



# Chapter 4 Queues

---

College of Computer Science, CQU

# Outline

---

- ❑ **Queue ADT**
- ❑ **Circular Queue**
- ❑ **Linked Queue**
- ❑ **Comparison of Array-Based and Linked Queues**

# Queues

---

- ❑ 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

---

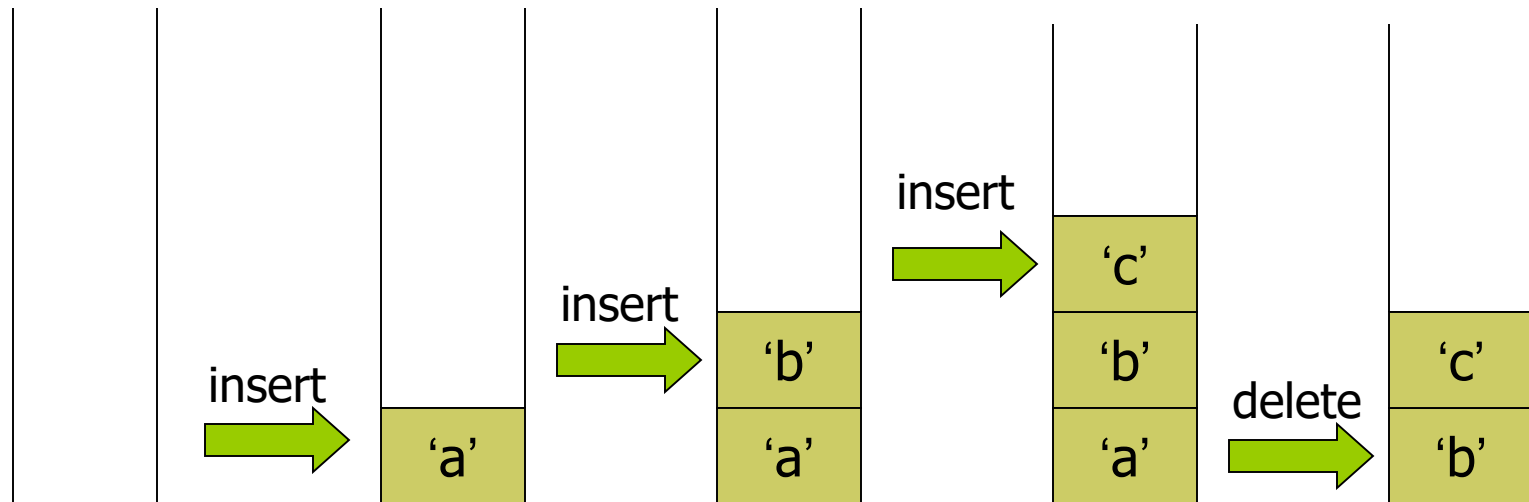
FIFO: First in, First Out

Restricted form of list: Insert at one end, remove from the other.

Notation:

- ▣ Insert: Enqueue
- ▣ Delete: Dequeue
- ▣ First element: Front
- ▣ Last element: Rear

# Example: Queue of Char



# Queue ADT

---

// Abstract queue class

```
template <typename E> class Queue {
```

```
private:
```

```
    void operator =(const Queue&) {} // Protect assignment
```

```
    Queue(const Queue&) {} // Protect copy constructor
```

```
public:
```

```
    Queue() {} // Default
```

```
    virtual ~Queue() {} // Base destructor
```

# Queue ADT

---

```
// Reinitialize the queue. The user is responsible for
// reclaiming the storage used by the queue elements.
virtual void clear() = 0;
// Place an element at the rear of the queue.
// it: The element being enqueued.
virtual void enqueue(const E&) = 0;
// Remove and return element at the front of the queue.
// Return: The element at the front of the queue.
virtual E dequeue() = 0;
// Return: A copy of the front element.
virtual const E& frontValue() const = 0;
// Return: The number of elements in the queue.
virtual int length() const = 0;
};
```



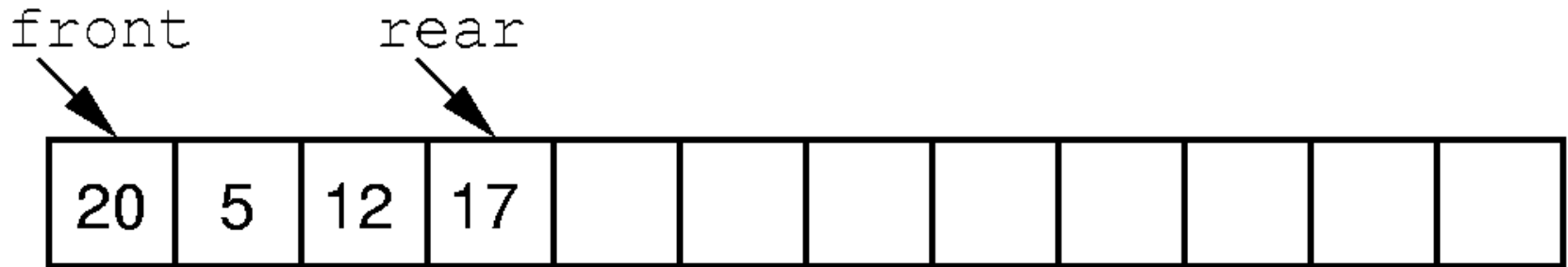
# Array\_based Queue

---

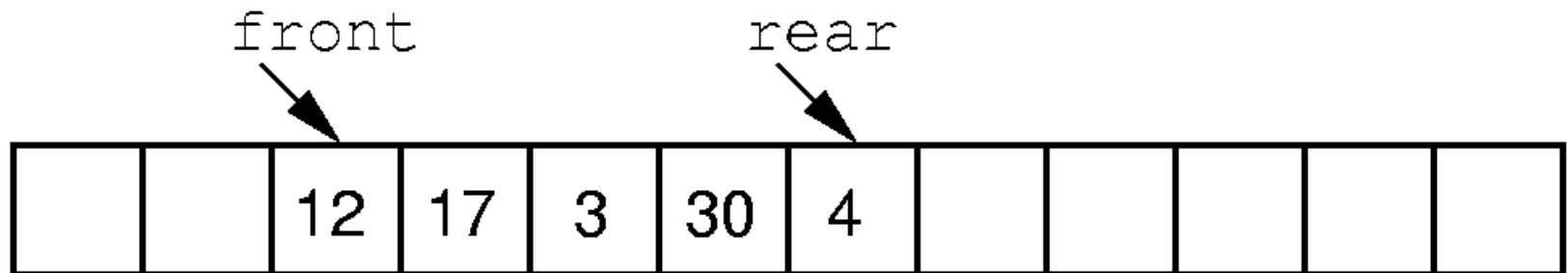
- ❑ The array-based queue is somewhat tricky to implement effectively. Assume that there are  $n$  elements in the queue.
  - If we choose the rear element of the queue to be in position 0, enqueue operations will shift the  $n$  elements currently in the queue one position in the array.
  - If instead we chose the rear element of the queue to be in position  $n-1$ , a dequeue operation must shift all of the elements down by one position to retain the property that the remaining  $n-1$  queue elements reside in the first  $n-1$  positions of the array.
  - A far more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the first  $n$  positions of the array.



# Queue Implementation (1)



(a)



(b)

# Queue Implementation (1)

---

- This implementation raises a new problem.
  - When elements are removed from the queue, the front index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered position in the array, the queue has run out of space. This happens despite the fact that there might be free positions at the low end of the array where elements have previously been removed from the queue.

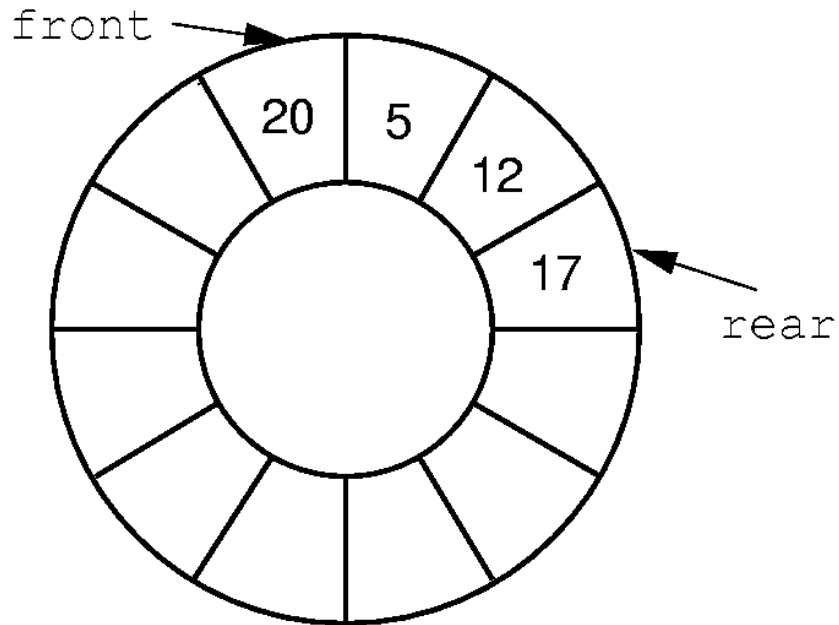


# Queue Implementation (2)

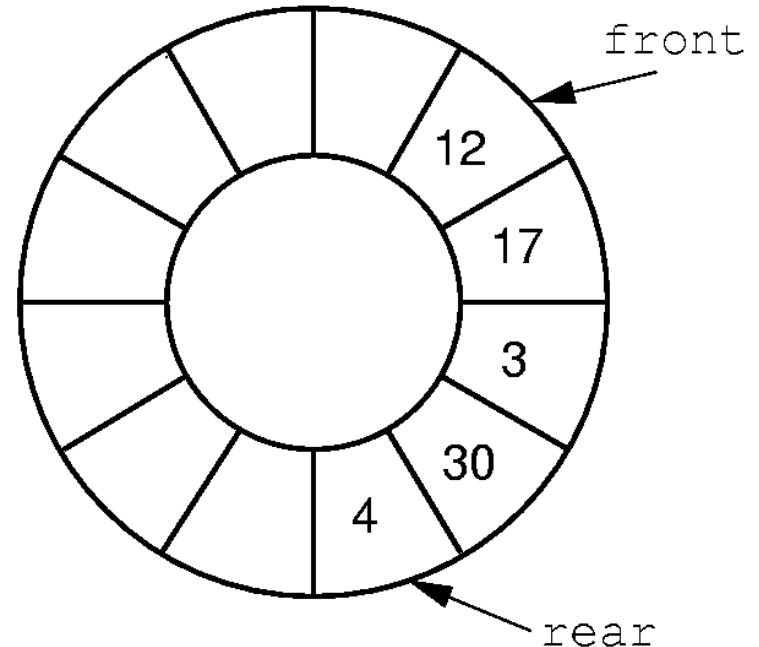
---

- The “drifting queue” problem can be solved by pretending that the array is circular and so allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position.

# Queue Implementation (2)



(a)



(b)

# Queue Implementation (2)

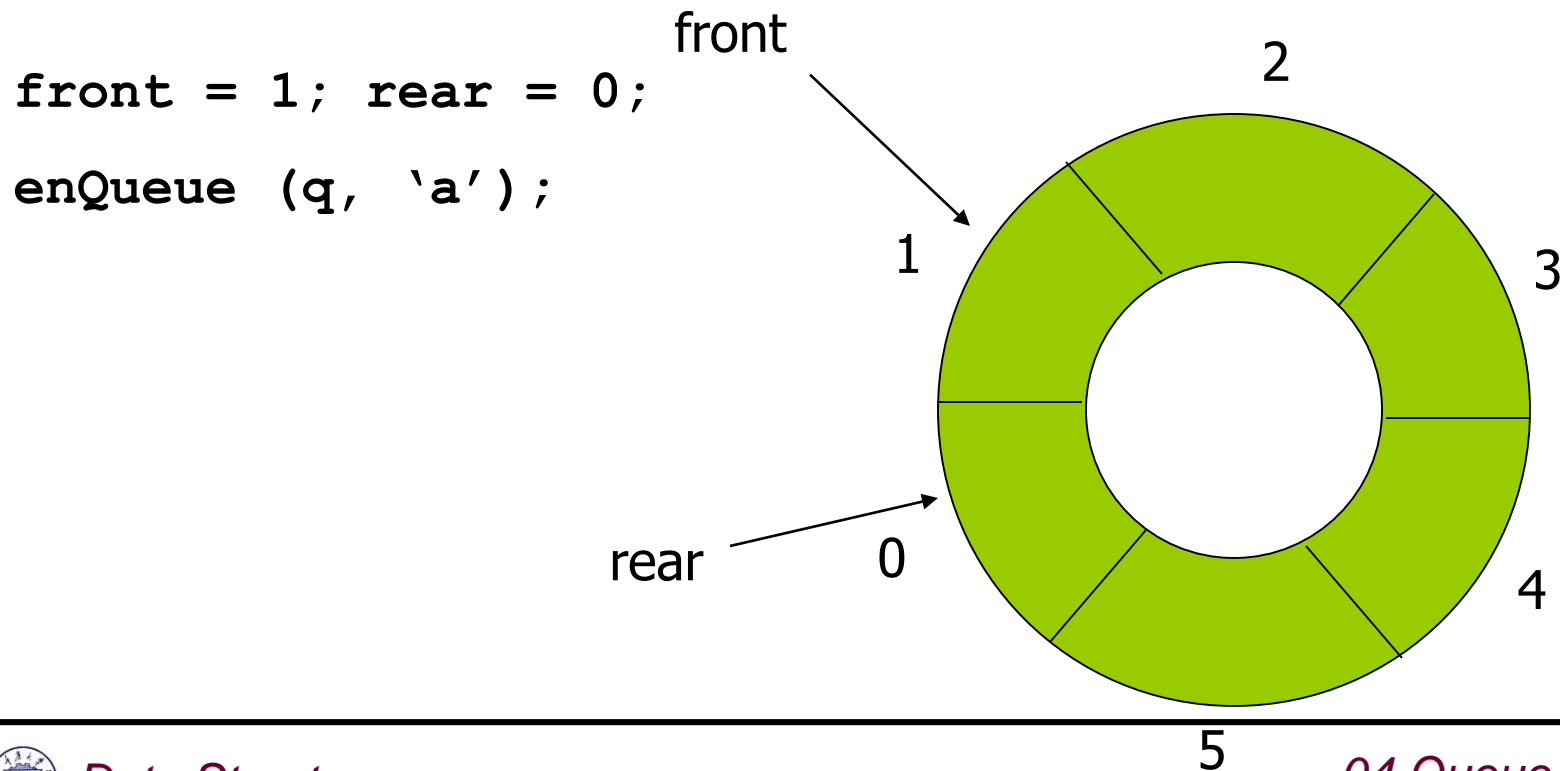
---

- How can we recognize when the circular queue is empty or full?
  - For both empty queue and full queues , the value for **rear** is one less than the **value** for front。
  - One obvious solution is to keep an explicit count of the number of elements in the queue, or a Boolean variable that indicates whether the queue is empty or not.
  - Another solution is to make the array be of size  $n+1$ , and only allow  $n$  elements to be stored.



# Circular Queue Demo

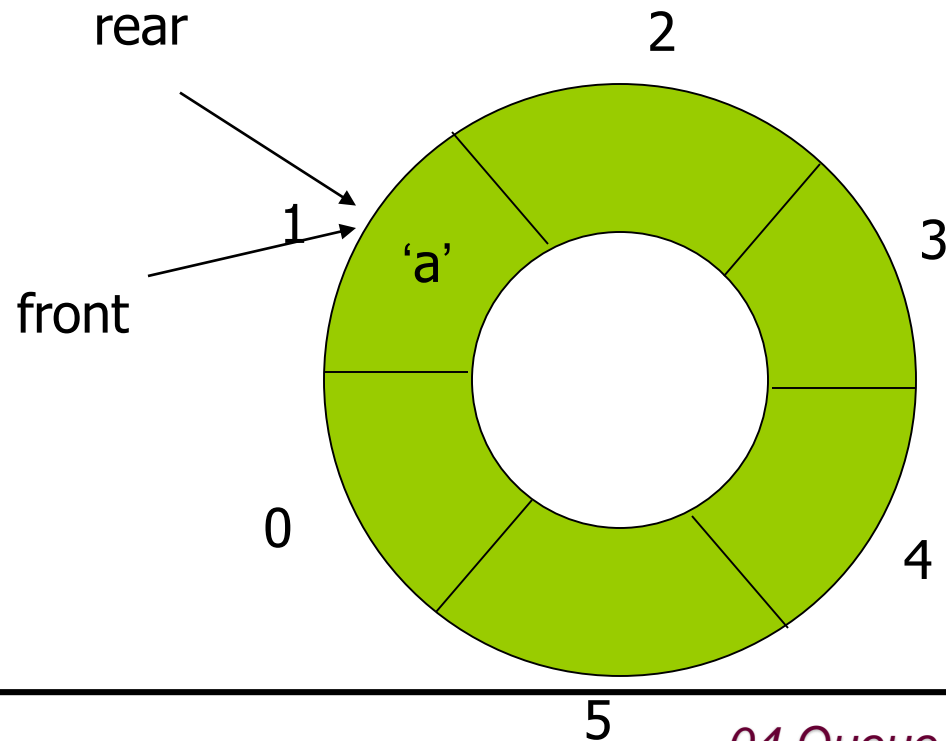
- A refinement of the lazy approach is the circular queue



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

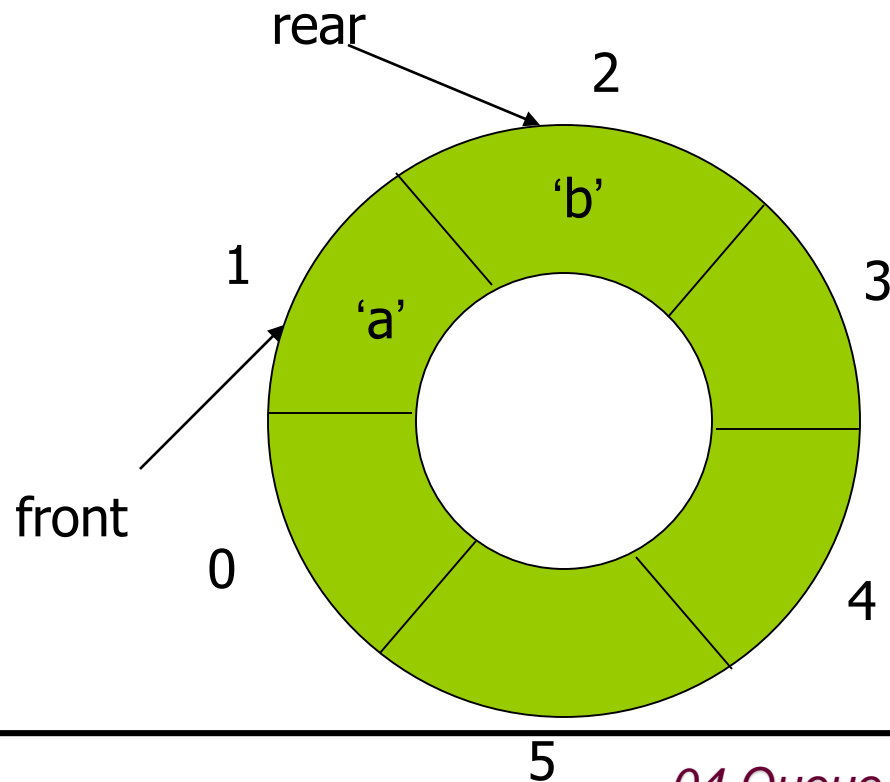
```
front = rear = 1;  
enqueue (q, 'a');  
enqueue (q, 'b');
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 1; rear = 2;  
enqueue (q, 'a');  
enqueue (q, 'b');  
enqueue (q, 'c');
```





# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

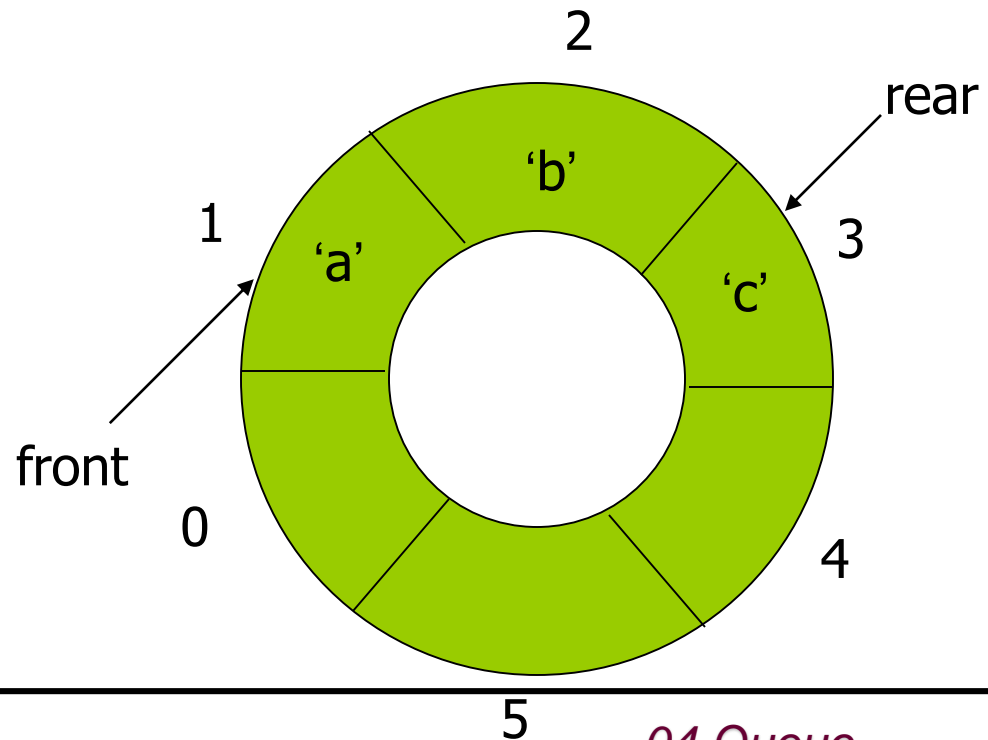
```
front = 1; rear = 3;
```

```
enqueue (q, 'a');
```

```
enqueue (q, 'b');
```

```
enqueue (q, 'c');
```

```
enqueue (q, 'd');
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 1; rear = 4;
```

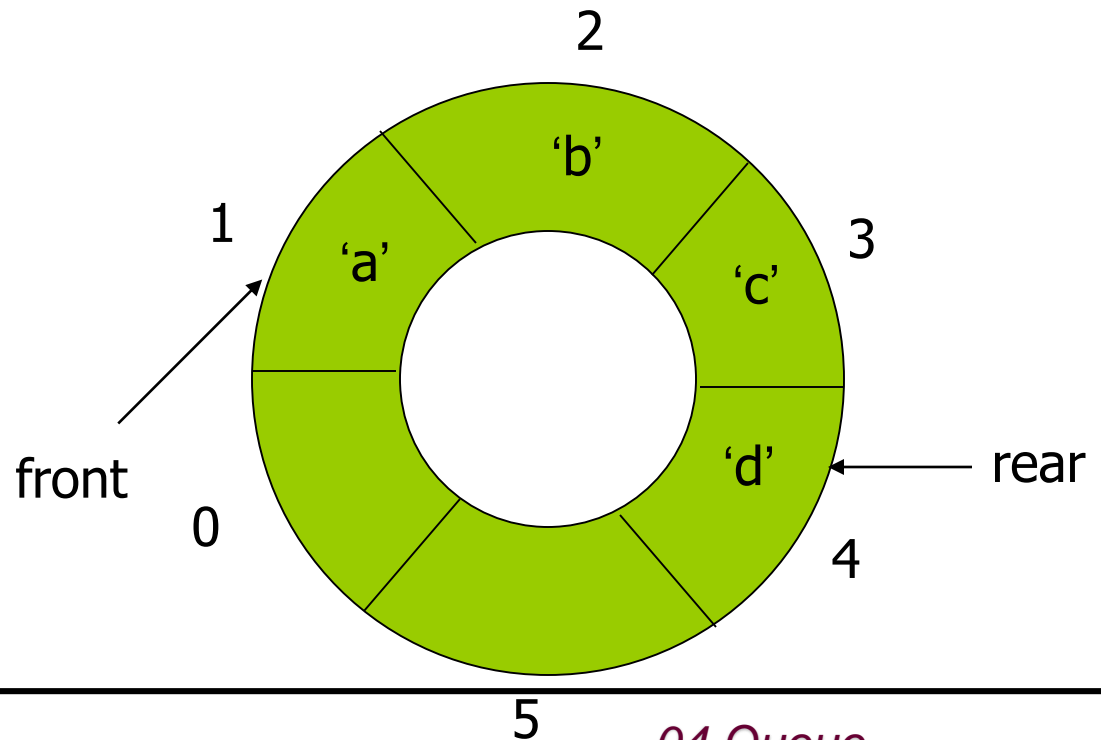
```
enqueue (q, 'a');
```

```
enqueue (q, 'b');
```

```
enqueue (q, 'c');
```

```
enqueue (q, 'd');
```

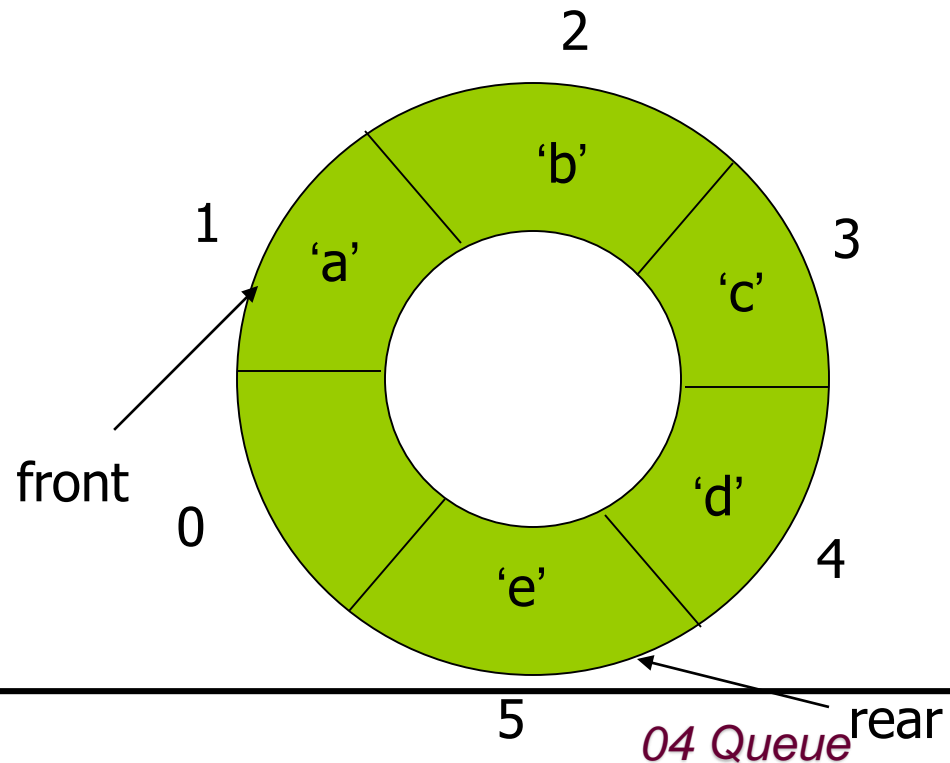
```
enqueue (q, 'e');
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 1; rear = 5;  
enqueue (q, 'a');  
enqueue (q, 'b');  
enqueue (q, 'c');  
enqueue (q, 'd');  
enqueue (q, 'e');  
enqueue (q, 'f'); ???
```

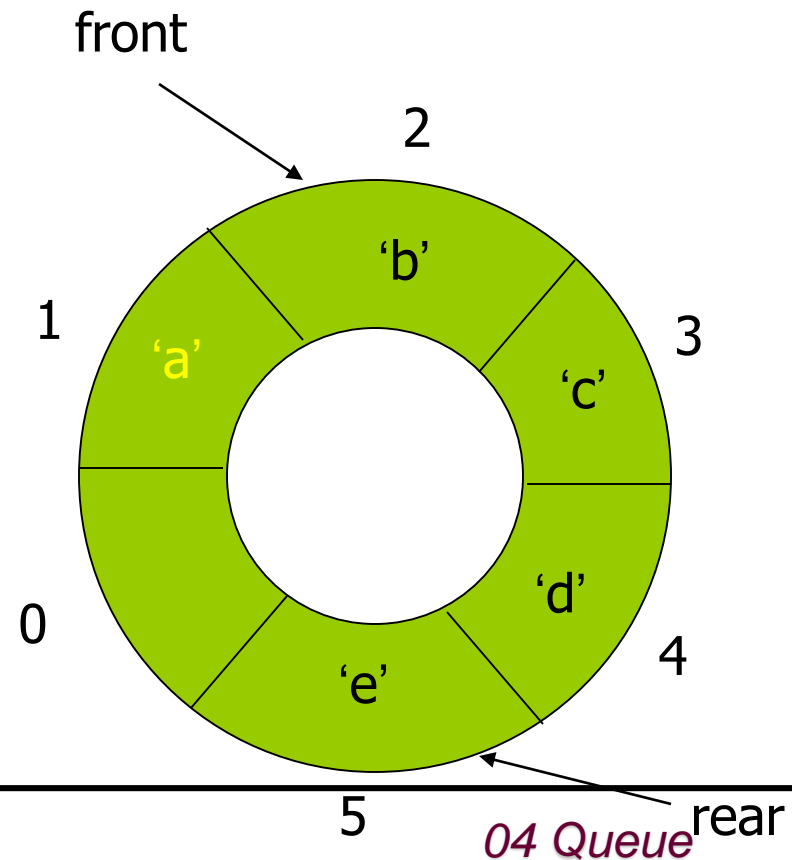


# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 2; rear = 5;
```

```
deQueue (q);
```



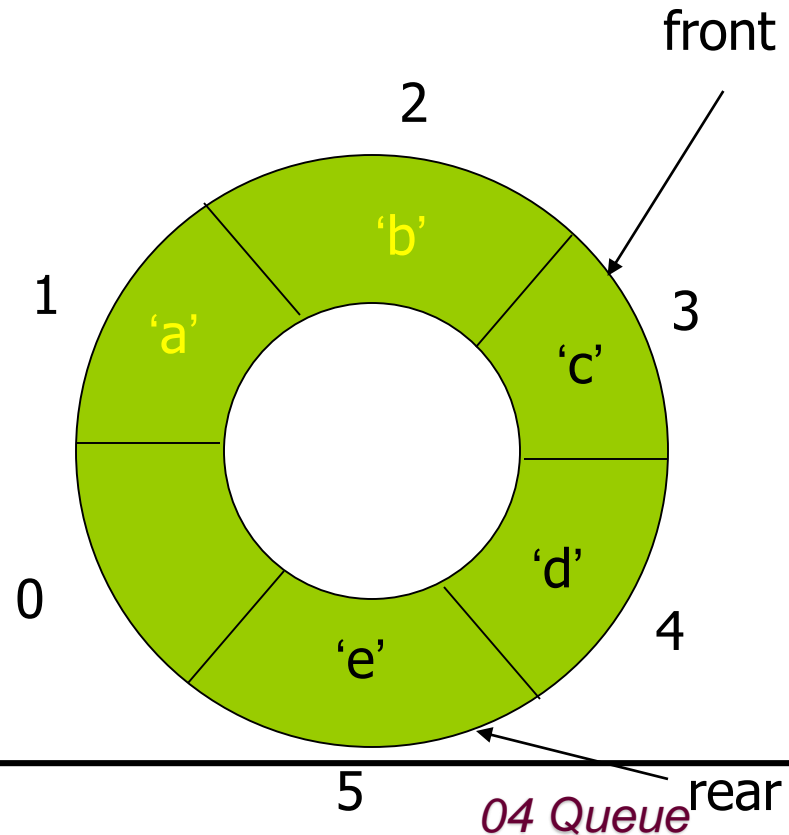
# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 3; rear = 5;
```

```
deQueue (q);
```

```
deQueue (q);
```



# Circular Queue Demo

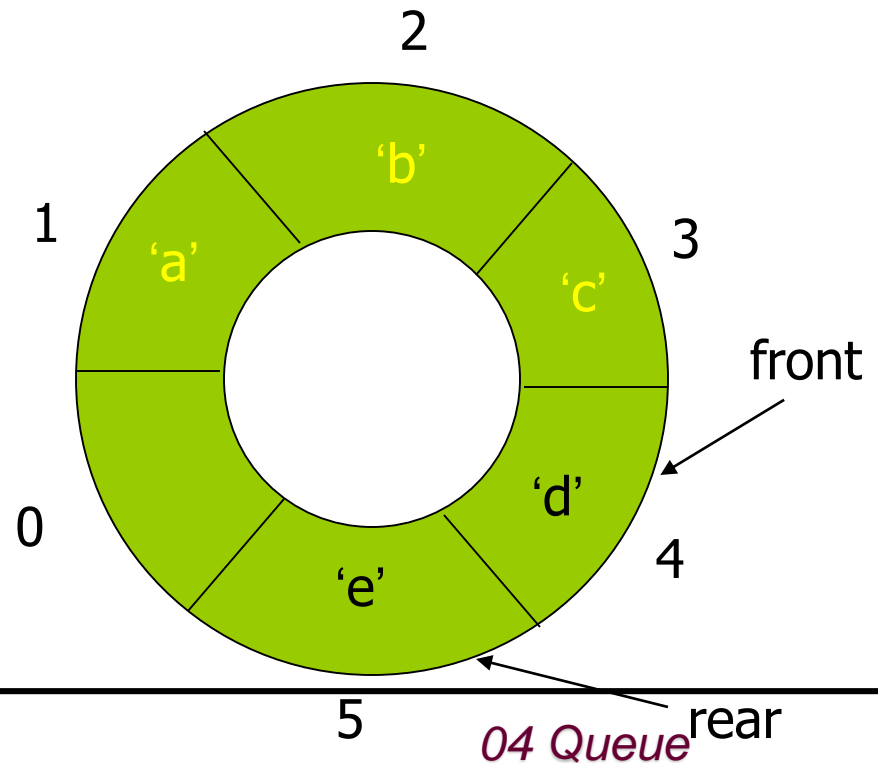
- A refinement of the lazy approach is the circular queue

```
front = 4; rear = 5;
```

```
deQueue (q);
```

```
deQueue (q);
```

```
deQueue (q);
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

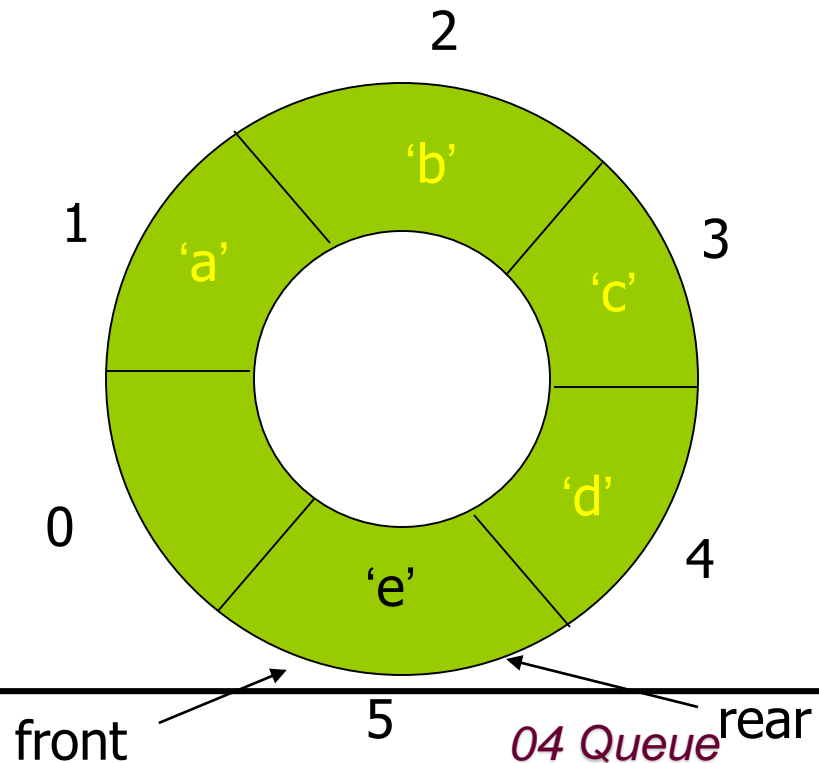
```
front = 5; rear = 5;
```

```
deQueue (q);
```

```
deQueue (q);
```

```
deQueue (q);
```

```
deQueue (q);
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 0; rear = 5;
```

```
deQueue (q) ;
```

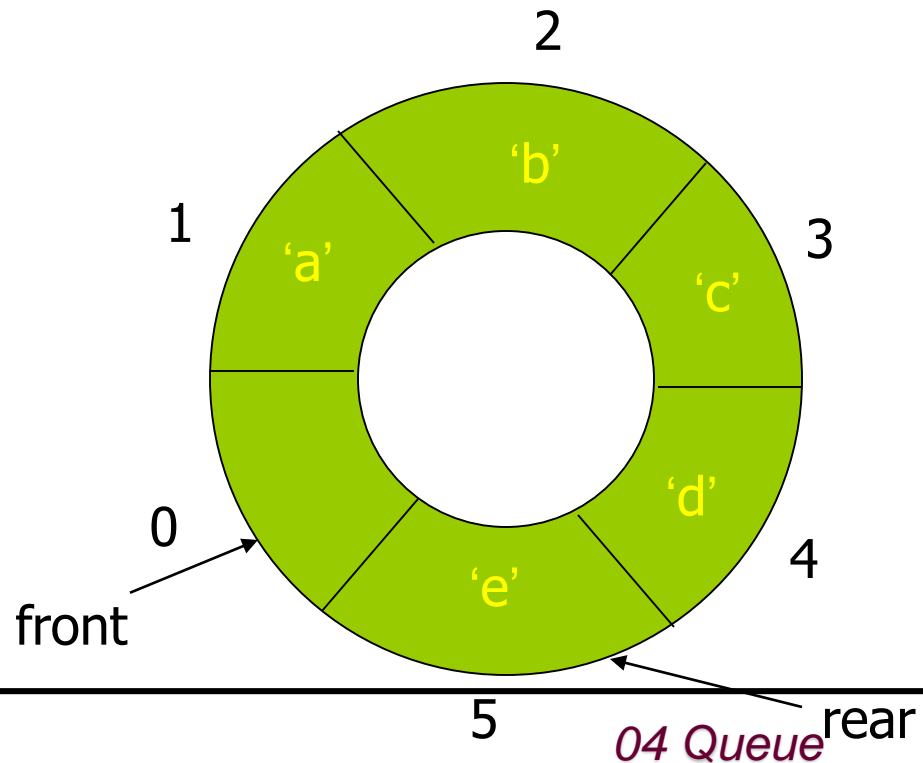
```
deQueue (q) ;
```

```
deQueue (q) ;
```

```
deQueue (q) ;
```

```
deQueue (q) ;
```

```
deQueue (q) ; ????
```



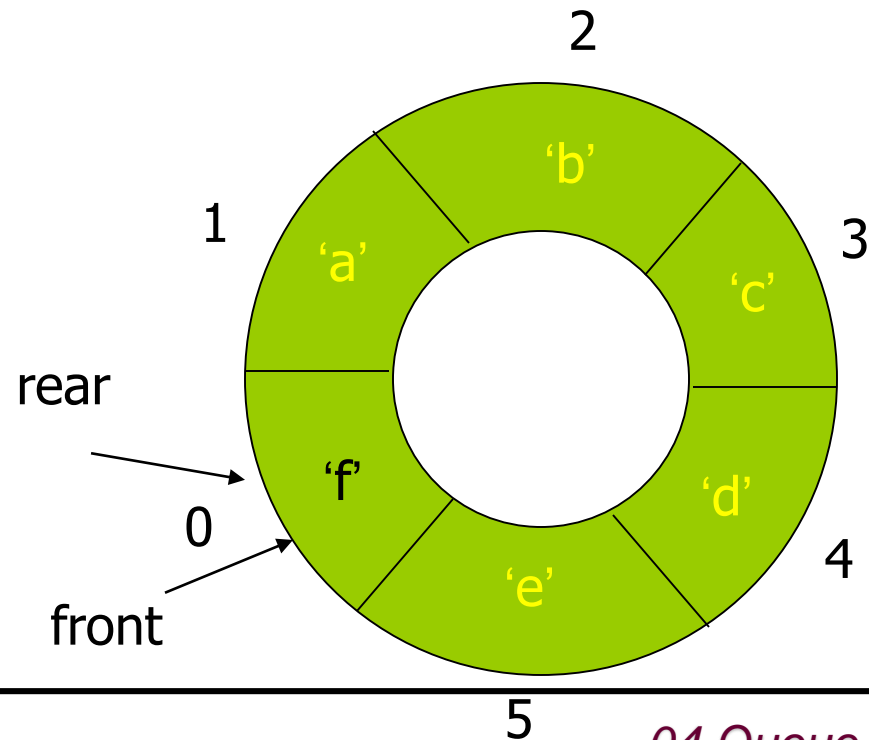


# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 0; rear = 0;
```

```
enqueue (q, 'f');
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
front = 0; rear = 4;
```

```
enqueue (q, 'f');
```

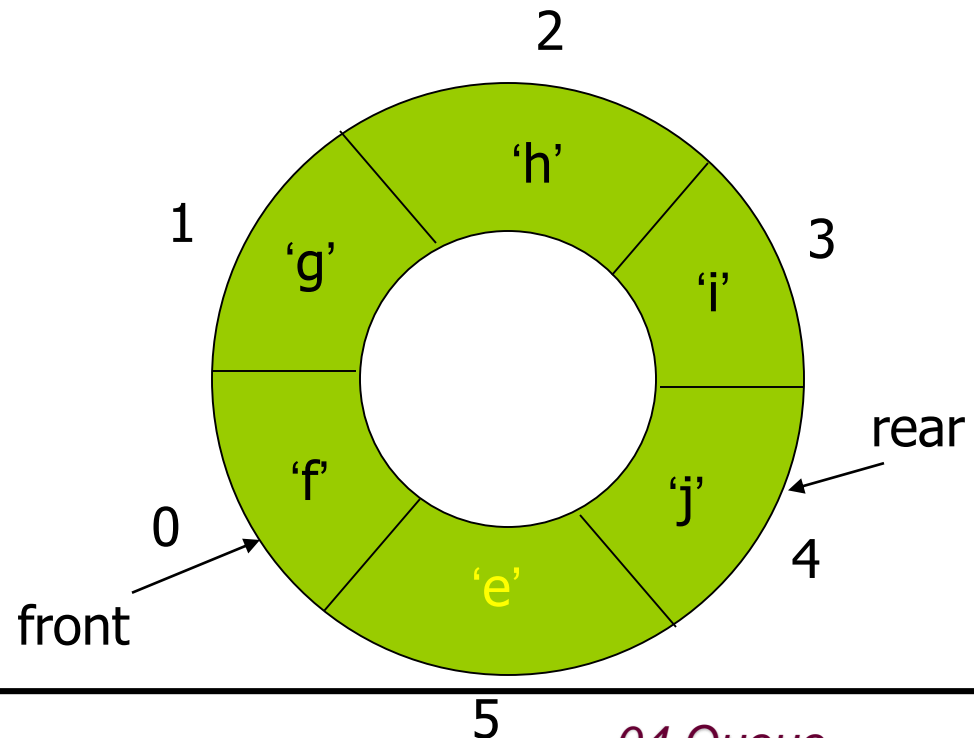
```
enqueue (q, 'g');
```

```
enqueue (q, 'h');
```

```
enqueue (q, 'i');
```

```
enqueue (q, 'j');
```

```
enqueue (q, 'k'); ???
```



# Circular Queue Demo

- A refinement of the lazy approach is the circular queue

Empty:

```
front == (rear+1) % N;
```

Full:

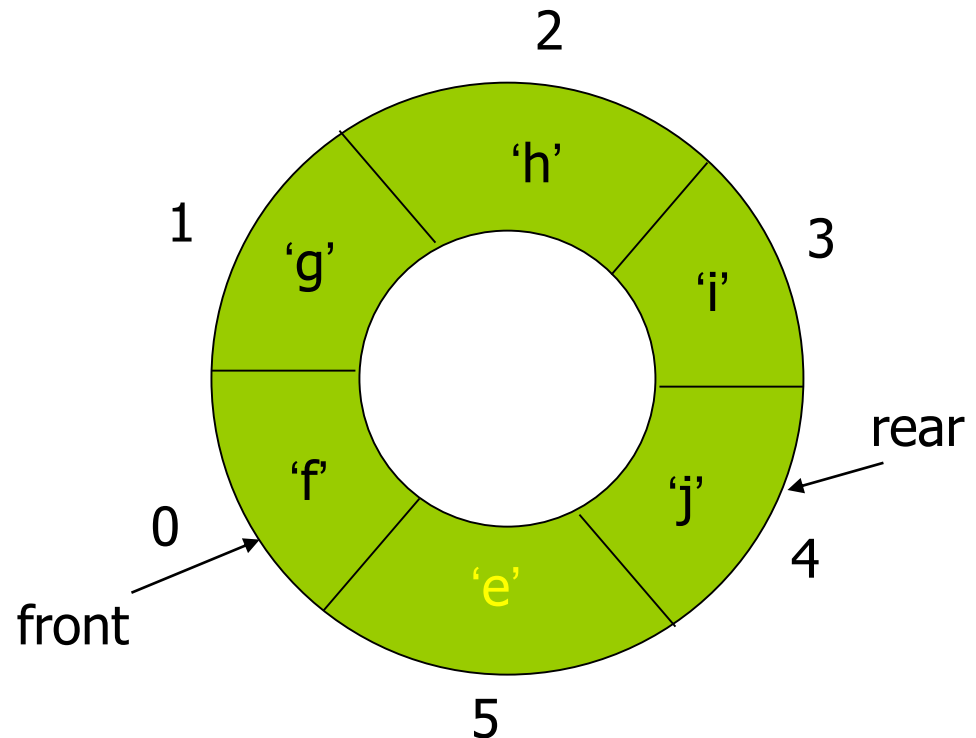
```
front == (rear+2) % N;
```

enqueue:

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

dequeue:

```
front = (front+1) % N;
```



# Array-based Queue Implementation

---

```
template <typename E> class AQueue: public Queue<E> {
private:
    int maxSize; // Maximum size of queue
    int front; // Index of front element
    int rear; // Index of rear element
    E *listArray; // Array holding queue elements
public:
    AQueue(int size =defaultSize) { // Constructor
        // Make list array one position larger for empty slot
        maxSize = size+1;
        rear = 0; front = 1;
        listArray = new E[maxSize];
    }
    ~AQueue() { delete [] listArray; } // Destructor
```



# Array-based Queue Implementation

---

```
void clear() { rear = 0; front = 1; } // Reinitialize
void enqueue(const E& it) { // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full")
    rear = (rear+1) % maxSize; // Circular increment
    listArray[rear] = it;
}
E dequeue() { // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize; // Circular increment
    return it;
}
```



# Array-based Queue Implementation

---

```
const E& frontValue() const { // Get front value
    Assert(length() != 0, "Queue is empty");
    return listArray[front];
}

virtual int length() const // Return length
{ return ((rear+maxSize) - front + 1) % maxSize; }
};
```



# Linked Queues Implementation

---

- ❑ The linked queue implementation is a straightforward adaptation of the linked list.
- ❑ On initialization, the **front** and **rear** pointers will point to the header node, and front will always point to the header node while rear points to the true last link node in the queue.

# Linked Queues Implementation

---

```
template <typename E> class LQueue: public Queue<E> {  
private:  
    Link<E>* front; // Pointer to front queue node  
    Link<E>* rear;  // Pointer to rear queue node  
    int size; // Number of elements in queue  
public:  
    LQueue(int sz =defaultSize) // Constructor  
        { front = rear = new Link<E>(); size = 0; }  
    ~LQueue() { clear(); delete front; } // Destructor
```





# Linked Queues Implementation

---

```
void clear() { // Clear queue
    while(front->next != NULL) { // Delete each link node
        rear = front;
        delete rear;
    }
    rear = front;
    size = 0;
}
```

```
void enqueue(const E& it) { // Put element on rear
    rear->next = new Link<E>(it, NULL);
    rear = rear->next;
    size++;
}
```



# Linked Queues Implementation

---

```
E dequeue() { // Remove element from front
    Assert(size != 0, "Queue is empty");
    E it = front->next->element; // Store dequeued value
    Link<E>* ltemp = front->next; // Hold dequeued link
    front->next = ltemp->next; // Advance front
    if (rear == ltemp) rear = front; // Dequeue last element
    delete ltemp; // Delete link
    size--;
    return it; // Return element value
}
```

# Linked Queues Implementation

---

```
const E& frontValue() const { // Get front element
    Assert(size != 0, "Queue is empty");
    return front->next->element;
}

virtual int length() const { return size; }
};
```

# Comparison of Array-Based and Linked Queues

---

- ❑ All member functions for both the array-based and linked queue implementations require constant time.
- ❑ The space comparison issues are the same as for the equivalent stack implementations.

# 思考题

- 在AQueue<E>中增加成员函数min(), 返回当前在队列中的所有元素的最小值。调用min, enqueue及dequeue的时间复杂度都是O(1)。

## 单调队列 (monotone queue)

- 与FIFO队列一样, 队列中的元素的先后顺序与入队顺序保持一致
- 在队列中元素的大小递增 (非递减) 排列
- 队头是队列中所有元素的最小元素
- 按入队顺序逐个出队, 但返回的是队列中所有元素 (含出队元素) 的最小值



# Array-based Queue Implementation

```
template <typename E> class MonoQueue : public Queue<E> {
```

Private:

```
    int maxSize; // Maximum size of queue
```

---

```
    int front; // Index of front element
```

```
    int rear; // Index of rear element
```

```
    E *listArray; // 元素数组，大小递增（非递减）排列
```

```
    int *enqueueTime; //记录listArray中各元素的入队时间
```

```
    int m_entime; //队列中所有元素的入队时间的最小值
```

Public:

```
    MonoQueue(int size =defaultSize) { // Constructor
```

```
        maxSize = size+1;
```

```
        listArray = new E[maxSize];
```

```
        enqueueTime = new int[maxSize];
```

```
        clear();
```

```
    }
```

```
    ~MonoQueue() { // Destructor
```

```
        delete [] listArray;
```

```
        delete [] enqueueTime;
```

# Array-based Queue Implementation

```
void clear() {  
    rear = 0; front = 1;  
    enqueueTime[rear] = m_etime = 0;  
} // Reinitialize
```

```
void enqueue(const E& it) { // Put "it" in queue  
    int last = enqueueTime[rear]; //记录最后入队元素的入队时间  
    while(front != (rear+1)%maxSize && listArray[rear] > it){  
        //队列不为空，且rear所指元素比新元素大  
        rear = (rear - 1 + maxSize)%maxSize; //rear前移  
    } //从后往前除掉大于新元素的元素，保持元素大小递增（非递减）排列。
```

```
    Assert(((rear+2) % maxSize) != front, "Queue is full") //队列未滿  
    rear = (rear+1) % maxSize; // Circular increment  
    listArray[rear] = it; //插入到队列的后面，形成递增（非递减）序列  
    enqueueTime[rear] = last + 1; //新入队元素的入队时间  
}
```



# Array-based Queue Implementation

```
E dequeue() {  
    Assert(length() != 0, "Queue is empty");  
    E it = listArray[front];    //队头元素值（最小值）  
    m_etime = m_etime + 1;  
    if( m_etime == enqueueTime[front] ){  
        front = (front+1) % maxSize; // Circular increment  
    }    //如果出队元素比队头元素先入队，则不改变队头位置。  
    return it;  
}  
  
const E& frontValue() const { // 返回队头元素值，即队列中所有元素的最小值  
    Assert(length() != 0, "Queue is empty");  
    return listArray[front];  
}  
  
virtual int length() const // Return length  
    { return ((rear+maxSize) - front + 1) % maxSize; }  
};
```





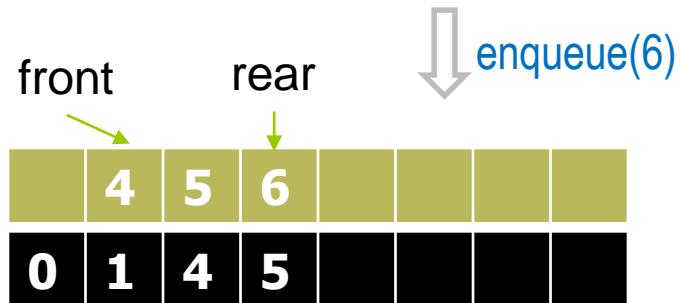
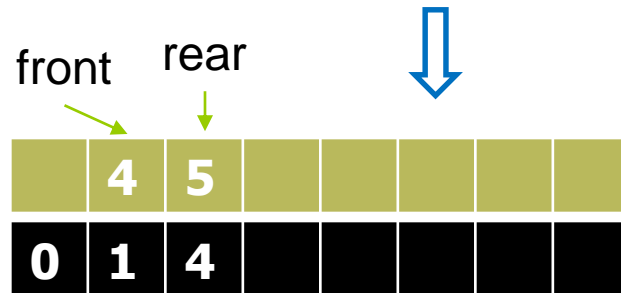
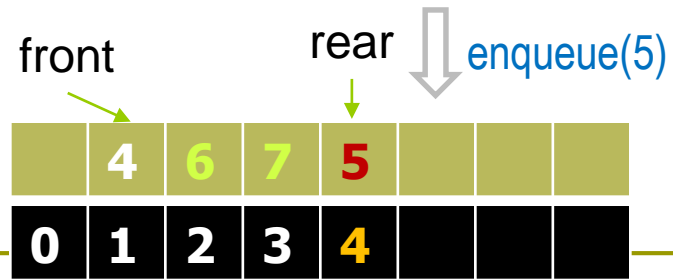
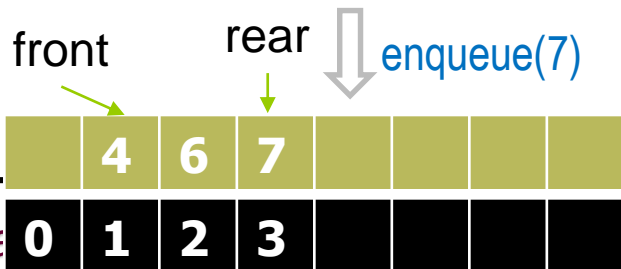
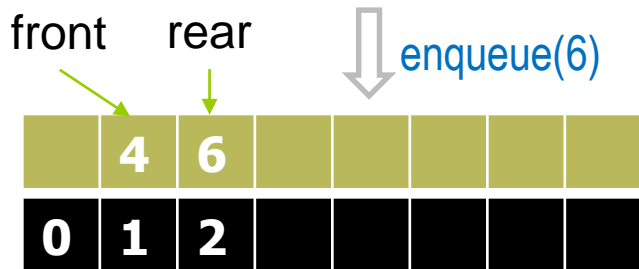
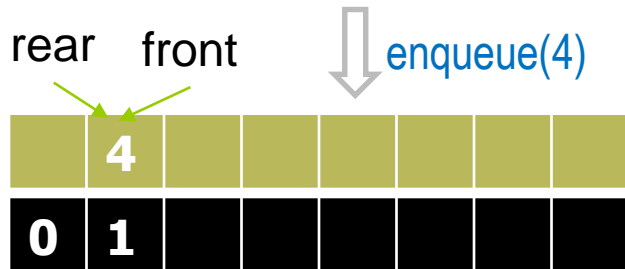
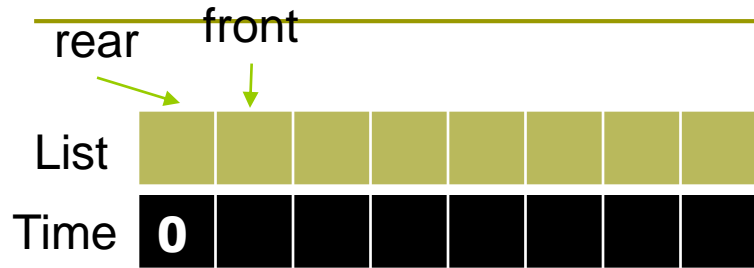
# Project

---

- You are to create programs that keep track of inventory and shipping at a store or business of your choosing. When items are made, they are put into LIFO inventory. When you ship something, it comes out of inventory and goes into a shipping queue. When an item is delivered, it is taken out of the shipping queue. You should keep a list of all delivered items. Users of your system should be able to do the following:
  1. Input an item (some name) and an amount. The item and amount should be inserted into a *stack* of inventory items.
  2. Push a button to get a list displayed of all the items and their quantities in the inventory.
  3. Push a button to take the next (top) item from inventory and put it in a shipping *queue*.
  4. Push a button to get a list displayed of all the items that are currently shipping.
  5. Push a button to indicate that a shipped item has been delivered and should be taken out of the shipping queue. Shipped items are put in and taken out FIFO.
  6. Push a button to display all the things that have been delivered.

# example

□ 入队顺序: 4, 6, 7, 5, 6, 3



# Project

---

- ❑ You are to create classes for the different data structures and nodes. You also need a driver program that takes user input and manipulates a node object and prints output. The driver should be written as a C++ application with buttons and text input and output.
- ❑ One way to do the application is to have two text fields for input, a text area for output, and buttons that say “add to inventory,” “display inventory,” etc. Think about your design and draw it out *before* you write the code. Also, draw your data structures before you write any code. Draw out the operations of taking things from one data structure and putting them into another. Think about what goes in your application and your class *before* you write any code.

