CS-2001
Data Structures
Spring 2022
Introduction to Queue ADT

Mr. Muhammad Yousaf

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

Queues

- Queue is First-In-First-Out (FIFO) data structure
  - First element added to the queue will be first one to be removed

- Queue implements a special kind of list
  - Items are inserted at one end (the rear)
  - Items are deleted at the other end (the front)

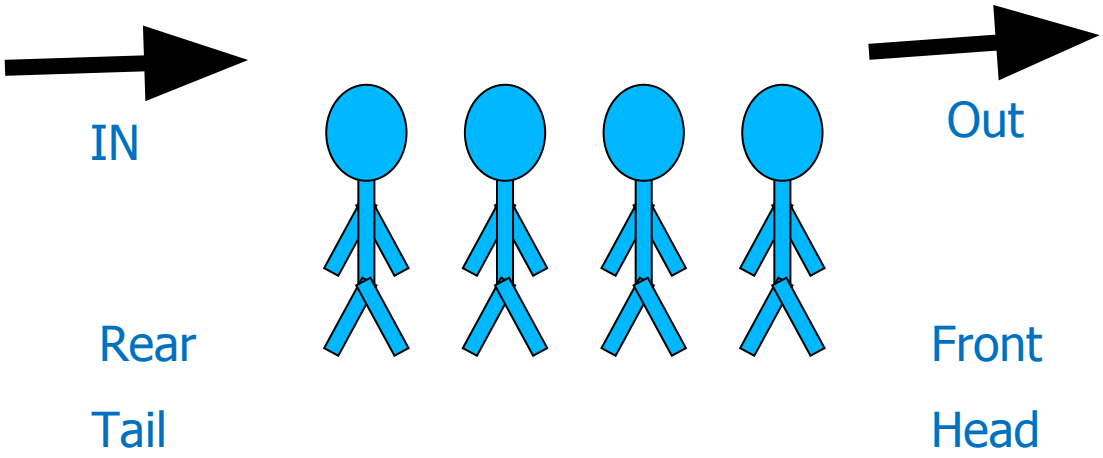
Queues Vs Stack

- Stack is Last-In-First-Out (**LIFO**) data structure

- Queue is First-In-First-Out (**FIFO**) data structure

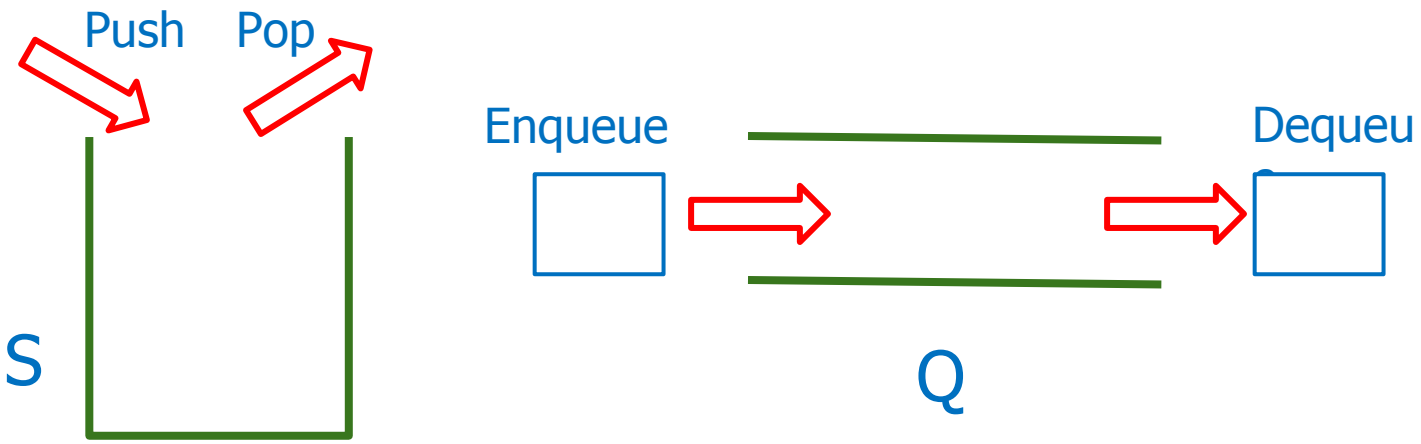
Queue – Analogy (1)

- A queue is like a line of people waiting for a bank teller
- The queue has a front and a rear



Queue – ADT

- A list or collection with the restriction that
  - insertion can be performed at one end (Rear)
  - deletion can be performed at another end (Front)
- Operations
  - Enqueue(x)
  - Dequeue(x)
  - Front()
  - IsEmpty()
  - IsFull()



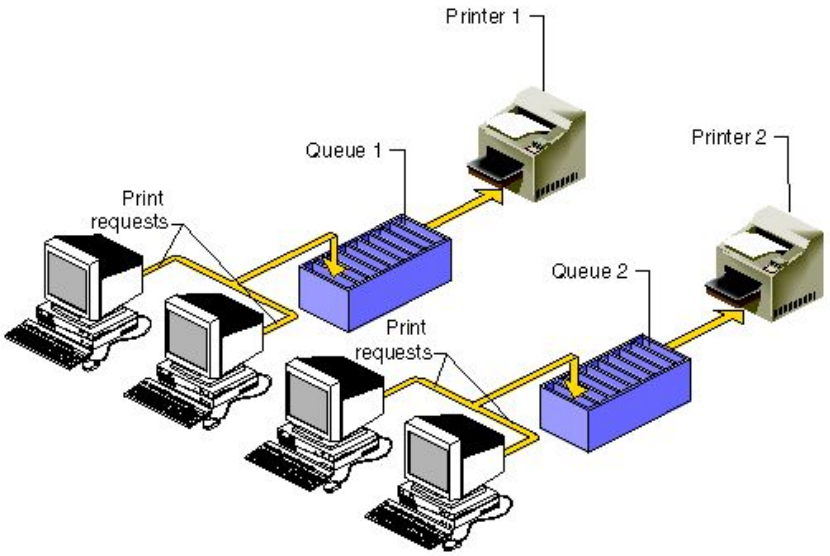
Queue - Library Implementation

```
// CPP code to illustrate Queue in  Standard Template Library (STL)
#include <iostream>
#include <queue>
using namespace std;
// Driver Code
int main()
{
        queue<int> gquiz;
        gquiz.push(10);
        gquiz.push(20);
        gquiz.push(30);

        cout << "\ngquiz.size() : " << gquiz.size();
        cout << "\ngquiz.front() : " << gquiz.front();
        cout << "\ngquiz.back() : " << gquiz.back();
        gquiz.pop();
        while (!g.empty()) {
                        cout << '\t' << g.front();
                        g.pop();
                }
        return 0;
}
```

Queues – Examples

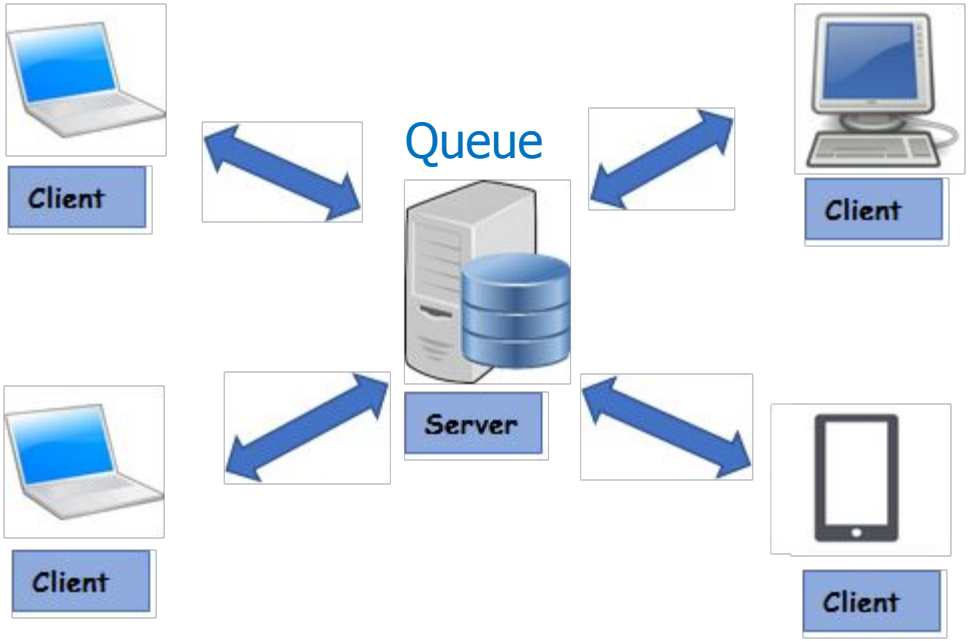
- Billing counter
  - Booking movie tickets
  - Queue for paying bills
- Vehicles on toll-tax bridge
- Luggage checking machine
- A printer queue of computers
- Processes scheduling queue in OS
- And others?



Queues – Applications

- Operating systems
  - Process scheduling in multiprogramming environment
  - Controlling provisioning of resources to multiple users (or processes)

- Middleware/Communication software
  - The most common application is in client-server models
    - Multiple clients may be requesting services from one or more servers

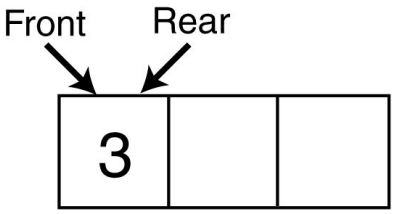


Basic Operations (Queue ADT)

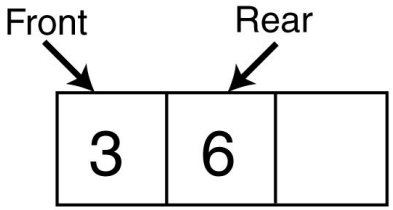
- MAKENULL(Q)
  - Makes Queue Q be an empty list
- FRONT(Q)
  - Returns the first element on Queue Q
- ENQUEUE(x,Q)
  - Inserts element x at the end of Queue Q
- DEQUEUE(Q)
  - Deletes the first element of Q
- EMPTY(Q)
  - Returns true if and only if Q is an empty queue

Enqueue And Dequeue Operations

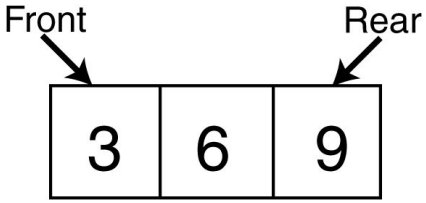
Enqueue(3);



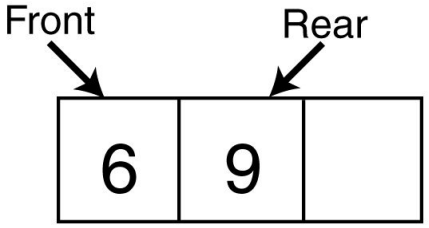
Enqueue(6);



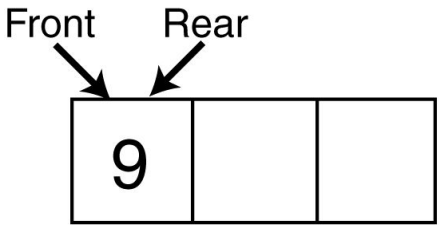
Enqueue(9);



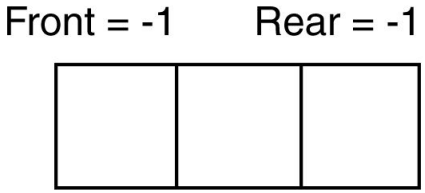
Dequeue();



Dequeue();



Dequeue();



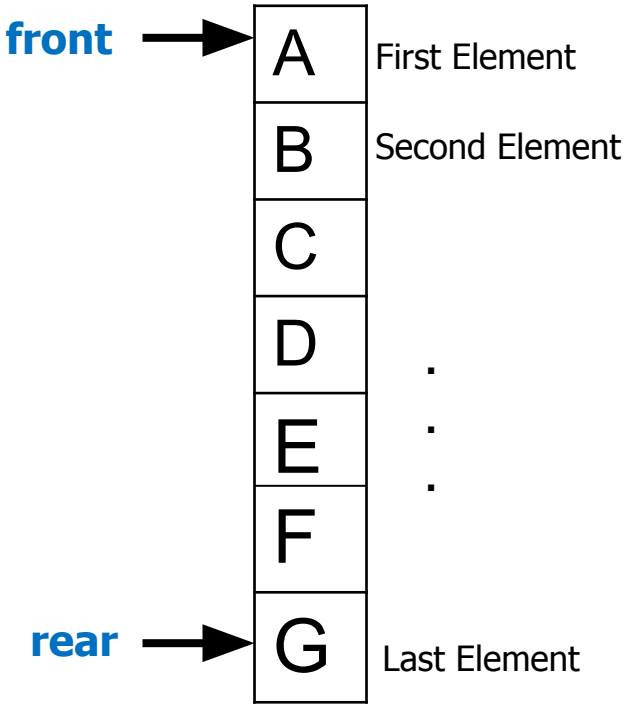
Implementation

- Static
  - Queue is implemented by an array
  - Size of queue remains fix

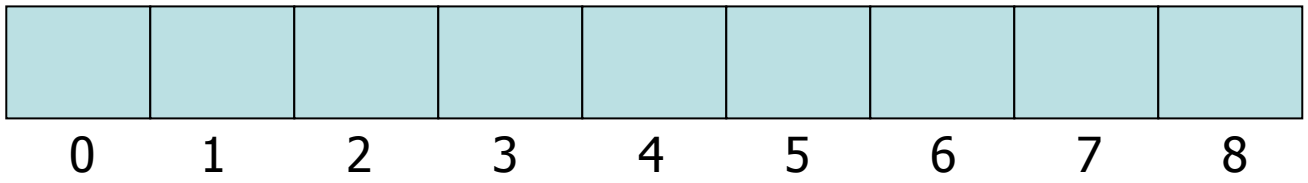
- Dynamic
  - A queue can be implemented as a linked list
  - Expand or shrink with each enqueue or dequeue operation

Array Implementation

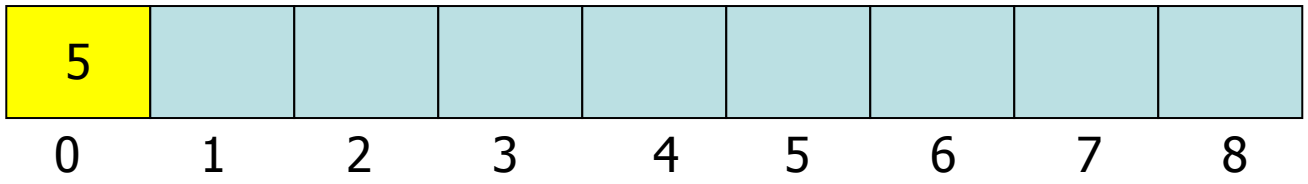
- Use two counters that signify rear and front
- When queue is empty
  - Both front and rear are set to -1
- When there is only one value in the Queue,
  - Both rear and front have same index
- While enqueueing increment rear by 1
- While dequeueing, increment front by 1



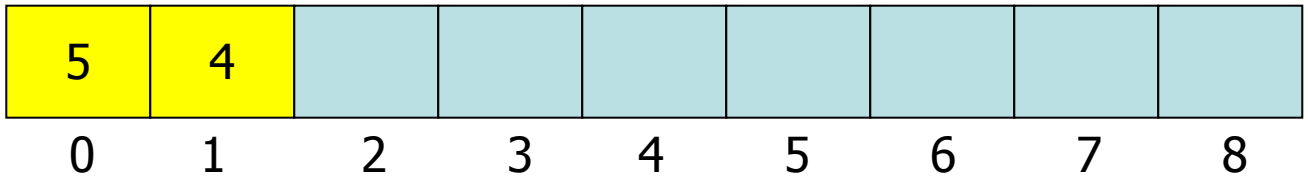
Array Implementation Example (1)



front= -1
rear = -1

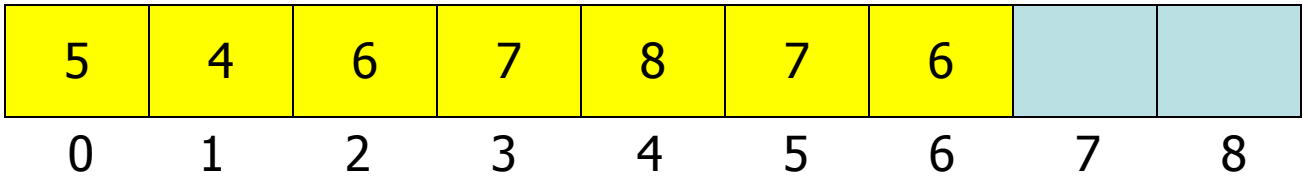


front= 0
rear = 0

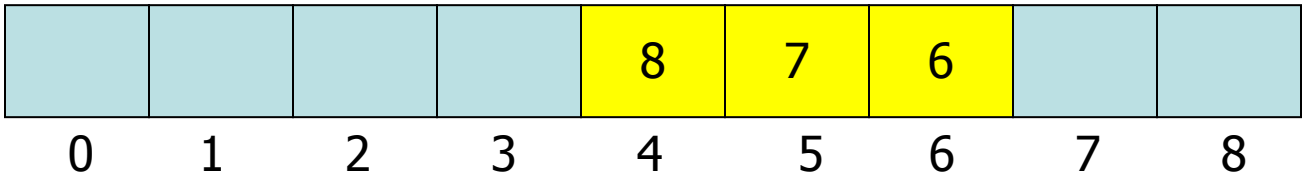


front= 0
rear = 1

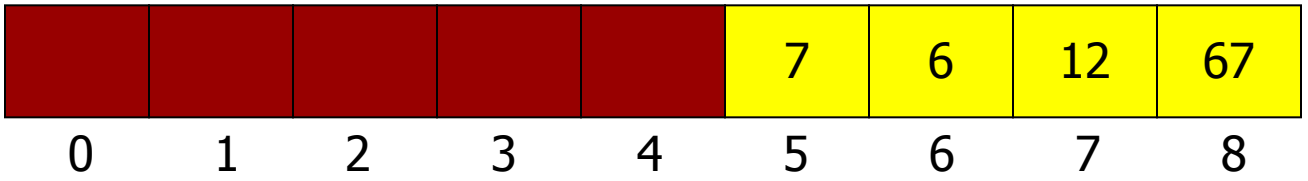
Array Implementation Example (2)



front=0
rear=6



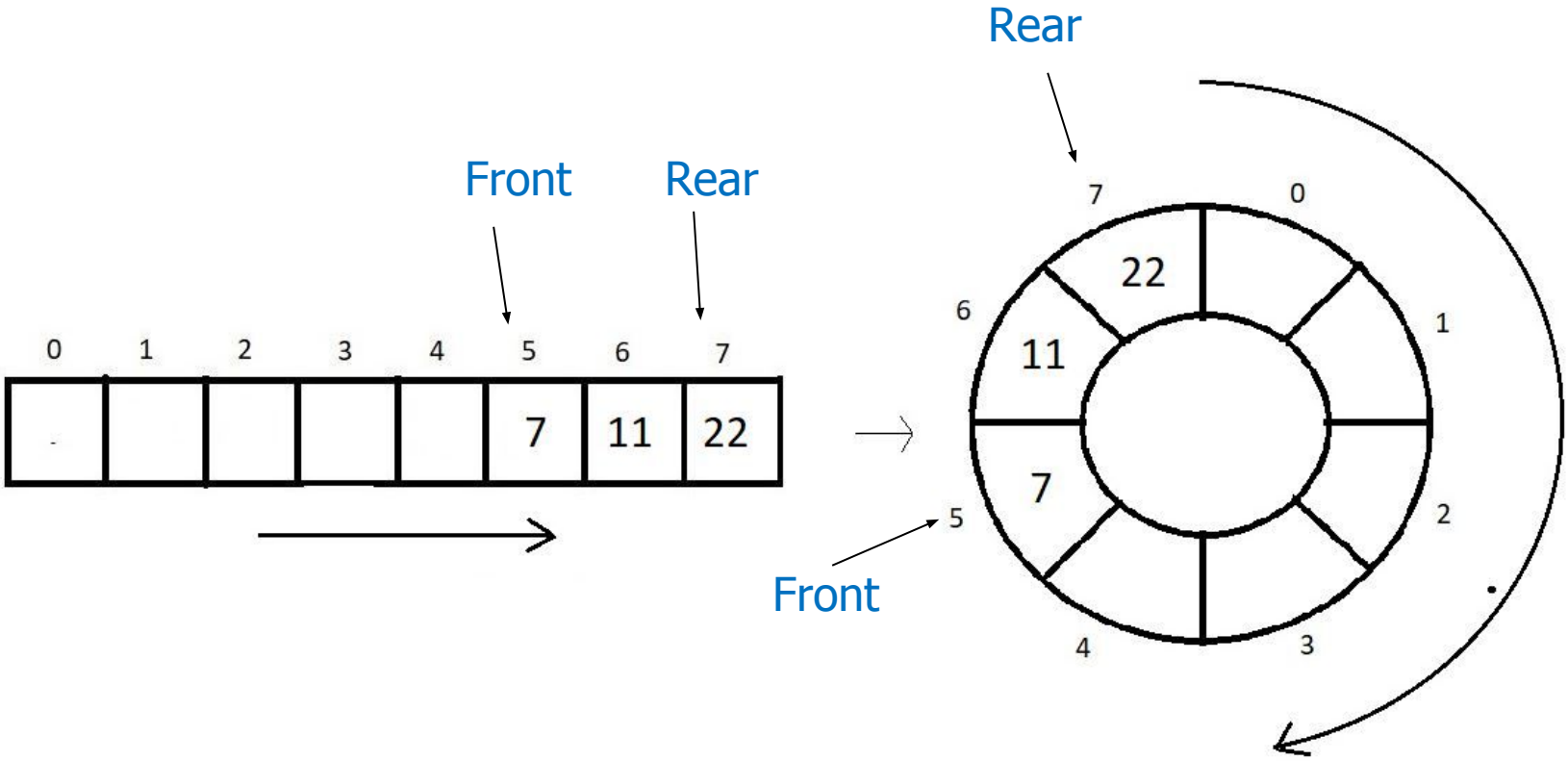
front=4
rear=6



front=5
rear=8

Problem: How can we insert more elements?
Rear index can not move beyond the last element….

Array Implementation Example (2)



current position = i
next position = i + 1 % N
previous position ???

Logical Representation

Using Circular Queue

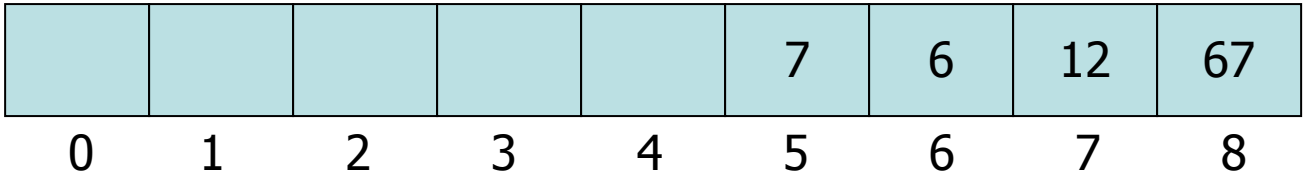
- Allow rear to wrap around the array

```
if(rear == queueSize-1)
      rear = 0;
else
      rear++;
```

- Alternatively, use modular arithmetic

```
rear = (rear + 1) % queueSize;
```

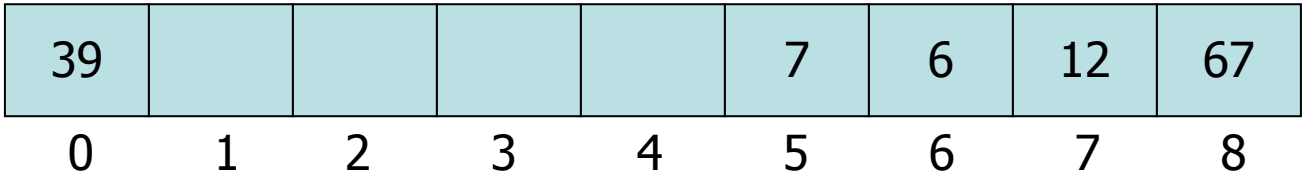
Example



front=5
rear=8

Enqueue 39

- Rear = (Rear+1) mod queueSize = (8+1) mod 9 = 0



front=5
rear=0

Problem: How to avoid overwriting an existing element?

How to Determine Empty and Full Queues?

- A counter indicating number of values/items in the queue
  - Covered in first array-based implementation

- Without using an additional counter (only relying on front and rear)
  - Covered in alternative array-based implementation

Array-based Implementation

Implementation – Code (1)

```
class CQueue
{
   Private:
      int *queueArray; // Pointer to array implemented as Queue
      int queueSize;   // Total size of the Queue
      int front;
      int rear;
   public:
      CQueue(int size);
      ~CQueue();
      bool isFull();
      bool isEmpty();
      bool enqueue(int num);
      bool dequeue(int& num);
      void makeNull();
};
```

Implementation – Code (2)

```
CQueue::CQueue(int size)
{
    front=rear=-1;
    queueArray =new int[size];
    queueSize = size;
}

CQueue::~CQueue() //destructor
{
      delete [] queueArray;
}


void IntQueue::makeNull()
{
    front = - 1;
    rear = - 1;
}
```

Implementation – Code (3)

- `isEmpty()` returns true if the queue is empty and false otherwise

```
bool CQueue::isEmpty()
{
   if (front==-1)
      return true; // we can check "rear" too
   else
      return false;
}
```

- `isFull()` returns true if the queue is full and false otherwise

```
bool CQueue::isFull()
{
   if ( ( (rear+1)%queueSize ) == front )
      return true;
   else
      return false;
}
```

Implementation – Code (4)

- Function `enqueue` inserts the value in `num` at the end of the Queue

```
bool CQueue ::enqueue(int num);
{
    if ( isFull() ) {
        cout<<"Overflow";
        return false;
    }
    if (isEmpty())
        rear = front = 0;
    else
        rear=(rear+1) % queueSize;

    queueArray[rear] = num;
    return true;
}
```

Comparison: `enqueue` Operation

```
bool CQueue ::enqueue(int num);
{
   if ( isFull() ) {
      cout<<"Overflow";
      return false;
   }
   if (isEmpty())
      rear = front = 0;
   else
      rear=(rear+1) % queueSize;

   queueArray[rear] = num;
   return true;
}
```

```
bool IntQueue::enqueue(int num)
{
     if (isFull())
     {
          cout << "Overflow.\n";
          return false;
     }

     // Calculate the new rear position
     rear = (rear + 1) % queueSize;
     // Insert new item
     queueArray[rear] = num;
     // Update item count
     numItems++;
     return true;
}
```

Implementation – Code (5)

- Function `dequeue` removes and returns the value at the front of the Queue

```
bool CQueue ::dequeue(int &num)
{
   if ( isEmpty() ) {
      cout<<"Underflow";
      return false;
   }
   num = queueArray[front];

   if ( front == rear ) //only one element in the queue
                //skipping this step will effect
      front = rear = -1; //the isEmpty() function
   else
      front = (front+1) % queueSize;

   return true;
}
```

Alternative Array-based Implementation

Array Implementation – Code (1)

```
class IntQueue
{
   private:
    int *queueArray; // Pointer to array implemented as Queue
    int queueSize;   // Total size of the Queue
    int front;
    int rear;
    int numItems;    // Number of items currently in the Queue
   public:
    IntQueue(int);
    ~IntQueue();
    bool isEmpty();
    bool isFull();
    bool enqueue(int);
    int dequeue();
    void makeNull();

};
```

Array Implementation – Code (2)

```
class IntQueue
{
   private:
    int *queueArray; // Pointer to array implemented as Queue
    int queueSize;   // Total size of the Queue
    int front;
    int rear;
    int numItems;    // Number of items currently in the Queue
   public:
    IntQueue(int);
    ~IntQueue();
    bool isEmpty();
    bool isFull();
    bool enqueue(int);
    int dequeue();
    void makeNull();

};
```

Clears the queue by resetting the `front` and `rear` indices, and setting the `numItems` to 0.

Array Implementation – Code (3)

- Constructor

```
IntQueue::IntQueue(int s) //constructor
{
   queueArray = new int[s];
   queueSize = s;
   front = -1;
   rear = -1;
   numItems = 0;
}
```

- Destructor

```
IntQueue::~IntQueue() //destructor
{
   delete [] queueArray;
}
```

Array Implementation – Code (4)

```
//*****************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.              *
//*****************************************

bool IntQueue::isEmpty()
{
    if (numItems == 0)
        return true;
    else
        return false;
}
```

Array Implementation – Code (5)

```
//*****************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.             *
//*****************************************

bool IntQueue::isFull()
{
    if (numItems == queueSize)
        return true;
    else
        return false;
}
```

Array Implementation – Code (6)

```
//*****************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*****************************************

bool IntQueue::enqueue(int num)
{
    if (isFull())
    {
        cout << "Overflow.\n";
        return false;
    }
    // Calculate the new rear position
    rear = (rear + 1) % queueSize;
    //front is only updated in de-queuing an item
    // Insert new item
    queueArray[rear] = num;
    // Update item count
    numItems++;
    return true;
}
```

Array Implementation – Code (7)

```
//*****************************************
// Function dequeue removes the value at the  *
// front of the queue, and copies it into num.*
//*****************************************

bool IntQueue::dequeue(int &num)
{
    if (isEmpty())
    {
        cout << "The queue is empty.\n";
        return false;
    }

    // Move front
    front = (front + 1) % queueSize;
    // Retrieve the front item
    num = queueArray[front];
    // Update item count
    numItems--;
    return true;
}
```

Array Implementation – Code (8)

```
//*****************************************
// Function clear resets the front and rear *
// indices, and sets numItems to 0.         *
//*****************************************

void IntQueue::makeNull()
{
    front = - 1;
    rear = - 1;
    numItems = 0;
}
```

Using Queues

```
int main()
{
   IntQueue iQueue(5);
   cout << "Enqueuing 5 items...\n";
   // Enqueue 5 items.
   for (int x = 0; x < 5; x++)
      iQueue.enqueue(x);
   // Attempt to enqueue a 6th item.
   cout << "Now attempting to enqueue again...\n";
   iQueue.enqueue(5);
   // Deqeue and retrieve all items in the queue
   cout << "The values in the queue were:\n";
   while (!iQueue.isEmpty()){
        int value;
    iQueue.dequeue(value);
        cout << value << endl;
   }
}
```

Output:

```
Enqueuing 5 items...
Now attempting to enqueue again...
Overflow.
The values in the queue were:
0
1
2
3
4
```

Any Question So Far?