



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

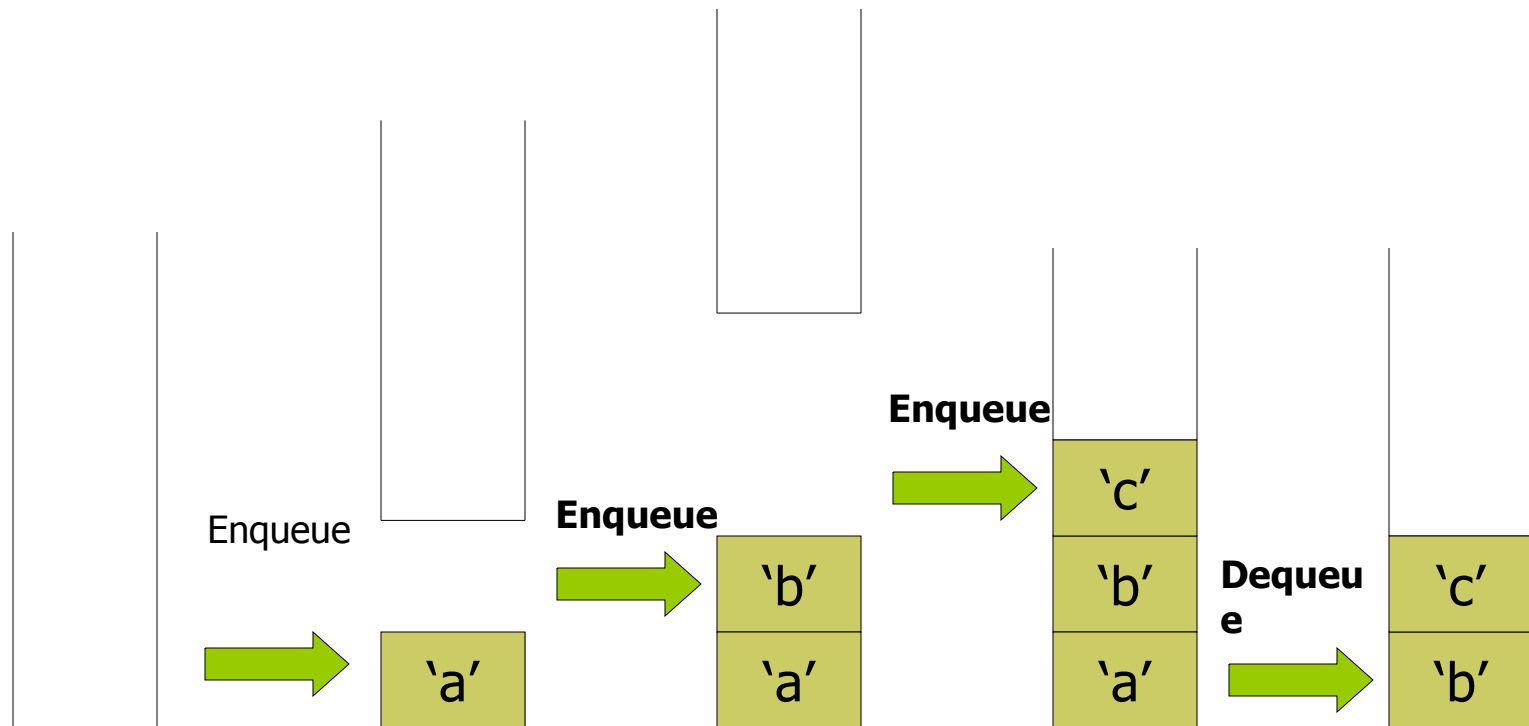
FIFO: First in, First Out

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

Notation:

- ▣ Insert: Enqueue
- ▣ Remove: 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

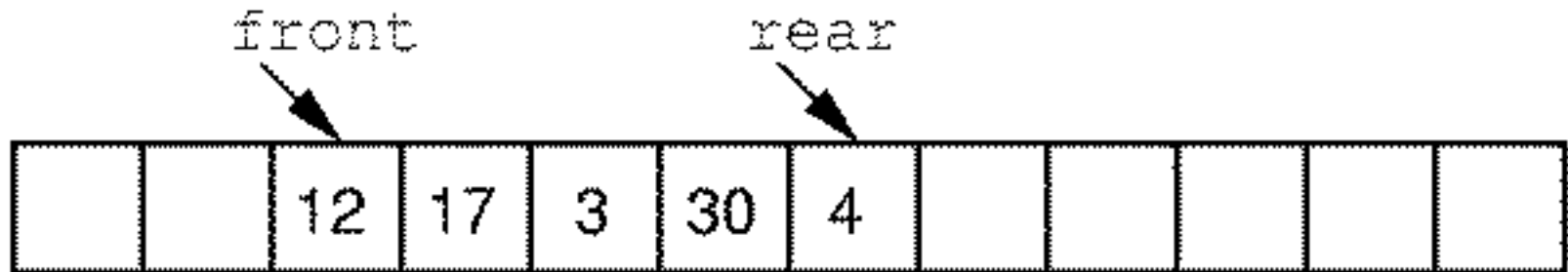
- ❑ The array-based queue is somewhat tricky to implement effectively. Assume that there are n elements in the queue.
 - If we choose the rear element of the queue to be in position 0, enqueue operations will shift the n elements currently in the queue one position in the array.
 - If instead we chose 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.

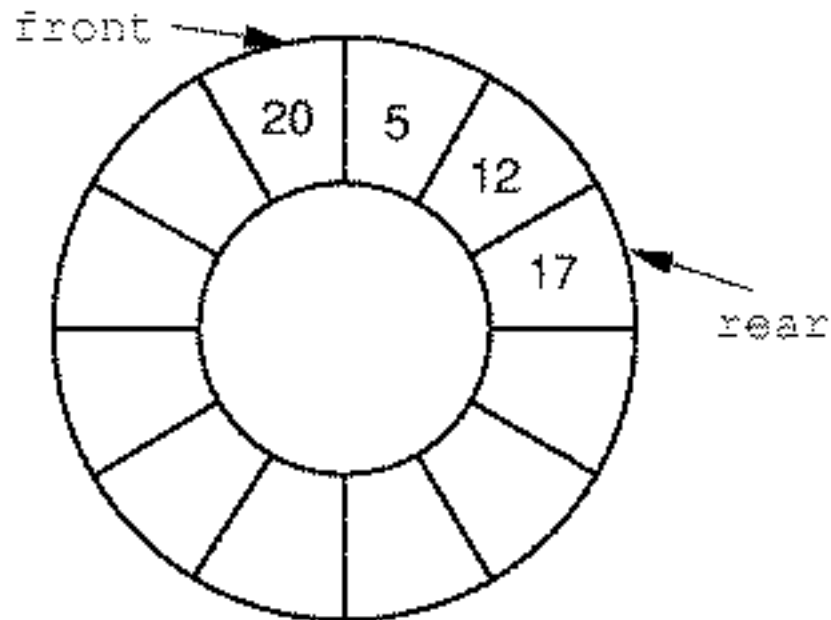


Queue Implementation (2)

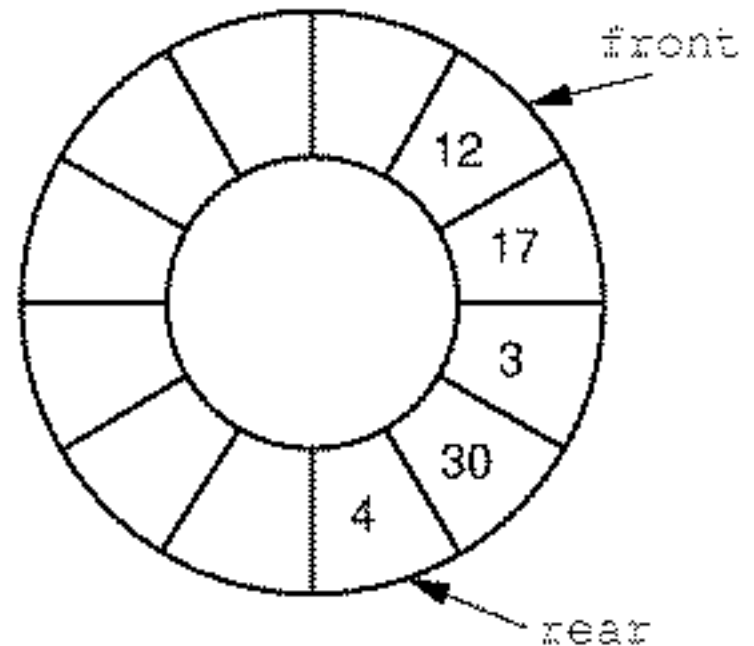
- The “drifting queue” problem can be solved by pretending that the array is circular and so allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position.



Queue Implementation (2)



(a)



(b)

Queue Implementation (2)

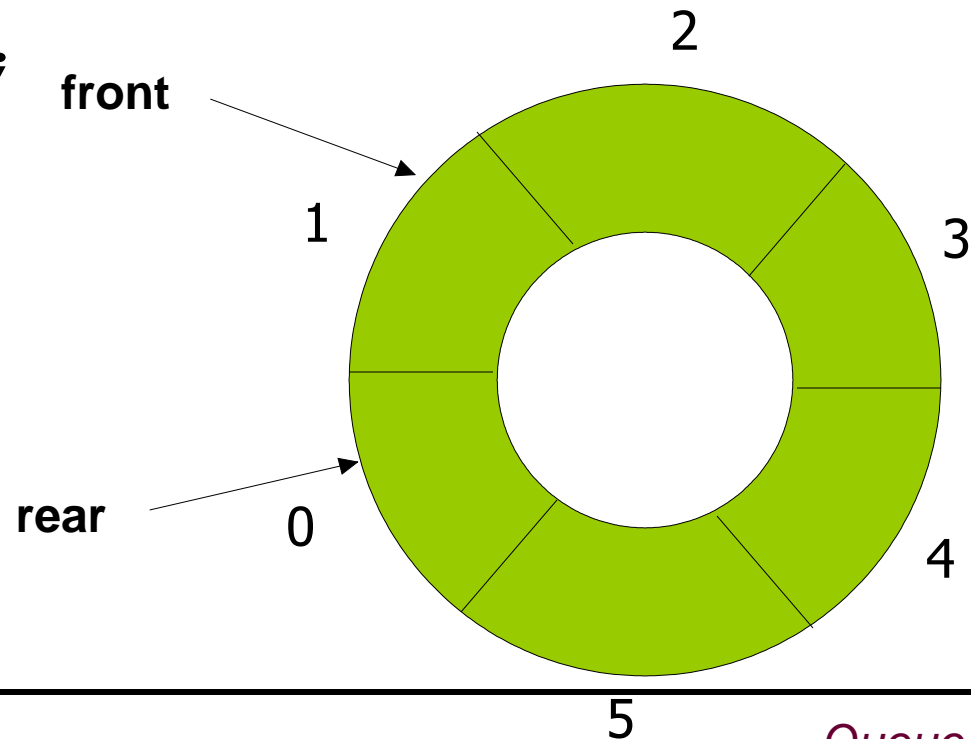
- How can we recognize when the circular queue is empty or full?
 - For both empty queue and full queues , the value for **rear** is one less than the **value** for front.
 - One obvious solution is to keep an explicit count of the number of elements in the queue, or a Boolean variable that indicates 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

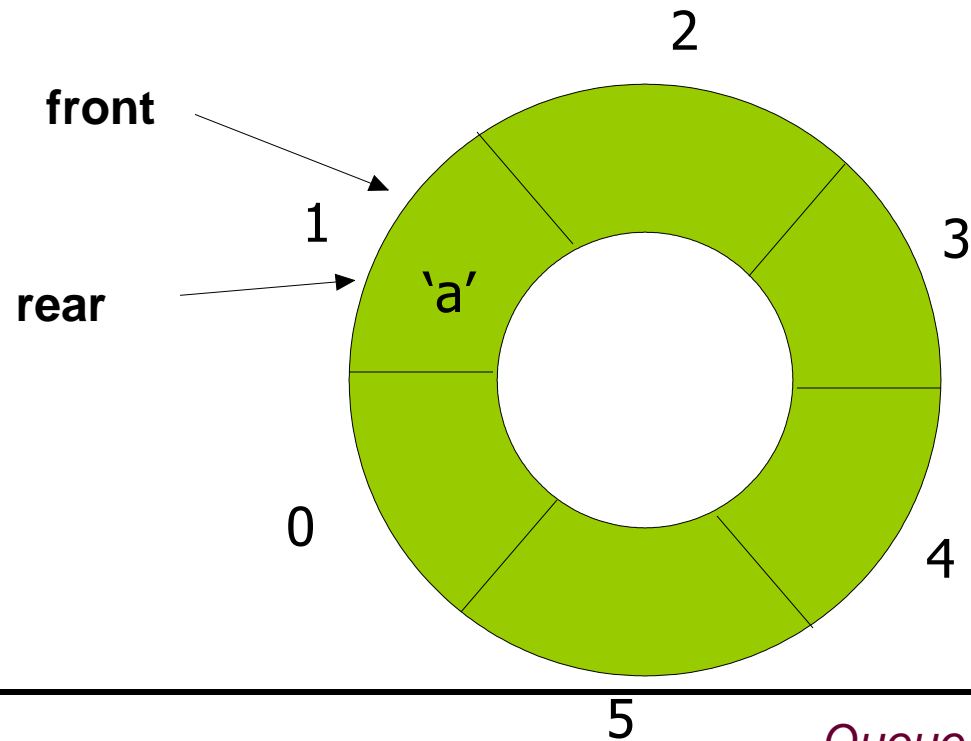
```
front = 1 ; rear = 0 ;  
enqueue ( 'a' ) ;
```



Circular Queue Demo

- A refinement of the lazy approach is the circular queue

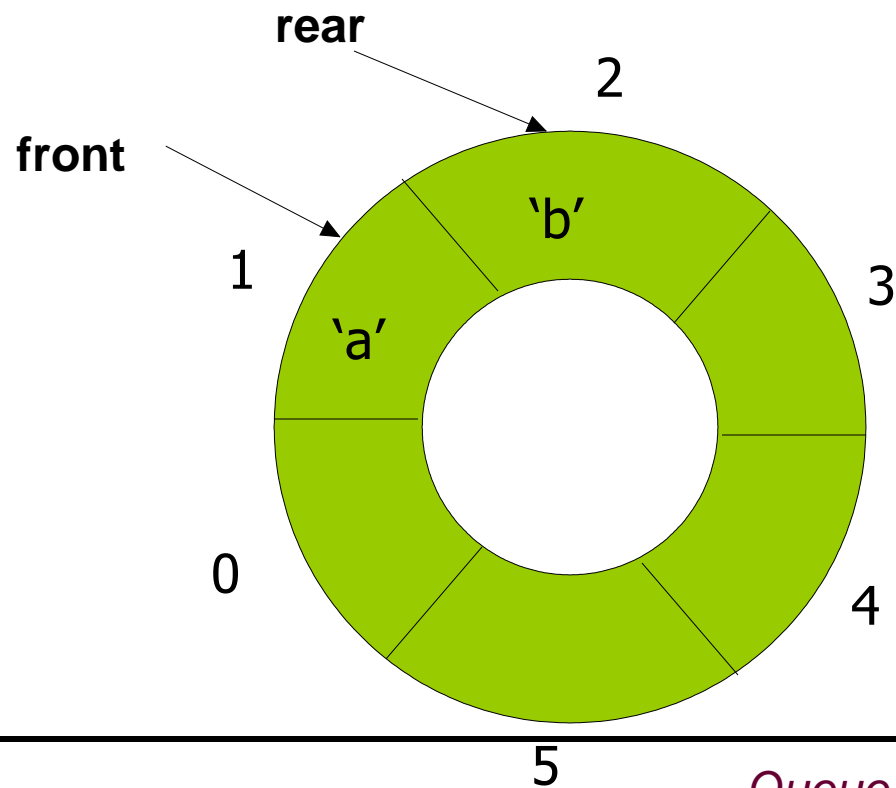
```
enQueue ('a');  
enQueue ('b');
```



Circular Queue Demo

- A refinement of the lazy approach is the circular queue

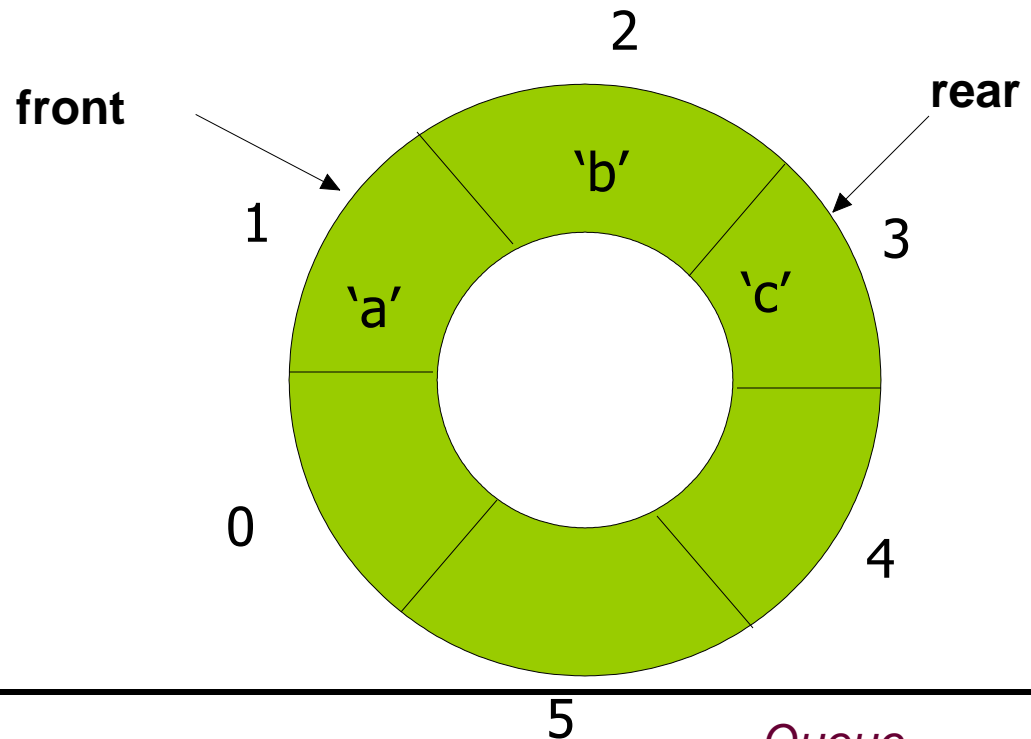
```
enQueue ('a');  
enQueue ('b');  
enQueue ('c');
```



Circular Queue Demo

- A refinement of the lazy approach is the circular queue

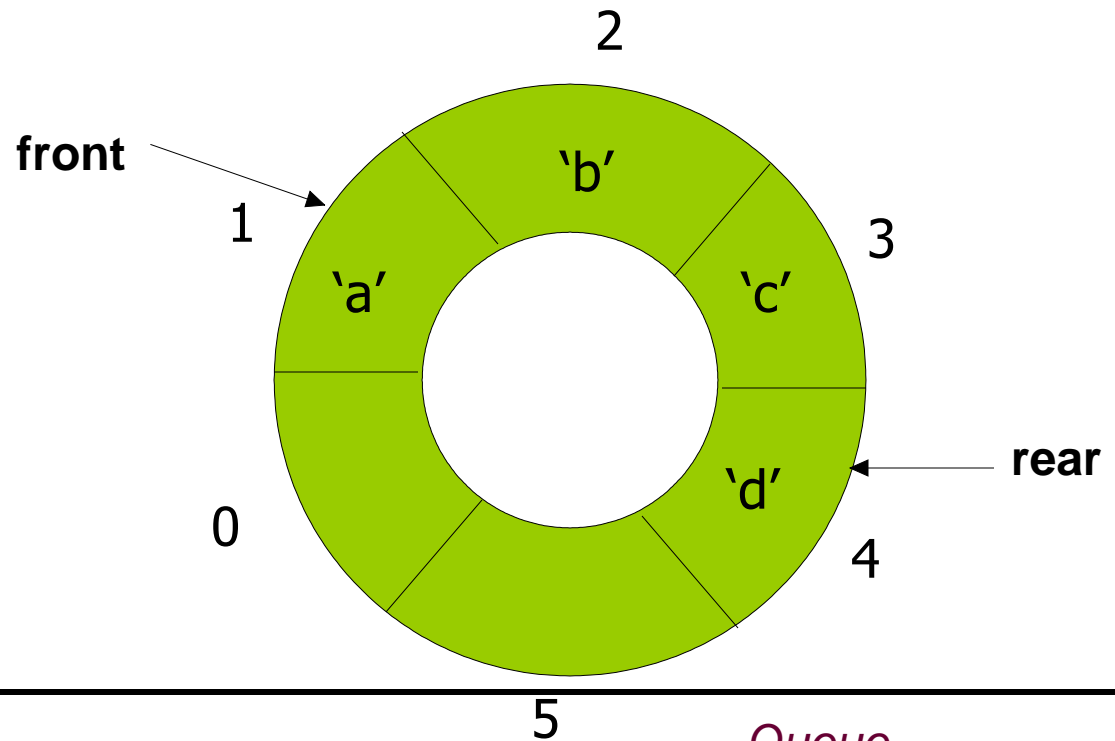
```
enqueue ('a');  
enqueue ('b');  
enqueue ('c');  
enqueue ('d');
```



Circular Queue Demo

- A refinement of the lazy approach is the circular queue

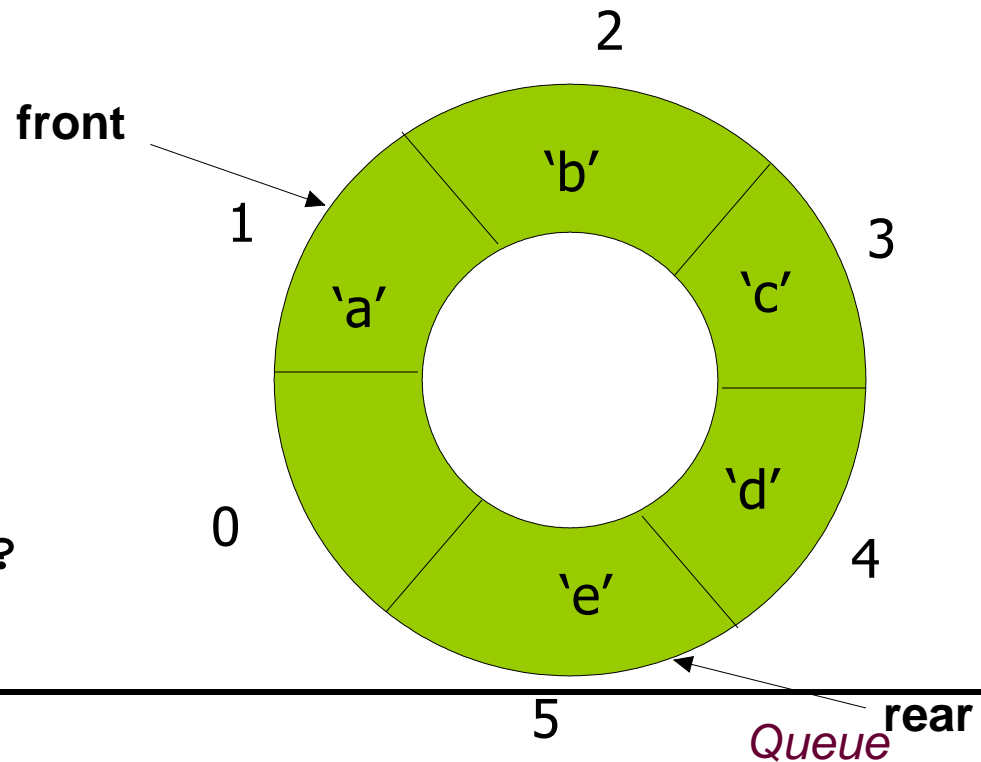
```
enqueue ('a');  
enqueue ('b');  
enqueue ('c');  
enqueue ('d');  
enqueue ('e');
```



Circular Queue Demo

- A refinement of the lazy approach is the circular queue

```
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

Empty:

`length() == 0;`

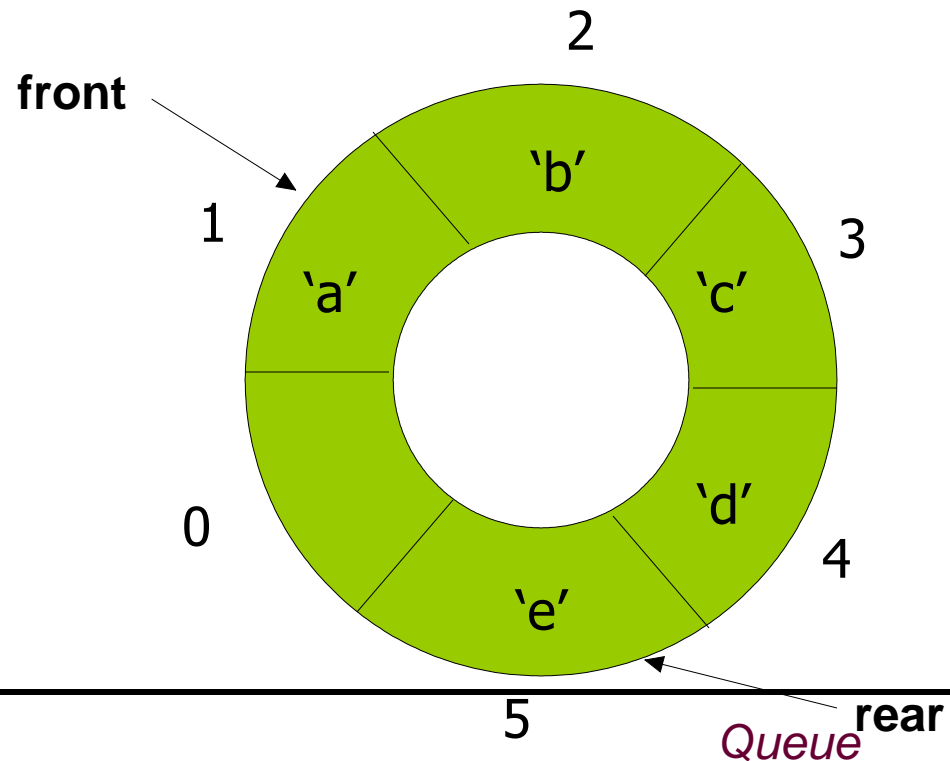
Full:

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

General Equations:

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

`rear = (rear + 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;
        front=front->next;
        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.

Homework

- 4.13
- 4.14
- 4.15



Reference

- **Data Structures and Algorithm Analysis in C++ .Third Edition.Clifford A. Shaffer(P.129-134)**



-END-

