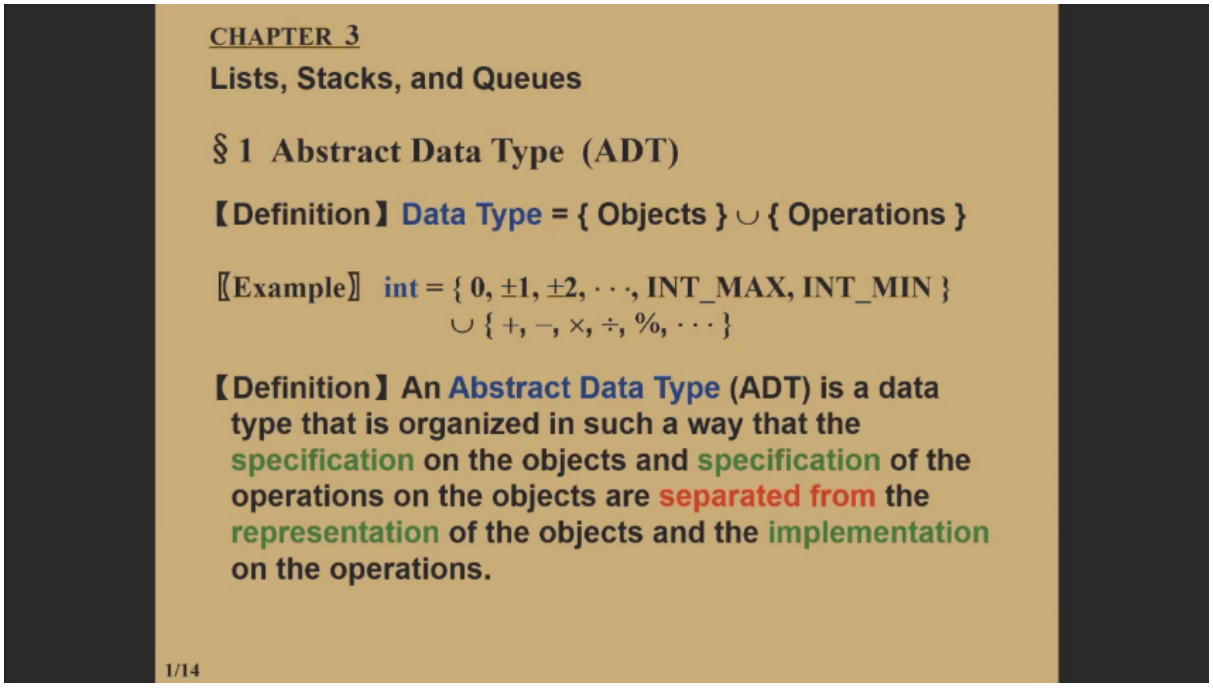
1 Abstract Data Type



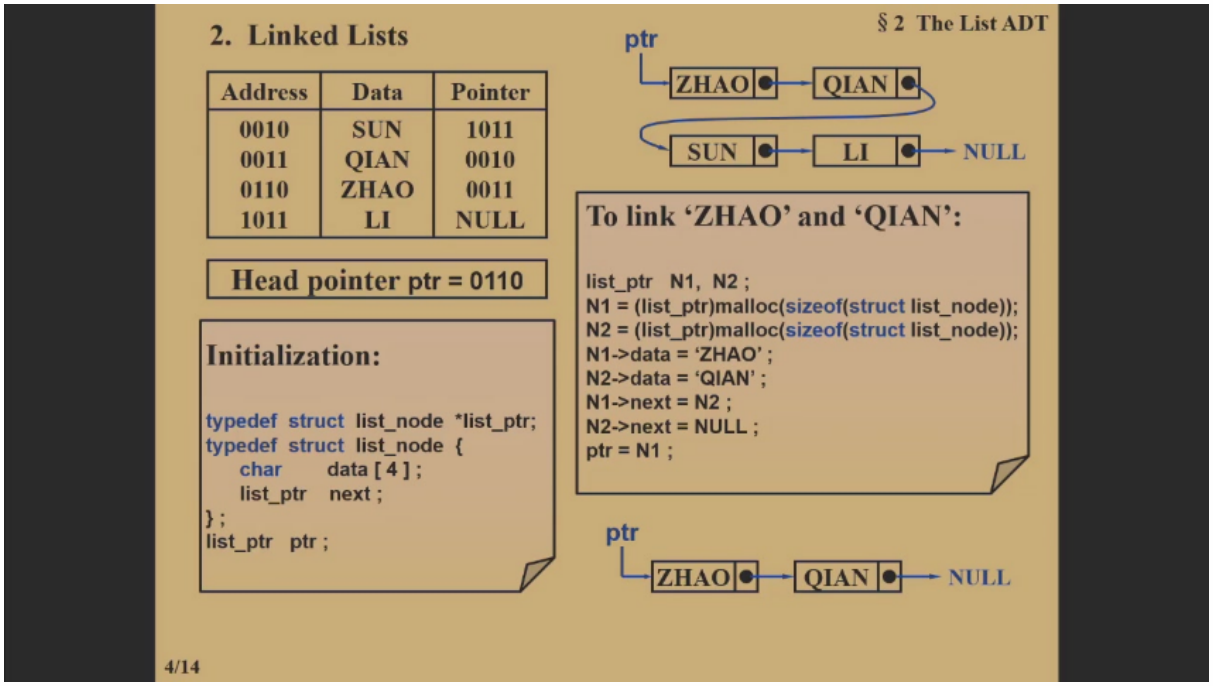
2 The List ADT

2.1 Simple Array implementation of Lists

2.1.1 Sequential mapping 连续的

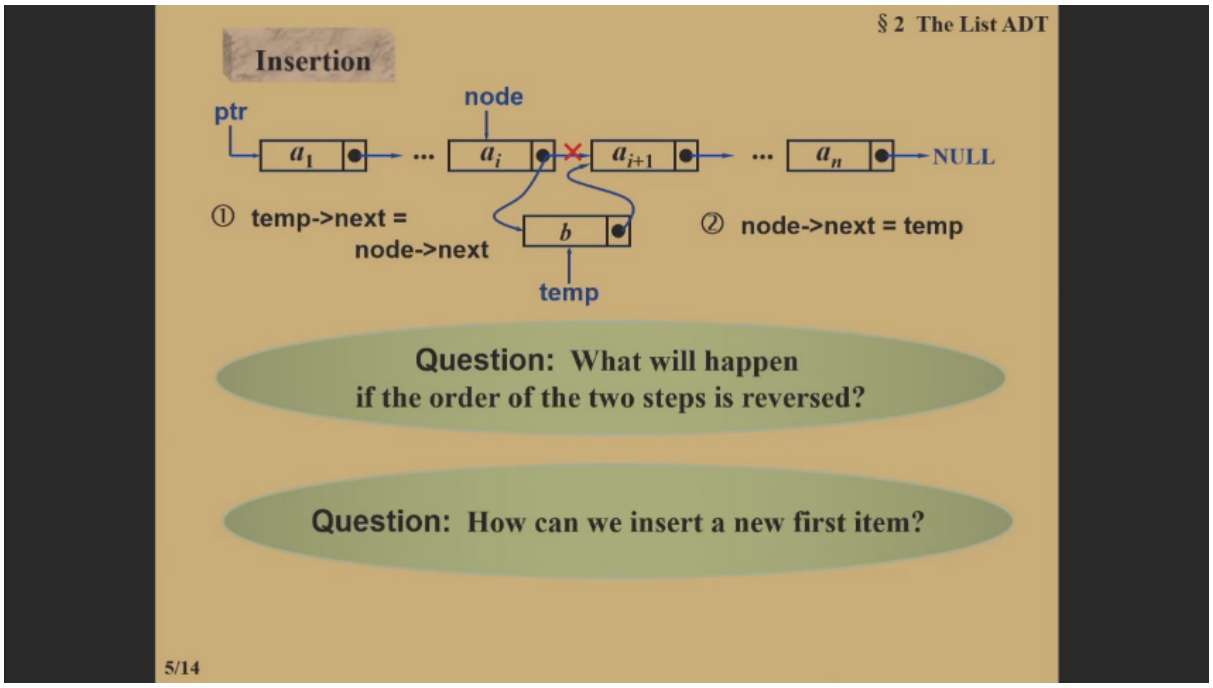
- negative
  - MaxSize 要预先制定，系统要确定开创的Buffer
  - Insertion and Deletion 不可以
- Positive：Find_Kth只需要$O(1)$的time

2.2 Linked Lists 链表



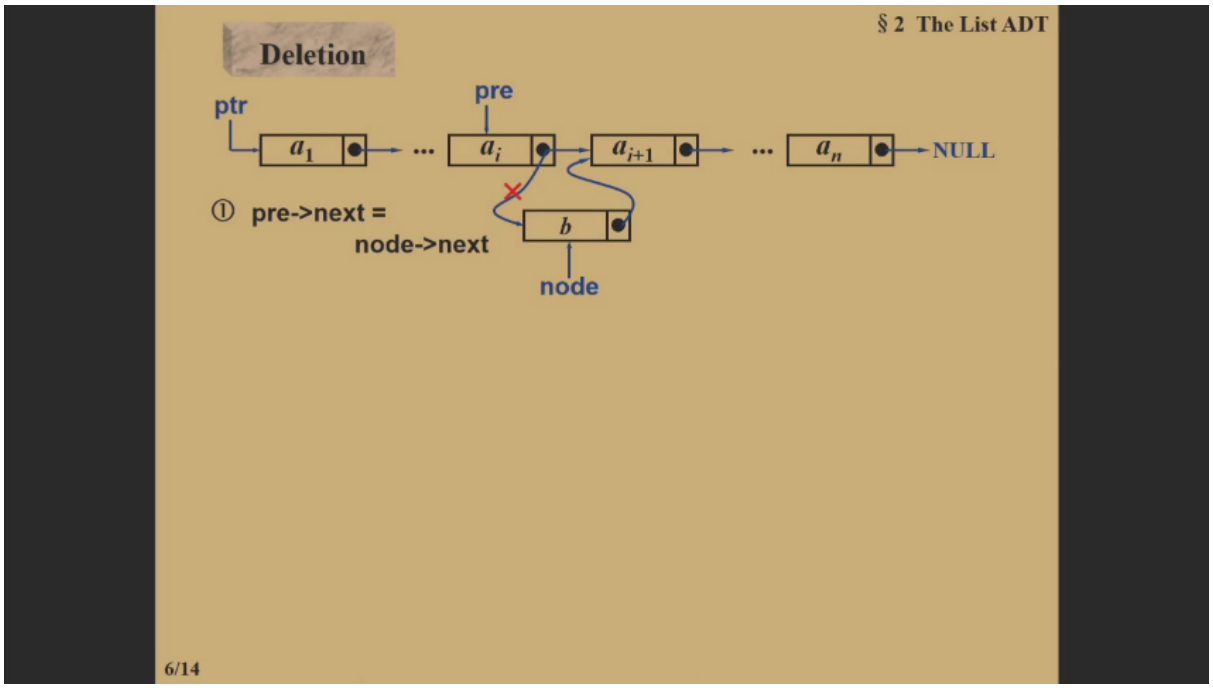
- 每个人只知道自己下一家是谁，不知道上一家是谁
- 每次malloc 比较花时间

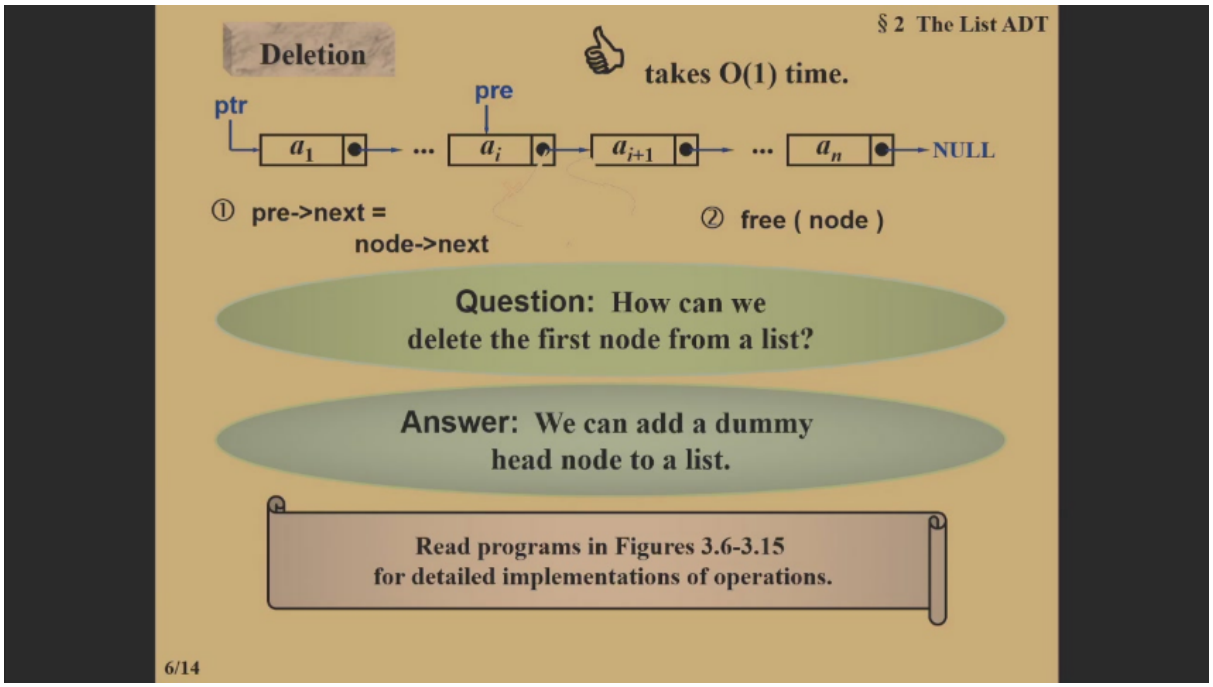
2.2.1 Insertion



- 注意先后不能反
- 否则会造成断掉，内存泄漏

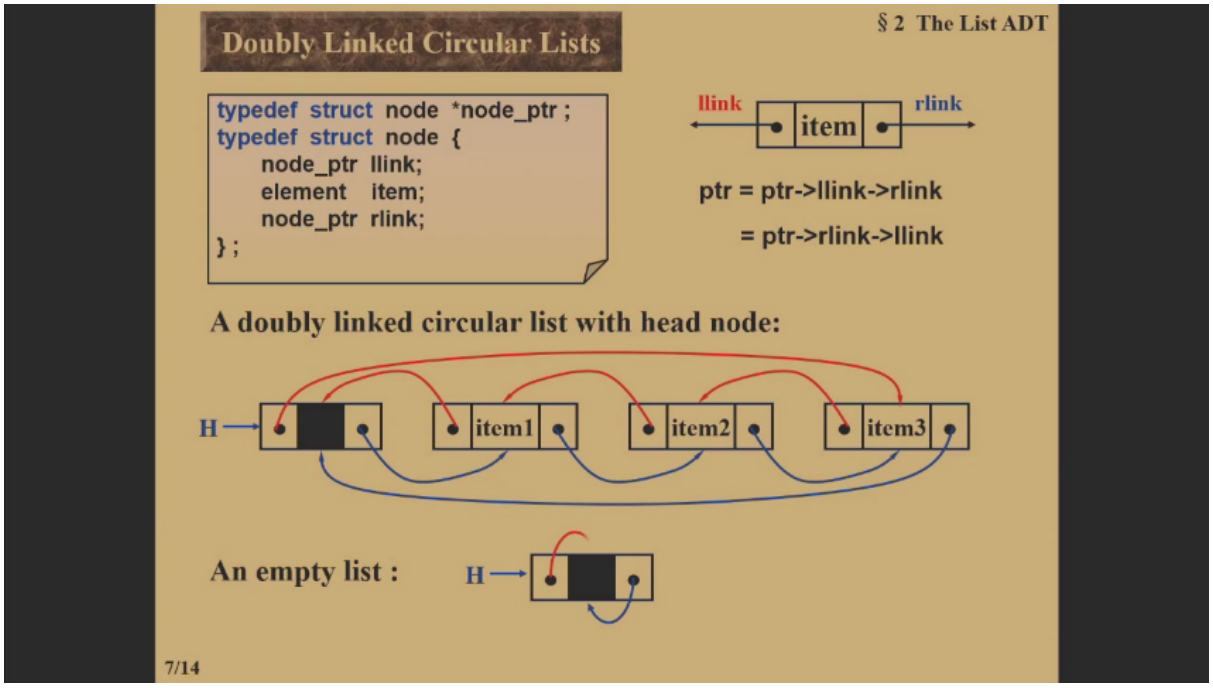
2.2.2 Deletion





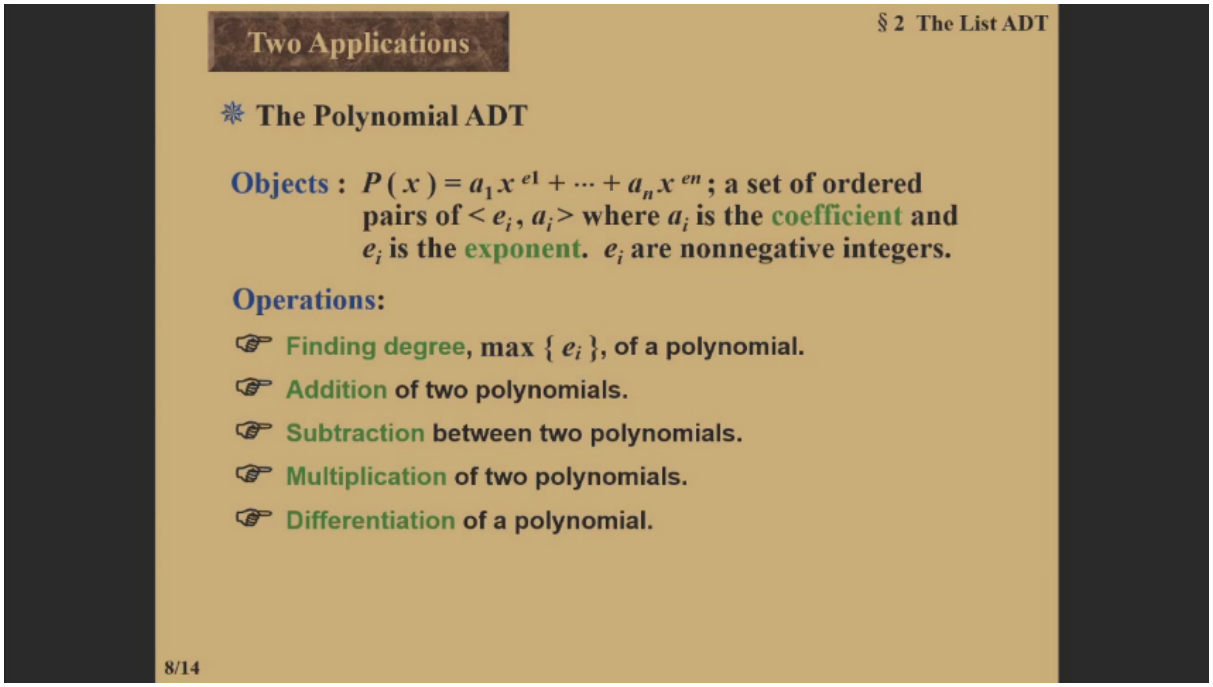
2.3 Doubly Linked Circular Lists 双向链表

2.3.1 Defination

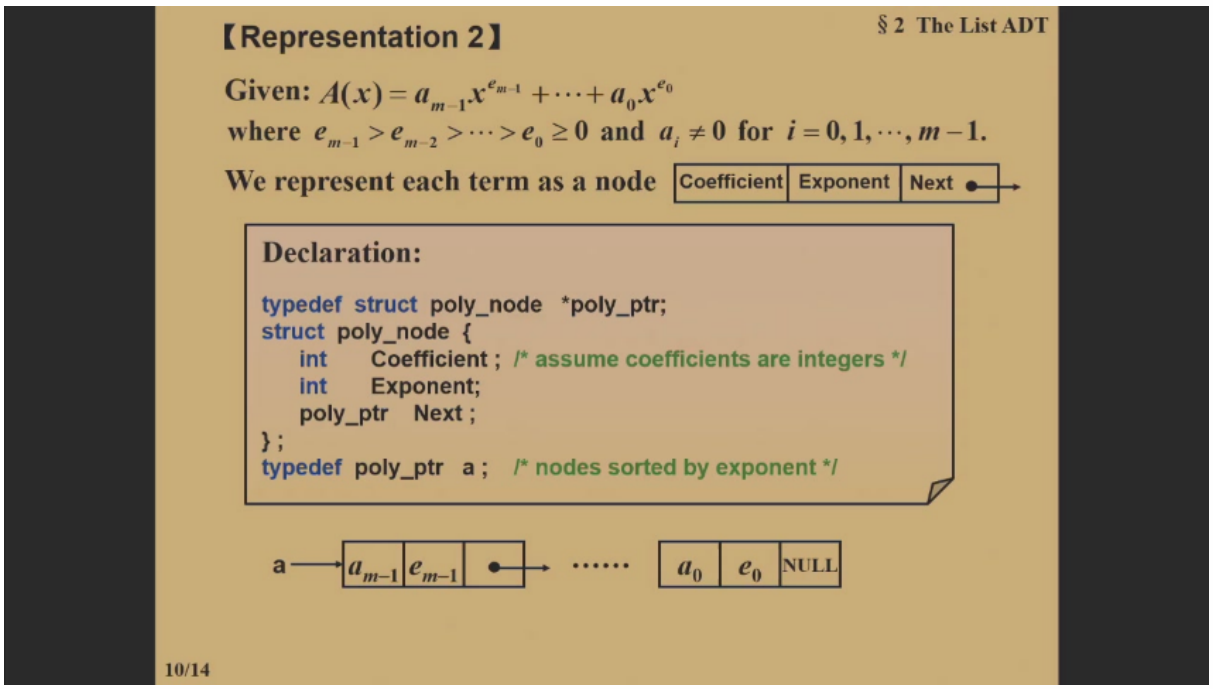



```
ptr = ptr -> llink -> rlink
ptr = ptr -> rlink -> llink
```

2.3.2 Two Applications



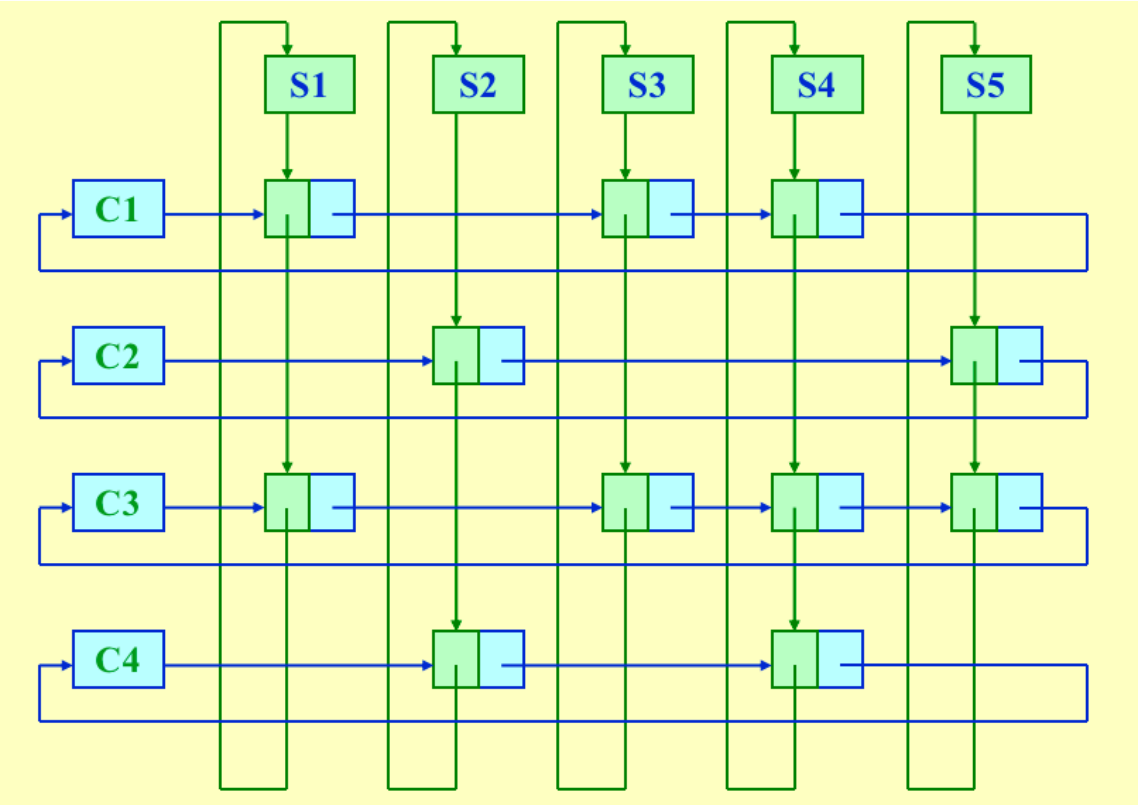

```c
// Representation 1
typedef struct{
  int CoeffArray[ MaxDegree + 1];
  int HighPower;
}
```



```c
// Representation 2
typedef struct poly_node *poly_ptr; // 先定义一种结构类型，在下面再具体表示
// 其中*poly_ptr是指向结构类型的指针，这种指针我先tpyedef成poly_ptr了，这个被指向的结构是poly_node
struct poly_node{
  int Coefficient;
  int Exponent;
  poly_ptr Next; // Next是指向结构体poly_node的指针
};
typedef poly_ptr a;
// 单纯结构的话，只能用 poly_node.Exponent; 结构指针用 poly_node->Exponent
```

2.4 Multilists

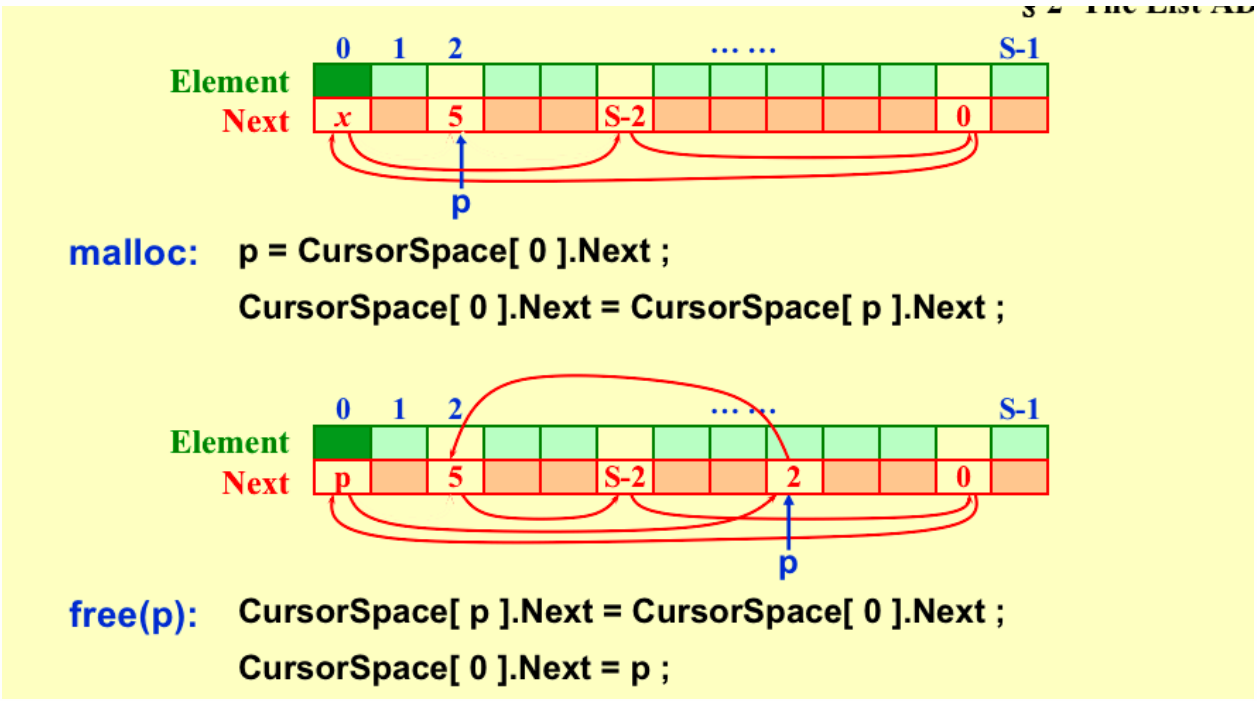
> Example: Suppose that we have 40,000 students and 2,500 courses. Print the students' name list for each course, and print the registered classes' list for each student.



- 这样存储的数据就少了很多

2.5 Cursor Implementation of Linked Lists (no pointer)

> Features that a linked list must have:
>
> a. The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.
>
> b. 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.

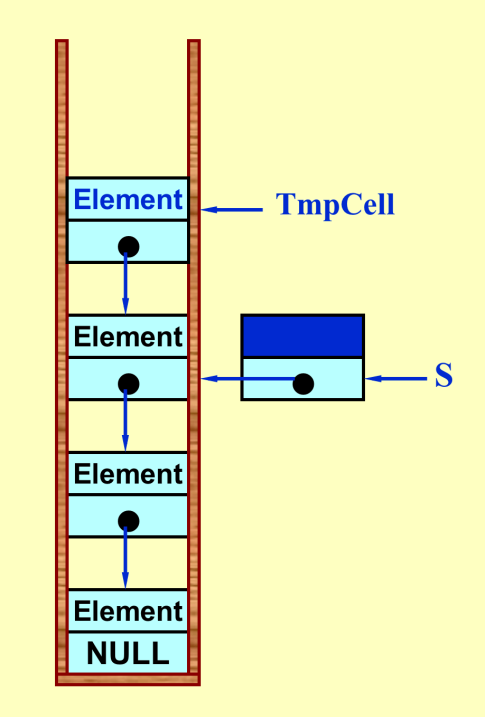
3.2 Operations

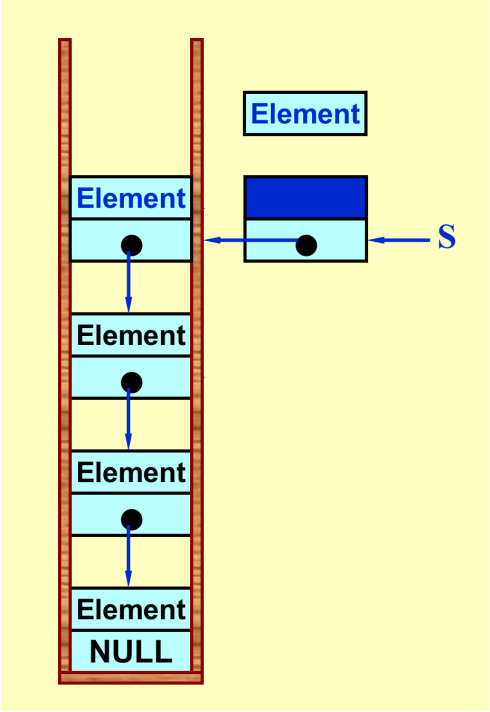
```
Int  IsEmpty( Stack S );

Stack CreateStack( );

DisposeStack( Stack S );

MakeEmpty( Stack S );

Push( ElementType X, Stack S );

ElementType  Top( Stack S );

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 Inplementation

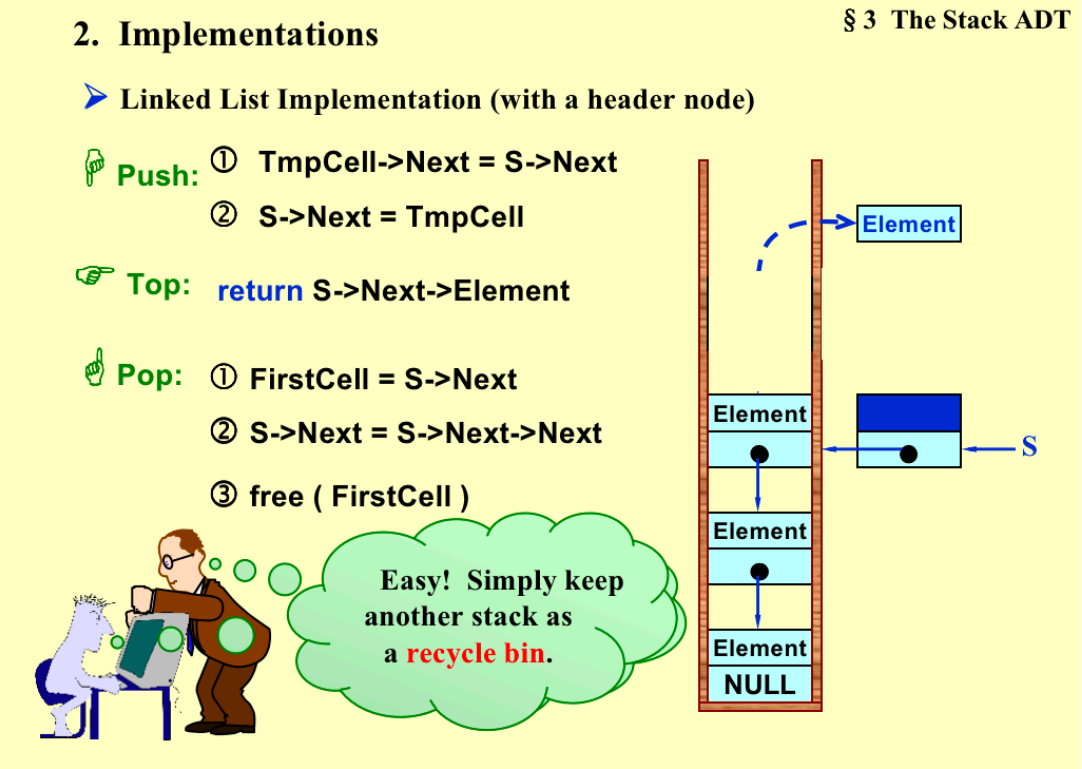
3.3.1 Push





- 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



- 实际上一对malloc和free很花时间，可以再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

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

> 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

```
Algorithm  {
    Make an empty stack S;
    while (read in a character c) {
        if (c is an opening symbol)
            Push(c, S);
        else if (c is a closing symbol) {
            if (S is empty)  { ERROR; exit; }
            else  {  /* stack is okay */
                if  (Top(S) doesn't match c)  { ERROR, exit; }
                else  Pop(S);
            }  /* end else-stack is okay */
        }  /* end else-if-closing symbol */
    } /* end while-loop */
    if (S is not empty)  ERROR;
}
```

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

3.4.2 Postfix Evaluation

§ 3 The Stack ADT

Postfix Evaluation

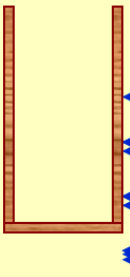
〖Example〗 An infix expression: $a + b * c - d / e$

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

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

Reverse Polish notation | operand | operator with the highest precedence | operator

〖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

§ 3 The Stack ADT

Infix to Postfix Conversion

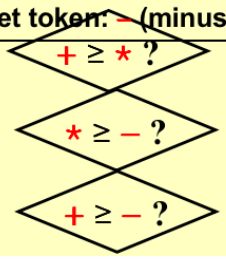
〖Example〗 $a + b * c - d = a\ b\ c * + d -$

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)	

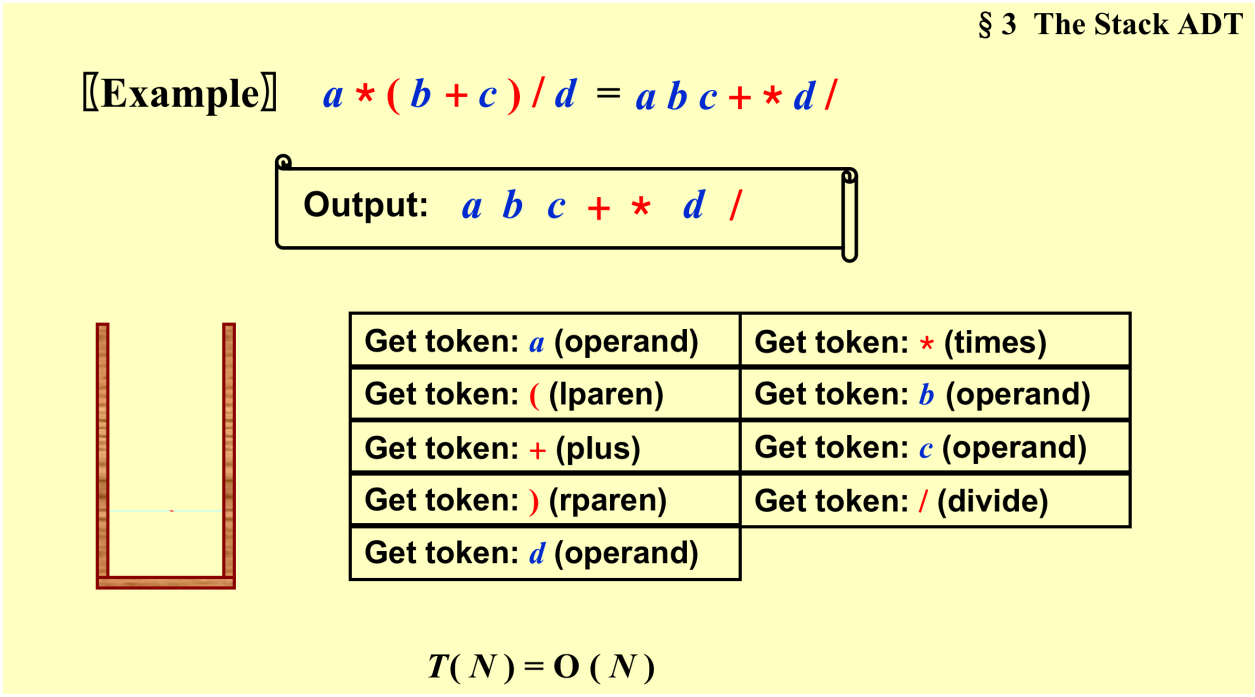


12

- 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



- 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

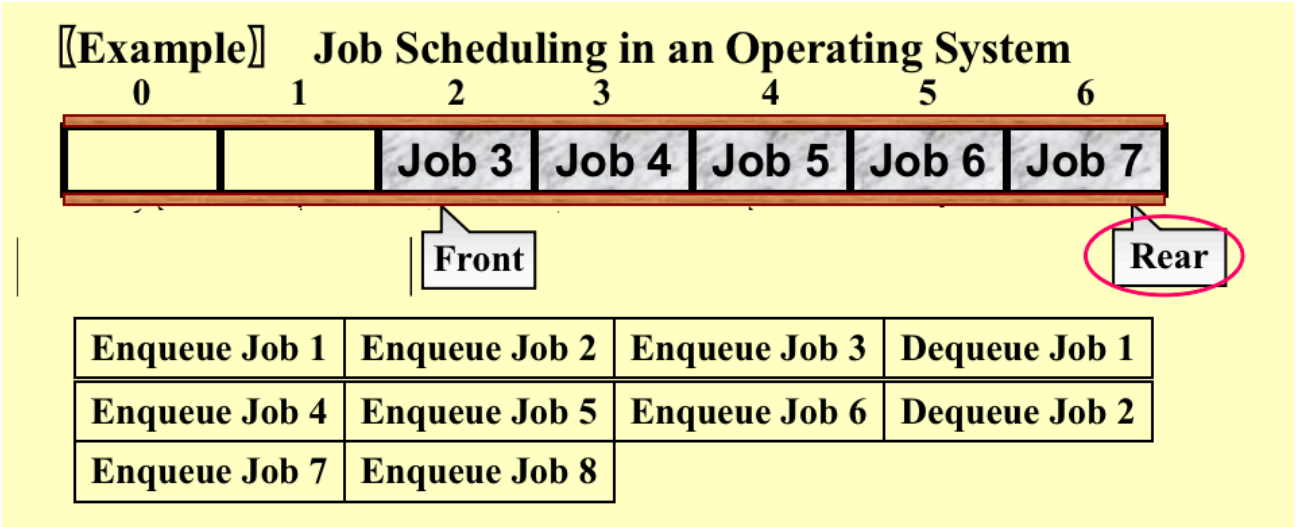
> A queue is a First-In-First-Out (FIFO) list, that is, an ordered list in which insertions take place at one end and deletions take place at the opposite end.

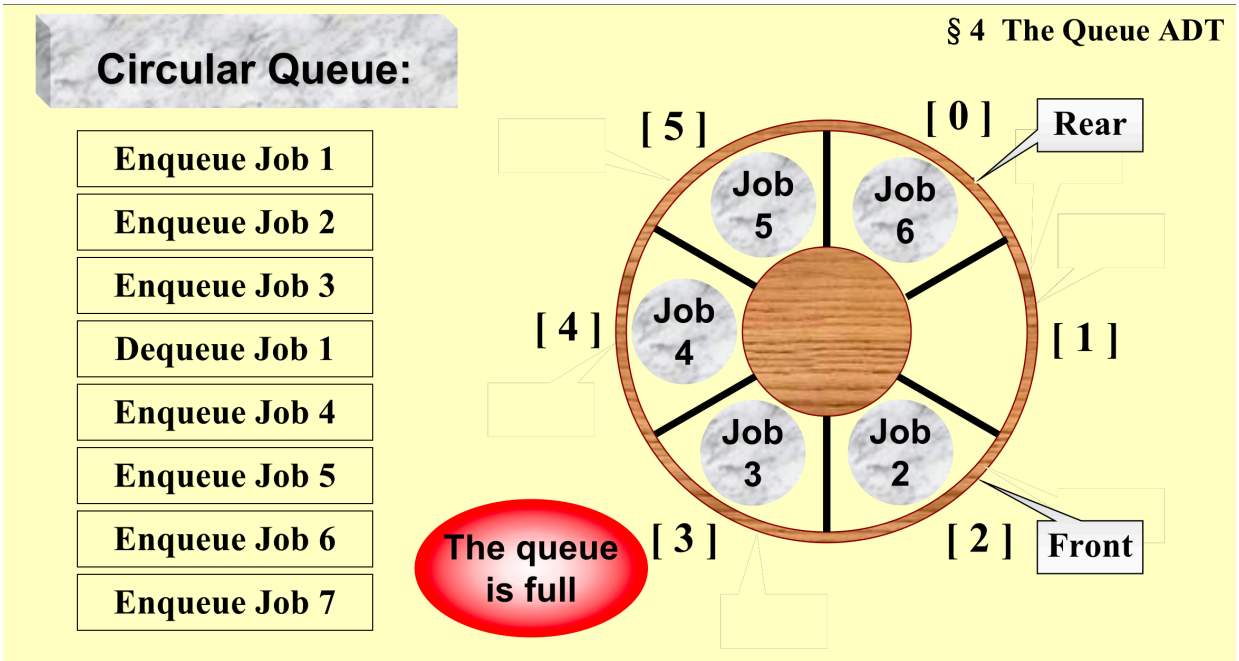
4.2 Operations:

```
int  IsEmpty( Queue Q );
Queue CreateQueue( );
DisposeQueue( Queue Q );
MakeEmpty( Queue Q );
Enqueue( ElementType X, Queue Q );
ElementType  Front( Queue Q );
Dequeue( Queue Q );
```

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

```
struct  QueueRecord {
  int     Capacity ;   /* max size of queue */
  int     Front;          /* the front pointer */
  int     Rear;           /* the rear pointer */
  int     Size;  /* Optional - the current size of queue */
  ElementType  *Array;    /* array for queue elements */
 } ;
```





> 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?
  -

$$rear = (front + size - 1)\%m \tag{1}$$