Data Structures

Lecture Seven

Second Stage

First Course - 2024-2025

ALI MUWAFAQ SHABAN

Master of Computer Engineering

Data Structures

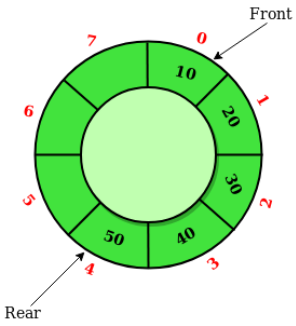
Circular Queue

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

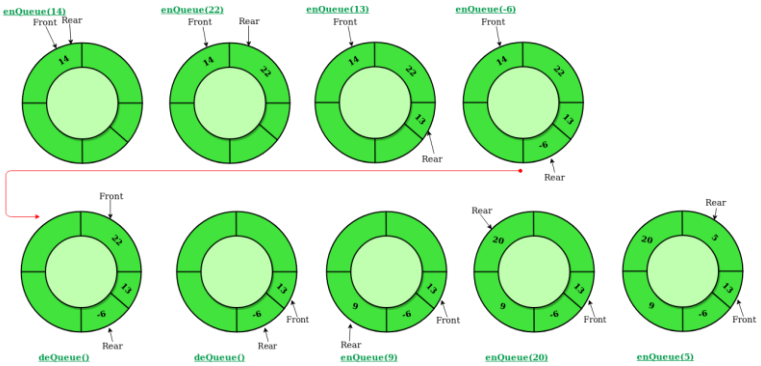
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.

- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

The operations of Circular Queue are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called ‘Ring Buffer’.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

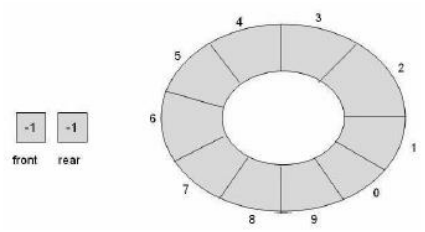


Operations on Circular Queue:

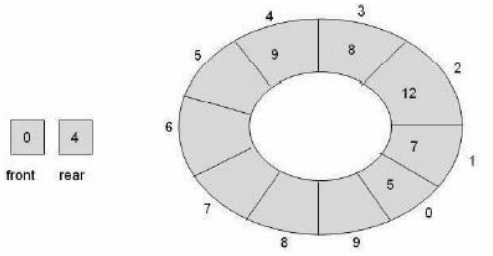
- Front: Get the front item from queue.
- Rear: Get the last item from queue.

The difficulty of managing front and rear in an array-based non-circular queue can be overcome if we treat the queue position with index 0 as if it comes after the last position (in our case, index 9), i.e., we treat the queue as circular. Note that we use the same array declaration of the queue.

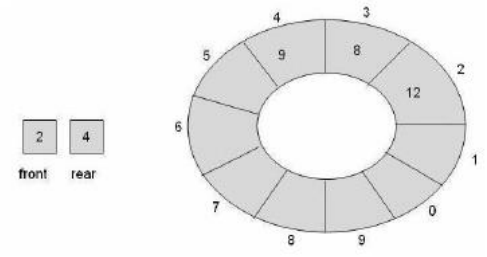
Empty queue:



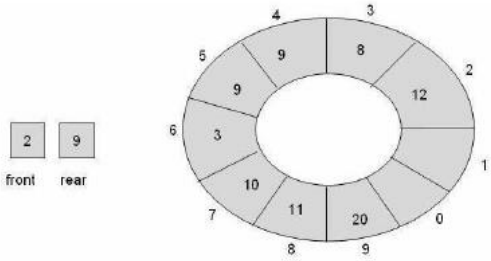
Enqueuing five items:



Dequeuing two items:



Enqueuing five items:



Implementation of operations on a circular queue:

- Testing a circular queue for **overflow** There are two conditions:
- (front=0) and (rear=capacity-1) or
- front=rear+1

If any of these two conditions is satisfied, it means that circular queue is full.

- The **Enqueue** Operation on a Circular Queue There are three scenarios which need to be considered, assuming that the queue is not full:

1. If the queue is empty, then the value of the front and the rear variable will be -1 (i.e., the sentinel value), then both front and rear are set to 0.

2. If the queue is not empty, then the value of the rear will be the index of the last element of the queue, then the rear variable is incremented.
3. If the queue is not full and the value of the rear variable is equal to capacity - 1 then rear is set to 0.
   - The **Dequeue** Operation on a Circular Queue. Also, there are three possibilities:
   1. If there was only one element in the circular queue, then after the dequeue operation the queue will become empty. This state of the circular queue is reflected by setting the front and rear variables to -1.
   2. If the value of the front variable is equal to CAPACITY-1, then set front variable to 0.
   3. If neither of the above conditions hold, then the front variable is incremented.

Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all user defined functions used in circular queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int cQueue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

EnQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

DeQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

Step 1: Check whether queue is EMPTY. (front == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.

Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.

Step 6: Set i to 0.

Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Full Program

```
#include<iostream.h>
const int size=5;
void enq(char q[size],int&f,int&r,char x);
void deq(char q[size],int&f,int&r,char &);
int full_q(int r,int f);
int empty_q(int f);
main()
{char q[size],x;
int no,f=-1,r=-1;
do
{
cout<<"\n1.en_q.\n";
cout<<"2.de_q.\n";
cout<<"3.exit!\n\n";
cout<<"ENTER YOUR CHOICE:";
        cin>>no;
        switch(no)
                {
                 case 1:cout<<"enter char which you want to add in
q:";
                                        cin>>x;
                                        enq(q,f,r,x);
                                        break;
                case 2:deq(q,f,r,x);
                                break;
                case 3:break;
                 }
                 }while(no!=3);
}
int full_q(int r,int f)
{
if(((r+1)%size)!=f) return 0;
else return 1;
}
int empty_q(int f){
if(f==-1)return 1;
else return 0;
}
void enq(char q[size],int&f,int&r,char x)
{
```

```
if(full_q(r,f))cout<<"Queue is full!!";
  else
        {
         r=(r+1)%(size);
         q[r]=x;
         if(f==-1)f=0;
         cout<<"Q["<<r<<"]="<<q[r];
        }
}
void deq(char q[size],int&f,int&r,char &x)
{
if(empty_q(r))cout<<"Queue is empty!!";
        else
        {
        x=q[f];
        cout<<"Q["<<f<<"]="<<x;

        if(f==r)f=r=-1;
        else f=(f+1)%size;
        }
}
```

Applications

- Scheduling jobs on a workstation holds jobs to be performed and their priorities. When a job is finished or interrupted, highest-priority job is chosen using Extract-Max. New jobs can be added using Insert function.
- Operating System Design – resource allocation
- Data Compression -Huffman algorithm
- Discrete Event simulation

(1) Insertion of time-tagged events (time represents a priority of an event -- low time means high priority)

(2) Removal of the event with the smallest time tag

- In a time-sharing computer system, for example, a large number of tasks may be waiting for the CPU. Some of these tasks have higher priority than others. Hence the set of tasks waiting for the CPU forms a priority queue. Other applications of priority queues include simulations of time-dependent events (like the airport simulation) and solution of sparse systems of linear equations by row reduction.

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.