Data Structures 2

Lab 1

Implementing Binary Heap & Sorting Techniques

رامز ماهر فيكتور - 24

يوسف شريف نشأت - 74

Contents

Data Structures 2 - Lab 1

Implementing Binary Heap & Sorting Techniques

1 Binary Heap

1.1 HeapNode

The class HeapNode -that implements the interface INode<T>- is a simple data structure that consists of the data held by the node, a reference to its right & left child and parents , also the value of its index in the Arraylist of the heap.

1.2 MyHeap<T>

This class implements the interface IHeap<T>, it consists of an Arraylist that holds all the nodes, its first node is a dummy node so that each element of the arraylist has a true value in the heap, and some methods that shall be discussed in the methodology part.

1.3 Methodology

Compare

Takes values of two nodes and determines if both are equal or the first one bigger than the other it returns false, else if the first one is smaller than the second it returns true

HasLeftChild & HasRightChild

Returns true if node has child -left or right depending on method- And false if not.

Heapify

Assumes that the whole heap is organized as MAX-HEAP and checks if the node implies this assumption with its children, it uses the methods above do its task, and if the current node does not imply the assumption this method will recursively fix the heap.

minHeap

Does the same job as heapify but instead of fixing the heap by going downward-recursively fix the heap by going in the direction of children – it goes in the direction of the parents.

Extract

Switches the root after storing it's value of the heap -largest element- by the last element and removes it then fixes the heap by using heapify

Insert

Insert an element to the heap and then fixes the heap by using min heap.

Build

Takes an unsorted list and puts it in a MAX-HEAP form using heapify in linear time.

Swap

Swap the values of two nodes without changing the actual nodes locations.

2 Sorting Algorithms

2.1 $O(n^2)$ Sorting

The $O(n^2)$ Sorting Algorithm chosen is **Insertion Sort.**

This algorithm splits the array into two partitions: Sorted and Unsorted. Every element in the unsorted partition is chosen and is put in its correct place in the Sorted partition. This means that the selection process is done in constant time, while the insertion process can be done in linear time in the worst case because it can include shifting all the elements of the Sorted partition.

No additional memory element or data structure is needed, and the sorting procedure is done **in-place.**

```java
private void insertionSort(ArrayList<T> unordered) {

    for (int selectedItem = 1; selectedItem < unordered.size(); selectedItem++) {
        int j;

        for (j = 0; j < selectedItem; j++) {
            int sign = unordered.get(j).compareTo(unordered.get(selectedItem));
            if (sign > 0)
                break;
        }
        if (j < selectedItem) {
            T temp = unordered.get(selectedItem);
            shiftArray(unordered, j, selectedItem);
            unordered.set(j, temp);
        }
    }
}
```

Implementation of the Insertion Sort

2.2 O(nlogn) Sorting

The O(nlogn) Sorting Algorithm chosen is the **QuickSort** with **Random Pivot** chosen**.**

The QuickSort algorithms works as follows:

Step 1: Chose a random element in the array to be the pivot.
Step 2: Swap the pivot with the first element in the array.
Step 3: Divide the remaining part of the array into two partitions: <pivot and >=pivot.
Step 4: Swap the pivot with the last element in the first partition.
Step 5: Recursively return to Step 1 in each of the two partitions.

No additional memory element or data structure is needed, and the sorting procedure is done **in-place.**

```java
private void quickSort(ArrayList<T> unordered, int first, int last) {
    int size = (last - first) + 1;
    if (size <= 1)
        return;

    Random random = new Random();
    int randomNum = first + random.nextInt(size);
    swap(unordered, first, randomNum);

    int pivot = first;
    int lastSmaller = first;

    for (int firstUnknown = first + 1; firstUnknown <= last; firstUnknown++) {
        int sign = unordered.get(firstUnknown).compareTo(unordered.get(pivot));
        if (sign < 0) {
            swap(unordered, firstUnknown, lastSmaller + 1);
            lastSmaller++;
        }

    }
    swap(unordered, pivot, lastSmaller);

    quickSort(unordered, first, lastSmaller);
    quickSort(unordered, lastSmaller + 1, last);

}
```

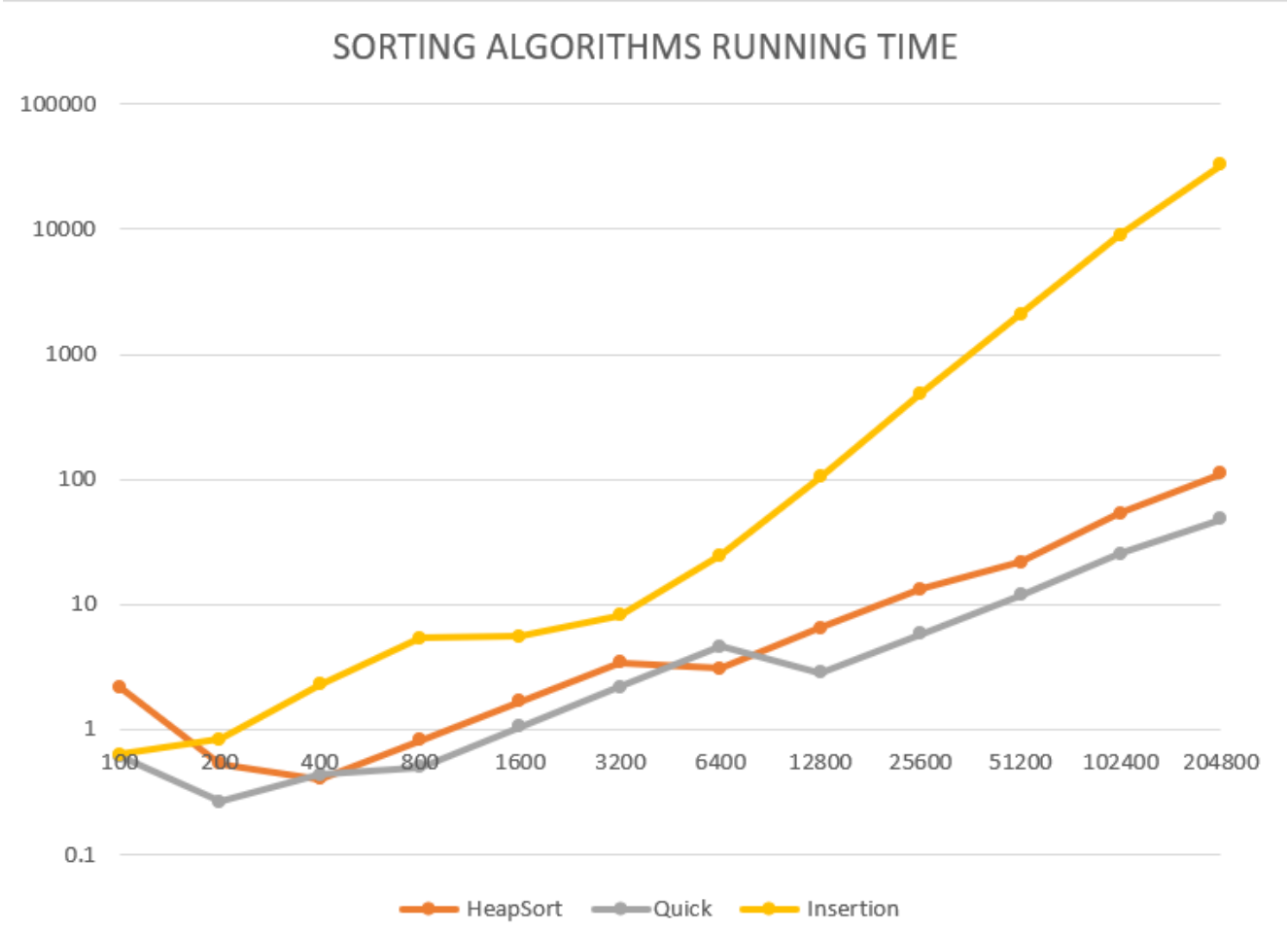
Implementation of the QuickSort

2.3 Heapsort

Builds a heap in linear time then start swapping the root element by the last element and heapify in O(log n) the whole heap except for the element at an indicated index -which is a variable that gets decreased by each swap to decrease the size of the array without removing the element- so that after each swap we are sure the biggest element is thrown in the back and so when the index variable reaches 1 we are sure that the array in sorted.

2.4 Analysis of Sorting Algorithms

An experiment was conducted to compare the different behaviors and running times of the different sorting algorithm against different input sizes.
A random dataset of size n was generated and fed to each of the three different sorting algorithms where $100 \leq n \leq 204{,}800$. For every dataset, the running time of the algorithm was calculated.
In the end, the relationship between the execution time of the sorting algorithm versus the input size was plotted, and the following results were found.



4 Test Cases

The program has passed all of the tests as shown in the figure below.