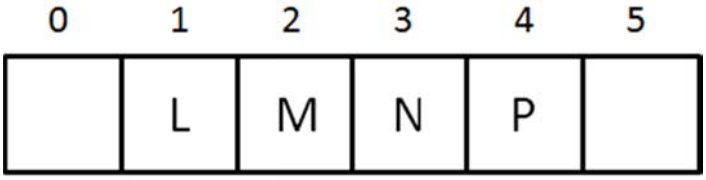
SCC120 Week 8 workshop answers

Queue Abstract Data Type

1) The answer is (a). The elements will still be in the same order.

2) The result is below. The "front" variable is still 1. The "back" variable is now 4.



3) Answer is (c). Removing all elements requires going through all the elements, so if there's N of them, it takes O(N) or linear time. This is assuming that we actually go through each element and remove the memory to store them.

If we do not remove the memory of the "removed" elements, we can just make the front and back pointers both point to "null". In this case, this takes O(1) time. If this is the case, none of the four answers would be correct. So the question itself could have been more clear here (even though we intended that the answer should be c).

4) Answer is (b). For a circular buffer that is full, "front" comes immediately after "back". So (back+1) will be equal to "front". If (back+1) is smaller than MAX_SIZE, then ((back+1)%MAX_SIZE) is equal to (back+1).

The only other case we need to consider is when "front" is 0 and "back" is the last element of the array. So (back+1) is MAX_SIZE, and ((back+1)%MAX_SIZE) is 0. And so front==((back+1)%MAX_SIZE) is also true.

5) This is the implementation of *peek()* with a linked list:

```
if (front == NULL) {
      PROBLEM: output some message to say queue is empty
} else {
      return front->value;
}
```

The complexity is O(1).

6) Complexity for remove() is O(1). We remove the first time from the front as usual. Since we have a front pointer, it only involves operations at the top of the linked list.

The add() method is O(n). We need to scan to the end of the linked list, before we can add one item at the end.

7) For add(X):

```
temp = new QueueElement(X);
temp->next = back->next;
back->next = temp;
back = temp;
```

Complexity is O(1).

For remove():

```
temp = back->next->next;
back->next = temp;
```

This is also O(1). You can optionally delete the memory for the removed item.

8) If we assume that when we add an element to the priority queue, we immediately place it at the "right" position such that the whole priority queue is ordered according to the priority values, then add() would cost O(n) time, as one has to go through all n elements in the worst case to place the new one at the "right" position. For remove(), it is just removing the element at the top with the highest priority, so would be O(1) time.

9) For O(1): take the last element, give it a priority value that is higher than the highest one, and put it at the front of the priority queue. For O(n): take the last element, increase its priority value according to its timestamp in some way, and re-position it to the correct order in the priority queue (this part requires scanning through potentially all elements in some way to re-position it in the right location).