

1 Abstract Data Type

CHAPTER 3

Lists, Stacks, and Queues

§ 1 Abstract Data Type (ADT)

【Definition】 **Data Type** = { Objects } \cup { Operations }

【Example】 **int** = { 0, ± 1 , ± 2 , \dots , INT_MAX, INT_MIN }
 \cup { +, -, \times , \div , %, \dots }

【Definition】 An **Abstract Data Type (ADT)** is a data type that is organized in such a way that the **specification** on the objects and **specification** of the operations on the objects are **separated from** the **representation** of the objects and the **implementation** on the operations.

1/14

2 The List ADT

2.1 Simple Array implementation of Lists

2.1.1 *Sequential mapping* 连续的

- negative
 - MaxSize 要预先制定，系统要确定开创的Buffer
 - Insertion and Deletion 不可以
- Positive: Find_Kth只需要 $O(1)$ 的时间

2.2 Linked Lists 链表

2. Linked Lists § 2 The List ADT

Address	Data	Pointer
0010	SUN	1011
0011	QIAN	0010
0110	ZHAO	0011
1011	LI	NULL

Head pointer ptr = 0110

Initialization:

```
typedef struct list_node *list_ptr;
typedef struct list_node {
    char data[4];
    list_ptr next;
};
list_ptr ptr;
```

To link 'ZHAO' and 'QIAN':

```
list_ptr N1, N2;
N1 = (list_ptr)malloc(sizeof(struct list_node));
N2 = (list_ptr)malloc(sizeof(struct list_node));
N1->data = 'ZHAO';
N2->data = 'QIAN';
N1->next = N2;
N2->next = NULL;
ptr = N1;
```

4/14

- 每个人只知道自己下一家是谁，不知道上一家是谁
- 每次malloc 比较花时间

2.2.1 Insertion

Insertion § 2 The List ADT

① temp->next = node->next

② node->next = temp

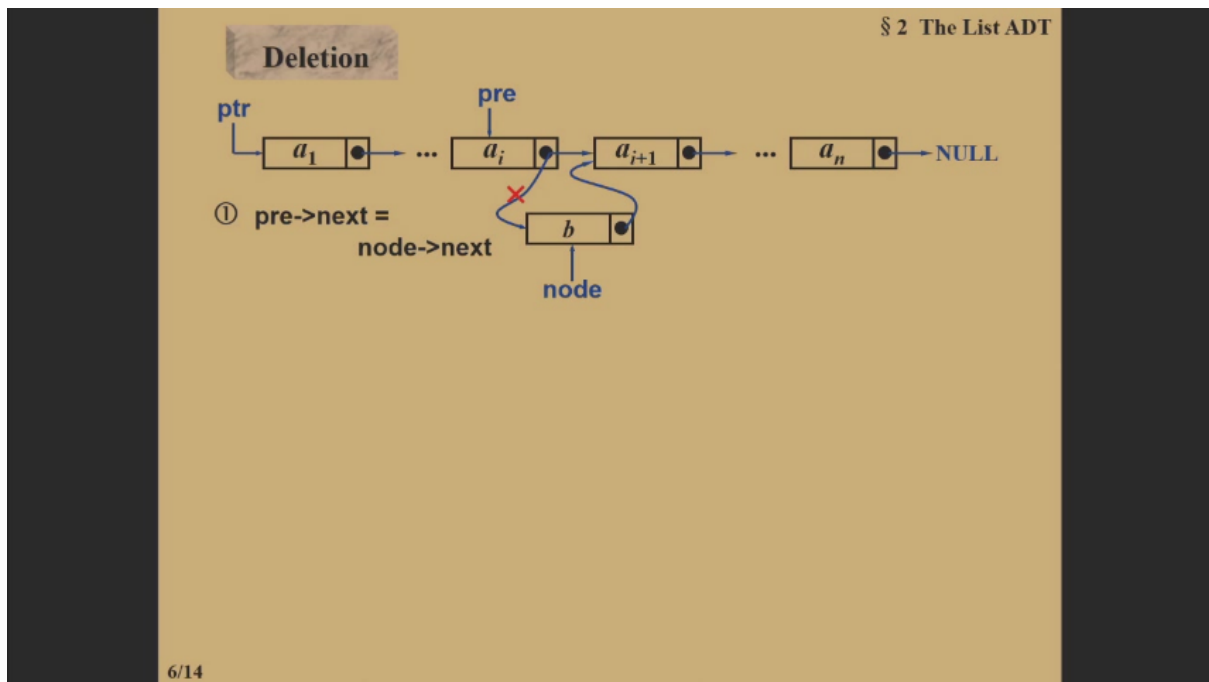
Question: What will happen if the order of the two steps is reversed?

Question: How can we insert a new first item?

5/14

- 注意先后不能反
- 否则会造成断掉，内存泄漏

2.2.2 Deletion



§ 2 The List ADT

Deletion

👍 takes $O(1)$ time.

① $\text{pre} \rightarrow \text{next} = \text{node} \rightarrow \text{next}$

② $\text{free}(\text{node})$

Question: How can we delete the first node from a list?

Answer: We can add a dummy head node to a list.

Read programs in Figures 3.6-3.15 for detailed implementations of operations.

6/14


2.3 Doubly Linked Circular Lists 双向链表

2.3.1 Defination

Doubly Linked Circular Lists

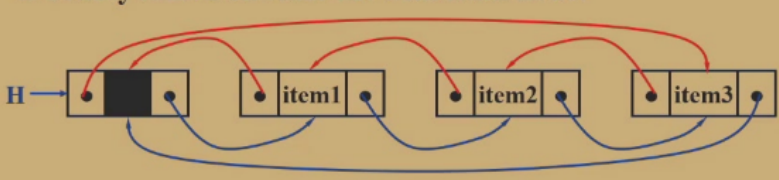
```
typedef struct node *node_ptr;
typedef struct node {
    node_ptr llink;
    element item;
    node_ptr rlink;
};
```

§ 2 The List ADT

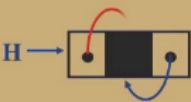


ptr = ptr->llink->rlink
= ptr->rlink->llink

A doubly linked circular list with head node:



An empty list :



7/14

```
1 ptr = ptr -> llink -> rlink
2 ptr = ptr -> rlink -> llink
```

2.3.2 Two Applications

Two Applications

§ 2 The List ADT

* The Polynomial ADT

Objects : $P(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$; a set of ordered pairs of $\langle e_i, a_i \rangle$ where a_i is the **coefficient** and e_i is the **exponent**. e_i are nonnegative integers.

Operations:

- ☞ **Finding degree**, $\max \{ e_i \}$, of a polynomial.
- ☞ **Addition** of two polynomials.
- ☞ **Subtraction** between two polynomials.
- ☞ **Multiplication** of two polynomials.
- ☞ **Differentiation** of a polynomial.

8/14

```

1 // Representation 1
2 typedef struct{
3     int CoeffArray[ MaxDegree + 1];
4     int HighPower;
5 }

```

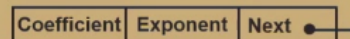
【Representation 2】

§ 2 The List ADT

Given: $A(x) = a_{m-1}x^{e_{m-1}} + \dots + a_0x^{e_0}$

where $e_{m-1} > e_{m-2} > \dots > e_0 \geq 0$ and $a_i \neq 0$ for $i = 0, 1, \dots, m-1$.

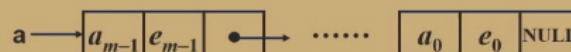
We represent each term as a node

**Declaration:**

```

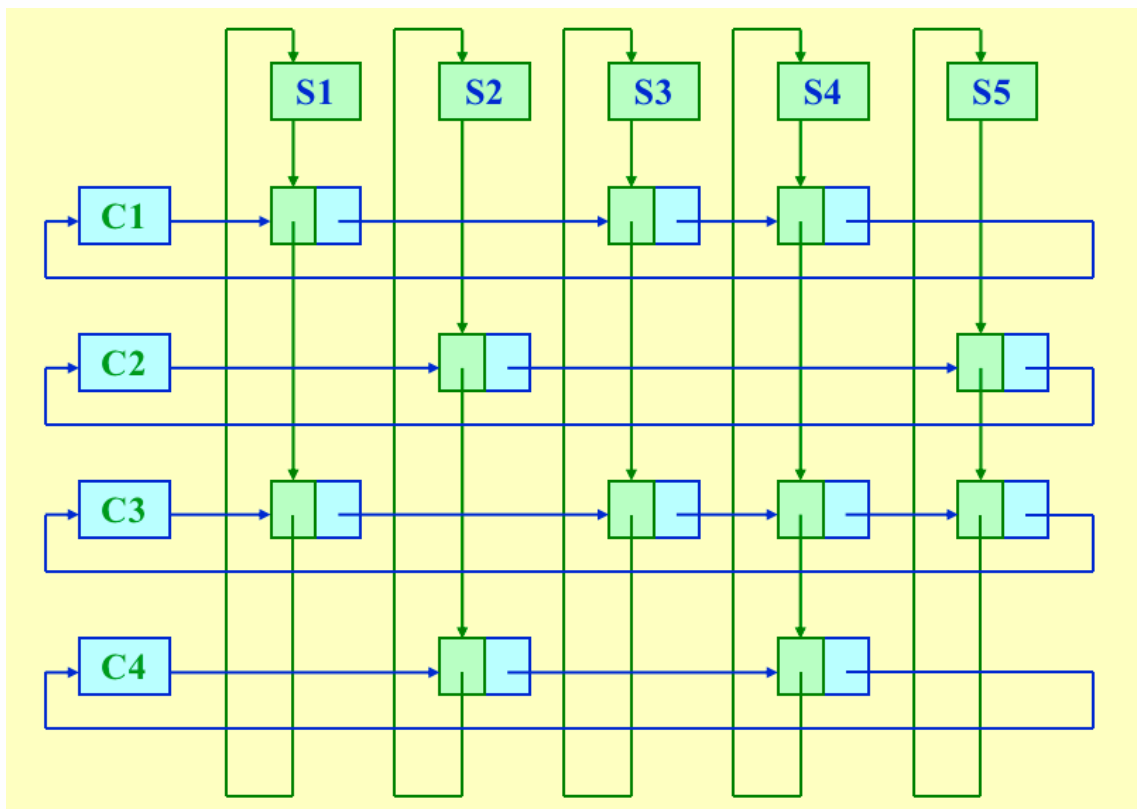
typedef struct poly_node *poly_ptr;
struct poly_node {
    int    Coefficient ; /* assume coefficients are integers */
    int    Exponent;
    poly_ptr Next;
};
typedef poly_ptr a; /* nodes sorted by exponent */

```



10/14

Example: Suppose that we have 40,000 students and 2,500 courses. Print the students' name list for each course, and print the registered classes' list for each student.

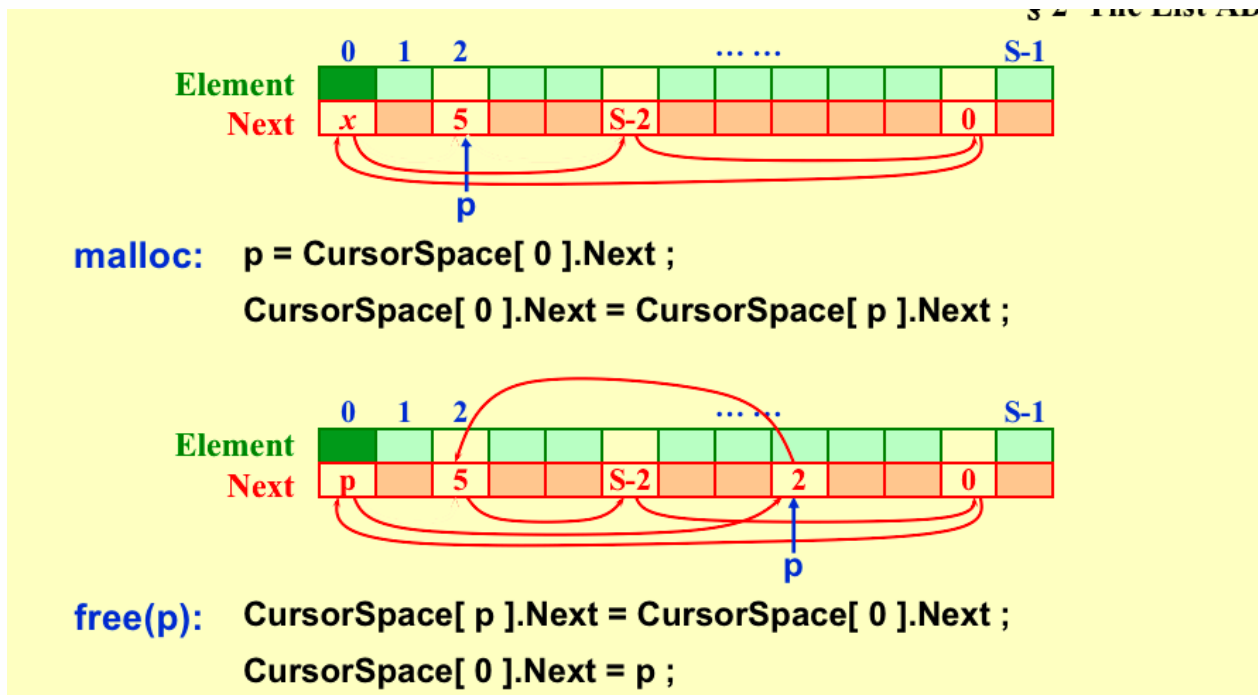


- No. 6 / 15

2.5 Cursor Implementation of Linked Lists (no pointer)

Features that a linked list must have:

- The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.
- New structure can be obtained from the system's global memory by a call to malloc and released by a call to free.



Read operation implementations given in Figures 3.31-3.35

Note: The cursor implementation is usually significantly faster because of the lack of memory management routines.

3 The Stack ADT

3.1 Defination

- A stack is a Last-In-First-Out (LIFO) list, that is, an ordered list in which insertions and deletions are made at the top only.

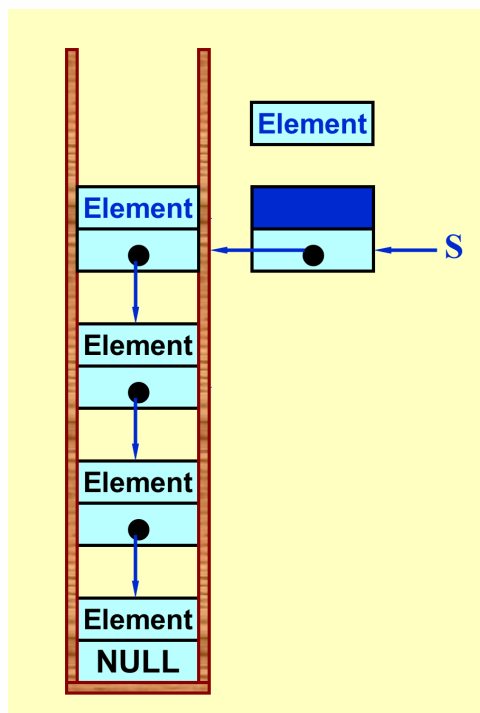
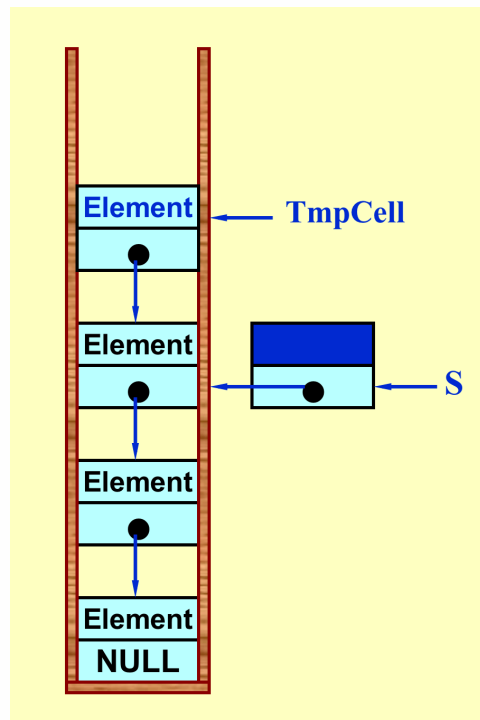
3.2 Operations

```
1  Int  IsEmpty( Stack S );
2
3  Stack CreateStack( );
4
5  DisposeStack( Stack S );
6
7  MakeEmpty( Stack S );
8
9  Push( ElementType X, Stack S );
10
11 ElementType Top( Stack S );
12
13 Pop( Stack S );
```

- Note:
- A Pop (or Top) on an empty stack is an error in the stack ADT.
- Push on a full stack is an implementation error but not an ADT error.

3.3 Implementation

3.3.1 *Push*



- push类似插入，先把想要push的元素指向栈顶的下一个元素
- 再把栈顶指向插入的元素

3.3.2 Pop

2. Implementations

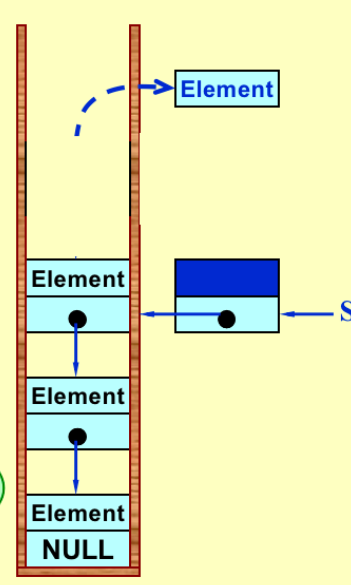
➤ **Linked List Implementation (with a header node)**

Push: ① `TmpCell->Next = S->Next`
 ② `S->Next = TmpCell`

Top: `return S->Next->Element`

Pop: ① `FirstCell = S->Next`
 ② `S->Next = S->Next->Next`
 ③ `free (FirstCell)`

§ 3 The Stack ADT



- 实际上一对malloc和free很花时间，可以用另外一个堆栈来做为暂时存放的栈，这样子下次malloc、free就少做一次

```

1 struct StackRecord {
2     int Capacity ;           /* size of stack */
3     int TopOfStack;         /* the top pointer */
4     /* ++ for push, -- for pop, -1 for empty stack */
5     ElementType *Array;     /* array for stack elements */
6 } ;

```

Note:

The stack model must be well encapsulated. That is, no part of your code, except for the stack routines, can attempt to access the Array or TopOfStack variable.

Error check must be done before Push or Pop (Top).

Read Figures 3.38-3.52 for detailed implementations of stack operations.

如何理解相同进栈顺序有不同的出栈结果

n个元素按顺序进栈，有多少种出栈顺序？

- 按顺序进来，不一定是等所有的数都进来之后再pop，可以pop和push交叉

- 3个元素，有五种出栈顺序

3.4 Applications

3.4.1 *Balancing Symbols check*: 检查括号是不是成对

```

1 Algorithm {
2   Make an empty stack S;
3   while (read in a character c) {
4     if (c is an opening symbol)
5       Push(c, S);
6     else if (c is a closing symbol) {
7       if (S is empty) { ERROR; exit; }
8       else { /* stack is okay */
9         if (Top(S) doesn't match c) { ERROR, exit; }
10        else Pop(S);
11      } /* end else-stack is okay */
12    } /* end else-if-closing symbol */
13  } /* end while-loop */
14  if (S is not empty) ERROR;
15 }
```

- $T(N) = O(N)$

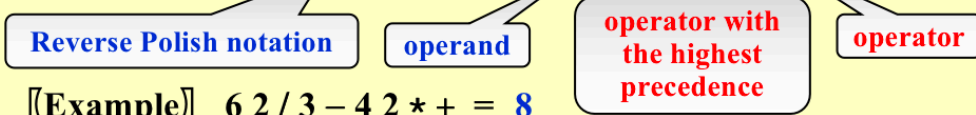
3.4.2 *Postfix Evaluation*

✧ Postfix Evaluation

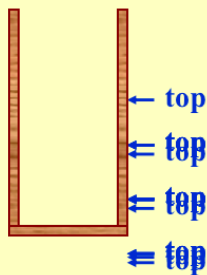
[[Example]] An **infix** expression: $a + b * c - d / e$

A **prefix** expression: $- + a * b c / d e$

A **postfix** expression: $a b c * + d e / -$



[[Example]] $6\ 2\ /\ 3\ -\ 4\ 2\ *\ +\ =\ 8$



Get token: 6 (operand)	Get token: 2 (operand)
Get token: / (operator)	Get token: 3 (operand)
Get token: - (operator)	Get token: 4 (operand)
Get token: 2 (operand)	Get token: * (operator)
Get token: + (operator)	Pop: 8

$T(N) = O(N)$. No need to know precedence rules.

3.4.3 Infix to postfix Conversion

✧ Infix to Postfix Conversion

[[Example]] $a + b * c - d = a b c * + d -$

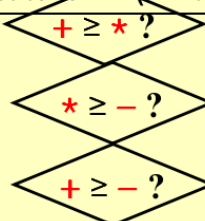
Note:

- The order of operands is the **same** in infix and postfix.
- Operators with **higher** precedence appear **before** those with **lower** precedence.

Output: $a\ b\ c\ *\ +\ d\ -$



Get token: a (operand)	Get token: + (plus)
Get token: b (operand)	Get token: * (times)
Get token: c (operand)	Get token: - (minus)
Get token: d (operand)	



12

- 优先级高执行push，优先级低执行pop。只考虑operator，operand直接打印

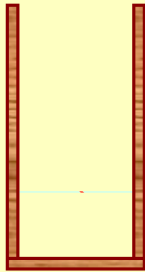
- 第一次进来, $*$ $>$ $+$, push
- 第二次进来, $- < *$, pop $*$
- 再比较, pop $+$

3.4.3.1 带括号的

§ 3 The Stack ADT

[[Example]] $a * (b + c) / d = a b c + * d /$

Output: $a b c + * d /$



Get token: a (operand)	Get token: $*$ (times)
Get token: $($ (lparen)	Get token: b (operand)
Get token: $+$ (plus)	Get token: c (operand)
Get token: $)$ (rpren)	Get token: $/$ (divide)
Get token: d (operand)	

$$T(N) = O(N)$$

- 在栈外面, 括号的优先级是最高的, 就是栈里面的东西和外面的比
- 但是进入栈后, 如果他的优先级还是最高那就会在半途, 括号内未算完就被pop出去
- 因此, 我们设计, 在栈中括号的优先级最低, 目的是把括号内的要先算完

3.4.4 Function Call - System Stack

4 The Queue ADT

4.1 Definition

A queue is a First-In-First-Out (FIFO) list, that is, an ordered list in which insertions take place at one end and deletions take place at the opposite end.

4.2 Operations:

```

1  int IsEmpty( Queue Q );
2  Queue CreateQueue( );
3  DisposeQueue( Queue Q );
4  MakeEmpty( Queue Q );
5  Enqueue( ElementType X, Queue Q );
6  ElementType Front( Queue Q );
7  Dequeue( Queue Q );

```

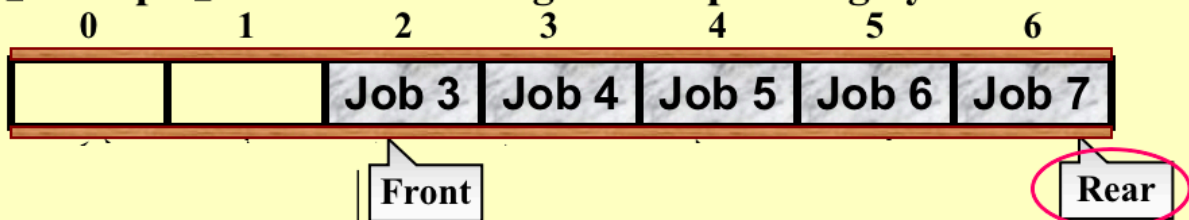
4.3 Array Implementation of Queues (Linked list implementation is trivial)

```

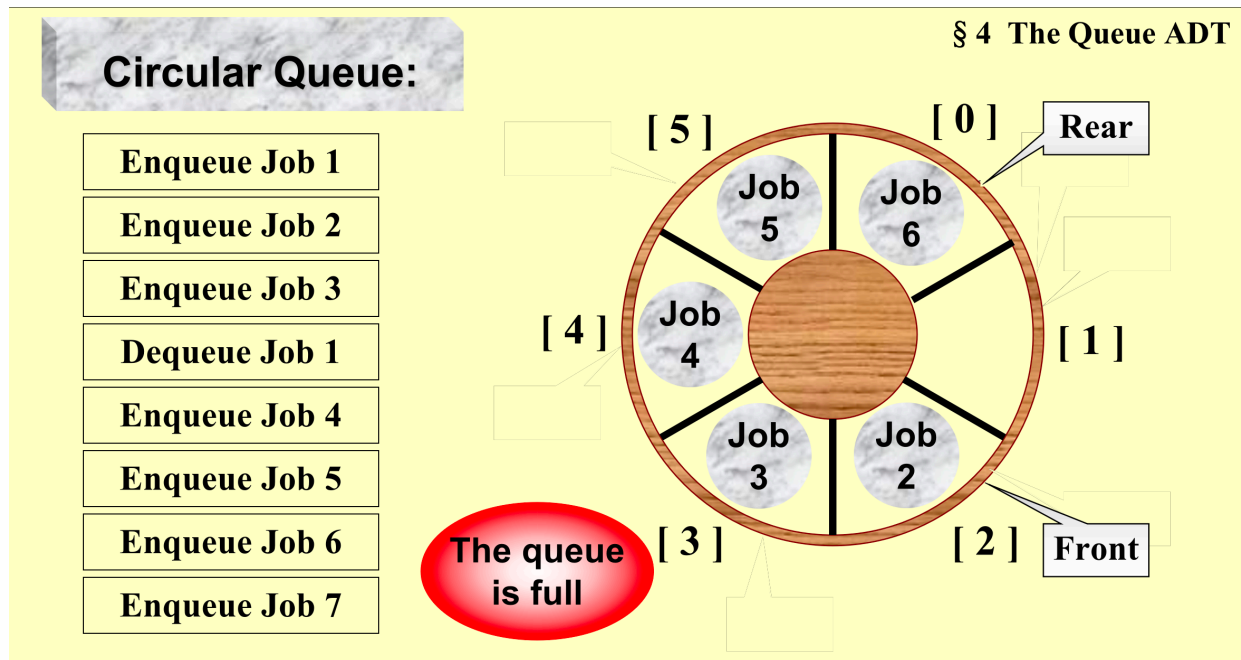
1  struct QueueRecord {
2      int Capacity ;    /* max size of queue */
3      int Front;        /* the front pointer */
4      int Rear;         /* the rear pointer */
5      int Size; /* Optional – the current size of queue */
6      ElementType *Array; /* array for queue elements */
7  } ;

```

[[Example]] Job Scheduling in an Operating System



Enqueue Job 1	Enqueue Job 2	Enqueue Job 3	Dequeue Job 1
Enqueue Job 4	Enqueue Job 5	Enqueue Job 6	Dequeue Job 2
Enqueue Job 7	Enqueue Job 8		



Question: Why is the queue announced full while there is still a free space left?

Answer: 空的时候，Front在 2 位置，而 Rear 在 1 位置，如果我把7放进来，我就无法区分到底这个时候是满的还是空的，因此留一个状态

- Note: Adding a **Size** field can avoid wasting one empty space to distinguish “full” from “empty”. Do you have any other ideas?

■

$$rear = (front + size - 1) \% m \quad (1)$$