Array-Based Queue Push Operation

Assume that we have the following lines of code:

```
myqueue queue1;      // Line 1

queue1.push(5);      // Line 2
queue1.push(8);      // Line 3
queue1.push(3);      // Line 4
queue1.push(6);      // Line 5
```

The following sequence of diagrams shows how the `myqueue` object and its associated dynamic storage changes as these lines are executed.

Figure 1: The new, empty `myqueue` object `queue1` created in Line 1 of the code above. The `q_array` pointer is `nullptr`, while `q_size` and `q_capacity` are both 0. `q_front` is 0 and `q_back` is equal to `(q_capacity - 1)` or -1.

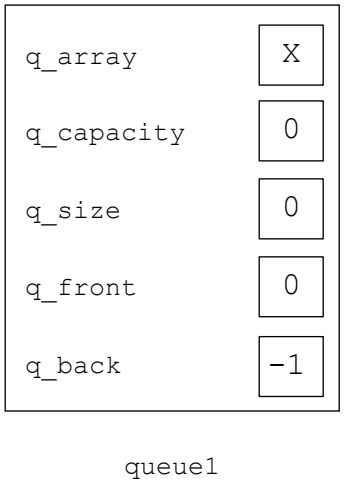
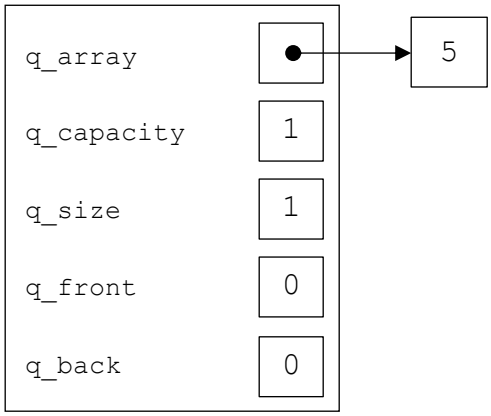
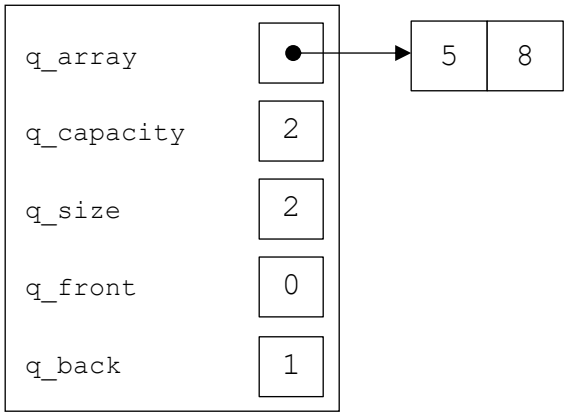


Figure 2: The `myqueue` object following the execution of Line 2. Since `q_size == q_capacity`, the `push()` method will call the `reserve()` method to allocate a new dynamic array. Since the current `q_capacity` is 0, the capacity requested for the new array will be 1. The contents of the existing array (if any) are copied to the new array (in this case, there's nothing to copy). The `q_capacity` is updated to the capacity of the new array. The existing array is then deleted (in this case, there's nothing to delete) and the `q_array` pointer is set to point to the new array. `q_back` is incremented by 1 (from -1 to 0), then divided by the `q_capacity` of 1, and the remainder of 0 is assigned to `q_back`. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 0) and then the `q_size` is incremented to 1.



queue1

Figure 3: The `myqueue` object following the execution of Line 3. Since `q_size == q_capacity`, the `push()` method will call the `reserve()` method to allocate a new dynamic array. Since the current `q_capacity` is not 0, the capacity requested for the new array will be 2 (two times the current capacity of 1). The contents of the existing array are copied to the new array. The `q_capacity` is updated to the capacity of the new array. The existing array is then deleted and the `q_array` pointer is set to point to the new array. `q_back` is incremented by 1 (from 0 to 1), then divided by the `q_capacity` of 2, and the remainder of 1 is assigned to `q_back`. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 1) and then the `q_size` is incremented to 2.



queue1

Figure 4: The `myqueue` object following the execution of Line 4. Since `q_size == q_capacity`, the `push()` method will call the `reserve()` method to allocate a new dynamic array. Since the current `q_capacity` is not 0, the capacity requested for the new array will be 4 (two times the current capacity of 2). The contents of the existing array are copied to the new array. The `q_capacity` is updated to the capacity of the new array. The existing array is then deleted and the `q_array` pointer is set to point to the new array. `q_back` is incremented by 1 (from 1 to 2), then divided by the `q_capacity` of 4, and the remainder of 2 is assigned to `q_back`. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 2) and then the `q_size` is incremented to 3.

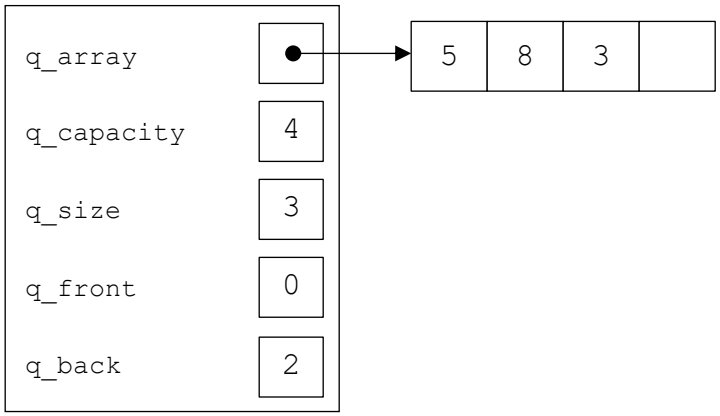
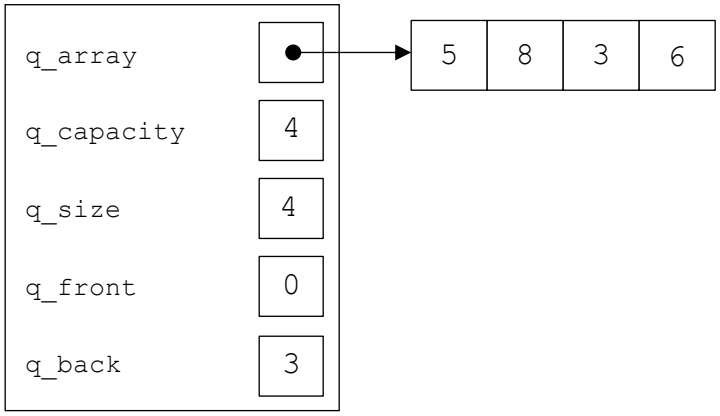


Figure 5: The `myqueue` object following the execution of Line 5. Since `q_size != q_capacity`, the `push()` method does not call the `reserve()` method. `q_back` is incremented by 1 (from 2 to 3), then divided by the `q_capacity` of 4, and the remainder of 3 is assigned to `q_back`. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 3) and then the `q_size` is incremented to 4.



Array-Based Queue Pop Operation

Assume that we then add the following lines of code after the code listed above:

```
queue1.pop();        // Line 6
queue1.pop();        // Line 7
```

The following sequence of diagrams shows how the `myqueue` object and its associated dynamic storage changes as these lines are executed.

Figure 6: The `myqueue` object following the execution of Line 6. `q_front` is incremented by 1 (from 0 to 1), then divided by the `q_capacity` of 4, and the remainder of 1 is assigned to `q_front`. The `q_size` is decremented to 3. That means that element 1 (the value 8) is now the front item in the queue, and element 0 (the value 5) is now outside the boundaries of the queue. Effectively, it has been removed from the queue even though the value is technically still present in the array.

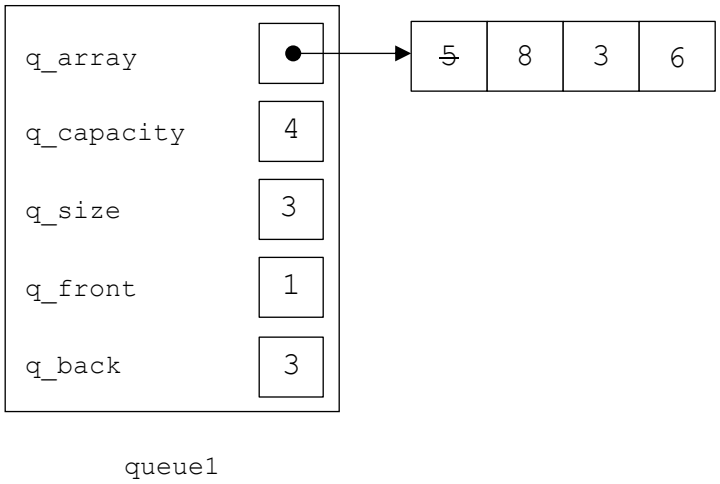
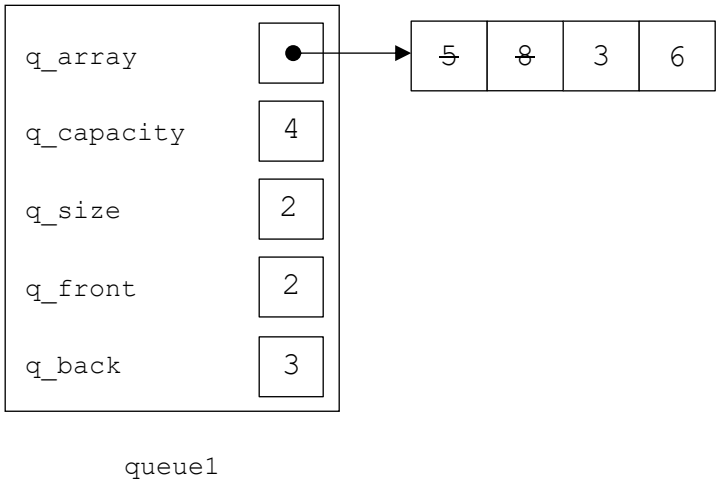


Figure 7: The `myqueue` object following the execution of Line 7. `q_front` is incremented by 1 (from 1 to 2), then divided by the `q_capacity` of 4, and the remainder of 2 is assigned to `q_front`. The `q_size` is decremented to 2. That means that element 2 (the value 3) is now the front item in the queue, and element 1 (the value 8) is now outside the boundaries of the queue.



Note that the `pop()` method (or at least the version outlined in the notes) does **not** change the stack capacity.

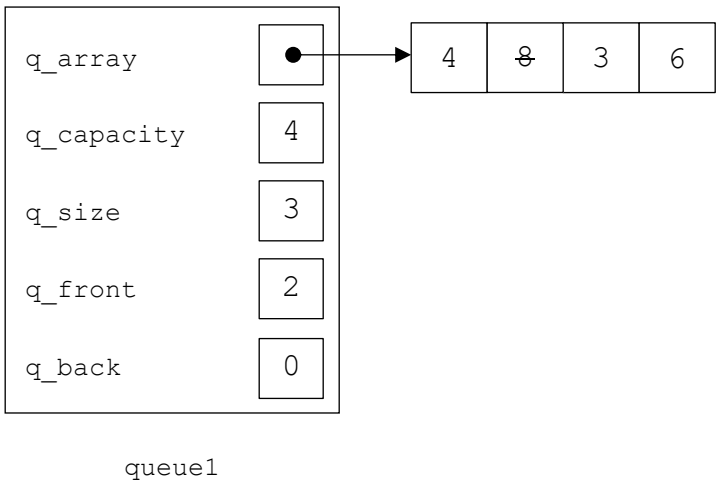
Wraparound on Array-Based Queue Push Operation

Assume that we then add the following lines of code after the code listed above:

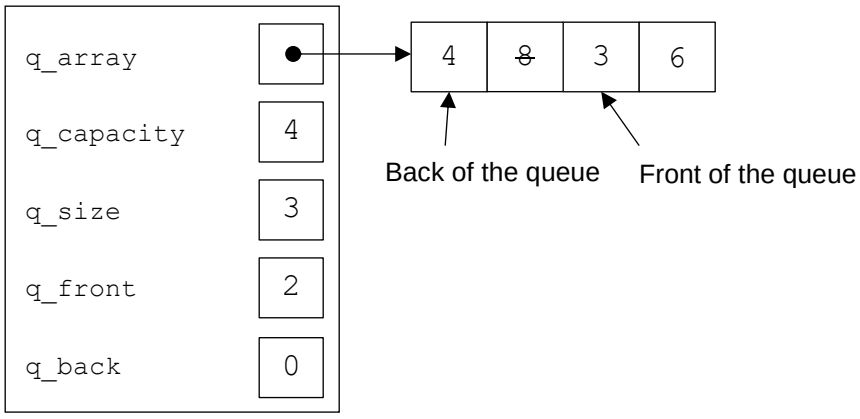
```
queue1.push(4);        // Line 8
queue1.push(7);        // Line 9
```

The following sequence of diagrams shows how the `myqueue` object and its associated dynamic storage changes as these lines are executed.

Figure 8: The `myqueue` object following the execution of Line 8. Since `q_size != q_capacity`, the `push()` method does not call the `reserve()` method. `q_back` is incremented by 1 (from 3 to 4), then divided by the `q_capacity` of 4, and the remainder of 0 is assigned to `q_back`. Thus, `q_back` has wrapped around to the beginning of the array. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 0) and then the `q_size` is incremented to 3.



The location in the array of the front item of the queue is now **after** the location in the array of the back item of the queue!



queue1

It may help to visualize the queue array as a circle. Starting from the front element of the queue, we proceed counter-clockwise to the back element of the queue. The *locations* of the front and back elements of the queue within the array are relatively unimportant as long as the *order* of the elements in the queue is maintained.

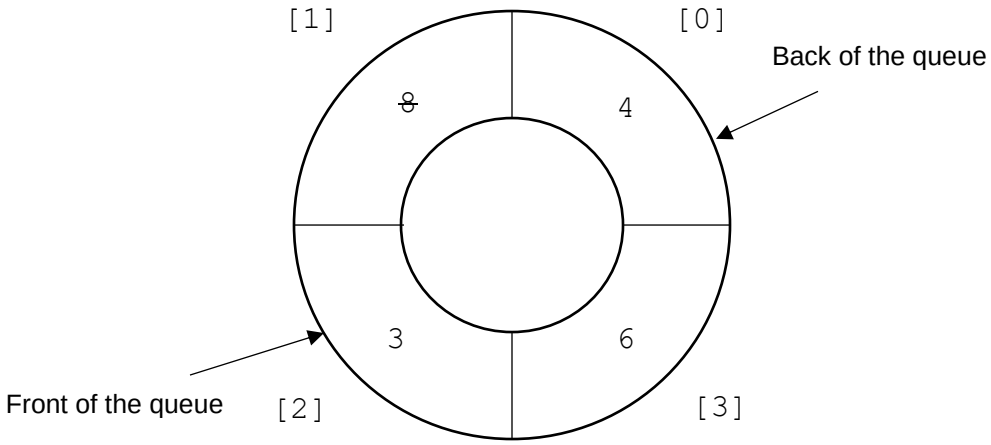
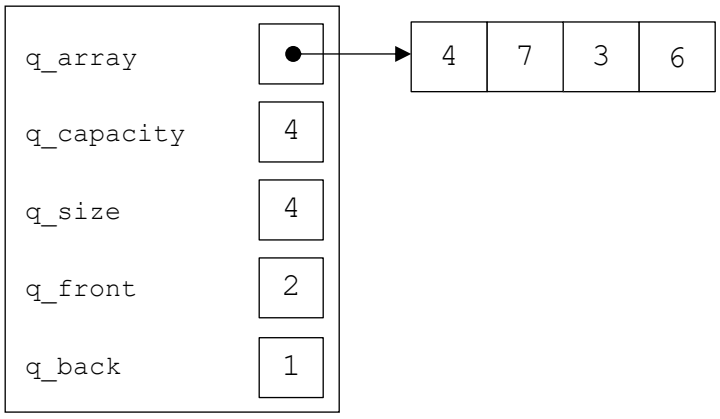
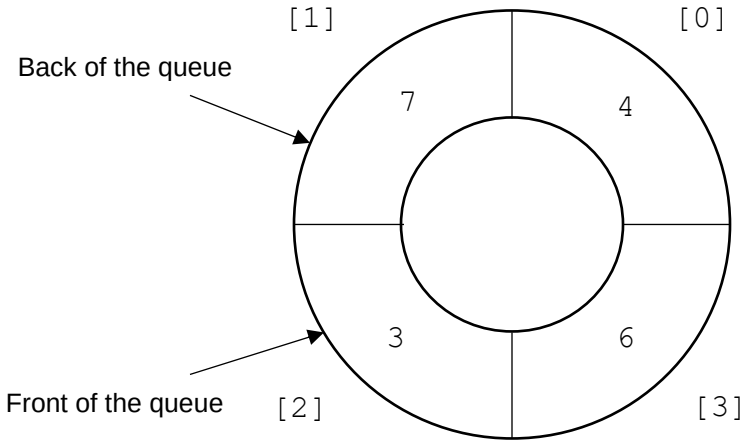


Figure 9: The `myqueue` object following the execution of Line 8. Since `q_size != q_capacity`, the `push()` method does not call the `reserve()` method. `q_back` is incremented by 1 (from 0 to 1), then divided by the `q_capacity` of 4, and the remainder of 1 is assigned to `q_back`. Finally, the value to insert is stored in the array at subscript `q_back` (subscript 1) and then the `q_size` is incremented to 4.



queue1

Visualized as a circle, the queue array now looks like this:



Wraparound on Array-Based Queue Pop Operation

We get the same kind of wraparound with the array-based queue's `pop()` operation.

Assume that we then add the following lines of code after the code listed above:

```
queue1.pop();        // Line 10
queue1.pop();        // Line 11
```

The following sequence of diagrams shows how the `myqueue` object and its associated dynamic storage changes as these lines are executed.

Figure 10: The `myqueue` object following the execution of Line 10. `q_front` is incremented by 1 (from 2 to 3), then divided by the `q_capacity` of 4, and the remainder of 3 is assigned to `q_front`. The `q_size` is decremented to 3. . That means that element 3 (the value 6) is now the front item in the queue, and element 2 (the value 3) is now outside the boundaries of the queue.

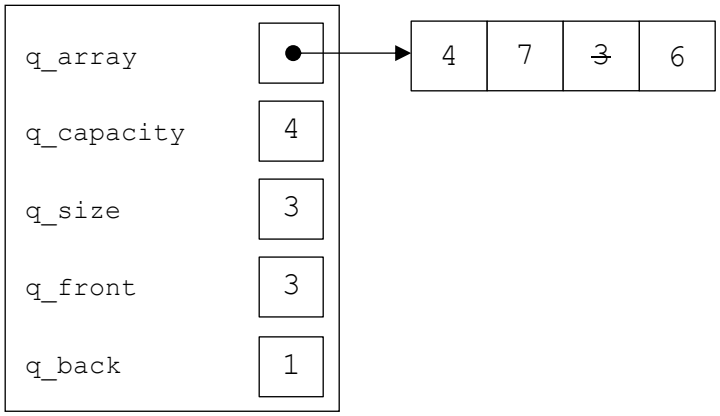
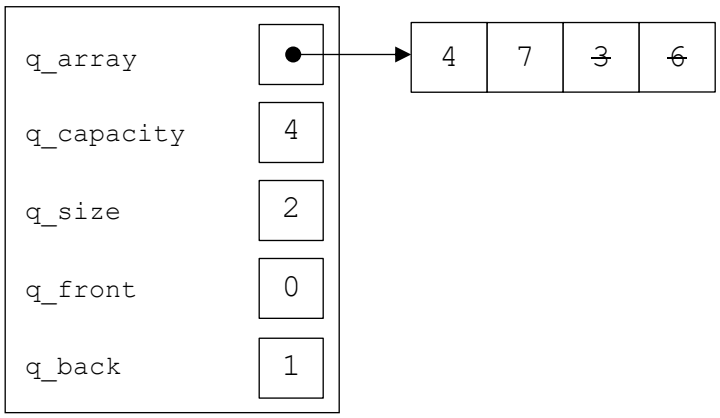
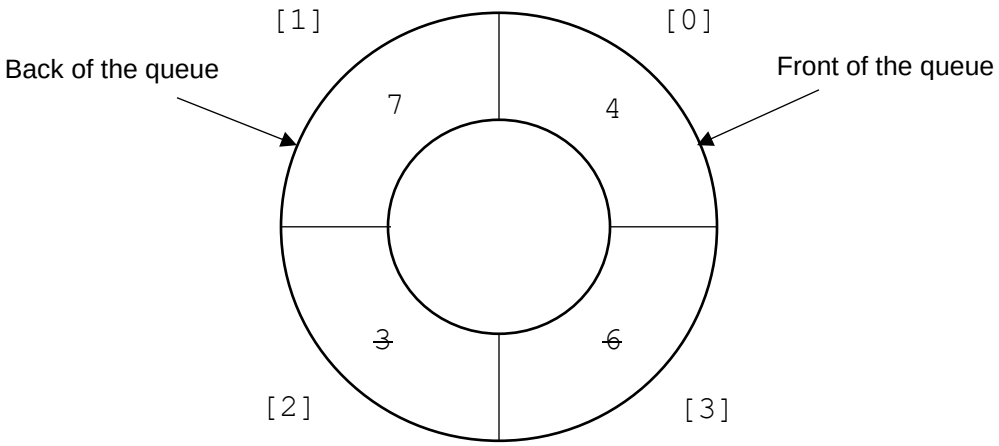


Figure 11: The `myqueue` object following the execution of Line 11. `q_front` is incremented by 1 (from 3 to 4), then divided by the `q_capacity` of 4, and the remainder of 0 is assigned to `q_front`. Thus, `q_front` has wrapped around to the beginning of the array. The `q_size` is decremented to 2. That means that element 0 (the value 4) is now the front item in the queue, and element 3 (the value 6) is now outside the boundaries of the queue.



Visualized as a circle, the queue array now looks like this:



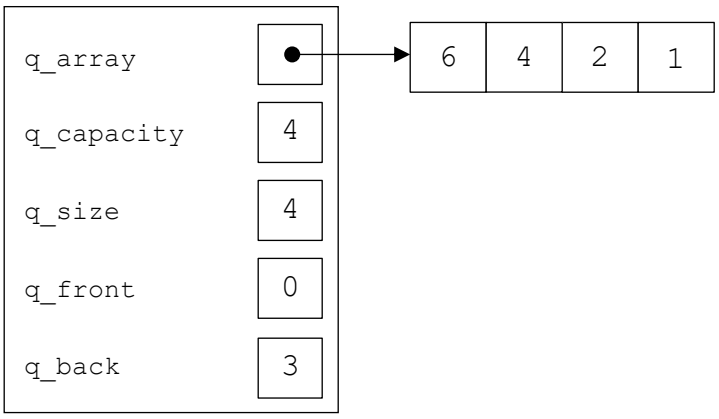
The `reserve()` Method

Coding a `reserve()` method for the array-based queue is complicated by the fact that the circular nature of the queue array means that at any given time `q_front` may be less than `q_back`, equal to `q_back`, or greater than `q_back`.

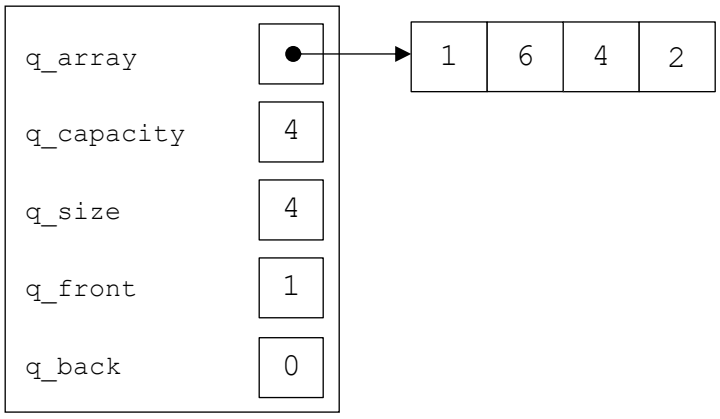
In the array-based stack, the stack elements always occupied array elements 0 to `stkSize - 1`. There is no guarantee that is the case with a circular array-based queue.

For example, assume that the queue has a capacity of four and currently contains four items. The items in the queue are stored in descending order; the front item in the queue has the value 6, while the back item in the queue has the value 1.

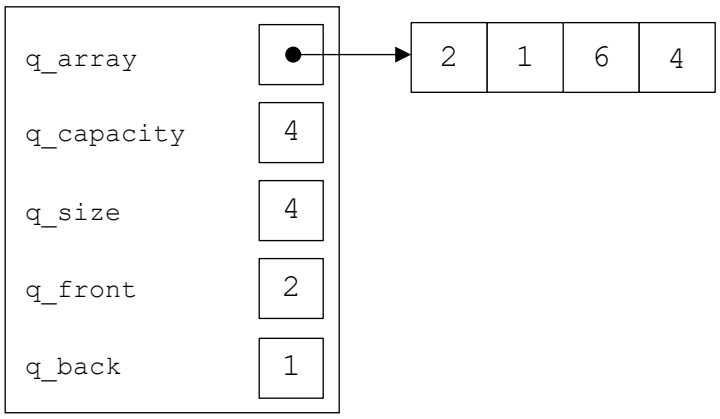
All four of the following diagrams represent valid arrangements for this queue:



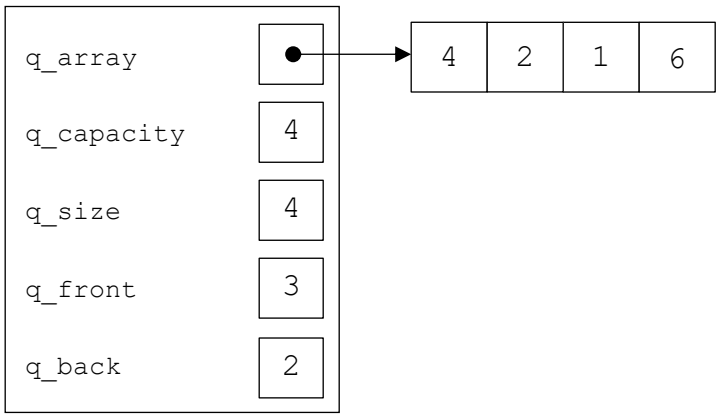
queue1



queue1



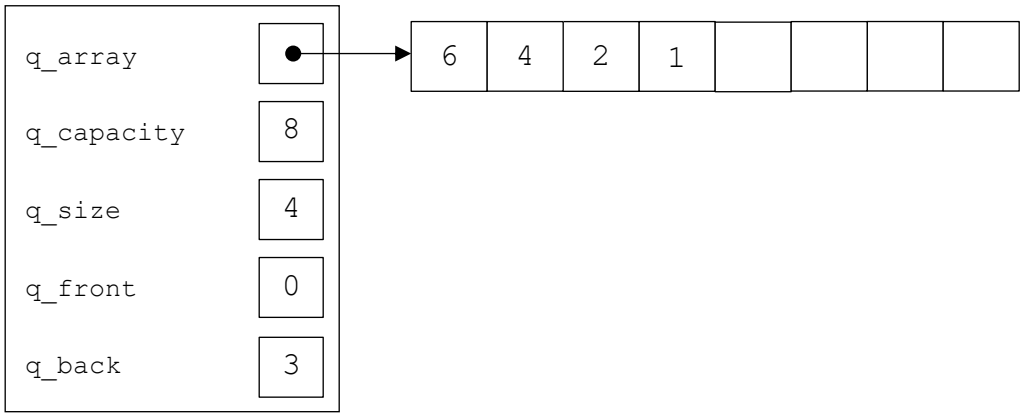
queue1



queue1

We can simplify the solution to this problem dramatically if we recognize that there is no need for the queue items to occupy the same elements in the new, larger array as they occupied in the original array. The `q_front` and `q_back` subscripts can be altered freely as long as the *order* of the elements in the queue remains the same.

No matter what the original arrangement of the queue items was in the original array, we can place the front item at subscript 0 in the new larger array, the next item at subscript 1, the next item at subscript 2, etc. In this arrangement, the back item will always be located at the subscript `(q_size - 1)`.



queue1

This can be accomplished using a loop like the following:

```
int current = q_front;
for (size_t i = 0; i < q_size; i++)
{
    temp_array[i] = q_array[current];
    current = (current + 1) % q_capacity;
}

q_front = 0;
q_back = q_size - 1;
```

The `q_capacity` for the queue should not be updated to the new, larger capacity until after this loop has completed.