The George Washington University

CSCI 3411 - Operating Systems - Fall 2014

Programming Exercise 2 - Queues

Due time: Thursday 9/11 at midnight.

1 Objective

This exercise expands on the goals of Programming Exercise 1 and is designed to explore various implementations of queues. You are to write this implementation from scratch but you are allowed to reuse code if possible from the previous exercise. In particular, use the same techniques to debug. Again, in order to allocate memory and deallocate memory, you must use `malloc` and `free` from `stdlib.h`

2 Queues

Recall that a queue is governed by a FIFO insertion/removal policy and is generally implemented as a linked list unless specifically noted.

A priority queue is an augmented queue where the insertion policy (to enqueue) is determined by a priority assigned to the value. A value with a higher priority must be inserted before any other value with a lower priority, a value with an equal priority is inserted after all other values of the same priority and a value with lowest priority is inserted at the end of the list.

N.B. Priority will be maintained as an integer type (unsigned). A "high" priority will be represented as a small integer value with 0 being the highest priority and a "low" priority will be represented as a large integer value. Therefore, in a priority queue with enqueued values ordered from q_1 to q_n:

$$Priority(q_1) \leq Priority(q_2) \leq \cdots \leq Priority(q_{n-1}) \leq Priority(q_n)$$

For example, if two values, v_1 and v_2, are inserted into a priority queue and v_1 has a priority of 5 and v_2 has a priority of 2, then $Priority(v_2) < Priority(v_1)$ and v_2 has a higher priority than v_1. Therefore, v_2 should be enqueued in front of v_1 regardless of the order in which they were enqueued.

2.1 O(n) Priority Queue

The complexity of a priority queue will be dominated by the efficiency of insertion. If the insertion operation must iterate over the list to search for an insertion location, then, in the worst-case, the entire list must be traversed and results in O(n) complexity. The worst-case can be optimized by simply maintaining the tail of the list and checking the tail's priority. However, the optimization fails to solve the O(n) complexity problem if the priority of an enqueuing value requires insertion at any position following the head and preceding the tail. A priority queue that uses a search insertion policy will be called an O(n) priority queue.

2.2 O(1) Priority Queue

In order to optimize the insertion policy, conceptualize the priority queue as a list of queues where each queue in the list handles a single priority. For optimal, direct access into individual queues, the list can be constructed from a fixed-sized array rather than from a linked list. A look-up for a given queue requires only a single operation by cross-referencing the priority with the array index. Because all priorities in a given queue are equal, enqueuing will always occur at the tail of a queue. Because there is no traversal of linked lists and because each operation occurs in constant time, the overall complexity is O(1). A priority queue that uses a direct access insertion policy will be called an O(1) priority queue.

2.3 Ring Buffer

In constrained memory environments, linked lists are problematic because they may grow to consume all available memory resources. A constrained memory resource should instead be based on a fixed-sized array. A fixed-sized array can be wrapped into a ring structure by applying the modulo operation when indexing. The ring structure facilitates implemention of a queue which will be called a ring buffer.

In a linked list, the head and tail are pointers; however, in the ring buffer, the head and tail are integer indexes representing addresses. Values are enqueued at a position relative to the tail and then the tail is advanced. Values are dequeued at the position indicated by the head and then the head is advanced. Very simple mathematical rules can then be evaluated to determine if the buffer is full or empty and all accesses are direct and efficient. If the head or tail were to advance beyond the end of the array, the modulo operation forces these indices to always remain valid references within the bounds of the ring buffer.

3 Assignment

3.1 O(n) Priority Queue

Implement an O(n) priority queue in a C file named `pqueueOn.h`. The lowest priority will be 9 and the highest priority will be 0.

N.B. Remember that the search is only necessary if the optimal cases fail. Make sure to take advantage of this. If you do not and you then follow the recommendations attached to the O(1) priority queue implementation, you will not have implemented an O(1) priority queue.

The implementation shall have the following defined values.

```
PQ_HIGHEST_PRIORITY = 0

PQ_LOWEST_PRIORITY = 9
```

The implementation shall have the following functions:

```
struct pqueueOn* pqOn_create(void)
```

```
void pqOn_delete(struct pqueueOn *pq)

int pqOn_enqueue(struct pqueueOn *pq, void *value, unsigned int priority)

void* pqOn_dequeue(struct pqueueOn *pq)
```

`pqOn_create` allocates and returns a pointer to an O(n) priority queue. If unable to allocate the priority queue, `pqOn_create` returns `NULL`.

`pqOn_delete` deallocates an O(n) priority queue.

`pqOn_enqueue` inserts a value into an O(n) priority queue under the insertion policy described above for an O(n) priority queue and returns -1 (logical true) on success. If `pqOn_enqueue` fails, it returns 0 (logical false).

`pqOn_dequeue` removes the value at the head of an O(n) priority queue and returns the value. If the priority queue is empty, `pqOn_dequeue` returns `NULL`

Example Given an initial state s_0 O(n) priority queue, *pq*, and the values, v_1, v_2, v_3, and v_4, where $Priority(v_1) = 1$, $Priority(v_2) = 2$, $Priority(v_3) = 1$ and $Priority(v_4) = 0$.

pq state					
s_0	s_1	s_2	s_3	s_4	s_5
	v_1	$v_1 \rightarrow v_2$	$v_1 \rightarrow v_3 \rightarrow v_2$	$v_3 \rightarrow v_2$	$v_4 \rightarrow v_3 \rightarrow v_2$

Table 1: O(n) Priority Queue Example

Enqueuing v_1 into *pq* yields a state s_1 *pq* containing only v_1 where $head \rightarrow v_1$ and $tail \rightarrow v_1$.

Enqueuing v_2 into *pq* yields a state s_2 *pq* containing $v_1 \rightarrow v_2$ where $head \rightarrow v_1$ and $tail \rightarrow v_2$ because $Priority(v_1) < Priority(v_2)$.

Enqueuing v_3 into *pq* yields a state s_3 *pq* containing $v_1 \rightarrow v_3 \rightarrow v_2$ where $head \rightarrow v_1$ and $tail \rightarrow v_2$ because $Priority(v_1) \leq Priority(v_3) < Priority(v_2)$. Note that v_3 has been inserted after v_1 but before v_2 because v_3 has the same priority as the already enqueued v_1 but v_3 has a higher priority than the already enqueued v_2.

Dequeuing from *pq* returns v_1 and yields a state s_4 *pq* containing $v_3 \rightarrow v_2$ where $head \rightarrow v_3$ and $tail \rightarrow v_2$.

Enqueuing v_4 into *pq* yields a state s_5 *pq* containing $v_4 \rightarrow v_3 \rightarrow v_2$ where $head \rightarrow v_4$ and $tail \rightarrow v_2$ because $Priority(v_4) < Priority(v_3) < Priority(v_2)$. Note that v_4 has been inserted at *head*.

3.2 O(1) Priority Queue

Implement an O(1) priority queue in a C file named `pqueue01.h`. The lowest priority will be 9 and the highest priority will be 0. Note that this implies that the array will have a length 10.

N.B. It is suggested that you implement the O(1) priority queue by simply making an array of O(n) priority queue pointers where a queue at index i in the array only stores priorities of i. Implement this right and your O(1) priority queue code will be very concise by relying on (ideally) bulletproof code.

The implementation shall have the following defined values:

```
PQ_HIGHEST_PRIORITY = 0

PQ_LOWEST_PRIORITY = 9
```

N.B. If you include `pqueueOn.h` and follow the attached note above for the O(1) priority queue then these definitions have already been defined and they should not appear in your `pqueueO1.h` file.

The implementation shall have the following functions:

```
struct pqueueO1* pqO1_create(void)

void pqO1_delete(struct pqueueO1 *pq)

int pqO1_enqueue(struct pqueueO1 *pq, void *value, unsigned int priority)

void* pqO1_dequeue(struct pqueueO1 *pq)
```

`pqO1_create` allocates and returns a pointer to an O(1) priority queue. If unable to allocate the priority queue, `pqO1_create` returns `NULL`.

`pqO1_delete` deallocates an O(1) priority queue.

`pqO1_enqueue` inserts a value into an O(1) priority queue under the insertion policy described above for an O(1) priority queue and returns -1 (logical true) on success. If `pqO1_enqueue` fails, it returns 0 (logical false).

`pqO1_dequeue` removes the value at the head of an O(1) priority queue and returns the value. If the priority queue is empty, `pqO1_dequeue` returns `NULL`

Example Given an initial state s_0 O(1) priority queue, pq, and the values, v_1, v_2, v_3, and v_4, where $Priority(v_1) = 1$, $Priority(v_2) = 2$, $Priority(v_3) = 1$ and $Priority(v_4) = 0$.

Enqueuing v_1 into pq yields a state s_1 pq containing only v_1 in the pq_1 subqueue.

Enqueuing v_2 into pq yields a state s_2 pq containing v_1 in the pq_1 subqueue and v_2 in the pq_2 subqueue.

Enqueuing v_3 into pq yields a state s_3 pq containing $v_1 \rightarrow v_3$ in the pq_1 subqueue and v_2 in the pq_2 subqueue.

Dequeuing from pq returns v_1 from the pq_1 subqueue and yields a state s_4 pq containing v_3 in the pq_1

	pq state						
$pq_{priority}$	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0						v_4	
1		v_1	v_1	$v_1 \rightarrow v_3$	v_3	v_3	v_3
2			v_2	v_2	v_2	v_2	v_2

Table 2: O(1) Priority Queue Example

subqueue and v_2 in the pq_2 subqueue.

Enqueuing v_4 into pq yields a state s_5 pq containing v_4 in the pq_0 subqueue, v_3 in the pq_1 subqueue, and v_2 in the pq_2 subqueue.

Dequeuing from pq returns v_4 from the pq_0 subqueue and yields a state s_6 pq containing v_3 in the pq_1 subqueue and v_2 in the pq_2 subqueue.

3.3 Ring Buffer

Implement a ring buffer in a C file named `ring.h`. The ring buffer will have a length of 16, the *head* will be initialized to the 0 index and the *tail* will be initialized to the $head + 1$ index. If the *tail* equals the *head*, the ring buffer is full, and if the *tail* equals the $head + 1$, the ring buffer is empty. Note that these rules imply that the maximum capacity of the ring buffer is actually $length - 1$.:

The implementation shall have the following defined value:

```
BUFFER_LENGTH = 16
```

The implementation shall have the following functions:

```
struct ring* rb_create(void)

void rb_delete(struct ring *rb)

int rb_isempty(struct ring *rb)

int rb_isfull(struct ring *rb)

int rb_enqueue(struct ring *rb, void *value)

void* rb_dequeue(struct ring *rb)
```

`rb_create` allocates and returns a pointer to a ring buffer. If unable to allocate the ring buffer, `rb_create` returns `NULL`.

`rb_delete` deallocates a ring buffer.

`rb_isempty` returns 0 if the ring buffer is non-empty (logical false) and -1 if the ring buffer is empty (logical true).

`rb_isfull` returns 0 if the ring buffer is non-full (logical false) and -1 if the ring buffer is full (logical true).

`rb_enqueue` checks to see if the ring buffer is full. If the ring buffer is non-full, `rb_enqueue` inserts a value at the $tail - 1$ of the ring buffer, advances the *tail*, and returns -1 (logical true). If the ring buffer is full or `rb_enqueue` fails for any reason, `rb_enqueue` returns 0 (logical false).

`rb_dequeue` removes the value at the *head* of the ring buffer, advances the *head*, and returns the value. If the ring buffer is empty, `rb_dequeue` returns `NULL`

Example Given an initial state s_0 ring buffer, *rb*, of length 4 and the values, v_1, v_2, v_3, and v_4.

state	*head*	*tail*	*rb*
s_0	0	1	
s_1	0	2	v_1
s_2	0	3	$v_1 \rightarrow v_2$
s_3	0	0	$v_1 \rightarrow v_2 \rightarrow v_3$
s_4	1	0	$v_2 \rightarrow v_3$
s_5	1	1	$v_2 \rightarrow v_3 \rightarrow v_4$

Table 3: Ring Buffer Example

Enqueuing v_1 into *rb* yields a state s_1 *rb* containing only v_1 with the *head* remaining at 0 and the *tail* advanced to 2.

Enqueuing v_2 into *rb* yields a state s_2 *rb* containing v_1 followed by v_2 with the *head* remaining at 0 and the *tail* advanced to 3.

Enqueuing v_3 into *rb* yields a state s_3 *rb* containing v_1 followed by v_2 followed by v_3 with the *head* remaining at 0 and the *tail* advanced to 0. Note the tail is advanced to 0 rather than 4 as it is constrained by the modulo operation and the ring buffer is now full (Remember the buffer capacity is $length - 1$ by definition of this document). Attempting to enqueue again into the full ring buffer will fail until a value has been dequeued.

Dequeuing from *rb* returns v_1 and yields a state s_4 *rb* containing v_2 followed by v_3 with the *head* advanced to 1 and the *tail* remaining at 0.

Enqueuing v_3 into *rb* yields a state s_5 *rb* containing v_2 followed by v_3 followed by v_4 with the *head* remaining at 1 and the *tail* advanced to 1. Note that the ring buffer is again full by definition.

4 Testing

You will need to test your `pqueueOn.h`, `pqueueO1.h` and `ring.h` files. Do this by creating separate files containing a `main` function, such as `pqueueOn_test.c`, `pqueueO1_test.c` and `ring_test.c`. You will need to test each of the functions that you have defined in varying ways. Apply the same considerations and lessons from Programming Exercise 1 when testing these implementations. Your implementations will be evaluated against randomized data.

5 Submission

Create a block comment header in each of your `pqueueOn.h`, `pqueueO1.h` and `ring.h` files and include your name and email address in the header. You will create a zip archive containing only your `pqueueOn.h`, `pqueueO1.h` and `ring.h` files and you will submit this zip file to Blackboard in the folder for Programming Exercise 2 by the appointed time indicated at the top of this document. The zip file will be named with the following naming convention:

`csci_3411_fa14_ex_02_<your GWNet Id>.zip`

Where `<your GWNet Id>` is the user name that you use to log into Blackboard.

6 Evaluation

You are obligated to complete this exercise on your own by the Code of Academic Integrity (`http://www.gwu.edu/~ntegrity/code.html`).

Your implementations will be compiled against standard test programs that will use randomized data. If your `pqueueOn.h`, `pqueueO1.h` and `ring.h` files do not match the declarations defined above, your implementation will not compile. If your program does not compile, your evaluation will be unsatisfactory.

The conciseness and simplicity of your code will affect your evaluation. There is a direct relationship between the probability of introducing bugs and both the number of lines of code implemented and the complexity of individual lines of code. Find a balance between the number of lines of code and the simplicity of each line.

Your code will be evaluated in terms of the comprehensiveness and descriptiveness of comments. Comments are critical to conveying information about your implementation. A block comment preceding your function definition should describe the function including any special information about the input parameters and output result. Line commments should clarify variable usage and complex logic. Do not assume that either you or others will understand code that you have written. The time it takes for anyone to understand code is non-productive.