数据结构

深圳技术大学
大数据与互联网学院

第三章 栈和队列

3.1 栈
3.2 栈的应用举例
3.3 栈与递归的实现
3.4 队列

3.1 栈

一. 栈的概念

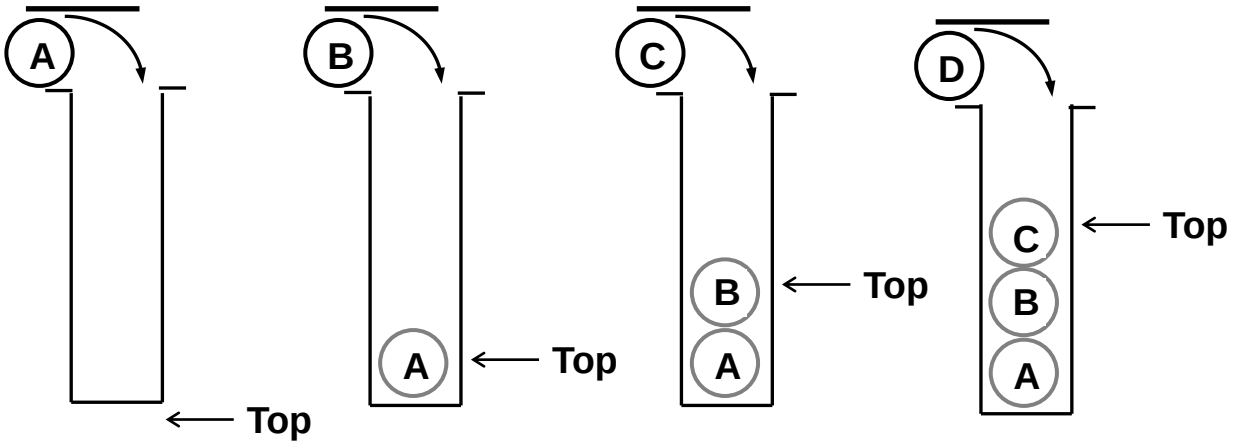
- 栈是限定仅在表尾(top)进行插入或删除操作的线性表
- 允许插入和删除的一端称为栈顶(top，表尾)，另一端称为栈底(bottom，表头)
- 特点：后进先出 (LIFO)

3.1 栈

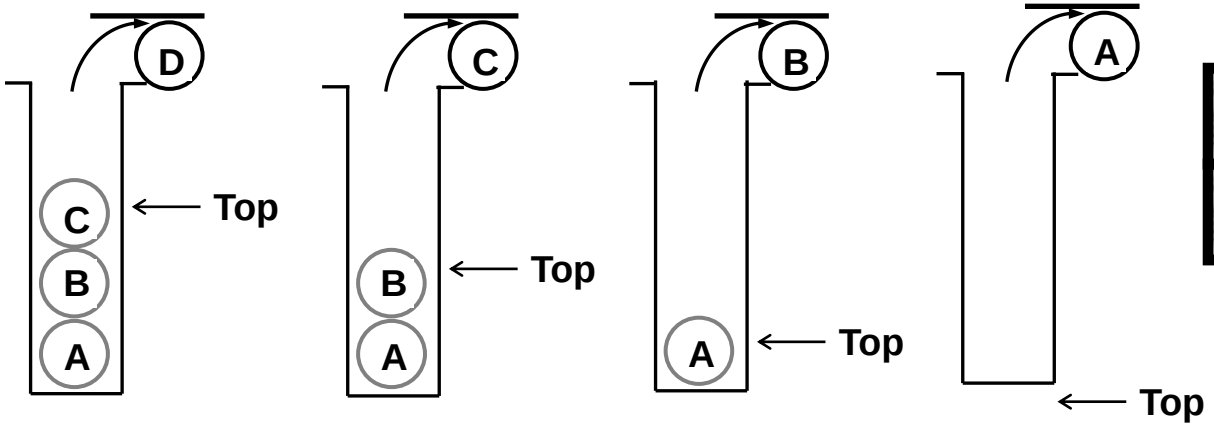
一. 栈的概念

- 栈的ADT，P45

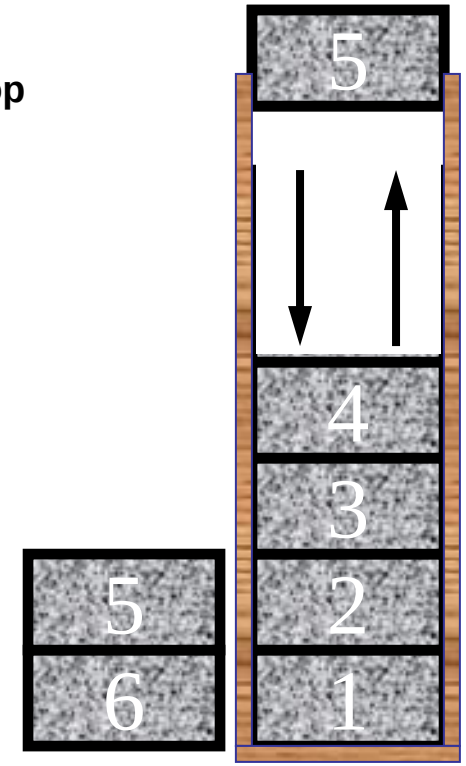
```
ADT Stack {
    数据对象：D = {a_i | a_i∈ElemSet, i=1,2,3,...,n}
    数据关系：R = {<a_i-1, a_i> | a_i-1,a_i∈D}
      基本操作：InitStack(&S)          // 构造空栈
                Push(&S, e)            // 进栈
                Pop(&S, &e)            // 出栈
                GetTop(S, &e)          // 取栈顶元素值
                StackEmpty(S)          // 栈是否为空
    } ADT Stack
```



CreatStack( ); **Push(S,A);** **Push(S,B);** **Push(S,C);**



x=Pop(S); **x=Pop(S);** **x=Pop(S);** **IsEmpty (S)**



- **Push 和 Pop 可以任意穿插交替进行；**
- **求后缀表达式 5 6 2 / + 3 4 * - 时： Push 和 Pop的序列是：**

```
Push(S,5);
Push(S,6);
Push(S,2);   /* S: 5 6 2 */
x=Pop(S);   /* x=2 */
y=Pop(S);   /* y=6 */
Push(S,y/x); /* y/x=3, S: 5 3 */
x=Pop(S);   /* x=3 */
y=Pop(S);  /* y=5 */
Push(S,x+y); /* y+x=8 , S: 8 */
Push(S,3);
Push(S,4);  /* S: 8 3 4 */
x=Pop(S);  /* x=4 */
y=Pop(S);  /* y=3 */
Push(S,x*y); /* y*x=12 , S: 8  12 */
x=Pop(S);  /* x=12 */
y=Pop(S);  /* y=8 */
Push(S,y-x); /* S: -4 */
x=Pop(S);  /* x=-4 */
```

3.1 栈

二. 顺序栈

- 顺序栈是栈的一种实现，是顺序存储结构，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
- 指针top指向栈顶元素在顺序栈中的下一个位置，base为栈底指针，指向栈底的位置

3.1 栈

二. 顺序栈

- 顺序栈的定义

```
#define STACK_INIT_SIZE     100          // 栈存储空间的初始分配量
#define STACKINCREMENT    10             // 栈存储空间的分配增量
class SqStack{
    SElemType  *base                     // 栈底指针，也是栈的基址
    SElemType  *top;                     // 栈顶指针
    int   stacksize;          // 当前分配的存储容量(元素数)
};
```

- 顺序栈在存储结构上类似于顺序表，都是一维数组

3.1 栈

二. 顺序栈

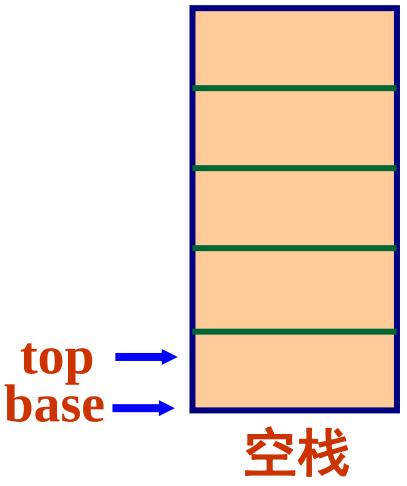
- 顺序栈的特性：
  - top=0 或top=base 表示空栈
  - base=NULL表示栈不存在
  - 当插入新的栈顶元素时，指针top+1
  - 删除栈顶元素时，指针top-1
  - 当top>stacksize时，栈满，溢出

3.1 栈

二. 顺序栈

- 顺序栈的创建

```
Status SqStack::InitStack(int size=STACK_INIT_SIZE) {
    stacksize = size;
    base = new SElemType[stack_size];
    if (! base) exit(OVERFLOW);      // 存储分配失败
    top = base;
    return OK;
} // InitStack
```

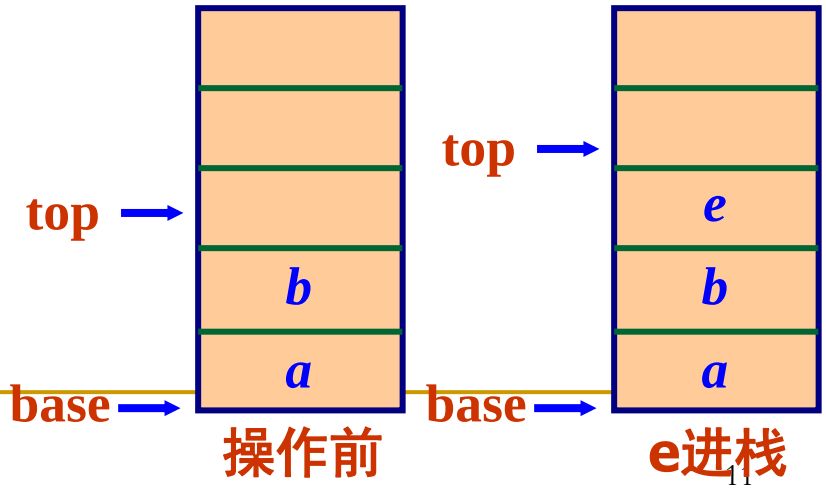


3.1 栈

二. 顺序栈

- 顺序栈的进栈（插入元素）

```
Status SqStack::Push(SElemType e) {
    if (top - base >= stacksize) {     // 栈满，追加存储空间
        SElemType* newbase = new SElemType[stacksize +
STACKINCREMENT];
        if (newbase) exit(OVERFLOW);
        for(int i=0; i<stacksize; i++) newbase[i] = base[i];
        base = newbase; top = base + stacksize;
        stacksize += STACKINCRMENT;
    }
    *(top++) = e;
    return OK;
} // Push
```

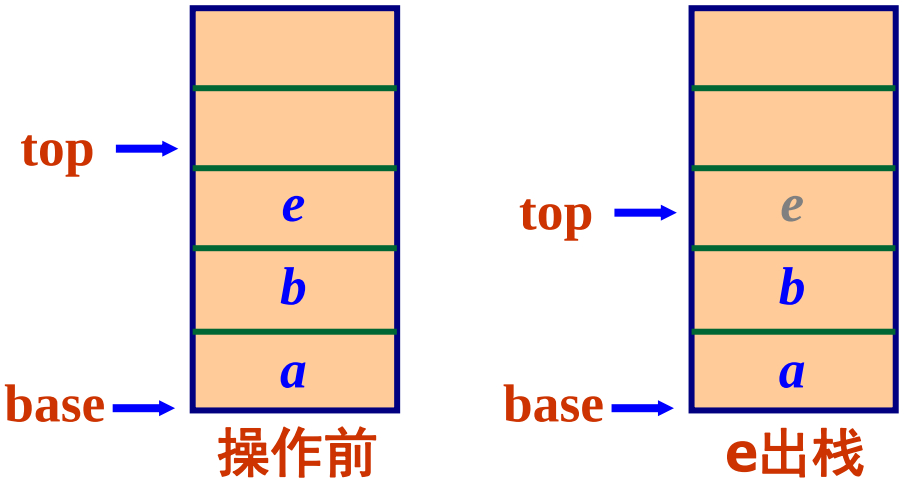


3.1 栈

二. 顺序栈

- 顺序栈的出栈（删除元素）

```
Status SqStack::Pop(SElemType &e) {
    if (top == base) return ERROR; // 空栈
    e = *(--top);
    return OK;
} // Pop
```



3.1 栈

二. 顺序栈

- 顺序栈的取栈元素

```
Status SqStack::GetPop(SElemType &e) {
    if (top == base) return ERROR; // 空栈
    e = *(top-1); // 只是取值不做出栈处理
    return OK;
} // GetPop
```

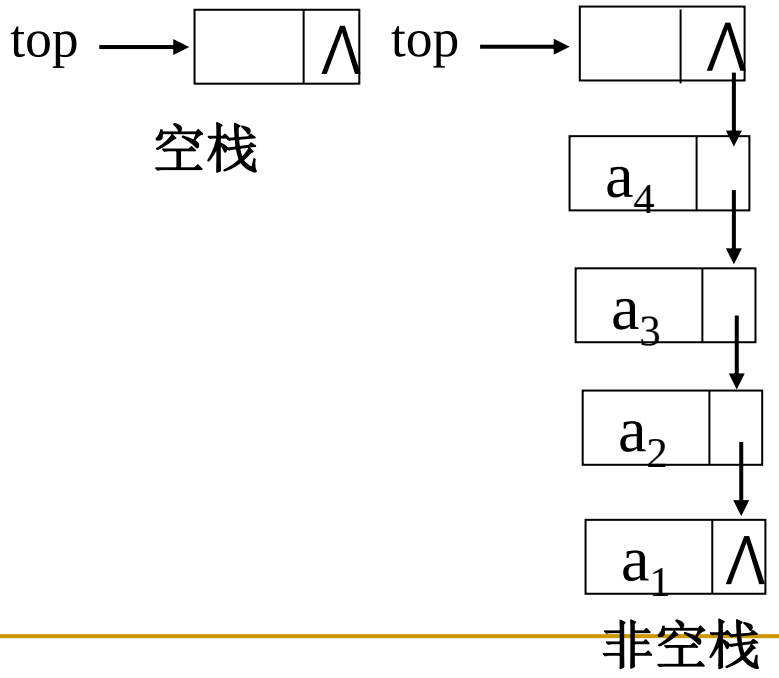
3.1 栈

三. 链栈

- 栈的链式存储结构称为链栈，是运算受限的单链表。其插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针top就是链表的头指针。

链栈的结点类型说明如下：

```
class  StackNode{
    ElemType data ;
    StackNode *next ;
};
class  LinkStack{
    StackNode *top;
    ... // 其他成员
};
```



3.1 栈

三. 链栈

- 初始化

```
void  LinkStack::Init(){
    top = new StackNode;
    top->next = NULL ;
} // 初始化空的链表栈
```

3.1 栈

三. 链栈

- 进栈

```
Status LinkStack::Push(ElemType e){
    StackNode *p = new StackNode;
    if (!p)  return ERROR; // 申请新结点失败，返回错误标志
    p->data=e ;
    p->next=top->next  ;
    top->next=p ;     //钩链
    return OK;
}
```

3.1 栈

三. 链栈

- 出栈

```
Status LinkStack::Pop(ElemType &e){
    if  (top->next==NULL )
        return ERROR ;    // 栈空，返回错误标志
    StackNode* p = top->next; e = p->data ; //取栈顶元素
    top->next = p->next ;      // 修改栈顶指针
    delete p ;
    return OK ;
}
```

3.2 栈的应用

一. 数制转换

- 将十进制转换为其它进制(d)，其原理为：

N = (N/d)*d + N mod d

例如：(1348)10 = (2504)8 ，其运算过程如下：

N	N /8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

3.2 栈的应用

一. 数制转换

- 实现函数（十进制 -> 八进制）

```
void conversion() {
    LinkStack S; S.Init();          // 创建新栈S
    int N; cin << N;                // 输入一个十进制数N
    while (N) {
        S.Push(N % 8);              // 将余数送入栈中
        N = N/8;                    // 求整除数
    }
    while (!S.Empty()) {            // 如果栈不空
        int e; S.Pop(e);            // 将栈中数出栈
        cout << e;
    }
} // conversion
```

3.2 栈的应用

二. 行编辑

- 用户输入一行字符
- 允许用户输入出差错，并在发现有误时，可以用退格符“#”及时更正
  - 假设从终端接受两行字符：

    whli##ilr#e（s#*s）
  - 实际有效行为： while（*s）

```
//对用户输入的一行字符进行处理，直到行结束("\n")
char ch = cin.get();          // 从终端输入一个字符
while (ch != '\n') {
       switch(ch) {
         case '#': S.Pop(c);       break; // 仅当栈非空时退栈
         default:  S.Push(ch);     break; // 有效字符进栈
       }
       ch = cin.get();                    // 从终端输入一个字符
}
//将从栈底到栈顶的栈内字符传送到调用过程的数据区;
```

3.2 栈的应用

三、括号匹配的校验

- 在处理表达式过程中需要对括号匹配进行检验，括号匹配包括三种：“(”和“)”，“[”和“]”，“{”和“}”。例如表达式中包含括号如下：

(	)	[	(	)	(	[	]	)	]	{	}
1	2	3	4	5	6	7	8	9	10	11	12

上例可以看出第1和第2个括号匹配，第3和第10个括号匹配，4和5匹配，6和9匹配，7和8匹配，11和12匹配

3.2 栈的应用

三、括号匹配的校验

- 求解算法
  - 初始化，i=0，建立堆栈，栈为空，输入表达式
  - 读取表达式第i个字符
  - 如果第i个字符是左括号，入栈
  - 如果第i个字符是右括号，检查栈顶元素是否匹配

    如果匹配，弹出栈顶元素

    如果不匹配，报错退出
  - i+1，是否已经表达式末尾

    未到末尾，重复步骤2

    已到达末尾
  - 堆栈为空，返回匹配正确，堆栈不为空，返回错误

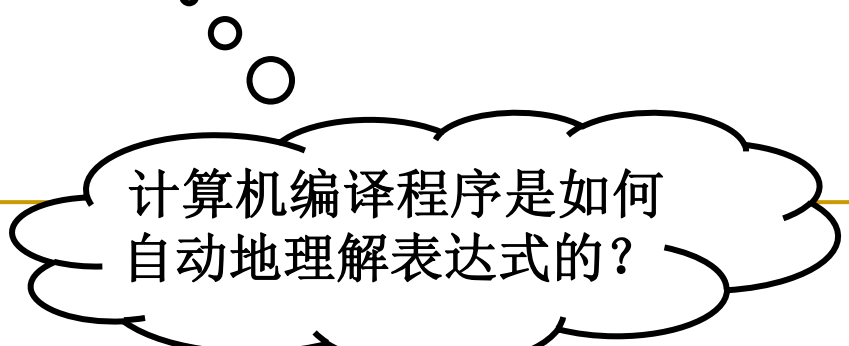
3.2 栈的应用

四、表达式求值

要实现表达式求值，首先需要正确理解一个表达式，主要是运算的先后顺序。

[例3.4] 对于算术表达式来说，其基本规则是：

- 先乘除，后加减；先括号内，再括号外；
- 相同优先级情况下从左到右。
- 比如，5+6/2-3*4就是一个算术表达式，它的正确理解应该是：
- 5+6/2-3*4 = 5+3-3*4 = 8-3*4 = 8-12 = -4。
- 可以看到这类表达式主要由两类对象构成的，即运算数（如2、3、4等）和运算符号（如+、-、*、/等）。
- 不同运算符号优先级是不一样的，而且运算符号均位于两个运算数中间。



3.2 栈的应用

四、表达式求值

❖ 后缀表达式

- 中缀表达式：运算符号位于两个运算数之间。如，$a+b*c-d/e$

  5+6/2-3*4

- 后缀表达式：运算符号位于两个运算数之后。如，$abc*+de/-$

  6 2 / 3 – 4 2 * +

3.2 栈的应用

四、表达式求值

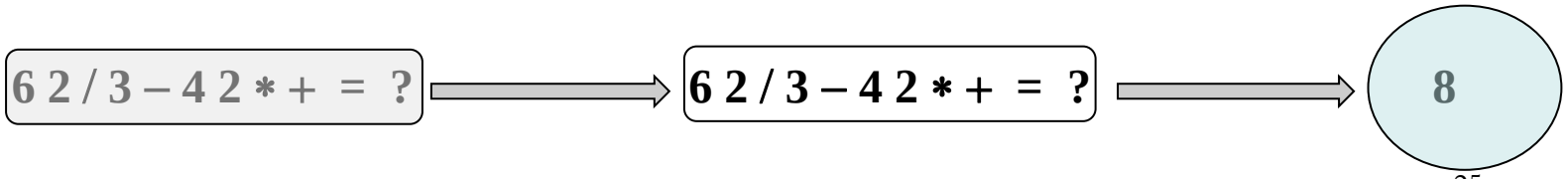
➢ 应用堆栈实现后缀表达式求值的基本过程：

. 从左到右读入后缀表达式的各项（运算符或运算数）；

. 根据读入的对象（运算符或运算数）判断执行操作；

. 操作分下列3种情况：

1. 当读入的是一个运算数时，把它被压入栈中；
2. 当读入的是一个运算符时，就从堆栈中弹出适当数量的运算数，对该运算进行计算，计算结果再压回到栈中；
3. 处理完整个后缀表达式之后，堆栈顶上的元素就是表达式的结果值。



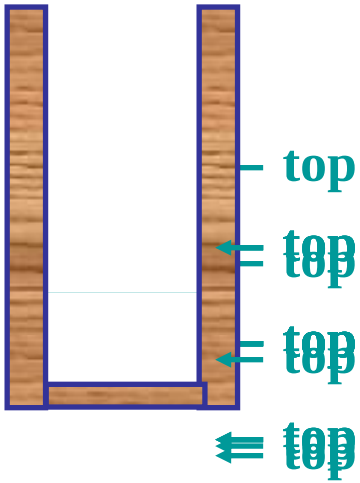
3.2 栈的应用

四、表达式求值

❖ 后缀表达式

- 中缀表达式：运算符号位于两个运算数之间。如，$a+b*c-d/e$
- 后缀表达式：运算符号位于两个运算数之后。如，$abc*+de/-$

〖例〗 6 2 / 3 – 4 2 * + = ?　　8



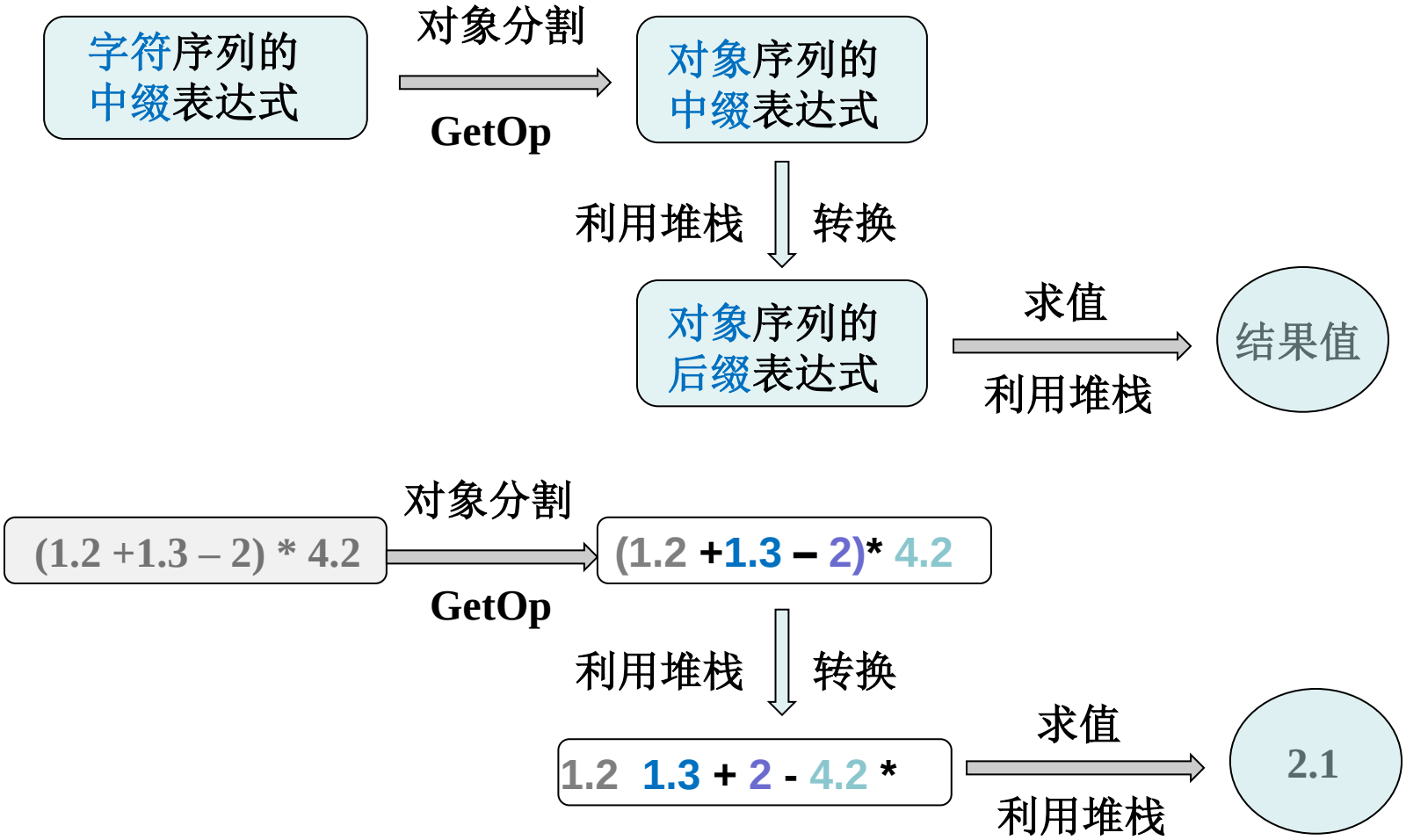
对象: 6 (运算数)	对象: 2 (运算数)
对象: / (运算符)	对象: 3 (运算数)
对象: – (运算符)	对象: 4 (运算数)
对象: 2 (运算数)	对象: * (运算符)
对象: + (运算符)	Pop: 8

$T(N)=O(N)$。不需要知道运算符的优先规则。

3.2 栈的应用

四、表达式求值

❖ 中缀表达式求值



3.2 栈的应用

四、表达式求值

❖中缀表达式转换为后缀表达式

顺序	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	×
)	>	>	>	>	×	>	>
#	<	<	<	<	<	×	=

3.2 栈的应用

四、表达式求值

❖中缀表达式转换为后缀表达式

➢ 从头到尾读取中缀表达式的每个对象，对不同对象按不同的情况处理。对象分下列6种情况：

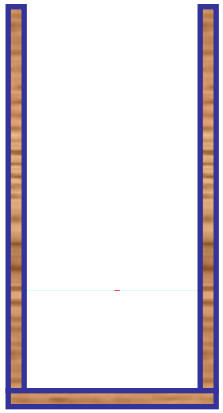
① 如果遇到空格则认为是分隔符，不需处理；
②若遇到运算数，则直接输出；
③若是左括号，则将其压入堆栈中；
④若遇到的是右括号，表明括号内的中缀表达式已经扫描完毕，将栈顶的运算符弹出并输出，直到遇到左括号（左括号也出栈，但不输出）；
⑤若遇到的是运算符，若该运算符的优先级大于栈顶运算符的优先级时，则把它压栈；若该运算符的优先级小于等于栈顶运算符时，将栈顶运算符弹出并输出，再比较新的栈顶运算符，按同样处理方法，直到该运算符大于栈顶运算符优先级为止，然后将该运算符压栈；
⑥若中缀表达式中的各对象处理完毕，则把堆栈中存留的运算符一并输出。

3.2 栈的应用

四、表达式求值

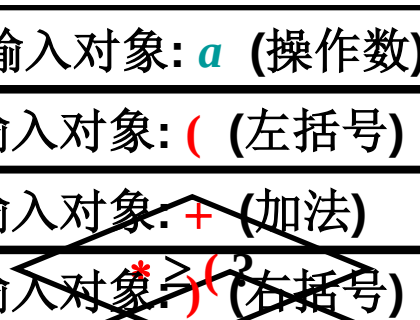
〖例〗 $a * (b + c) / d$ = ?　　$a\ b\ c + * d /$

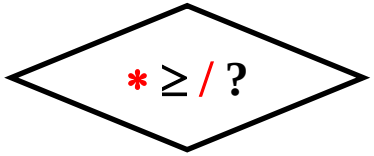
输出: a b c + * d /



← top

输入对象: a (操作数)	输入对象: * (乘法)
输入对象: ((左括号)	输入对象: b (操作数)
输入对象: + (加法)	输入对象: c (操作数)
输入对象:) (右括号)	输入对象: / (除法)
输入对象: d (操作数)	





NO?!

$T(N) = O(N)$

3.3 栈与递归实现

- 栈的另一个重要应用是在程序设计语言中实现递归调用递归调用：一个函数(或过程)直接或间接地调用自己本身，简称递归(Recursive)。
- 为保证递归调用正确执行，系统设立一个“递归工作栈”，作为整个递归调用过程期间使用的数据存储区。
  - 每一层递归包含的信息如：参数、局部变量、上一层的返回地址构成一个“工作记录”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。
- 例如：求n!

```
int f(n)
{ int res=n;
  if (n==1)
     return 1;
  else
     res = res * f(n-1);
  return res;
}
```

3.3 栈与递归实现

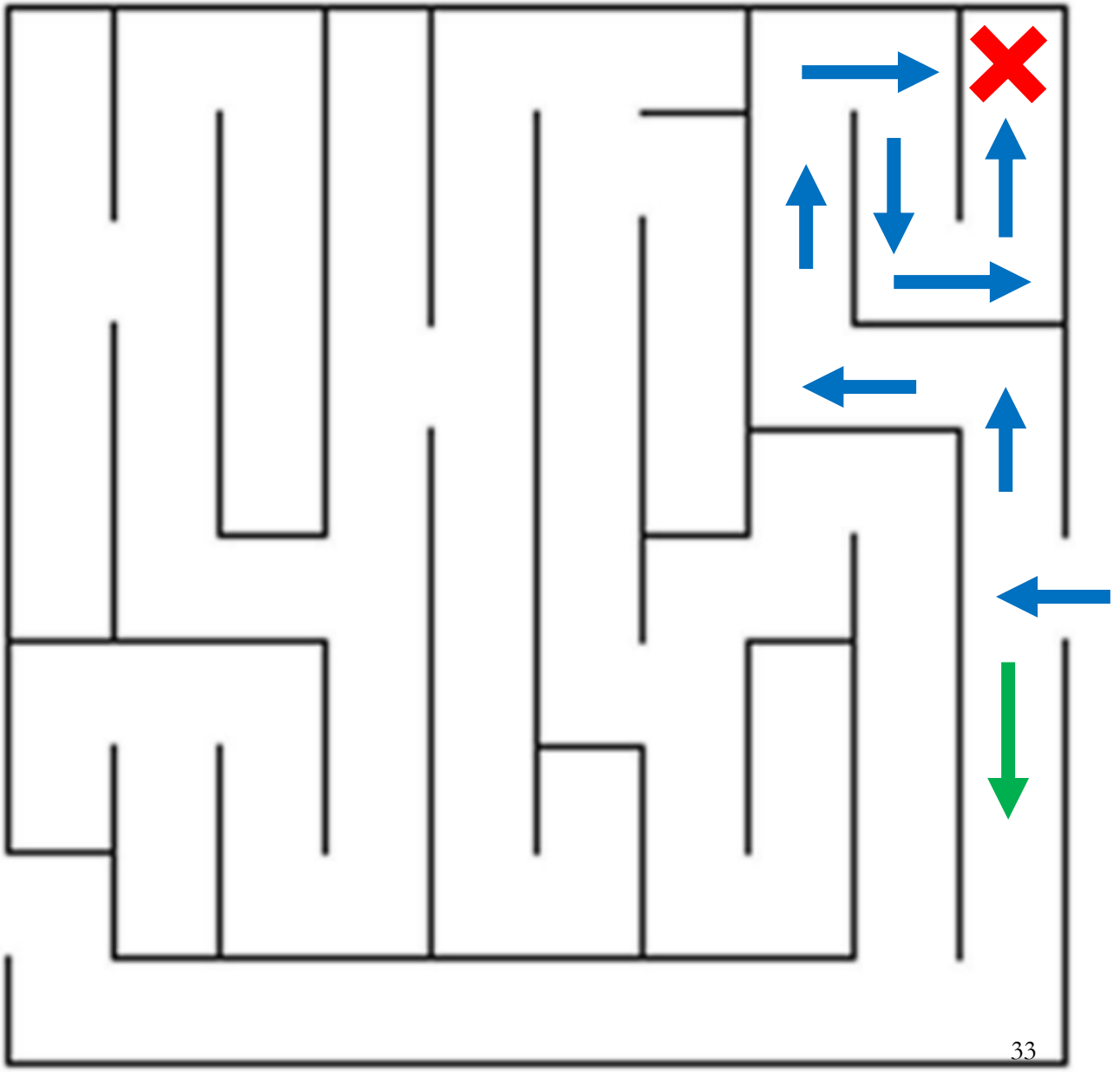
五、迷宫求解

- 迷宫求解一般采用“穷举法”
- 逐一沿顺时针方向查找相邻块（一共四块－东(右)、南(下)，西(左)、北(上)）是否可通，即该相邻块既是通道块，且不在当前路径上
- 用一个栈来记录已走过的路径

3.3 栈与递归实现

五、迷宫求解

- 举例



3.3 栈与递归实现

五、迷宫求解

- 举例

```
int move(dir, path){
        // dir: 移动方向
        // path: 当前路径
        if 反向移动或无法移动: return 0;
        移动位置，记录path;
        if 到达终点:
                输出path;
                exit 0;
        if 死路:
                return 0;
        move(right, path);
        move(up, path);
        move(left, path);
        move(down, path);
        return 0;
}
```

3.3 栈与递归实现

六. 汉诺塔

- 汉诺塔（又称Hanoi塔）问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。当所有金片从一根柱移到另一根柱时，世界将毁灭。
- 设定移动次数是f(n)
  - f(1)=1
  - f(2)=3
  - f(3)=7
  - f(k+1）=2*f(k)+1
  - 不难证明$f(n)=2^n-1$

3.3 栈与递归实现

六. 汉诺塔

- 用递归的方法实现

```
int Count=0;
void move(char x, int n, char z);
void hanoi (int n, char x, char y, char z) {
  if (n==1)
    move(x, 1, z);
  else {
    hanoi(n-1,x,z,y);
    move(x, n, z);
    hanoi(n-1, y, x, z);
  }
}

void move(char x, int n, char z) {
   printf("  %2i. Move disk %i from %c to
%c\n",++Count,n,x,z);
}
```

3.3 栈与递归实现

六. 汉诺塔

- 当n=64，移动一次用1秒，共需多长时间呢？
  - 一个平年365天有31536000 秒，闰年366天有31622400秒，平均每年31556952秒，得到：

    18446744073709551615/31556952=584554049253.855年

  - 表明移完这些金片需要5845亿年以上，而地球存在至今不过45亿年

Take Home Message

- 栈：先进后出
  - 只在top端进行数据进栈与出栈的操作
- 栈的应用：
  - 表达式的计算：先将中缀转后缀，再使用后缀进行计算
  - 递归算法