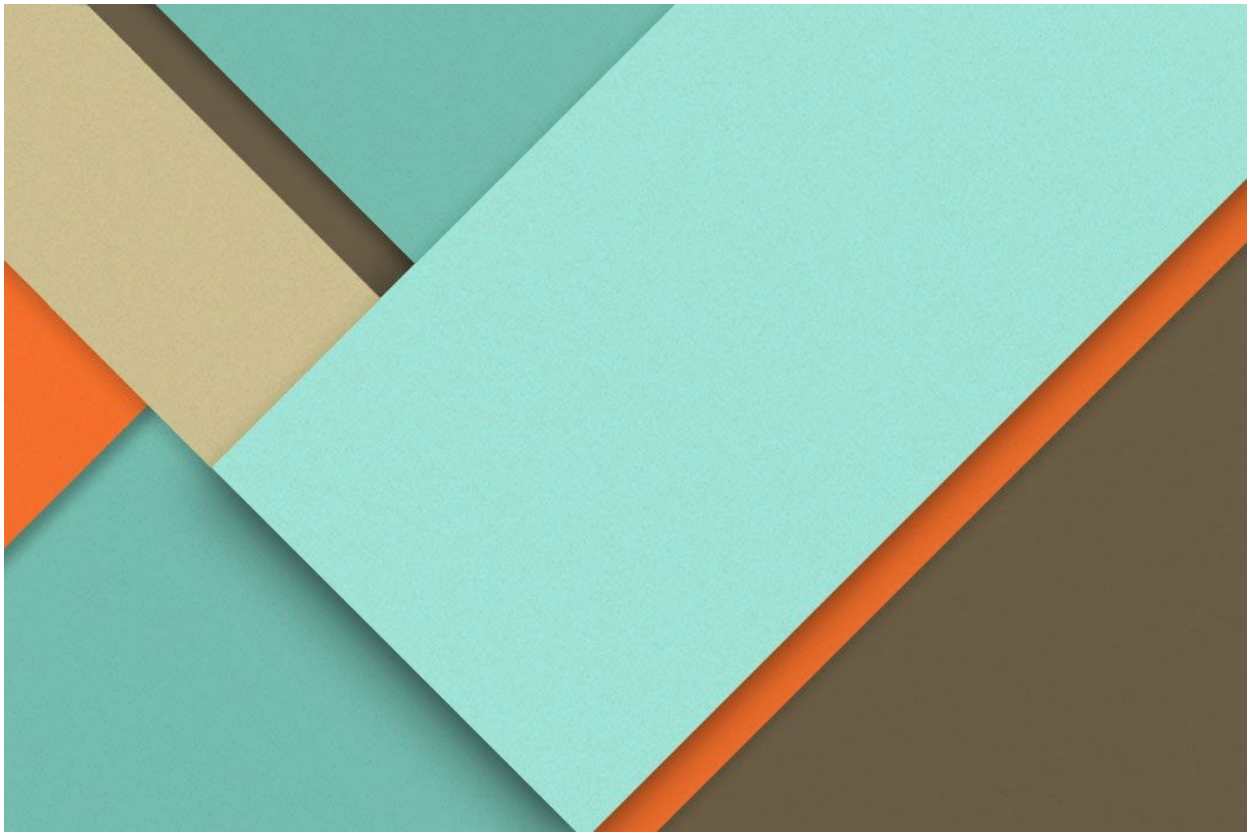


Data Structures 2 - Lab 1

Implementing Binary Heap & Sorting Techniques

Afnan Mousa Mabrouk 15

Shimaa kamal 36

Overview

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.

Goals

1. Implementation of ,MAX-HEAP and Build it,Insert,Extract.
2. Implementation of ,Sorting Techniques Such as ,Heap Sort ,Bubble Sort for slow sorting,Merge Sort for fast sorting .

Design Decisions

I. INode Interface

Each node represent element in the tree which have :

A. Value .
B. Parents .
C. Left child .
D. Right child.
E. Index (i).

- The tree is represented by an array of Nodes.
- The Parent of the node gets by equation (i-1)/2.
- The Left child of the node gets by equation (i*2)+1.
- The Right child of the node gets by equation (i*2)+2.

II. IHeap Interface

Each Heap represent Tree which have array of nodes and that heap have many properties such as :

A. Size .
B. Root .
C. Extract node .
D. Insert node.

E. Build Heap.
F. Heapify

- The Size gets the number of nodes in the array.
- The Root gets the node of index zero in Array.
- The mechanism of Insert nodes in heap work by inserting nodes in the index equal the size of heap in array and increment the size then compare this node and its parent if the value of its parent is lower than it swap their values while the condition does not occur the mechanism of insert is end this work in O(log n) .
- The mechanism of Extract nodes from heap working by swapping the root and last element in the heap then removing the last and finally calling the heapify O(log n).
- The mechanism of Build heap is to make a for loop of all elements in the Collection and then call the Insert to insert element by element O(log n) .
- The Heapify method runs in O(lg n) time, it makes sure that the node is in its right place in the heap by making sure that all its children are less than it.

III. ISort Interface

Each node represent element in the tree which have :

A. Heap Sort.
B. Fast Sort.
C. Slow Sort.

- The Heap Sort runs in O(n lg n) time, it depends on extracting element by element from the heap and then putting it in the last place in the heap which leads to a sorted heap.
- The Fast Sort runs in O(n lg n) time as Merge Sort which separates in two parts the main part is the merge function which separates the array to many small parts recursive and then call the MergeSort function which compares between small arrays and sort it .
- The Slow Sort runs in O(n lg n) time as Bubble Sort which work by two for loop first hold the element in index (i) and searches about the smallest element of it .

Sorting Techniques

This code in the Try class.

```
public static void main(String[] args) {

    LinkedList<ArrayList<Integer>> Input=new LinkedList<A
    ArrayList<Integer> array1 = new ArrayList();
    ArrayList<Integer> array2 = new ArrayList();
    ArrayList<Integer> array3 = new ArrayList();
    ArrayList<Integer> array4 = new ArrayList();
    ArrayList<Integer> array5 = new ArrayList();
   // System.out.print("Slow Sort time"+" "+"   ");
    for(int i=0;i<10;i++){
        Random r = new Random();
        int val = r.nextInt( bound: 1000);
        array1.add(val);
    }
    for(int i=0;i<100;i++){
        Random r = new Random();
        int val = r.nextInt( bound: 1000);
        array2.add(val);
    }

    for(int i=0;i<1000;i++){
        Random r = new Random();
        int val = r.nextInt( bound: 1000);
        array3.add(val);
    }

    for(int i=0;i<10000;i++){
        Random r = new Random();
        int val = r.nextInt( bound: 1000);
        array4.add(val);
    }
```

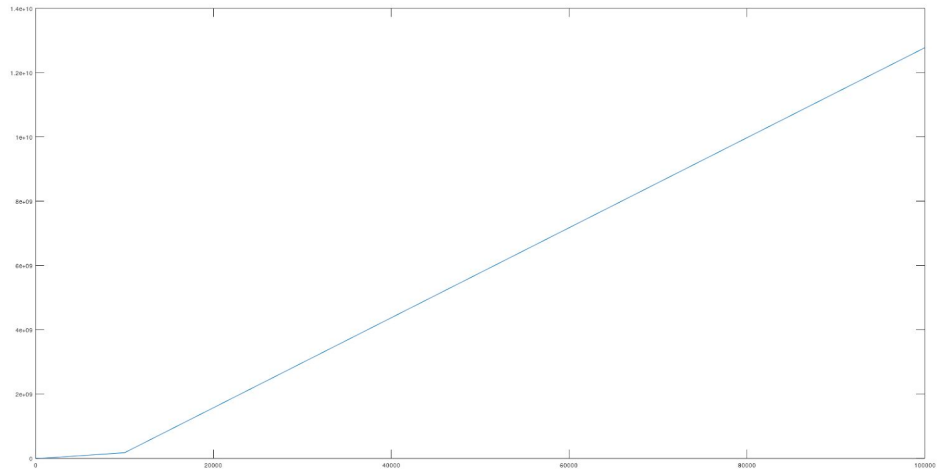
```
for(int i=0;i<10000;i++){
    Random r = new Random();
    int val = r.nextInt( bound: 1000);
    array4.add(val);
}

for(int i=0;i<100000;i++){
    Random r = new Random();
    int val = r.nextInt( bound: 1000);
    array5.add(val);
}
Input.add(array1);
Input.add(array2);
Input.add(array3);
Input.add(array4);
Input.add(array5);
ISort heap1=new Sort();
System.out.println();
System.out.print("Slow Sort time");
for(int i=0;i<Input.size();i++){
    long startTime = System.nanoTime();
    heap1.sortSlow(Input.get(i));
    long endTime = System.nanoTime();
    System.out.print(" "+(endTime-startTime)+"   ");
}
System.out.println();
System.out.print("Fast Sort time");
for(int i=0;i<Input.size();i++){
    long startTime = System.nanoTime();
    heap1.sortFast(Input.get(i));
    long endTime = System.nanoTime();
    System.out.print(" "+(endTime-startTime)+"   ");
}
```

OUTPUT:

```
Slow Sort time 93300   1677300   15997200   165241800   15392659100
Fast Sort time 190300   527600   4112200   19229800   77938400
```

Comparison between O(n lg n) and O(n^2)

O(n^2)



O(n lg n)