Homework 3 – Answers to Questions

Jade Pearl

CMSC 451-7380

Professor Dennis Didulo

**Question 1**:

Below is the recursion tree for bubbleSort. The triangle is the call to bubbleSort, the circles are the calls to sort, and the squares are the calls to bubble. The numbers in the circles represent the value of i and the numbers in the squares represent the value of j.

3

3

2

3

2

1

The best case would have less nodes in the recursion tree because no swaps are performed as the best case calls for an array that is already sorted and worst case calls for an array in the reverse order than what we want it to be. The time complexities of the two cases are different.

**Question 2**:

Where the value of *n* is 4, we notice three branches of the sort and bubble calls contain 2, 3, and 4 nodes. That can be expressed by the summation . Since there are two additional nodes outside those branches, the root node and the leaf node (last node of the call to sort), we can add 2 to the summation.

Try 42-4-1 = 11 which is the correct number of nodes.

This allows for the conclusion that the Big Θ for execution time is Θ(n2) in bubble sort.

From the recursion tree, we see that the height of the tree is the length of the rightmost branch + 1 (includes the root node). So the height *h*(*n*) = *n* + 1. So the efficiency of memory for Big Θ is Θ(n) because *h* ɛ Θ(n).

**Question 3**:

The implementation I have for the SortedPriorityQueue is as follows:

public class SortedPriorityQueue {

    private int[] array;

    private int size;

    private int cap; //the size of the queue, not to be confused with size

    public SortedPriorityQueue(int cap) {

        this.cap = cap;

        this.array = new int[cap];

        this.size = 0;

    }

    public void add(int value) {

        if (size >= cap) {

            throw new RuntimeException("The priority queue is full.");

        }

        // Find the index where the value should be inserted to maintain sorted order.

        int index = size;

        while (index > 0 && value < array[index - 1]) {

            array[index] = array[index - 1];

            index--;

        }

        array[index] = value;

        size++;

    }

    public int removeElement() {

        if (size == 0) {

            throw new RuntimeException("The priority queue is empty.");

        }

        int elem = array[0];

        // Shift the remaining elements to fill the gap.

        for (int i = 1; i < size; i++) {

            array[i - 1] = array[i];

        }

        size--;

        return elem;

    }

    public int size() {

        return size;

    }

    public boolean isEmpty() {

        return size == 0;

    }

    public boolean isFull() {

        return size == cap;

    }

}

**Question 4**:

Analyzing the time complexity of the given void sort(int[] array) method:

As I had implemented, the *add* method of the *SortedPriorityQueue* in worst case has to search through the length of the array for the index where a value is to be inserted to maintain a sorted queue. There are no calls to other methods other than throwing an exception. In the worst case, the array needs to be searched to its length to insert a new element so the time complexity for the *add* method in the *SortedPriorityQueue* is Θ(*n*). The first loop in the sort method of calling the *add* method *n* times where *n* is the size of the queue, the time complexity is Θ(*n*2) because it will be calling the add method *n* times where the length of *n* is searched in worst case.

When the *remove* method is called in *SortedPriorityQueue*, it removes the first element in the queue (since it is already sorted) and shifts the other elements in the queue by going through the array. It only needs to go through the array once since it is already sorted this gives a time complexity of Θ(*n*) in that particular situation because it goes through the length of the queue to shift all of the elements. In the second loop of *void sort(int[] array)* where the *remove* method is called *n* times where *n* is the size of the queue, the time complexity is Θ(*n*2) because, just like the *add* method, the *remove* method is called *n* times and goes through the length of the array each time to remove the first element in the queue.

Since both loops are Θ(*n*2), the overall complexity is Θ(*n*2). According to the UMGC course resource material and another resource I used for this research, DigitalOcean, the Java priority queue has a time complexity of Θ(log *n*) for enqueue and dequeue methods so Θ(*n* log *n*) using these loops in *sort*. In worst case, Θ(*n* log *n*) time complexity is still better than the quadratic Θ(*n*2) time complexity. This implementation is therefore less efficient than the Java priority queue with its approach using heaps.

**Resources**

*Priority Queue Java | DigitalOcean*. (n.d.). Www.digitalocean.com. <https://www.digitalocean.com/community/tutorials/priority-queue-java>