CHAPTER FIVE
5. DATA STRUCTURES AND APPLICATIONS: QUEUES

5.1. Introduction

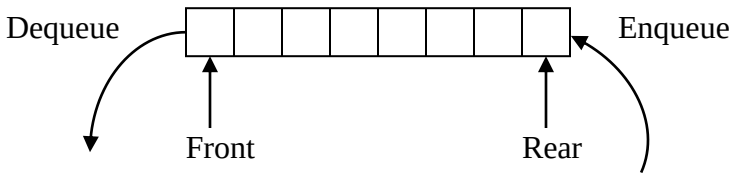
Like the stack, the queue is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an enqueue operation) and removed from the front (called a dequeue operation). Queues operate like standing in line at a movie theater ticket counter. If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. A queue is a "FIFO" list, which stands for "First-In, First-Out."

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the *front* of the queue, and element insertion is restricted to the end of the sequence, which is called the *rear* of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.

A queue is an ordered collection of items where an item is inserted at one end and usually called REAR or TAIL, and an item is removed from the other end, usually called FRONT or HEAD of the queue. So, a queue is a dynamic list with limited across to elements. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time.

A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed.

In a multi-processing system, for example, when jobs are submitted to a printer, we expect the least recent or most senior job to be printed first. This order is not only fair, but it also is required to guarantee that the first job does not wait forever. Thus you can expect to find printer queues on all large systems.



5.2. Fundamental operations on queues

The queue abstract data type (ADT) supports the following two fundamental operations:

- ***Enqueue*** (el): Insert element el at the rear of the queue.
- ***Dequeue*** ( ): Remove and return from the queue the object at the front; an error occurs if the queue is empty.

Additionally, similar to the case with the Stack ADT, the queue ADT includes the following supporting methods:

- ***Size*** ( ): Return the number of objects in the queue.
- ***Empty*** ( ): Return true if the queue is empty and false otherwise.
- ***Front*** ( ): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

A series of enqueue and dequeue operations is shown in the following figure. This time, unlike for stacks, the changes have to be monitored both at the beginning of the queue and at the end. The elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue.

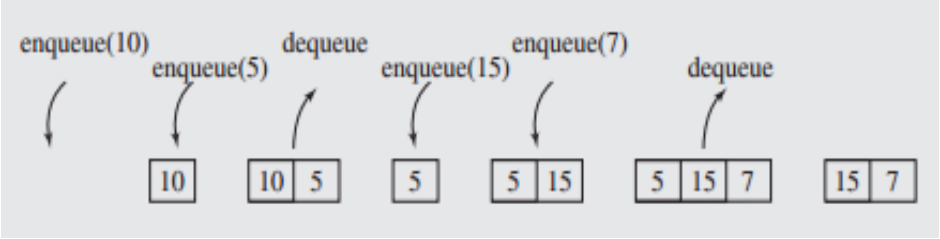


Figure 5.1: A series of operations executed on a queue.

The following table shows a series of queue operations and their effects on an initially empty queue, Q, of integers.

Operation	***Output***	*front ← Q ← rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()

5.3. Implementation of queues

5.3.1. Array implementation

One possible queue implementation is an array, although this may not be the best choice. Elements are added to the end of the queue, but they may be removed from its beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new elements, whereby the end of the queue may occur at the beginning of the array.

i. Simple array implementation of enqueue and dequeue

The analysis for the simple array based queue is stated here after.

Consider the following array: int Num[MAX_SIZE];

We need to have two integer variables that tell the index of the front element and the index of the rear (last) element. We also need an integer variable that tells the total number of data in the queue.

```
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

- To enqueue data to the queue
  - ✓ check if there is space in the queue

REAR < MAX_SIZE - 1 ?

Yes:
- Increment REAR
- Store the data in Num[REAR]
- Increment QUEUESIZE
- FRONT = = -1?
  - Yes: - Increment FRONT

No: - Queue Overflow

- To dequeue data from the queue
  - ✓ check if there is data in the queue

QUEUESIZE > 0 ?

Yes:
- Copy the data in Num[FRONT]
- Increment FRONT
- Decrement QUEUESIZE

No: - Queue Underflow

The implementation for these operations look like the following:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;
```

```
void enqueue(int x)
 {
    if(REAR < MAX_SIZE - 1)
     {
         REAR++;
         Num[REAR] = x;
         QUEUESIZE ++;
         if(FRONT = = -1)
           FRONT++;
     }
     else
        cout<<"Queue Overflow";
     }
```

```
int dequeue ( )
{
  int x;
  if(QUEUESIZE > 0)
   {
    x = Num[FRONT];
    FRONT++;
    QUEUESIZE - -;
    return(x);
   }
  else
   cout<<"Queue Underflow";
  }
```

ii. Circular array implementation of enqueue and dequeue

A problem with simple arrays is we run out of space even if the queue never reaches the size of the array. Thus, simulated circular arrays (in which freed spaces are re-used to store data) can be used to solve this problem. Whenever front or rear gets to the end of the array, it is wrapped around to the beginning.

The queue is full if the first element immediately precedes the last element in the counterclockwise direction. However, because a circular array is implemented with a "normal" array, the queue is full if either the first element is in the first cell & the last element is in the last cell (Figure 5.2a) or if the first element is right after the last (Figure 5.2b). Similarly, enqueue ( ) and dequeue ( ) have to consider the possibility of wrapping around the array when adding or removing elements. For example, enqueue ( ) can be viewed as operating on a circular array (Figure 5.2c), but in reality, it is operating on a one-dimensional array. Therefore, if the last element is in the last cell and if any cells are available at the beginning of the array, a new element is placed there (Figure 5.2d). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 5.2e). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 5.2f).

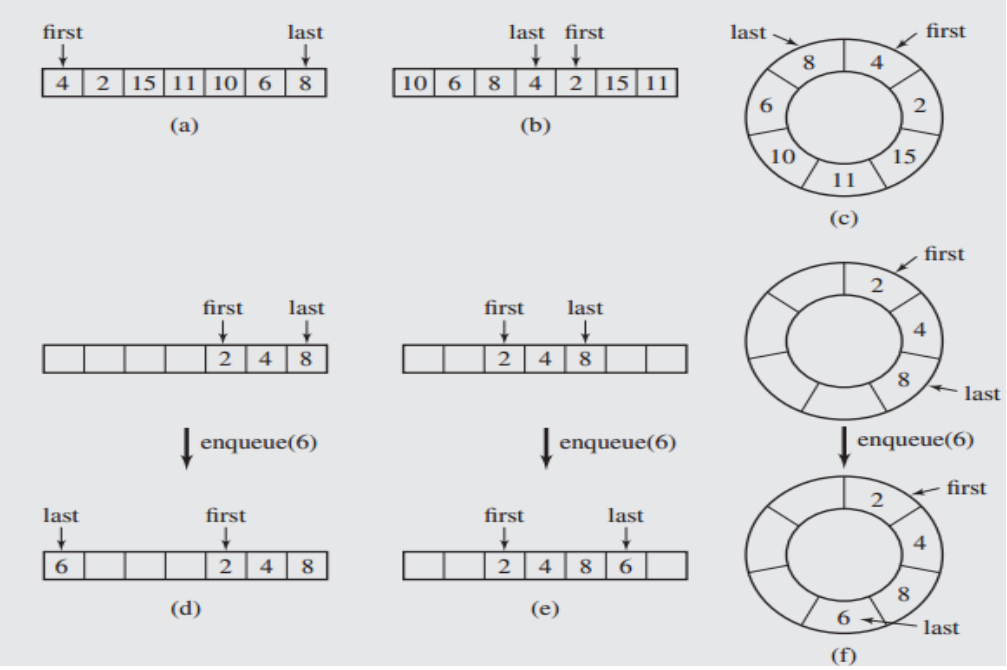


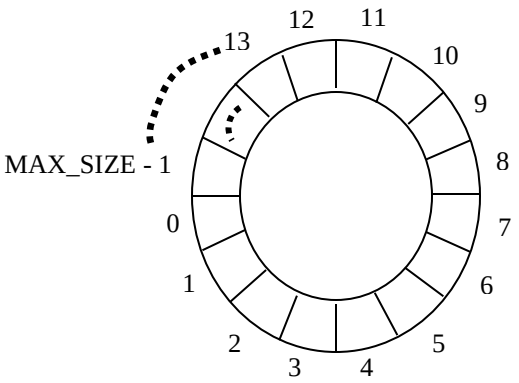
Figure 5.2: (a–b) Two possible configurations in an array implementation of a queue when the queue is full. (c) The same queue viewed as a circular array. (f) Enqueuing number 6 to a queue storing 2, 4, and 8. (d – e) The same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle.

Example: Consider a queue with MAX_SIZE = 4

The following table shows the same possible operations performed on both simple array based and circular array based queue.

Operation	Simple array							Circular array						
	Content of the array				Content of the Queue	QUEUE SIZE	Message	Content of the array				Content of the queue	QUEUESIZE	Message
Enqueue(B)	B				B	1		B				B	1	
Enqueue(C)	B	C			BC	2		B	C			BC	2	
Dequeue( )		C			C	1			C			C	1	
Enqueue(G)		C	G		CG	2			C	G		CG	2	
Enqueue (F)		C	G	F	CGF	3			C	G	F	CGF	3	
Dequeue( )			G	F	GF	2				G	F	GF	2	
Enqueue(A)			G	F	GF	2	Overflow	A		G	F	GFA	3	
Enqueue(D)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	
Enqueue(C)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	Overflow
Dequeue( )				F	F	1		A	D		F	FAD	3	
Enqueue(H)				F	F	1	Overflow	A	D	H	F	FADH	4	
Dequeue ( )					Empty	0		A	D	H		ADH	3	
Dequeue( )					Empty	0	Underflow		D	H		DH	2	
Dequeue( )					Empty	0	Underflow			H		H	1	
Dequeue( )					Empty	0	Underflow					Empty	0	
Dequeue( )					Empty	0	Underflow					Empty	0	Underflow

The circular array implementation of a queue with MAX_SIZE can be simulated as follows:



The analysis for the circular array based queue is stated in the following manner.

Consider the following array: int Num[MAX_SIZE];

We need to have two integer variables that tell the index of the front element and the index of the rear (last) element. We also need an integer variable that tells the total number of data in the queue.

```
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

- To enqueue data to the queue
  - ✓ check if there is space in the queue

```
QUEUESIZE < MAX_SIZE?
  Yes: - Increment REAR
            REAR = = MAX_SIZE?
                  Yes:  REAR = 0
       - Store the data in Num[REAR]
       - Increment QUEUESIZE
         FRONT = = -1?
            Yes: - Increment FRONT
  No:  - Queue Overflow
```

- To dequeue data from the queue
  - ✓ check if there is data in the queue

```
QUEUESIZE > 0 ?
  Yes: - Copy the data in Num[FRONT]
       - Increment FRONT
            FRONT = = MAX_SIZE?
                  Yes:  FRONT = 0
       - Decrement QUEUESIZE
  No:  - Queue Underflow
```

The implementation for these operations look like the following:

```
const int MAX_SIZE=100;
int FRONT = -1, REAR = -1;
int QUEUESIZE = 0;
```

```
void enqueue(int x)
    {
      if(QUEUESIZE<MAX_SIZE)
        {
          REAR + +;
          if(REAR = = MAX_SIZE)
            REAR = 0;
          Num[REAR] = x;
          QUEUESIZE + +;
          if(FRONT = = -1)
            FRONT + +;
        }
      else
        cout<<"Queue Overflow";
    }
```

```
int dequeue()
    {
      int x;
      if(QUEUESIZE > 0)
       {
        x = Num[FRONT];
        FRONT + +;
        if(FRONT = = MAX_SIZE)
          FRONT = 0;
        QUEUESIZE - -;
        return (x);
        }
      else
       cout<<"Queue Underflow";
     }
```

5.3.2. Linked list implementation

We can efficiently implement the queue ADT using linked list. For efficiency reasons, we choose the front of the queue to be at the head of the list, and the rear of the queue to be at the tail of the list. In this way, we remove from the head and insert at the tail. Note that we need to maintain references to both the head and tail nodes of the list.

To implement the queue using singly linked list, assume we have two pointers head and tail, which point to the front and rear of the queue respectively. That is, we place the front of the queue at the head of the linked list and the rear of the queue at the tail.

Hence, enqueue is inserting a node at the end of a linked list and dequeue is deleting the first node in the list. Then, the algorithm for them looks like the following.

```
void enqueue (int elem)
{
 Node *node = new Node;
 node -> data = elem;
 node -> next = NULL; // node will be new tail node
 if(size = = 0)
   head = node; // special case of a previously empty queue
 else
   tail -> next = node; // add node at the tail of the list
 tail = node; // update the reference to the tail node
 size + +;
}
```

```
int dequeue ( )
{
  if (size = = 0)
    cout<<"Queue is empty";
  else
   {
     tmp = head -> data;
     head = head -> next;
     size - - ;
     if (size = = 0)
       tail = NULL; // the queue is now empty
     return tmp;
   }
}
```

Each of the methods of the singly linked list implementation of the queue ADT runs in O(1) time. We also avoid the need to specify a maximum size for the queue, as was done in the array-based queue implementation, but this benefit comes at the expense of increasing the amount of space used per element. Still, the methods in the singly linked list queue implementation are more complicated than we might like, for we must take extra care in how we deal with special cases where the queue is empty before an enqueue or where the queue becomes empty after a dequeue.

5.4. Deque (Double - ended Queue)

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Insertion and deletion can occur at either end. Such an extension of a queue is called a double-ended queue, or deque, which is usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q." An easy way to remember the "deck" pronunciation is to observe that a deque is like a deck of cards in the hands of a crooked card dealer - it is possible to deal off both the top and the bottom.

The fundamental functions of the deque ADT are stated as follows.

- **EnqueueFront** (elem): Insert a new element *elem* at the beginning of the deque.
- **EnqueueRear** (elem): Insert a new element *elem* at the end of the deque.
- **DequeueFront** ( ): Remove the first element of the deque; an error occurs if the deque is empty.
- **DequeueRear** ( ): Remove the last element of the deque; an error occurs if the deque is empty.

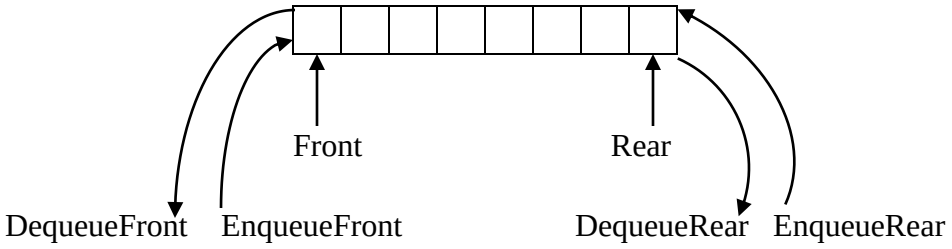
Additionally, the deque includes the following support functions:

- **Front** ( ): Return the first element of the deque; an error occurs if the deque is empty.
- **Rear** ( ): Return the last element of the deque; an error occurs if the deque is empty.
- **Size** ( ): Return the number of elements of the deque.
- **IsEmpty** ( ): Return true if the deque is empty and false otherwise.

The following table shows a series of operations and their effects on an initially empty deque of integers.

Operations	Output	Deque
EnqueueFront (3)	-	(3)
EnqueueFront (5)	-	(5, 3)
Front ( )	5	(5, 3)
DequeueFront ( )	-	(3)
EnqueueRear (7)	-	(3, 7)
Size ( )	2	(3, 7)
Rear ( )	7	(3, 7)
DequeueFront ( )	-	(7)
IsEmpty ( )	False	(7)
DequeueRear ( )	-	( )

A deque can be seen in the following figure.



As like that of queue, a deque can be implement both using array and linked list. Implementing using array may require shifting (for example inserting at front, when we don't have free spaces at front part), and this shifting requires cost. The best thing to implement a deque is using doubly linked list.

5.5. Priority Queue

In many situations, simple queues are inadequate, because first in/first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have

priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue.

Similarly, in a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally, a process is allowed to run only for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue if it does not finish. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running. Furthermore, some jobs that are not short are still very important and should also have precedence.

This particular application seems to require a special kind of queue, known as a priority queue.

A priority queue is an abstract data type for storing a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time. This ADT is fundamentally different from the position-based data structures such as stacks, queues, deques and lists. These other data structures store elements at specific positions, which are often positions in a linear arrangement of the elements determined by the insertion and deletion operations performed. The priority queue ADT stores elements according to their priorities, and has no external notion of "position."

In general, a situation in which items are queued not just by time of receipt, but on the basis of a value or priority is called a priority queue. In priority queues, elements are dequeued according to their priority and their current queue position.

Having described the priority queue abstract data type at an intuitive level, we now describe it in more detail. As an ADT, a priority queue *P* supports the following functions:

- **Size** ( ): Return the number of elements in *P*.
- **Isempty** ( ): Return true if *P* is empty and false otherwise.
- **Min** ( ): Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.
- **Insert** (elem): Insert a new element *elem* into *P*.
- **RemoveMin** ( ): Remove from *P* the element referenced by *min* ( ); an error condition occurs if the priority queue is empty.

As mentioned above, the primary functions of the priority queue ADT are the *insert*, *min*, and *removeMin* operations. The other functions, *size* and *isempty*, are generic collection operations. Note that we allow a priority queue to have multiple entries with the same key.

The following table shows a series of operations and their effects on an initially empty priority queue P. Each element consists of an integer, which we assume to be sorted according to the natural ordering of the integers. Note that each call to min returns a reference to an entry in the queue, not the actual value. Although the "Priority Queue" column shows the items in sorted order, the priority queue need not store elements in this order.

Operation	**Output**	**Priority Queue**
Insert (5)	-	{ 5 }
Insert (9)	-	{ 5, 9 }
Insert (2)	-	{ 2, 5, 9 }
Insert (7)	-	{ 2, 5, 7, 9 }
Min ( )	[2]	{ 2, 5, 7, 9 }
RemoveMin ( )	-	{ 5, 7, 9 }
Size ( )	3	{ 5, 7, 9 }
Min ( )	[5]	{ 5, 7, 9 }
RemoveMin ( )	-	{ 7, 9 }
RemoveMin ( )	-	{ 9 }
RemoveMin ( )	-	{ }
IsEmpty ( )	True	{ }
RemoveMin ( )	" Error"	{ }

Example: Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Male	Female	Male	Male	Female	Male

Dequeue ( ) - deletes Aster

Abebe	Alemu	Belay	Kedir	Meron	Yonas
Male	Male	Male	Male	Female	Male

Dequeue ( ) - deletes Meron

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues.

Dequeue ( ) - deletes Abebe

Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male

Dequeue ( ) - deletes Alemu

Belay	Kedir	Yonas
Male	Male	Male

5.6. Applications of Queue

There are many algorithms that use queues to give efficient running times. For now, we will give some simple examples of queue usage.

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.
- In universities, where resources are limited, students must sign a waiting list if all computers are occupied. The student who has been at a computer the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.
- Task scheduler in multiprocessing system - maintains priority queues of processes.
- Telephone calls in a busy environment - maintains a queue of telephone calls.