Queues

A queue is a First-In First-Out (FIFO) structure. You probably know queues by their more common alias, "lines." As you might imagine, because most people want to deal with request in a "fair" first come, first served manner, a queue is the type of structure that has many practical uses.

One of the most common uses of a queue would be to simulate a situation where you have a line with customers waiting to be served. When a customer arrives, he/she goes to the back of the line. We'll call this function, enqueue. When a customer is serviced, we'll say that they get dequeued. Further more, we may also want to check whether a queue is empty or full, and it may make sense to have a function that checks to see what request is at the front of the queue without actually dequeing it.

Thus, we have the following set of functions to support for a queue (for a C implementation of a queue of integers.)

```
queue* makequeue();
void enqueue(queue* myQueue, int element);
int dequeue(queue* myQueue);
int front(queue* myQueue);
int empty(queue* queue);
int full(queue* queue);
```

Let's consider tracing through a set of instructions for a couple queues:

```
queue* p = makequeue();
queue* q = makequeue();
enqueue(p, 5);
enqueue(q, 3);
enqueue(p, 8);
enqueue(p, dequeue(q));
enqueue(q, 2);
enqueue(q, 7);
val1 = dequeue(p);
val2 = front(q);
val3 = dequeue(p);
dequeue(q);
```

Using pictures, hopefully this trace is fairly simple to follow. But, now, we'd like to think about some implementation details.

What variables do we need to store a queue?
Will it work the same as a stack?

Consider trying a queue with an array and a back, to keep track of where to add the next element.

How would this queue work?

In particular, how would you dequeue an element?

The difficulty is that one needs to know the front of the queue. If we don't have that as a part of the struct, then we have to FIX the front of the queue to 0.

Why is this inefficient?

When we dequeue, we would have to shift ALL the elements up one spot in the array after a dequeue. This is quite a tedious process that takes O(n) time for a queue with n elements.

So now we come up with the idea of moving both the front and the back of the queue. Consider tracing through the following steps for an array of size 5:

```
queue* p = makequeue();
enqueue(p, 3);
enqueue(p, 2);
enqueue(p, 4);
int tmp = dequeue(p);
tmp = dequeue(p);
enqueue(p, 5);
tmp = dequeue(p);
enqueue(p, 3);
tmp = dequeue();
tmp = dequeue();
enqueue(p, 9);
```

Judging from this trace, we'll have to account for this "wrap-around" issue that was absent in our stack implementation.

So now, we come up with a revamped idea to store our queue structure:

1) An array
2) An integer that represents the front of the queue.
3) An integer that stores the current number of elements in the queue.

When we create a new queue, we'll allocate space for the array and set front, back, and size to 0.

To enqueue, we'll simply add the given element to the index back in the array. BUT, we aren't storing the index to the back of the array!!! What must we do instead?

If we know front, and we know size, adding these will give us the proper index into the back of the array. BUT wait, there's another problem, what if the front of the array is somewhere and when we add size to it, that becomes an out of bounds array index? This situation corresponds to our wrap around case. So how can we calculate the proper index?

Add the two and THEN mod by the ARRAY_SIZE:

(front+size)%ARRAY_SIZE is the proper index into the array.

Afterwards, we must just increment size by 1.

Now, consider dequeuing. Here we do the following:

1) Save the element at index front in the array.
2) Increment front. (But we must ALSO account for wrap-around here.)
3) Return the dequeued element.

Hopefully it should be clear how to check if the queue is empty or full and how we can return the element stored at the front of the queue.

Though I haven't shown you the implementation, these are the key ideas necessary to create one.

In particular, but biggest concept is dealing with the wrap-around issue.

Using a Dynamically Allocated Array to Store a Queue

We've analyzed most of the issues that will come up in this improvement upon the queue implementation when we discussed the same idea for a stack.

We will still do the following:

1) Allocate a new array, larger than the first (twice the size).
2) Copy the elements from the old array into the new.
3) Deallocate the space for the old array.
4) Make the old array point to the new one.

All of these four steps sound like one step: realloc. But there's a problem because of how realloc automatically does step #2 and what we really want to do for step #2.

In this step, we can no longer, simply loop through the elements one by one and copy them into the corresponding array element in the new array. Why is this problematic? (This essentially means that we can't do a realloc to carry out all four steps at once!)

The problem deals with the wrap-around. Consider the following situation:

6	**3**	**4**	**5**

front

size = 4
Enqueue(12);

Now consider what happens when we copy into the new array:

6	**3**	**4**	**5**				

Question: Where do front and back go?

Is this now possible??? Where should 6 really be in this array?

The problem becomes that the indexes for the wraparound are only accurate for one array size, not for other ones.

So, what we really need to do, is reset front to 0, and copy the elements into the array accordingly:

4	**5**	**6**	**3**	**12**			

front
size = 5

Now, let's see how we'd do this in code:

```
for (i=front, j=0; i<ARRAY_SIZE; i++, j++)
   temp[j] = values[i];
for (i=0; i<front; i++, j++)
   temp[j] = values[i];
```

Although this can be done with one array index, it's probably easier to think about what's going on using two array indexes. Here, i is used to index into the values array, and j is used to index into the temp array.

Linked List Implementation of a Queue

Let's consider using the linked list code developed before for this instance. Can we simply use one linked list?

We can, but what would be the drawback of doing so?

One of the problems would be that one of the two operations: either enqueue or dequeue would take O(n) time instead of O(1), which we achieved with an array implementation.

Why is this the case?

Because we have to access BOTH ends of the linked list for these two operations, no matter how we implement them.

So, if we were to use a linked list to represent a queue, we would have to use to pointers, one to the front of the list and one to the back.

One way to think about this is that our struct to store the queue would actually store to pointers to linked list structs. The first would point to the head of the list and the second would always point to the last node in that list. Here is a potential struct for a queue using a linked list:

```
typedef struct queue {
    node* front;
    node* back;
} queue;
```

and node would be a usual linked list node with a field to store data and a field point to the next node in the list.

Let's consider how we'd carry out some operations:

enqueue
1) Create a new node and store the inserted value into it.
2) Link the back node's next pointer to this new node.
3) Move the back node to point to the newly added node.

dequeue
1) Store a temporary pointer to the beginning of the list
2) Move the front pointer to the next node in the list
3) Free the memory pointed to by the temporary pointer.

front
1) Directly access the data stored in the first node through the front pointer to the list.

empty
1) Check if both pointers (front, back) are null.

A very good exercise for you guys (after you finish with the assignment) would be to see if you can use these general guidelines to put together a linked list implementation of a queue.