

Alexandria University,  
Faculty of engineering,  
Data structures.  
Assignment 1.

Name: Mohamed Mohamed Abdlhakem. (43)

## **Problem statement:**

### **Heap implementation:**

It is required to implement the following some basic procedures:

- The MAX-HEAPIFY procedure, which runs in  $O(\log n)$  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 unordered input array.
- The HEAPSORT procedure, which runs in  $O(n \log n)$  time, sorts an array in place.
- The MAX-HEAP-INSERT, and HEAP-REMOVE-MAX procedures, which run in  $O(\log n)$  time, allow the heap data structure to implement a priority queue.

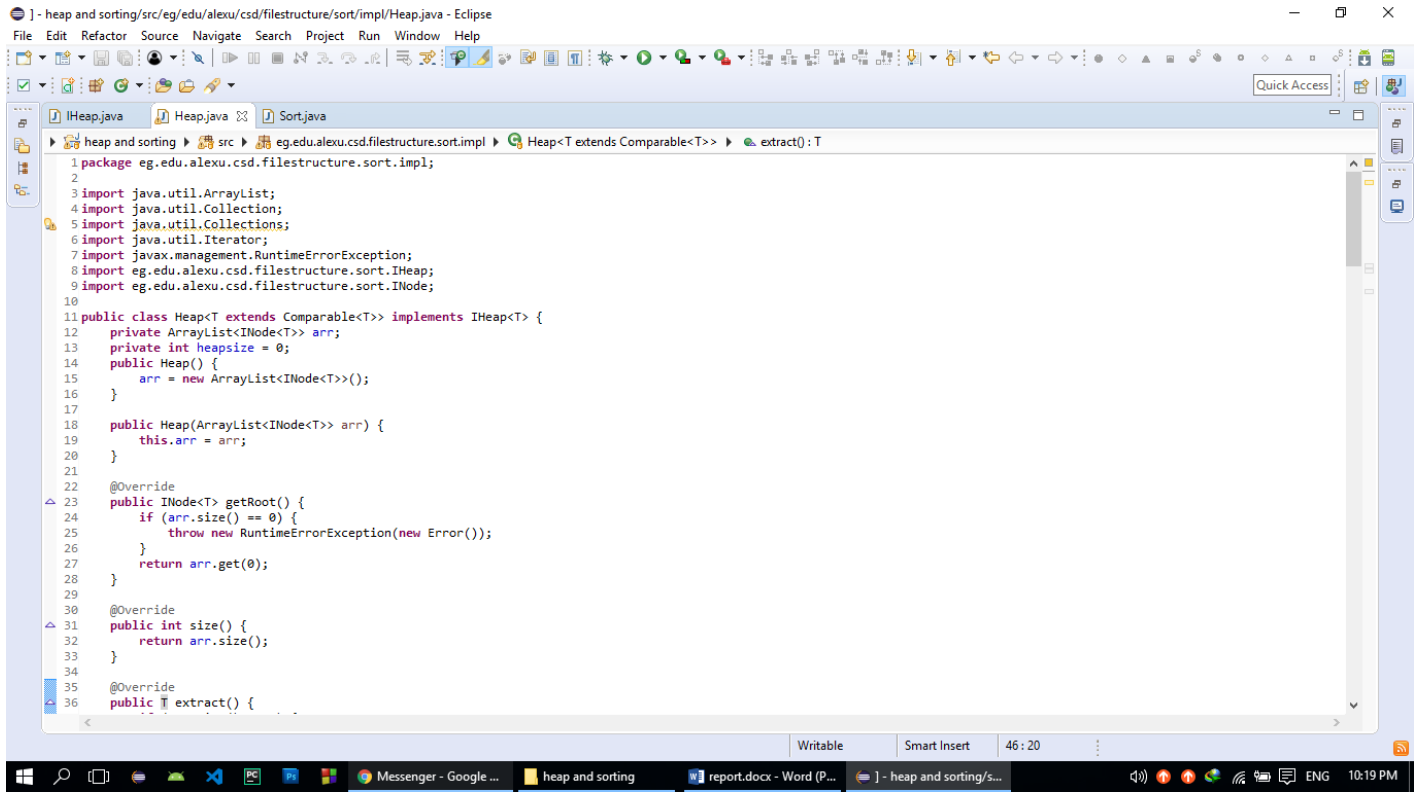
### **Sorting techniques:**

- It is required to implement the heap sort algorithm as an application for binary heaps. It is advised to compare the running time of your implementation against:
  - An  $O(n^2)$  sorting algorithm such as Selection Sort, Bubble Sort, or Insertion sort.
  - An  $O(n \log n)$  sorting algorithm such as Merge Sort or Quick sort algorithm in the average case.
- In addition to heap sort, it is required to implement any of the sorting algorithms from each class mentioned above,  $O(n \log n)$  and  $O(n^2)$ . For example, I choose to implement Merge Sort and Bubble Sort.

## **Data structures used:**

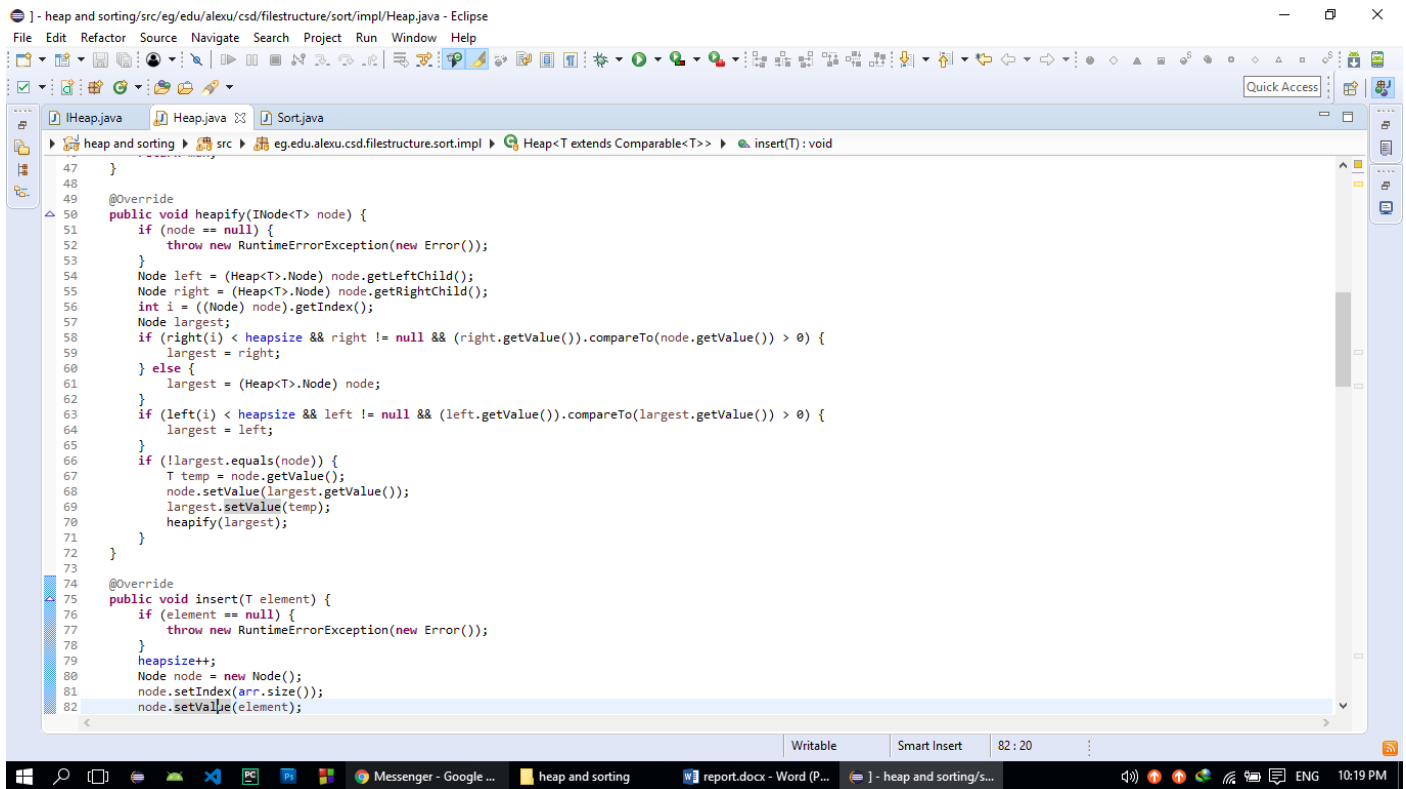
- Java Array list: to store the nodes of the binary heap. The order of access its element is  $O(1)$  because of its random access property.

## Code snippets:

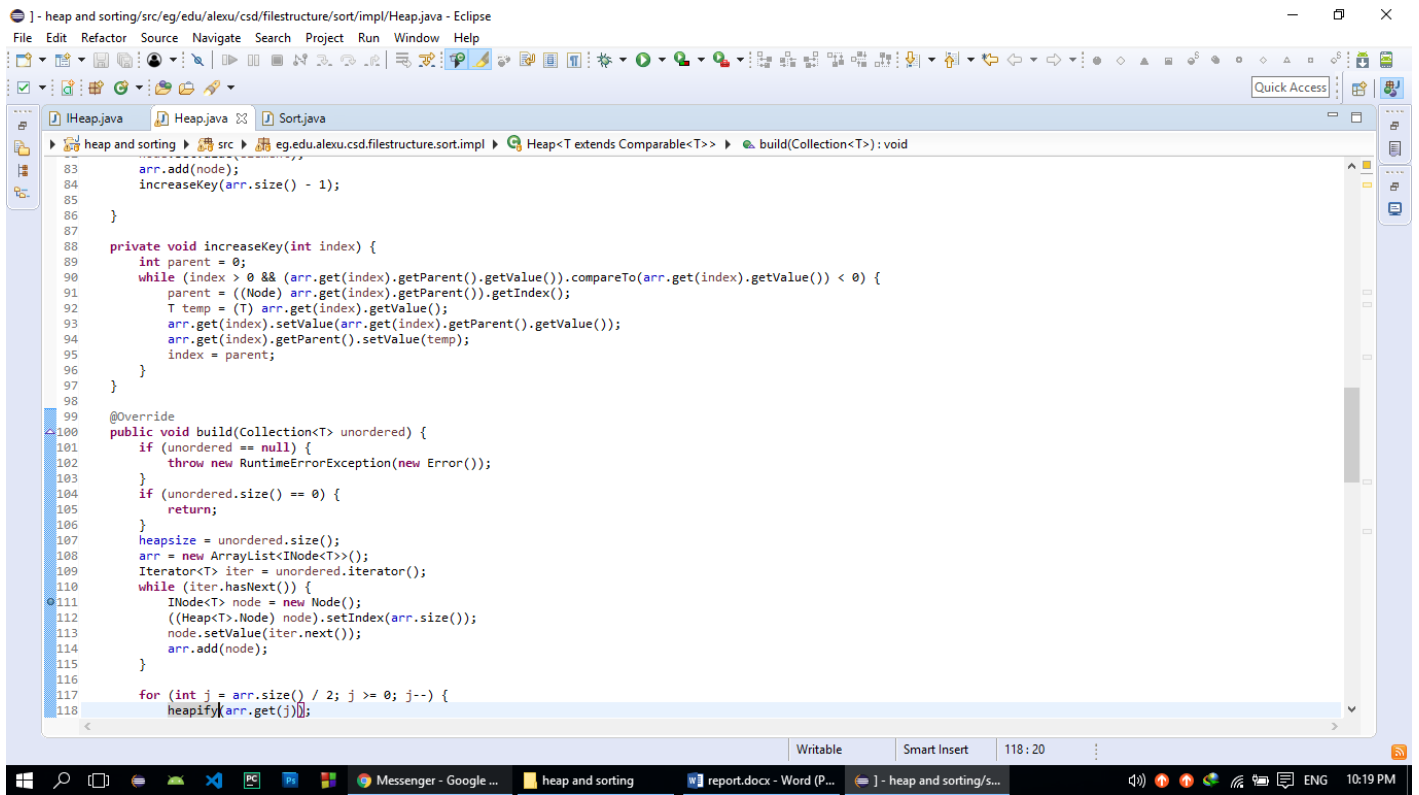


The screenshot shows the Eclipse IDE with a Java project named 'heap and sorting'. The package explorer on the left shows the project structure: 'src' > 'eg.edu.alexu.csd.filestructure.sort.impl'. The editor displays the code for 'Heap.java', which implements the 'IHeap' interface. The code includes imports for various Java utilities and exceptions, and defines a 'Heap' class with methods for getting the root, size, and extracting an element.

```
1 package eg.edu.alexu.csd.filestructure.sort.impl;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.Collections;
6 import java.util.Iterator;
7 import javax.management.RuntimeErrorException;
8 import eg.edu.alexu.csd.filestructure.sort.IHeap;
9 import eg.edu.alexu.csd.filestructure.sort.INode;
10
11 public class Heap<T extends Comparable<T>> implements IHeap<T> {
12     private ArrayList<INode<T>> arr;
13     private int heapSize = 0;
14     public Heap() {
15         arr = new ArrayList<INode<T>>();
16     }
17
18     public Heap(ArrayList<INode<T>> arr) {
19         this.arr = arr;
20     }
21
22     @Override
23     public INode<T> getRoot() {
24         if (arr.size() == 0) {
25             throw new RuntimeException(new Error());
26         }
27         return arr.get(0);
28     }
29
30     @Override
31     public int size() {
32         return arr.size();
33     }
34
35     @Override
36     public T extract() {
```

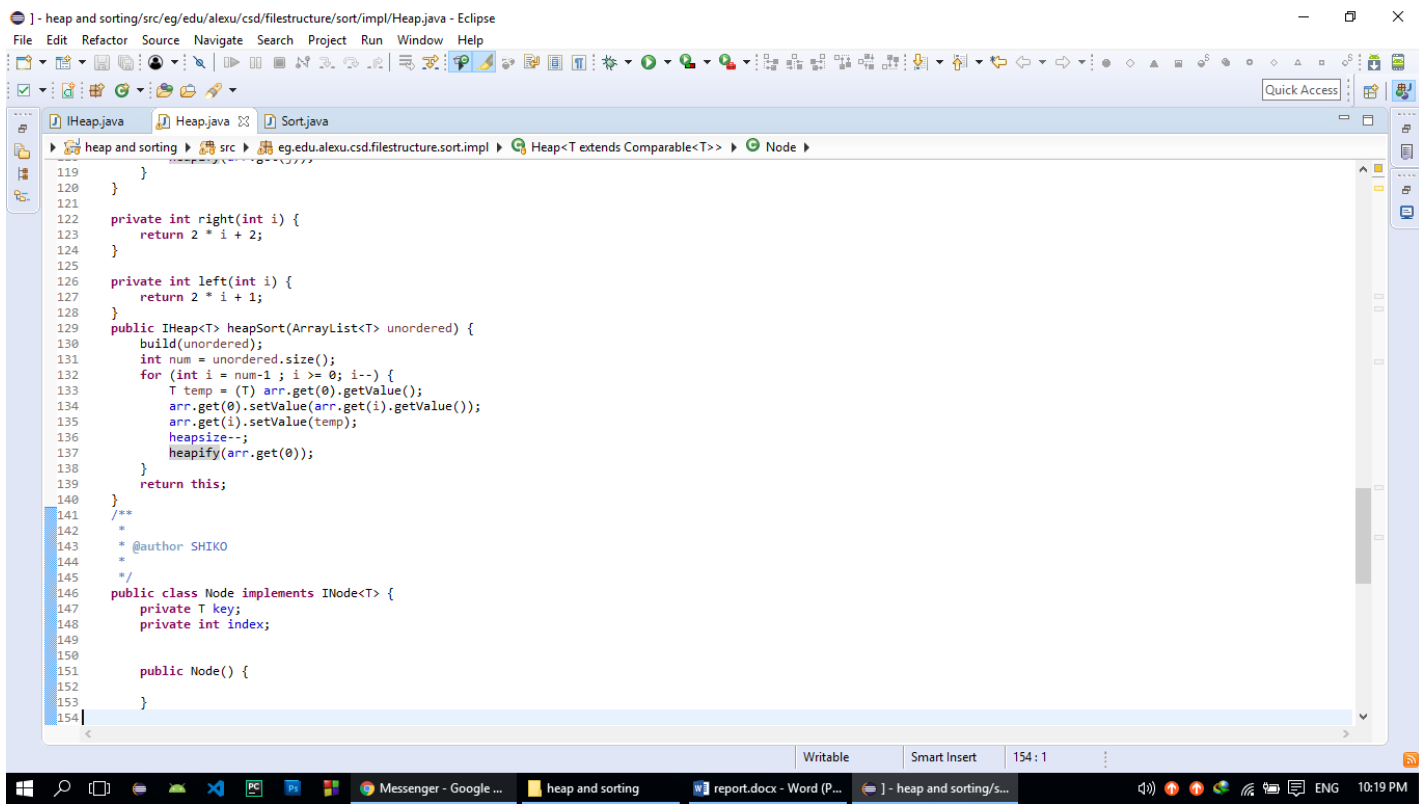


```
47 }
48
49 @Override
50 public void heapify(INode<T> node) {
51     if (node == null) {
52         throw new RuntimeException(new Error());
53     }
54     Node left = (Heap<T>.Node) node.getLeftChild();
55     Node right = (Heap<T>.Node) node.getRightChild();
56     int i = ((Node) node).getIndex();
57     Node largest;
58     if (right(i) < heapsize && right != null && (right.getValue()).compareTo(node.getValue()) > 0) {
59         largest = right;
60     } else {
61         largest = (Heap<T>.Node) node;
62     }
63     if (left(i) < heapsize && left != null && (left.getValue()).compareTo(largest.getValue()) > 0) {
64         largest = left;
65     }
66     if (!largest.equals(node)) {
67         T temp = node.getValue();
68         node.setValue(largest.getValue());
69         largest.setValue(temp);
70         heapify(largest);
71     }
72 }
73
74 @Override
75 public void insert(T element) {
76     if (element == null) {
77         throw new RuntimeException(new Error());
78     }
79     heapsize++;
80     Node node = new Node();
81     node.setIndex(arr.size());
82     node.setValue(element);
```



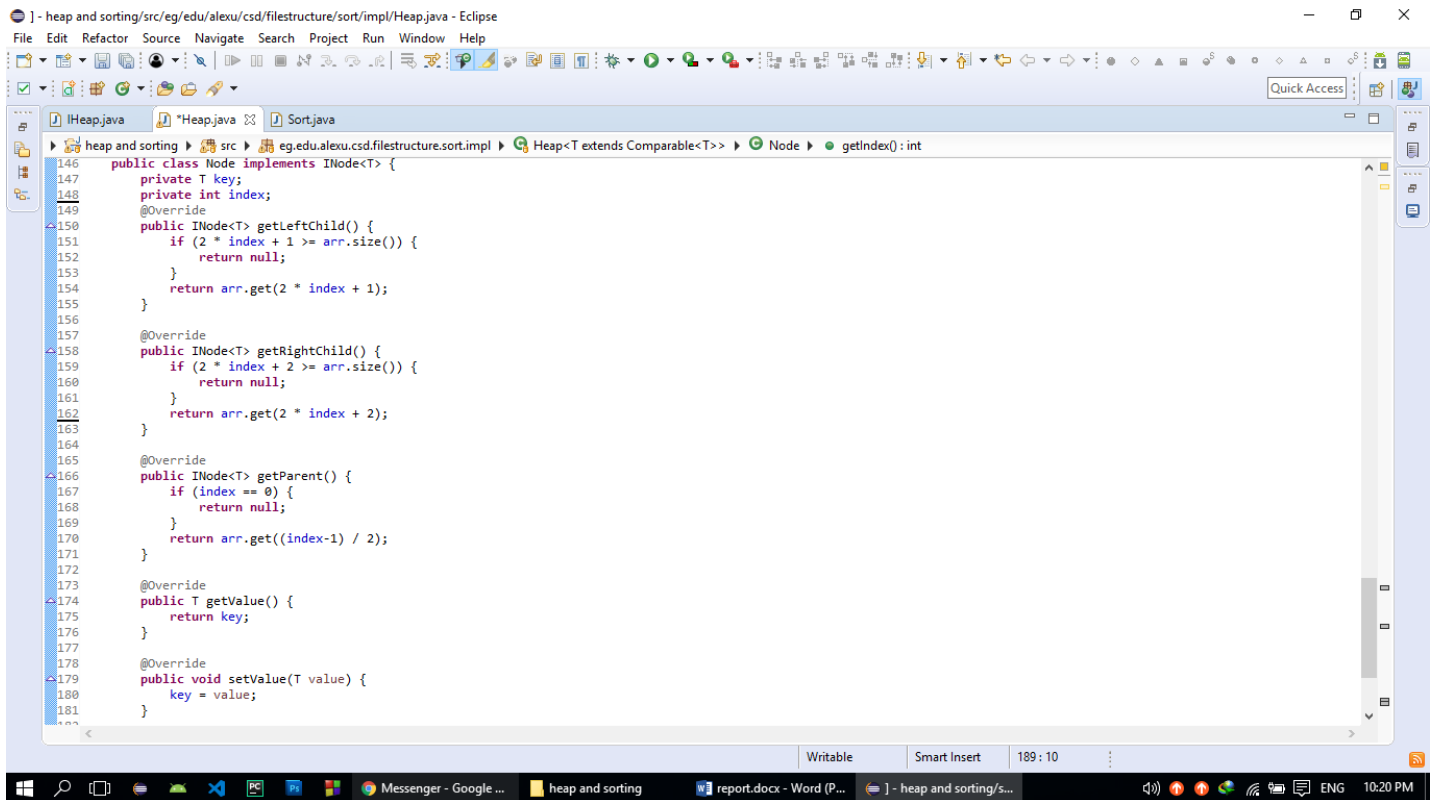
The screenshot shows the Eclipse IDE with the file `Heap.java` open. The code implements a binary heap structure. It includes methods for adding nodes, increasing keys, and building the heap. The `build` method iterates through the array and calls `heapify` on each element. The `heapify` method is not fully visible in this snippet but is referenced at the end of the `build` method.

```
83     arr.add(node);
84     increaseKey(arr.size() - 1);
85 }
86
87 private void increaseKey(int index) {
88     int parent = 0;
89     while (index > 0 && (arr.get(index).getValue().compareTo(arr.get(parent).getValue()) < 0) {
90         parent = ((Node) arr.get(index).getParent()).getIndex();
91         T temp = (T) arr.get(index).getValue();
92         arr.get(index).setIndex(arr.get(parent).getIndex());
93         arr.get(index).setParent(arr.get(parent).getParent());
94         arr.get(index).setParent().setIndex(index);
95         arr.get(index).setParent().setValue(temp);
96         index = parent;
97     }
98 }
99
100 @Override
101 public void build(Collection<T> unordered) {
102     if (unordered == null) {
103         throw new RuntimeException(new Error());
104     }
105     if (unordered.size() == 0) {
106         return;
107     }
108     heapsize = unordered.size();
109     arr = new ArrayList<INode<T>>();
110     Iterator<T> iter = unordered.iterator();
111     while (iter.hasNext()) {
112         INode<T> node = new Node();
113         ((Heap<T>).Node) node.setIndex(arr.size());
114         node.setValue(iter.next());
115         arr.add(node);
116     }
117     for (int j = arr.size() / 2; j >= 0; j--) {
118         heapify(arr.get(j));
119     }
120 }
```

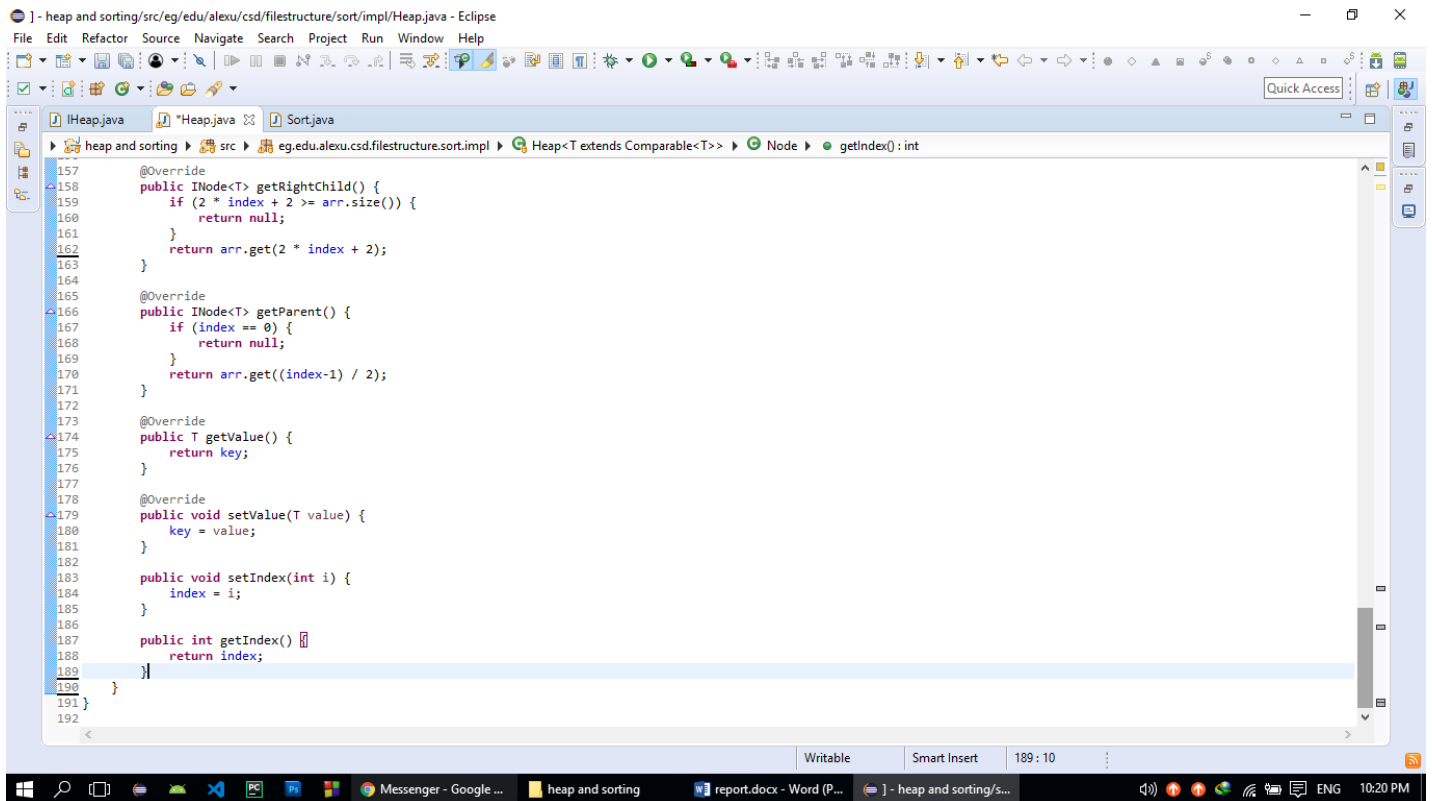


The screenshot shows the Eclipse IDE with the file `Heap.java` open, displaying the continuation of the `heapSort` method and the `Node` class definition.

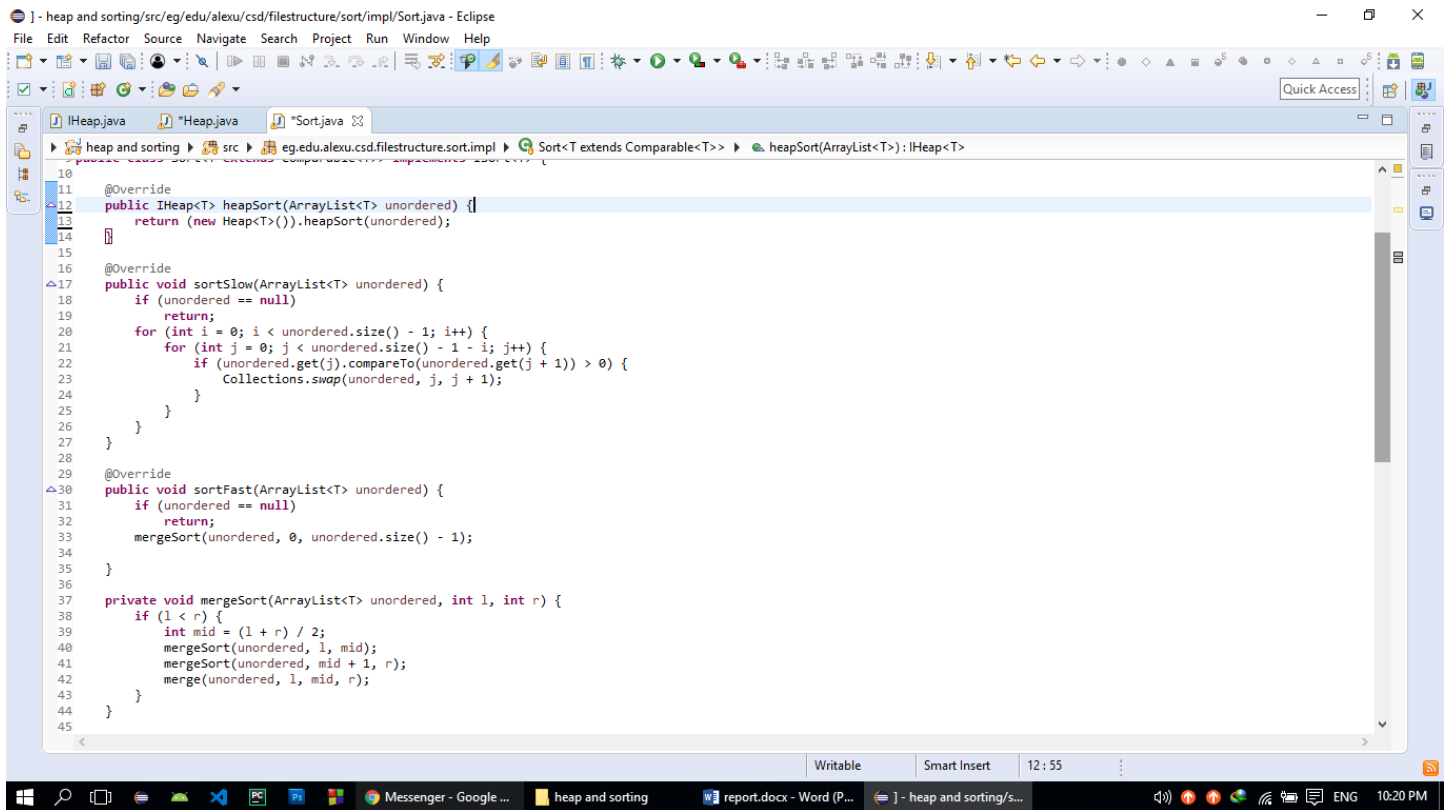
```
119 }
120 }
121
122 private int right(int i) {
123     return 2 * i + 2;
124 }
125
126 private int left(int i) {
127     return 2 * i + 1;
128 }
129
130 public IHeap<T> heapSort(ArrayList<T> unordered) {
131     build(unordered);
132     int num = unordered.size();
133     for (int i = num - 1; i >= 0; i--) {
134         T temp = (T) arr.get(0).getValue();
135         arr.get(0).setIndex(arr.get(i).getIndex());
136         arr.get(i).setIndex(0);
137         arr.get(i).setParent(arr.get(0).getParent());
138         heapify(arr.get(0));
139     }
140     return this;
141 }
142
143 /**
144  * @author SHIKO
145  */
146 public class Node implements INode<T> {
147     private T key;
148     private int index;
149
150     public Node() {
151     }
152 }
153
154 }
```



```
146 public class Node implements INode<T> {
147     private T key;
148     private int index;
149     @Override
150     public INode<T> getLeftChild() {
151         if (2 * index + 1 >= arr.size()) {
152             return null;
153         }
154         return arr.get(2 * index + 1);
155     }
156     @Override
157     public INode<T> getRightChild() {
158         if (2 * index + 2 >= arr.size()) {
159             return null;
160         }
161         return arr.get(2 * index + 2);
162     }
163     @Override
164     public INode<T> getParent() {
165         if (index == 0) {
166             return null;
167         }
168         return arr.get((index - 1) / 2);
169     }
170     @Override
171     public T getValue() {
172         return key;
173     }
174     @Override
175     public void setValue(T value) {
176         key = value;
177     }
178 }
```

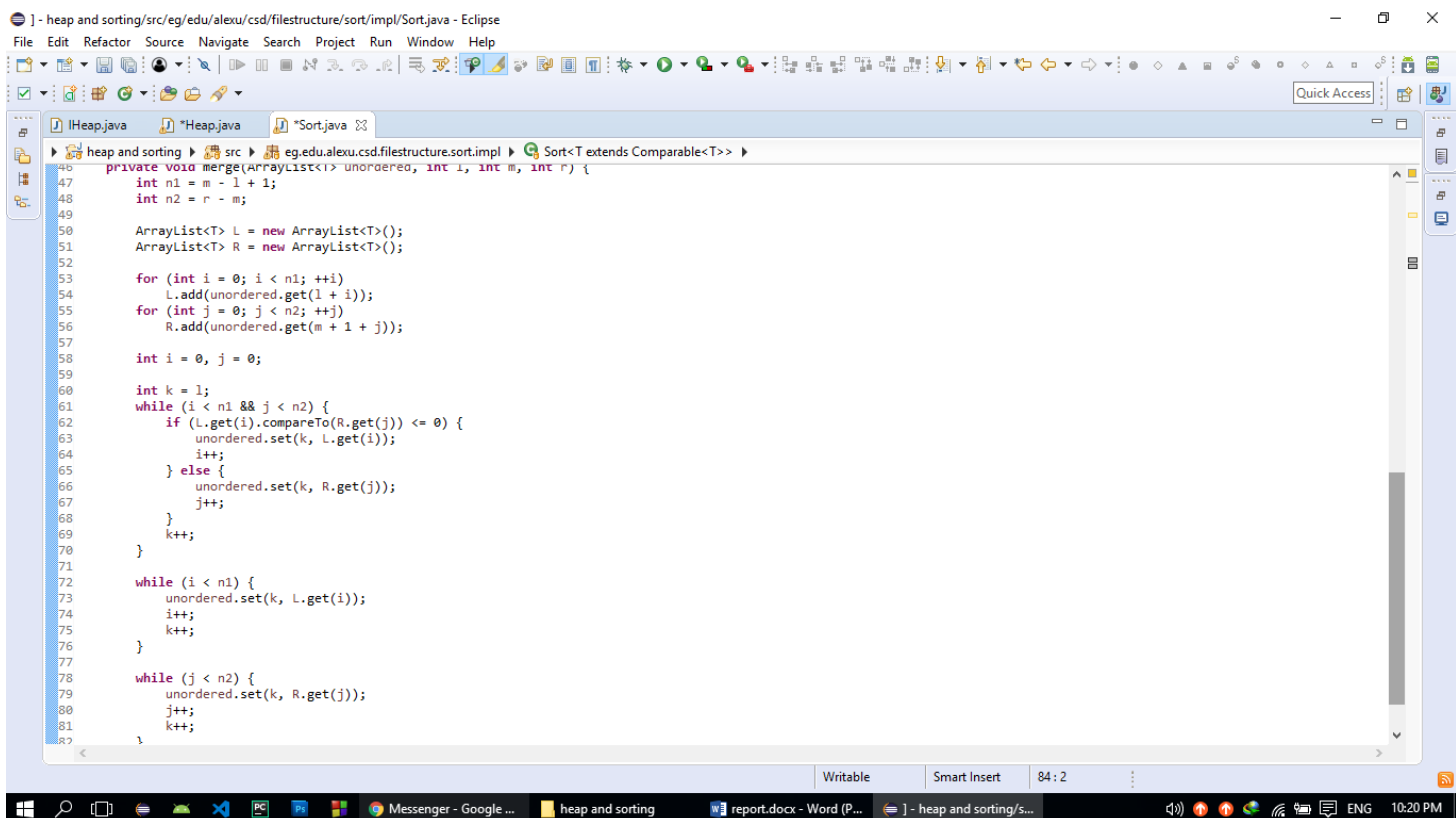


```
157     @Override
158     public INode<T> getRightChild() {
159         if (2 * index + 2 >= arr.size()) {
160             return null;
161         }
162         return arr.get(2 * index + 2);
163     }
164
165     @Override
166     public INode<T> getParent() {
167         if (index == 0) {
168             return null;
169         }
170         return arr.get((index-1) / 2);
171     }
172
173     @Override
174     public T getValue() {
175         return key;
176     }
177
178     @Override
179     public void setValue(T value) {
180         key = value;
181     }
182
183     public void setIndex(int i) {
184         index = i;
185     }
186
187     public int getIndex() {
188         return index;
189     }
190 }
191
192 }
```



The screenshot shows the Eclipse IDE with the file `Sort.java` open. The code implements a sorting interface `Sort<T>` that extends `Comparable<T>`. It includes methods for heap sort, slow sort, fast sort, and merge sort. The `heapSort` method is highlighted in blue.

```
10
11 @Override
12 public IHeap<T> heapSort(ArrayList<T> unordered) {
13     return (new Heap<T>()).heapSort(unordered);
14 }
15
16 @Override
17 public void sortSlow(ArrayList<T> unordered) {
18     if (unordered == null)
19         return;
20     for (int i = 0; i < unordered.size() - 1; i++) {
21         for (int j = 0; j < unordered.size() - 1 - i; j++) {
22             if (unordered.get(j).compareTo(unordered.get(j + 1)) > 0) {
23                 Collections.swap(unordered, j, j + 1);
24             }
25         }
26     }
27 }
28
29 @Override
30 public void sortFast(ArrayList<T> unordered) {
31     if (unordered == null)
32         return;
33     mergeSort(unordered, 0, unordered.size() - 1);
34 }
35
36 private void mergeSort(ArrayList<T> unordered, int l, int r) {
37     if (l < r) {
38         int mid = (l + r) / 2;
39         mergeSort(unordered, l, mid);
40         mergeSort(unordered, mid + 1, r);
41         merge(unordered, l, mid, r);
42     }
43 }
44
45
```



The screenshot shows the Eclipse IDE with the file `Sort.java` open. The code continues with the implementation of the `mergeSort` method, including the `merge` method. The `mergeSort` method is highlighted in blue.

```
46 private void mergeSort(ArrayList<T> unordered, int l, int m, int r) {
47     int n1 = m - l + 1;
48     int n2 = r - m;
49
50     ArrayList<T> L = new ArrayList<T>();
51     ArrayList<T> R = new ArrayList<T>();
52
53     for (int i = 0; i < n1; i++)
54         L.add(unordered.get(l + i));
55     for (int j = 0; j < n2; j++)
56         R.add(unordered.get(m + 1 + j));
57
58     int i = 0, j = 0;
59
60     int k = l;
61     while (i < n1 && j < n2) {
62         if (L.get(i).compareTo(R.get(j)) <= 0) {
63             unordered.set(k, L.get(i));
64             i++;
65         } else {
66             unordered.set(k, R.get(j));
67             j++;
68         }
69         k++;
70     }
71
72     while (i < n1) {
73         unordered.set(k, L.get(i));
74         i++;
75         k++;
76     }
77
78     while (j < n2) {
79         unordered.set(k, R.get(j));
80         j++;
81         k++;
82     }
83 }
```