

Lists, Stacks and Queues

List ADT

- Objects: item0, item1, ...
- Operations
 - length
 - print
 - making an empty list
 - finding kth item from the list
 - inserting a new item after kth item
 - deleting an item from the list
 - finding next of the current item
 - finding previous of the current item

Array Implementation

1. Simple Array implementation of Lists

§ 2 The List ADT

$\text{array}[i] = \text{item}_i$

Sequential mapping

Address	Content
.....
array+i	item _i
array+i+1	item _{i+1}
.....



MaxSize has to be estimated.



Find_Kth takes O(1) time.



Insertion and **Deletion** not
only take O(N) time, but also
involve a lot of data movements
which takes time.



Linked lists

2. Linked Lists

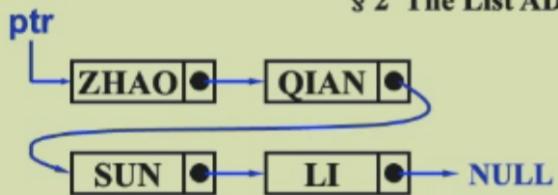
Address	Data	Pointer
0010	SUN	1011
0011	QIAN	0010
0110	ZHAO	0011
1011	LI	NULL

Head pointer ptr = 0110

Initialization:

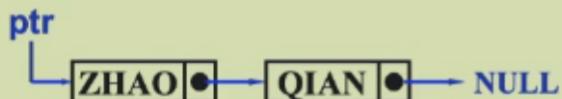
```
typedef struct list_node *list_ptr;
typedef struct list_node {
    char     data [ 4 ];
    list_ptr next ;
} ;
list_ptr ptr ;
```

§ 2 The List ADT



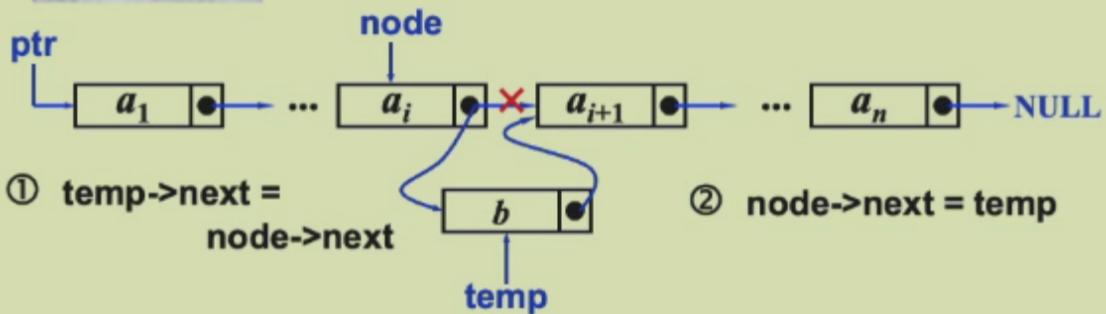
To link 'ZHAO' and 'QIAN':

```
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 ;
```



§ 2 The List ADT

Insertion



- 双向链表

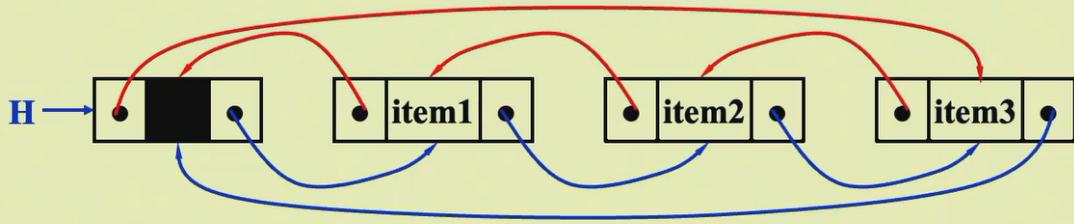
Doubly Linked Circular Lists

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



$$\begin{aligned} \text{ptr} &= \text{ptr} \rightarrow \text{llink} \rightarrow \text{rlink} \\ &= \text{ptr} \rightarrow \text{rlink} \rightarrow \text{llink} \end{aligned}$$

A doubly linked circular list with head node:



An empty list :



set a dummy head

- 查找环：快慢指针

Applications

Two Applications

§ 2 The List

* 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.

【Representation 1】

```
typedef struct {
    int    CoeffArray [ MaxDegree + 1 ];
    int    HighPower;
} *Polynomial ;
```

数组实现，计算简单，对稀疏数据不友好

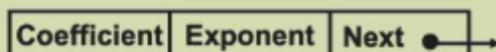
【Representation 2】

§ 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 $i = 0, 1, \dots, m-1$.

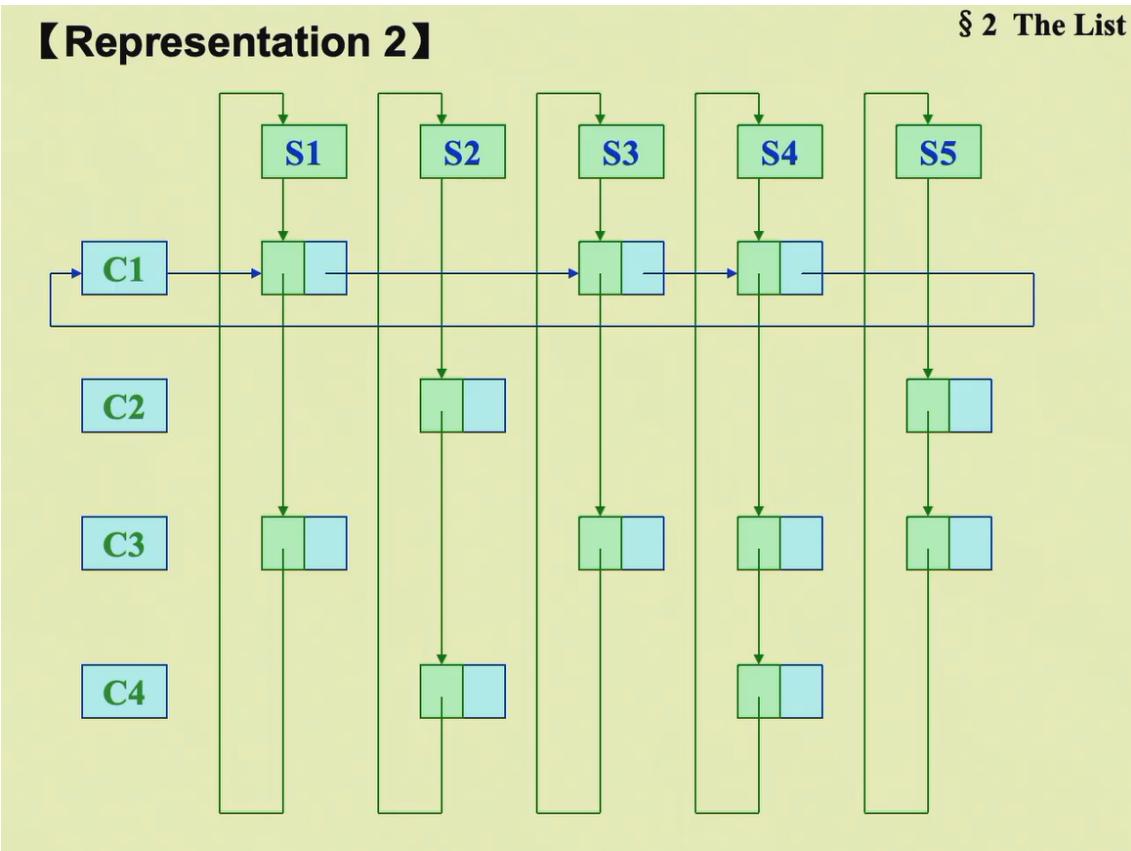
We represent each term as a node



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 */
```

- multilist

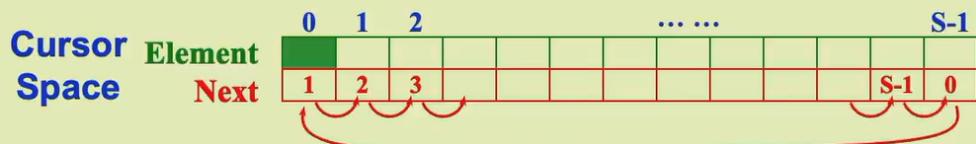


- Cursor Implementation of Linked Lists

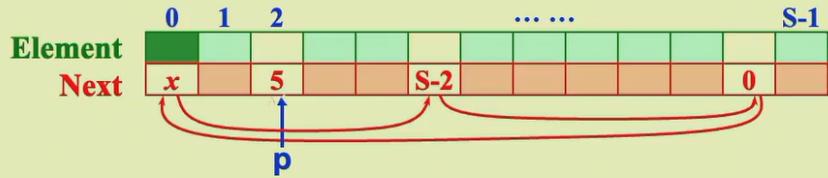
3. Cursor Implementation of Linked Lists (no pointer)

☞ Features that a linked list must have:

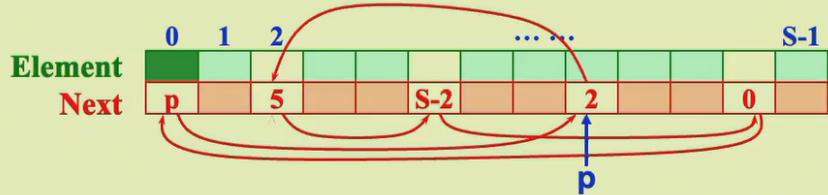
- a) The data are stored in a collection of structures. Each structure contains **data** and a pointer to the **next** structure.
- b) A new structure can be obtained from the system's global memory by a call to **malloc** and released by a call to **free**.



Note: The interface for the cursor implementation (given in Figure 3.27 on p. 52) is identical to the pointer implementation (given in Figure 3.6 on p. 40).



```
malloc: p = CursorSpace[ 0 ].Next ;
CursorSpace[ 0 ].Next = CursorSpace[ p ].Next ;
```



```
free(p): CursorSpace[ p ].Next = CursorSpace[ 0 ].Next ;
CursorSpace[ 0 ].Next = p ;
```

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.

```
position cursor_alloc( void )
{
    position p;
    p = CURSOR_SPACE[0].next;
    CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;
    return p;
}

void cursor_free( position p)
{
    CURSOR_SPACE[p].next = CURSOR_SPACE[0].next;
    CURSOR_SPACE[0].next = p;
}
```

Figure 3.30 Routines: cursor-alloc and cursor-free

Slot	Element	Next
0	-	6
1	b	9
2	f	0
3	header	7
4	-	0
5	header	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

1. 整个数组模拟了一个global memory
2. 从0开始, 形成的一个环就是freelist
3. alloc时, 从这个环中删除一个元素
4. free时, 将这个元素插入这个环

Stack ADT

ADT

- Last in first out(LIFO)

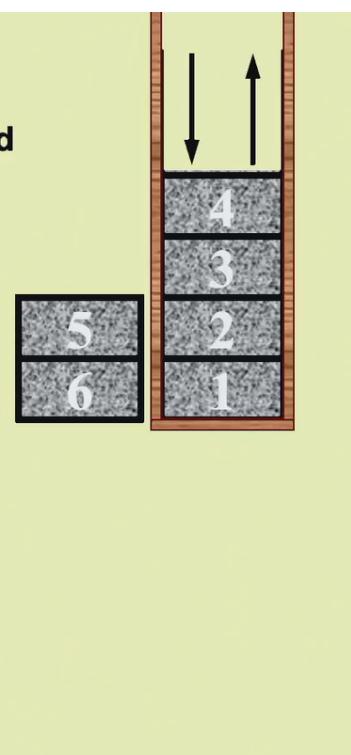
1. ADT

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.

Objects: A finite ordered list with zero or more elements.

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);**



- operation only at top
- push: insert, pop: insert
- ADT error & implementation error

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.

Implementation

- Linked list implementation

➤ Linked List Implementation (with a header node)

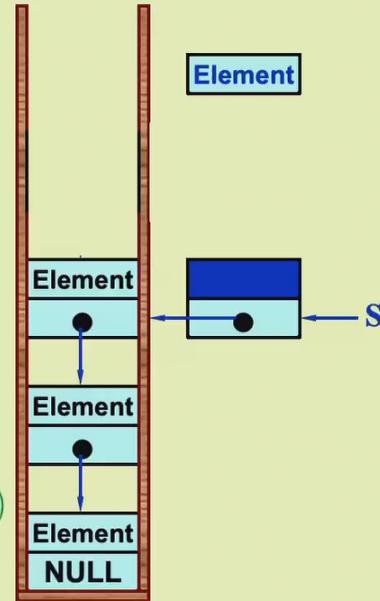
 **Push:** ① `TmpCell->Next = S->Next`
② `S->Next = TmpCell`

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

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



But, the calls to malloc and free are expensive.



- Recycle bin

维护另一个Stack: Recycle bin, 在本栈pop后不执行free, push到rb里。要执行push时, 从rb里pop出来一个。

- Array Implementation

➤ Array Implementation

```
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: ① 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.
 ② Error check must be done before **Push** or **Pop (Top)**.

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

Applications

Balancing Symbols



Check if parenthesis (), brackets [], and braces { } are balanced.

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;
```

Postfix Evaluation

- 遇到operand入栈，遇到operator从栈中取数计算
- 计算后的结果继续push进去
- 结束后pop得到答案
- error: 出现pop空栈

Infix to Postfix Conversion

- 遇到operand输出，栈维护operator
- 若operator<=栈顶，将栈顶pop出来
- 若operator大于栈顶，operator入栈

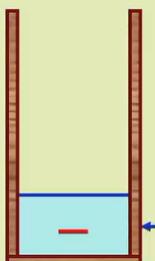
§ 3 The Stack ADT

* Infix to Postfix Conversion

【Example】 $a + b * c - d = a b c * + d -$

Note:

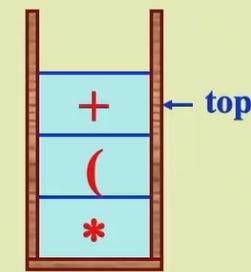
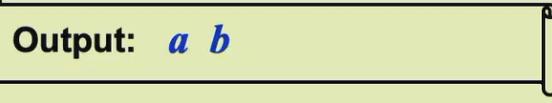
- The order of operands is the same in infix and postfix.
- Operators with higher precedence appear before those with lower precedence.



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

- 有括号的情况，左括号不进stack优先级最高，左括号进stack后优先级最低
- 遇到右括号pop至左括号

【Example】 $a * (b + c) / d = a b c + * d /$



Get token: a (operand)	Get token: * (times)
Get token: ((lparen)	Get token: b (operand)
Get token: + (plus)	

Solutions:

- ① Never pop a **(** from the stack except when processing **a)**.
- ② Observe that when **(** is **not in** the stack, its precedence is the **highest**; but when it is **in** the stack, its precedence is the **lowest**. Define **in-stack** precedence and **incoming** precedence for symbols, and each time use the corresponding precedence for comparison.

Note: $a - b - c$ will be converted to $a b - c -$. However, 2^2^3 (2^{2^3}) must be converted to $2 2 3 ^ ^$, not $2 2 ^ 3 ^$ since exponentiation associates **right to left**.

Function Calls

* Function Calls -- System Stack

§ 3 The Stack ADT



```
void PrintList ( List L )
{
    if ( L != NULL ) {
        PrintElement ( L->Element );
        PrintList( L->next );
    }
} /* a bad use of recursion */
```

```
void PrintList ( List L )
{
    top: if ( L != NULL ) {
        PrintElement ( L->Element );
        L = L->next;
        goto top; /* do NOT do this */
    }
} /* compiler removes recursion */
```

Queue ADT

ADT

First In First Out (FIFO)

§ 4 The Queue ADT

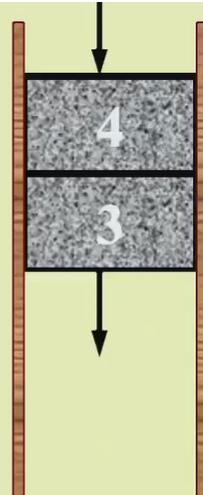
1. ADT

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.

Objects: A finite ordered list with zero or more elements.

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);`



Implementation

- Linked list

本身就是队列

- Array

【Example】 Job Scheduling in an Operating System

0 1 2 3 4 5 6

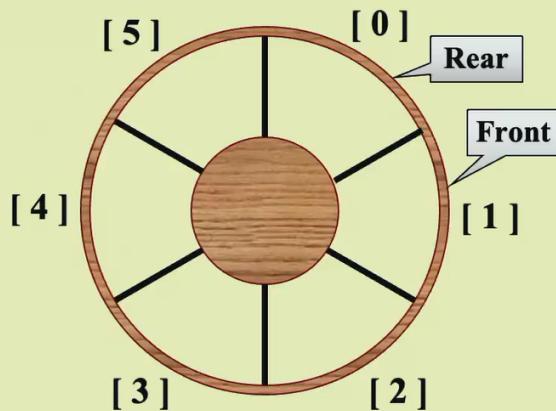
Job 1 | Job 2 | Job 3 |

Front

Rear

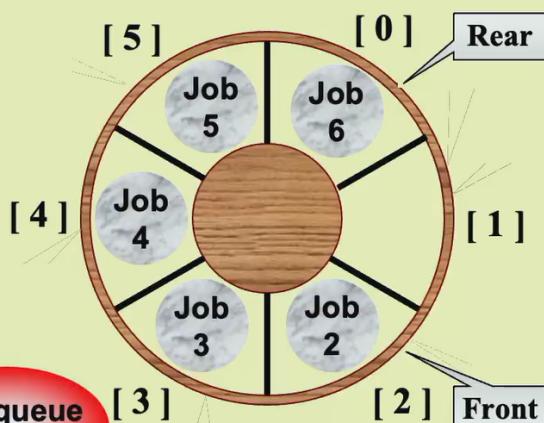
Enqueue Job 1 | Enqueue Job 2 | Enqueue Job 3 | Dequeue Job 1 |

- Enqueue: rear++
- Dequeue: front++
- 如何解决队尾问题: Circular Queue

Circular Queue:**Circular Queue:**

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

The queue
is full



否则无法区分满or空

- Method 1: 空一格
- Method 2: 加一个size字段