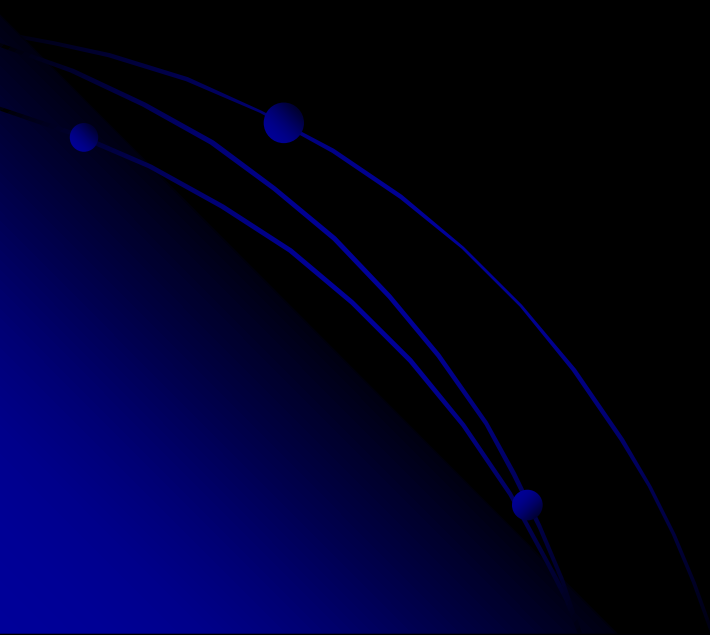




Chapter 03 Stack & Queue

第三章 栈和队列

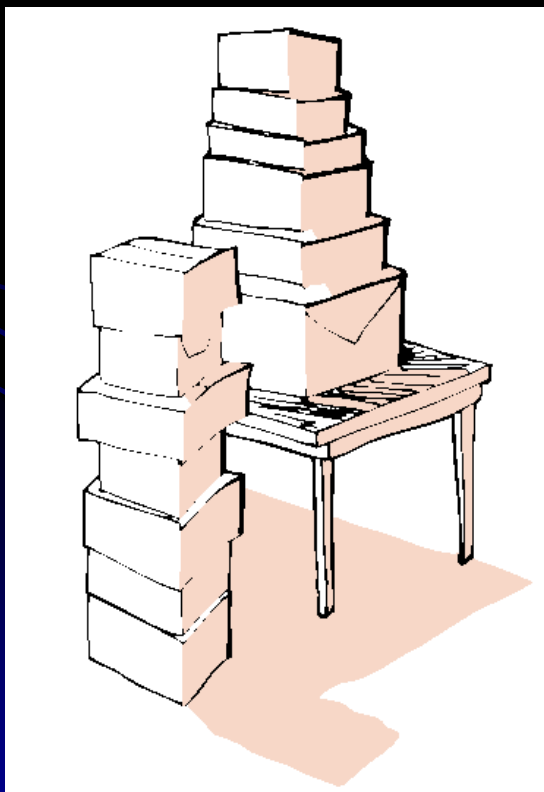


Content

- Stack and its ADT
- Representation & Implementation of Stack
- Application of Stack
- Recursion and Stack
- Queue and its ADT
- Representation & Implementation of Queue
- Application of Queue
- Conclusion

Chapter 03 Stack & Queue

栈和队列也是线性表，只不过是操作受限的线性表。但从数据类型角度看，它们是和线性表不相同的两类重要的抽象数据类型。广泛应用于各种软件系统中。



3.1 Stack and its ADT

- Definition

- A stack is a kind of special linear list.
- A stack is a data structure in which all insertions and removals of entries are made at one end, called the top of the stack.
- **LIFO** structure: The last entry which was inserted is the first one that will be removed.
(**LIFO**: Last In First Out)

Stack ADT

ADT Stack {

数据对象:

$$D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$$

数据关系:

$$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$$

约定 a_n 端为栈顶, a_1 端为栈底。

基本操作:

} ADT Stack

Top & Bottom

- **Top**

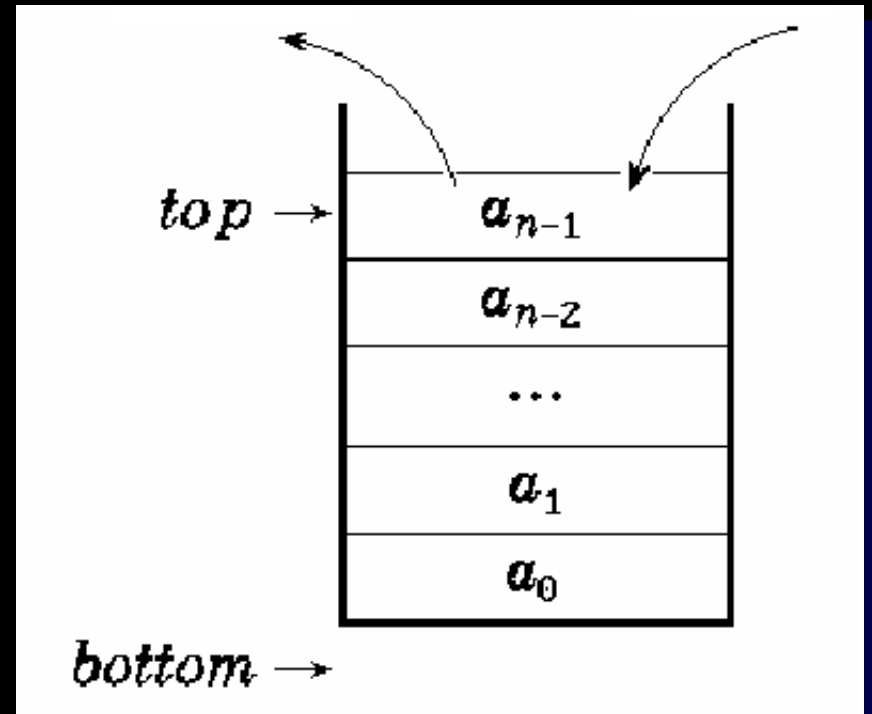
- The specific end of linear list at which element is inserted and removed.

- **Bottom**

- Another end of the stack.
- Fixed

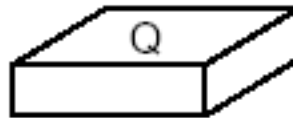
Removal
(Pop)

Insertion
(Push)



Examples of Push and Pop

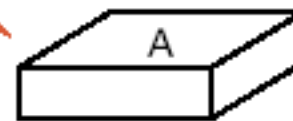
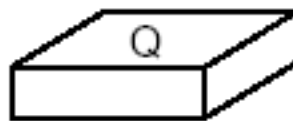
Push box Q onto empty stack:



Push box A onto stack:

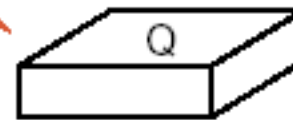


Pop a box from stack:

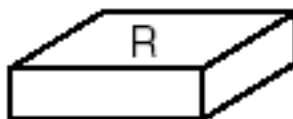


Pop a box from stack:

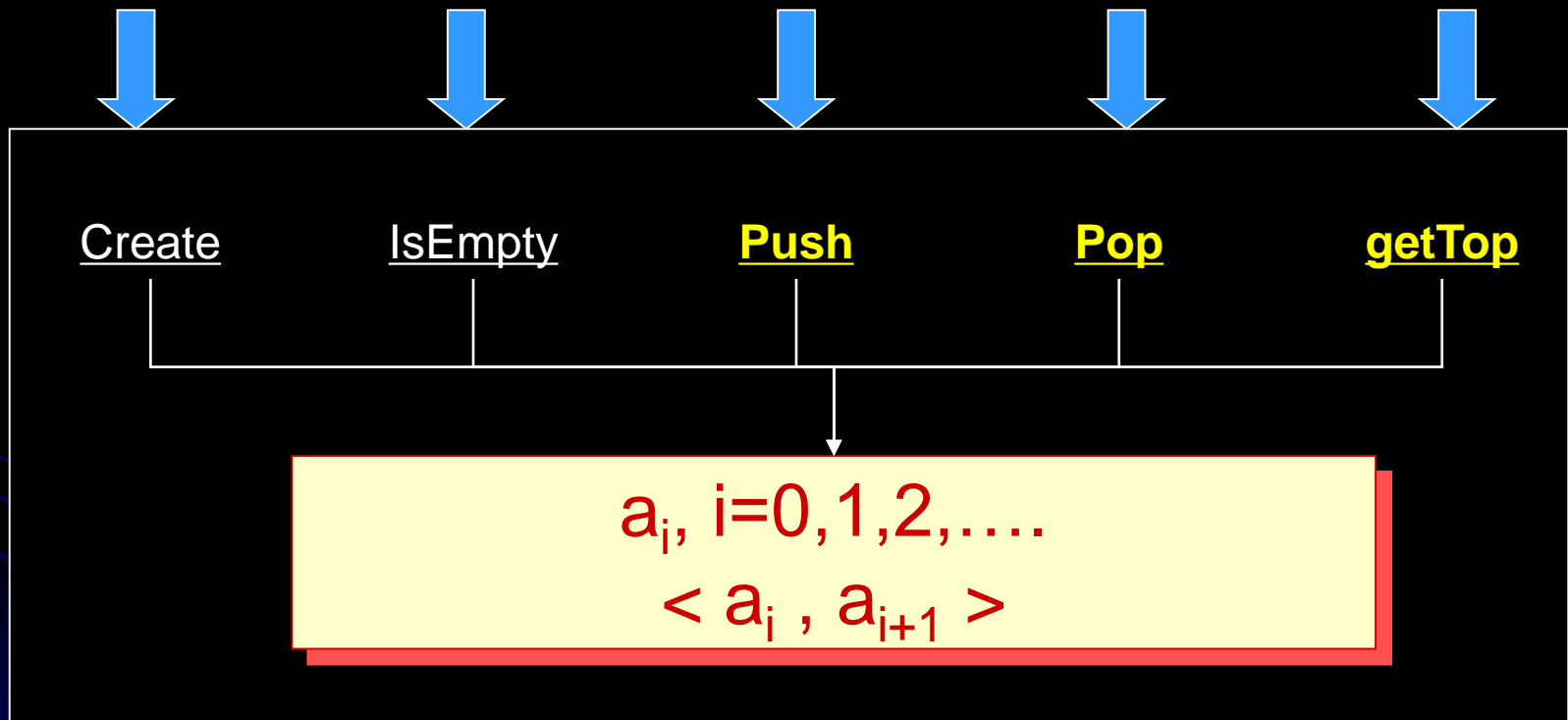
(empty)



Push box R onto stack:



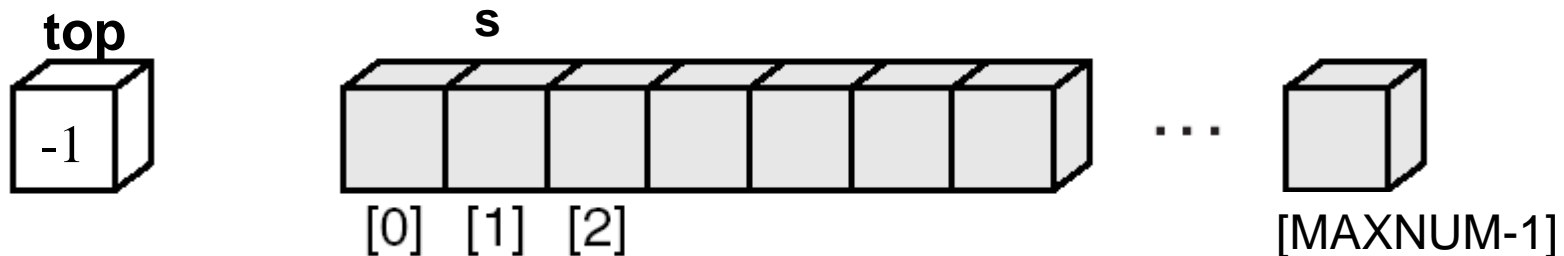
Basic operations



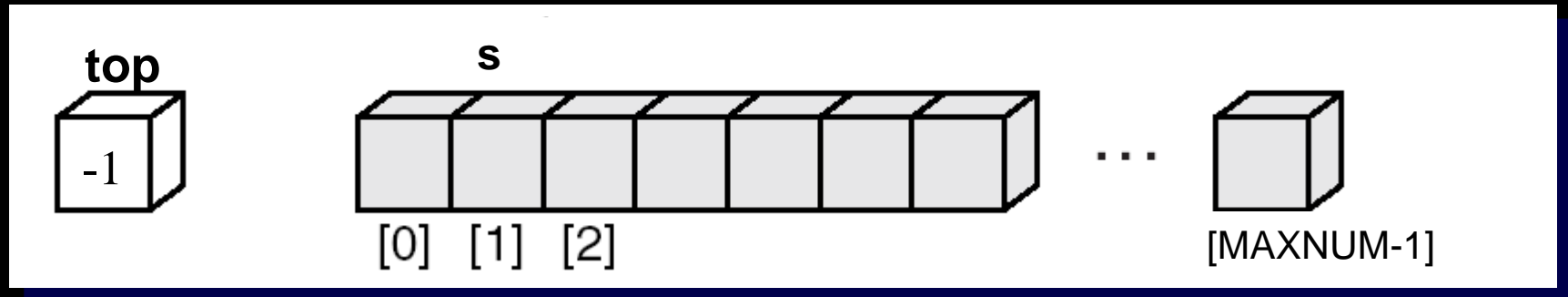
3.2 Implementation of Stack

1) Sequential form

```
typedef int ElemType;          /* 定义栈元素的数据类型,
                                这里定义为整型 */
#define MAXNUM 100             /* 栈中能达到的最大容量,
                                这里设为100 */
typedef struct SeqStack        /* 顺序栈类型定义 */
{
    ElemType s[MAXNUM];
    int top; /* 栈顶指针 */
}SeqStack, *PSeqStack;
```



Sequential Stack



Create

IsEmpty

Push

Pop

GetTop

Algorithm 3.1 Initialization

```
PSeqStack createEmptyStack_seq ( void )  
{  
    PSeqStack pastack;  
  
    pastack = (PSeqStack) malloc(sizeof(SeqStack));  
    if (pastack==NULL)  
        printf("Out of space!! \n");  
    else  
        pastack->top=-1;  
    return pastack;  
}
```



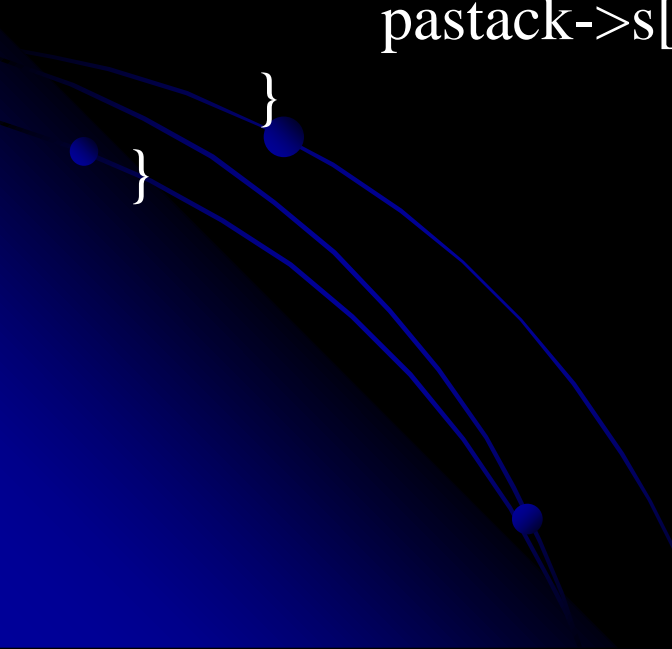
Algorithm 3.2 Judge a stack is empty or not

```
int isEmptyStack_seq ( PSeqStack pastack )  
{  
    return ( pastack->top == -1 );  
}
```



Algorithm 3.3 Push an element into the stack

```
void push_seq ( PSeqStack pastack, ElemType x )  
/* 在栈中压入一元素x */  
{  
    if ( pastack->top >= MAXNUM - 1 )  
        printf( "overflow! \n" );  
    else {  
        pastack->top = pastack->top + 1;  
        pastack->s[pastack->top] = x;  
    }  
}
```



Algorithm 3.4 Pop the top element from the stack

```
ElemType pop_seq( PSeqStack pastack )  
/* 删除栈顶元素 */  
{  
    ElemType temp;  
    if ( isEmptyStack_seq( pastack ) )  
        printf( "Underflow!\n" );  
    else {  
        temp = pastack.s[pastack->top];  
        pastack->top = pastack->top - 1;  
    }  
    return temp;  
}
```



Algorithm 3.5 Get the value of top element

```
ElemType top_seq( PSeqStack pastack )  
/* 当pastack所指的栈不为空栈时，求栈顶元素的值 */  
{  
    if (isEmptyStack_seq(pastack))  
        Error("Empty Stack!");  
    else  
        return pastack->s[pastack->top];  
}
```

Error("Empty Stack!"); \longleftrightarrow printf("Empty Stack!");



Quiz

1. 设一个栈的输入序列为A,B,C,D,则借助一个栈所得到的输出序列不可能是__。

(A) A,B,C,D

(B) D,C,B,A

(C) A,C,D,B

(D) D,A,B,C

答:可以简单地推算,得容易得出D,A,B,C是不可能的,因为D先出来,说明A,B,C,D均在栈中,按照入栈顺序,在栈中顺序应为D,C,B,A,出栈的顺序只能是D,C,B,A。所以本题答案为D。

Quiz

2. 已知栈的输入序列为 $1, 2, 3, \dots, n$ 。输出序列为 a_1, a_2, \dots, a_n ，若 $a_1 = n$ ，问 $a_i = ?$

$$a_i = n - i + 1$$

3. I表示入栈，O表示出栈，若元素入栈顺序为1234，为了得到1342的出栈顺序，相应的I和O的操作串是什么？

$$I_1 O_1 I_2 I_3 O_3 I_4 O_4 O_2$$

Quiz

4. 设 n 个元素进栈序列是 $1, 2, 3, \dots, n$, 其输出序列是 p_1, p_2, \dots, p_n , 若 $p_1 = 3$, 则 p_2 的值__。

(A) 一定是2

(B) 一定是1

(C) 不可能是1

(D) 以上都不对

答:当 $p_1 = 3$ 时, 说明 $1, 2, 3$ 先进栈, 立即出栈 3 , 然后可能出栈, 即为 2 , 也可能 4 或后面的元素进栈, 再出栈。因此, p_2 可能是 2 , 也可能是 $4, \dots, n$, 但一定不能是 1 。所以本题答案为C。

2) Linked list form

```
typedef int DataType ;
```

/* 定义栈中元素类型为整型，
也可定义为其他类型 */

```
struct Node
```

/* 单链表结点结构 */

```
{
```

```
    DataType info;
```

```
    struct Node *link;
```

```
};
```

```
struct LinkStack
```

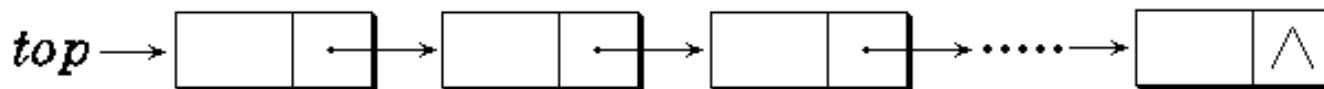
/* 链接栈类型定义 */

```
{
```

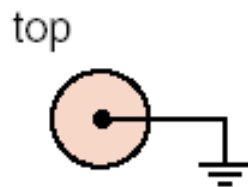
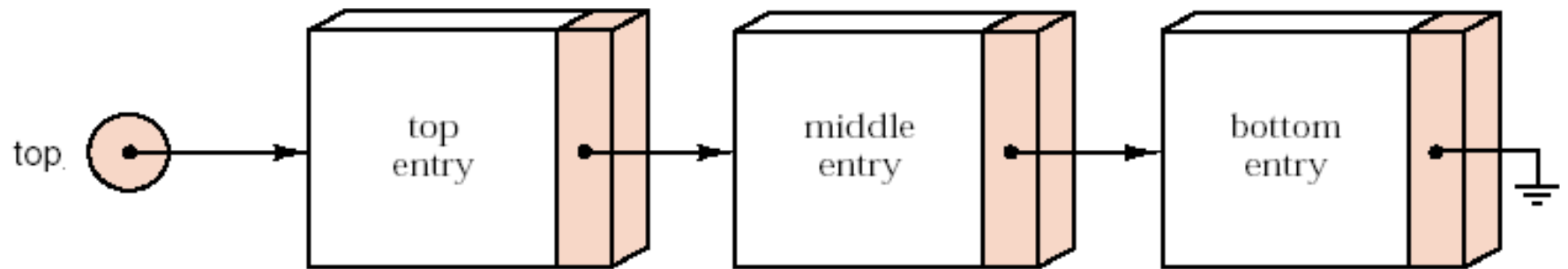
```
    struct Node *top;
```

/* 指向栈顶结点 */

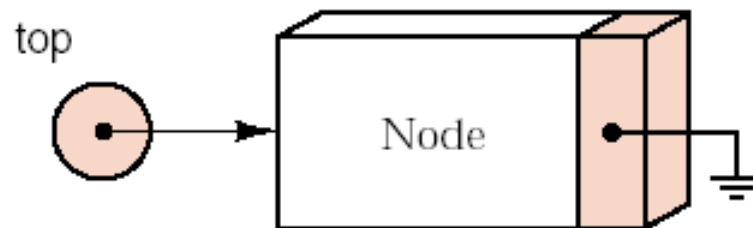
```
}LinkStack, *PLinkStack;
```



Example

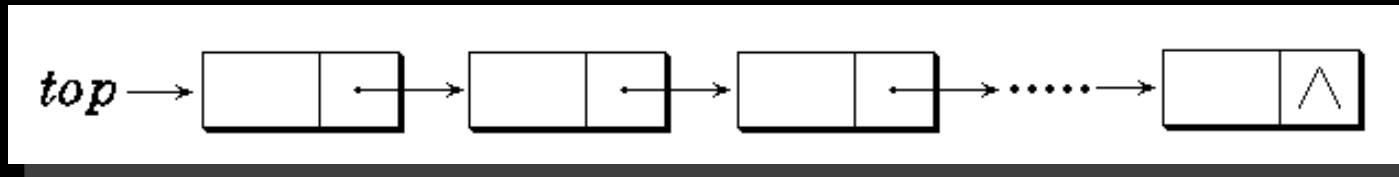


Empty stack



Stack of size 1

Linked stack



Create

IsEmpty

Push

Pop

GetTop

Algorithm 3.6 Initialization

```
PLinkStack createEmptyStack_link( )  
{  
    PLinkStack plstack;  
  
    plstack = (struct LinkStack *) malloc( sizeof(struct LinkStack));  
    if (plstack != NULL)  
        plstack->top = NULL;  
    else  
        printf("Out of space! \n");  
    return plstack;  
}
```



Algorithm 3.7 Judge a linked stack is empty or not

```
int isEmptyStack_link( PLinkStack plstack )  
{  
    return (plstack->top == NULL);  
}
```



Algorithm 3.8 Push an element into the linked stack

```
void push_link( PLinkStack plstack, DataType x )  
/* 在栈中压入一元素x */  
{  
    struct Node *p;  
    p = (struct Node *) malloc( sizeof( struct Node ) );  
    if ( p == NULL )  
        printf("Out of space!\n");  
    else {  
        p->info = x;  
        p->link = plstack->top;  
        plstack->top = p;  
    }  
}
```



Algorithm 3.9 Pop the top element from the linked stack

DataType **pop_link**(PLinkStack plstack)

/* 在栈中删除栈顶元素 */

```
{  
    struct Node *p;  
    DataType elem;  
    if ( isEmptyStack_link( plstack ) )  
        printf( "Empty stack pop.\n" );  
    else {  
        p = plstack->top;  
        elem = p->info;  
        plstack->top = plstack->top->link;  
        free(p);  
    }  
    return elem;  
}
```

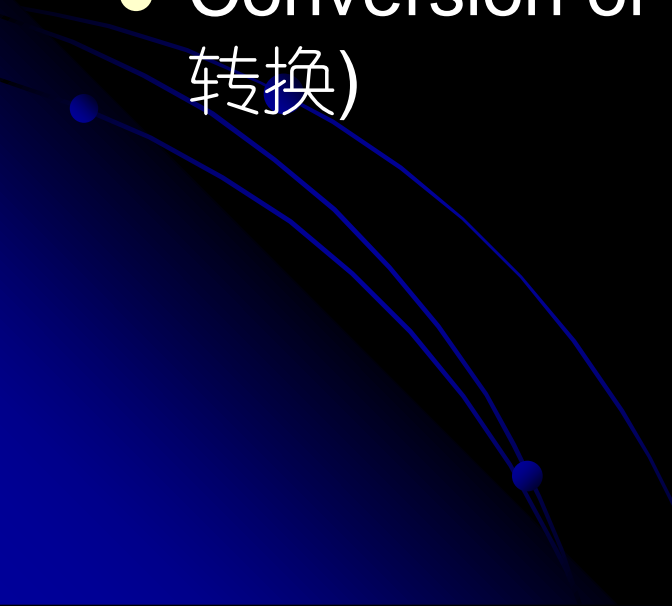


Algorithm 3.10 Get the value of the top element

```
DataType top_link( PLinkStack plstack )  
/* 对非空栈求栈顶元素 */  
{  
    if ( isEmptyStack_link(plstack) )  
        Error("Empty Stack!");  
    else  
        return plstack->top->info;  
}
```



3.3 Applications of Stack

- Number system conversion (数制转换)
 - Bracket matching (括号匹配)
 - Infix expression calculator (中缀表达式计算器)
 - Reverse Poland calculator (逆波兰计算器)
 - Conversion of the arithmetic expression (表达式转换)
- 

Application 1: Number system conversion

转换方法：将十进制数N转换为d进制的数：

$$N = (N \text{ div } d) * d + N \text{ mod } d$$

其中：div为整除运算,mod为求余运算.

例如: $(1348)_{10} = (2504)_8$, 其运算过程如下:

N_i	$N_i \div 8$	$N_i \bmod 8$
-------	--------------	---------------

算法思想如下:

当 $N \geq 0$, 重复1, 2

1. 若 $N \neq 0$, 则将 $N \bmod d$ 压入栈s中, 执行2; 若 $N = 0$, 将栈S的内容依次出栈, 算法结束。

2. 用 $N \div d$ 代替 N

计算顺序

输出顺序

Application 2: Bracket Matching

设在表达式中

([] ()) 或 [([] [])]

等为正确的格式,

[(]) 或 ([()) 或 (()])

均为不正确的格式。

可能出现的不匹配的情况:

1. 到来的右括弧不是所“期待”的;
2. 直到结束, 也没有到来所“期待”的。

[([] [])]

心法

- 1) 凡出现**左括弧**，则**进栈**；
- 2) 凡出现**右括弧**，首先检查栈是否空
若**栈空**，则表明该“**右括弧**”**多余**，
否则和栈顶元素**比较**，
若相**匹配**，则“**左括弧出栈**”，
否则表明**不匹配**。
- 3) 表达式检验结束时，
若**栈空**，则表明表达式中**匹配正确**，
否则表明“**左括弧**”**有余**。


```
status matching(string& exp) {  
    // 检验表达式中所含括弧是否正确嵌套, 若是, 则返回  
    // OK, 否则返回ERROR  
    int state = 1; i=1;  
    while (i<=length(exp) && state) {  
        switch exp[i] {  
            case "(": { Push(S,exp[i]); i++; break; }  
            case ")":  
                { if (NOT StackEmpty(S) && GetTop(S) = "(")  
                    { Pop(S,e); i++; }  
                  else { state = 0 }  
                break;  
            }  
            ..... }  
        }  
        if ( state && StackEmpty(S) ) return OK  
        else return ERROR;  
    }  
}
```

Application 3: Infix expression calculator

❖ 限于二元运算符的表达式定义:

表达式 ::= (操作数) + (算符) + (操作数)

操作数 ::= 常量 | 变量 | 常数

算符 ::= 运算符 (分为算术运算符、关系运算符和逻辑运算符)
| 界限符 (左右括号和表达式结束符等)

$\text{Exp} = S1 + OP + S2$

前缀表示法 $OP + \underline{S1} + \underline{S2}$

中缀表示法 $\underline{S1} + OP + \underline{S2}$

后缀表示法 $\underline{S1} + \underline{S2} + OP$

❖ 表达式表示方法

例如: $\text{Exp} = \underline{a * b} + \underline{(c - d / e) * f}$

前缀式: $+ \underline{* a b} \underline{* - c / d e f}$

中缀式: $\underline{a * b} + \underline{c - d / e * f}$

后缀式: $\underline{a b *} \underline{c d e / - f *} +$

● 中缀表达式求值的算法

- 采用“算符优先法”，在表达式中，优先级的顺序是：
 - 括号的优先级最高，对括号内的各种运算符有：先乘除、再加减，同级运算从左至右。
 - 先括号内，后括号外，多层括号，由内向外。
- 每一个运算步，相邻的算符 c_1 和 c_2 之间的关系是如下三种情况（ c_1 出现在 c_2 之前）：
 - $c_1 < c_2$, c_1 的优先级低于 c_2
 - $c_1 = c_2$, c_1 的优先级等于 c_2
 - $c_1 > c_2$, c_1 的优先级大于 c_2

算符间优先级

$c_1 \backslash c_2$	+	-	*	/	()	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

说明:

1. 计算表达式 $3-(6/2)\times 4$ ，‘#’是表达式的结束符。为了算法简洁，在表达式的最左边也虚设一个‘#’构成整个表达式的一对括号。表中的‘(’=‘)’表示当左右括号相遇时，括号内的运算已经完成。同理，‘#’=‘#’表示整个表达式求值完毕。

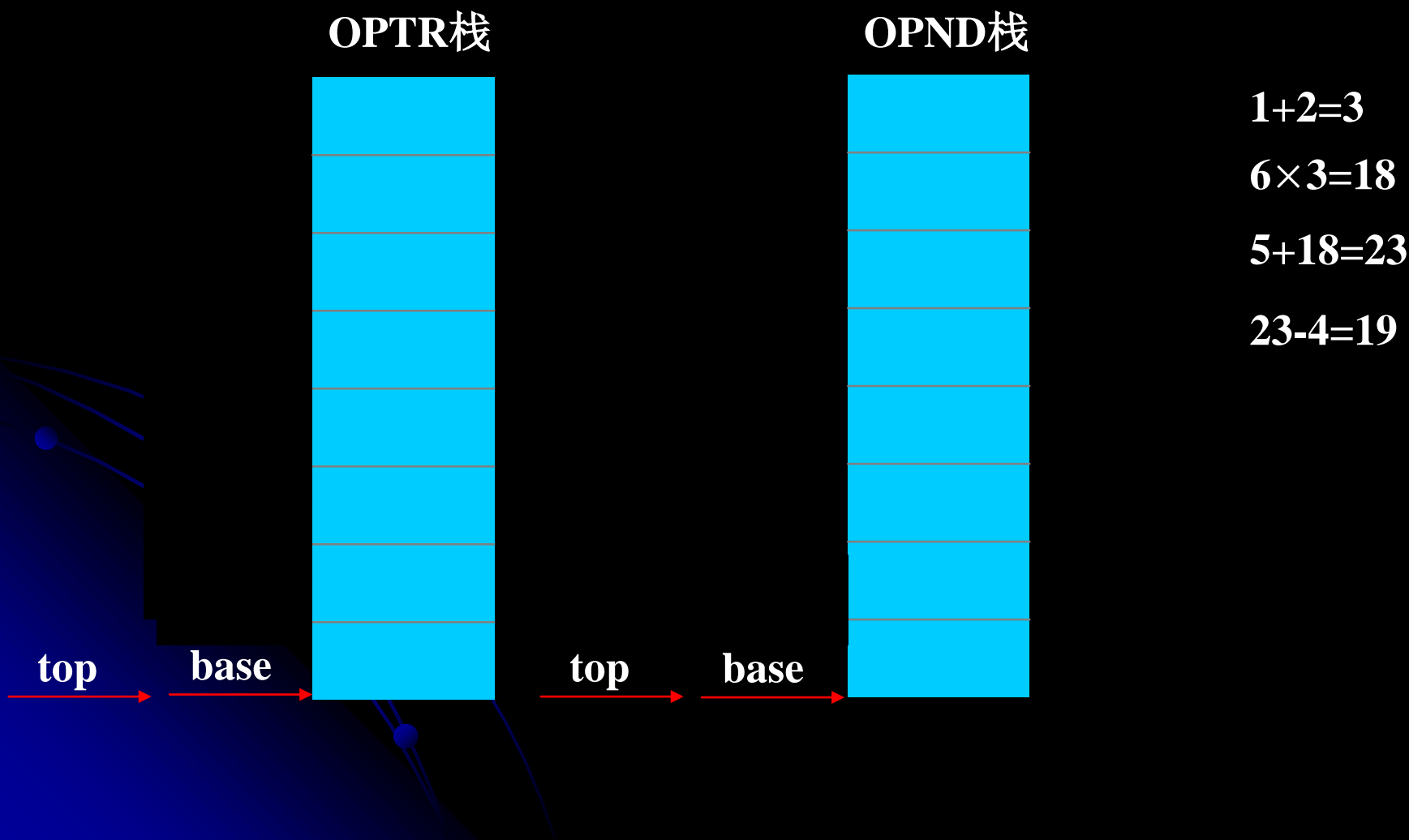
2. ‘)’与‘(’、‘#’与‘)’以及‘(’与‘#’之间无优先关系，这是因为表达式中不允许它们相继出现，一旦遇到这种情况，则可以认为出现了语法错误。在下面的讨论中，我们暂假定所输入的表达式不会出现语法错误。

算法核心步骤

- 为实现算符优先算法，在这里用了两个工作栈。一个存放算符OPTR，另一个存放数据OPND。算法思想是：
 - 首先置数据栈为空栈，表达式起始符“#”为算符栈的栈底元素
 - 自左向右扫描表达式，读到操作数进OPND栈，读到运算符，则和OPTR栈顶元素比较（栈顶元素为 c_1 ，读到的算符为 c_2 ）
 - 若 $c_1 < c_2$ ，则 c_2 进栈继续扫描后面表达式；
 - 若 $c_1 = c_2$ ，则（“=”），即括号内运算结束，将 c_1 出栈，并且 c_2 放弃，继续扫描后面表达式；
 - 若 $c_1 > c_2$ ，则将 c_1 出栈，并在操作数栈取出两个元素 a 和 b 按 c_1 做运算，运算结果进OPND。
 - 重复直到表达式求值完毕。

表达式求值示意图: $5+6\times(1+2)-4$

读入表达式过程: $5+6\times(1+2)-4\# = 19$



算法：求中缀表达式值

```
OperandType EvaluateExpression() {  
    InitStack(OPTR); Push(OPTR, '#'); InitStack(OPND); c=getchar();  
    while (c!= '#' || GetTop(OPTR)!= '#') {  
        if (!In (c, OP)) {  
            Push((OPND, c); c=getchar(); }  
        else  
            switch (Precede(GetTop(OPTR), c)) {  
                case '<': //栈顶元素优先权低  
                    Push(OPTR, c); c=getchar(); break;  
                case '=': //脱括号并接收下一字符  
                    Pop(OPTR, x); c=getchar(); break;  
                case '>': //退栈并将运算结果入栈  
                    Pop(OPTR, theta); Pop(OPND, b); Pop(OPND, a);  
                    Push(OPND, Operate(a, theta, b)); break;  
            }  
    }  
    return GetTop(OPND);  
}
```



Application 4: Reverse Polish calculator

- Expression
 - Infix: $a+b*(c-d)-e/f$
 - Postfix (Reverse Polish Notation): $abcd-*+ef/-$
 - Prefix (Polish Notation): $-+a*b-cd/ef$
- Operands: a, b, c, d, e, f
- Operators: $+, -, *, /$
- Delimiter: $(,)$

Expression Evaluation

- How to calculate the value of the expression?

$a + b * (c - d) - e / f$

Infix expression

$abcd - * + ef / -$

Postfix expression

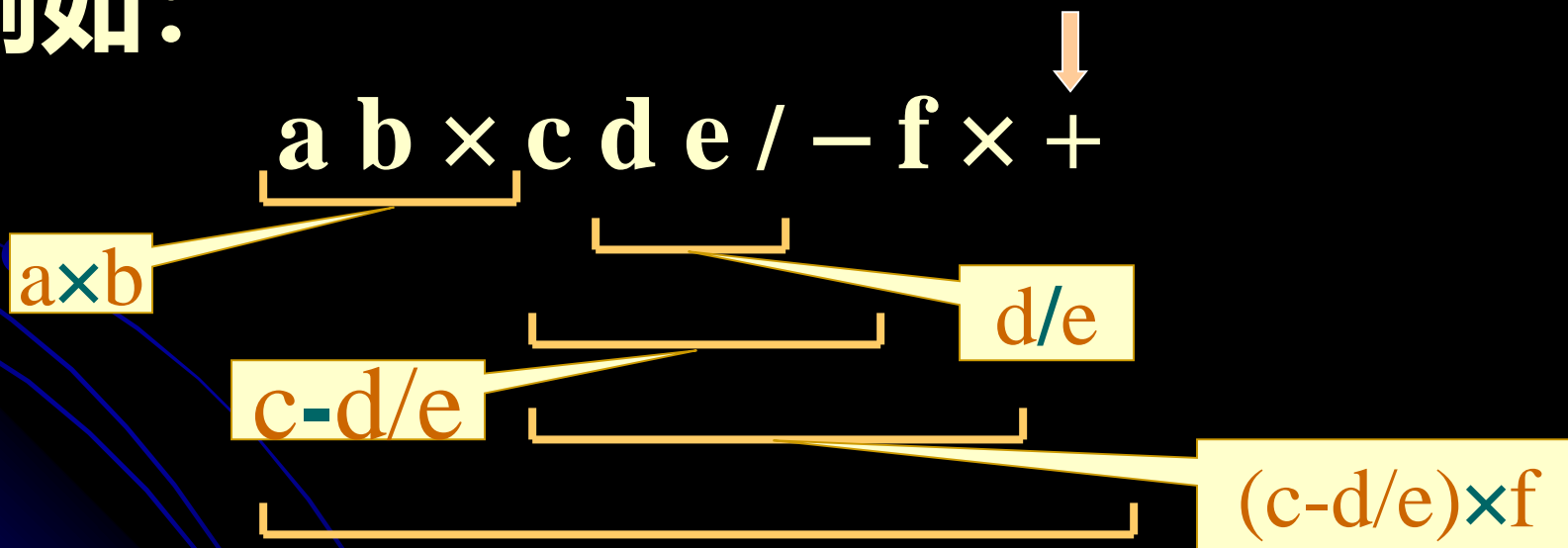
Prefix expression

$- + a * b - cd / ef$

❖ 后缀表达式求值


- 先找运算符，再找操作数

例如：



- 利用后缀表达式求值时，从左向右顺序地扫描表达式，并使用一个**数据栈**暂存扫描到的操作数或计算结果，例如，

abcd-*+ef/-



The diagram illustrates the evaluation of the postfix expression "abcd-*+ef/-". It features three yellow curved lines representing a stack. The first line connects 'a' and 'b' to the '-' operator. The second line connects the result of 'a-b' and 'c' to the '*' operator. The third line connects the result of '(a-b)*c' and 'd' to the '+' operator. This visualizes the step-by-step calculation where operands are pushed onto the stack and operators are applied to the top elements.

算法：求后缀表达式值

Elemtype EvaluateExpression_postfix()

```
{
    initStack(OPND); c=getchar( );
    while(c != '#' ) {
        if (!In(c, OP)) { Push((OPND, c); c=getchar( ); }
        else { //退栈并将运算结果入栈
            Pop(OPND, b) ; Pop(OPND, a);
            Push(OPND, Operate(a, theta, b));
            c=getchar( );
        }
    }
    return GetTop(OPND);
}
```