COP 3502: Computer Science I
Spring 2004

– Note Set 14 –
Data Structures: Queues

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

A Modification to our Stack Data Structure

- If you studied the stack implementation that we covered in the last section of the notes, you will have discovered that our stack is array based and has a maximum size of 20 elements (see Note Set 13, page 33).
- What would happen if we needed to place more than 20 elements into the stack? Suppose that in the application to reverse a string, the string was more than 20 characters in length. Then our program would fail to properly reverse the string because the stack would overflow.
- However, it would be extremely likely that there was a sufficient amount of memory in the machine to have made the stack larger. We don't want to waste memory by making the static stack too large, but we also want it to be large enough to handle any expected input without causing the stack to overflow.
- The solution is to dynamically resize the stack if necessary.

A Modification of the Stack Data Structure (cont.)

- We have already seen how to dynamically create an array, so the technique for resizing our stack is easy to grasp. The basic technique is:
  - Given an array named `values` (our stack) that is already full and stores `length` number of elements, we want to add `newval` to the end of the array, we need to do the following:

1. Dynamically create a new array `temp`, that has more space than values.
2. Copy each element in values into `temp`, one by one.
3. Add the new element that didn't fit in the original array into a remaining open slot in `temp`.
4. Deallocate the memory for the original array.
5. Assign the pointer `values` to `temp`.

- An implementation of this technique is shown on the next page.

Dynamically Re-Sizing An Array

```
int *temp = (int *)malloc((length+1)*sizeof(int));
for (i=0; i<length; i++)  // copy original values
      temp[i] = values[i];
temp[i] = newval;  //add the new value to the array
delete [ ] values;
values = temp;  //reset pointer to new memory
```


Dynamic Resizing

- This implementation on the previous page for dynamically resizing the array will work, but do you notice any potential weakness of the code?
- What would happen if we had to add one more element into this array? We would resize the array again, copying all of the values into new memory.
- How much time would it take for each addition in terms of the total number of elements currently in the array? O(n)
- Clearly, this strategy of extending the array by one element each time is too costly. We need a better better strategy:
- Can you think of a better strategy that would reduce the average time required to insert an additional element above and beyond our original allocation?

Dynamic Resizing

- A better strategy turns out to be that whenever you expand the array you don't just add one element but rather you double its size.
- This strategy leads to excellent performance in the long run. Instead of requiring O(n) time to add an element, where n is the total number of elements, this strategy requires O(1) on average.
- In order not to waste huge amounts of memory, it is also advisable to "shrink" this type of array by half when the array is less than one quarter full. Once again, it can be shown that this procedure leads to efficient average case running times.
- These ideas can easily be incorporated into a stack class so that the size of the stack is not so limiting. Even so, a StackIsFull function is advisable since sometimes a malloc call may return NULL.

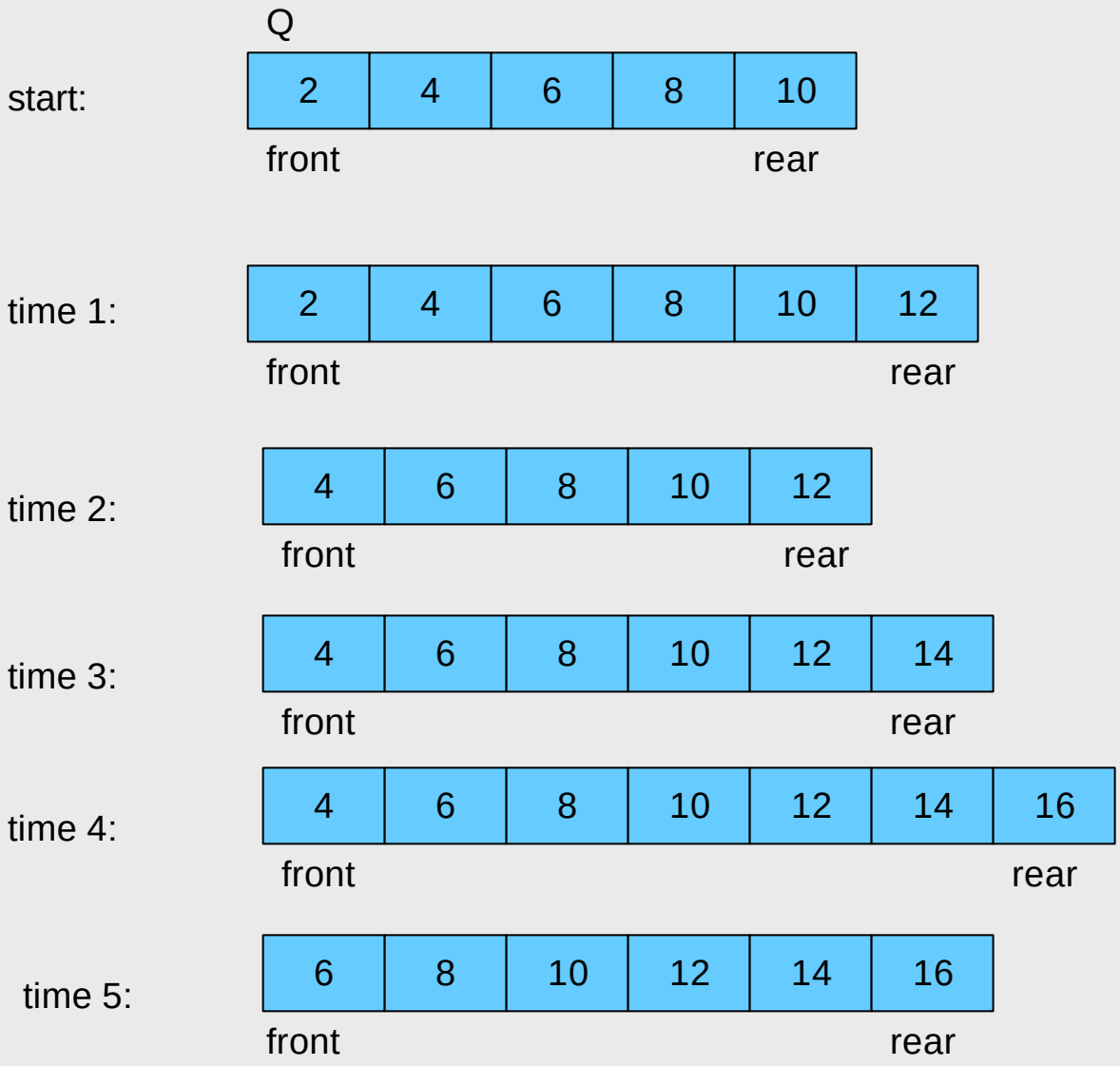
Queues

- A queue is simply a waiting line that grows by adding elements to its end and shrinks by removing elements from its front.
- Unlike a stack, a queue is a structure in which both ends are accessible: one for adding new elements (usually called the rear) and one for deleting elements already in the queue (usually called the front).
- Therefore, the last element has to wait until all the elements preceding it on the queue are removed.
- The access pattern to a queue is called FIFO (First In First Out).

Queues (cont.)

- Queue operations (interface) are similar to stack operations. The following operations are needed to properly manage a queue:
  - clear(q) – empty the queue named q, in queue order. O(n).
  - isEmpty(q) – checks to see if queue q is empty. O(1).
  - isFull(q) – checks to see if queue q is full. O(1).
  - enqueue(q, x) – inserts element x at the rear of queue q. O(1).
  - dequeue(q) – removes the element at the front of queue q. O(1).
  - peek(q) – looks at the element at the front of queue q without removing it from the queue. O(1).

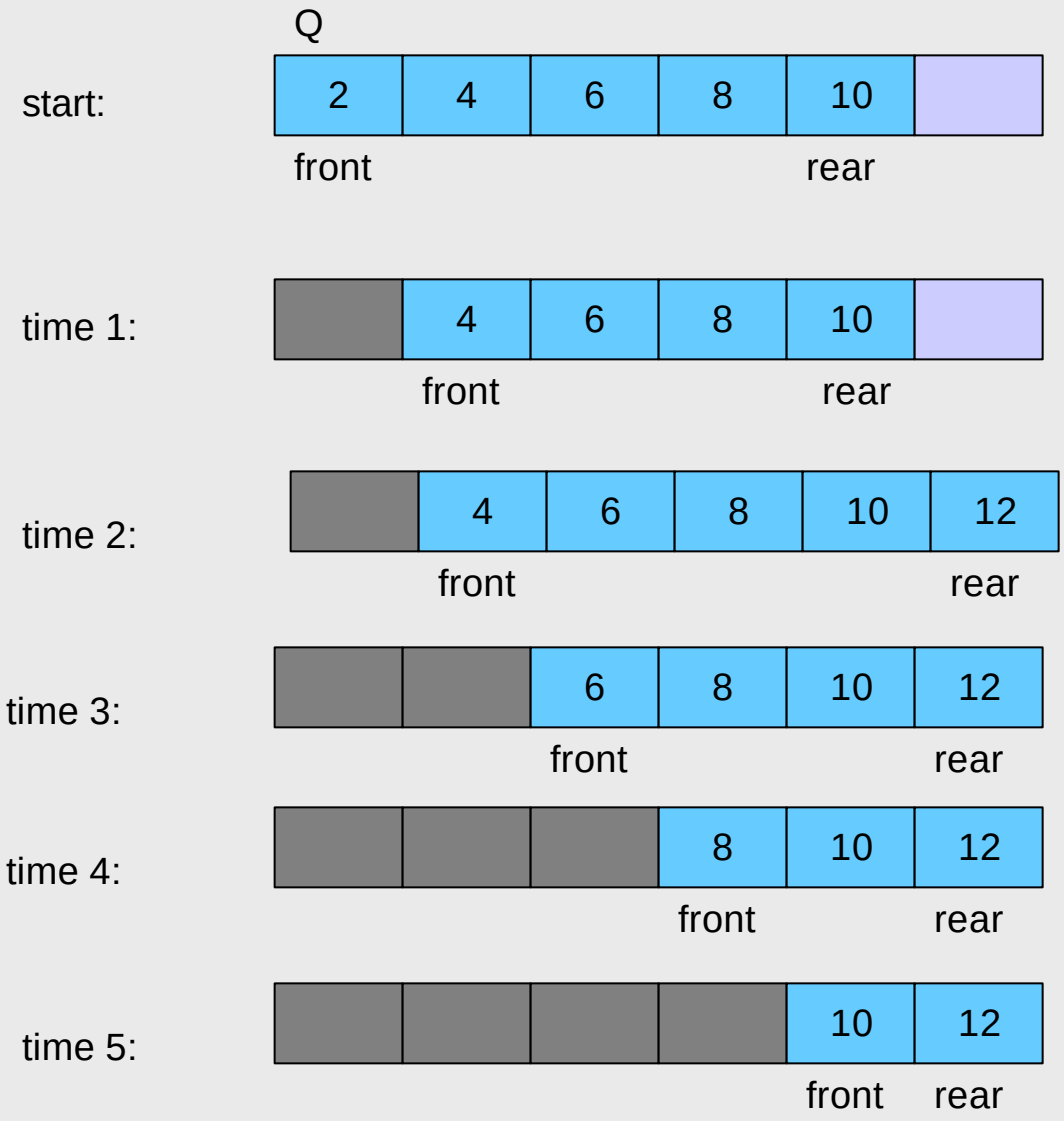
FIFO Nature of a Queue



Sequence of operations

Time	Operation
1	insert 12
2	remove
3	insert 14
4	insert 16
5	remove

Example - Array Based Queues

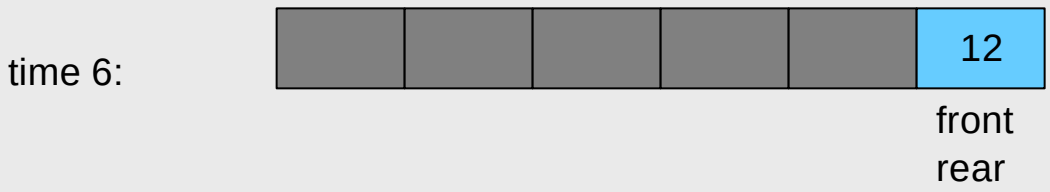


Sequence of operations

Time	Operation
1	remove
2	insert 12
3	remove
4	remove
5	remove
6	remove
7	remove

Notice that the queue now appears to be full even though there are locations available in the array!

Example - Array Based Queues (cont.)



Sequence of operations

Time	Operation
1	remove
2	insert 12
3	remove
4	remove
5	remove
6	remove
7	remove

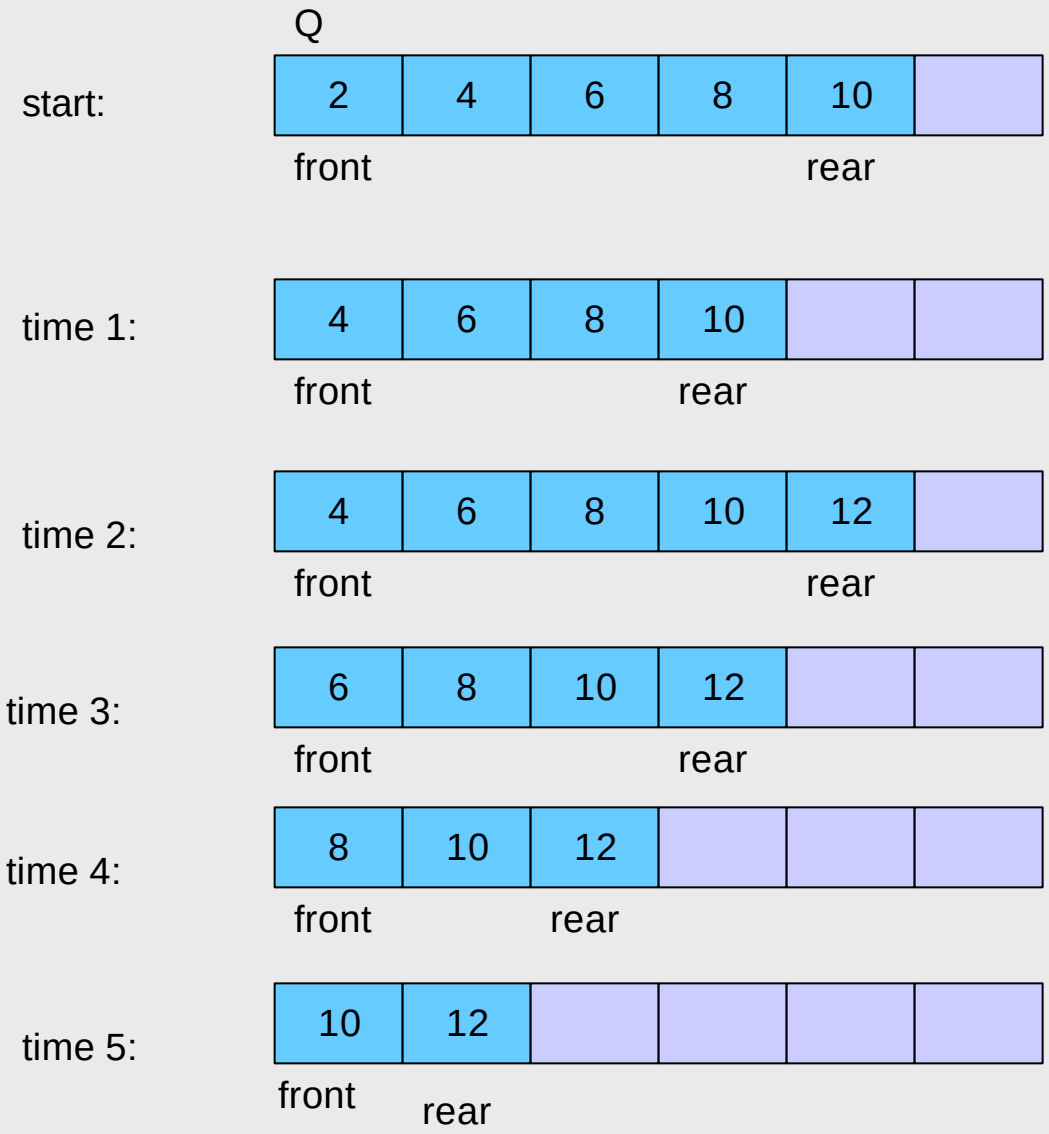
Notice that the queue now appears to have no space available even though no locations in the array hold any values!

front index has gone "out of range"

Problems with an Array-based Queue

- As the previous example illustrated, there is a problem with implementing a queue using an array similar to the way we did with a stack.
- The basic problem is that removing elements from the queue causes the front of the queue to move.
- The next page illustrates a brute-force method for handling the problem. What is the problem with this brute force technique?

Handling an Array Based Queue

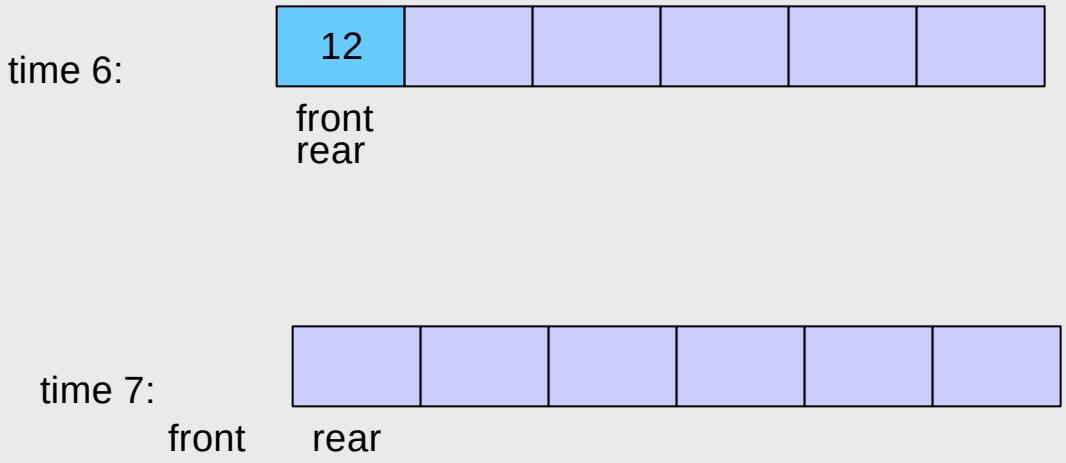


Sequence of operations

Time	Operation
1	remove
2	insert 12
3	remove
4	remove
5	remove
6	remove
7	remove

Notice that the array now has room to add elements to the queue.

Handling an Array Based Queue (cont.)



Sequence of operations

Time	Operation
1	remove
2	insert 12
3	remove
4	remove
5	remove
6	remove
7	remove

Problems with an Array-based Queue (cont.)

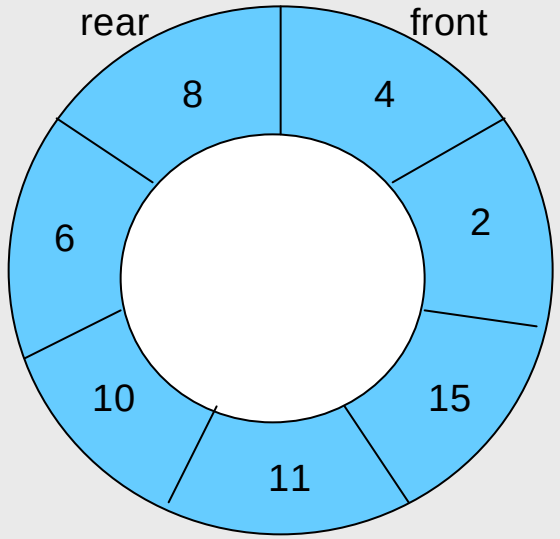
- As the previous example illustrates, this brute-force technique for handling the deletions from a queue is extremely costly in that all of the elements that remain in the queue after the deletion of the element at the front of the queue must be moved forward one position in the array.
- On average, this is an O(n) time per deletion! However, conceptually, we know that the time to delete from a queue should be O(1).
- How can we make efficient use of the array, as well as implement our insertion and deletion algorithms efficiently?

Problems with an Array-based Queue (cont.)

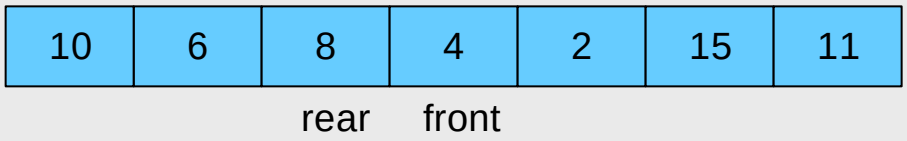
- For array based implementations of a queue, a very common and efficient implementation views the array as if it were circular.
- In a circular implementation, the queue is considered to be full whenever the front of the queue immediately precedes the rear of the queue in the counterclockwise direction.
- The examples on the following pages should help you to visualize a "circular" array. Circular arrays are sometimes referred to as ring buffers (particularly in OS terminology).

Circular Array Implementation of a Queue

normal array implementation: queue is full



visualization of a queue implemented as a circular array

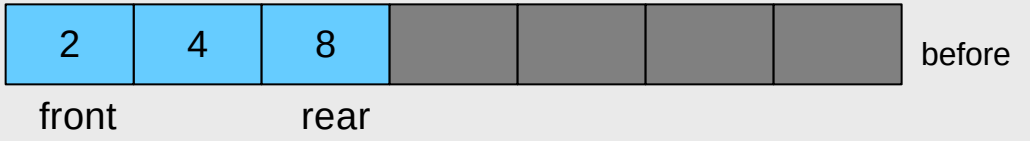


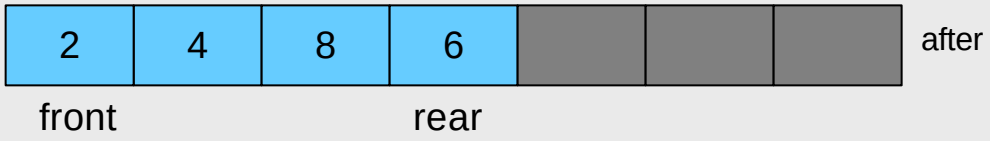
circular array implementation: queue is full

Circular Array Implementation of a Queue (cont.)

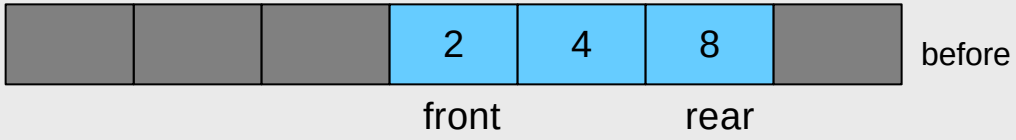
- The next several slides illustrate the operation of a circular array based implementation of a queue.
- The normal implementation (brute-force) is also shown for comparative purposes. However, remember that it is extremely inefficient due to the amount of data movement required by dequeue operations.
- The scenario begins at some point in time before which other enqueue and dequeue operations have occurred on the queue. Our scenario begins with some elements already in the queue. As you can see on the next page, these elements were enqueued in the order of: 2, 4, and 8. The scenario continues by enqueuing 6, enqueuing 10, dequeue, enqueuing 18, dequeue, dequeue, dequeue, enqueuing 9, dequeue, dequeue, and finally one last dequeue which empties the queue at this point.

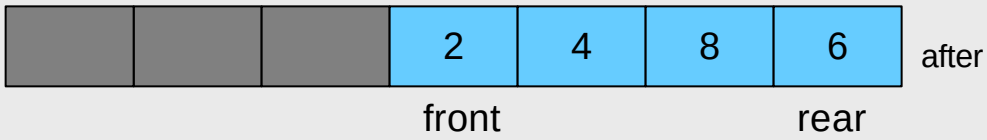
Circular Array Implementation of a Queue (cont.)



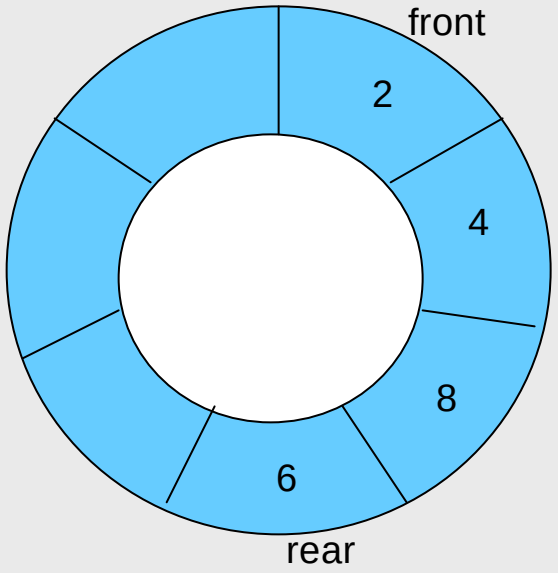


normal array implementation





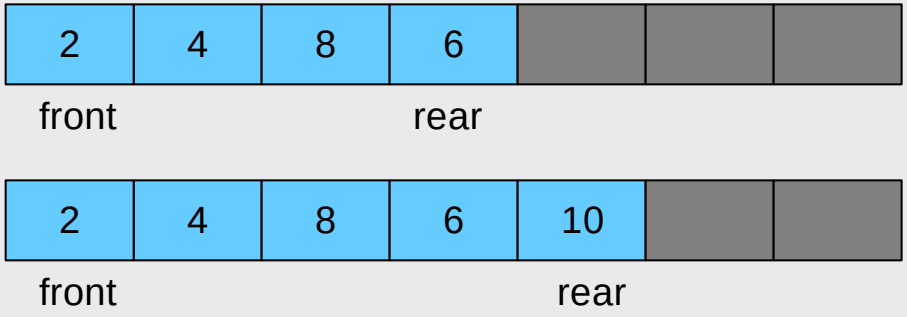
circular array implementation



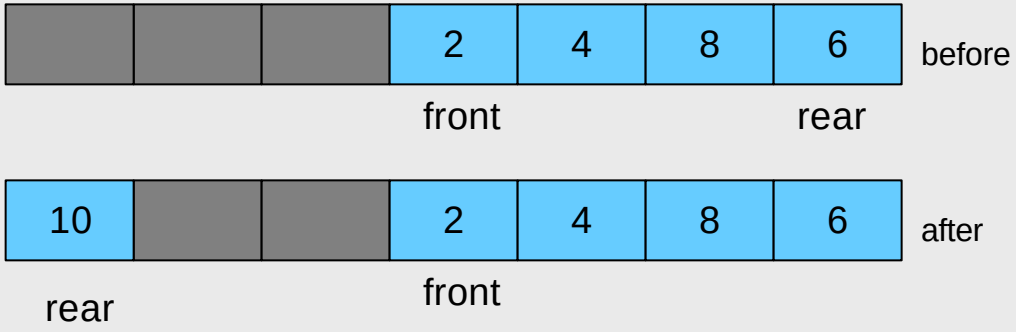
visualization of a queue implemented as a circular array after insertion of element 6

Enqueue element 6

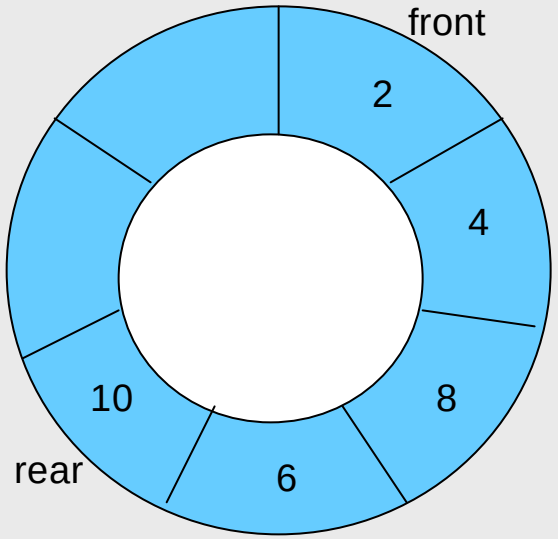
Circular Array Implementation of a Queue (cont.)



normal array implementation



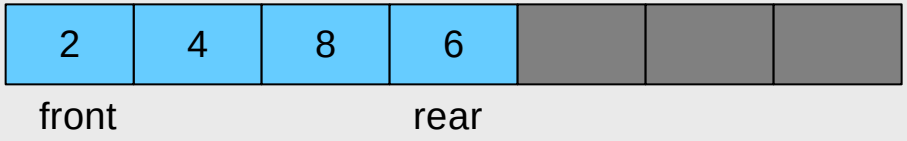
circular array implementation



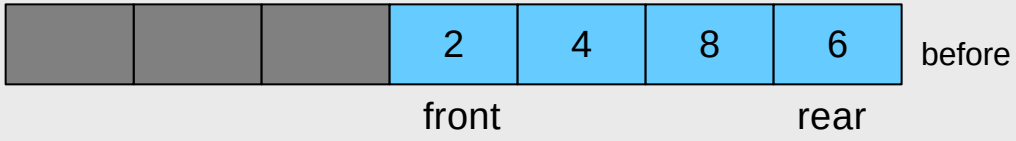
visualization of a queue implemented as a circular array after insertion of element 10

Enqueue element 10

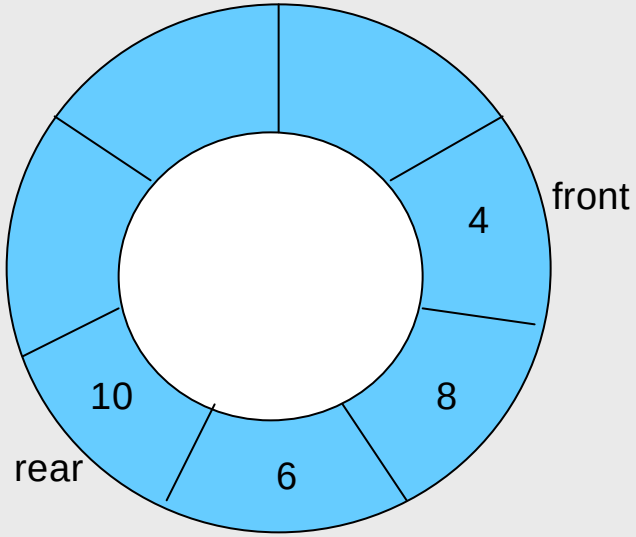
Circular Array Implementation of a Queue (cont.)



normal array implementation



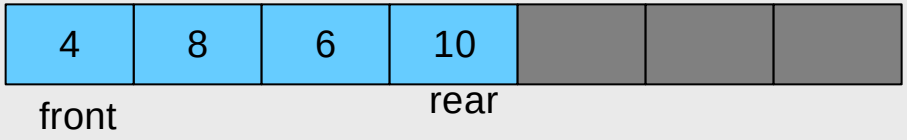
circular array implementation

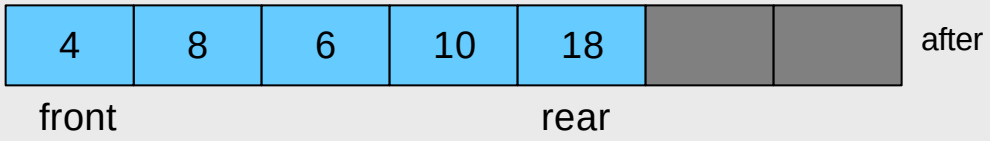


visualization of a queue implemented as a circular array after dequeue operation

Dequeue

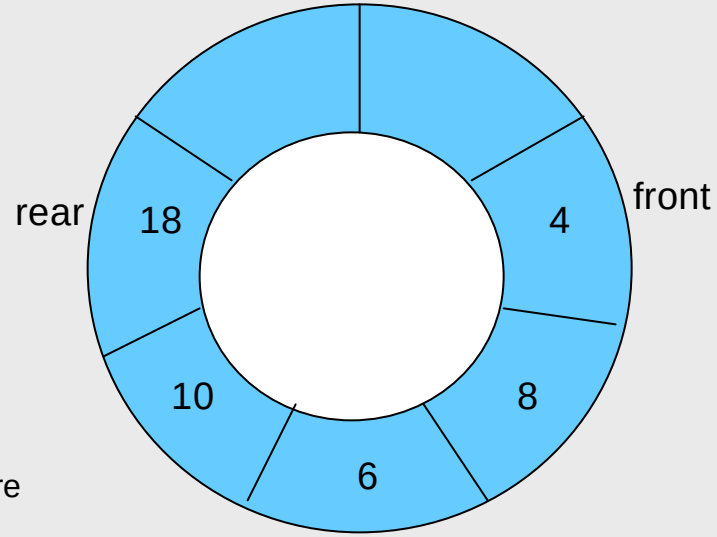
Circular Array Implementation of a Queue (cont.)





normal array implementation

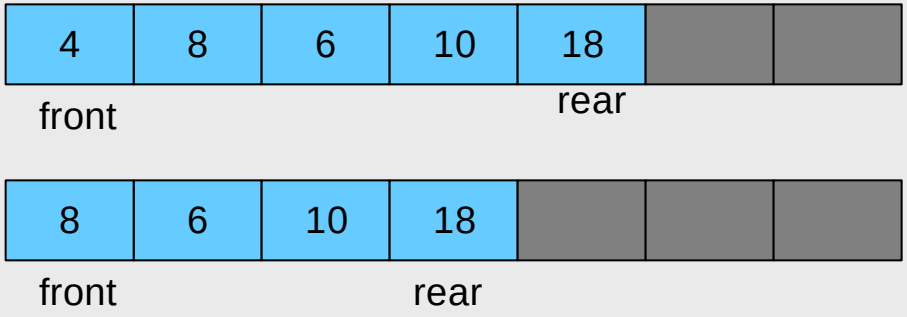
circular array implementation



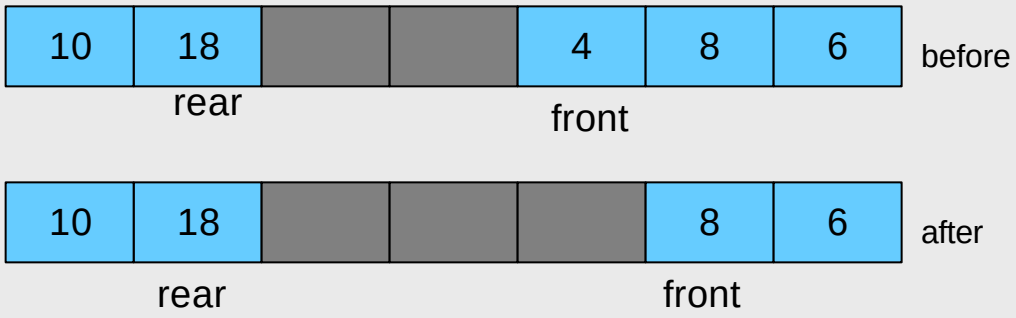
visualization of a queue implemented as a circular array after enqueue element 18

Enqueue element 18

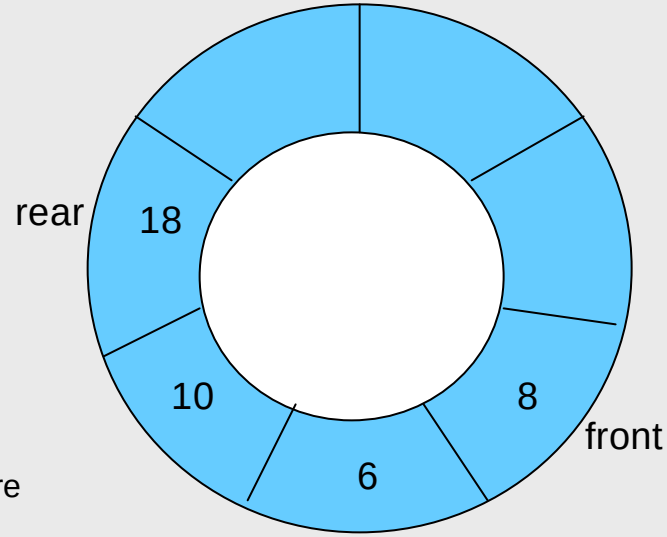
Circular Array Implementation of a Queue (cont.)



normal array implementation



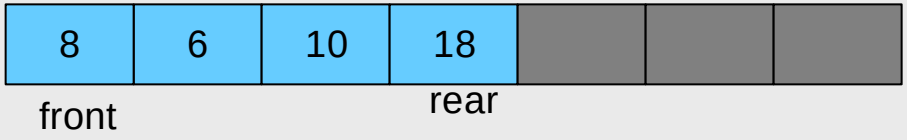
circular array implementation

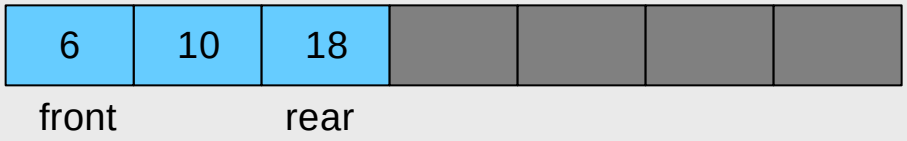


visualization of a queue implemented as a circular array after dequeue operation

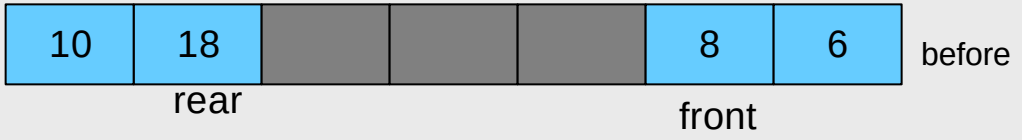
Dequeue

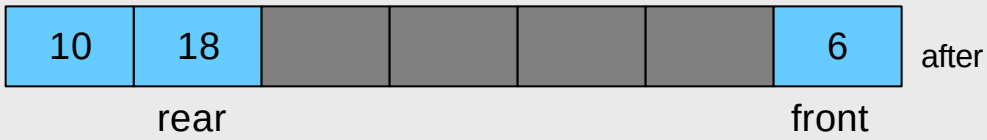
Circular Array Implementation of a Queue (cont.)



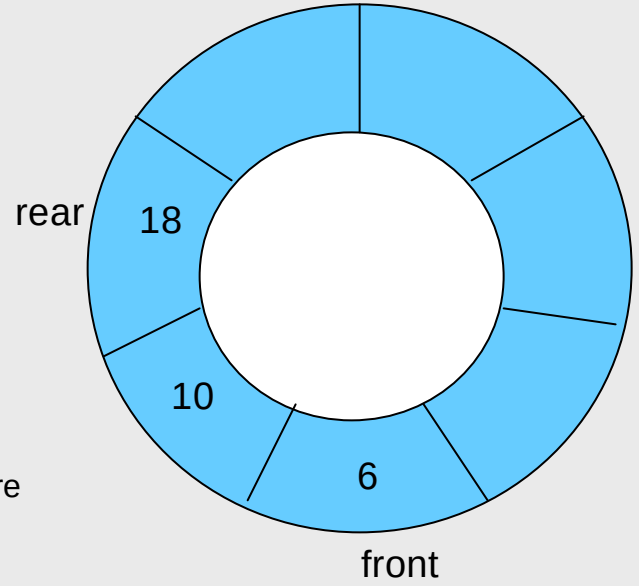


normal array implementation





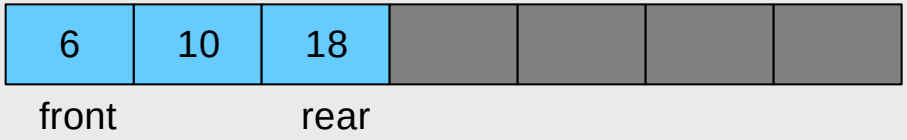
circular array implementation

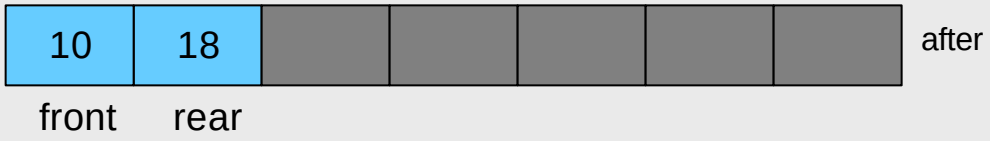


visualization of a queue implemented as a circular array after dequeue operation

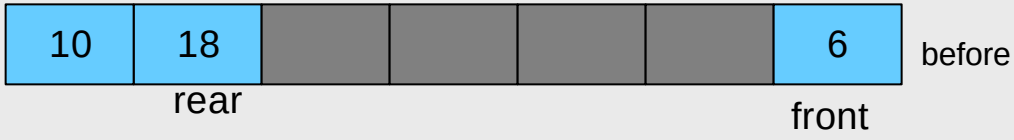
Dequeue

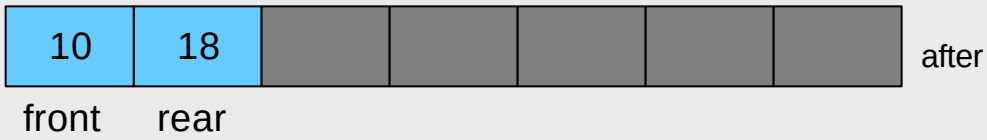
Circular Array Implementation of a Queue (cont.)



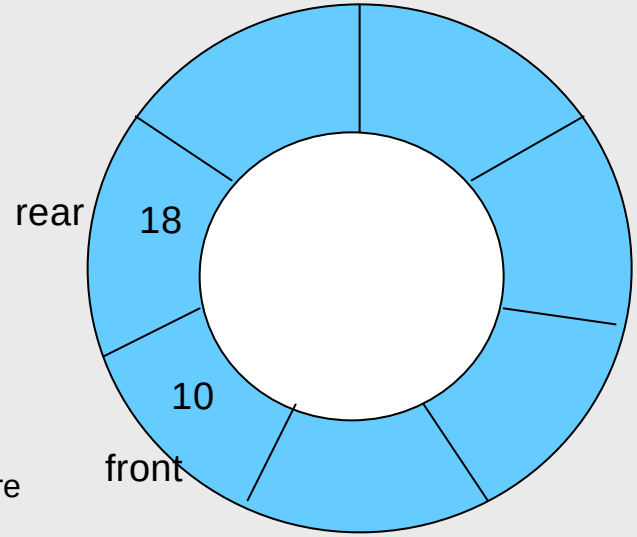


normal array implementation





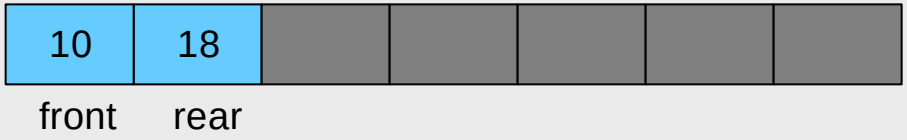
circular array implementation

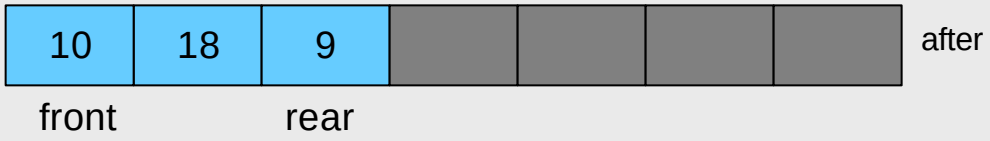


visualization of a queue implemented as a circular array after dequeue operation

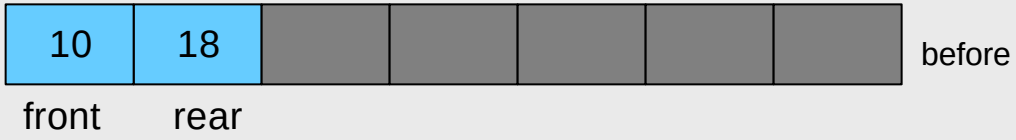
Dequeue

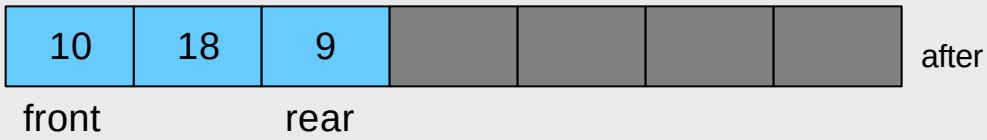
Circular Array Implementation of a Queue (cont.)



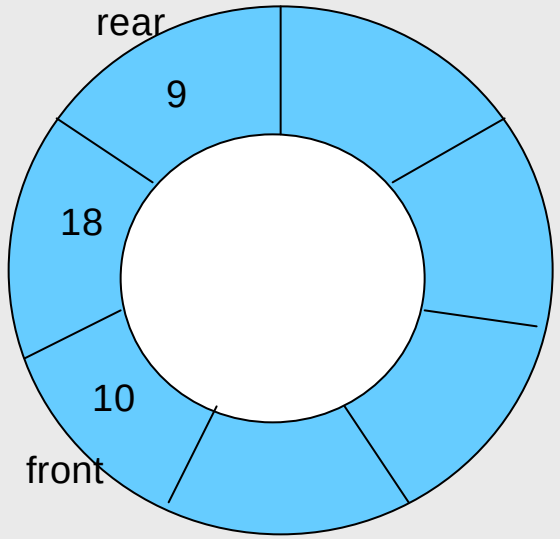


normal array implementation





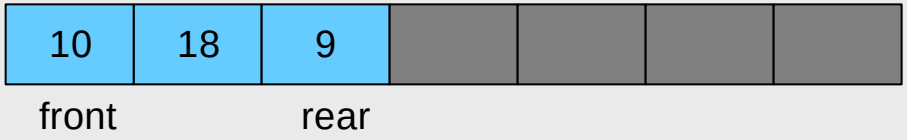
circular array implementation



visualization of a queue implemented as a circular array after enqueue element 9

Enqueue element 9

Circular Array Implementation of a Queue (cont.)



front rear

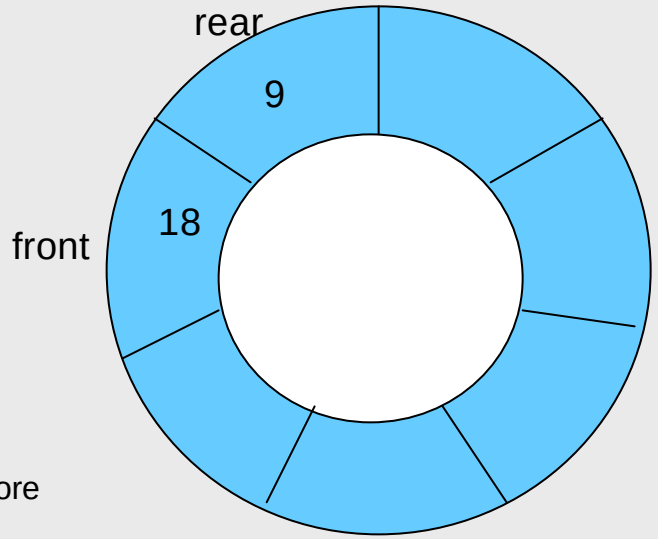
front rear

normal array implementation

front rear

front rear

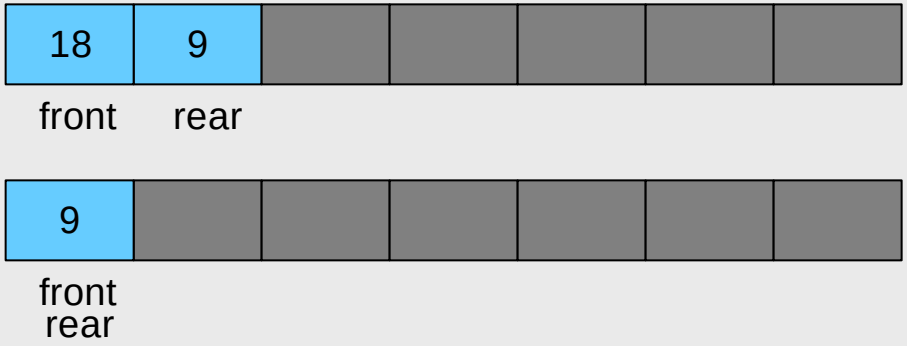
circular array implementation



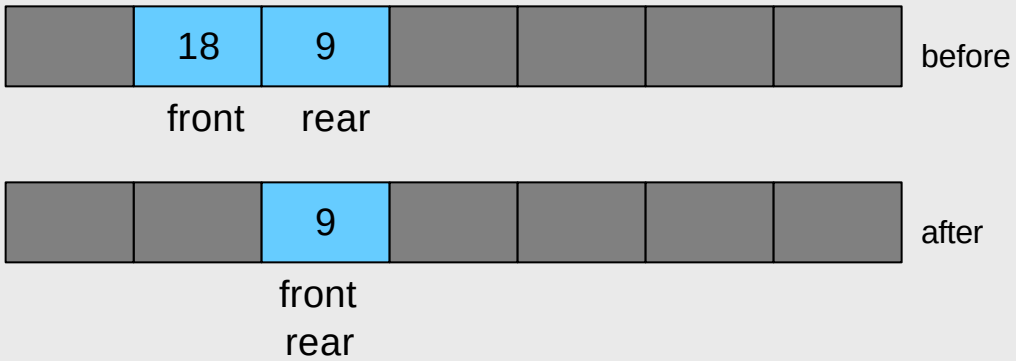
visualization of a queue implemented as a circular array after dequeue operation

Dequeue

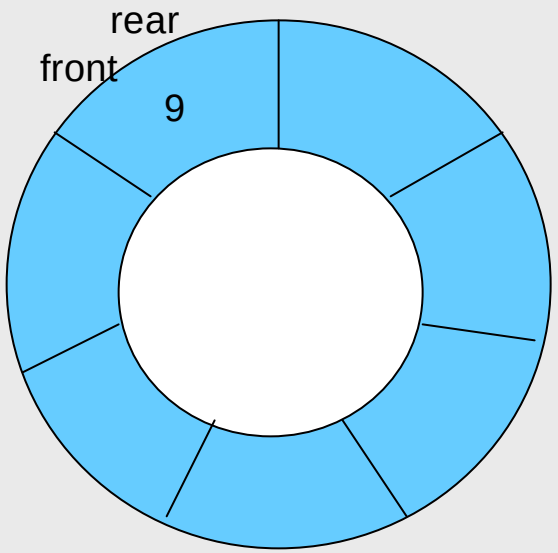
Circular Array Implementation of a Queue (cont.)



normal array implementation



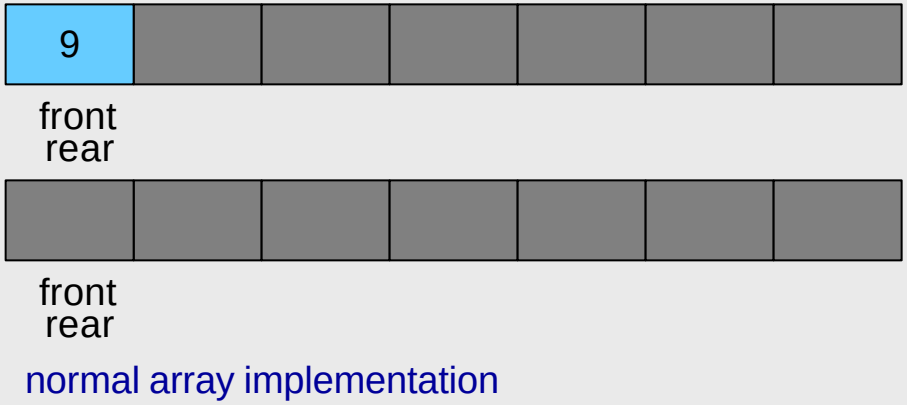
circular array implementation



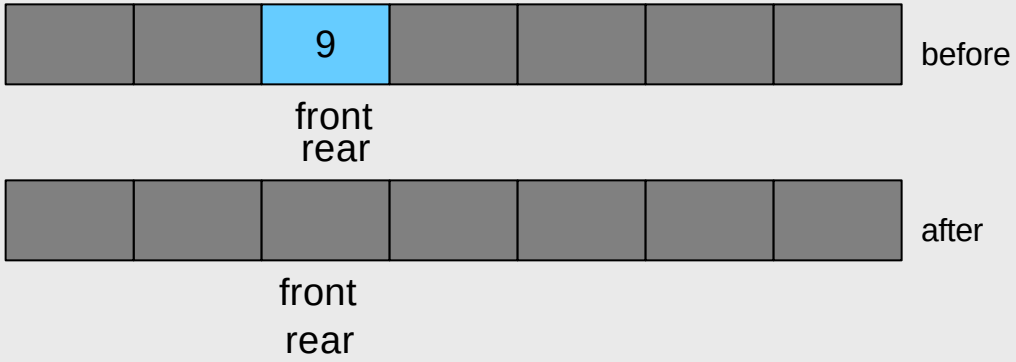
visualization of a queue implemented as a circular array after dequeue operation

Dequeue

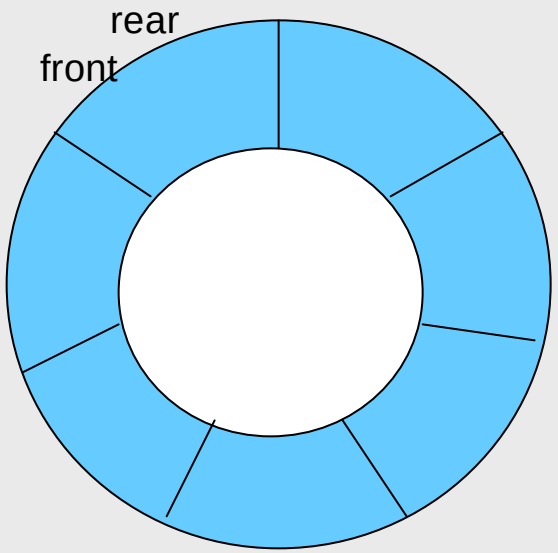
Circular Array Implementation of a Queue (cont.)



normal array implementation



circular array implementation

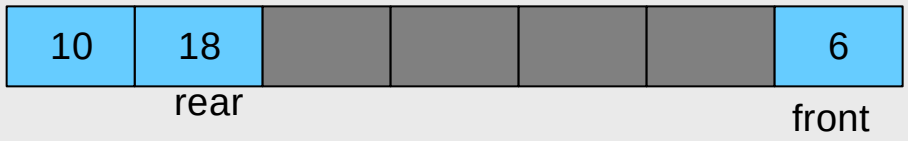


visualization of a queue implemented as a circular array after dequeue operation

Dequeue – queue empties

Array-based Implementation of a Queue

- There is a circular array implementation of a queue in C on pages 434-435 in your textbook.
- Note that this implementation implements the circular nature of the array using modulo arithmetic which simplifies the actual "movement" of front and rear within the queue.
- For example, suppose we have the situation shown below (same as on page 25):



if we dequeue at this point we, need to increment front (index 6) by 1, but index 7 is out of bounds, however, (front +1)%7 = 0 which is exactly the index where front should be located after the dequeue operation.