French Azerbaijani University
Arash Habibi
Second year
2017-2018

Advanced C Programming

Practical Work 6 : makefiles and queues

1 makefiles

1. Come back to your first practical work session on colors, images and ppm files, and write a makefile that simply compiles the program for you (without making any object files). Writing simply `make` in the termina, should compile the program for you.
2. Make sure that if the executable file is up to date, writing `make` does NOT recompile the program.
3. Define a macro with all your source files ;
4. Define a macro with all your source files, but with ".o" instead of ".c" ;
5. make a rule which compiles the source files and produces objects files ;
6. make a rule which compiles the executable file from the object files ;
7. change the makefile such that running `make` compiles the modified source files (and only the modified source files) and updates the executable file ;
8. make a rule called `tamiz` which removes all the object files.

2 Queues

A *queue* is a data structure in which the data are waiting for their turn as in a real queue in front of a good bakery shop. This means that there are three possible operations on a stack :

— store a new element `(enqueue)` ;

— recall the value of the oldest element `(head)` ;

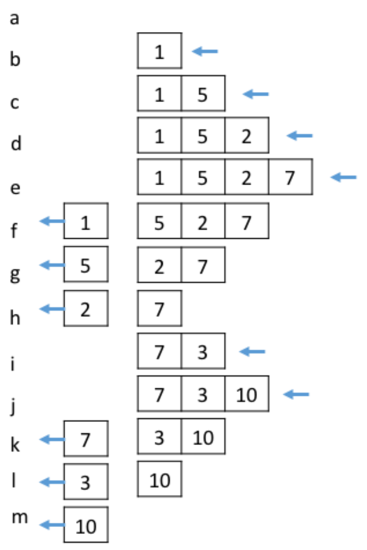
— remove the oldest element `(dequeue)` ;

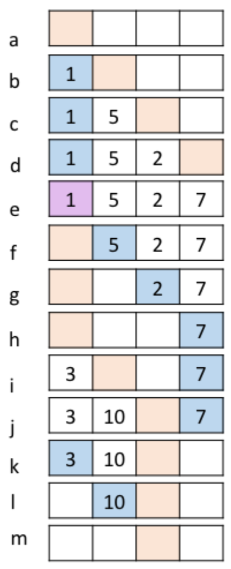
Here is an example of possible operations on a queue, with the corresponding state on figure 1.

Sometimes, a queue is also called a FIFO : *First In, First Out*, which means that the first element that entered the stack is the first one to get out.

We need to make a queue data structure, with the functions operating on it. We assume that there is a maximum number of data elements N in a queue. Let us choose $N = 4$.

```
(a) let q be a new empty queue   // size of queue : 0, head/tail un
(b) st = enqueue(st,1);   // size : 1, head : 1, tail : 1
(c) st = enqueue(st,5);   // size : 2, head : 1, tail : 5
(d) st = enqueue(st,2);   // size : 3, head : 1, tail : 2
(e) st = enqueue(st,7);   // size : 4, head : 1, tail : 7
(f) st = dequeue(st);     // size : 3, head : 2, tail : 7
(g) st = dequeue(st);     // size : 2, head : 5, tail : 7
(h) st = dequeue(st);     // size : 1, head : 7, tail : 7
(i) st = enqueue(st,3);   // size : 2, head : 7, tail : 3
(j) st = enqueue(st,10);  // size : 3, head : 7, tail : 10
(k) st = dequeue(st);     // size : 2, head : 3, tail : 10
(l) st = dequeue(st);     // size : 1, head : 10, tail : 10
(m) st = dequeue(st);     // size : 0, head/tail undefined
```





In this exercice, we choose to represent the queue by

- — an array of $N = 4$ integers ;
- — an integer `i_head` representing the index of the oldest element (the *head*) (in blue) ;
- — an integer `size` representing the number of stored elements in the queue ;

But in order to use the empty places left by dequeued elements, we can use them to put new elements. But this must be transparent for the user. The user should only have to use `enqueue` and `dequeue`.

1. What does the `%` operator do in C ?
2. What is the value of `3 % 10` ?
3. What is the value of `13 % 10` ?
4. What is the value of `(n+5) % 10` ?

For the following questions, do not forget to test each function before you start to write the next functions. Testing a function means : calling it with different values and seeing if the function behaves as expected. The `QU_show` function is used precisely to check the contents of a stack. Try the operations illustrated on figure 2. Between each operation, print the contents of the queue (`QU_show`) to check if the operation was performed as expected.

5. Write in C a function with prototype `int nextIndex(int ind, int max);` which for any value of `ind` comprised in `[0,max-2]` returns `ind+1` but for higher values, it should cycle from the beginning again. For example :
   - — `nextIndex(0,5)` should be `1` ;
   - — `nextIndex(1,5)` should be `2` ;

— `nextIndex(3,5)` should be `4` ;

— `nextIndex(4,5)` should be `0` ;

— `nextIndex(5,5)` should be `1` ;

— etc

6. With the structure mentioned above, we always know where is the oldest element (`i_head`), but we don't know where to put the next new element. Write in C a function with prototype `int QU_freeIndex(struct queue q);` which for a given queue returns the index of the place where we should pu the next new element ;
7. Define a data structure called `struct queue` composed of the elements mentioned above ;
8. Write a function with prototype `struct queue QU_new(void);` which returns a queue structure with size 0. We assume that the first element will be placed in index 0 ;
9. Write a function with prototype `void QU_show(struct queue);` which outputs the contents of the queue (starting with the head and finishing with the newest element ;
10. Write a function with prototype `int QU_size(struct queue q);` which, for a given queue, returns the number of elements stored in the queue ;
11. Write a function with prototype `struct queue QU_enqueue(struct queue q, int n);` which, for a given queue, adds `n` to the queue and returns the new queue, unless if the queue already contains N elements. In that case, the function should print an error message ;
12. Write a function with prototype `struct queue QU_dequeue(struct queue q);` which, for a given queue, removes the oldest element of the queue and returns the new queue, unless if the queue is empty. In that case, the function should print an error message.
13. Write a function with prototype `int QU_head(struct queue q);` which, for a given queue, returns the value of the oldest element of the queue, unless if the queue is empty. In that case, the function returns an error message.
14. Check if you can do all the operations described on figure 1 and 2.