**Module IN2002—Data Structures and Algorithms**

# Exercise Sheet 2 (Sample Answers)

Note that you may not have enough time to finish this during the tutorial slot. Get started, see

where you have difficulties, study, ask questions, and try again. You should aim to finish all

the questions before the next lecture.

Additional take away for you to take from this week’s exercises: when creating an algorithm, it is extremely helpful for you if you start with some higher-level abstraction of what you are thinking of implementing. Usually plain English suffices, but you could also use flow charts, if you prefer.

Note that every time you are defining an algorithm, you may use additional auxiliary functions. However, every time that you do so, you must also define these auxiliary functions.

You will find some sample answers below. We STRONGLY encourage you to only look at the sample answers below once you have created your own answers. Your justifications to support your answers might be slightly different, and your algorithms may also be different. This does not mean that they are wrong. If in doubt post your solution on Moodle, or ask a member of the module team during their office hours.

## Question 1

The array priority queue implementation provided in the lecture notes assumed a sorted array, where the method *add* contains a call to a method *insert* (see slide 13). Provide the algorithm for the method *insert* (in pseudocode). Explain in plain English the underlying principles of your algorithm (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

### Answer:

Explanation:

Since this is a sorted array, we must first find the array position where the new value belongs. We then must make room to insert it by moving the larger entries (higher priorities) up by one place. Once the space is available, it is just a matter of adding the new value in its rightful place.

Finding the correct position could be done using any search that can be performed in a sorted array. The example below uses sequential search.

void insert(array data, int size, int element) {

int position ← 1;

// find the position in the array where the new element

// needs to be added

WHILE (position < size AND data[position] < element)

position++;

// move all larger existing elements up one position

int inPlace ← size

WHILE (inPlace > position)

data[inPlace] ← data[inPlace-1];

inPlace--;

// insert the new element in its correct place

data[position] ← element;

}

* Note that finding the position here has been done using sequential search. It could also have been done using binary search. That would have improved the time complexity of the first loop, but not of the second. Hence, the overall complexity would remain unchanged.

## Question 2

Had the priority queue been implemented using an unsorted array, there would be some differences to the implementation of its basic operations (*isEmpty*, *add*, *extractMax*). Discuss which of these would need to be changed and why, while providing new algorithms for those requiring changes (again, using pseudocode). Explain in plain English the underlying principles of each of your algorithms (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

### Answer:

* isEmpty remains the same, as what really matters is simply whether there are any elements in the array or not.
* add would be faster and changed, as we would simply add the new element to the end of the existing array, as it doesn’t matter where we add it (unsorted).

Explanation: increment count to reflect that we have one more entry, and add the new entry to the end of the array.

void add(int elt) {

count++;

data[count] ← elt;

}

* extractMax would be slower and changed, as we would need to traverse the whole array keeping track of the largest value found so far and its position. Then remove that element and reshuffle all other elements that were in the array in positions after it, actually removing the largest element.

Explanation:

Traverse the array keeping track of the largest value and its position (using auxiliary variables). Once the largest value has been identified, we must move all elements that came after it to one position lower, effectively removing the highest value, and reduce the count by one (so that the last value is not counted twice). Once that is done, it is just a matter of returning the largest value (highest priority).

int extractMax() {

int largest ← data[1]

int position ← 1

int i ← 1

// find largest value and its position

WHILE i < count

i++

IF largest < data[i]

largest ← data[i]

position ← i

// reshuffle other elements

i ← position

WHILE i < count

data[i] ← data[i+1]

i++

count--

Return largest

}

## Question 3

Provide pseudocode for a queue implemented using an array. This implies the functions *isEmpty*, *enqueue*, and *dequeue* (see slide 43). Explain in plain English the underlying principles of each of your algorithms (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

### Answer:

Explanation:

For isEmpty, all we need to check is whether the queue contains any elements at all; count tells us how many elements are currently in the queue. So, if it is zero, it is empty; otherwise, it isn’t.

In the example below, I am using the front of the queue to be in the first position of the array (one could have done this the other way around). In that case, when adding something to the end of queue (enqueue), it just means adding to the end of the array.

Since we have the first element that got in the queue stored in the first position of the array, to dequeue something we just need to remove the element from the first position in the array. That means, keep track of the element that was in the first position (using some aux variable), move all other entries forward by one space, then reduce the size of the queue to show that is contains one fewer element. Only after that return the element that had been stored in the aux variable (and was at the front of the queue).

*public class ArrayQueue implements Queue {*

*private int[] a;*

*private int count = 0;*

*public ArrayQueue(int size) {a = new int[size];}*

*public boolean isEmpty() { return count == 0; }*

*public void enqueue(int elt) { a[count++] = elt; }*

*public int dequeue() {*

*int value = a[0];*

*int i = 1;*

*while (i<count) {*

*a[i-1] = a[i];*

*i++;*

*}*

*count--;*

*return value;*

*}*

*}*

## Question 4

Create an algorithm that given a queue of integers returns a queue with the same integers in the reverse order from the input one. Don’t forget to provide an explanation in plain English of the underlying principles of your algorithm (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

### Answer:

* + As always with algorithms, there are multiple possibilities for answering this question correctly. Here is one possible solution.

*Function reverseQueue(queueIn)*

// In this algorithm we will add each entry from the input queue into some auxiliary

// data structure (students could have used many different ones), and then from there

// into the output queue. In this example solution, we are using an array. Once all

// items have been added to the array, we can add them to the output queue by

// reversing their order. And once all items have been added to the output queue, it

// can be returned.

create aux array

index ← 0

WHILE !queueIn.isEmpty()

// if we are here, it means that here is still something in the queue to be added

// to the auxiliary array.

aux[index] ← queueIn.dequeue()

index ← index +1

// We have now placed all items from the input queue into the aux array. Now we

// must add the elements from the array into the output queue.

create a queue queueOut

index ← index -1

WHILE index >= 0

// By going through the aux array from back to front, we are enqueueing the

// entries in the reverse order of what they were in the input.

queueOut.enqueue(aux[index])

index ← index -1

// We can now simply return the output queue.

RETURN queueOut