Unit 6 - Queues

Overview:

Queues are data structures that follow the First In First Out (FIFO) principle where in the first element that is added to the queue is the first one to be removed. In this unit we will explore queues and its applications and learn how to implement it.

Module Objectives:

After successful Completion of this module, you should be able to:

- **Define and describe queue and its characteristics**
- **Discuss the different operations on queue**
- **Develop your own implementation of queue**
- **Apply your queue implementation in solving programming challenges**

Course Materials:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both ends. One end (REAR; also called tail) is always used to insert data (enqueue) and the other (FRONT; also called head) is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first. A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Figure 31 Real-world example of Queue (Single-lane One-way road)

In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Application of Queues

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Queue Representation and Implementation

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the head (FRONT) and the tail (REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

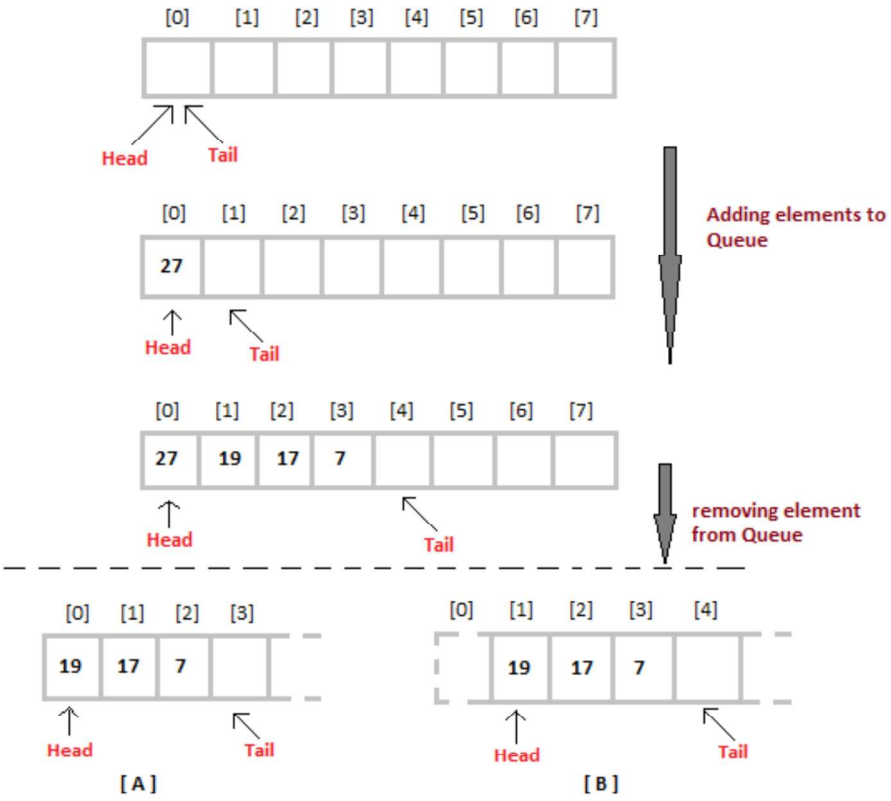

Figure 32 Implementation of Queue

When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at head position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from head position and then move head to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

Circular Queue

In order to resolve the problem with [B] we can implement what we call a circular queue. Given an array A of a default size (≥ 1) with two references back and front, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:

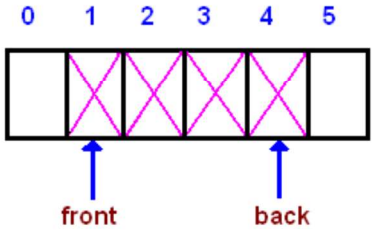

Figure 33 Circular Queue implementation

As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements

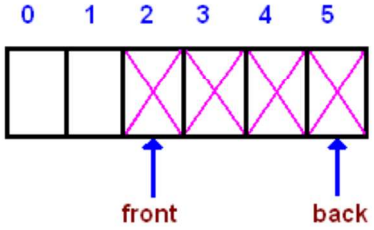

Figure 34 Circular queue (end of linear list reached)

However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until front. Such a model is called a wrap-around queue or a circular queue.

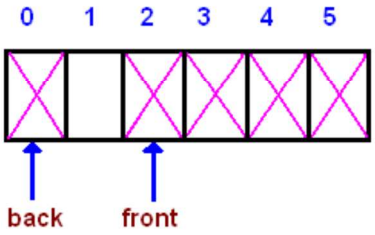

Figure 35 Circular queue (wrap around)

Finally, when back reaches front, the queue is full. There are two choices to handle a full queue: a) throw an overflow error ; b) double the array size (increase storage space).

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: current - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty. Here's the C programming code −

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks. The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.
Step 2 − If the queue is full, produce overflow error and exit.
Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
Step 4 − Add data element to the queue location, where the rear is pointing.
Step 5 − return success.

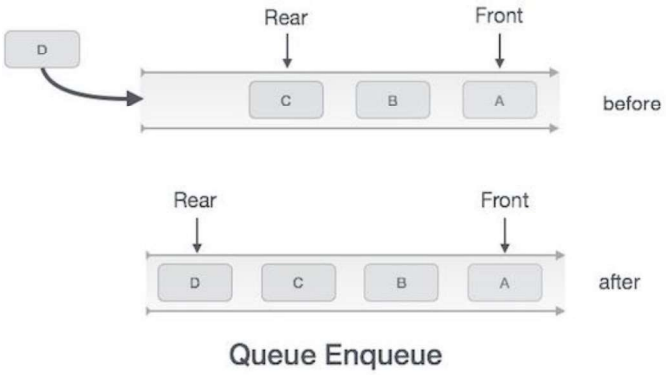

Figure 36 Queue Insert Operation (Enqueue)

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true
end procedure
```

Implementation of enqueue() in C programming language −

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

Step 1 − Check if the queue is empty.
Step 2 − If the queue is empty, produce underflow error and exit.
Step 3 − If the queue is not empty, access the data where front is pointing.
Step 4 − Increment front pointer to point to the next available data element.
Step 5 − Return success.

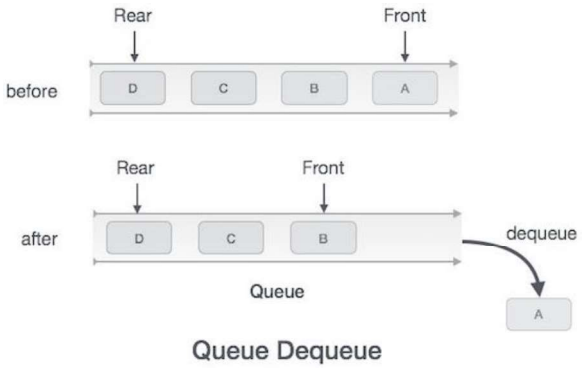

Figure 37 Queue Remove Operation (Dequeue)

Algorithm for dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true
end procedure
```

Implementation of dequeue() in C programming language –

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

Watch:

- Introduction to Queues By mycodeschool (https://youtu.be/XuCbpw6Bj1U)
- Queues By CS50 (https://youtu.be/3TmUv1uS92s)
- Array implementation of queues By mycodeschool (https://youtu.be/okr-XE8yTO8)
- Linked list implementation of queues By mycodeschool (https://youtu.be/A5_XdiK4J8A)
- Circular Queue Implementation – Array By Blue Tree Code (https://youtu.be/8sjFA-IX-Ww)

Read:

- Section 3.7 The Queue ADT (pp. 112-116)
  Weiss, Mark Allen (2014). Data Structures And Algorithm Analysis In C++ 4th Edition

- Section 6.2 Queues (pp.238-247)
  Goodrich, Michael T. (2014). Data Structures and Algorithms in Java 6th Edition

- Section 1.3 Bags, Queues, and Stacks (pp. 120-157)
  Sedgewick, Robert, Wayne, Kevin (2011). Algorithms 4th Edition

Review:

1. What is a queue? How is it different from stack?
2. How are queues implemented?
3. Give two queue operations, and what do these operations do?
4. What are the different types of queues?
5. What is the advantage of using the linked list implementation of queues, as opposed to the array implementation?

Activities/Assessments:

1. Suppose that Q is an initially empty array-based queue of size 5. Show the values of the data members front and back after each statement has been executed. Indicate any errors that might occur.

Queue<Character> Q(5);	front = NULL	back = NULL
Q.enqueue ('A');	front = A	back = A
Q.enqueue ('B');	front = A	back = B
Q.enqueue ('C');	front = A	back = C
char c = Q.dequeue( );	front = B	back = C
Q.enqueue ('A');	front = B	back = A

2. A letter means enqueue and an asterisk means dequeue in the following sequence. Give the sequence of values returned by the dequeue operations when this sequence of operations is performed on an initially empty FIFO queue.

EASYQUESTION

E A S * Y * Q U E * * * S T * * * I O * N * * *

3. Suppose that a client performs an intermixed sequence of (queue) enqueue and dequeue operations. The enqueue operations put the integers 0 through 9 in order onto the queue; the dequeue operations print out the return value. Which of the following sequence(s) could not occur?

b, c, d

   a. 0 1 2 3 4 5 6 7 8 9
   b. 4 6 8 7 5 3 2 9 0 1
   c. 2 5 6 7 4 8 9 3 1 0
   d. 4 3 2 1 0 5 6 7 8 9

Programming:

1. The Josephus problem is the following game: N people, numbered 1 to N, are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins. Thus, if M = 0 and N = 5, players are eliminated in order, and player 5 wins. If M = 1 and N = 5, the order of elimination is 2, 4, 1, 5.

   a) Write a program to solve the Josephus problem for general values of M and N. Try to make your program as efficient as possible. Make sure you dispose of cells.

   b) What is the running time of your program?

   c) If M = 1, what is the running time of your program? How is the actual speed affected by the delete routine for large values of N (N > 100,000)?