INode<T> getParent() {
    int parentIndex = 0;
    if (index > 1) {
        parentIndex = index / 2;
        if (parentIndex > 0) {
            return (INode<T>) arr.get(parentIndex - 1);
        } else {
            return null;
        }
    }
    return null;
}

@Override
public T getValue() {

    return value;
}

@Override
public void setValue(T value) {
    this.value = value;

}
```

ISort interface implementation:



```java
public class Sort<T extends Comparable<T>> implements ISort<T> {

    @Override
    public IHeap<T> heapSort(ArrayList<T> unordered) {
        Heap heap = new Heap();
        heap.build(unordered);
         int count =1;
        for (int i=heap.arr.size();i>=2;i--) {
            heap.swap((INode)heap.arr.get(0),(INode)heap.arr.get(heap.sizeHeap - 1));
            INode<T> node =  (INode<T>) heap.arr.get(0);
            heap.sizeHeap--;
            count++;
            heap.currentIndex=0;
            heap.heapify(node);
            if (i==2) {
                for (int j=0;j<heap.arr.size()-1;j++) {
                heap.sizeHeap++;
                }
            }
        }
        return heap;
    }

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

```
@Override
public void sortFast(ArrayList<T> unordered) {
    if(unordered != null && unordered.size() != 0) {
    mergeSort(unordered, 0, unordered.size() - 1);
    }
    else {
        return;
    }

}
void mergeSort( ArrayList<T> array, int p, int r) {
    if(p < r) {
        int q = (p + r) / 2;
        mergeSort(array, p, q);
        mergeSort(array, q + 1, r);
        merge(array, p, q, r);
    }
}
void merge(ArrayList<T> array, int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    ArrayList<T> L = new ArrayList<>();
    ArrayList<T> R = new ArrayList<>();
    for(int  i = 0; i < n1; i ++) {
        L.add(array.get(p + i));
    }
    for(int j = 0; j < n2; j++) {
        R.add(array.get(q + 1 + j));
    }
    int i = 0;
    int j = 0;
    int k = p;
    while( i < n1 && j < n2) {
        if(L.get(i).compareTo(R.get(j)) <= 0) {
            array.set(k, L.get(i));
            i ++;
        }
        else {
            array.set(k, R.get(j));
            j ++;
        }
        k ++;
        }
        while(i < n1) {
            array.set(k, L.get(i));
            i++;
            k++;
        }
        while( j < n2) {
            array.set(k, R.get(j));
            j++;
            k++;
        }
    }
}
```