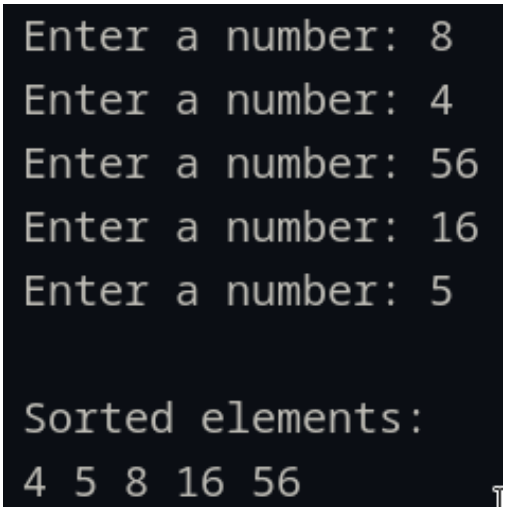
Assignment 6: Heaps and Trees
Part 1: Heaps
CS3305/W01 Data Structures

Casey Hampson

September 25, 2024

Program Output



Changes Needed

Since I didn't just copy/paste the listing in the book (I changed some style and added/removed comments for my own personal clarity), I wanted to just write the (three) changes I made to convert it to a MIN heap.

In the `AddObject()` method for a MAX heap, we would test if the current node is larger than its parent, in which case we swap the two. This is done with the `compareTo()` function, which returns a positive result if the first element is larger than the second. For our MIN heap, we want to do the opposite: if the current node is *less* than its parent, we do the swap. Since the `compareTo()` function returns a negative result if the first argument is less than its parent, all we need to do is swap the comparator in the if statement.

Similar logic applies to the `RemoveRoot()` method. Instead of testing for if the children are larger than one another and if the current node is larger than its largest child, we look for if the children are smaller than one another and if the current node is smaller than its smallest child, which involves a switching of the comparator in the if statements.

Source Code

```
// Name: Casey Hampson
// Class: CS 3305/W01
// Term: Fall 2024
// Instructor: Sharon Perry
// Assignment: 6-Part-1-Heaps

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;


class Heap<E extends Comparable<E>> {
    final private List<E> list = new ArrayList<>();


    public Heap() {}

    public void AddObject(E obj) {
        // add it to the end of the list (heap)
        this.list.add(obj);

        // make the index of this new node the ``current node''
        int current_idx = list.size() - 1;

        // while it's not the root, we swap it with its parent if its larger
        while(current_idx > 0) {
            // grab the parent's index:
            int parent_idx = (current_idx - 1)/2;
            // swap if current node is larger than parent node
            if (this.list.get(current_idx).compareTo(this.list.get(parent_idx)) < 0) {
                // swap
                E temp = this.list.get(current_idx);
                this.list.set(current_idx, this.list.get(parent_idx));
                this.list.set(parent_idx, temp);
            } else {
                // otherwise, the node is in the correct position and we can quit the loop
                break;
            }
            // we now set the current index to the new parent index (i.e. itself)
            // to keep swapping it up
            current_idx = parent_idx;
        }
    }



    public E RemoveRoot() {
        // bounds checking
        if (this.list.isEmpty()) return null;

        // grab the root
        E root = this.list.get(0);
```

```
        // set the new root to be the final element in the list, and remove that final element
        this.list.set(0, this.list.get(this.list.size() - 1));
        list.remove(list.size() - 1);

        // keep track of this node's index and keep propagating it down
        int current_idx = 0;
        while (current_idx < this.list.size()) {
            // grab the indices of its two children
            int left_child_idx = 2*current_idx + 1;
            int right_child_idx = 2*current_idx + 2;

            // if this left-child's index is greater than or equal to the list size,
            // then the index that we are currently at is the last one, so we are done
            if (left_child_idx >= this.list.size()) break;
            // otherwise, it has children, so we find the max between them,
            int max_idx = left_child_idx;
            // if we have no right_child, i.e. the right_child index isn't less than the this.list size
            // we can just continue with the current max index as the left child index,
            // else the right_child index is the max
            if (right_child_idx < this.list.size()) {
                if (this.list.get(max_idx).compareTo(this.list.get(right_child_idx)) > 0) {
                    max_idx = right_child_idx;
                }
            }

            // now we swap our current node with either of its children, whichever happens to be greate
            // if the node is greater than both, we are done
            if (this.list.get(current_idx).compareTo(this.list.get(max_idx)) > 0) {
                E temp = this.list.get(current_idx);
                this.list.set(current_idx, this.list.get(max_idx));
                this.list.set(max_idx, temp);
                // also swap the indices so that we follow our node
                current_idx = max_idx;
            } else break;
        }

        return root;
    }

    // helper functions
    public int      Size() { return this.list.size(); }
    public boolean  IsEmpty() { return this.list.isEmpty(); }
    public void     Print() { this.list.forEach(System.out::println); }

}



public class P1 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Integer> list = new ArrayList<>();
```

```
        for (int i=0; i<5; i++) {
            System.out.printf("Enter a number: ");
            list.add(sc.nextInt());
        }
        System.out.println();

        Heap<Integer> heap = new Heap<>();
        list.forEach(obj -> heap.AddObject(obj));

        System.out.printf("Sorted elements:\n");
        while (!heap.IsEmpty()) {
            System.out.printf("%d ", heap.RemoveRoot());
        }
        System.out.println();

        sc.close();
    }
}
```