French Azerbaijani University
Arash Habibi
Second year
2019-2020

Data Structures and Algorithms 1

Exercice session 4,5 : Queues

1 Theoretical background

1.1 Abstract data structures

In a previous practical work session, you discovered stacks. Stacks, queues and lists are called *abstract data structures*. They are *abstract* because they can be implemented with different types of concrete data structures : arrays or linked lists. Stacks, queues and lists are data structures in which you may store and recover data.

— in stacks, the recovered data corresponds always to the youngest data elements (those that were stored the most recently). For example if we put first 7, then 0, then 9 and, at last, 5 in a stack, the first element that we recover is 5, then 9, then 0 and, at last, 7. Stacks are also called *LIFO*s : *Last In First Out*.

— in queues, the recovered data corresponds always to the oldest data elements (those that are in the queue for the longest time). For example if we put first 7, then 0, then 9 and, at last, 5 in a queue, the first element that we recover is 7, then 0, then 9 and, at last, 5. Queues are also called *FIFO*s : *First In First Out*.

— in lists (which are NOT necessarily *linked* lists) the data may be recoverd from anywhere : the beginning, the end, or anywhere in the middle.

In this exercise session, the concrete data structure that we will use to implement queues are arrays.

1.2 Specification and test

The organisation of a queue is the same as the organisation of the people lining up to have an autograph of their favorite star. At one end, arriving people start lining up : they will have a long time to wait. At the other end, stands the star. At this end, as soon as people get their autograph (they are those who have waited the longest), they leave the line.

In the rest of this document, we will make integer queues, even if you can make queues with any data type. There are several operations that are interesting to implement in queues. For stacks, you implemented `push, pop, top, size` and `isEmpty`. With queues, we will need to :

— create a new empty queue ;
```
queue Q_new(void);
```

— store an element x in queue q ;
```
queue Q_enqueue(queue q, int x);
```

— remove the oldest element from queue q ;
```
queue Q_dequeue(queue q); :
```

— get the value of the oldest element in the queue ;
```
int Q_oldest(queue q);
```

— get the number of elements stored in the queue ;
```
int Q_size(queue q); :
```

— know whether a queue is empty or not
```
bool Q_isEmpty(queue);
```

— print on the terminal a string (label) followed by contents of the queue (for debug)
`void Q_show(queue q, char *label);` :

For example here is a set of function calls and the expected output :

```
queue q = Q_new();
Q_show(q,"beginning"); // prints only "beginning : "
q = Q_enqueue(q,5);
q = Q_enqueue(q,0);
q = Q_enqueue(q,7);
q = Q_enqueue(q,4);
Q_show(q,"before dequeue"); // prints : "before dequeue : 5 0 7 4"
q = Q_dequeue(q); // 5 is removed from the queue
q = Q_dequeue(q); // 0 is removed from the queue
Q_show(q,"after dequeue"); // prints : "after dequeue : 7 4"
```

In the following sections, you will have to decide what data structure you would use in different situations.

2 Infinite memory

In this section, we assume that we can use an array of 100000 integers to store the queue, and you can assume that our application will never add more than 100000 integers in the queue. So, as far as our application is concerned, we can assume that this space is infinite.

1. Apart from the array of 100000 integers, what are the minimum set of fields that are enough to enable you to carry out all of the required processing. For stacks, we had the array and the size of the stack (the number of stored elements). Are these two data enough for queues in this situation ?
2. In a file named `queue1.h`, write the definition of the queue structure and the list of function prototypes described above ;
3. In a file named `queue1.c`, define the functions themselves and test them in the main function.

Do the same work as in the previous section, but in file `queue2.c`.

3 No global data movements

In the previous section, there was no constraint on copying or moving data. You may probably have shifted or mirrored all the elements of the queue in one way or the other. In this section, we still have infinite memory space. But the extra constraint is that you must not touch the stored data. When you add an element in the queue, it must stay at the same memory location until it leaves the queue.

1. Does this condition change anything on the number and nature of the numbers that characterize a queue ? In order to use a queue, do you need anything else than the array, the index at which you must insert the next element, etc ?
2. In a file named `queue2.h`, write the definition of the queue structure and the list of function prototypes described above (the prototypes are the same as in `queue1.h`) ;
3. In a file named `queue2.c`, define the functions themselves and test them in the main function.

4 Finite memory

Finally, we assume that our queue will be used on a server that hardly ever stops. That means that we need to put a very great number of integers in this queue. Much more than 100000 or even 1000000000. We cannot put an upper bound on the number of data entering in the queue. But there will also be a great number of integers *leaving* the queue. We can assume that the maximum number of integers stored on the queue is 10. In other words, at any time, there will be, at most, 10 integers that have entered the queue and have not yet left the queue.

1. Does this condition change anything on the number and nature of the numbers that characterize a queue ? In order to use a queue, do you need anything else than the array, the index at which you must insert the next element, etc ? In particular, how can you detect that the queue is empty ? How can you detect that the queue is full ?
2. In a file named `queue3.h`, write the definition of the queue structure and the list of function prototypes described above (the prototypes are the same as in `queue1.h`) ;
3. In a file named `queue3.c`, define the functions themselves and test them in the main function.