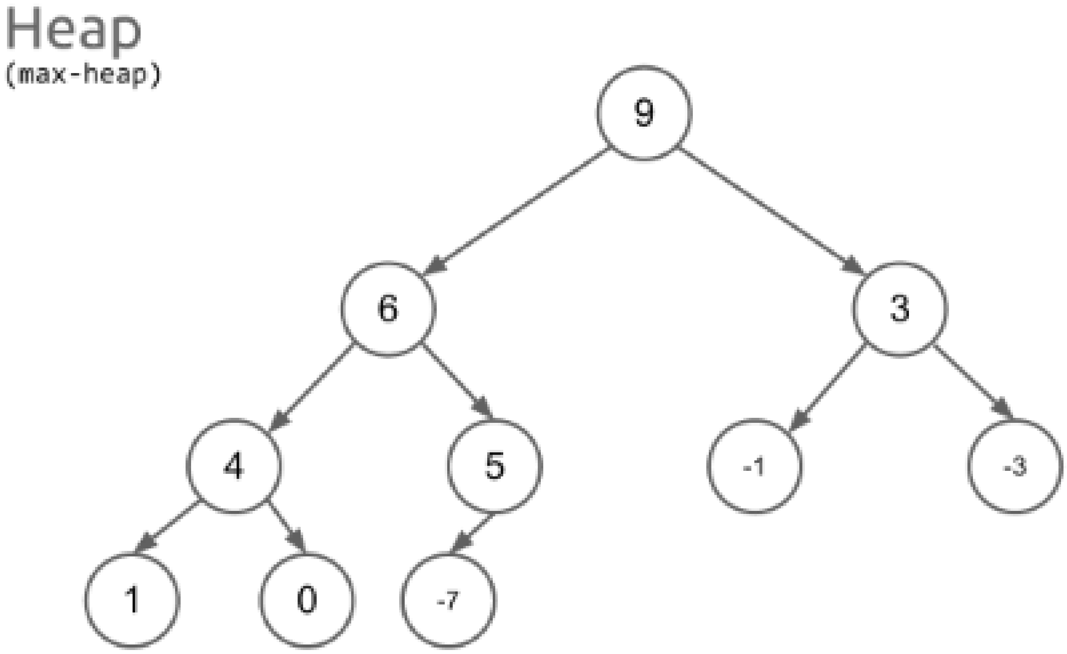
LAB1 REPORT

Implementing Binary Heap & Sorting Techniques



Aya Gamal (01)

Linh Ahmed (50)

16.03.2020

CS 2022

DESCRIPTION:

binary heap is defined as a binary tree with two additional constraints:

- **Shape property:** a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- **Heap property:** the key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children, according to some total order.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure

Requirements:

In this assignment, you’re required to implement some basic procedures and show how they could be used in a sorting algorithm:

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

DATA STRUCTURE:

- Heap is represented as an ArrayList of INode<T> (elements).
- Every Single element in the ArrayList is represented by a node that implements INode.
- Sort implements ISort interface.

PSEUDOCODE:

- Function heapify (INode<T> node) {

```
If node <>  null Then
        If  Leftchild <>  null  and  LeftChild.value > node.value  Then
                bigger ← left;
        Else Then
                bigger ← node
        End If
        If  Rightchild <>  null  and  RightChild.value > bigger.value  Then
                bigger ← right
        End If
        If  bigger <> node Then
                swap( node, bigger )
                If node <> getRoot() Then
                        heapify (node.getParent())
                End If
                heapify(bigger)
        End If
End If
}
```

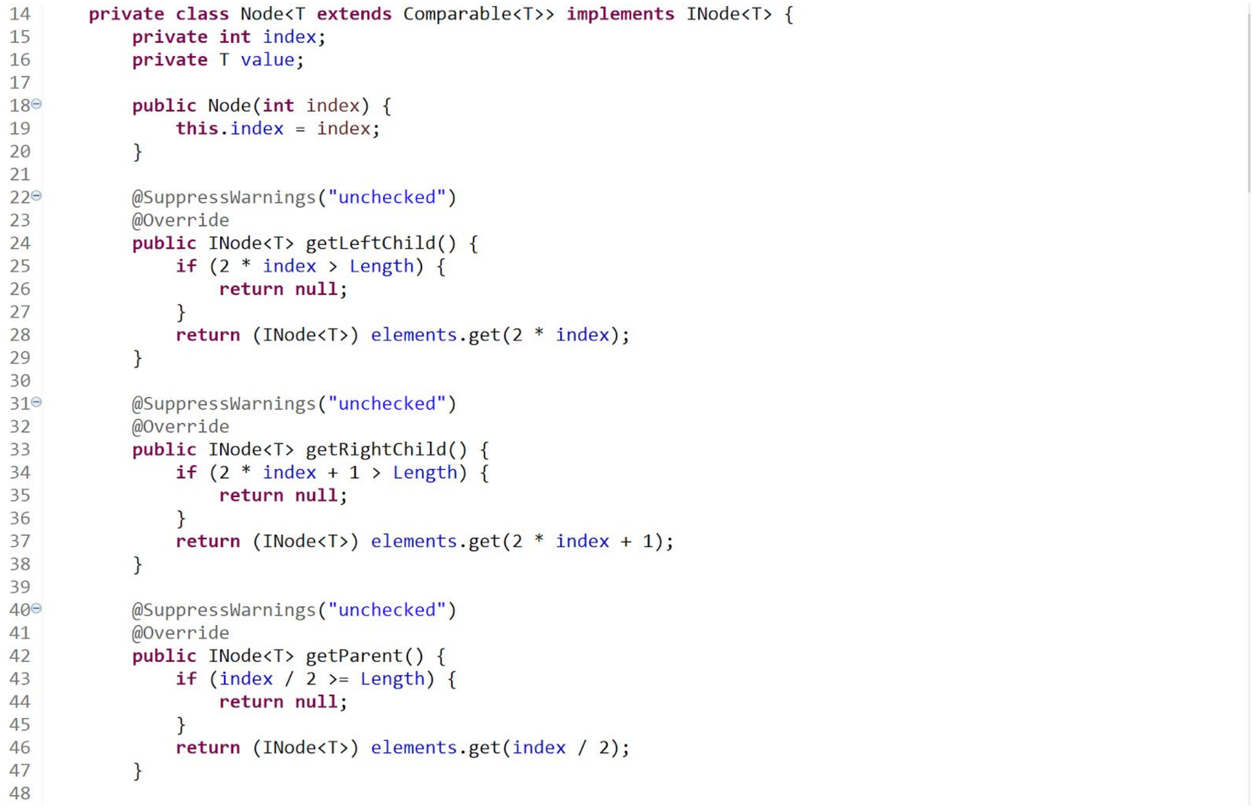
```
● Function build (Collection unordered)  {
  If unordered = null Then
      Length ← 0
      Return
  End If
  ArrayList<T> set ← new ArrayList<>(unordered)
  elements.add(0, null)
  For i from 0 → unordered.size()
      INode<T> k ← new Node<T>(i + 1)
      k.setValue(set.get(i))
      elements.add(i + 1, k)
      Length← Lenght+1
      heapifyUp(elements.get(i + 1))
  End Loop
  }
```

-
  ```
  Function T extract() {
  If elements.size() <= 1 Then
          return null
  End If
  T max ←  elements.get(1).getValue()
  elements.get(1).Value ← elements.get(Length).Value
  elements.remove(Length)
  Length← Length-1
  If Length <> 0 Then
          heapify(elements.get(1))
  End If
  return max
  }
  ```

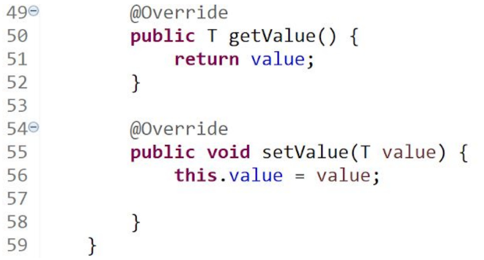
-
  ```
  Function sortSlow (ArrayList<T> unordered) {
  If unordered = null then
      return
  End If
  int i, j
  int n ← unordered.size()
  For i from 0 → n-1 do
      For j from 0 → n-i-1 do
          If unordered.get(j) > unordered.get(j + 1) Then
              T temp ← unordered.get(j)
              unordered.set(j, unordered.get(j + 1))
              unordered.set(j + 1, temp)
          End If
      End Loop
  End Loop
  }
  ```

CODE SNIPPETS:

- Class Node

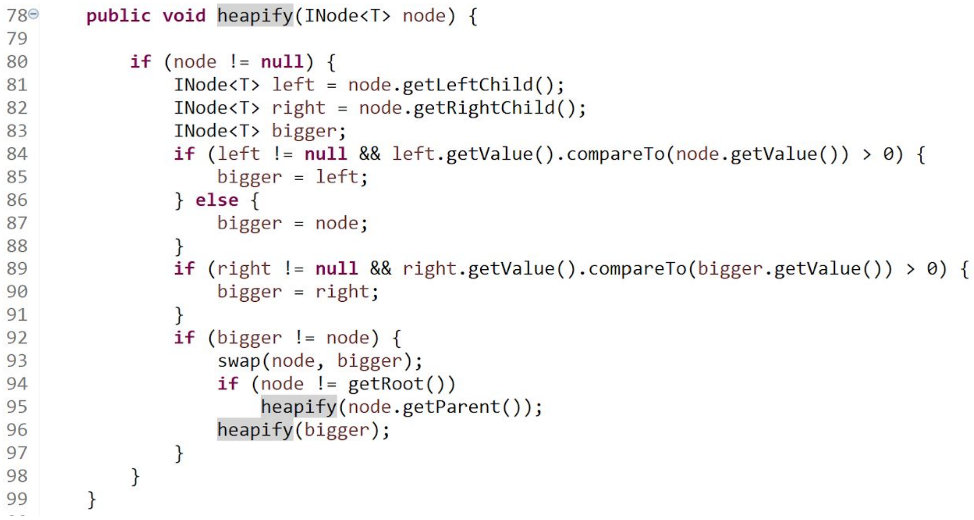


```java
    private class Node<T extends Comparable<T>> implements INode<T> {
        private int index;
        private T value;

        public Node(int index) {
            this.index = index;
        }

        @SuppressWarnings("unchecked")
        @Override
        public INode<T> getLeftChild() {
            if (2 * index > Length) {
                return null;
            }
            return (INode<T>) elements.get(2 * index);
        }

        @SuppressWarnings("unchecked")
        @Override
        public INode<T> getRightChild() {
            if (2 * index + 1 > Length) {
                return null;
            }
            return (INode<T>) elements.get(2 * index + 1);
        }

        @SuppressWarnings("unchecked")
        @Override
        public INode<T> getParent() {
            if (index / 2 >= Length) {
                return null;
            }
            return (INode<T>) elements.get(index / 2);
        }

```



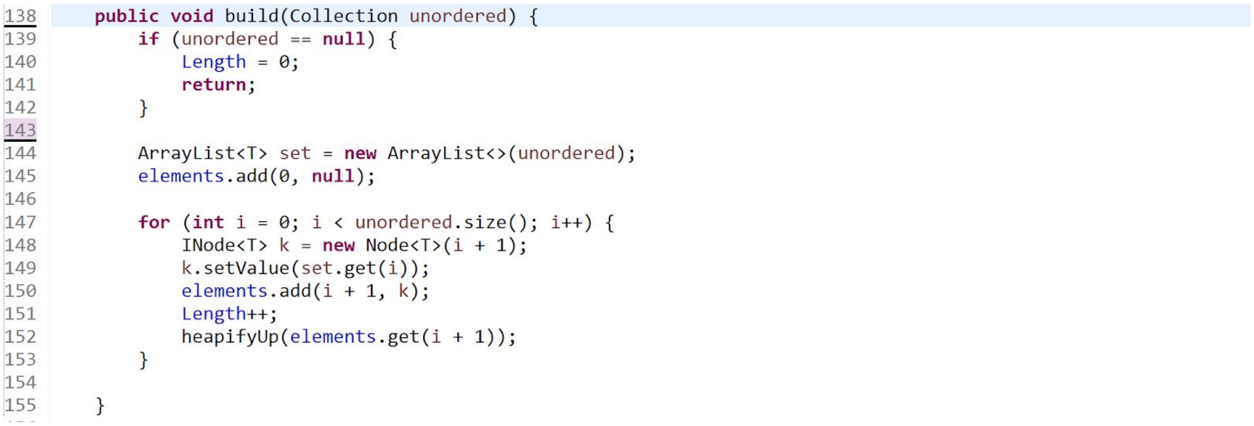
```java
        @Override
        public T getValue() {
            return value;
        }

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

        }
    }
```

- MAX-HEAPIFY



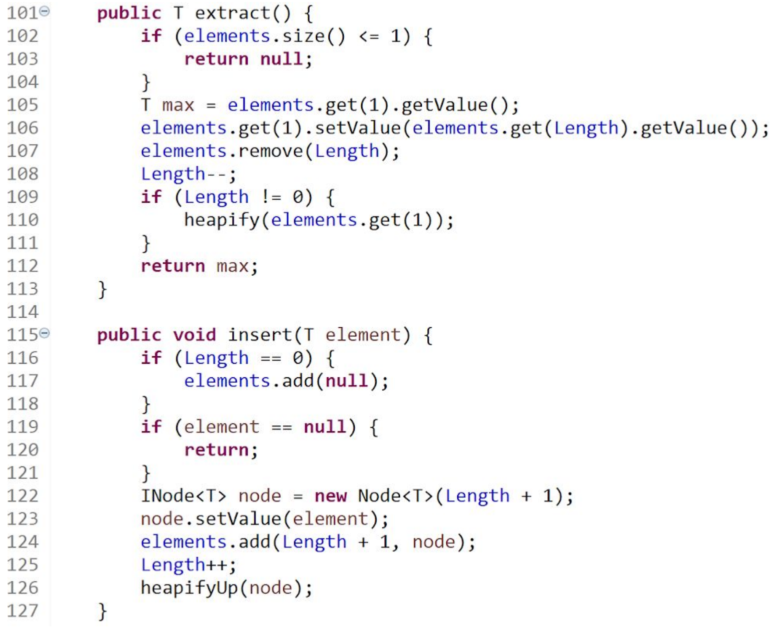
```java
    public void heapify(INode<T> node) {

        if (node != null) {
            INode<T> left = node.getLeftChild();
            INode<T> right = node.getRightChild();
            INode<T> bigger;
            if (left != null && left.getValue().compareTo(node.getValue()) > 0) {
                bigger = left;
            } else {
                bigger = node;
            }
            if (right != null && right.getValue().compareTo(bigger.getValue()) > 0) {
                bigger = right;
            }
            if (bigger != node) {
                swap(node, bigger);
                if (node != getRoot())
                    heapify(node.getParent());
                heapify(bigger);
            }
        }
    }
```

- BUILD-MAX-HEAP



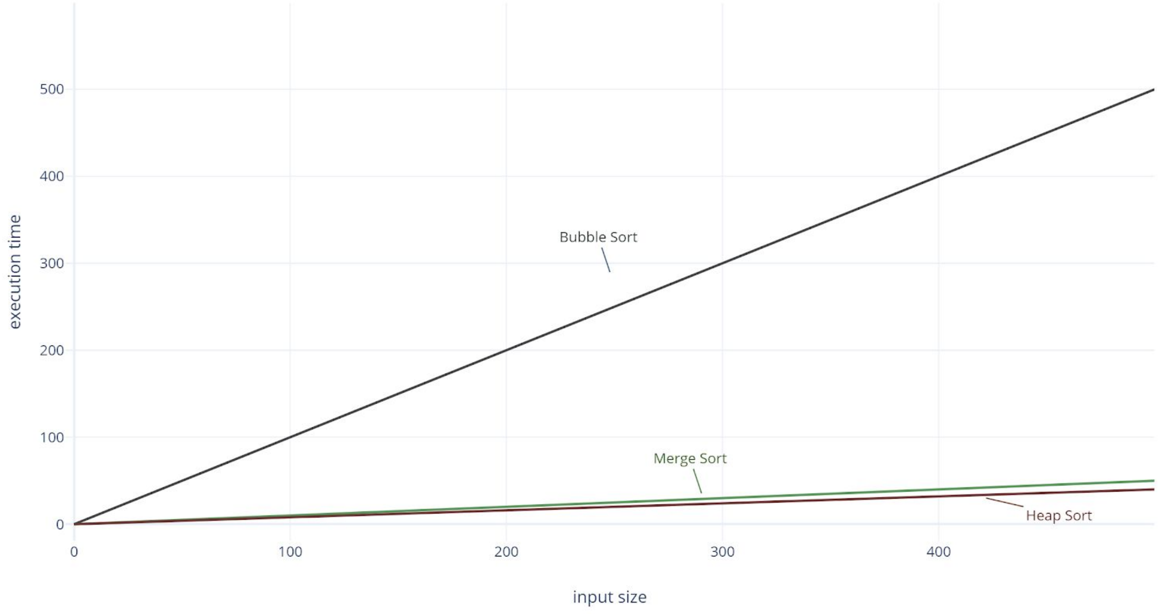
```java
    public void build(Collection unordered) {
        if (unordered == null) {
            Length = 0;
            return;
        }

        ArrayList<T> set = new ArrayList<>(unordered);
        elements.add(0, null);

        for (int i = 0; i < unordered.size(); i++) {
            INode<T> k = new Node<T>(i + 1);
            k.setValue(set.get(i));
            elements.add(i + 1, k);
            Length++;
            heapifyUp(elements.get(i + 1));
        }

    }
```

- MAX-HEAP-INSERT, and HEAP-EXTRACT-MAX



```
public T extract() {
    if (elements.size() <= 1) {
        return null;
    }
    T max = elements.get(1).getValue();
    elements.get(1).setValue(elements.get(Length).getValue());
    elements.remove(Length);
    Length--;
    if (Length != 0) {
        heapify(elements.get(1));
    }
    return max;
}

public void insert(T element) {
    if (Length == 0) {
        elements.add(null);
    }
    if (element == null) {
        return;
    }
    INode<T> node = new Node<T>(Length + 1);
    node.setValue(element);
    elements.add(Length + 1, node);
    Length++;
    heapifyUp(node);
}
```

Time Complexity Graph:


execution time
Bubble Sort
Merge Sort
Heap Sort
input size