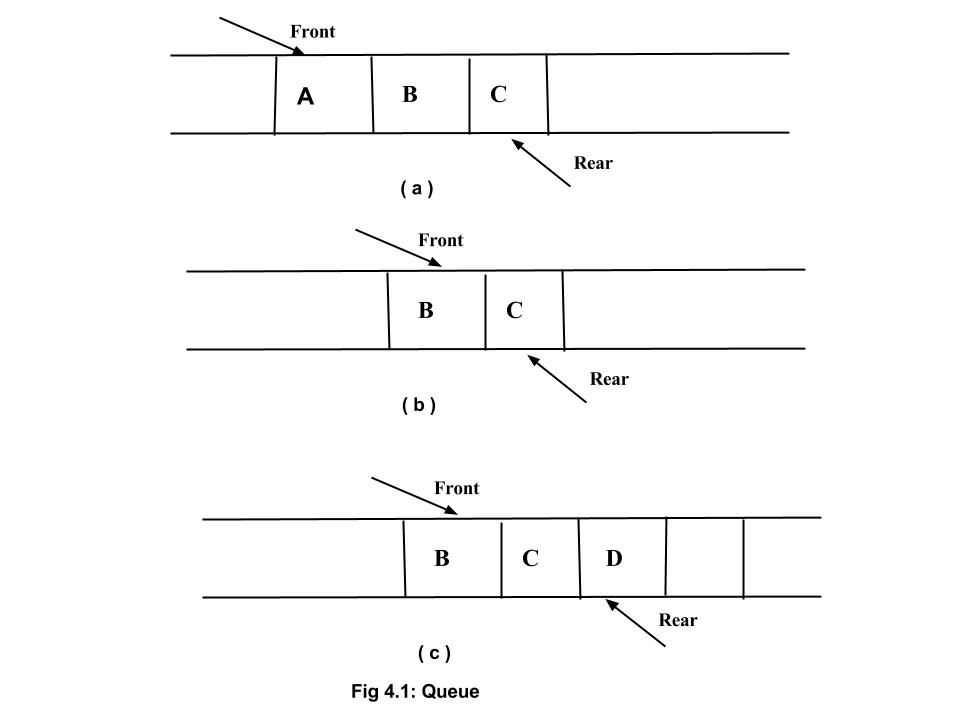
**Unit 4: Queues**

Total hours : 4 Total Marks: 5

1. **Concept and Definition**

A ***queue*** is an ordered collection of items from which an items may be deleted at one end (called the ***front***/***head*** of the queue) and into which items may be inserted at the other end (called the ***rear/tail*** of the queue). Figure 4.1a illustrates a queue containing three elements A, B and C. A is at the front of the queue and C is at the rear. In figure 4.1b an element has been deleted from the queue. Since element may be deleted from the front of the queue, A is removed and B is now at the front. In figure 4.1.c, when item D is inserted, it must be inserted at rear of the queue.



Since C was inserted into the queue before D, it will be removed earlier. The first element inserted into a queue is the first element to removed. For this reason a queue is sometimes called a ***fifo*** (first-in, first-out) list as opposed to a stack, which is ***lifo*** ( last-in, first-out ) list. A line at a bank or at a bus stop are familiar examples of queues.

Three primitive operations which can be performed on queue are explained as follows:

1. insert ( q, elt ) : insert element elt at the rear of the queue q
2. DATA = remove ( q ) : deletes the front element from the queue q and set DATA to its content
3. empty ( q ) : returns true or false depending on whether or not the queue contains any element

The insertion and deletion operation is often called enqueue and dequeue on queue. There is no limit on insert operation on queue, so any number of item can be inserted on queue. The remove operation, however, can be applied only the queue is non empty.The result of an illegal attempt to remove an element from an empty queue is called underflow.

1. **Queue as an ADT**

The representation of a queue as an abstract data type is straightforward. We use eltype to denote the type of the queue element and parametrize the queue type with eltype.

**abstract typedef** << eltype >> QUEUE (eltype) ;

**abstract** empty(q)

QUEUE(eltype) q;

**postcondition**  empty == (len (q) == 0)

**abstract** eltype remove(q)

QUEUE(eltype) q;

**precondition:** empty(q) == FALSE;

**postcondition:** remove == first ( q’ ) ;

q = = sub ( q’ , 1, len ( q’ ) - 1) ;

**abstract insert ( q , elt)**

QUEUE(eltype) q;

eltype elt ;

**postcondition:** q == q’ + < elt > ;

In other way, Queue is expressed as an ADT as follows:

A queue of elements of type T is a finite sequence of elements of T together with the operations.

1. Enqueue/Insert/Add (q, x): to insert an item, x at the rear of the queue.
2. Dequeue/Delete/Remove/serve (q): if the queue is not empty, remove the element at the rear of the queue.
3. IsEmpty( q ): Determine whether queue is empty or not. Return true if it is empty else false.
4. IsFull( q) : Determine whether queue is full or not. Return true if it is full else false. (Note: Is Full is only used if it is implemented using array to check upper bound)
5. **Application of Queue**

The queue is very useful in various field of computer science. The real time application of queue are as follows:

Familiar applications.

1. iTunes playlist.
2. Data buffers (iPod, TiVo).
3. Asynchronous data transfer (file IO, pipes, sockets).
4. Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

1. Traffic analysis.
2. Waiting times of customers at call center.
3. Determining number of cashiers to have at a supermarket
4. **Implementation of Queues**

We can implement queue in two different ways. Like stack, we can implement queue using static and dynamic memory allocation method. The static implementation of queue is called array implementation and the dynamic implementation is also called Linked list implementation.

Queue can be implemented in following two ways:

1. Array (static) implementation: In Array implementation of queue, array is used to store collection of data items. It is further classified into following two categories:
   1. Linear Array implementation

In linear array implementation two array pointers, front and rear, are used to keep track of start and end of data items. It is also called linear queues.

* 1. Circular Array implementation

In a Circular Array implementation of queues, we view the array, holding the queue, as a circular rather than a straight line. Due to which it is also called circular queues.

1. Linked list (dynamic) implementation : It is a dynamic implementation of queue. We will discuss it more on linked list. (next chapter)
2. **Array implementation of Linear Queues in C**

In a Linear queue, front is set to 0 and rear is set to -1 to indicate empty queue. Let us define queue data structure .

**#define** MAXQUEUE 100

**struct** queue{

**int** items[MAXQUEUE] ;

**int** front, rear;

};

In array implementation, overflow condition should be always checked before insertion operation. In linear queue, initially, front of queue is set to 0 and rear is set to -1.The no. of element at any time is equal to the value of q.rear-q.front+1. Algorithm to insert an element elt on queue q is implemented as follows:

1. If q->rear = SIZE – 1, then:
   1. Display “The queue is in overflow condition”
   2. Exit
2. rear = rear + 1
3. Queue [rear] = ITEM
4. Exit

To implement dequeue operation, at first, it should be checked whether queue is empty or not. Element can’t be dequeued if queue contains no element. If we try to delete an element from the empty queue, the condition arises is called underflow condition. It is implemented on queue by:

1. if ( q-> front > q-> rear )

return TRUE;

1. else

return FALSE;

The remove operation on queue is implemented by:

1. If q->rear < q->front, then:
   1. Display “The queue is in underflow condition”
   2. Exit
2. ITEM = Queue [front]
3. front = front + 1
4. Exit

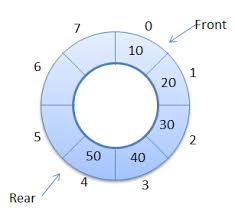
In each operation of queue, front and rear are always increases. Let us define an array of size 3 to store three element on queue. Fig 4.2.a is an empty queue in which front is at 0 and queue is at -1. After insertion of three element A, B and C, the new queue becomes as shown in fig 4.2.b. Figure 4.2.c illustrate the queue becomes after deletion of an element on queue. The rear of the queue is at SIZE-1, which indicates queue is full, however, first position of array is vacant. If we try to add an new element, overflow condition will be arises.



First technique to avoid this is just shift all element on array after each deletion. it will resolve the problem of creating vacant cell on array at front. If number of element on queue is large, for example 500, we need to pay high cost to shift 500-1 = 499 elements which make its inefficient to implement.

Second technique to avoid this, we will view array as an circular rather than straight line as we did always. The circular view or implementation of queue is also called circular queue.

In circular queue, when REAR is at the last position and new element is to be inserted, the REAR pointer goes to first position in the list. The logical representation of circular queue implemented using array is shown in the figure.



To achieve this, the increment operations of FRONT and REAR pointers need to be changed as follows:

After inserting an element REAR will be assigned a value of (REAR+1) % SIZE , similarly after removing an element FRONT will be (FRONT+1) % SIZE

**#define** MAXQUEUE 100

**struct** queue{

**int** items[MAXQUEUE] ;

**int** front, rear;

};

**struct** queue q;

q.front = q.rear = MAXQUEUE-1;

Here q.front and q.rear are initialized to the last index of the array, rather than to -1 or 0. Because the last element of the array immediately precedes the first one within the queue under this representation. Since q.front equals q.rear , the queue is initially empty.

* **Algorithm for Empty Condition**

1. if ( q.front = = q.rear )

return true ;

1. else

return false;

The equivalent empty function of above algorithm is written as:

**int** empty (**Struct** queue \*pq)

{

**return** ( ( pq -> front == pq -> rear ) ? TRUE : FALSE ) ;

}/\* end empty\*/

* **Algorithm for Remove / Delete / Dequeue operation**
  1. If front = = rear, then:
     + Display “The queue is in underflow condition”
     + Exit
  2. If front = MAXQUEUE -1
     + set front to 0
  3. front = front + 1;
  4. display x, where x = items[front]

The enqueue function is written as:

**int** dequeue ( **Struct** queue \*pq )

{

**if** ( empty ( pq ) ) {

printf(“queue is in Underflow Condition”);

exit ( 1 );

} /\* End if \*/

**if** ( pq -> front == MAXQUEUE - 1)

pq -> front = 0;

**else**

(pq -> front) ++;

**return** ( pq -> items[ pq -> front]);

}/\*End dequeue\*/

* **Algorithm to Insert/ Enqueue Condition**

1. if rear = MAXQUEUE - 1
   * set rear to zero
2. rear = rear + 1
3. if rear = = front
   * print queue is in overflow condition
   * exit
4. items[rear] = x

**C implementation of insert operation is defined below:**

The insert routine may then be written as follows:

v**oid** insert ( **Struct** queue \*pq, int x)

{

/\* make room for new element\*/

**if** ( pq -> rear = = MAXQUEUE - 1)

pq -> rear = 0;

**else**

( pq -> rear ) ++;

/\*check for overflow\*/

**if** ( pq -> rear == pq -> front ) {

printf(“Queue overflow”);

exit(1);

}/\* End if \*/

pq- > items[pq->rear] = x;

**return;**

}/\* End insert \*/

Here, test for overflow in insert occurs after **pq -> rear** has been adjusted, whereas the test for underflow in r**emove/dequeue** occurs immediately upon entering the routine, before **pq->front** is updated.

1. **Concept of Priority Queue**

A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority.

A priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates. The values put into the priority queue would be the priority of the sender

(e.g., using 4 for the chair, 3 for professors, 2 for grad students, and 1 for undergrads), and the associated information would be the document to print. Each time the printer is free, the job with the highest priority would be removed from the print queue, and printed. (Note that it is OK to have multiple jobs with the same priority; if there is more than one job with the same highest priority when the printer is free, than any one of them can be selected.

Two types of priority queues are,

i) Ascending priority queue – In this queue, the items can be inserted arbitrarily and only the smallest item will be removed.

ii) Descending priority queue- This allows insertion of items arbitrarily, and only the maximum element from queue will be removed first

What are the limitations in priority queue? How it could be rectified?

Deletion of elements make a location empty. The search of items include the empty spaces too. This takes more time. To avoid this, use an empty indicator or shift elements forward when each element is deleted. We can also maintain priority queue as an array of ordered elements, to avoid risk in searching.