DATA STRUCTURES

Course Notes

Fatimah Adamu-Fika

Fatimah Adamu-Fika
Department of Cyber Security
Faculty of Computing
Air Force Institute of Technology
Nigerian Air Force Base
PMB 2104 Kaduna
Kaduna State, Nigeria
f.a.fika@afit.edu.ng

UNIT 6 Queues

Unit Contents

This unit examines another basic data structure, queues and identifies common operations that must be provided with queue implementation. Learners will also study how to arrays and linked lists can be used to implement a queue, and the advantages and disadvantages of using both . The unit also presented various types of queues, some applications of queues, and how to use the queue data structure to solve problems.

Upon completion of this unit, readers should be able to:

- Understand basic concepts of queue.
- Describe queue operations.
- Know different types of queues.
- Implement the queue ADT using both array and linked list data structures.
- Solve problems using queue and implement these solutions.
- Understand and describe similarities and differences between stacks and queues.
- Know how to implement a stack with queues and implement a queue with stacks.

Key concepts

• Queue ADT • enqueue operation • dequeue operation • FIFO • LIFO • underflow • overflow • circular queue • priority queue • deque

6.1 Overview, Concepts and the Queue ADT

A queue is an abstract data type that represents a collection of objects, called **elements**. A queue ADT allows insertion operation at one end of the queue and deletion operation at another end. At any given time, we can access the front element of the queue and the back of the queue. Hence, the element which is placed first is accessed first.

A queue is empty when it contains no elements. Like with stack, to efficiently utilise a queue, we need to check the status of queue.

The queue data structure can be implemented using arrays (usually static arrays) or linked lists (usually singly linked list). An array implemented queue can organise only a limited number of elements, i.e., the queue has a static (fixed) size. A linked list implemented queue can organise an unlimited number of elements, i.e., the queue has a dynamic size.

- A queue is a linear data structure whose operations are done based on the **FIFO**, "first-in", "first-out", (or LILO, "last-in", "last out") principle. This means that the least recently added elements in the queue are removed first.
- In a queue, elements are added at the end of the queue and removed from the beginning of the queue.
- A queue is a structure that maintains two pointers. One pointer references the element that was least recently added in the queue, this pointer is called $\boldsymbol{front}$ (or **head**). And the other is called **rear** (or **tail**), and it references the element that is most recently added to the queue.
- In queue terminology, insertion operation is called $\boldsymbol{enqueue}$, and removal operation is called $\boldsymbol{dequeue}$.
- The number of elements currently organised in the queue is referred to as its $\boldsymbol{size}$.
- For a queue, the total number of elements that can be enqueued into the queue is called its $\boldsymbol{capacity}$.
- If the queue is full and does not contain enough space to accept the element to be enqueued, the queue is then considered to be in an $\boldsymbol{overflow}$ state.
- If a queue has no element, it is known as an $\boldsymbol{empty\ queue}$. For an $\boldsymbol{empty\ queue}$, both $\boldsymbol{front}$ and the $\boldsymbol{rear}$ are $\boldsymbol{-1}$ (for an array implemented queue), or $\boldsymbol{null}$ (for a linked list implemented queue).
- If the queue is empty and a dequeue operation is attempted on it, the queue is then considered to be in an $\boldsymbol{underflow}$ state.

6.1.1 Specifying a Queue ADT

The common set of operations that can be performed on a queue are:

- A constructor for creating an empty queue.
- **Enqueue**: adds an element to the **rear** of the queue if it is not **full**.
- **Dequeue**: removes an element from the **front** of the queue if the queue is not **empty**.
- **Peek**: If queue is not **empty**, it looks at the element at the **front** of the queue without modifying the queue.

- **isEmpty** or **empty**: checks if the queue is **empty**, and only returns true if the queue is **empty**.
- **isFull** or **full**: Checks if the queue is **full** and returns true if the queue is **full**. This operation is only implemented for a static queue.
- **Search**: checks the queue for an element and returns true or the location of the element if it exists in the queue.
- **Size**: returns the size of the queue.

6.2 Representation

How a queue is represented in memory depends on its implementation. The elements in an array-based queue are organised in contiguous memory locations. Whereas the elements in a linked-list-based queue are organised in random memory locations and are ordered via the reference links stored in the nodes.

Figure 6-1 depicts what a typical queue looks like.

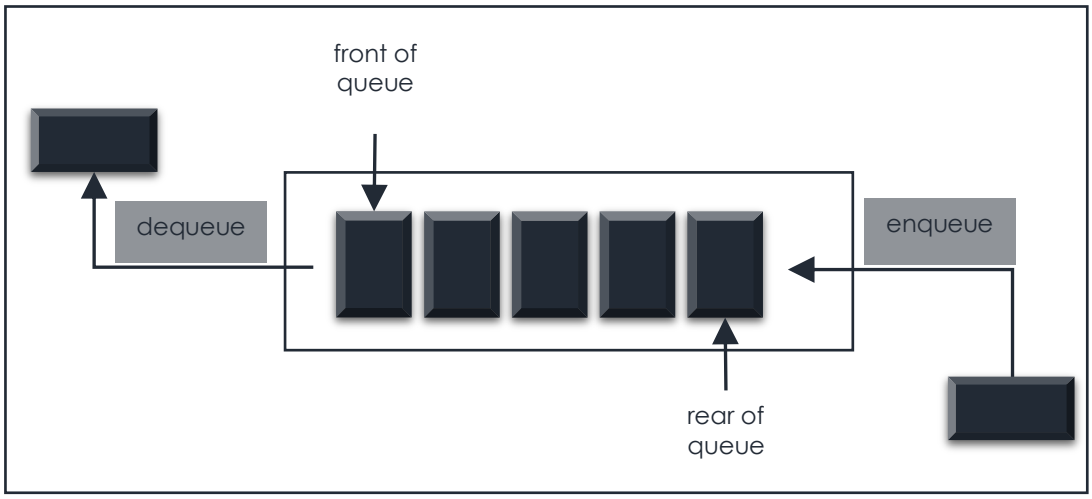


Figure 6-1. Simple representation of a queue.

6.3 Implementation

We actualise the queue ADT to turn it into a useable data structure. List structures such as array and linked list can be used to implement a queue. Representing a queue with arrays or linked lists is a natural idea. Array implementation of a queue is usually static. And linked list implementation is dynamic and in general does not become full. Now, we will look at how to use these two list structures to implement queues.

6.3.1 Array Implementation

Since the array implementation is static, we need to declare the $\boldsymbol{capacity}$ (maximum size, i.e., the total number of elements the queue can store) of the array ahead of time. We maintain an instance variable, $\boldsymbol{size}$, to keep track of the number of elements in the queue. An array, $\boldsymbol{arrQueue}$, that stores $\boldsymbol{size}$ elements, with the most recently added element in $\boldsymbol{arrStack[size-1]}$ and the least recently added element in $\boldsymbol{arrStack[0]}$. This principle allows us to insert at the end of the queue in $\boldsymbol{arrStack[size]}$, and delete elements at the start of the array in $\boldsymbol{arrStack[0]}$ without moving any other elements in the queue. As $front$ is usually the index of the first element in the queue

and $rear$, the index of the last element, we can compute $size$ by adding 1 to $rear - size$, i.e., $\boldsymbol{size = rear - front + 1}$. So, we do not need to keep incrementing or decrementing $size$ with every enqueue or dequeue operation. When the queue is full, $rear$ will have its maximum value, which is $\boldsymbol{capacity - 1}$.

A queue can be implemented using array as follows

6. Define an instance variable $\boldsymbol{capacity}$ with specific value.
7. Declare all the **functions** used in queue implementation.
8. Create a one-dimensional array with fixed size $\boldsymbol{capacity}$.
9. Define an integer variable $\boldsymbol{front}$ and initialise it with **-1**.
10. Define an integer variable $\boldsymbol{rear}$ and initialise it with **-1**.
11. Implement method calls for queue operations. We will explore this in more detail, in the next section of this unit.

6.3.1.1 Performance and limitations

So far, we have been discussing linear (or simple) queue. An array-based queue has fast operations, all operations are completed in constant time. The implementation has a space complexity of linear time. Although the array implementation of queue has efficient time, it has a major limitation: capacity is fixed, which may also lead to the risk of space being wasted. We have to know the upper bound of growth and allocate memory accordingly, which does not change during execution. If the $rear$ reaches the end position of the queue, then there might be a chance that some vacant spaces are left in the beginning which cannot be reused. The queue has to be reset or a new one with larger capacity has to be created, and elements of the old one would have to be copied unto the new queue If the array is full and there is another enqueue operation, we will then encounter an implementation-specific exception.

The case where the array is full is not an exception defined in the Queue ADT. If the array is filled, then we have a few options. The first option is to throw an exception. Another way is to disregard the excess element being enqueued. We can also adopt the concept of a circular structure to circumvent the limitation of space wastage when rear reaches the limit of the queue.

Thus, a queue can be implemented as a cyclic array.

6.3.1.2 Circular queue

Circular Queue is also called a **ring-buffer**. The circular queue is an improvement over the simple array queue, it logically connects the last position of the queue to the first position of the queue. It still operates on the FIFO principle. Its operations and algorithms remain the same as with the simple queue. To form the circle, the front and rear indices are calculated by performing a modulo operation on indices (as computed in linear queue) against the capacity of the queue: $\boldsymbol{front = (front + 1)\ \%\ capacity}$ and $\boldsymbol{rear = (rear + 1)\ \%\ capacity}$.

Example 6.1: Assuming the array size for a simple queue is 8, and we enqueued four elements to $queue$ and did three dequeue operations. When we try to enqueue two more elements to $queue$, there will be an $\boldsymbol{overflow}$ on the second enqueue operation even though we have three empty slots at the beginning of $queue$ as shown in Figure 6-2.

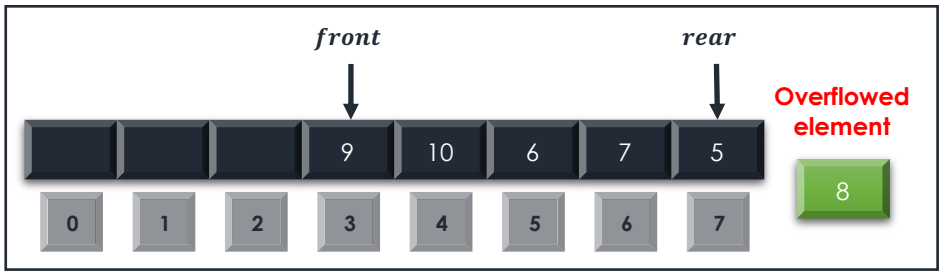


Figure 6-2. A simple queue with empty spaces at the beginning of the queue.

Using a circular queue efficiently reutilises the available space. It doesn't cause an overflow if there is available space at the front of the queue. To check for an $overflow$, all we need to check is the $size$ of the queue. If $size$ of $queue$ is less than $capacity$ of $queue$, i.e., $size < cpapcity$, then there is at least one empty space to add a new element in the queue. Our second enqueue operation will be successful in a circular queue. The element will be inserted at the first available empty space, in our example, this will be index 0. Figure 6-3 shows our queue as a circular queue.

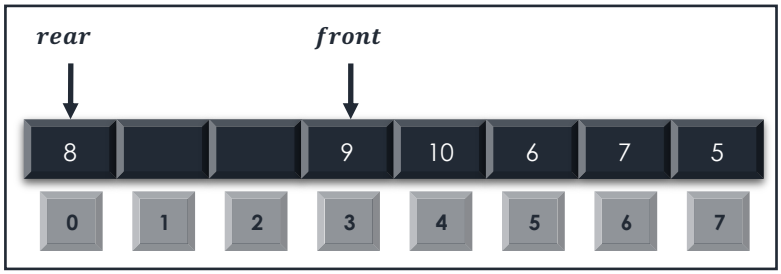


Figure 6-3. A circular queue with empty spaces being reused.

The circular queue works like a ring, hence why it is called the ring-buffer. Figure 6-4 shows ring view of our circular queue.

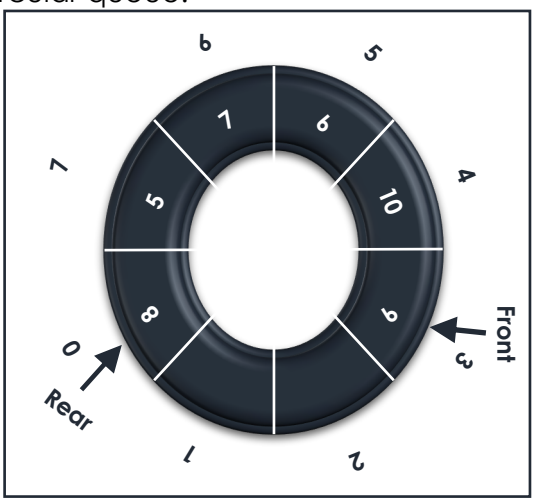


Figure 6-4. Simple ring representation of the circular queue from Figure 6-3.

Although, the circular queue reuses freed up space from dequeue operations, it does not solve the problem of limited capacity of the array implementation. An overflow will still occur when the queue is at full capacity. For example, in **Example 6.1** above, if we enqueue two more elements and try to enqueue a third without performing any dequeue operation, our queue will go into overflow.

6.3.2 Linked List Implementation

A linked list can be used to implement a queue, this will give the queue a dynamic capacity. Remember, even though the elements of the queue are logically linear, physically, they are stored in arbitrarily memory locations that may not be close to each other. The elements are linked with pointers. A linked-list-based Implementation of the queue ADT is required when the queue needs to grow and shrink dynamically (without the overhead of copying the entire list when more memory is needed).

We maintain an instance variable $\boldsymbol{front}$ that stores a reference to the least recently added element, and $\boldsymbol{rear}$ that stores a reference to the most recently added element. This notion allows us to insert elements at the end of the linked list and delete elements at the beginning of it, without accessing links of any other elements in the linked list. We do not have to worry about the size when the queue grows. The entire memory pool is the limit for a linked-list-based queue. Since the queue only requires us to keep track of only a single link for its manipulations (we only move between elements in one direction), singly linked list is the natural choice for implementing it. The, $front$ is basically the $head$ pointer of the linked list and $rear$ is the $tail$ pointer. Hence, enqueue is the same as adding a new $tail$ and dequeue is equivalent to deleting the $head$.

To represent a queue using a linked list, we need to set the following things before implementing actual operations.

4. Define a $\boldsymbol{Node}$ structure with two attributes $\boldsymbol{data}$ and $\boldsymbol{next}$
5. Define a $\boldsymbol{Node}$ pointer $\boldsymbol{front}$ and set it to $\boldsymbol{NULL}$.
6. Define a $\boldsymbol{Node}$ pointer $\boldsymbol{rear}$ and set it to $\boldsymbol{NULL}$.
7. Implement method calls for queue operations.

To keep the size operation constant time, we maintain an instance variable, $\boldsymbol{size}$, to keep track of the number of elements in the queue. Whenever we enqueue an element into the queue, we increment $size$ and decrement it when we dequeue an element out of the queue.

6.3.3 Priority Queue

A priority queue is a type of queue, where every node has some sort of priority attached to it. The nodes are processed based on this priority. The application of the queue and its implementation defines the domain of the priority.

- A higher priority element is processed before a lower priority one.
- If two elements are of the same priority, they will be processed according to their sequence in the queue.
- When deletion is performed, the element which has the highest priority is removed. This element is usual at the front of the queue.

- When insertion is performed, the priority of the elements determines its position in the queue. Higher priority elements are placed towards the front of the work, this means, the highest priority element in the queue will always be the one at the front of the queue. And the lowest priority will always be at the end of the queue. Since insertions is determined by the priorities of the elements, the priority queue does not maintain a $tail$ pointer.

Figure 6-5 shows an example representation of the priority queue. In this example, the number on the node denotes the priority of the element, not the value of the element. Also, in this particular case, larger valued priority indicates higher priority.

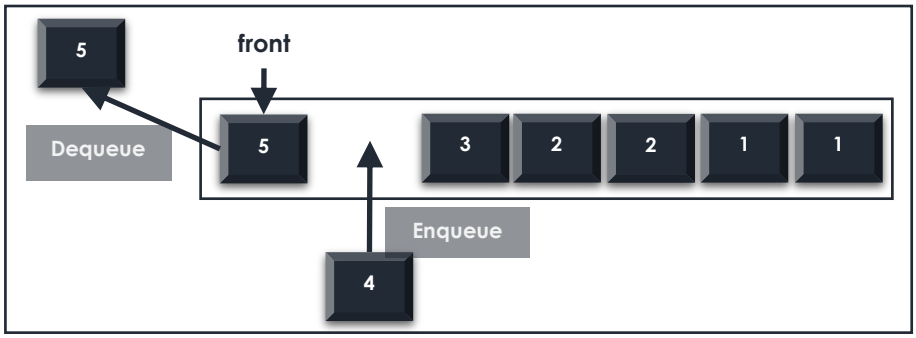


Figure 6-5. A simple representation of a priority queue.

There are two types of priority queue:

- Ascending order priority queue, where a lower numbered priority is considered to have a higher priority than a larger numbered one.
- Descending order priority queue, where a larger numbered priority is given as having a higher priority than a lower numbered one.

A priority queue can be implemented using arrays or linked list.

6.3.4 Double Ended Queue (Deque)

A double ended queue also called a deque (pronounced *deck*) is a queue variant where enqueue and dequeue operations are done from both ends of the queue. A deque can be implemented with an array or a linked list.

Figure 6-6 depicts a simple deque. We will discuss more on deque in later section of this unit.

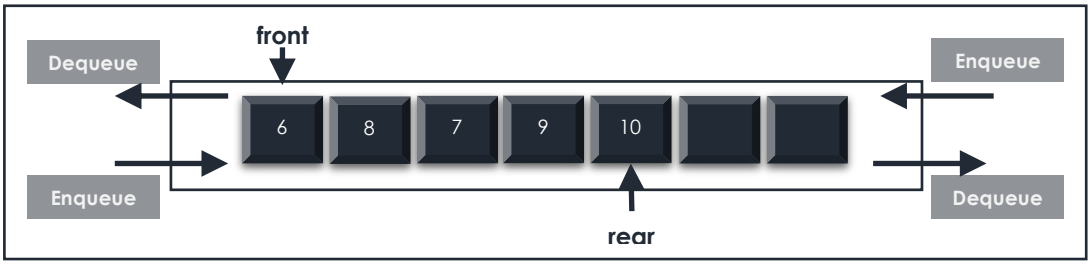


Figure 6-6. A simple representation of a deque.

6.4 Basic Operations

As we have mentioned earlier, queue operations may involve initialising the queue, using it and then de-initialising it. In addition to these, we mentioned the two primary operations of a queue: enqueue and dequeue. Also, we indicated the need to check the status of the queue using operations such as peek, full and empty.

Now we study the procedures that make queue usable.

6.4.1 Insertion Operation (Enqueue)

The process of placing a new element at the back of a queue is known as an **enqueue operation**. How the queue is implemented determines how the enqueue operation is performed.

6.4.1.1 Putting an element in an array-based queue

Enqueue operation involves a series of steps:

1. Check if the queue is full.
2. If the queue is $full$, produces an error and exit.
3. If the queue is not $full$, increments $\boldsymbol{rear}$ to point to the next empty space.
4. Add new element to the queue location where $\boldsymbol{rear}$ is pointing.
5. Check if queue was previously empty.
6. If $empty$, point $front$ to the new element.

6.4.1.1.1 Enqueue algorithm for an array-based (simple) queue

Algorithm Enqueue($\boldsymbol{element}$ **)**
Input: $\boldsymbol{element}$, the new element to be added.
Output: A $queue$ with a newly enqueued $\boldsymbol{element}$ at $\boldsymbol{rear}$ position, if $queue$ is not $full$.
Data Structure: An array *queue* with pointer to the end position, $\boldsymbol{rear}$ and pointer to the front position, $\boldsymbol{front}$.

Begin
1. **If** $queue$ is $full$ **Then**
2. **Print** "Queue Overflow!"
3. **Exit**
4. **Else**
5. $rear = rear + 1$
6. $queue[rear] = element$
7. **If** $queue$ **was** $empty$ **Then**
8. $front = front + 1$
9. **End If**
10. **End If**

End

6.4.1.1.2 Java implementation of the enqueue algorithm

```
public void enqueue( Object element )   {
      // Add data as the rear element of this queue.
```

```
        if( !isFull() )  {
              rear++;
              queue[rear] = element;

              if( isEmpty() )    {
                    front++;
              }
        else  {
              System.out.println("OVERFLOW! Queue is Full!");
        }
  }
```

6.4.1.1.3 How enqueuing elements into an array-implemented queue works

Example 6.2: Assuming we have an array queue with a capacity of 5 elements and we then add the elements 8, 7, and 4 at end of the queue (in the same sequence) . Figure 5-3 shows the state of the $queue$ after each enqueue operation. The $rear$ is usually set as -1 for an empty queue, and the $size$ is of course 0. When we enqueue an element into a queue (array implementation) the $rear$ is **increased by 1**. When we perform an enqueue operation, the $front$ pointer is not usually affected except if the queue was empty before the enqueue operation started. In this case, the $front$ will be -1, and we will need to increase it by 1 to point to the location of the newly inserted element. The process of enqueuing a queue is shown in Figure 6-7.

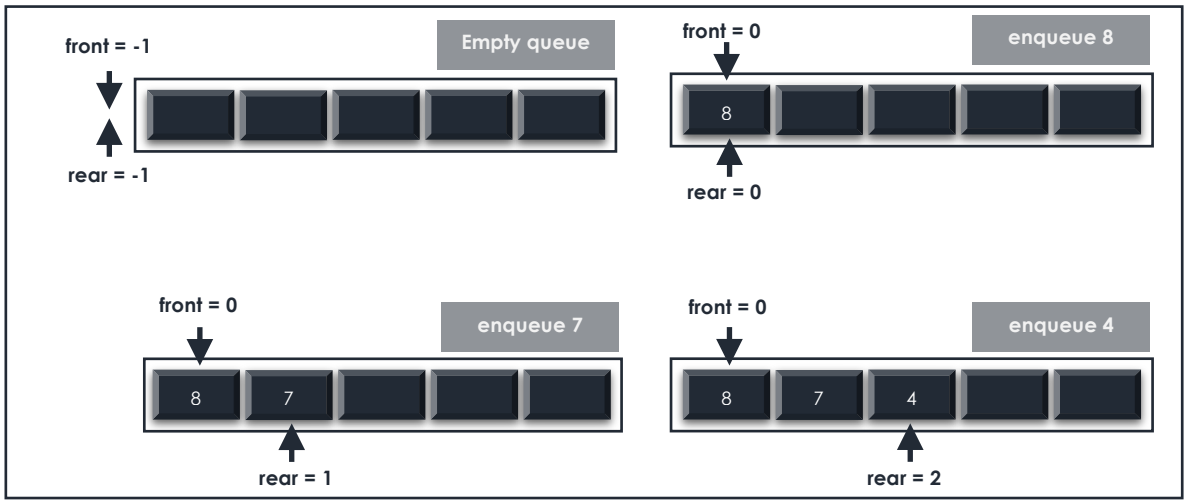


Figure 6-7. Simple representation of a (array) queue runtime with enqueue operation.

6.4.1.1.4 Enqueuing a circular queue

The enqueuing process of a circular queue is almost the same with that of a simple queue. The difference is in how we perform the $rear$ and $front$ pointers increment. After incrementing as for simple queue, we will then perform a modulo operation against the capacity of the queue. Thus, $rear$ is incremented as $rear = (rear + 1)\,\%\,capacity$, and $front$ as $front = front + 1\,\%\,capacity$. Another difference is, to keep track of the $size$ of the queue, we need to increase it by 1 after each enqueue operation.

Example 6.3: The following is the Java implementation of the enqueuing a circular queue.

```
public void enqueue( Object element )   {
      // Add data as the rear element of this queue.

      if( !isFull() )  {

            rear = ( rear + 1 ) % capacity;
            queue[rear] = element;

            if( isEmpty() )
                  front = ( front + 1 ) % capacity;

            size++;
      }
      else  {
            System.out.println("OVERFLOW! Queue is Full!");
      }
}
```

6.4.1.2 *The putting an element Into a linked-list-based queue*

If a linked list is used to implement the queue ADT, we check whether the queue is **empty** but NOT full. And we dynamically allocate the empty space needed for the new element.

1. Create a new node.
2. Set the $value$ of the new node as the element.
3. Set the successor of the new node as NULL.
4. Check whether the $queue$ is $empty$
5. If $empty$, set the newly created node as both the new $front$ and the new $rear$.
6. If not $empty$, set the $successor$ ($next\ link$) of $rear$ to the new node.
7. Set the newly created node as the new $rear$.
8. Increment the $size$ of the queue by 1.

6.4.1.2.1 *Enqueue algorithm for a linked-list-based queue*

Algorithm Enqueue(element)
Input: $\boldsymbol{element}$, the new element to be enqueued.
Output: A $\boldsymbol{queue}$ with a newly enqueued $\boldsymbol{element}$ at the $\boldsymbol{rear}$ position.
Data Structure: A singly linked list structure whose pointer to the head is known from $\boldsymbol{head}$, pointer to the tails is known from $\boldsymbol{tail}$, $\boldsymbol{front}$ is the pointer to the first node, $\boldsymbol{rear}$ is the pointer to the last node and $\boldsymbol{size}$ is the number of elements in $\boldsymbol{queue}$.
Begin

[create a new node with the given value, element]

1. $newNode = new\ SLLNode()$
2. $newNode \rightarrow data = element$
3. $newNode \rightarrow next = NULL$

[Check whether $queue$ is $\boldsymbol{empty}$
($front == NULL$ **OR** $front = rear \rightarrow next$)]

4. **If** $queue$ is $empty$ **Then**
5. $front = rear = newNode$

[Add node to the front of the list]

6. **Else**

```
7.          rear → next = newNode
8.          rear = newNode
9.      End If
10.     size = size + 1
End
```

6.4.1.2.2 Java implementation of the enqueue algorithm

```
public void enqueue ( Object element ) {

        // Add element as the rear element in the queue.
        QueueNode oldRear = rear;
        rear = new QueueNode( element, null );
        if( isEmpty() ){
              front = rear;
        }
        else {
              oldRear.next = rear;
        }
        size++;
}
```

6.4.1.2.3 How enqueuing elements onto a linked-list-implemented queue works

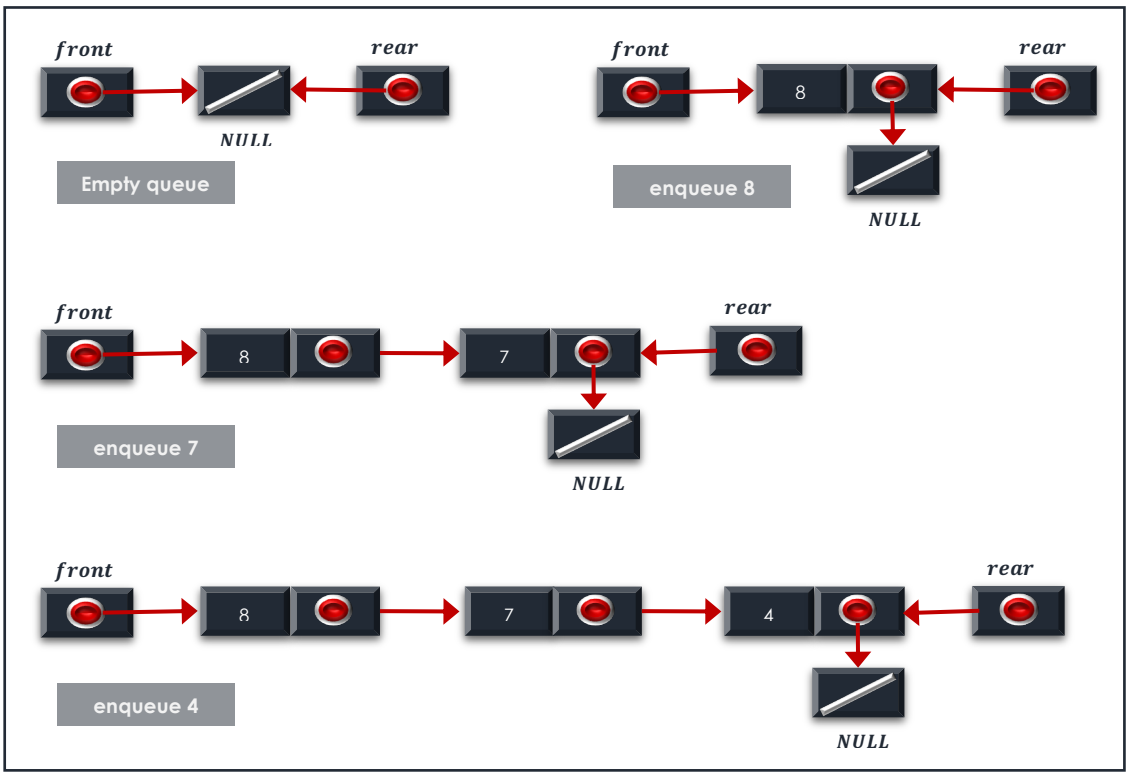


Figure 6-8. Simple representation of a (linked list) queue runtime with enqueue operation.

Example 6.4: Assuming we implemented a queue using linked list, and added the elements 8, 7 and 4 at its end. Figure 6-8 shows the state of the queue when it is created, as well as after each enqueue operation. For an empty queue, the $front$ and the $rear$ point to $null$, and the $size$ is set to 0. When we enqueue an element into

a queue, the successor of $rear$ is set to the new element and the new element becomes the new $\boldsymbol{rear}$.

6.4.1.3 Enqueuing a priority queue

Adding elements to a linked list priority queue, works slightly different from a standard linked list queue. A linked list priority queue only maintains the front pointer because insertion locations are determined by priority of the element. The node of the element has an extra attribute, priority.

Example 6.5: Let's assume, the priority of an element is an integer, and the lower the value of the integer the greater the priority. The following is the steps we will need to insert a new element into the priority queue.

1. Create new node, set its value as the new element, and its priority as the priority of the new element.
2. Check whether the queue is empty. If $\boldsymbol{front}$ is null, then the queue is empty.
3. If $empty$, insert the new element as $front$ of the queue.
4. If not empty, check whether the priority of the $front$ element is larger than the priority of the new element.
5. If priority of $front$ is larger, insert the new element at the $front$ of the queue. Set the successor of the new element to point to $front$ and make the new element to be the new $front$.
6. If priority of $front$ is lower, traverse the queue to find an element that has a larger priority than the new element, and insert the new element before it.
7. Create a temporary node, $\boldsymbol{current}$, to keep track of the element we are currently processing. Copy $front$ to it.
8. While $current$ is not the last node($current \rightarrow next$ is not null) and the priority of its successor is less than the priority of the new element ($current \rightarrow priority$ is less than element's priority), move $current$ forward along the queue ($current = current \rightarrow next$).
9. When we get to the end of the queue or find an element with a larger priority, we add the element there. Set successor of the new element to the successor of $current$. Set the new element as the new successor of $current$.
10. Increase $size$ of the queue by 1.

The following is the Java implementation of the enqueue method for a priority queue.

```
public void enqueue ( Object element, int priority )   {

      // Add element as a new node in this queue.
      QueueNode newElement = new QueueNode( element, priority, null );
      rear = new QueueNode( element,  null );
      if( isEmpty() ){
            front = newElement;
      }
      else   {
            if( front.priority > priority ){
                  newElement.next = front;
                  front = newElement;
            }
            else {
```

```
                    // Traverse the queue and find a
                    // position to insert new element
                    QueueNode current = this.front;
                    while( current.next != null &&
                                current.next.priority < priority )  {
                    current = currentNode.next;
                    }

                    newElement.next = current.next;
                    current.next = newElement;
                }

                size++;
            }
        }
```

6.4.2 Deletion Operation (Dequeue)

The process of removing an element from a queue is called a dequeue operation. Similar with enqueue operation, the operation is determined by the implementation of the queue.

6.4.2.1 Removing an element from an array-based queue

Since in an array-based queue the element is always removed from the front position. The dequeue operation does not take any value as input. We can use the following steps to dequeue an element from the queue.

1. Check whether $\boldsymbol{queue}$ is $\boldsymbol{empty}$.
2. If it is $\boldsymbol{empty}$, throw an exception and exit.
3. If it is **not** $\boldsymbol{empty}$, then delete $\boldsymbol{queue}[\boldsymbol{front}]$ and move $\boldsymbol{front}$ to point to the next element in the queue.
4. Return the element deleted.

6.4.2.1.1 Dequeue algorithm for an array-based queue

Algorithm Dequeue()
Input: none.
Output: Removed element is stored in $\boldsymbol{element}$, if $queue$ is not empty.
Data Structure: An array, $\boldsymbol{queue}$, with $\boldsymbol{front}$ as a pointer to the index of the first element.
Begin

1. **If** $empty$ is $empty$ **Then**
2. **Print** "Quick Underflow!"
3. **Exit**
4. **Else**
5. $element = queue[front]$
6. $front = front + 1$
7. **Return** $element$
8. **End If**

End

6.4.2.1.2 Java implementation of the dequeue algorithm

```
public Object dequeue() {

      if( isEmpty() ){
            System.out.println("UNDERFLOW! Queue is Empty!");
            return null
      }

      // Remove and return the front element from the queue.
      Object element = queue[front];
      front++;

      return element;
}
```

6.4.2.1.3 How dequeuing elements from an array-implemented queue works

Example 6.6: Let's continue to manipulate our queue from **Example 6.2**. Let's perform two $dequeue$ operations on it. Each time we $dequeue$ an element, $front$ is **increased by 1**. Rear never changes in a dequeue operation. Figure 6-9 shows the state of the queue after each $dequeue$ operation.

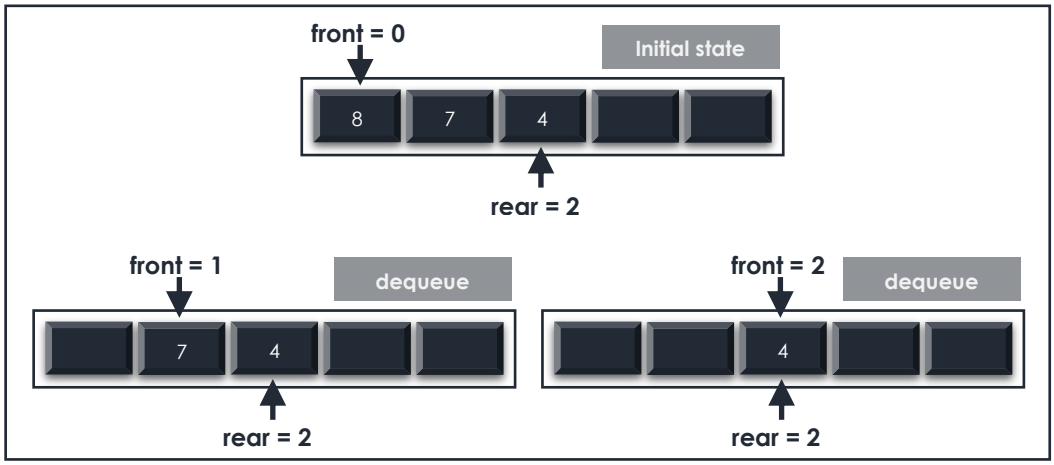


Figure 6-9. Simple representation of a (array) queue runtime with dequeue operation.

If we perform a third dequeue, $front$ will become 3. And we will not be able to do anymore dequeue operations until, at least one enqueue operation occurs.

6.4.2.1.4 Dequeuing a circular queue

Similar to the enqueuing operation of a circular queue, its dequeuing operation is almost identical to dequeuing a simple queue. The dissimilarity is how front is adjusted and the size of the queue is decreased by 1 when an element is dequeued. The $front$ is adjusted the same way as in enqueuing operation, i.e., $front = (front + 1)\ \%\ capacity$.

The following is the Java implementation of the dequeuing a circular queue.

```
public Object dequeue() {

      if( isEmpty() ){
            System.out.println("UNDERFLOW! Queue is Empty!");
            return null
      }

      // Remove and return the front element from the queue.
      Object element = queue[front];
      front = (front + 1)  % capacity;

      size--;

      return element;
}
```

6.4.2.2 Removing an element from a linked-list-based queue

As deletion is always done at the head of the linked list, the dequeue operation does not require an input value. We can use the following steps to delete a node from the queue.

1. Check whether ***queue*** is ***empty.***
2. If it is ***empty***, then throw an exception and exit.
3. If it is **not *empty***, then define a new object to copy the value of the ***element*** at the front position.
4. Then set ***successor*** of ***front*** to be the new ***front***, i.e., $front = front \rightarrow next$
5. Check whether the queue has become empty.
6. If ***queue*** is ***empty***, set ***rear*** to ***null***. Otherwise do nothing.
7. Decrease ***size*** of the queue by 1.
8. Return ***element***.

6.4.2.2.1 Dequeue algorithm for a linked-list-based queue

Algorithm Dequeue()
Input: None.
Output: Removed element is stored in ***element***, if ***queue*** is not ***empty***.
Data Structure: ***queue***, a singly linked list structure whose pointer to the ***head*** is known from ***front***, pointer to the first node, point to the last node is known as ***rear***, pointer to the last node and ***size*** is the number of elements in ***queue***.
Begin

1. **If** *queue* is *empty* **Then**
2. **Print** "Queue Underflow!"
3. **Exit**
4. **Else**
5. $element = front \rightarrow data$
6. $front = front \rightarrow next$
7. **If** *queue* **Is** *empty* **Then**
8. $rear = null$
9. **End if**
11. $size = size - 1$

12. **Return** *element*
13. **End if**

End

6.4.2.2.2 *Java implementation of the dequeue algorithm*

```
public Object dequeue() {

      if( isEmpty() ){
            System.out.println("UNDERFLOW! Queue is Empty!");
            return null;
      }

      // Remove and return the front element from the queue.
      Object element = front.data;
      front = front.next;

      if( isEmpty() ){
            rear = null;
      }
      size--;

      return element;
}
```

6.4.2.2.3 *How dequeuing elements out of a linked-list-implemented queue works*

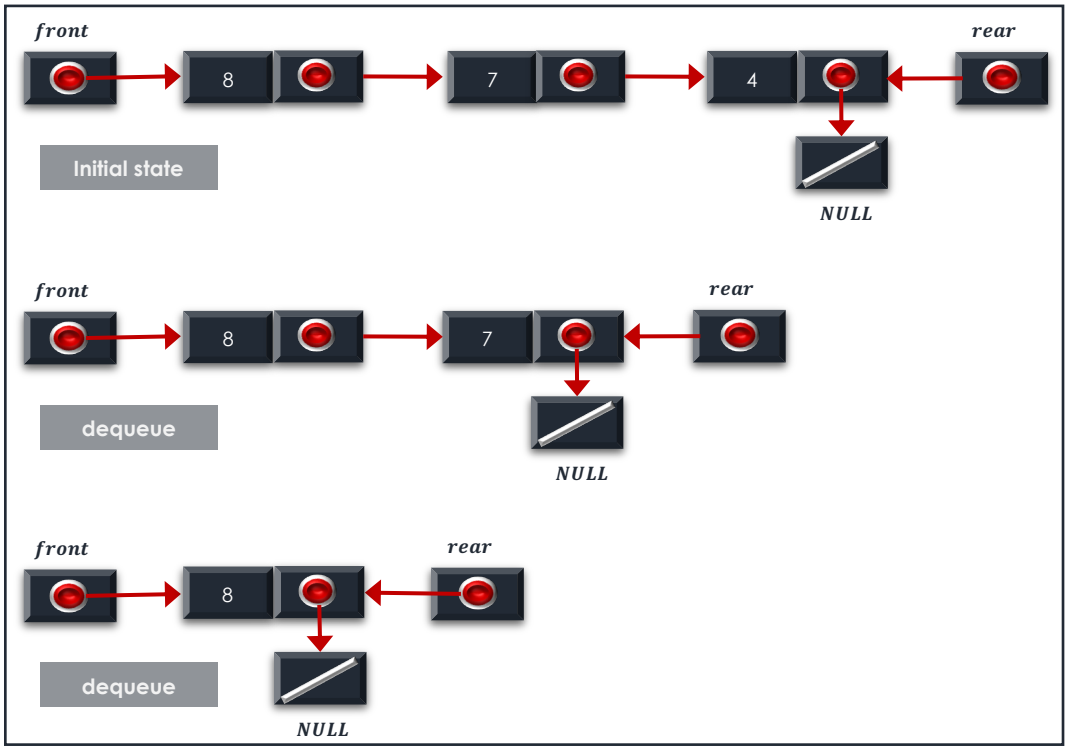


Figure 6-10. Simple representation of a (linked list) queue runtime with dequeue operation.

Example 6.7: Continuing with our queue **Example 6.4**, we will perform two $dequeue$ operations on the $queue$. Figure 6-10 shows the state of the queue after each dequeue operation.

6.4.2.2.4 Dequeuing a priority queue

Removing elements from a linked list priority queue works the same way as a standard linked list queue. The only variance is in priority queue we do not reset $rear$ since it is not defined.

6.4.3 Queue Status Checking Operations

As we have discussed, we need functionality of checking the status of the queue to be able to use the structure efficiently. In this section we will learn the secondary operations we need to support the primary queue functions. These operations do not alter the queue. They access the queue without transforming it.

6.4.3.1 Peek operation

The peek operation provides a quick look at the front position of the queue. It returns the element at $front$ without removing it from the queue.

6.4.3.1.1 Peeking into an array-based queue

Algorithm Peek()
Input: None.
Output: Element at the front position is stored in $\boldsymbol{element}$, if $\boldsymbol{queue}$ is not $\boldsymbol{empty}$.
Data Structure: $\boldsymbol{queue}$, an array structure with $\boldsymbol{front}$, pointer to the first element in the array.
Begin

1. **If** $queue$ is $empty$ **Then**
2. **Print** "Queue is Empty!"
3. **Exit**
4. **Else**
5. **Return** $queue[front]$

End

6.4.3.1.2 Java implementation of the peek operation for array-based queue

```
public int peek() {
      if( isEmpty() ){
            System.out.println( "Queue is Empty!" );
            return null;
      }
      // Return the element at the front of this queue.
      return queue[front];
}
```

6.4.3.1.3 Peeking into a linked-list-based queue

Algorithm Peek()

Input: None.
Output: Element at the front position is stored in ***element***, if ***queue*** is not ***empty***.
Data Structure: ***queue***, a singly linked list structure whose pointer to the ***head*** is known from ***front***, pointer to the first node in ***queue***.
Begin

1. **If** $queue$ is $empty$ **Then**
2. **Print** "Queue is Empty!"
3. **Exit**
4. **Else**
5. **Return** $front \rightarrow data$

End

6.4.3.1.4 Java implementation of the peek operation for linked-list-based queue

```
public int peek() {
      if( isEmpty() ){
            System.out.println( "Queue is Empty!" );
            return null;
      }
      // Return the element at the front of this queue.
      return front.data;
}
```

6.4.3.2 (Is) Empty Operation

The Is Empty functionality returns true only when there are zero elements in the queue.

6.4.3.2.1 Testing whether an array-based queue is empty

Algorithm IsEmpty()
Input: None.
Output: ***true***, if ***queue*** is ***empty***, otherwise return ***false***.
Data Structure: ***queue***, an array structure with ***front***, pointer to the first element in the array and ***rear***, pointer to the last element in the array.
Begin

1. **If** $front = -1$ **Or** $front > rear$ **Then**
2. [queue has no elements]
3. **Return** $true$
4. **Else**
5. **Return** $false$

End

6.4.3.2.2 Java implementation of the isEmpty operation for array-based queue

```
public boolean isEmpty() {
      if( front == -1 ||  front > rear )   {
            // Queue is Empty!
            return true;
      }
```

```
        return false
    }
```

6.4.3.2.3 How to determine whether a circular queue is empty

The nature of a circular queue necessitates checking for emptiness by testing the equality of $size$ to 0, i.e., **if**($size == 0$).

6.4.3.2.4 Testing whether a linked-list-based queue is empty

Algorithm IsEmpty()
Input: None.
Output: $\boldsymbol{true}$, if $\boldsymbol{queue}$ is $\boldsymbol{empty}$, otherwise return $\boldsymbol{false}$.
Data Structure: $\boldsymbol{queue}$, a singly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$, pointer to the first node in the linked list and pointer to the $\boldsymbol{tail}$ is known from $\boldsymbol{rear}$, pointer to the last node in the linked list.
Begin

1. **If** $front \rightarrow data = null$ **Or** $front = rear \rightarrow next$ **Then**
2. [queue has no elements]
3. **Return** $true$
4. **Else**
5. **Return** $false$

End

6.4.3.2.5 Java implementation of the isEmpty operation for linked-list-based queue

```
public boolean isEmpty() {
      if( front.data == null  || front == rear.next) ){
            // Queue is Empty!
            return true;
      }

      return false;
}
```

6.4.3.2.6 How to determine whether a priority queue is empty

Owing to the fact that a priority queue does not maintain a **rear**, the second condition is irrelevant. So, the only test parameter for a priority queue is, $\boldsymbol{if}(front.data == null)$.

6.4.3.3 (Is) Full Operation

The Is Full functionality is required only for queue with a fixed $capacity$. It returns $true$ only when the queue is filled to $capacity$.

6.4.3.3.1 Testing whether an array-based queue is full

Algorithm IsFull()
Input: None.

Output: $\boldsymbol{true}$, if $\boldsymbol{queue}$ is $\boldsymbol{full}$, otherwise return $\boldsymbol{false}$.
Data Structure: $\boldsymbol{queue}$, an array structure with $\boldsymbol{rear}$, pointer to the last element in the array and $\boldsymbol{capacity}$ is the maximum number of elements the array can hold.
Begin

1. **If** $rear = capacity - 1$ **Then**
2. [queue has is filled to maximum size]
3. **Return** $true$
4. **Else**
5. **Return** $false$

End

6.4.3.3.2 Java implementation of the isFull operation for array-based queue

```
public boolean isFull() {
      if( rear == capacity - 1 ) ){
            // Queue is Full!
            return true;
      }

      return false;
}
```

6.4.3.3.3 How to determine whether a circular queue is full

Due to the characteristics of a circular queue, the queue is checked for fullness by testing the equality of $size$ to $capacity$, i.e., $\boldsymbol{if}(\ size == capacity\ )$.

6.4.4 Other support Operations of a Queue

In addition to the functions we have mentioned, we can add more functionality to the queue. For example, we have mentioned that maintaining an instance variable to keep track of $size$ in a linked-list-based queue can improve the efficiency of computing the size from linear time to constant time. We can provide a functionality to return the size of the queue. Another example is, we may provide functionality for operations we often do with our queue such as resetting (emptying) the queue, displaying the content of the queue or searching through the queue to test whether an element exists in it.

6.4.4.1 Size operation

The size operation returns the number of current elements in the queue.

6.4.4.1.1 Sizing an array-based queue

Algorithm Size()
Input: None.
Output: $\boldsymbol{size}$, the number of elements currently in the array.
Data Structure: $\boldsymbol{queue}$, an array structure with $\boldsymbol{front}$, pointer to the first element in the array and $\boldsymbol{rear}$, pointer to the last element in the array. $\boldsymbol{size}$ is 0, when then queue is empty.
Begin

1. **If** $queue$ **is** $empty$ **Then**
2. **Return** $size$
3. **Else**
4. **Return** $(rear - front) + 1$
5. **End If**

End

6.4.4.1.2 Java implementation of the size operation for array-based queue

```
public int size()   {
      if( isEmpty() )
            return size;
      else
            return ( rear – front ) + 1;
}
```

6.4.4.1.3 Sizing a circular queue

Owing to the nature of a circular queue, the size() function simply returns the value of $size$ attribute of the queue.

6.4.4.1.4 Sizing a linked-list-based queue

Algorithm Size()
Input: None.
Output: $\boldsymbol{size}$, the number of elements currently in the linked list.
Data Structure: $\boldsymbol{queue}$, a singly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$, pointer to the first node in the linked list; and $\boldsymbol{size}$ is the number of elements in the linked list.
Begin

1. **Return** $size$

End

6.4.4.1.5 Java implementation of the size operation for linked-list-based queue

```
public int size() {
      return size;
}
```

6.4.4.2 Clear operation

The clear operation transforms the queue. It resets $front$ and $rear$ to their default value, -1 or $null$ for array-based queue and linked-list-based queue respectively.

6.4.4.2.1 Clearing an array-based queue

Algorithm Clear()
Input: None.
Output: None.
Data Structure: $\boldsymbol{queue}$, an array structure with $\boldsymbol{front}$, pointer to the first element in the array and $\boldsymbol{rear}$, pointer to the last element in the array.

Begin

1. $front = -1$
2. $rear = -1$

End

6.4.4.2.2 Java implementation of the clear operation for array-based queue

```
public void clear() {
        // Make the queue empty
        front = -1;
        front = -1;
}
```

6.4.4.2.3 Resetting a circular queue

The nature of a circular queue makes its clear() method also resets the $size$ attribute of the queue, i.e., $size = 0$.

6.4.4.2.4 Clearing a linked-list-based queue

Algorithm Clear()
Input: None.
Output: None.
Data Structure: ***queue***, a singly linked list structure whose pointer to the ***head*** is known from ***front***, pointer to the first node in the linked list; and ***size*** is the number of elements in the linked list.
Begin

1. $front = null$
2. $size = 0$

End

6.4.4.2.5 Java implementation of the clear operation for linked-list-based queue

```
public int clear() {
        // Make the queue empty
        front = null;
        size = 0;
}
```

6.4.4.3 Display operation

Display operation is essentially a traversal operation in which each element of the queue is printed, starting from front to back. This operation accesses the queue without modifying its content.

6.4.4.3.1 Printing the elements of an array-based queue

Algorithm DisplayElements()
Input: None.
Output: None.

Data Structure: ***queue***, an array structure with $\boldsymbol{front}$, pointer to the first element of the array and $\boldsymbol{rear}$, pointer to the last element in the array.
Begin

1. $tempFront = front$
2. **While** $queue$ **Is Not** $empty$ **Do**
3. **Print** element at the $front$
4. $front = front + 1$
5. **End While**
6. $front = frontTop$

End

6.4.4.3.2 Java implementation of the display operation for array-based queue

```
public void displayElements() {

      int tempFront = front

      // print each element of the stack
      System.out.print( "[ " );
      while( !isEmpty()  ){
            System.out.print( peek() + " " );
            front++;
      }
      System.out.println( "]" );

      front = tempFront;
}
```

6.4.4.3.3 Displaying elements of a circular queue

Due to the nature of a circular queue, printing its elements varies slightly from the above. As previously mentioned, the adjustment of $front$ is always $(front + 1)\ \%\ capacity$. Also, a local variable, $tempSize$, is initialised and set to the number of elements in the queue. During each iteration of the loop $size$ is decreased by 1. After the loop, $size$ is restored to its original value, i.e., $size = tempSize$.

```
      int tempSize = size();

      while(  !isEmpty()  ){
            System.out.print( peek() + " " );
            front = ( front + 1 )  % capacity;
            size--;
      }

      size = tempSize;
```

6.4.4.3.4 Printing the elements of a linked-list-based queue

Algorithm DisplayElements()
Input: None.
Output: None.

Data Structure: ***queue***, a singly linked list structure whose pointer to the ***head*** is known from ***front***, pointer to the first node in the linked list.
Begin

1. $tempFront = front$
2. **While** $queue$ **Is Not** $empty$ **Do**
3. **Print** element at the $front$
4. $front = front \rightarrow next$
5. **End While**
6. $front = tempFront$

End

6.4.4.3.5 Java implementation of the display operation for linked-list-based queue

```
public void displayElements() {

      Node tempFront = front

      // print each element of the stack
      System.out.print( "[ " );
      while( !isEmpty() )   {
            System.out.print( peek() + " " );
            front = front.next;
      }
      System.out.println( "]" );

      front = tempFront;
}
```

6.4.4.4 Search operation

Given a queue, to check whether it contains an element requires traversing the queue starting from ***front***, if the queue is not $empty$, until the element is found, or the end of the queue is reached.

6.4.4.4.1 Searching an array-based queue

The following steps are needed to search a non-empty array implemented queue.

1. Initialise a local variable, ***tempFront***, with the value of ***front***.
2. While queue is **not empty** and search ***element*** is not found at ***front***; repeat 3 and 4.
3. Increase ***front*** by 1 (for a circular queue, then do a modulo operation against $capacity$)
4. Initialise a local variable ***position***, to 1 if element is found at $front$, otherwise to the value of current $front$, i.e., the value of ***front*** at the end of the loop.
5. Restore ***front*** to its original value, i.e., set ***front*** to ***tempFront***.
6. Return ***position***. This will return -1, if the search element is not found or the position of the element in the queue.

Algorithm Search(element)
Input: ***element***, the value to look for in the queue.

Output: $\boldsymbol{position}$, the position of $element$ in the queue or -1 if $element$ is not in the queue or the queue is $empty$.
Data Structure: $\boldsymbol{queue}$, an array structure with $\boldsymbol{front}$, pointer to the last element in the array.
Begin

1. $tempFront = front$
2. $tempSize = size$
3. **While** $queue$ **Is Not** $empty$ **And** $queue[front] != element$ **Do**
4. $front = front + 1$
5. **End While**
6. **If** $queue[tempFront] = element$ **Then**
7. $position = 1$
8. **Else**
9. $position = front$
10. $front = tempFront$
11. **Return** $position$

End

6.4.4.4.2 Java implementation of the search operation for array-based queue

```
public int search( Object element ) {

      int position = -1

      if( !isEmpty() ){

            int tempFront = front;

            while( !isEmpty()  && peek()  != element  ){
                  front++;
            }

            if( position < 0 && queue[tempFront] == element )
                  position++;
            else if(  !isEmpty()  && peek() == element )
                  position = front;

            // Restore front to its original value.
            front = tempFront;
      }
      // Return the position of the element in the queue.
      return position;
}
```

6.4.4.4.3 Searching a circular queue

A circular queue is searched the same as a simple queue. However, a second local variable, $\boldsymbol{tempSize}$ is initialised and set to the current $\boldsymbol{size}$ of the queue. As we traverse the queue, we decrement $size$ by 1, whenever the search condition holds. As usual, front is adjusted against the modulo of the capacity. After the loop, we restore size to its original state, i.e., $size = tempSize$.

```
int tempSize = size();

while(   !isEmpty()  &&  peek() != element )  {
      front = ( front + 1 )  % capacity;
      size--;
}

size = tempSize;
```

6.4.4.4.4 Searching a linked-list-based queue

The following steps are needed to search a linked list implemented queue.

1. Initialise a local variable $\boldsymbol{position}$, set it to -1.
2. Test whether the queue is empty.
3. If it is empty go to step 13.
4. Initialise a local variable, $\boldsymbol{tempFront}$, set it $\boldsymbol{front}$.
5. While queue is **not empty** and search $\boldsymbol{element}$ is not found at $\boldsymbol{front}$; repeat steps 6 and 7.
6. Set $\boldsymbol{front}$ to the node that follows it.
7. Increment $\boldsymbol{position}$ by 1.
8. Check whether the element was found at the beginning of the queue.
9. If element is the first element of the queue, set $\boldsymbol{position}$ to 0.
10. Test to see if the end of queue was reached and element was not found.
11. If element is not found, reset $\boldsymbol{position}$ to -1.
12. Restore $\boldsymbol{front}$ to its original value, i.e., set $\boldsymbol{front}$ to $\boldsymbol{tempFront}$.
13. Return the $\boldsymbol{position}$. This will return -1, if the search element is not in the queue or if the queue is empty.

Algorithm Search(element)
Input: $\boldsymbol{element}$, the value to look for in the queue.
Output: $\boldsymbol{position}$, the position of $element$ in the queue or -1 if $element$ is not in the queue or the queue is $empty$.
Data Structure: $\boldsymbol{queue}$, a singly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$, pointer to the first node in the linked list.
Begin

1. $position = -1$
2. **If** $queue$ **Is Not** $empty$
3. $tempFront = front$
4. **While** $queue$ **Is Not** $empty$ **And** $front \rightarrow data \; != element$ **Do**
5. $front = front \rightarrow next$
6. $position = position + 1$
7. **End While**
8. **if** $queue$ **Is Not** $empty$ **And** $front \rightarrow data == element$ **Do**
9. $position = position + 1$
10. **Else**
11. **if** $queue$ **Is** $empty$ **Then**
12. $position = -1$
13. **End If**
14. **End If**

15. $front = tempFront$
16. **End if**
17. **Return** $position$

End

6.4.4.4.5 Java implementation of the search operation for linked-list-based queue

```
public int search( Object element ) {

      int position = -1;

      if( !isEmpty() )     {
            QueueNode tempFront = front;

            while(  !isEmpty() &&  peek() != element  ){
                  front = front.next;
                  position++;
            }
            if(  !isEmpty() &&  peek() == element  ){
                  // element is the first node of the queue
                  position++;
            }
            else   {
                  if(  isEmpty()  ){
                        // element is not found in the queue
                        position = -1;
                  }
            }

            // Restore front to its original value.
            front = tempFront;
      }
      // Return the position of the element in the queue.
      return position;
}
```

6.5 Complexity of Queue Operations

Queue has an efficient (constant time) implementation using cyclic arrays (circular queues) and singly linked lists. The time complexity for common queue operations and space complexity for queue are summarised in the Table 6-1.

Table 6-1. Complexity of queue operations

Operations	Array Queue	Linked Queue
Time Complexity		
enqueue	$O(1)$	$O(1)$
dequeue	$O(1)$	$O(1)$

search	$O(n)$	$O(n)$
peek	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$
full	$O(1)$	$O(1)$
Space complexity		
	$O(n)$	$O(n)$

6.6 More on Deque

We cannot conclude our discussion on queues without exploring deque in greater detail. As we have mentioned earlier, insertions and deletions operations are performed from both **front** and **rear**. Thus, we can say a deque is a generalised variant of the queue data structure.

As deque allows insertions and deletions from both ends, it can be used as both stack and queue.

- In deque, insertion and deletion can be done on a single end. The stack follows the LIFO principle which permits insertion and deletions of elements from a single point.
- In deque, insertion can be done at one end and deletion can be from another end. The queue follows FIFO principle which permits elements to be inserted at one end and removed from another end.

There are two types of deques:

- Input-restricted deque: In this type of deque, restriction is applied to insertion operations. So, insertion operation is only done from one end whereas deletion operation can be done from both ends.
- Output-restricted deque: In this type of deque, a constraint is placed on deletion operations. So, deletion operation is limited to one end while insertion operation is possible at both ends.

6.6.1 Operations on a Deque

All queue operations can be done on a deque. However, because of the nature of deque, the main queue operations are adapted as follows:

- Insert at **front**: adds an element to the front position, if deque is not full (only for a static deque).
- Delete from **front**: removes an element from the front position, if deque is not empty.
- Insert at **rear**: adds an element to the rear position, if deque is not fully (applicable only to a static deque).
- Delete from **rear**: removes an element from the rear position, if deque is not empty.

Similarly, we can perform the peek operation at both ends of the deque.

6.6.2 Memory Representation of a Deque

How a deque is represented in memory depends on its implementation. A deque can be implemented using a circular queue and a doubly linked list.

6.6.3 Implementation of a Deque using a Circular Queue

Operations of a deque that are common to a circular queue are executed exactly as in a circular queue. Let's look at the operations that are not intrinsic to the circular queue.

6.6.3.1 Enqueue at front

Inserting at the front of an empty deque works the same way as enqueuing a normal circular queue. The element will be inserted at the first empty space and both the front and rear will point to it. If we want to insert at the front point of the deque, we first have to a decrement of the *front*, i.e., move the front pointer one step backwards. Once the front pointer is set, we will insert the value. Because we are working with a circular queue, when we decrease front by 1, we will perform a modulo operation against capacity, i.e., *front = (front – 1) % capacity*. However, for Java, we will need to use *Math.floorMod()* method to perform obtain an accurate modulo for negative numbers.

Example 6.8: The following is Java implementation of the enqueuing an element at the front of a deque.

```
public void enqueueFront( Object element ) {
      // Add value as the front element in this queue.

      if( isFull() ){
            System.out.println( "DeQue is full!" );
            return;
      }

      if( isEmpty() ){
            front = ( front + 1 ) % capacity;
            rear = ( rear + 1 ) % capacity;
      }
      else{
            front = Math.floorMod( ( front() – 1 ), capacity () );
            //front = ( front - 1 ) % capacity
      }

      queue[front] = element;
      size++;
}
```

6.6.3.2 Dequeue at rear

Attempting to dequeue from the rear of an empty deque behaves the same way as attempting to dequeue the front of a standard circular queue. To remove an element

at the rear, first we take out the element and then do a decrement on rear, i.e., move rear one spot backwards. To adjust rear, we will need to decrease it by 1 and then perform a modulo operation against capacity, i.e., rear = (rear – 1) % capacity. Remember when implementing in Java we will use Math.floorMod() method to get accurate modulo when having negative dividend.

Example 6.9: The following is Java implementation of the dequeuing an element from the rear of a deque.

```
public Object dequeueRear() {
      // Add value as the front element in this queue.

      if( isEmpty() ){
            System.out.println( "DeQue is Empty!" );
            return;
      }

      Object element = pollRear();
      rear = Math.floorMod( ( rear() – 1 ), capacity () );
      // rear = ( rear - 1 ) % capacity;
      size--;

      return element;
}
```

6.6.3.3 *Peeking at rear*

Peeking at rear, simply return the element at the rear of the deque if the deque is not empty, i.e., $\boldsymbol{return}\ queue[rear]$.

6.6.3.4 *Processing a circular deque*

Example 6.10: Let's continue with the queue from **Example 6.2**. Assuming the structure is a deque. Let's add 10 at the front of deque. Then dequeue from the rear. And add 5 at the front again. Figure 6-11 shows the enqueuing and dequeuing operations of a deque.

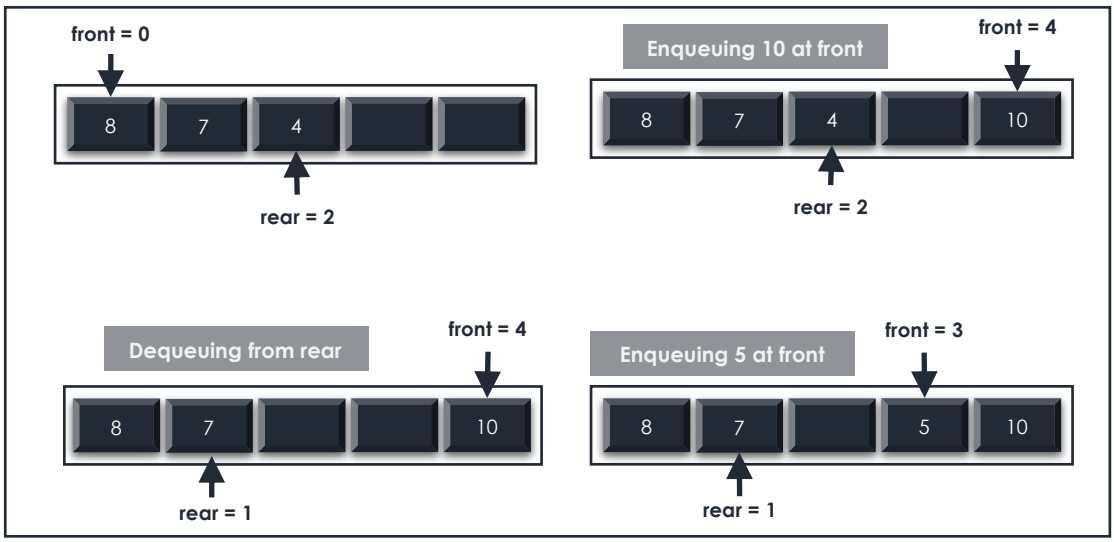


Figure 6-11. Simple representation of a deque enqueue/dequeue operations.

6.6.4 Implementation of a Deque using a Doubly Linked List

A deque is implemented as a linked list using doubly linked list because due to the nature of a deque elements are processed in both directions. Like with regular queues, we keep track of two pointers, **front** and **rear**.

We **enqueue(push)** an item at the rear or the front end of the deque and **dequeue(pop)** an item from both rear and front end.

6.6.4.1 Insertion at front

To insert an element at the front of deque we do following steps:

1. Create a new doubly linked list node, with the element to be inserted as its value and set both its links **rear** and **front** to **null**.
2. Check if **deque** is empty.
3. If **empty**, then set both front and rear to the new node.
4. If **not empty**, then do steps 5 to 8
5. Set successor of the new element to **front**.
6. Set the predecessor of **front** to the new element.
7. Make the new element the new **front**.
8. Increase size by 1.

Algorithm EnqueueFront(element)
Input: $\boldsymbol{element}$, the item to add to the front of deque.
Output: nothing, but deque will be modified to with the element as its new from.
Data Structure: $\boldsymbol{deque}$, a doubly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$, pointer to the first node in the linked list and $\boldsymbol{size}$ is the number of elements in $deque$.
Begin

1. $Node\ newNode\ =\ new\ Node()$
2. $newNode \rightarrow data\ =\ null$
3. $newNode \rightarrow next\ =\ null$
4. **If** $deque$ **Is** $empty$ **Then**
5. $\quad rear\ =\ front\ =\ newNode$
6. **Else**
7. $\quad newNode \rightarrow next\ =\ front$
8. $\quad front \rightarrow prev\ =\ newNode$
9. $\quad front\ =\ newNode$
10. $\quad size\ =\ size\ +\ 1$
11. **End If**

End

6.6.4.2 Insertion at rear

To insert an element at the front of deque we do following steps:

1. Create a new doubly linked list node, with the element to be inserted as its value and set both its links to **rear** and **front** to **null**.

2. Check if **deque** is empty.
3. If **empty**, then set both front and rear to the new node.
4. If **not empty**, then do steps 5 to 8
5. Set predecessor of the new element to **rear**.
6. Set the successor of **rear** to the new element.
7. Make the new element the new **rear**.
8. Increase size by 1.

Algorithm EnqueueRear(element)
Input: $\boldsymbol{element}$, the item to add to the rear of the deque.
Output: nothing, but deque will be modified to with the element as its new from.
Data Structure: $\boldsymbol{deque}$, a doubly linked list structure whose pointer to the $\boldsymbol{tail}$ is known from $\boldsymbol{rear}$, pointer to the last node in the linked list and $\boldsymbol{size}$ is the number of elements in $deque$.
Begin

1. $Node\ newNode\ =\ new\ Node()$
2. $newNode \rightarrow data\ =\ null$
3. $newNode \rightarrow next\ =\ null$
4. **If** $deque$ **Is** $empty$ **Then**
5. $front\ =\ rear\ =\ newNode$
6. **Else**
7. $newNode \rightarrow prev\ =\ rear$
8. $rear \rightarrow next\ =\ newNode$
9. $rear\ =\ newNode$
10. $size\ =\ size\ +\ 1$
11. **End If**

End

6.6.4.3 Deletion from front

To delete an element from front of deque we do following steps:

1. Check if **deque** is **empty**.
2. If empty, then throw and underflow error and exit.
3. If not empty, then do steps 4 to 10.
4. Create a new doubly linked list node and initialise with **front**.
5. Make the successor of element at front the new **front**.
6. Check if **deque** has become **empty**.
7. If empty, then set **rear** to **null**.
8. If **not empty**, then set the predecessor of **front** to **null**.
9. Decrease size by 1.
10. Return the value of the deleted node.

Algorithm DequeueFront()
Input: none.
Output: $\boldsymbol{deletedNode \rightarrow data}$, the value of the deleted node.
Data Structure: $\boldsymbol{deque}$, a doubly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$ (pointer to the first node in the linked list), pointer to the $\boldsymbol{tail}$ is known

from $\boldsymbol{rear}$ (pointer to the last node in the linked list) and $\boldsymbol{size}$ is the number of elements in $deque$.

Begin

1. **If** $deque$ **Is** $empty$ **Then**
2. **Print** "Underflow"
3. **Exit**
4. **Else**
5. $deletedNode = front$
6. $front = front \rightarrow next$
7. **If** $deque$ **Is** $empty$ **Then**
8. $rear = null$
9. **Else**
10. $front \rightarrow prev = null$
11. **End If**
12. $Size = size - 1$
13. **Return** $deletedNode \rightarrow data$
14. **End If**

End

6.6.4.4 Deletion from rear

To delete an element from the rear of the deque we do following steps:

1. Check if **deque** is **empty**.
2. If **empty**, then throw and underflow error and exit.
3. If **not empty**, then do steps 4 to 10.
4. Create a new doubly linked list node and initialise with **rear**.
5. Make the predecessor of element at rear the new **rear**.
6. Check if **deque** has become **empty**.
7. If **empty**, then set **front** to **null**.
8. If **not empty**, then set the successor of **rear** to **null**.
9. Decrease size by 1.
10. Return the value of the deleted node.

Algorithm DequeueRear()
Input: none.
Output: $\boldsymbol{deletedNode \rightarrow data}$, the value of the deleted node.
Data Structure: $\boldsymbol{deque}$, a doubly linked list structure whose pointer to the $\boldsymbol{head}$ is known from $\boldsymbol{front}$ (pointer to the first node in the linked list), pointer to the $\boldsymbol{tail}$ is known from $\boldsymbol{rear}$ (pointer to the last node in the linked list) and $\boldsymbol{size}$ is the number of elements in $deque$.

Begin

1. **If** $deque$ **Is** $empty$ **Then**
2. **Print** "Underflow"
3. **Exit**
4. **Else**
5. $deletedNode = rear$
6. $rear = rear \rightarrow prev$

```
7.      If deque Is empty Then
8.              front = null
9.      Else
10.             rear → next = NULL
11.     End If
12.     size = size – 1
13.     Return deletedNode → data
14. End If

End
```

6.7 Real Life Examples

What are real life applications of the queue data structure?

- Print spooling.
- CPU scheduling and disk scheduling. Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Data transfer between two processes in the asynchronous manner. In this application, the queue is used for synchronization. For example – IO buffers, pipes, file IO, etc.
- Handling interruptions in real-time systems
- The call centre phone systems also use the queue structure. It is used to hold the customer calls in order until a representative is free.
- Traffic System. In computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set. Converting decimal numbers into binary numbers.
- Performing **Breadth First Search** (BFS). BFS is an algorithm in data structure that traverses and searches a graph.

6.8 Stack vs Queues

Stacks and queues are similar data structures. There are two major similarities between the stack and queue:

- They are both linear data structure, which means that their elements are stored sequentially and accessed in a single run.
- They can both be static or dynamic.
    - The static implementation of the stack and queue can be done with the help of arrays. This means their capacity (maximum number of elements they can hold) is bounded at run time.
    - The dynamic implementation of the stack and queue can be done with the help of a linked list. This means they can grow and shrink according to the requirements at the run-time.

The following table summarises and highlights the differences between the array-based stack and array-based queue.

Basis for Comparison	Array Stack	Simple Queue
Principle	LIFO (Last-In First-Out) or FILO (First-In Last-Out)	FIFO (First-In First-Out) or LILO (Last-In Last-Out)
Structure	It only uses one end, called **top**, for both insertions and deletions of elements.	It uses two ends, one end is called **rear,** for insertions and another end, called **front,** for deletions.
Number of pointers used	It has one pointer, **top**. The top references the last inserted or the topmost element of the stack.	It has two pointers, **front** and **rear**. The front pointer references the first element, whereas the rear pointer references the last element in a queue.
Main operations performed	Push and pop.	Enqueue and dequeue
Checking for empty condition	$\boldsymbol{top} == -1$ (same $as\ \boldsymbol{top} < 0$), which means that the stack is empty.	$\boldsymbol{front} == -1$ OR $\boldsymbol{front} == \boldsymbol{rear} + 1$ (same as $\boldsymbol{front} > \boldsymbol{rear}$), which means that the queue is empty.
Checking for full condition	**top** == **capacity** – 1 (same as $(\boldsymbol{top} + 1) == \boldsymbol{capacity}$), this condition implies that the stack is full.	$\boldsymbol{rear} = \boldsymbol{capacity} - 1$ (same as $(\boldsymbol{rear} + 1) == \boldsymbol{capacity}$), this condition implies that the stack is full.
Variants	Double stacks	Circular queue, priority queue and double-ended queue.
Implementation	Simpler	Comparatively more complex than stack
Visualisation	Typically visualized as vertical.	Typically visualized as horizontal.

6.9 Summary

- A queue is collection of similar data items in which insertion and deletion operations are performed based on FIFO principle.
- A queue is an abstract data type, and the implemented queue ADT becomes a queue data structure.
- The queue structure is characterised by four major operations: create the collection, insert an element, delete an element and check whether the collection is empty.
- A major disadvantage of the array-based queue is spaces freed from dequeuing operations remain unused.
- Circular queues bypass this drawback. However, the disadvantage of having a fixed capacity remains. Thus, at any point of time, the array is either filled or not. This means memory is either not enough or is being wasted.
- Linked list implementation solves this problem. However, the drawback of the linked list implementation is that it needs to store several addresses (in pointer

fields of nodes). If the data type of the elements in the queue is small compared to pointer, then this implementation is not memory efficient.

- The table that follows compares the array and linked list implementations of the queue ADT.

Array Implemented Queue	Linked List Implemented Queue
They are generally static.	They are dynamic.
They have a fixed $\boldsymbol{capacity}$, which is the maximum number of elements the queue can hold.	Have "unlimited" $\boldsymbol{capacity}$.
$\boldsymbol{front}$ is the index of element at the front of the quest (first element in the array).	$\boldsymbol{front}$ is the node at the **start** of the queue (head of the linked list).
$\boldsymbol{rear}$ is the index of element at the end of the quest (last element in the array).	$\boldsymbol{rear}$ is the node at the **end** of the queue (tail of the linked list).
The $\boldsymbol{size}$ of the queue is usually $\boldsymbol{rear - front + 1}$ and $\boldsymbol{size \leq capacity}$.	The $\boldsymbol{size}$ of the queue is usually the number of $\boldsymbol{enqueues}$ operations done on the queue minus the number of $\boldsymbol{dequeues}$ operations done. Hence, $\boldsymbol{size}$ is $\boldsymbol{increased\ by\ 1}$ with every enqueue operation and $\boldsymbol{decreased\ by\ 1}$ with every dequeue operation.

- The following tables summarise **statuses** of an array implemented queue with $\boldsymbol{capacity}$ **N** based on the values of the $\boldsymbol{front}$ and $\boldsymbol{rear}$:

Values	Meaning
$\boldsymbol{front = -1\ OR\ front > rear}$	Queue is $\boldsymbol{empty}$. When a dequeue operation is attempted on an empty queue, the queue goes into $\boldsymbol{underflow}$.
$\boldsymbol{rear = 0}$	Queue has a single element.
$\boldsymbol{rear = N - 1}$	The queue is $\boldsymbol{full}$.
$\boldsymbol{rear = N}$	The queue is in $\boldsymbol{overflow}$, this happens when an enqueue is attempted on an already $\boldsymbol{full}$ queue.

- For a linked list implemented queue, when queue is empty, $\boldsymbol{front}$ is $\boldsymbol{null}$.
- In Java, Queue is represented as an interface, **java.util.Queue** that falls under the **Collection Framework**. Most frequently used Java Queue implementation classes are **LinkedList**, **PriorityQueue** and **BlockingQueue**. **AbstractQueue** provides a skeletal implementation of the Queue interface to reduce the effort in implementing Queue. Deque is represented as an interface, **java.utiil.Deque** and it is implemented by **ArrayDeque** and **LinkedList**.
- Deque is a generalised version of a queue. It allows insertion at front and rear and also permits deletion from front and rear. This property of a deque makes it possible to be as both a stack and a queue.
- There are two types of deques:
    1. Input-restricted queue: In this queue, elements can be removed from both ends of the queue but can only be inserted at one end.

2. Output-redistricted queue: In this queue, elements can be inserted at both ends of the queue but can only be removed from one end.

6.9.1 What Comes Later

- In later units, we will look at a data structure called heap and how it can be used to implement a priority queue.
- In the next unit, we will revisit stacks and look at the concept of recursion. We will also look at some of their applications including how they are used to keep track of system function calls.

6.10 Exercises

1. If the elements "A", "B", "C" and "D" are placed in a queue and are deleted one at a time, what is the order of removal?

2. Five (5) items: 1, 2, 3, 4, and 5 are pushed on top of a stack, one after the other starting with 1. Fours item were popped out of the stack, and each element was inserted in a queue as soon as it was deleted. Two elements are deleted from the queue and pushed back on top of the stack. What element is at top of the stack?

3. Given the following empty (output-restricted) deque (circular queue implementation). In this implementation of deque, when a queue becomes empty as a result of a dequeuing operation, the deque is reset.

If the following sequence of operations is performed:

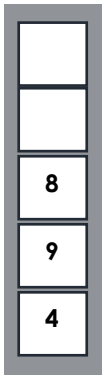
```
 I.   enqueueFront('A');
 II.  enqueueRear('B');
 III. enqueueRear('C');
 IV.  enqueueRear('E');
 V.   dequeueRear();
 VI.  dequeueRear();
 VII. dequeueRear();
 VIII. dequeueRear();
 IX.  enqueueFront('F');
 X.   enqueueFront('G');
 XI.  dequeueRear();
 XII. enqueueRear('D');
```

(a) With the aid of pictorial representation of the deque, show after each operation the content of the deque and the positions of $front$ and $rear$.

(b) What will happen if we call a $dequeueFront()$?

(c) From the operations in (a) above, determine what is the principle governing the deque, in terms of the ends the deletion and insertion operations are permitted?

4. Given the following static stack, containing integer elements 4, 9, 8,



Move the elements, one at a time, from this stack to a new static queue of the same capacity.

(a) Show the queue and its content.
(b) What are the elements at FRONT and REAR of the queue?
(c) What is the size and capacity of the queue?

5. Suppose that an intermixed sequence of enqueue and dequeue operations is performed on a queue. The enqueues adds the integers 1 through 10 in order; the pops print out the deleted value. Consider the following sequence *[5, 4, 3, 2, 1, 6, 7, 8, 9, 10]* and state whether it could occur or not. If the answer is yes, indicate the sequence of enqueues and dequeues needed. If the answer is no, explain why.

6. Given a static queue with capacity of 3. Assuming three (3) enqueue operations were performed on it in sequence, and then two (2) dequeue operations were done. Now, what will happen to this queue when another enqueue operation is executed, if the queue is:
(a) a standard linear queue?
(b) a circular queue?

7. The following characters are enqueued into a newly created array-based queue in the order they appear. Dashes will cause a dequeue. The underlying array started with a capacity of one. And it doubles in size when the enqueued element will fill the array and halves its size when a dequeue will leave it one- quarter full.
(a) A B C D – D E F – – G H I – – – –
(b) A B C D – E – F G H I

8. The following operations are applied to an initially empty queue of integers:

```
 I.   enqueue(6);
II.   enqueue(5);
III.  enqueue(8);
```

```
 IV.   enqueue(10);
  V.   enqueue(11);
 VI.   dequeue();
VII.   dequeue();
VIII.  enqueue(14);
```

(a) What is the FRONT and what is the REAR of the queue?
(b) What is the size of the queue?
(c) Given this queue,
    (i) what will be the element returned if a peek operation is performed on the queue?
    (ii) What happens to the size of the queue after the peel operation?

9. Given an array of *N* disks of distinct sizes, the size of the largest disk is also *N*, this means all disks are distinct integers in the range 1 to *N*. You are supposed to build a tower with the larger disk placed before the smaller disks. This means the larger disks go to the bottom of the tower and the smaller disks go to the top. The order to build the tower is as follows:
    - You cannot put a new disk on top of the tower until all disks larger than it are placed.
    - You can place disks that cannot yet go on the tower in a temporary placeholder. The order you place them in the placeholder is the order you take them out.

    (a) Identify the data structures that you can use for the tower and for the placeholder.
    (b) Assuming you are given 5 disks in the order `3 5 1 2 4`. Your tower and placeholder are declared as the relevant data structure(s) as identified in (a). You can make use of a disk variable to save disks you take out of the placeholder before you use them in the next operation.
        (i) Show the operations you will need to perform on both your tower and placeholder to complete your tower.
        (ii) Show your placeholder when you receive all three disks.

10. Mr. Henry is given a rectangular box, in which he can put blocks of the same width into it. The box has the same width as the blocks, so only one block can be inserted into or taken out of the box at any given time. Mr. Henry can put a block into the box at only one end and can remove a block only from the same end. With the aid of only one extra box, help Mr. Henry organise the blocks within the main box in such a way that the least recent ball placed in the box would be the first to be removed, and the most recently placed block would be the last to be removed. However, you can only:
    - Add balls in **linear** time.
    - Remove balls in **constant** time.

    (a) What data structure or abstract data type could be used to represent the box?
    (b) What data structure or abstract data type did you help Mr. Henry implement?

(c) How can you accomplish this task?
(d) Assuming Mr. Henry is given blocks that are labelled: 'A', 'B', 'C', 'D'. Using the algorithm in (c) above, show how the blocks can be put into the box and then removed.

11. Mohammed is given a cylindrical tube, in which he can put balls of the same radius into it. The tube has the same radius as the balls, so only one ball can be inserted into or taken out of the tube at any given time. Mr. Bali can put a ball into the tube at only one end and can remove a ball only from the opposite end. With the aid of one extra tube only, help Mr. Bali organise the balls within the main tube in such a way that the most recent ball placed in the tube would be the first to be removed, and the least recently placed ball would be the last to be removed. However, you can only:
    - Add balls in **constant** time.
    - Remove balls in **linear** time.

    (a) What data structure or abstract data type could be used to represent the tube?
    (b) What data structure or abstract data type did you help Mohammed implement?
    (c) How can you accomplish this task?
    (d) Assuming the balls are labelled: '1', '2', '3', 'D'. Using the algorithm in (c) above, show how the balls can be put into the tube and then removed.

12. What does the following Java code fragment do to the Stack *stack*?.

```
Queue queue = new Queue();
while ( !stack.isEmpty() ) {
      queue.enqueue( stack.pop() );
}
while ( !queue.isEmpty() )  {
      stack.push( queue.dequeue() );
}
```

13. Following is a Java like pseudocode of a function that takes a Queue *queue* as an argument and uses a Stack *stack* to do processing.

```
void mystery(Queue queue) {

      // assumes this creates an empty stack.
      Stack stack;

      // execute while queue is not empty.
      while( !queue.isEmpty() )   {
            // dequeue an element from queue and push it
            onto stack.
            stack.push( queue.dequeue() );
      }

      // execute while stack is not empty.
      while( !stack.isEmpty() )   {
```

```
                    // pop an element from stack and enqueue it in
                    queue.
                    queue.enqueue( stack.pop() );
                }
            }
```

What does the above function do in general?

14. What does the following Python code fragment do for positive integers? And will it print when *n = 7*?

```
queue = Queue()
queue.enqueue( 0 )
queue.enqueue( 1 )
for i in range( n ):
      a = queue.dequeue()
      print( f"{a}\n" )
      b = queue.peek()
      queue.enqueue( a + b )
```

15. Let ***queue*** denote a queue containing 8 elements and ***stack*** be an empty stack. **front(queue)** returns the element at the front of **queue** without removing it from **queue**. Similarly, **top(stack)** returns the element at the top of **stack** without removing it from **stack**. Consider the following algorithm snippet.

I. **While** $queue$ **Is Not** $empty$ **Do**
II. **If** $stack$ **Is** $empty$ **Or** $top(stack) \leq front(queue)$ **Then**
III. $element = dequeue\ (queue)$
IV. $push(stack, element)$
V. **Else**
VI. $element = dequeue\ (queue)$
VII. $enqueue(queue, element)$
VIII. **End If**
IX. **End While**

(a) What does the algorithm do?
(b) What is the maximum possible number of iterations of the while loop in the algorithm?
(c) Let ***queue*** denote a queue containing 8 elements and ***stack*** be an empty stack **front(queue)**.

16. Write a function that gets an integer, *n* from standard input, and then uses a queue to generate and display all binary numbers with decimal values from 1 to *n*.

17. Design and implement an algorithm that takes in two integer, *n* and *k*, as input. And then generate a queue with n elements. Next, it finds and deletes every k^{th} element. And finally display the elements in the order they were deleted.

18. Implement the algorithms of the deque operations in Section: 6.6.3.

19. A priority queue can be implemented with arrays. The priorities of the elements can be represented as one dimension while the values of the elements can be represented as a second dimension.
    (a) Write an algorithm or pseudocode for the main operations.
    (b) Implement your operations in (a).

20. Consider the situation where a stack is the only data structure you have available. Assume you can only use the same stack as the front and rear of the queue. If you make dequeue operation costly,
    (a) How many stacks do you need to implement a queue?
    (b) Design and implement the queue?
    (c) What is the complexity of enqueuing into and dequeuing from this queue?

21. Consider the situation where a queue is the only data structure you have available. Assume you can only use one queue as the top stack. If you make pop operation costly. If you make push operation costly,
    (a) How many queues do you need to implement a stack?
    (b) Design and implement the stack?
    (c) What is the complexity of pushing into and popping from this stack?