


《数据结构与算法》课程组
重庆大学计算机学院



Data Structures & Algorithms

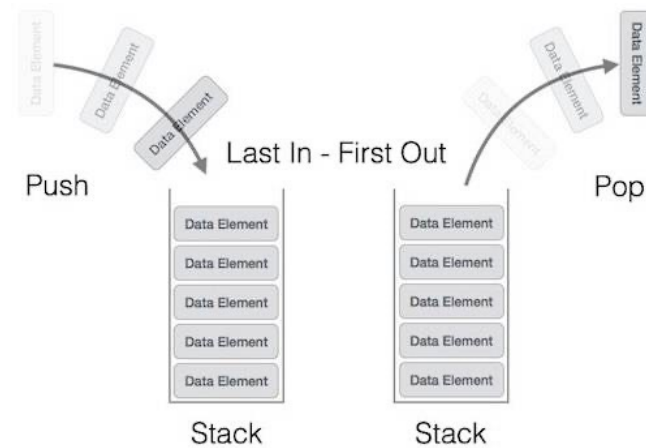




STACK AND QUEUE



6.1 Stack



Outline

- **Stack ADT**
- **Array-Based Stack**
- **Linked Stack**
- **Comparison of Array-Based and Linked Stacks**

Stacks

- The stack is a **list-like structure** in which elements may be inserted or removed from only one end, called the **top of the stack**.
- All access is restricted to the most recently inserted elements
- Basic operations are **push**, **pop**

Stacks

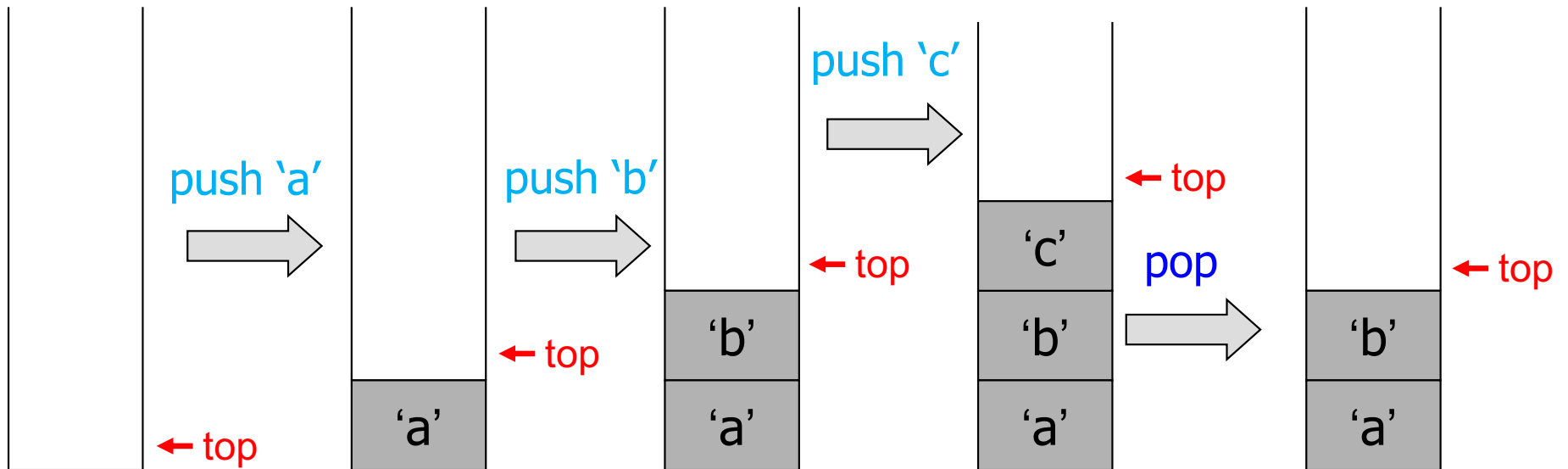
LIFO: Last In, First Out.

Restricted form of list: Insert and remove only at front of list.

Notation:

- **Insert: PUSH**
- **Remove: POP**
- **The accessible element is called TOP.**

Example: Stack of Char



empty stack

Stack ADT

// Stack abstract class

template <typename E> class Stack {

private:

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

Stack(const Stack&) {} // Protect copy constructor

public:

Stack() {} // Default constructor

virtual ~Stack() {} // Base destructor

// Reinitialize the stack. The user is responsible for

// reclaiming the storage used by the stack elements.

virtual void clear() = 0;

Stack ADT

// Push an element onto the top of the stack.

// it: The element being pushed onto the stack.

virtual void push(const E& it) = 0;

// Remove the element at the top of the stack.

// Return: The element at the top of the stack.

virtual E pop() = 0;

// Return: A copy of the top element.

virtual const E& topValue() const = 0;

// Return: The number of elements in the stack.

virtual int length() const = 0;

}

Array-Based Stack

- The array-based stack implementation is essentially a simplified version of the array-based list.
- The only important design decision to be made is **which end of the array** should represent the **top of the stack**.
 - One choice is to make the top be at **position 0** in the array. This implementation is inefficient.
(**Why?**)
 - The other choice is have the top element be at **position n** when there are n elements in the stack.

Array-Based Stack

```
// Array-based stack implementation
private:
    int size;          // Maximum size of stack
    int top;           // Index for top element
    Elem *listArray;   // Array holding elements
public:
    AStack(int size =defaultSize) // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }

    ~AStack() { delete [] listArray; } // Destructor
    void clear() { top = 0; } // Reinitialize

    void push(const E& it) { // Put "it" on stack
        Assert(top != maxSize, "Stack is full");
        listArray[top++] = it;
    }

    E pop() { // Pop top element
        Assert(top != 0, "Stack is empty");
        return listArray[--top];
    }

    const E& topValue() const { // Return top element
        Assert(top != 0, "Stack is empty");
        return listArray[top-1];
    }

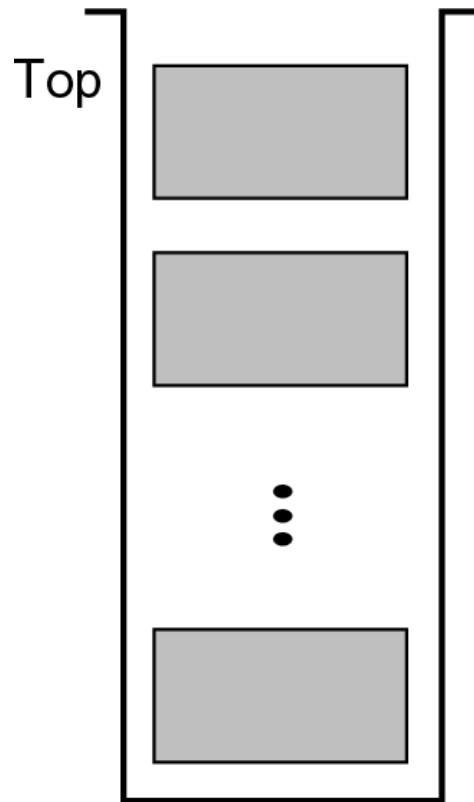
    int length() const { return top; } // Return length
};
```

Array-Based Stack

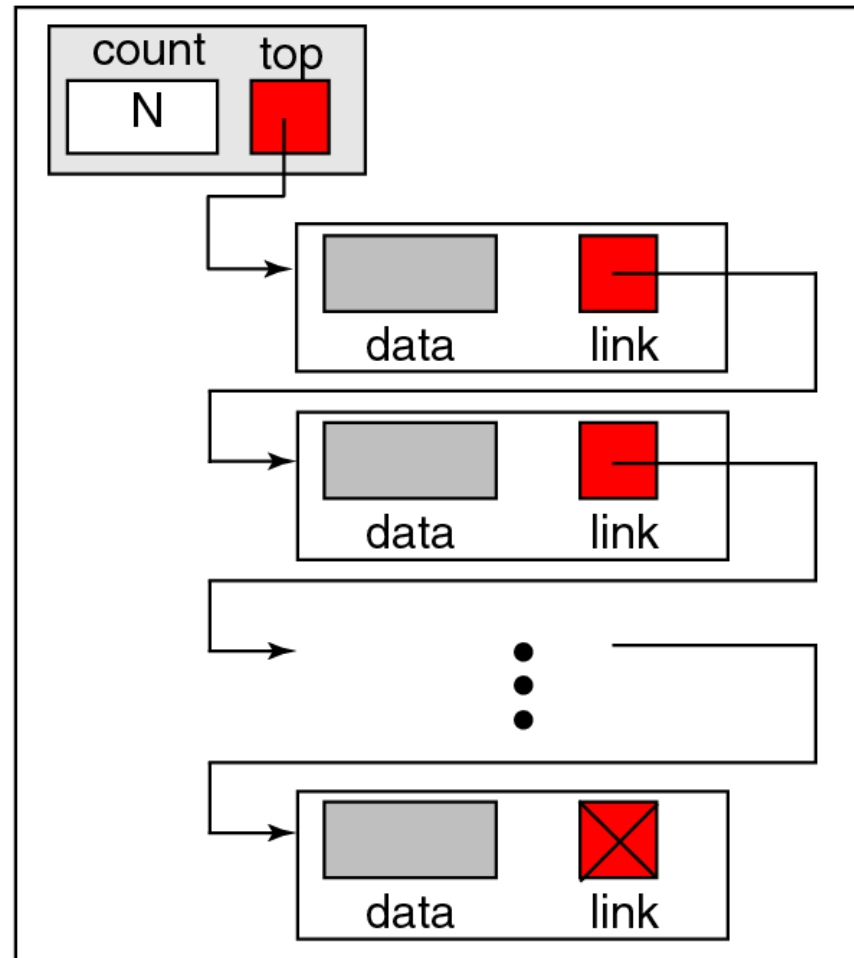
Question:

- Which end is the top?
- What is the cost of the operations?

Linked Stack



Conceptual view



Linked list implementation

Linked Stack

- The linked stack implementation is quite simple. For instance, the **freelist** is an example of a linked stack.
- Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements.
- The only data member is **top**, a pointer to the first (top) link node of the stack.

Linked Stack

```
// Linked stack implementation
private:
    Link<Elem>* top; // Pointer to first elem
    int size;        // Count number of elems
public:
    LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }
    ~LStack() { clear(); } // Destructor
```

Linked Stack

```
void clear() { // Reinitialize
    while (top != NULL) { // Delete link nodes
        Link<E>* temp = top;
        top = top->next;
        delete temp;
    }
    size = 0;
}

void push(const E& it) { // Put "it" on stack
    top = new Link<E>(it, top);
    size++;
}

E pop() { // Remove "it" from stack
    Assert(top != NULL, "Stack is empty");
    E it = top->element;
    Link<E>* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
}
```


Linked Stack

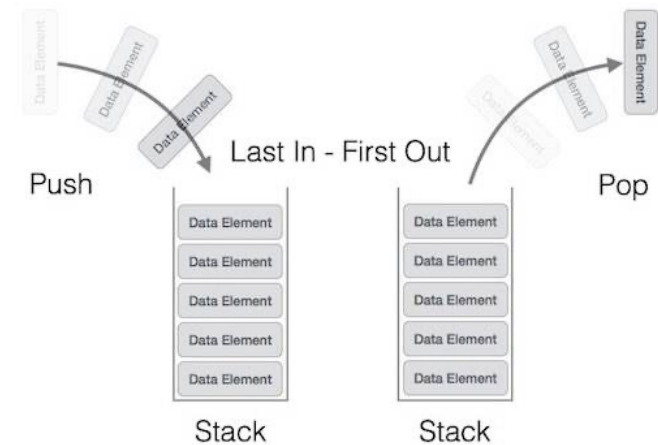
```
const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
}
int length() const { return size; } // Return length
};
```

- What is the cost of the operations?
- How do space requirements compare to the array-based stack implementation?

Comparison of Array-Based and Linked Stacks

- All operations for the array-based and linked stack implementations take **constant time**
- Total space required: the analysis is similar to that done for list implementations
- When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. **This can be done by using a single array to store two stacks.**

6.2 Stack Applications



Data conversion

- A simple algorithm uses the following law:

$$N = (N / d) \times d + N \% d$$

eg: Convert decimal to binary

$$(100)_{10} = (1100100)_2$$

- divide decimal number by 2 continuously
 - record the remainders until the quotient becomes 0
 - print the remainders in backward order.
- **Stack is an ideal structure to implement this process.**
 - How to implement it?

2	100	
2	50	0
2	25	0
2	12	1
2	6	0
2	3	0
2	1	1
	0	1

Balancing Symbols

- Stack can be used to check a program for balanced symbols (such as { }, (), []).
- Example: {()} is legal, {(}) is not (so simply counting symbols does not work).
- When a **closing symbol** is seen, it matches the most recently seen **unclosed opening symbol**. Therefore, a stack will be appropriate.

Example

- How do we know the following parentheses are nested correctly ?

$$7-((X*((X+Y)/(J-3))+Y)/(4-2.5))$$

1. There are an equal number of right and left parentheses.
2. Every right parenthesis is preceded by a matching left parenthesis.

$((A+B)$ or $A+B($ **violate 1**

$)A+B(-C$ or $(A+B))- (C+D$ **violate 2**

Balancing Symbols

- Make an empty stack. Read characters until end of file.
- If the character is an **opening symbol**, **push** it onto stack.
- If it is a **closing symbol**, then if the stack is empty, report an error. Otherwise, **pop** the stack.
- If the symbol popped is not the corresponding opening symbol, then report an error.
- At the end of file, if the stack is not empty, report an error.

Evaluating RPN Expressions

Notations for arithmetic expressions:

- **Infix notation:** operators written between the operands
- **Prefix notation :** operators written before the operands
- **Postfix notation (RPN):** operators written **after** the operands

- **Examples:**

INFIX

$A + B$

$A * B + C$

$A * (B + C)$

$A - (B - (C - D))$

$A - B - C - D$

RPN (POSTFIX)

$A B +$

$A B * C +$

$A B C + *$

$A B C D - - -$

$A B - C - D -$

PREFIX

$+ A B$

$+ * A B C$

$* A + B C$

$- A - B - C D$

$- - - A B C D$

Evaluating RPN Expressions

- "By hand": Underlining technique:
 1. Scan the expression from left to right to find an operator.
 2. Locate ("underline") the last two preceding operands and combine them using this operator.
 3. Repeat until the end of the expression is reached.

- Example:

$2\ 3\ 4\ +\ 5\ 6\ -\ -\ *$
→ $2\ \underline{3\ 4}\ +\ 5\ 6\ -\ -\ *$
→ $2\ 7\ 5\ 6\ -\ -\ *$
→ $2\ 7\ \underline{5\ 6}\ -\ -\ *$
→ $2\ 7\ -1\ -\ *$
→ $2\ \underline{7\ -1}\ -\ *$
→ $2\ 8\ * \rightarrow \underline{2\ 8}\ * \rightarrow 16$

Evaluating RPN Expressions

- **STACK ALGORITHM**

Input: An RPN expression.

Output: A **stack** whose top element is the value of RPN expression
(unless an error occurred).

1. Initialize an empty stack.
2. Repeat the following until the end of the expression is encountered:
 - a. Get next **token** in the RPN expression.
(token: constant, variable or arithmetic operator)
 - b. If token is an **operand**, push it onto the stack.
If it is an **operator**, then
 - (i) Pop top two values from the stack.
(If stack does not contain two items, error due to a malformed RPN)
 - (ii) Apply the operator to these two values.
 - (iii) Push the resulting value back onto the stack.
3. When the end of expression encountered, its value is on top of the stack
(and, in fact, must be the only value in the stack).

Evaluating RPN Expressions

Example: 2 3 4 + 5 6 - - *

Push 2

Push 3

Push 4

Read +

Pop 4, Pop 3, $3 + 4 = 7$

Push 7

Push 5

Push 6

Read -

Pop 6, Pop 5, $5 - 6 = -1$

Push -1

Read -

Pop -1, Pop 7, $7 - (-1) = 8$

Push 8

Read *

Pop 8, Pop 2, $2 * 8 = 16$

Push 16

Evaluating RPN Expressions

- **Unary minus causes problems**

Example:

5 3 - -

→ 5 3 - -

→ 5 -3 - → 8

5 3 - -

→ 5 3 - -

→ 2 - → -2

Use a different symbol:

5 3 ~ -

5 3 - ~

Converting Infix to RPN

1. Initialize an empty stack of operators.
2. While no error has occurred and end of infix expression not reached
 - a. Get next Token in the infix expression.
 - b. If Token is
 - (i) **a left parenthesis**: Push it onto the stack.
 - (ii) **a right parenthesis**: Pop and display stack elements until a left parenthesis is encountered, but do not display it.
(Error if stack empty with no left parenthesis found.)
 - (iii) **an operator**: If stack empty or Token has higher priority than top stack element, push Token on stack. Otherwise, pop and display the top stack element; then repeat the comparison of Token with new top stack item.
(Left parenthesis in stack has lower priority than operators)
 - (iv) **an operand**: Display it.
3. When end of infix expression reached, pop and display stack items until stack is empty.

Converting Infix to RPN

- **Example: $((A+B)*C)/(D-E)$**

Push (

Push (

Display A

A

Push +

Display B

AB

Read)

Pop +, Display +, Pop (

AB+

Push *

Display C

AB+C

Read)

Pop *, Display *, Pop (

AB+C* // stack now empty

Push /

Push (

Display D

AB+C*D

Push -

Display E

AB+C*DE

Read)

Pop -, Display -, Pop (

AB+C*DE-

Pop /, Display /

AB+C*DE-/

Converting Infix to RPN

```
void convertingInfixToRPN(const string& exp)
{
    Stack<char>* op = new Stack<char>(exp.length());
    char ch;
    for(int i=0; i<exp.length; i++)
    {
        switch(exp[i]){
            case '(': op->push(exp[i]);
                    break;
            case ')': while( (ch=op->pop()) != '(' ) print(ch);
                    break;
            case '*':
            case '/': while(op->length()>0 && (op->topValue()=='*' || op->topValue()=='/'))
                    print(op->pop());
                    op->push(exp[i]);
                    break;
            case '+':
            case '-': while( op->length()>0 && op->topValue()!='(' )
                    print(op->pop());
                    op->push(exp[i]);
                    break;
            default: print(exp[i]);    //operand
        }
    } //end for loop
    while (op->length()>0) print(op->pop());
}
```

Summary

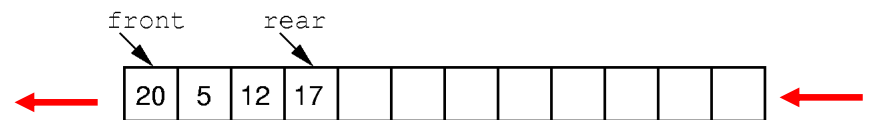
- **ADT stack operations have a last-in, first-out (LIFO) behavior**
- **Stack applications**
- **A strong relationship exists between recursion and stacks**

面试题

- 判断元素出栈、入栈顺序的合法性。如入栈的序列(1,2,3,4,5), 出栈序列为 (3,2,4,1,5)) 或 (3,4,1,5,2)
- 一个数组实现两个栈(共享栈)
- 在Stack<int>中增加成员函数min(), 返回当前栈中的最小值。在该栈中, 调用min、push、及pop的时间复杂度都是 $O(1)$ 。

(Hint: 为了保证当我们弹出当前最小元素后, 下一个最小元素也能够立即得出。因此我们可以把每次的最小元素放到另一个辅助栈中)

6.3 Queue



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

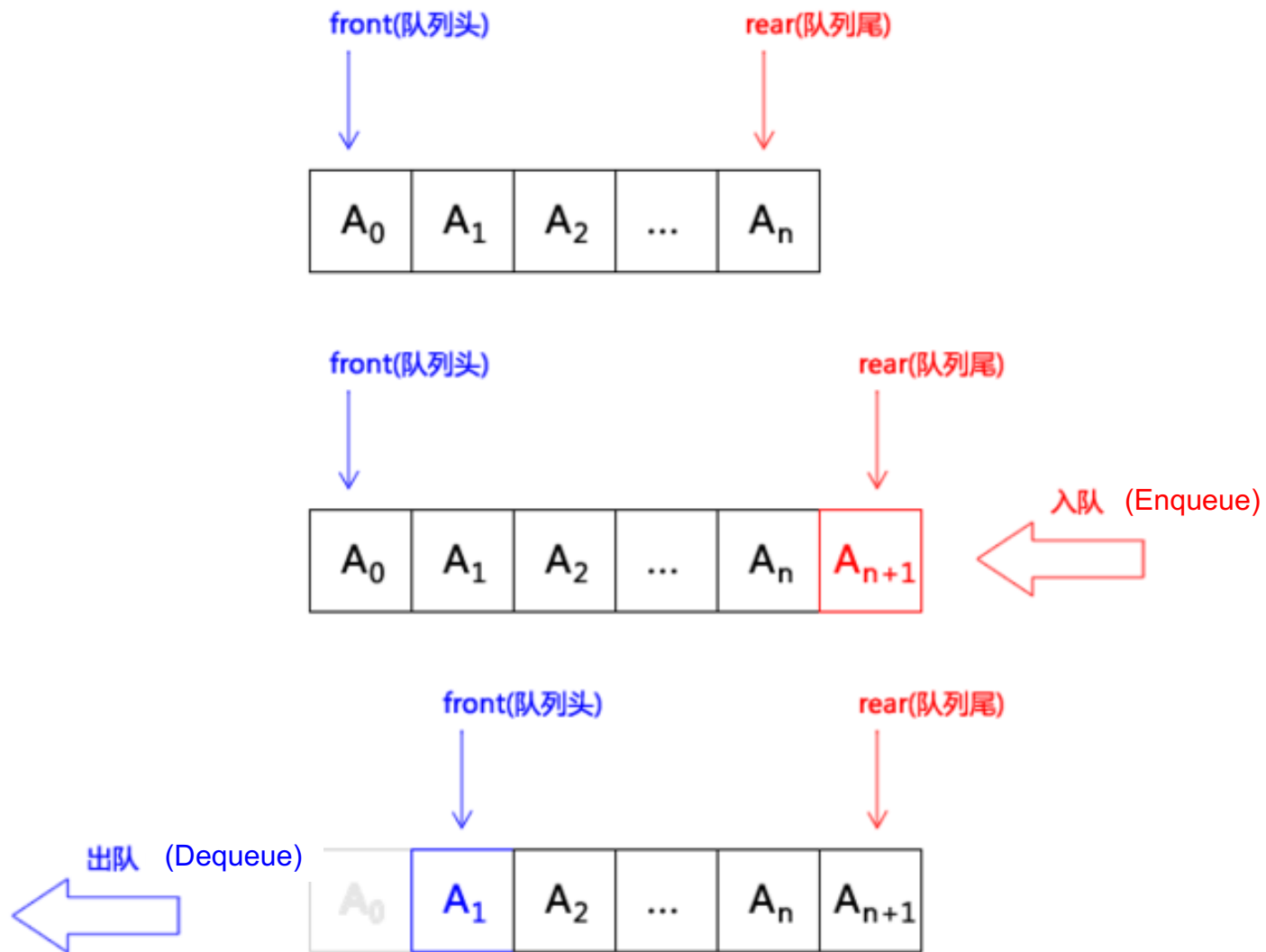
FIFO: First in, First Out

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

Notation:

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

Queues



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

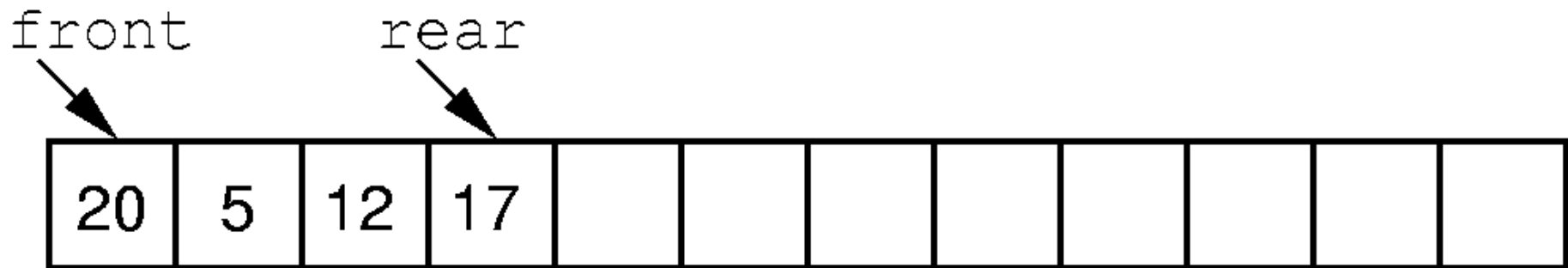
Queue ADT

```
// 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;
};
```

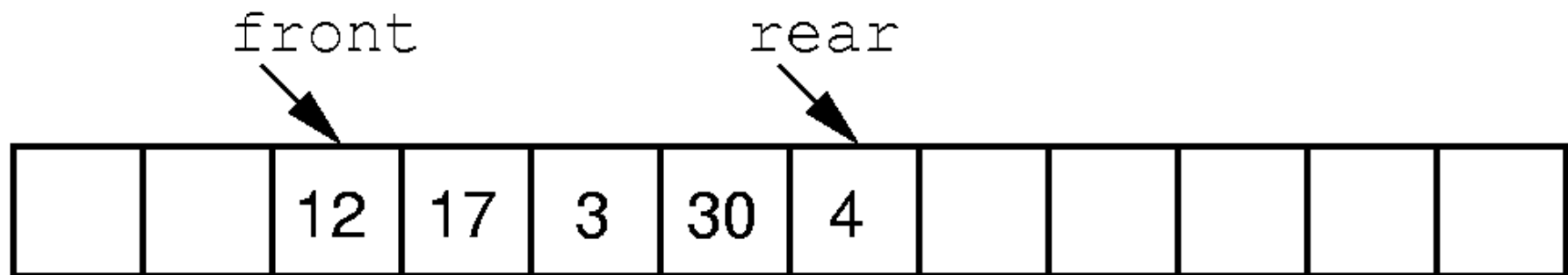

Array-based Queue

- 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)

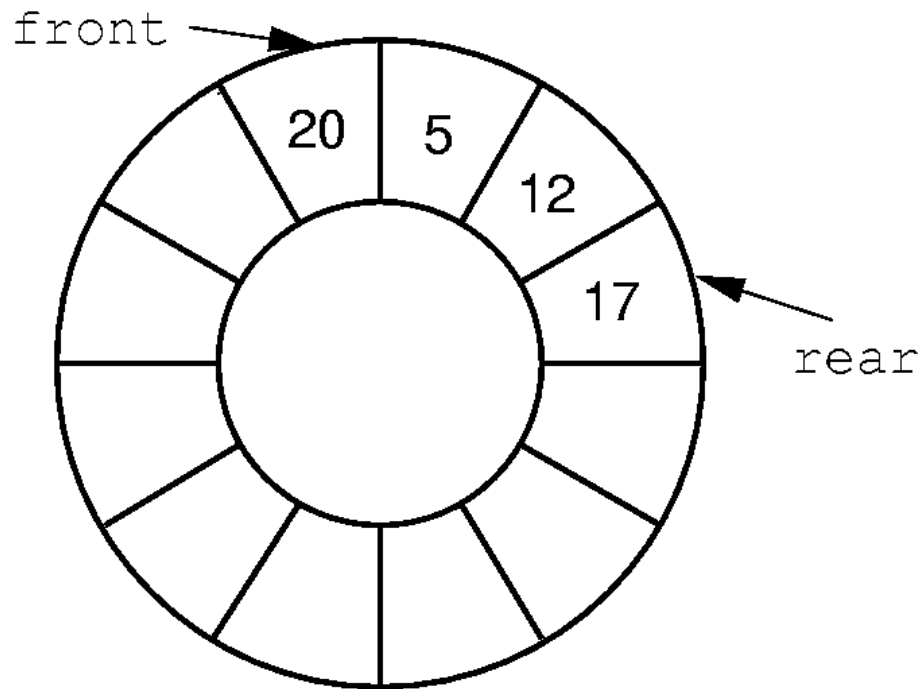
Queue Implementation (1)

- 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.

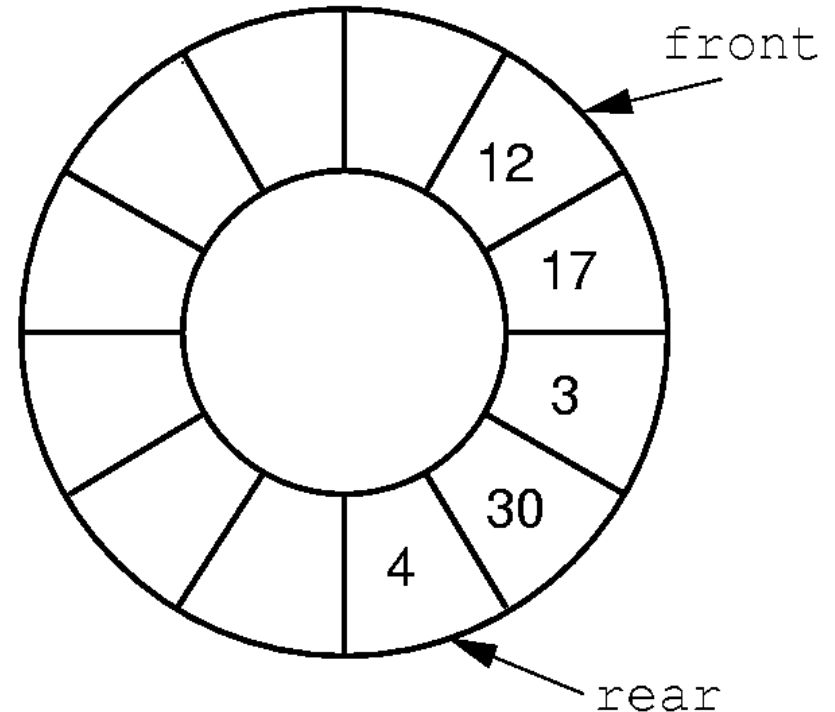
Queue Implementation (2)

- 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)

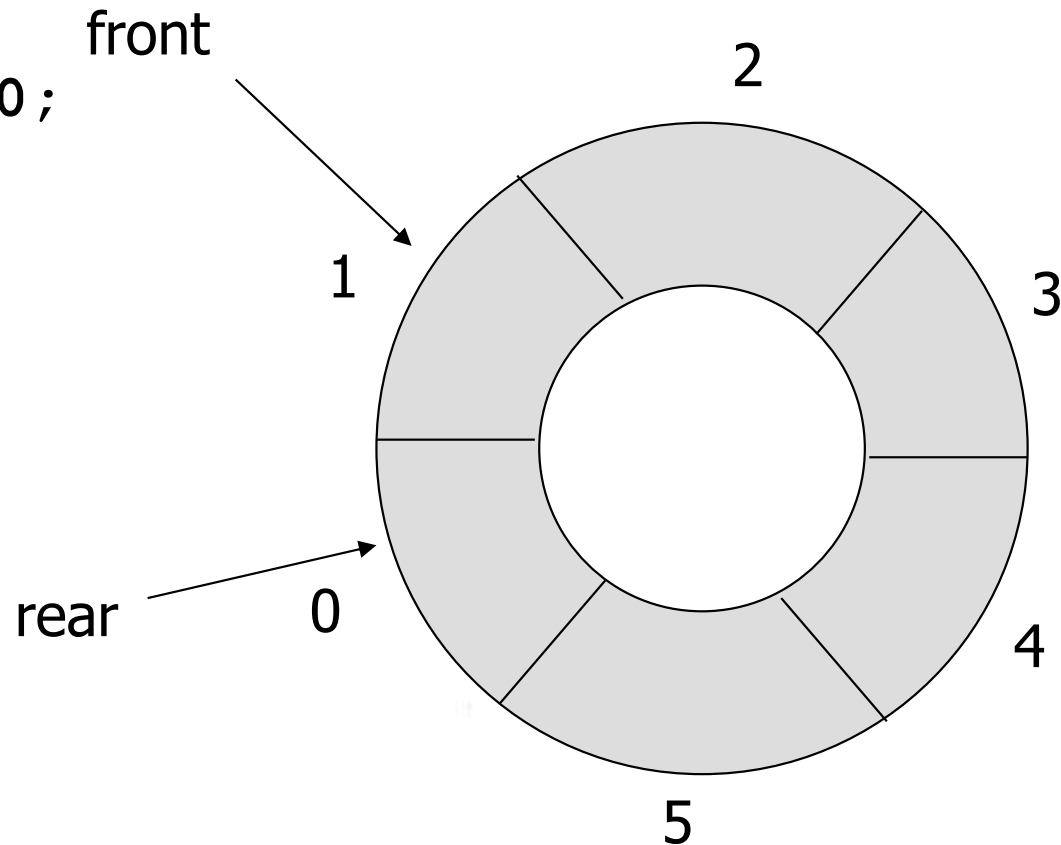
Queue Implementation (2)

- 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**

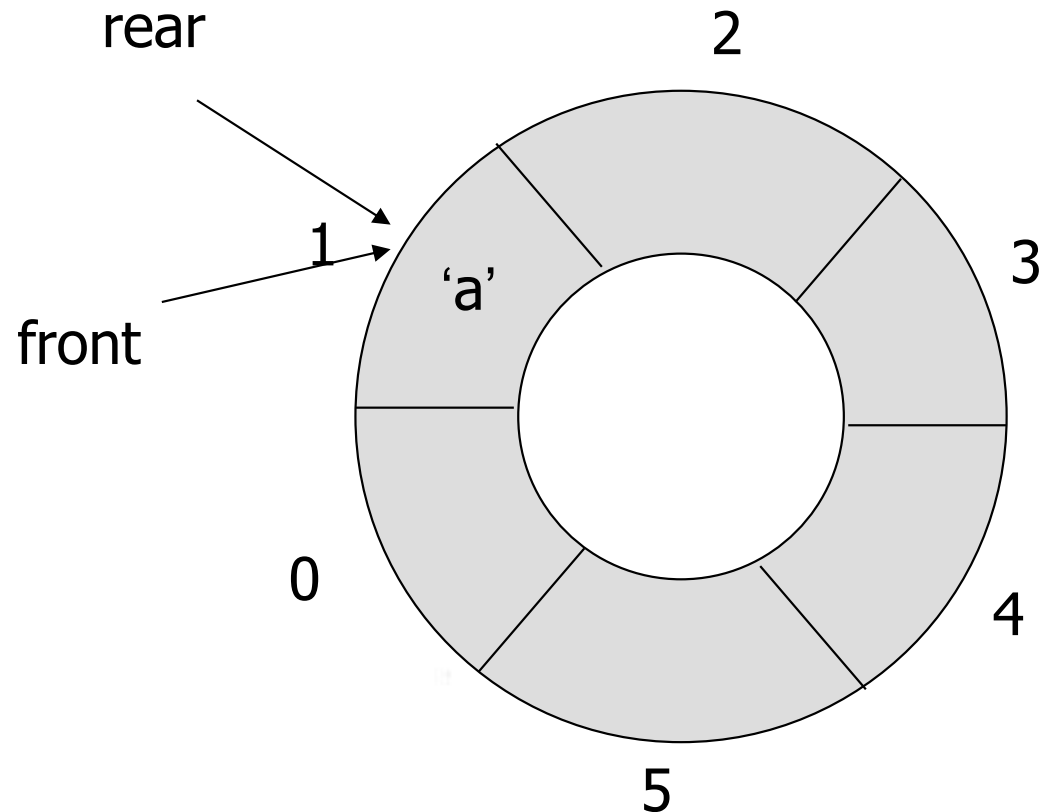
```
front = 1; rear = 0;  
enqueue (q, 'a');
```



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

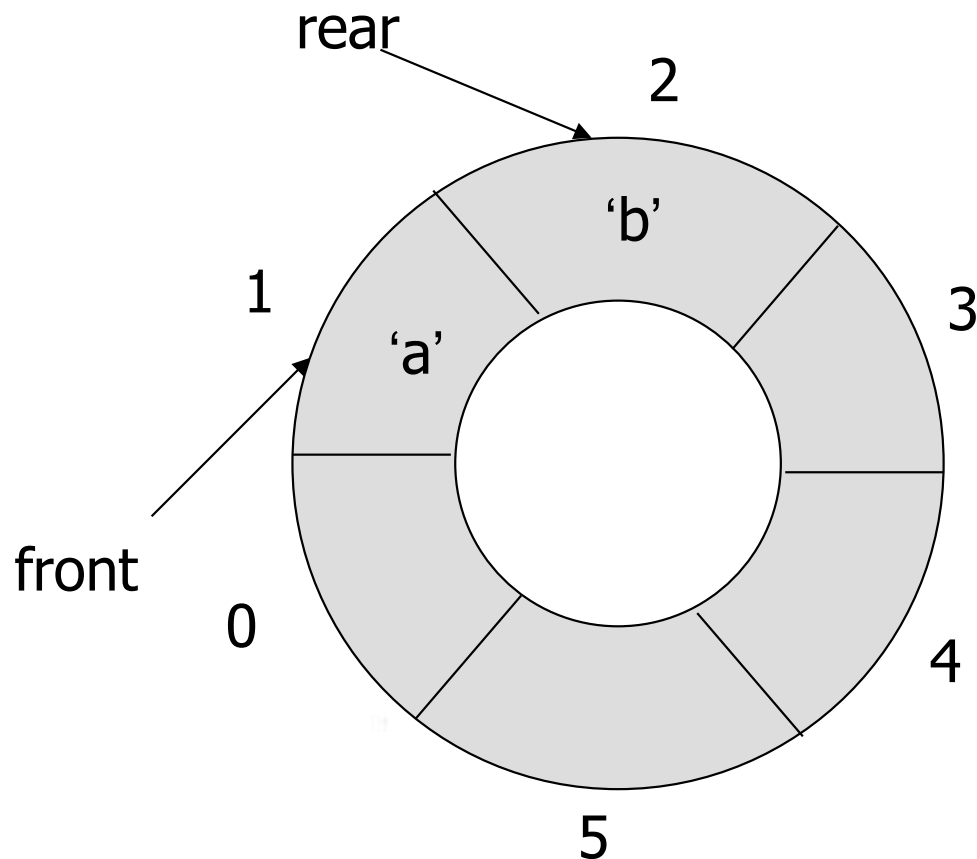
- A refinement of the lazy approach is the **circular queue**

```
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**

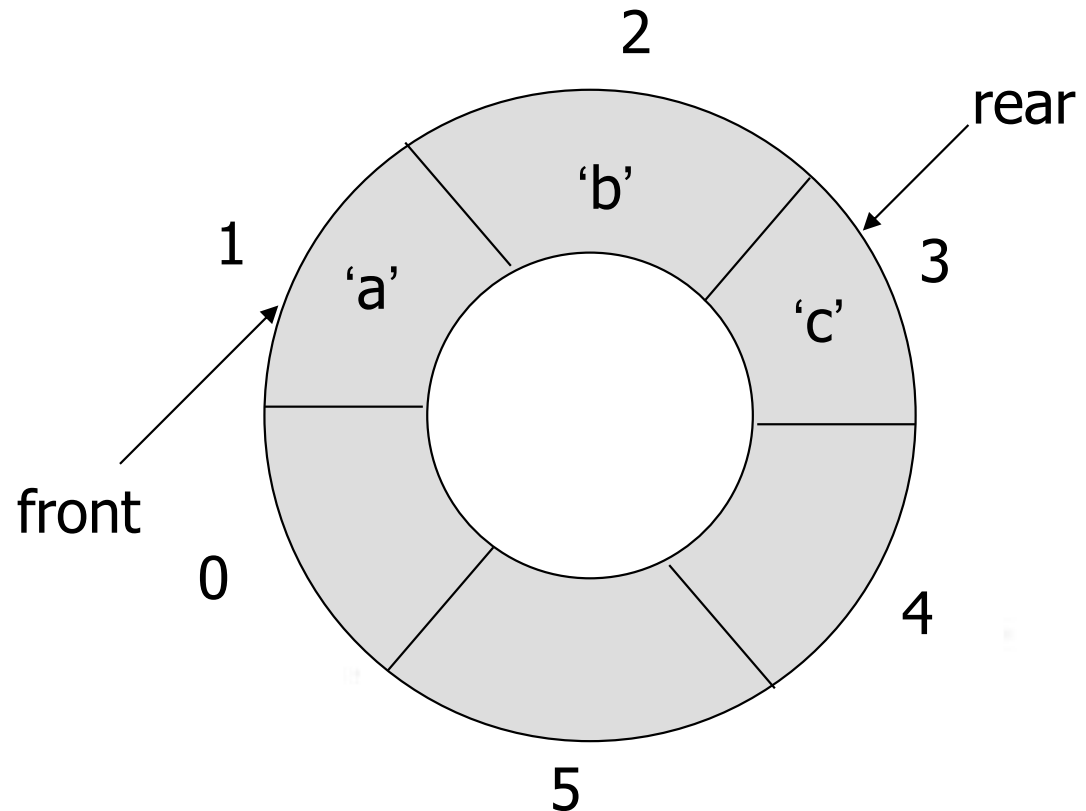
```
front = 1; rear = 3;
```

```
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;
```

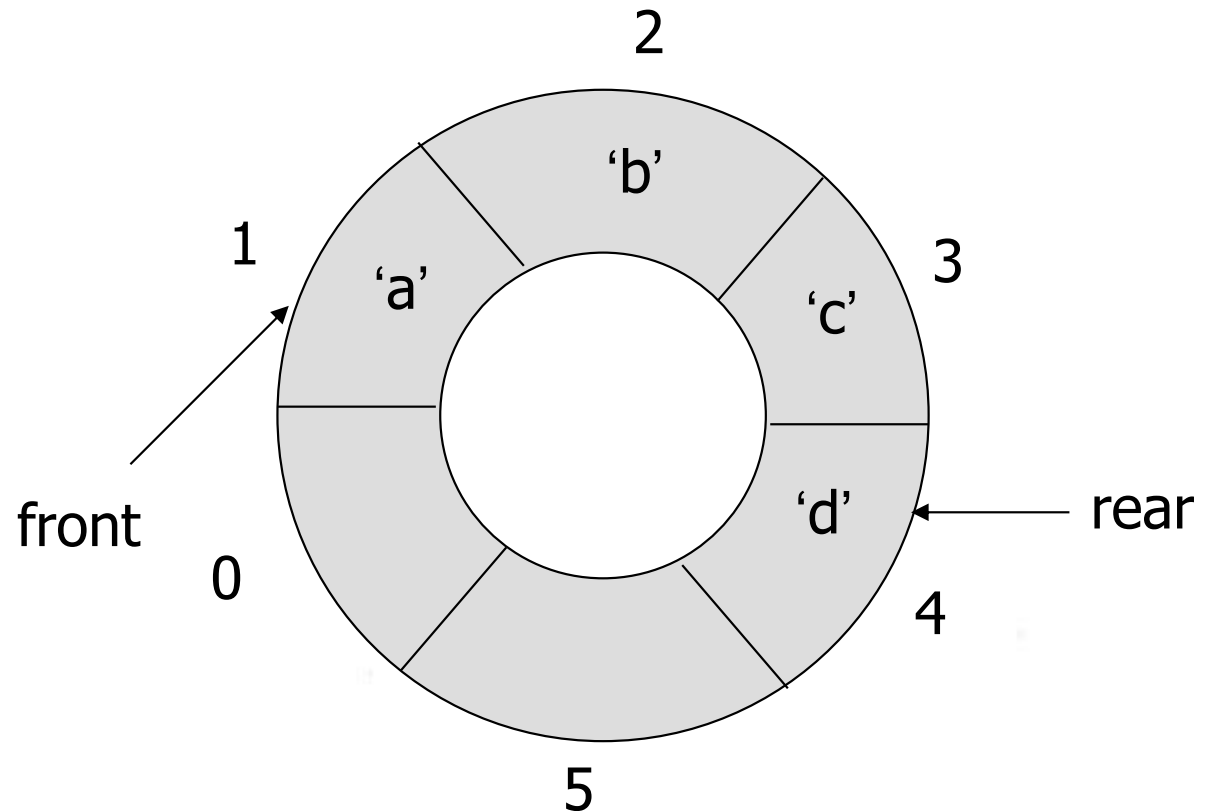
```
enqueue (q, 'a');
```

```
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');
```

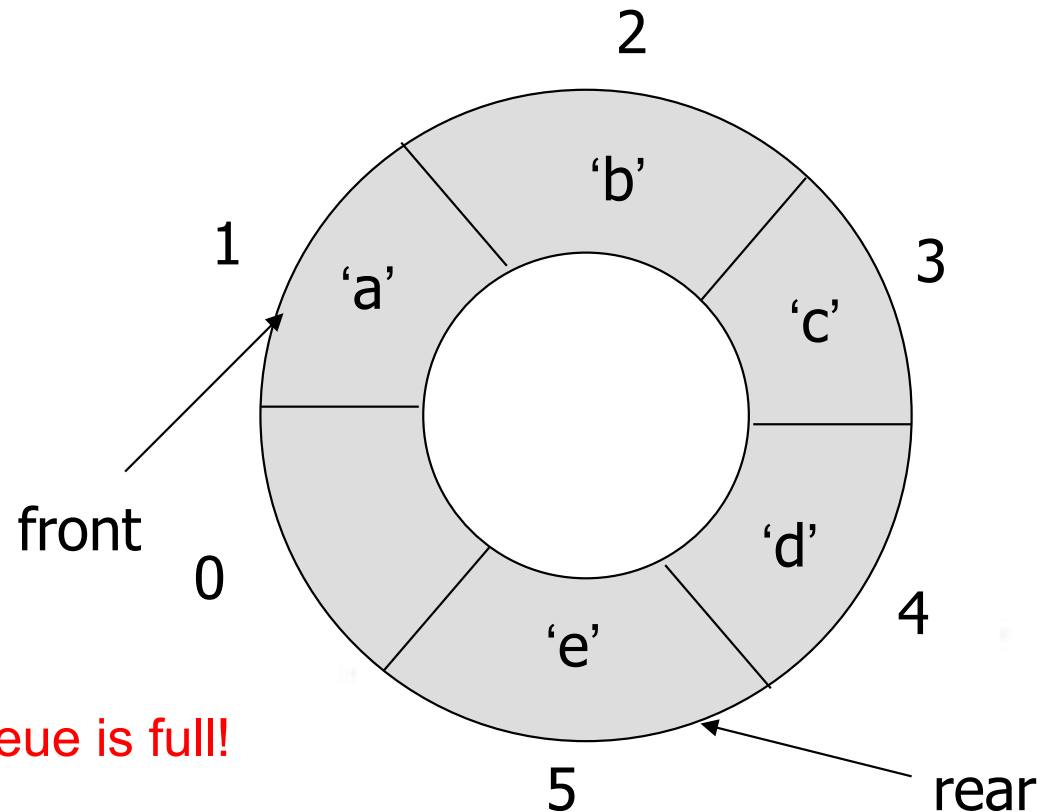
```
enqueue (q, 'c');
```

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

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

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

return error, because the queue is full!

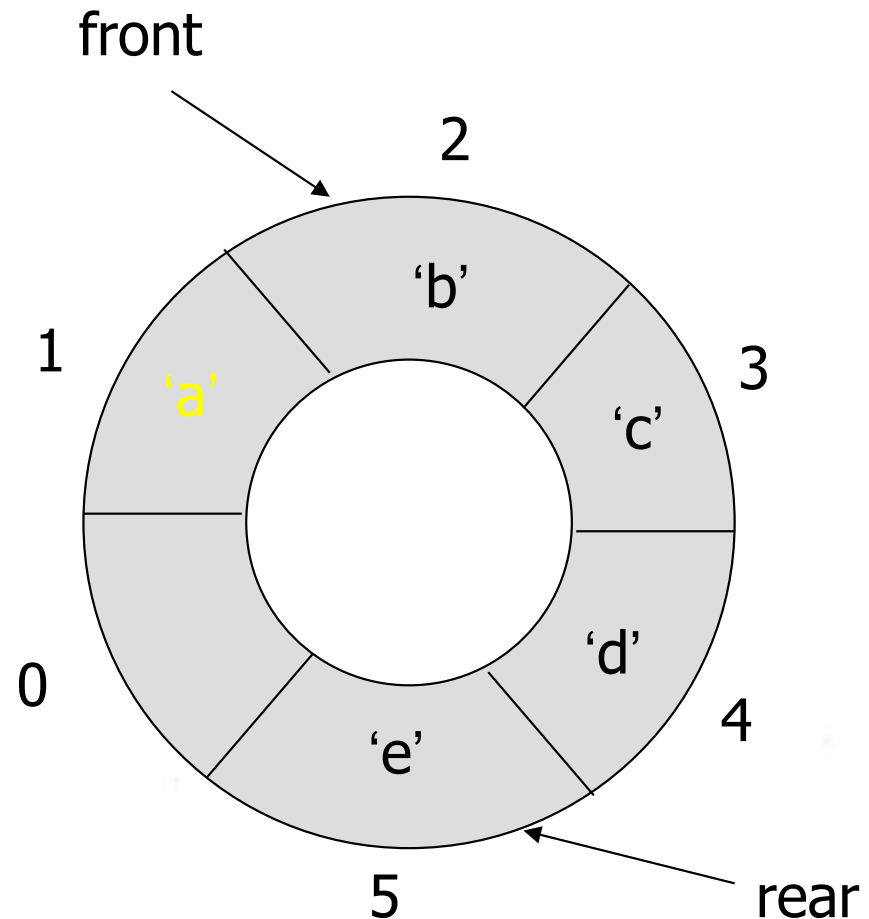


Circular Queue Demo

- A refinement of the lazy approach is the **circular queue**

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

```
deQueue (q) ;
```



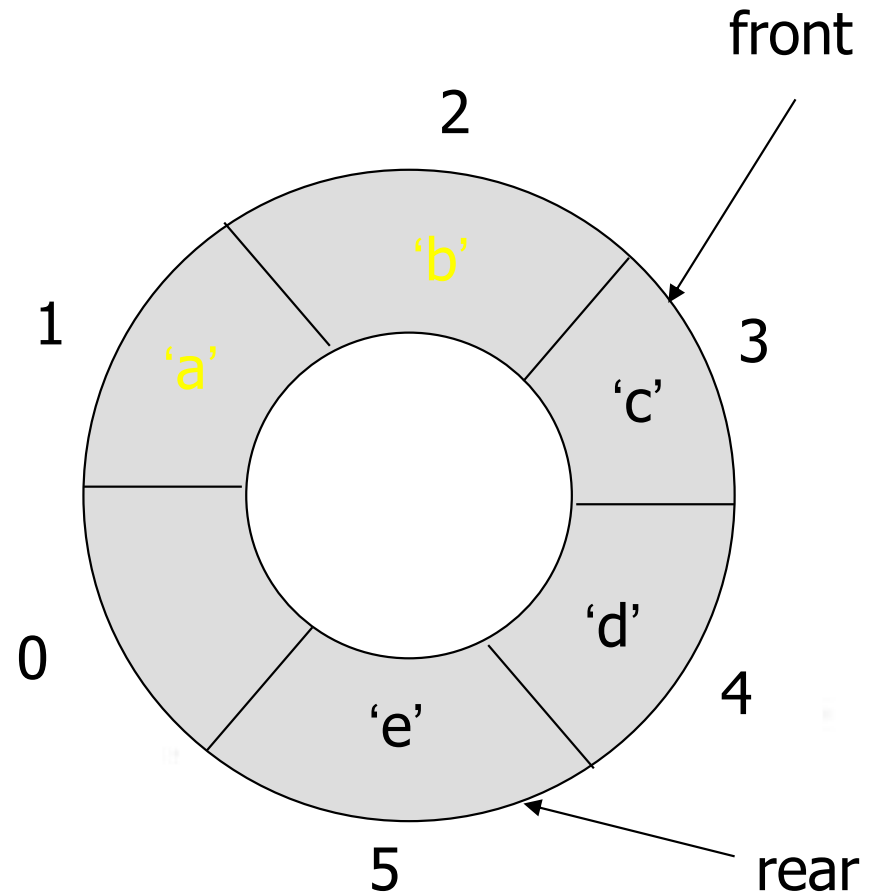
Circular Queue Demo

- A refinement of the lazy approach is the **circular queue**

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

```
deQueue (q);
```

```
deQueue (q);
```



Circular Queue Demo

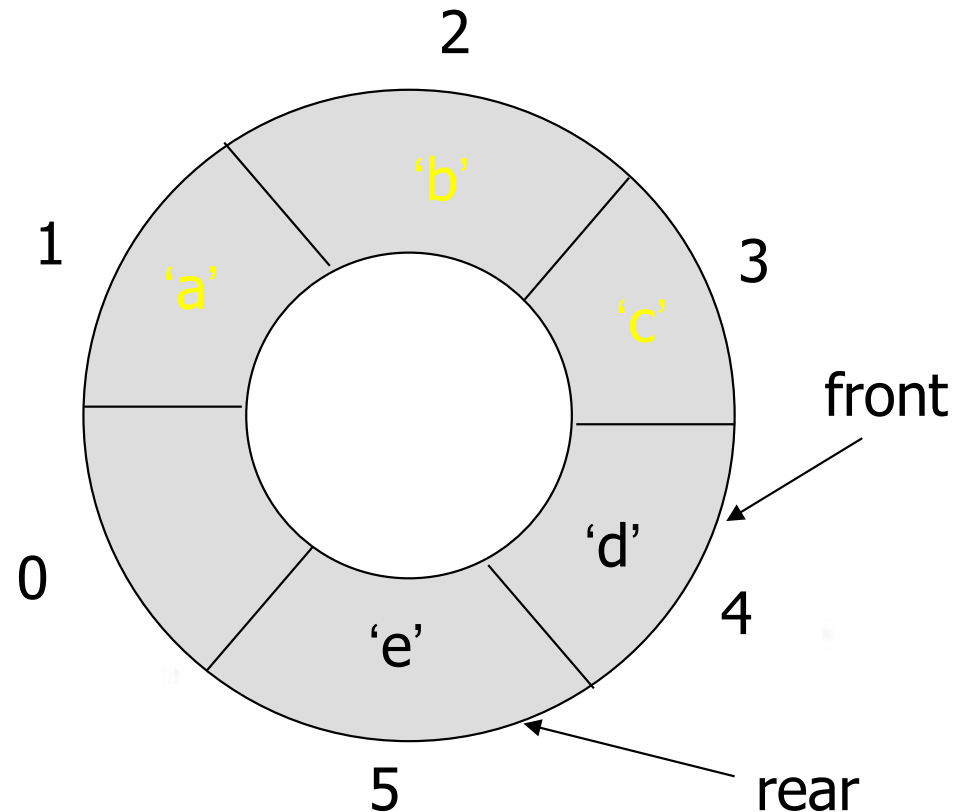
- A refinement of the lazy approach is the **circular queue**

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

```
deQueue (q) ;
```

```
deQueue (q) ;
```

```
deQueue (q) ;
```



Circular Queue Demo

- A refinement of the lazy approach is the **circular queue**

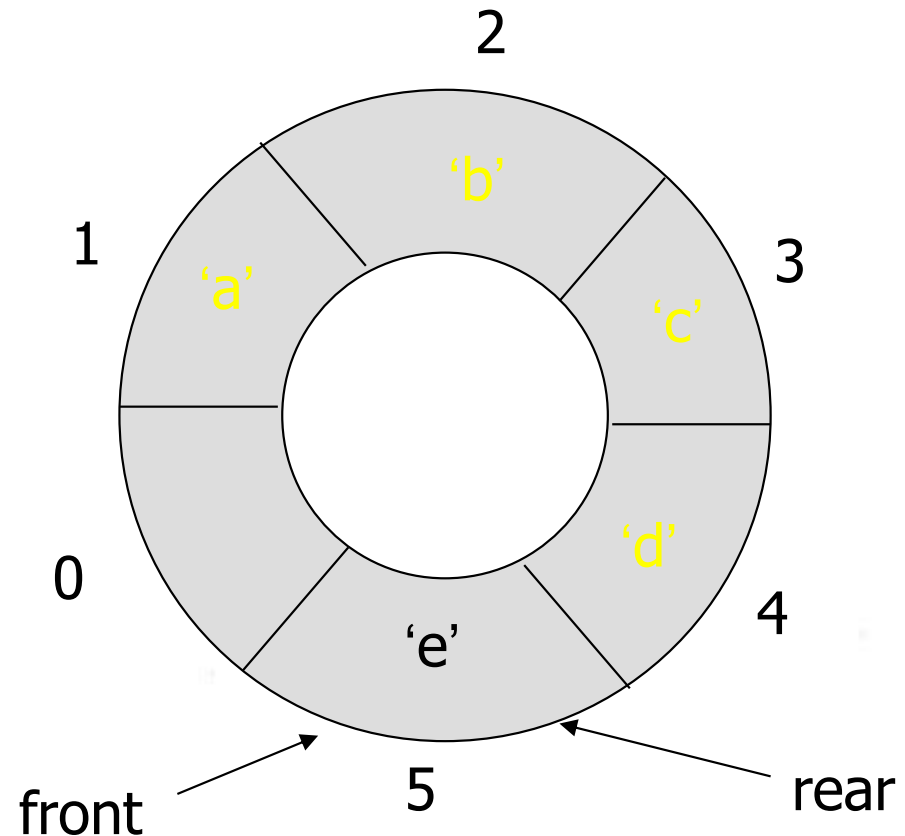
```
front = 5; rear = 5;
```

```
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);
```

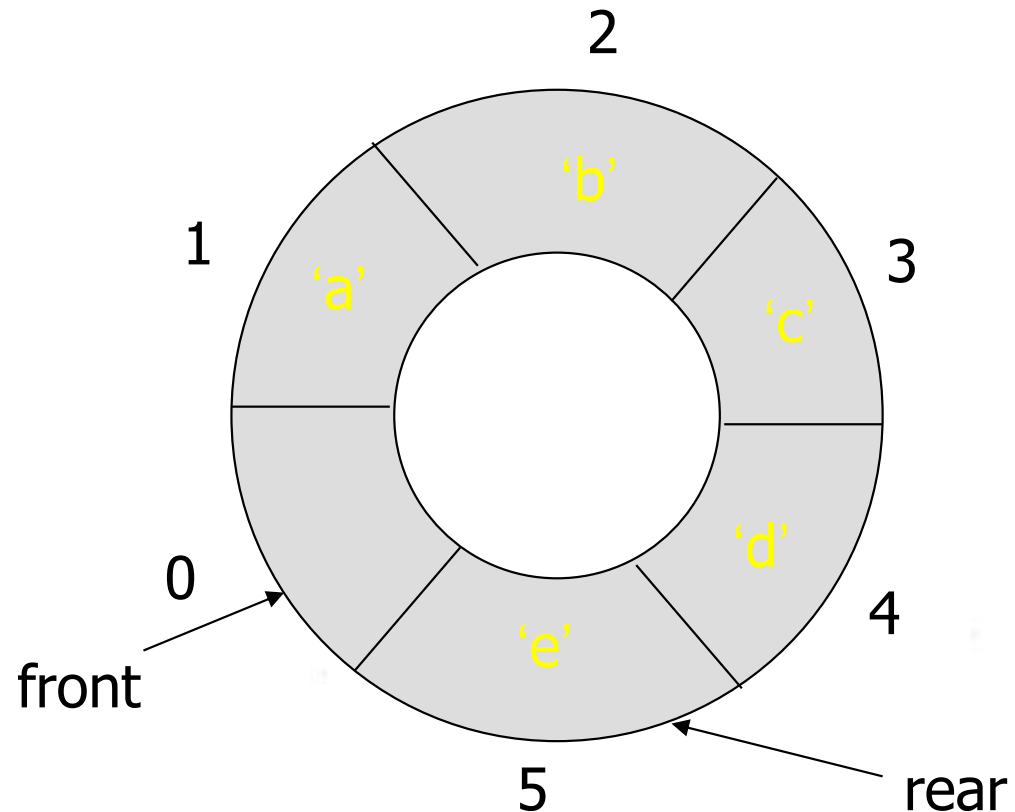
```
deQueue (q);
```

```
deQueue (q);
```

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



return error, because the queue is empty!

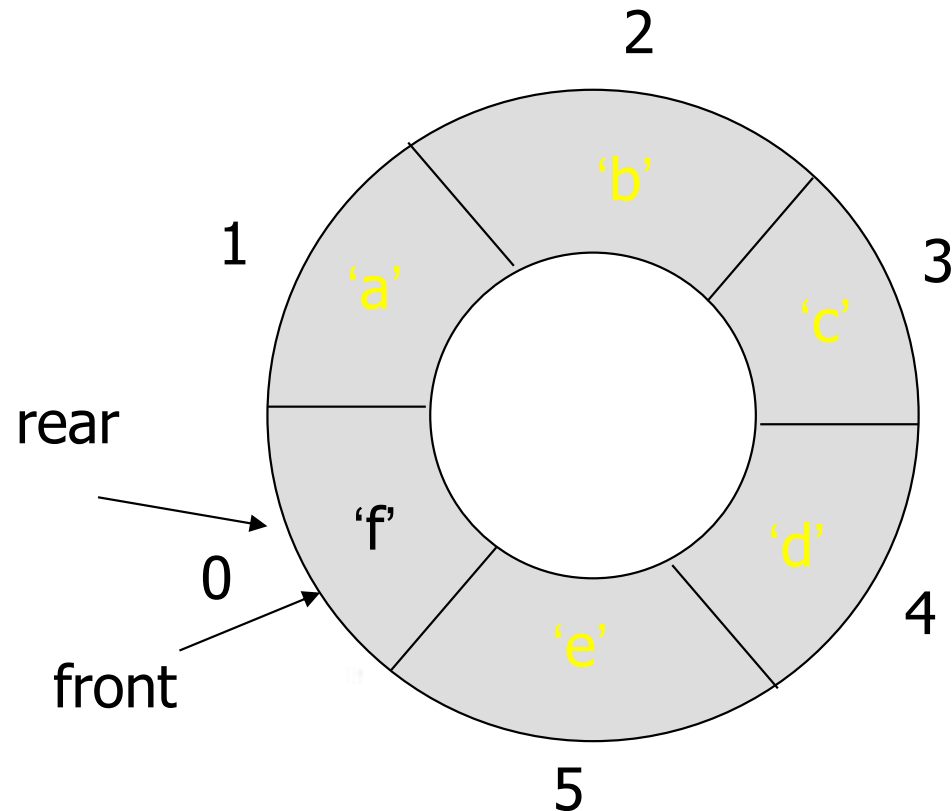


Circular Queue Demo

- 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');
```

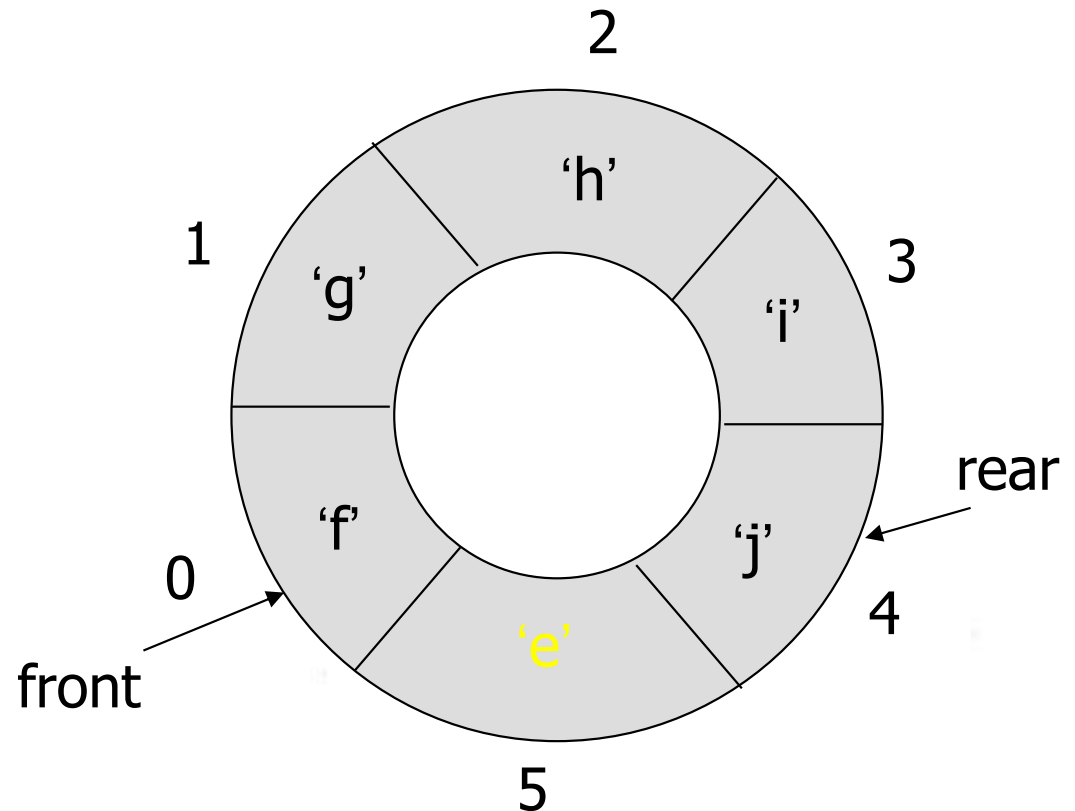
```
enqueue (q, 'g');
```

```
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:

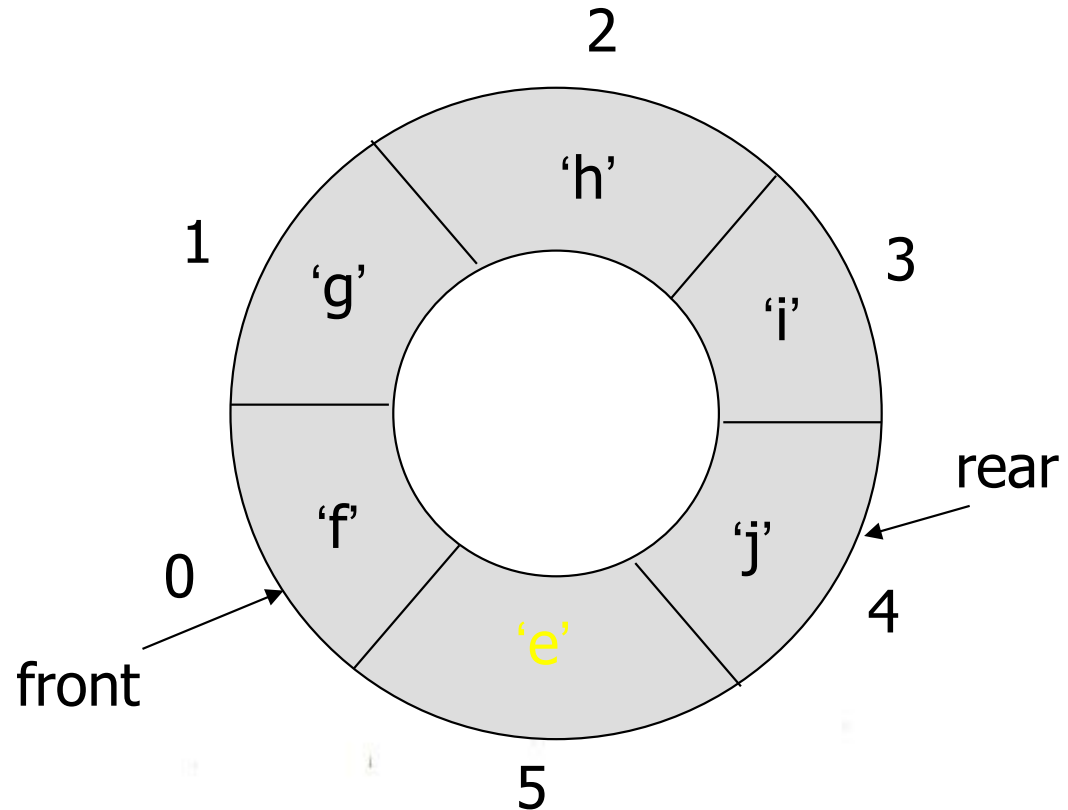
```
front == (rear+2)%N;
```

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  
        maxSize = size+1;  
        // Make list array one position larger for empty slot  
        rear = 0;  
        front = 1; //Empty queue at the initial time  
        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 Queue 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 Queue 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 Queue 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 Queue 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 Queue 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.


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.



《数据结构与算法》课程组
重庆大学计算机学院



End of Chapter

