Queues using arrays in C

Sam Serbouti serbouti@kth.se

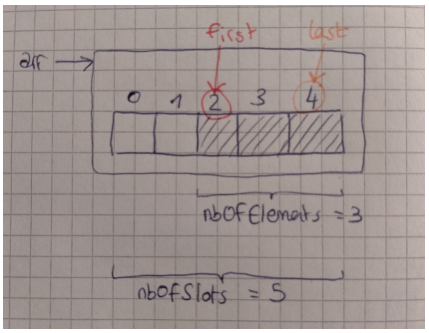
October 5, 2024

Introduction

The goal of this assignment is to implement a queue structure using arrays in C. This implies that our data structure will require several variables for it to work. The trickiest part is ensuring the array can resize itself when needed. In a second part, I will improve this structure by adding a "wrap-around" feature. Because my first implementation will already be rather complete, this won't need much changes.

1 First implementation of queues using arrays

Data structure and rules




```c
typedef struct queue {
    int *arr;
    int first; //initially 0
    int last;  //initially -1
    int nbOfElements; //initially 0
    int nbOfSlots; //whatever we want:
                   //(will dynamically resize)
} queue;
```

`first` is the index of the first used slot within the array. It can be 0 or something lower or equal to the number of slots in the array. `last` is the index of the last used slots within the array so `last+1` is the index of the first available slot that we can use when we want to enqueue an element. When `last+1=nbOfSlots` we risk an overflow and we have to double `nbOfSlots`.

Note that we could have done without the variable `nbOfSlots` since it could be deduced from `*arr` but it comes in handy for `full()` and will even more so later.

Basic procedures

```
int empty(queue *q) {return q->nbOfElements == 0;}
int full(queue *q) {return q->nbOfElements == q->nbOfSlots;}
```

These procedures are pretty straightforward. I am using the structure's built-in trackers to check if the queue is empty or full.

Resizing the array

The difficulty of this data structure is that it is not dynamic so I must create a new, bigger array once it is full and copy the values. I decide to double its size. But I mustn't forget to free the old *array* (and not the old queue) to avoid memory leaks. I also think we could benefit from resizing the queue when it is under-used: I shrink it by half whenever less than a third of the array is used (and the rest is empty).

```
void double_queue(queue *q) {
    int new_nbOfSlots=q->nbOfSlots*2;
    int *new_arr=(int *)malloc(..
        ..new_nbOfSlots*sizeof(int));

   for(inti=0;i<q->nbOfElements;i++){
        new_arr[i]=q->arr[q->first+i];
    }

    free(q->arr);
    q->arr=new_arr;
    q->first=0;
    q->last=q->nbOfElements-1;
    q->nbOfSlots=new_nbOfSlots;
}
```

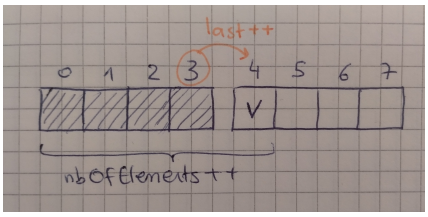
```
void half_queue(queue *q){
    if (q->nbOfSlots<=1){
        return;  //don't shrink if only one slot
    }

    if (q->nbOfElements<q->nbOfSlots/3){
        int new_nbOfSlots=q->nbOfSlots/2;
        if (new_nbOfSlots<1) new_nbOfSlots=1;

        int *new_arr=(int *)malloc(..
            ..new_nbOfSlots*sizeof(int));

        for (int i=0;i<q->nbOfElements;i++) {
            new_arr[i]=q->arr[q->first+i];
        }

        free(q->arr);
        q->arr=new_arr;
        q->nbOfSlots=new_nbOfSlots;

        q->first=0;
        q->last=q->nbOfElements-1;
    }
}
```

Enqueueing and dequeueing

For dequeueing, if the queue is empty, we send an error message. Then we get the oldest element from the queue [1] to return at the end. The index of the first element of the queue (the one we just got) is incremnented. Then we decrement the number of used slots by one and if less than 1/3 of slots are used, we shrink the array. At the end, if the queue is empty, we reset it.

[1] `arr[q->first]`

```c
void enqueue(queue *q, intv) {
    if (full(q)){double_queue(q);}
    q->last++;
    q->arr[q->last] = v;
    q->nbOfElements++;
}
```




```c
int dequeue(queue *q){
    if (empty(q)){
        printf("Queue is empty\n");
        return -1;
    }
    int value=q->arr[q->first];
    q->first++;
    q->nbOfElements--;
    half_queue(q); //shrink if needed
    if (q->nbOfElements==0)* {
        q->first=0;
        q->last=-1;
    }
    return value;
}
```

2 Testing the first implementation

Functionality tests

```
Initial queue:
  Queue elements: [10 20 30 40 _ ]
After 1 enqueueing (and double size):
  Queue elements: [10 20 30 40 50 _ _ _ _ ]
After 2 dequeueing:
  Queue elements: [30 40 50 _ _ _ _ ]
After 2 enqueueings:
  Queue elements: [30 40 50 60 70 _ _ ]
After 3 dequeueings (and shrinking):
  Queue elements: [60 70 _ _ ]
```

Growth rate after one hour

Assuming we perform 1000 operations per second in a balanced way [2]: we have 3,600,000 operations per hour. In a balanced way, this makes 1,800,000 enqueueings, that means 1,800,000 elements in the queue after an hour or at least 21 `double_queue` calls [3]. **So the array has 2,097,152 elements after an hour.**

Similarly, after a year, we have 15,768,000,000 enqueueings that would require 34 `double_queue` calls for **17,179,869,184 elements after a year.**

3 Wrapping around to make use of free space

Now, we can try and optimize this structure by making use of the empty space right before `q->first` in the array and seeing the indices as modulo `nbOfSlots`. The queue is no longer bound at its index 0.

[2] This implies that we never shrink the array because half of the array is always used: the condition that less than a third of the slots are used is never true

[3] $2^{21} = 2,097,152$

The data structure won't change, since we have already taken into account `nbOfSlots` in our first implementation. However, when we have to reset or create a new queue, the index showing the last used slot should be 0 and not -1 like before because now, we are not bound by 0.

The circular nature of the array also comes in whenever[4] we need to go through the elements of the queue: `q->arr[X+i]` becomes `q->arr[(X+i)%q->nbOfSlots]` because we go around the boundary of the array (where `X` can be `q->first` or `q->last`).

4 Testing the wrap-around feature

```
1  Initial queue:
2          Memory block: [_ _ _ _ ]
3          Queue is empty
4  After 3 enqueues:
5          Memory block: [10 20 30 _ ]
6          Queue elements: [10 20 30 ]
7  After 2 enqueue:
8          Memory block: [10 20 30 40 50 _ _ _ ]
9          Queue elements: [10 20 30 40 50 ]
10 After 2 dequeues:
11         Memory block: [10 20 30 40 50 _ _ _ ]
12         Queue elements: [30 40 50 ]
13 After 4 enqueues:
14         Memory block: [90 20 30 40 50 60 70 80 ]
15         Queue elements: [30 40 50 60 70 80 90 ]
16 After 5 dequeues:
17         Memory block: [90 20 30 40 50 60 70 80 ]
18         Queue elements: [80 90 ]
19 After 2 enqueues:
20         Memory block: [90 100 110 40 50 60 70 80 ]
21         Queue elements: [80 90 100 110 ]
```

This time, I need to print both the elements in the queue [5] and the memory blocks [6] to see clearly what is going on.

Note how, on line 10, 10 and 20 are still present in memory and aren't accessed by the queue but by the array. If needed, to avoid this memory leak we could scratch the values before discarding them when dequeueing (ie replace it by a random integer and than proceed as before).

Conclusion

During this assignment, is was fun stretching the limits of a simple array. The array approach for building queues gives fast access and simple memory management, but can be inefficient when dequeueing, with unused slots at the beginning of the array.

[4] for displaying the queue, enqueueing, dequeueing or resizing

[5] `q->arr[(q->first+i)%q->nbOfSlots]` from 0 to `nbOfElements`

[6] `q->arr[i]` from 0 to `nbOfSlots`