第三章　栈和队列

1. 栈的定义和栈的存储结构
2. 栈的应用
3. 栈与递归
4. 队列的定义和链队列
5. 循环队列

3.1 栈的定义和存储结构

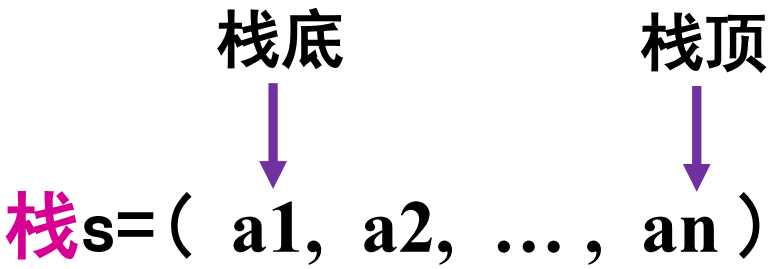
1 栈的定义

栈(Stack)：限定仅只能在表尾进行插入和删除的线性表。

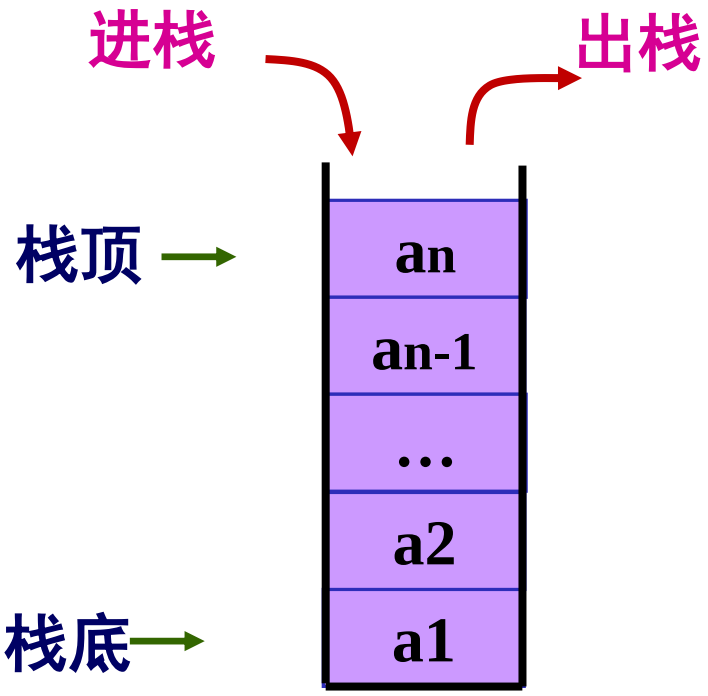
栈顶(Top)：表尾。

栈底(Bottom)：表头。

空栈：不含元素的空表。



栈s=（ a1，a2，…，an）



栈中元素按a_1，a_2，…a_n的次序进栈，退栈的第一个元素应为栈顶元素。栈的修改是按后进先出的原则进行的。因此，栈又称为后进先出(Last In First Out)的线性表。

栈的抽象数据类型定义：

ADT Stack{

数据对象：$D=\{a_i \mid a_i \in ElemSet, i=1,2,\cdots,n, n\geq 0\}$

数据关系：$R1=\{\langle a_{i-1}, a_i\rangle \mid a_{i-1}, a_i \in D, i=2,\cdots,n\}$

约定a_n端为栈顶，a_1端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈S。

DestroyStack(&S)

初始条件：栈S已存在。

操作结果：栈S被销毁。

ClearStack(&S)

初始条件：栈S已存在。

操作结果：将S清为空栈。

StackEmpty(S)

初始条件：栈S已存在。

操作结果：若栈S为空栈，则返回TRUE，否则FALE。

StackLength(S)

初始条件：栈S已存在。

操作结果：返回S的元素个数，即栈的长度。

GetTop(S, &e)

初始条件：栈S已存在且非空。

操作结果：用e返回S的栈顶元素。

Push(&S, e)

初始条件：栈S已存在。

操作结果：插入元素e为新的栈顶元素。

Pop(&S, &e)

初始条件：栈S已存在且非空。

操作结果：删除S的栈顶元素，并用e返回其值。

} ADT Stack

2 栈的存储结构

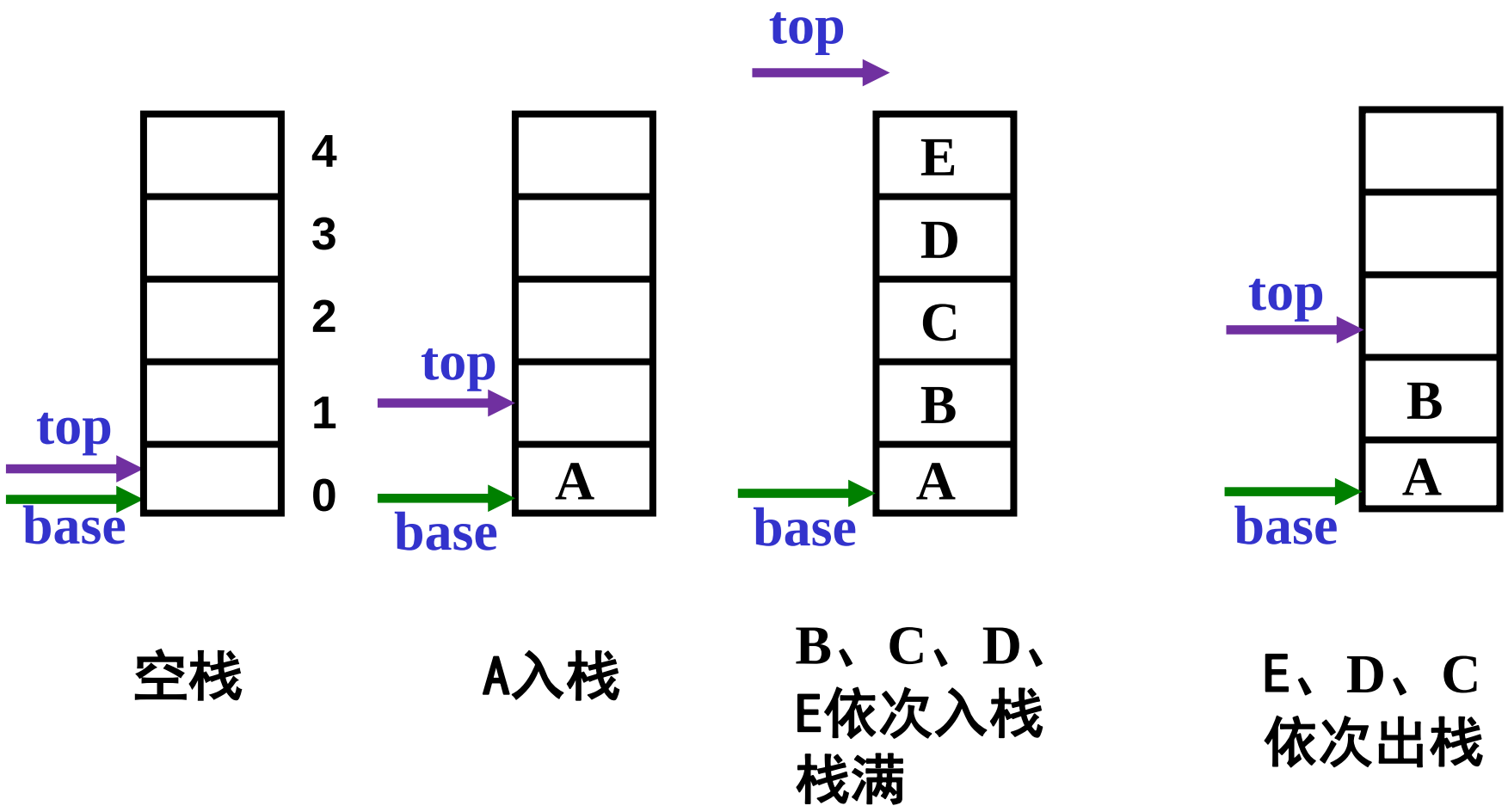
1. **顺序栈**:

利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针top指示栈顶元素在顺序栈中的位置。

2. **链栈**:利用链表实现。

顺序栈的类型声明：

```
#define INIT_SIZE 100          // 存储空间初始分配量
#define INCREMENT 10      //存储空间分配增量
typedef struct{
      SElemType  *base;       //栈底指针
      SElemType  *top;        //栈顶指针
      int StackSize;          //栈的当前已分配的存储空间
}SqStack;
```



空栈

A入栈

B、C、D、E依次入栈
栈满

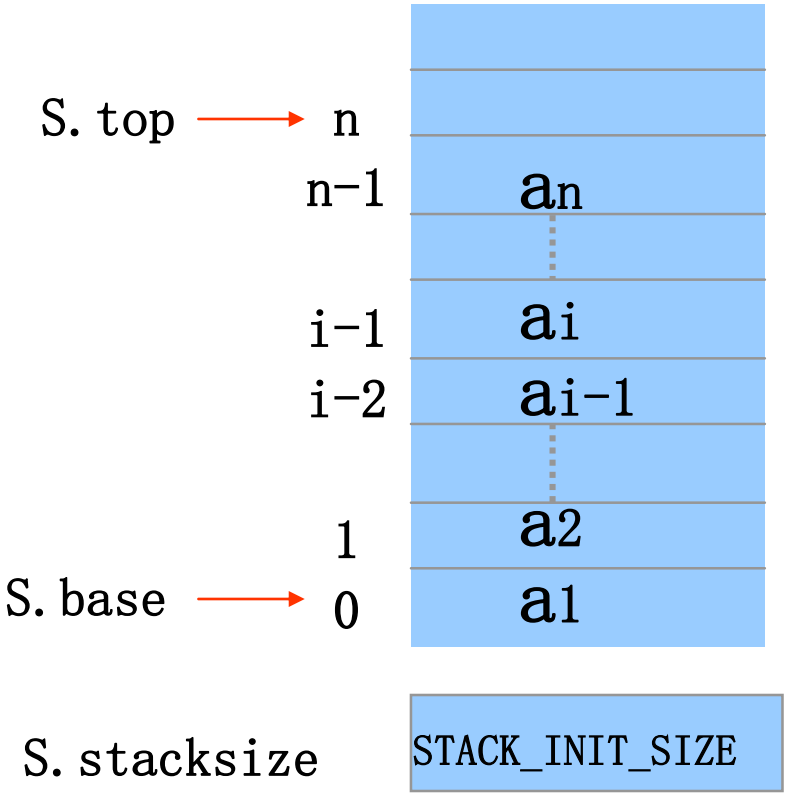
E、D、C
依次出栈

top== base 是栈空标志

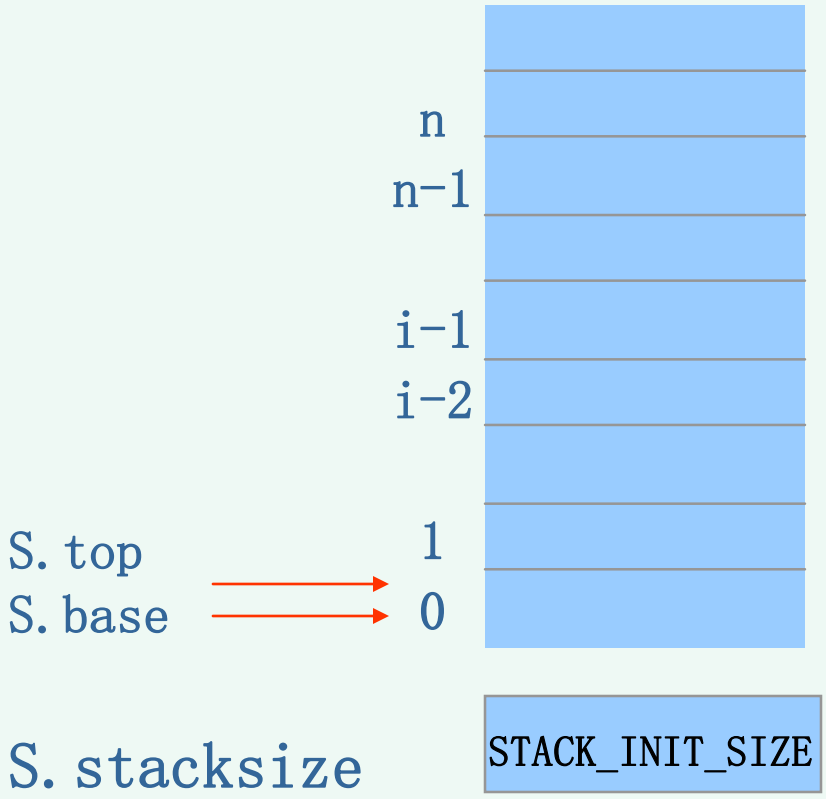
stacksize = 5

插入新的栈顶元素时，指针top增1；删除栈顶元素时，指针top减1，因此，非空栈中的栈顶指针始终在栈顶元素的下一个位置上。

顺序栈的图示


S.top
n
n-1
an
i-1
ai
i-2
ai-1
1
a2
S.base
0
a1
S.stacksize
STACK_INIT_SIZE

初始化操作图示

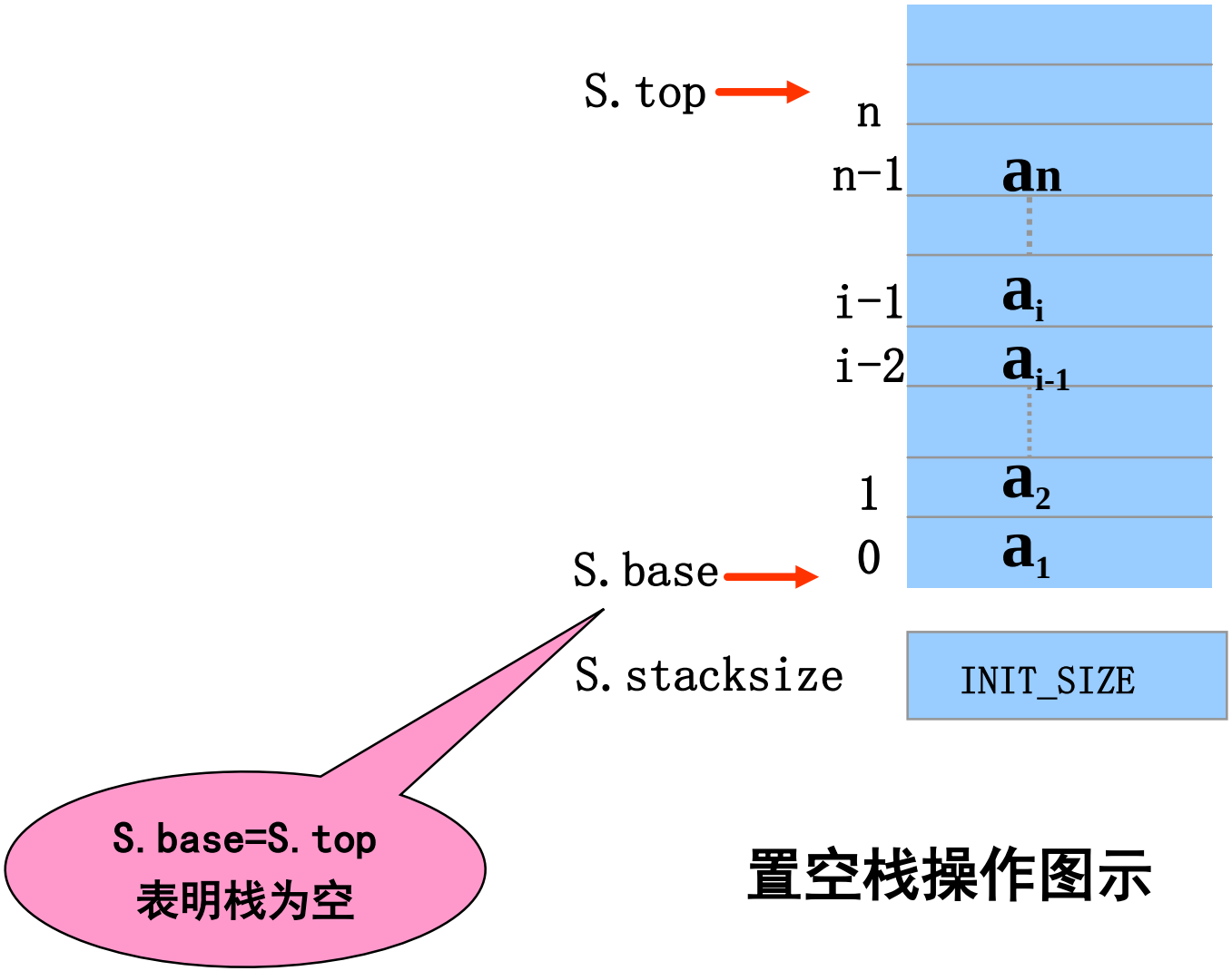


顺序栈基本操作的实现

顺序栈的初始化 InitStack (&S)

```
Status InitStack (SqStack &S)
{   //构造一个空栈S
     S.base=(SElemType *)malloc(INIT_SIZE *sizeof(SElemType));
     if(!S.base) exit(OVERFLOW);    //存储分配失败
     S.top=S.base;         //top初始为base
     S.stacksize=INI_SIZE;
    return OK;
}
```

顺序栈的置空 ClearStack(&S)



置空栈操作图示

```
Status ClearStack(SqStack &S)
{  //将S清为空栈
    If (!S.base)  return ERROR;
    S.top=S.base;
}
```

顺序栈的判空 StackEmpty (S)

```
Status StackEmpty(SqStack S)
{//若栈S为空栈，则返回TRUE，否则FALE。
    If (!S.base) return ERROR;
    If  (S.top==S.base)
        return TRUE;
     else
        return FALSE;
}
```

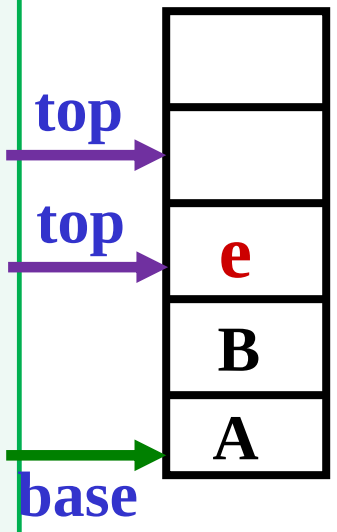
顺序栈的销毁DetroyStack (&S)

```
Status DetroyStack( SqStack &S)//销毁一个已存在的栈
{
      If (!S.base) return ERROR;
      free (S.base);                    // 回收栈空间
     S.base = S.top = NULL;
      S.stacksize = 0;
       return OK;
}
```

入栈　Push(&S, e)

在栈顶插入一个新的元素。

```
Status Push(SqStack &S, SElemType e)
{   //插入元素e为新的栈顶元素
    if(S.top - S.base>=S.stacksize) { //栈满，追加存储空间
        newbase=(SElemType *) realloc(S.base,
            (S.stacksize + INCREMENT) * sizeof(SElemType));
        if(!newbase)  exit(OVERFLOW);  //存储分配失败
        S.base=newbase;
        S.top=S.base+S.stacksize;
        S.stacksize+=INCREMENT;
    }
  *S.top++=e;      //元素e压入栈顶，栈顶指针加1
   return OK;
}
```

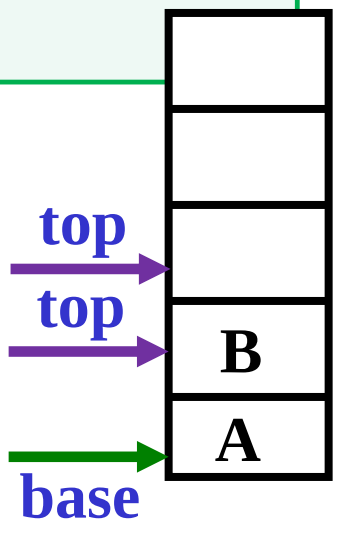


出栈　Pop(&S, &e)

将栈顶元素删除。

```
Status Pop(SqStack &S, SElemType &e)
{  //删除S的栈顶元素，用e返回其值
     if (S.top==S.base )  return ERROR;
      e=*--S.top;  //栈顶指针减1，将栈顶元素赋值给e
    return OK;
}
```

e　B

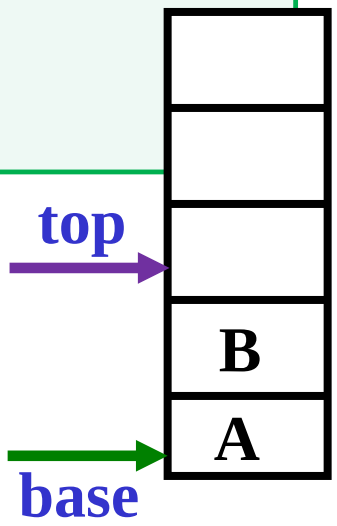


取栈顶元素　GetTop(S, &e)

用e返回S的栈顶元素，栈顶指针保持不变。

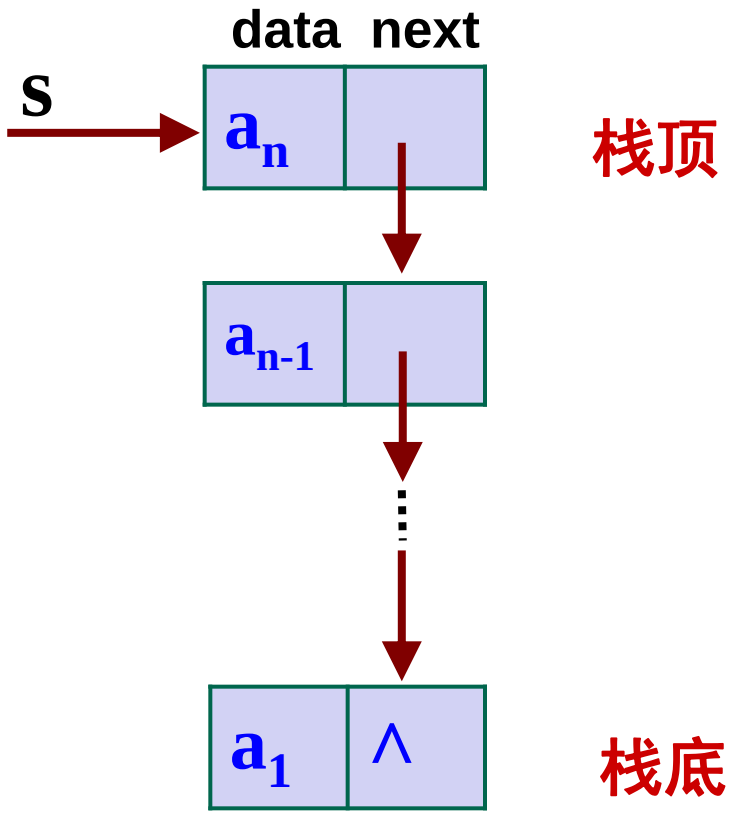
```
Status GetTop(SqStack S, SElemType &e)
{   //用e返回S的栈顶元素

     if(S.top==S.base) return ERROR;

     e=*(S.top-1);

   return OK;
}
```

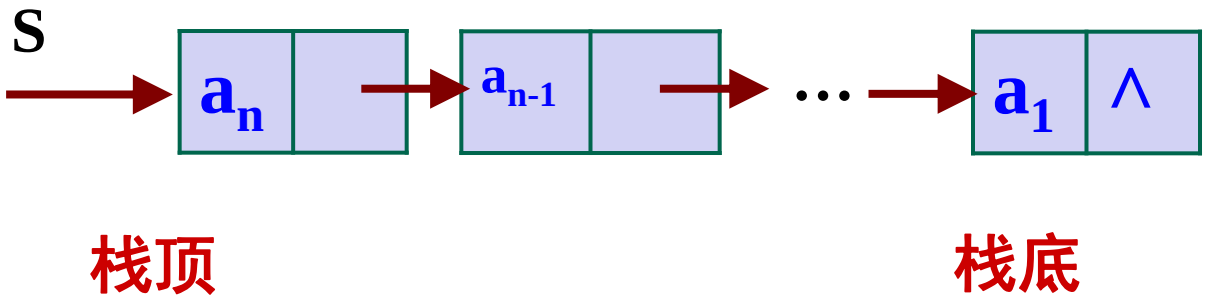
e B



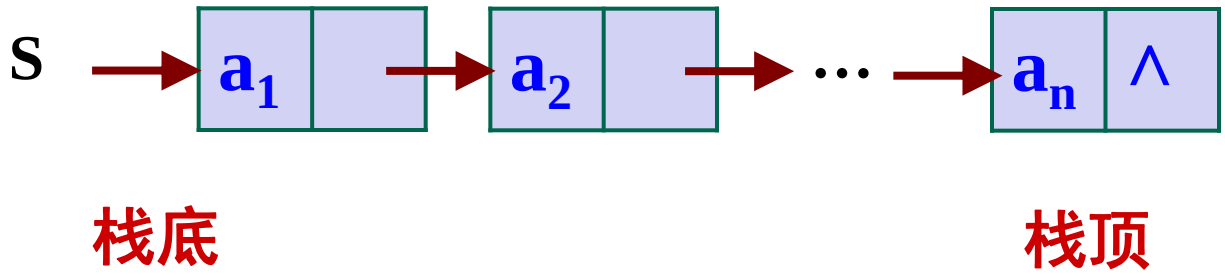
栈的链式存储结构

链栈：栈的链式存储结构，通常用单链表来表示。它是运算是受限的单链表，即插入和删除操作仅限制在链表头位置上进行。栈顶指针就是链表的头指针。





如果插入和删除操作仅限制在链表尾进行呢？



链栈中结点类型的声明:

```
typedef   struct  StackNode {
        SElemType              data;        // 数据域
        struct   StackNode     *next; // 指针域
}SNode,  *LinkStack;
```

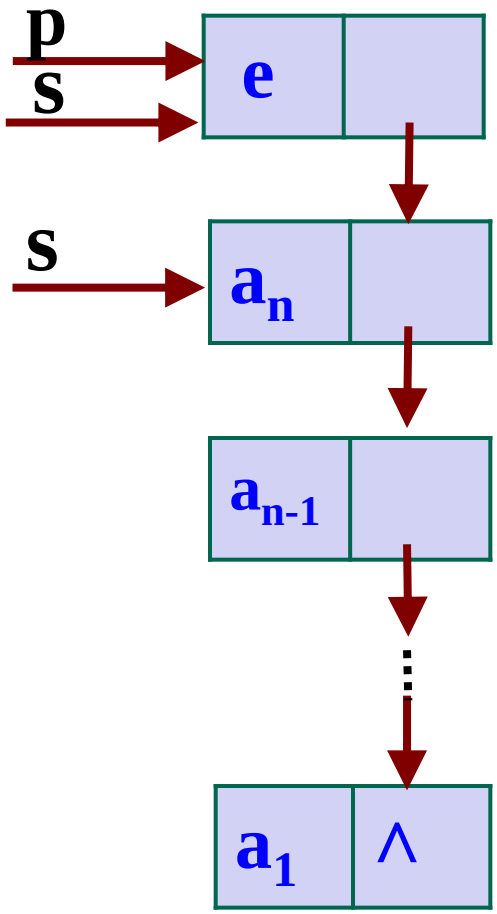
链栈基本操作的实现

1. 链栈的初始化 InitStack (&S)

```
void InitStack(LinkStack &S)
{  //构造一个空栈S，栈顶指针置空
        S=NULL;

}
```

2. 入栈　Push(&S, e)

在栈顶插入一个新的元素。



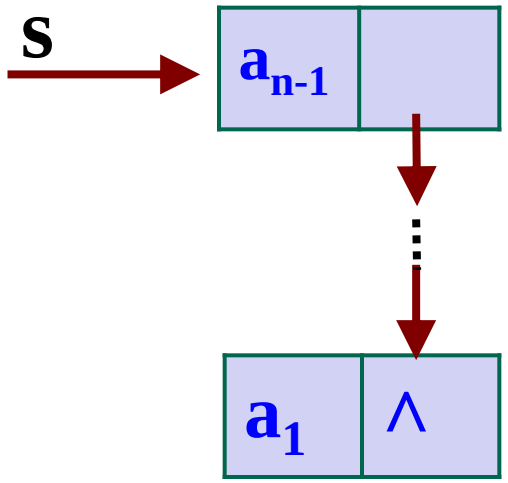

```
void Push(LinkStack &S, SElemType e)
{   //插入元素e为新的栈顶元素
   p=(LinkStack)malloc(sizeof(SNode));
   //生成x新结点
   p→data=e;
   p→next=S;     //将新结点插入栈顶

   S=p;                  //修改栈顶指针为p
}
```

链栈的入栈过程

3. 出栈　Pop(&S, &e)

将栈顶元素删除。

```
Status Pop(LinkStack &S, SElemType &e)
{  //删除S的栈顶元素，用e返回其值
    if (S ==NULL )  return ERROR; //栈空
    e=S->data;
    p=S;     //用p临时保存栈顶元素空间，以备释放
    S=S->next;    //修改栈顶指针
    free(p);          //释放原栈顶元素的空间

   return OK;
}
```



链栈的出栈过程

3.2 栈的应用

由于栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。

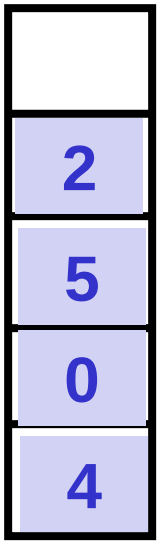
1 . 数制转换

十进制数N  其它R进制数

除R取余数法：

例如 $(1348)_{10}=(2504)_8$，其运算过程如下：

		余数
8	1348	
8	168	4
8	21	0
8	2	5
	0	2



2 5 0 4

```
void conversion (int N)
{// 对于一个非负十进制整数，打印输出与其等值的八进制数
    InitStack(S);    // 构造空栈S
    while(N){
        Push(S,  N%8);   // 把N与8求余得到的数入栈
        N=N/8;      //N更新为N与8的商
     }
   while(! StackEmpty(S)){   //当栈S非空时，循环
        Pop(S, e);                  //弹出栈顶元素e
        printf(“%d”,e);         //输出e
    }
}
```

2. 括号匹配的检验

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即（[] ()）或[（[ ] [ ]）]等为正确的格式，[（]）或（[（ ））或（ （ ）]）均为不正确的格式。检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。例如考虑下列括号序列：

[	(	[	]	[	]	)	]
1	2	3	4	5	6	7	8

```
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
  InitStack(S);
   int state = 1;  //state表明状态，state=1,正确；state=0,已出错。
  while (i<=length(exp) && state) {
      switch  exp[i] {
         case 左括弧: { Push(S,exp[i]); i++; break; }
         case ")":
             { if (! StackEmpty(S) && GetTop(S) = "(")
                   { Pop(S,e); i++; }
             else { state = 0; }
             break;
              }
          …… }
   }
   if ( state && StackEmpty(S) ) return OK
  else return ERROR;
}
```

3 行编辑程序

一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。

“每接受一个字符即存入用户数据区”不恰当。

较好的做法:设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入出差错，并在发现有误时可以及时更正。

例如，可用一个退格符“#”表示前一个字符无效；可用一个退行符“@”，表示当前行中的字符均无效。

例如，

假设从终端接受了这样两行字符：　则实际有效的是下列两行：

whli##ilr#e（s#*s)　　**while (*s)**

outcha@putchar(*s=#++);　　**putchar(*s++);**

可设这个输入缓冲区为一个栈结构，每当从终端接受一个字符后先作如下判别：

a.如果它既不是退格符也不是退行符，则将它进栈；

b.如果是退格符，则从栈顶删去一个字符；

c.如果是退行符，则将栈清空。

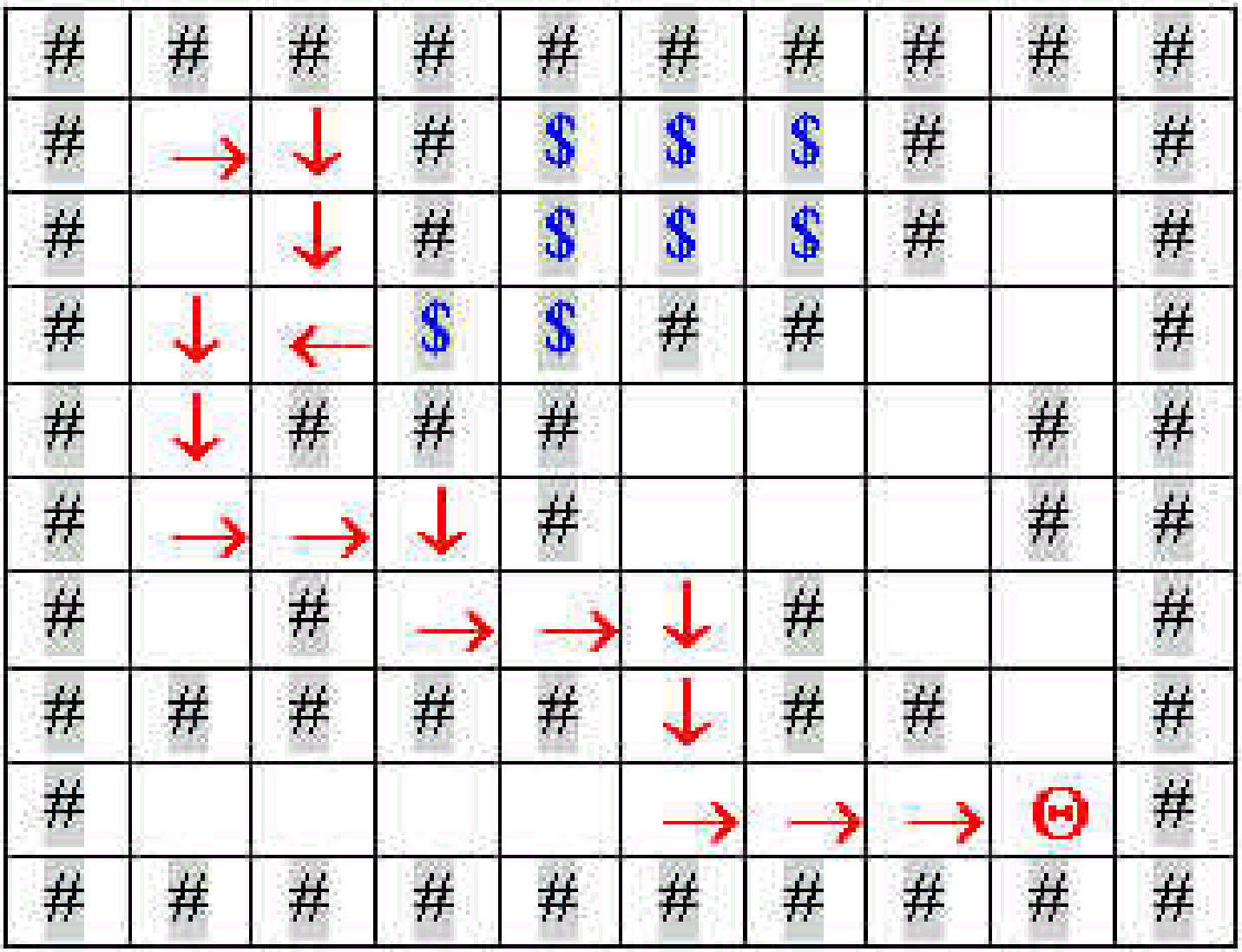
```
void LineEdit() {
// 利用字符栈S，从终端接收一行并传送至调用过程 的数据区。
  InitStack(S); //构造空栈S
  ch = getchar();     //从终端接收第一个字符
  while (ch != EOF) { //EOF为全文结束符
    while (ch != EOF && ch != '\n') {
      switch (ch) {
          case '#' : Pop(S, c); break; // 仅当栈非空时退栈
          case '@': ClearStack(S); break; // 重置S为空栈
          default : Push(S, ch); break;
                        // 有效字符进栈，未考虑栈满情形
       }
       ch = getchar(); // 从终端接收下一个字符
   }
    将从栈底到栈顶的字符传送至调用过程的数据区；
    ClearStack(S); // 重置S为空栈
    if (ch != EOF) ch = getchar();
  }
   DestroyStack(S);
 }
```

4. 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。由于计算机解迷宫时，通常用的是“穷举求解”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。

为了保证在任何位置上都能沿原路退回，显然需要用一个后进先出的结构来保存从入口到当前位置的路径。因此，在求迷宫通路的算法中应用“栈”也就是自然而然的事了。

假设迷宫如下图所示：



假设“当前位置”指的是“在搜索过程中某一时刻所在图中某 个方块位置”，

则求迷宫中一条路径的算法的基本思想是：

若当前位置“可通”，则纳入“当前路径”，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；

若当前位置“不可通”，则应顺着“来向”退回到“前一通道块”，然后朝着除“来向”之外的其他方向继续探索；

若该通道块的四周四个方块均“不可通”，则应从“当前路径”上删除该通道块。

所谓“下一位置”指的是“当前位置”四周四个方向（东、南、西、北）上相邻的方块。假设以栈S记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道块”。由此，“纳入路径”的操作即为“当前位置入栈”；“从当前路径上删除前一通道块”的操作即为“出栈”。

求迷宫中一条从入口到出口的路径的算法可简单描述如下：

```
设定当前位置的初值为入口位置；
do {
    若当前位置可通，
    则 {  将当前位置插入栈顶；              // 纳入路径
          若该位置是出口位置，则结束；     // 求得路径存放在栈中
          否则切换当前位置的东邻方块为新的当前位置；
       }
    否则 {
        若栈不空，且栈顶位置尚有其他方向未被探索，
              则设定新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的
               下一相邻块；
        若栈不空，但栈顶位置的四周均不可通，
              则 { 删去栈顶位置；            // 从路径中删去该通道块
                   若栈不空，则重新测试新的栈顶位置，
                       直至找到一个可通的相邻块或出栈至栈空；
                  }
          }
    } while (栈不空）；
```

```
typedef struct {
    int  ord;       // 通道块在路径上的“序号”
    PosType seat;   // 通道块在迷宫中的“坐标位置”
    int di;   // 从此通道块走向下一通道块的“方向”
} SElemType;   // 栈的元素类型
```

5. 表达式求值

表达式求值是编译系统中最基本的一个问题，其实现是栈应用的一个典型例子。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能够正确解释表达式。算符优先算法就是根据算术四则运算法则确定的运算优先关系，实现对表达式的编译或解释执行的。

表达式的构成：操作数+运算符+界限符（如括号）

操作数：常数、变量或常量标识符

运算符：/ * + -

界限符：( ) #

运算符和界限符统称算符

算符优先关系表：

表达式中任何相邻算符θ1、θ2的优先关系有：

θ1<θ2：θ1的优先级低于θ2

θ1=θ2：θ1的优先级等于θ2

θ1>θ2：θ1的优先级高于θ2

由四则运算法则，可得到如下的算符优先关系表：

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

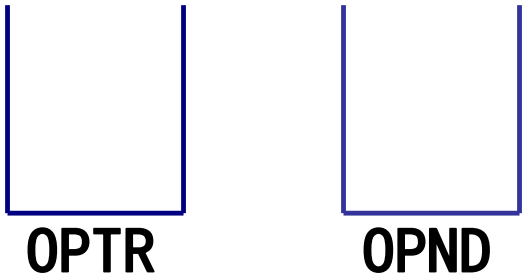
注： θ1、θ2是相邻算符， θ1在左， θ2在右

算符间的优先关系表

算符优先算法:

在算符优先算法中，建立了两个工作栈。

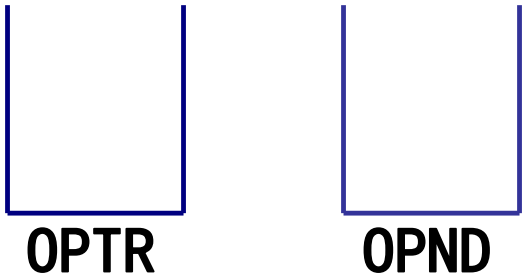
- OPTR栈　　用以保存运算符；
- OPND栈　　用以保存操作数或运算结果。



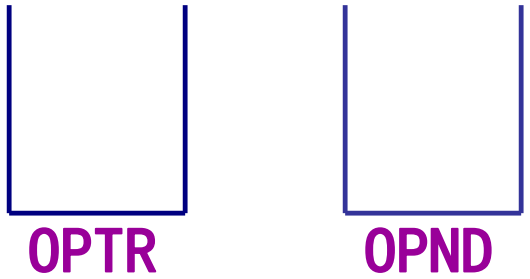
算法基本思想：

1 首先置操作数栈为空栈，表达式起始符＃为运算符栈的栈底元素；

2 依次读入表达式中每个字符，若是操作数则进OPND栈，若是运算符，则和OPTR栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕。



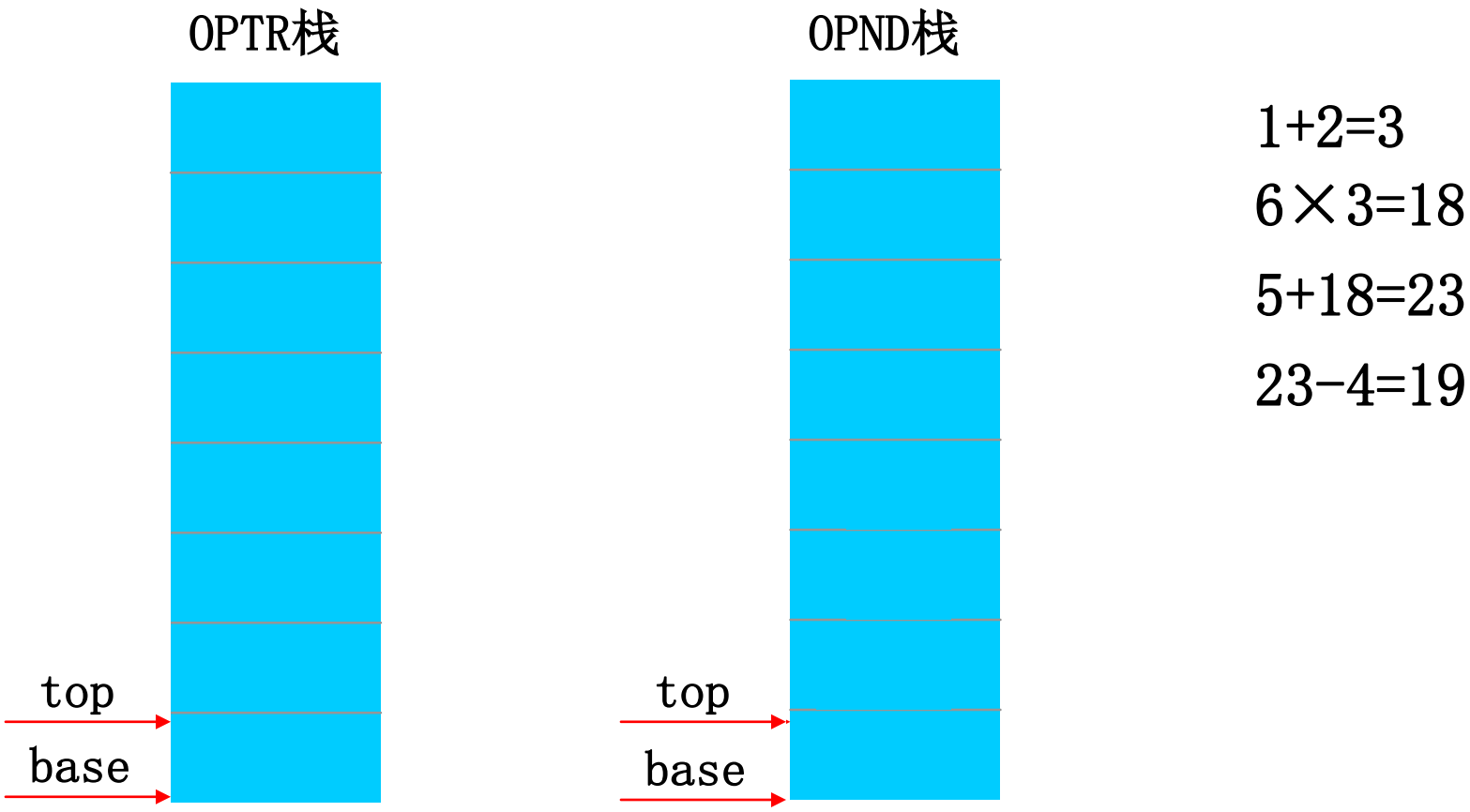
```
OperandType EvaluateExpression( )
{//算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和运算数栈
    InitStack(OPTR);     //初始化OPTR栈
    Push (OPTR, #);       //将表达式起始符“#”压入OPTR 栈
    InitStack(OPND);
    c=getchar( );
```



```
while(c!=‘ #’ || GetTop(OPTR)!=‘#’)
 {   //表达式没有扫描完毕或OPTR的栈顶元素不是“#”
   if (! In (c, OP)) {      // In(c, OP)判断c是否是运算符的函数
        Push(OPND,  c); c=getchar( );} //c不是运算符则进OPND栈
    else
        switch (Precede(GetTop(OPTR),  c)  { //比较优先级
              case ‘<‘:      // 新输入的算符c优先级高，c进栈
                   Push(OPTR,  c);   c=getchar( );   break;
              case ‘=‘:      // 脱括号并接收下一字符
                   Pop(OPTR, x);  c=getchar( ); break;
              case ‘>’:     //新输入的算符c优先级低，即栈顶算符优先权高
                Pop(OPTR,  theta);
                Pop(OPND, b);  Pop(OPND, a); //弹出OPND栈的两个运算数
                Push(OPND, Operate(a, theta, b)); //将运算结果入栈OPND
                break;
              }
 }
   return GetTop(OPND);   //OPND栈顶元素即为表达式求值结果
}
```

表达式求值示意图：5+6×(1+2)-4 =19

读入表达式过程：　　5+6×(1+2)-4#



1+2=3

6×3=18

5+18=23

23-4=19