数据结构与算法

栈和队列

栈和队列

3.1 栈
- 3.1.1 栈的基本概念
- 3.1.2 顺序栈
- 3.1.3 链式栈

3.2 队列
- 3.2.1 队列的基本概念
- 3.2.2 链队列
- 3.2.3 循环队列——队列的顺序存储结构
- 3.2.4 队列应用

3.3 实例研究——表达式求值

栈和队列

- 栈和队列都是线性表。
- 基本操作是线性表操作的子集，是操作受限的线性表。所以可以称为限定性的数据结构。
- 从数据类型的角度看，他们各自有不同的操作方法，也可以认为是两种不同的抽象数据类型。

3.1.1 栈的基本概念

- 定义：栈(Stack)是限制在一端进行插入和删除运算的线性表。

- 通常称插入、删除的这一端为栈顶(Top)，另一端为栈底(Bottom)。假设栈

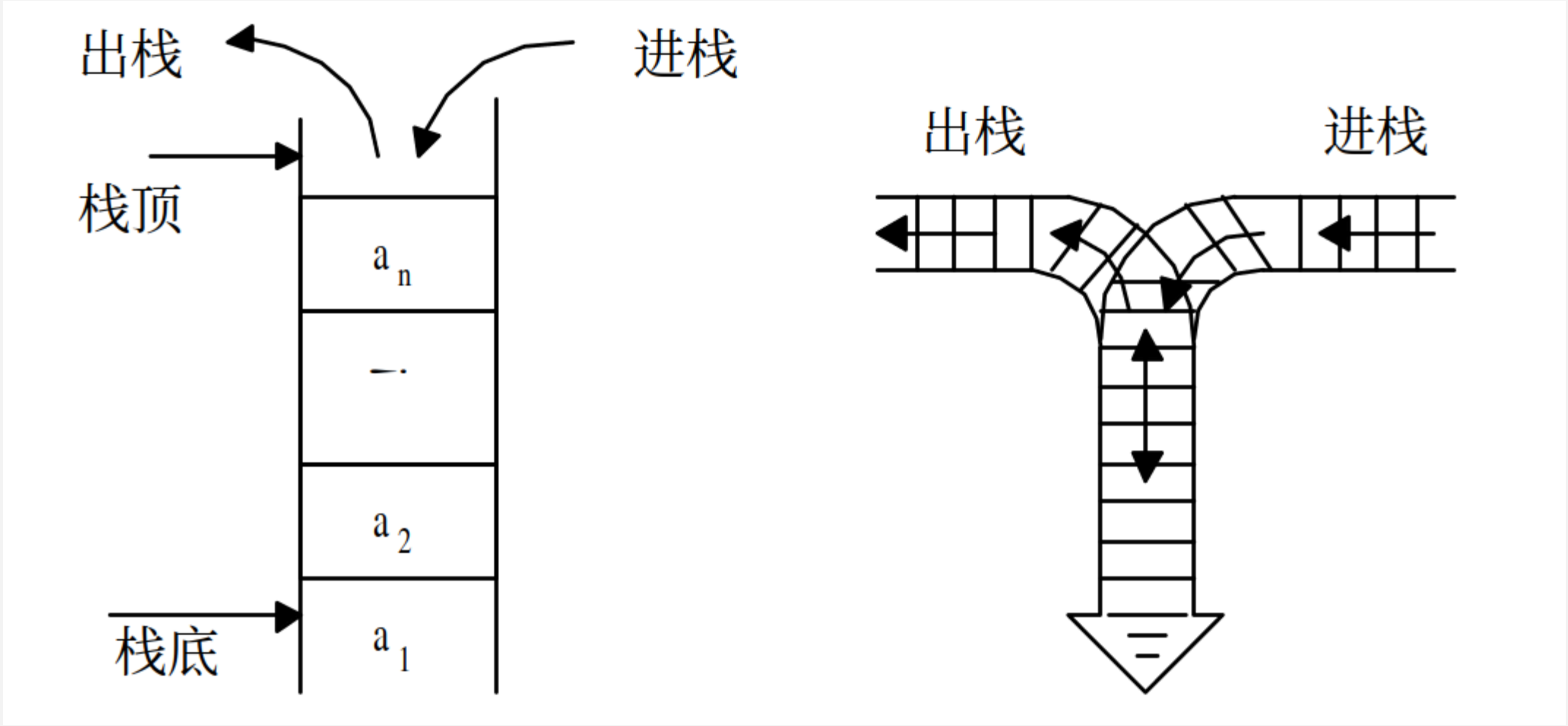
$$S=(a_1, a_2, a_3, \cdots a_n)$$

则a_1称为栈底元素，a_n为栈顶元素。栈中元素按$a_1, a_2, a_3, \cdots a_n$的次序进栈，退栈的第一个元素应为栈顶元素。

3.1.1 栈的基本概念

- 栈的修改是按后进先出的原则进行的。因此，栈称为后进先出表（LIFO）。
- 表尾称为栈顶，表头称为栈底。
- 不含元素的空表称为空栈。

3.1.1 栈的基本概念



3.1.1 栈的基本概念

ADT Stack□

数据对象： 可以是任意类型的数据，但必须属于同一个数据对象。

数据关系： 栈中数据元素之间是线性关系。 □

基本操作： □

(1) int Length() const

初始条件：栈已存在。

操作结果：返回栈元素个数。

(2) bool Empty() const

初始条件：栈已存在。

操作结果：如栈为空，则返回true，否则返回false

(3) void Clear()

初始条件：栈已存在。

操作结果：清空栈。

3.1.1 栈的基本概念

ADT Stack□

基本操作: □

(4) void Traverse(void (*visit)(const ElemType &)) const

初始条件: 栈已存在。

操作结果: 从栈底到栈顶依次对栈的每个元素调用函数(*visit)

(5) StatusCode Push(const ElemType &e)

初始条件: 栈已存在。

操作结果: 插入元素e为新的栈顶元素。

(6) StatusCode Top(ElemType &e) const

初始条件: 栈已存在且非空。

操作结果: 用e返回栈顶元素。

(7) StatusCode Pop(ElemType &e)

初始条件: 栈已存在且非空。

操作结果: 删除栈顶元素, 并用e返回栈顶元素。

3.1.1 栈的基本概念

- 栈最主要特点：栈的修改按后进先出的原则进行。栈又称为后进先出线性表。
- 栈的基本操作：栈的初始化、判空、取栈顶元素、在栈顶进行插入或删除。
- 插入元素的操作为入栈；删除栈顶元素的操作为出栈。

3.1.2 顺序栈

- 顺序栈：栈的顺序存储结构，是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，同时设置指针top指示栈顶元素在顺序栈中的位置。

- 在顺序实现中，利用数组依次存放从栈底到栈顶的数据元素。

3.1.2 顺序栈

用count存储数组中存储的栈的实际元素个数。

- 当count = 0时表示栈为空
- 入栈操作时，如栈未满，操作成功，count的值将加1
- 出栈时，如栈不空，操作成功，并且count的值将减1

3.1.2 顺序栈 - 类模板

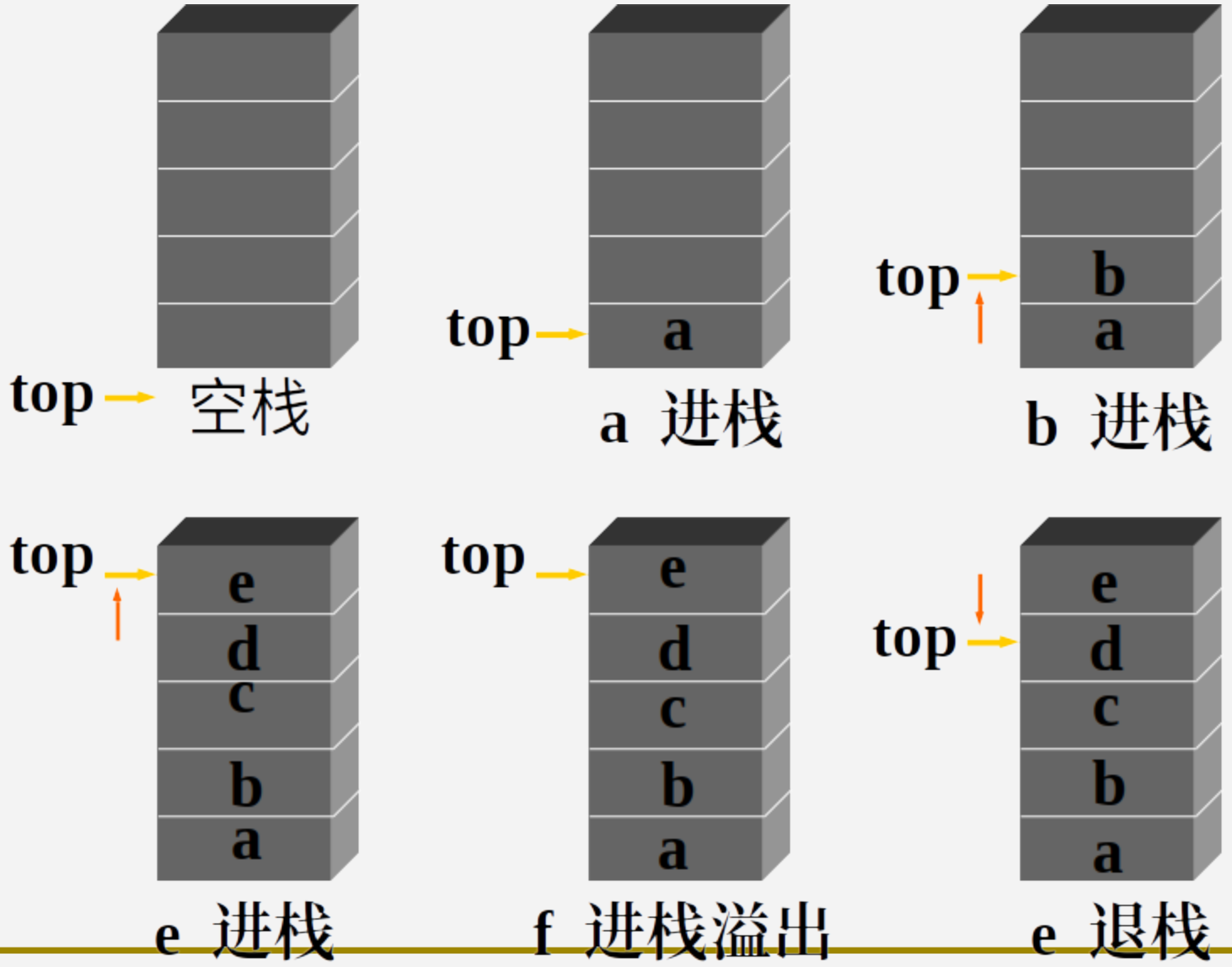
```
template<class ElemType>
class SqStack
{
protected:
   //  顺序栈的数据成员:
    int count;              // 元素个数
    int maxSize;            // 栈最大元素个数
    ElemType *elems;        // 元素存储空间

   // 辅助函数模板:
    bool Full() const;      // 判断栈是否已满
    void Init(int size);    // 初始化栈
```

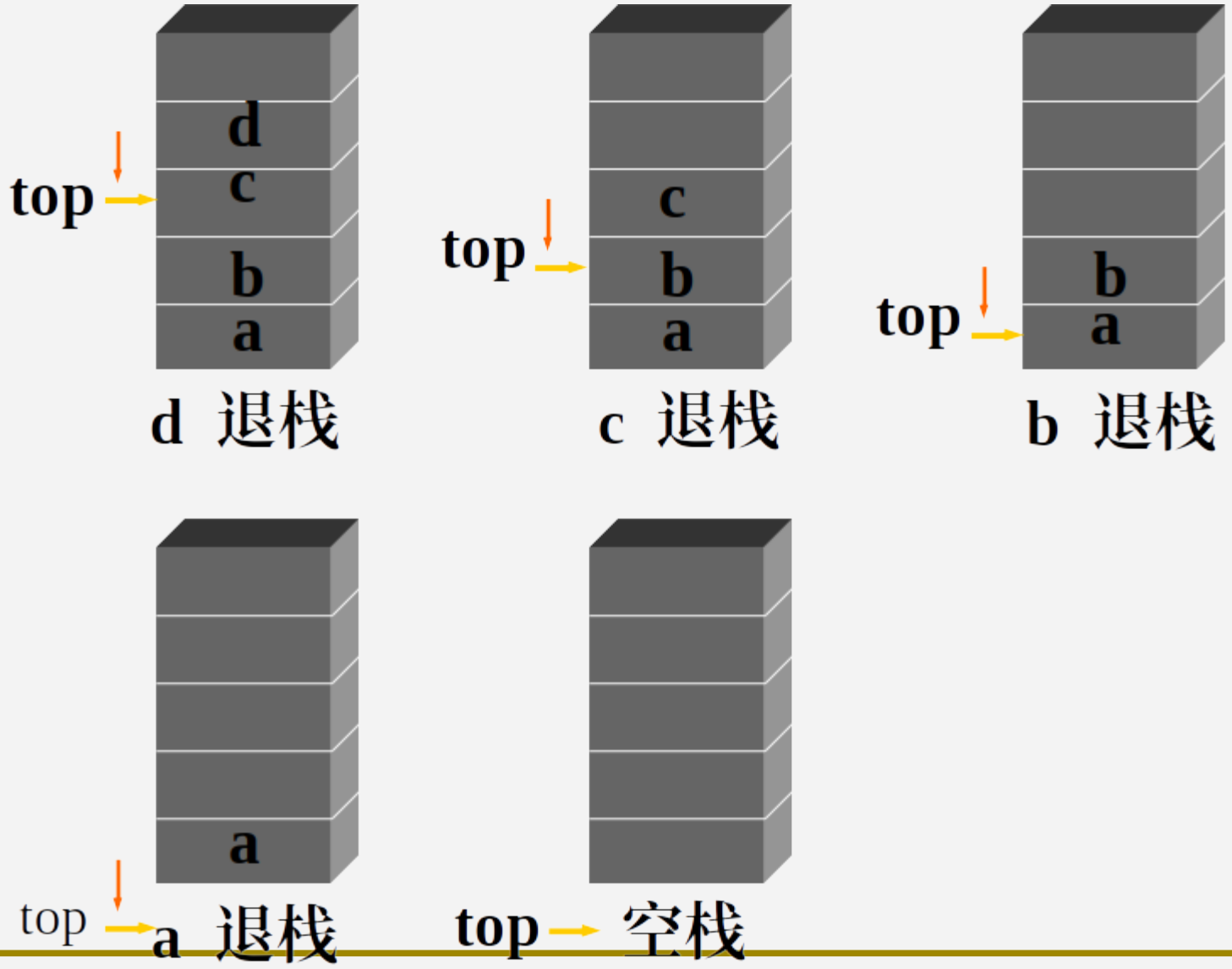
3.1.2 顺序栈 - 类模板

```
public:
    //  抽象数据类型方法声明及重载编译系统默认方法声明:
    SqStack(int size = DEFAULT_SIZE);   // 构造函数模板
    virtual ~SqStack();                 // 析构函数模板
    int Length() const;                 // 求栈长度
    bool Empty() const;                 // 判断栈是否为空
    void Clear();                       // 将栈清空
    void Traverse(void (*visit)(const ElemType &)) const;// 遍历栈
    StatusCode Push(const ElemType &e);    // 入栈
    StatusCode Top(ElemType &e) const;     // 返回栈顶
    StatusCode Pop(ElemType &e);           // 出栈
    SqStack(const SqStack<ElemType> &copy);// 复制构造函数模板
    SqStack<ElemType> &operator =(const SqStack<ElemType> &copy);
//重载运算符
};
```

3.1.2 顺序栈


top
空栈
top
a
a 进栈
top
b
a
b 进栈
top
e
d
c
b
a
e 进栈
top
e
d
c
b
a
f 进栈溢出
top
e
d
c
b
a
e 退栈

3.1.2 顺序栈



3.1.2 顺序栈

- 多栈共享空间：多个栈共享一个或多个数组，可以合理利用存储单元。
- 主要是2个栈共用一个数组，一个用前部分，一个用后部分。
- 最常用的是两个栈的共享技术：它主要利用了栈“栈底位置不变，而栈顶位置动态变化”的特性。

3.1.2 顺序栈

两个栈的共享技术

- 为两个栈申请一个共享的一维数组空间S[M]。

- 将两个栈的栈底分别放在一维数组的两端，分别是0， M-1。

- 由于两个栈顶动态变化，这样可以形成互补，使得每个栈可用的最大空间与实际使用的需求有关。由此可见，两栈共享比两个栈分别申请M/2的空间利用率高。

3.1.2 顺序栈

[例] 请用一个数组实现两个堆栈，要求最大地利用数组空间，使数组只要有空间入栈操作就可以成功。

【分析】 一种比较聪明的方法是使这两个栈分别从数组的两头开始向中间生长；当两个栈的栈顶指针相遇时，表示两个栈都满了。

3.1.2 顺序栈

```
#define MaxSize <存储数据元素的最大个数>
struct DStack {
ElementType Data[MaxSize];
int Top1;// 堆栈1的栈顶指针
int Top2;// 堆栈2的栈顶指针
} S;
S.Top1 = -1;
S.Top2 = MaxSize;
```

3.1.2 顺序栈

```
void Push( struct DStack *PtrS, ElementType item, int
Tag )
{ // Tag作为区分两个堆栈的标志，取值为1和2
   if ( PtrS->Top2 - PtrS->Top1 == 1) { // 堆栈满
      printf(“堆栈满”);
      return ;
   }
   if ( Tag == 1 ) // 对第一个堆栈操作
      PtrS->Data[++(PtrS->Top1)] = item;
   else    // 对第二个堆栈操作
      PtrS->Data[--(PtrS->Top2)] = item;
}
```

3.1.2 顺序栈

```
ElementType Pop( struct DStack *PtrS, int Tag )
{ // Tag作为区分两个堆栈的标志，取值为1和2
   if ( Tag == 1 ) { // 对第一个堆栈操作
      if ( PtrS->Top1 == -1 ) { //堆栈1空
         printf(“堆栈1空”);
         return NULL;
      } else return PtrS->Data[(PtrS->Top1)--];
   } else { // 对第二个堆栈操作
      if ( PtrS->Top2 == MaxSize ) { //堆栈2空
         printf(“堆栈2空”); return NULL;
      } else return PtrS->Data[(PtrS->Top2)++];
   }
}
```

3.1.3 链式栈

栈的链式存储结构称为链式栈，它是运算是受限的单链表。

- 链式栈无栈满问题，空间可扩充。
- 插入和删除操作仅限制在栈顶位置上进行。
- 链式栈的栈顶在链头。
- 由于只能在链表头部进行操作，故链表没有必要像单链表那样附加头结点。栈顶指针就是链表的头指针。

3.1.3 链式栈 - 类模板

```
template<class ElemType>
class LinkStack
{
protected:
   //  链栈实现的数据成员:
   Node<ElemType> *top;    // 栈顶指针
   // 辅助函数模板:
   void Init();            // 初始化栈
public:
   LinkStack();            // 无参数的构造函数
   virtual ~LinkStack();   // 析构函数
   int Length() const;     // 求栈长度
```

3.1.3 链式栈 - 类模板

```
    bool Empty() const;      // 判断栈是否为空
    void Clear();            // 将栈清空
    void Traverse(void (*Visit)(const ElemType &))const;//遍历栈
    StatusCode Push(const ElemType &e);    // 入栈
    StatusCode Top(ElemType &e) const;     // 返回栈顶元素
    StatusCode Pop(ElemType &e);           // 出栈
    LinkStack(const LinkStack<ElemType> &copy);//复制构造函数
    LinkStack<ElemType> &operator =
       (const LinkStack<ElemType> &copy); //重载赋值
};
```

3.1.3 链式栈 - 部分成员函数模板的实现

```
template<class ElemType>
LinkStack<ElemType>::LinkStack()
// 操作结果：构造一个空栈表
{
   Init();
}
template<class ElemType>
LinkStack<ElemType>::~LinkStack()
// 操作结果：销毁栈
{
   Clear();
}
```

3.1.3 链式栈 - 部分成员函数模板的实现

```
template<class ElemType>
StatusCode LinkStack<ElemType>::Push(const ElemType &e)
// 操作结果：将元素e追加到栈顶，如成功则返加SUCCESS
{
    Node<ElemType> *new_top = new Node<ElemType>(e, top);
    if (new_top == NULL) return OVER_FLOW; // 动态内存耗尽
    else
    {
        top = new_top;
        return SUCCESS; // 操作成功
    }
}
```

3.1.3 链式栈 - 部分成员函数模板的实现

```
template<class ElemType>
StatusCode LinkStack<ElemType>::Pop(ElemType &e)
// 操作结果：如栈非空，删除栈顶元素，并用e返回栈顶元素
{
   if (Empty()) return UNDER_FLOW;      // 栈空
   else {
      Node<ElemType> *old_top = top; // 旧栈顶
      e = old_top->data;              // 用e返回栈顶元素
      top = old_top->next;            // top指向新栈顶
      delete old_top;                 // 删除旧栈顶
      return SUCCESS;                 // 操作成功
   }
}
```

3.1.3 链式栈 - 部分成员函数模板的实现

```
template<class ElemType>
StatusCode LinkStack<ElemType>::Top(ElemType &e) const
// 操作结果：如栈非空，用e返回栈顶元素，返回
{
   if(Empty())
      return UNDER_FLOW;   // 栈空
   else
   {
      e = top->data;       // 用e返回栈顶元素
      return SUCCESS;      // 栈非空，操作成功
   }
}
```

3.1 栈 - 应用举例

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。以下是几个栈应用的例子。

- 括号匹配的检验
- 迷宫求解
- 栈与递归

3.1 栈 - 括号匹配的检验

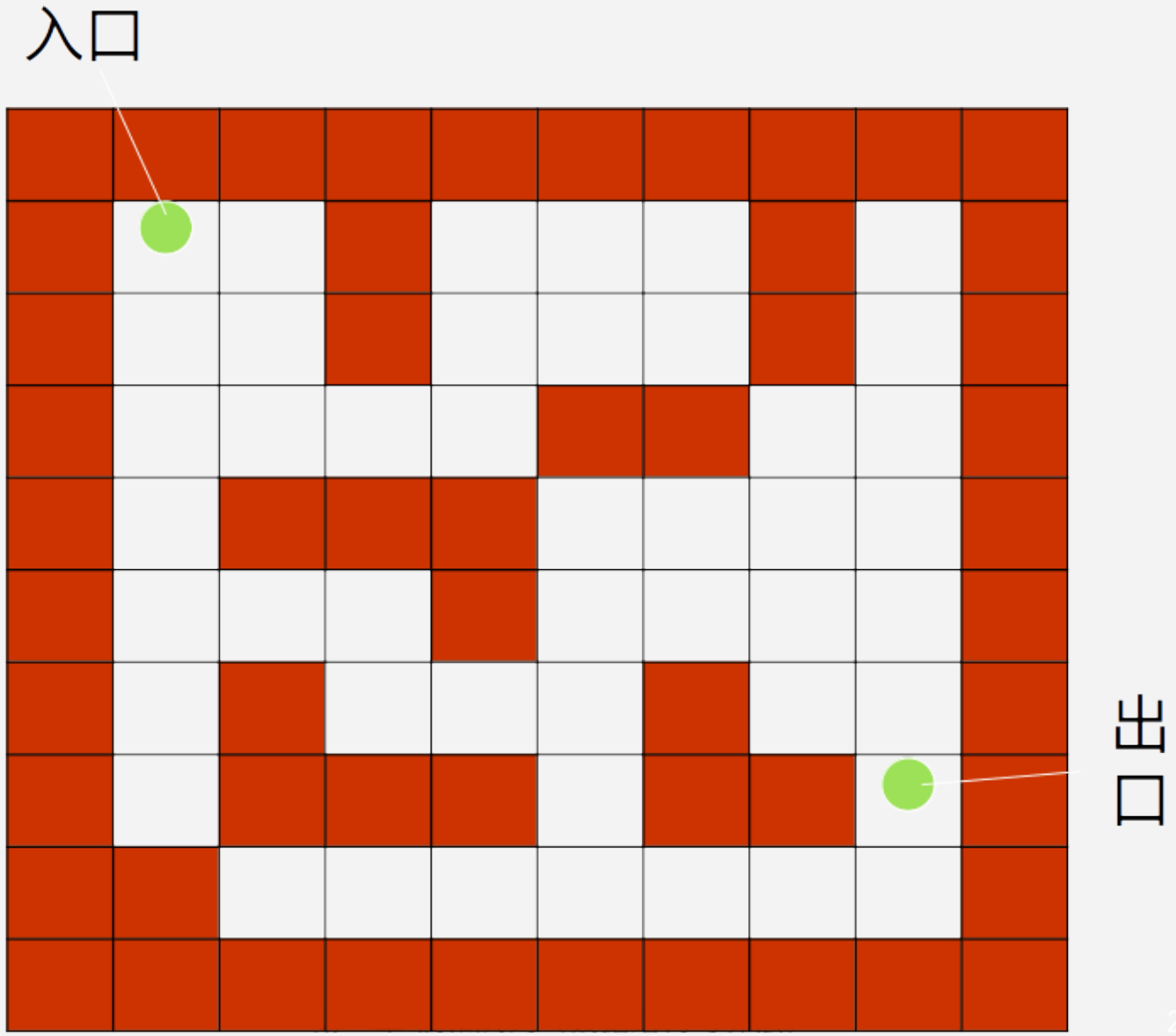
- 假设表达式中充许括号嵌套,则检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。
- 假设表达式中包含三种括号：圆括号、方括号和花括号，它们可互相嵌套，如([{}])(([]))或({([])[()]})等均为正确的格式，而{[]})或{[()]或([]}均为不正确的格式。

3.1 栈 - 括号匹配的检验

- 在检验算法中可设置一个栈，每读入一个括号，若是左括号，则直接入栈，等待相匹配的同类右括号；若读入的是右括号，且与当前栈顶的左括号同类型，则二者匹配，将栈顶的左括号出栈，否则属于不合法的情况。

- 另外，如果输入序列已读尽，而栈中仍有等待匹配的左括号，或者读入了一个右括号，而栈中已无等待匹配的左括号，均属不合法的情况。当输入序列和栈同时变为空时，说明所有括号完全匹配。

3.1 栈 – 迷宫求解



3.1 栈 - 栈与递归的实现

- 递归是指在定义自身的同时又出现了对自身的调用。
- 如果一个函数在其定义体内直接调用自己，则称其为直接递归函数。
- 如果一个函数经过一系列的中间调用语句， 通过其它函数间接调用自己，则称其为间接递归函数。

3.1 栈 - 栈与递归的实现

- 递归进层（i→i+1层）系统需要做三件事:
  - 保留本层参数与返回地址（将所有的实在参数、 返回地址等信息传递给被调用函数保存）
  - 给下层参数赋值（为被调用函数的局部变量分配存储区）
  - 将程序转移到被调函数的入口
- 而从被调用函数返回调用函数之前，递归退层（i←i+1层）系统也应完成三件工作:
  - 保存被调函数的计算结果
  - 恢复上层参数（释放被调函数的数据区）
  - 依照被调函数保存的返回地址， 将控制转移回调用函数

3.1 栈 - 练习

设一个栈的入栈序列是ABCD，则借助于一个栈所得到的出栈序列不可能是（　　）。

A）ABCD　　B）DCBA　　C）ACDB　　D）DABC

【分析】由于栈的入栈与出栈操作都在栈顶进行，在答案D中D最先出栈，此时从栈顶到栈底的元素分别为D、C、B、A，可知出栈序列应为DCBA，而不可能为DABC，所以本题应选择D。

3.1 栈 - 练习

有 5个元素，其入栈次序为：A、B、C、D、E，在各种可能的出栈次序中，以元素C第一个出栈，D第二个出栈的次序有哪几个?

【解答】按照栈的特点可知以元素C第一个出栈，D第二个出栈的次序有CDEBA 、CDBAE 和CDBEA 3种。

3.1 栈 - 练习

若元素a, b, c, d, e, f依次进栈，允许进栈、退栈操作交替进行。但不允许连续三次进行退栈工作，则不可能得到的出栈序列是（　D　）

A: dcebfa　　B: cbdaef　　C: cdbeaf　　D: afedcb

一个栈的入栈序列为1, 2, 3, . . . , n，其出栈序列是$p_1p_2p_3, . . . , p_n$。若p_2=3，则p_3可能取值的个数是

A. n-1　　B. n-2　　C. n-3　　D. 无法确定

答案 A

解析：除了3本身以外，其他的值均可以取到，因此可能取值的个数为n-1。

3.2 队列

- 抽象数据类型队列的定义
- 链队列——队列的链式表示和实现
- 循环队列——队列的顺序存储结构

3.2.1 队列 - 基本概念

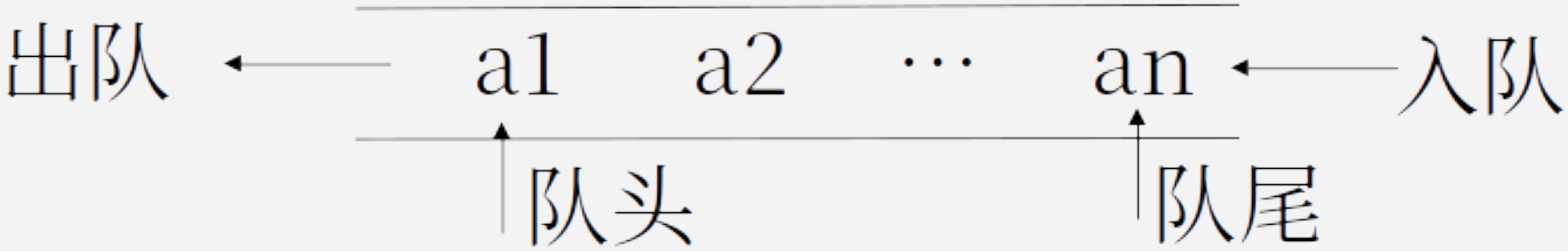
- 定义：队列是只允许在一端删除，在另一端插入的线性表。
- 在队列中，允许插入的一端叫做队尾；允许删除的一端叫做队头。
- 特性：先进先出(FIFO, First In First Out)

3.2.1 队列 - 基本概念

- 例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。
- 因此队列亦称作先进先出(First In First Out)的线性表，简称FIFO表。
- 当队列中没有元素时称为空队列。
- 在空队列中依次加入元素$a_1, a_2, \cdots a_n$之后，a_1是队头元素，a_n是队尾元素。显然退出队列的次序也只能是$a_1, a_2, \cdots a_n$ ，也就是说队列的修改是依先进先出的原则进行的。

3.2.1 队列 - 基本概念

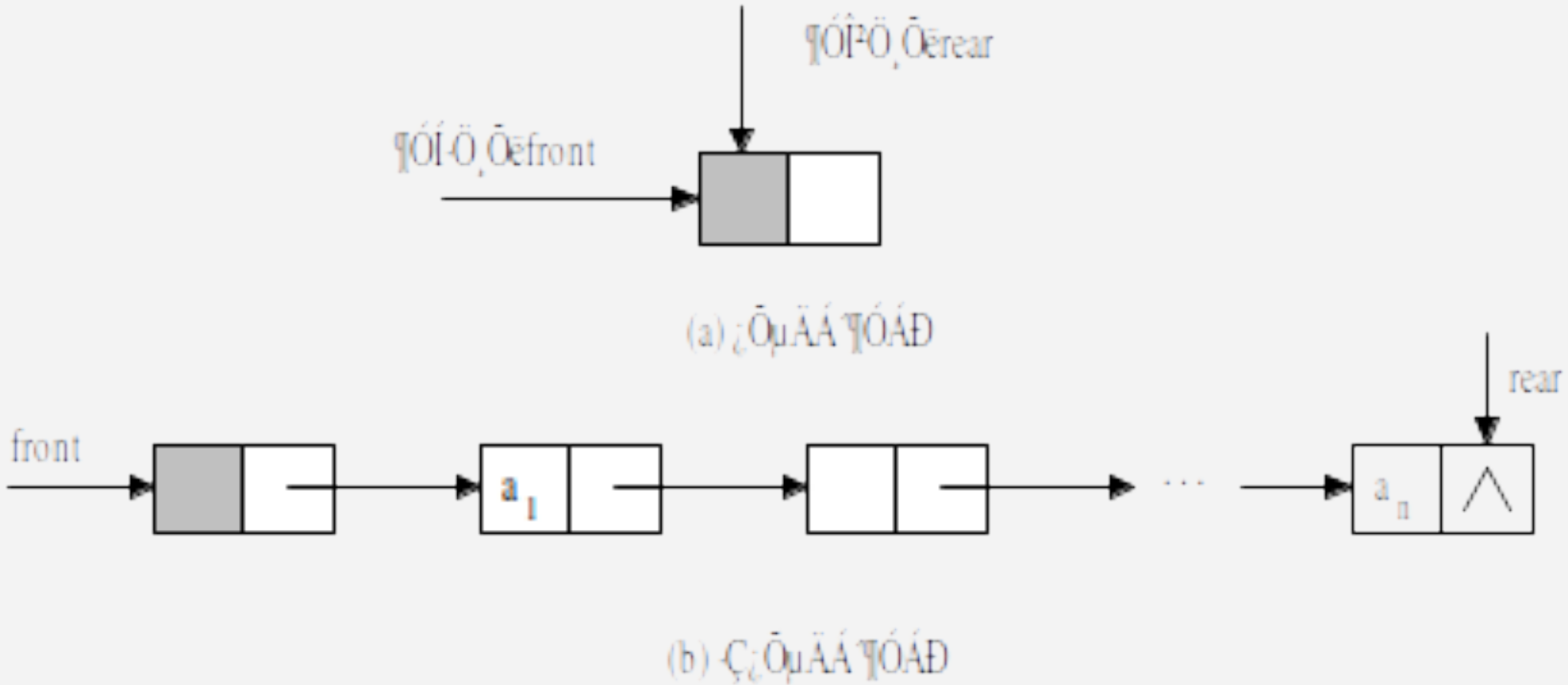
队列的示意图：



3.2.2 队列 - 链队列

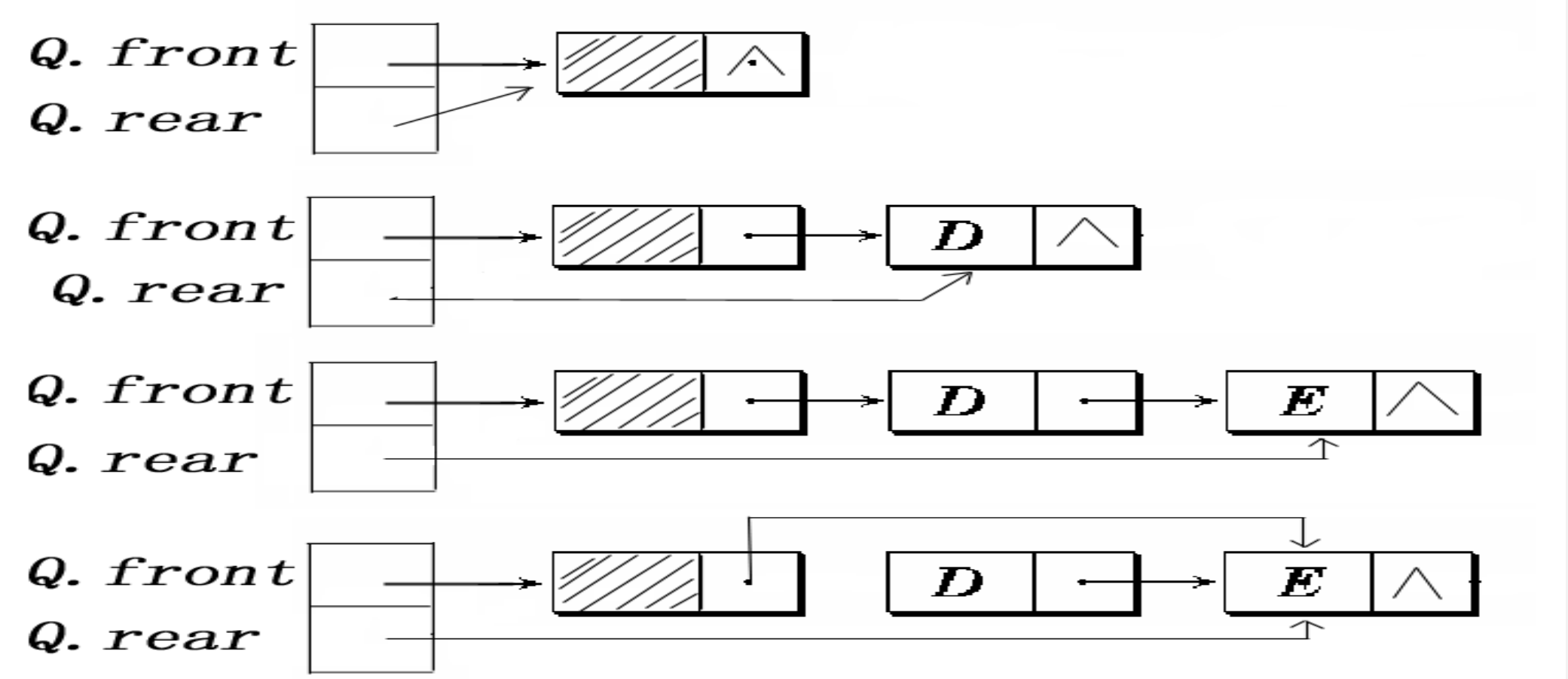
- 链式队列在入队时无队满问题，但有队空问题。
- 队空条件为 front == rear
- 链队列的操作为单链表的插入和删除操作的特殊情况，只需要修改尾指针或头指针。
- 一般情况下，删除队列头元素时只需要修改头结点中的指针，但是当队列中最后一个元素被删除后，队列尾指针也丢失了，因此需要对队尾指针重新赋值。

3.2.2 队列 – 链队列


¶ÓÎ²Ö¸Õērear
¶ÓÍ·Ö¸Õēfront
(a) ¿ÕµÄÁ¬¶ÓÁĐ
front
rear
a1
an
···
(b) ·Ç¿ÕµÄÁ¬¶ÓÁĐ

3.2.2 队列 – 链队列

队列运算指针变化状况



3.2.2 队列 - 链队列

```
template<class ElemType>
class LinkQueue
{
protected:
   //  链队列实现的数据成员:
   Node<ElemType> *front, *rear;// 队头队尾指针

   // 辅助函数模板:
   void Init();                          // 初始化队列
};
```

3.2.3 循环队列——队列的顺序存储结构

- 队列的顺序存储结构称为顺序队列，一般用一维数组来表示队列的顺序存储结构。
- 由于队头和队尾的位置是变化的，所以附设两个指针分别指向队列头元素和队列尾元素的位置。
- 每当插入新的队尾元素，尾指针增1；每当删除队头元素，头指针增1。

3.2.3 循环队列——队列的顺序存储结构

入队和出队

3.2.3 循环队列——队列的顺序存储结构

- 队满时再入队将溢出出错;
- 队空时再出队将队空处理。
- 顺序队列具有假溢现象

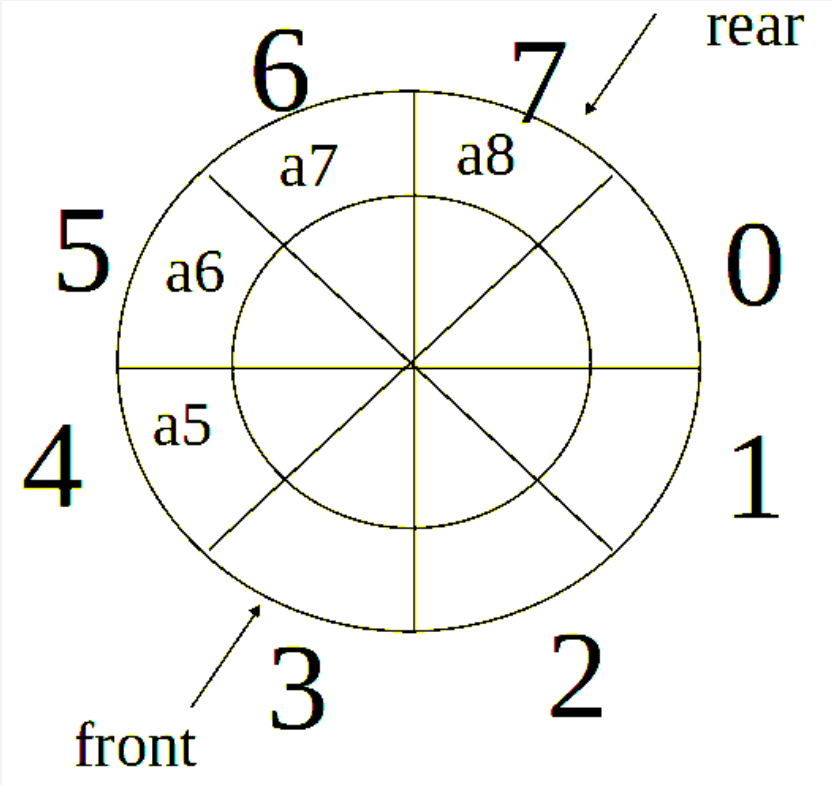
  尾指针虽然已经指向队尾，但是队列的前面部分仍然有可用空间，在这种状态时，不可再继续插入新的队尾元素，否则会出现数组越界的现象；然而此时也不宜进行再分配扩大数据空间，因为队列的实际空间并没有被占满，这是一种假溢出。

3.2.3 循环队列——队列的顺序存储结构

- 解决假溢出方法：将队列元素存放数组首尾相接，形成循环(环形)队列；另一个方法是采用数据前移，让空存储单元留在队尾。

3.2.3 循环队列——队列的顺序存储结构

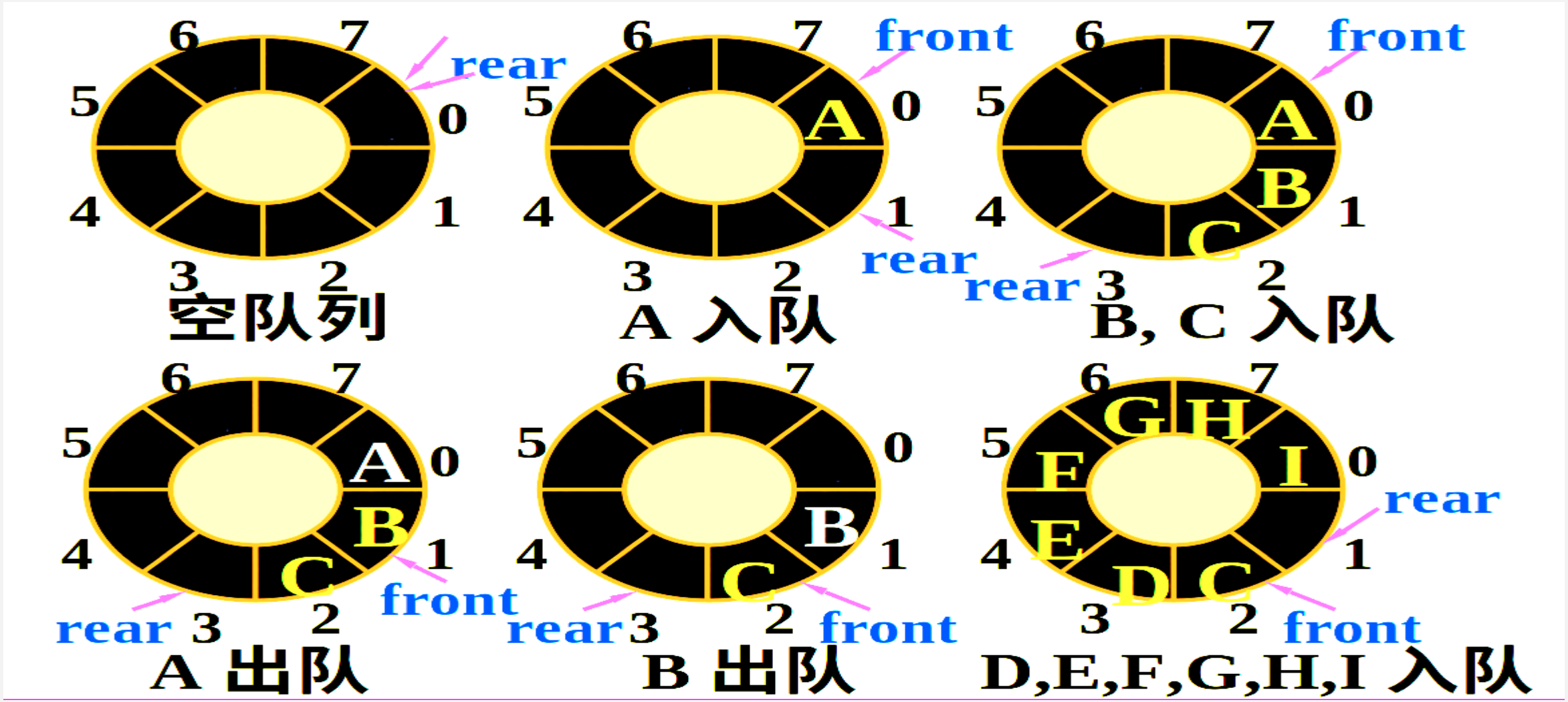
- 将存储队列元素的一维数组首尾相接，形成一个环状。我们将这种形式表示的队列称之为循环队列。



3.2.3 循环队列——队列的顺序存储结构

- 队头front、队尾rear自加1时从maxSize-1直接进到0，可用语言的取模(余数)运算实现。
- 队头front自加1:front =(front+1) % maxSize;
- 队尾rear自加1: rear =(rear+1) % maxSize;
- 队列初始化: front = rear = 0;
- 队空条件: front == rear 或 Length() == 0;
- 队满条件: (rear+1) % maxSize == front
  或 Length() == maxSize - 1。

3.2.3 循环队列——队列的顺序存储结构



3.2.3 循环队列——队列的顺序存储结构

判断循环队列满的方法

- 另设置一个标志位来区别队列是空还是满。假设此标志的名称为flag，当插入数据后遇到front等于rear的情况时，则表示队列已满，让flag=1；当删除数据遇到front等于rear，表示队列为空，让flag=0。当发生front等于rear的情况时，查看flag标志的值是0还是1，就可以知道队列是满还是空。
- 使用一个计数器记录队列中元素的总数。

3.2.3 循环队列——队列的顺序存储结构

- 判断循环队列满的方法
  - 少用一个元素空间，约定以“队列头指针在队列尾指针的下一位置（环状的下一位置）上”作为队列呈“满”状态的标志。就是允许队列最多只能存放maxsize-1个数据，牺牲数组的一个空间来避免无法分辨空队列或非空队列的问题。因此当rear指针的下一个是front的位置时，就认定为满，无法再让数据插入。
  - Q.front==(Q.rear+1)%maxsize
- 判断队列是否为空的条件是： Q.front==Q.rear

3.2.3 循环队列——类模板

```
template<class ElemType>
class SqQueue
{
protected:
   int front, rear;  // 队头队尾
   int maxSize;      // 队列最大元素个数
   ElemType *elem;   // 元素存储空间

   // 辅助函数模板:
   bool Full() const;// 判断栈是否已满
   void Init();      // 初始化队列
```

3.2.3 循环队列——类模板

```
public:
    //  抽象数据类型方法声明及重载编译系统默认方法声明:
    SqQueue(int size = DEFAULT_SIZE);// 构造函数模板
    virtual ~SqQueue();                    // 析构函数模板
    int Length() const;                    // 求队列长度
    bool Empty() const;                    // 判断队列是否为空
    void Clear();                          // 将队列清空
    void Traverse(void (*visit)(const ElemType &))const;// 遍历队列
    StatusCode OutQueue(ElemType &e);// 出队操作
    StatusCode GetHead(ElemType &e) const;// 取队头操作
    StatusCode InQueue(const ElemType &e);// 入队
    SqQueue(const SqQueue<ElemType> &copy);// 复制构造函数模板
    SqQueue<ElemType> &operator =(const
       SqQueue<ElemType> &copy);//重载赋值运算符
};
```

3.2.3 循环队列——部分成员函数的实现

```
template <class ElemType>
bool SqQueue<ElemType>::Full() const
// 操作结果：如队列已满，则返回true，否则返回false
{
    return Length() == maxSize - 1;
}

template <class ElemType>
void SqQueue<ElemType>::Init()
// 操作结果：初始化队列
{
    rear = front = 0;
}
```

3.2.3 循环队列——部分成员函数的实现

```
template<class ElemType>
int SqQueue<ElemType>::Length() const
// 操作结果：返回队列长度
{
   return (rear - front + maxSize) % maxSize;
}
template <class ElemType>
void SqQueue<ElemType>::Traverse(void (*visit)(const ElemType &))
// 操作结果：依次对队列的每个元素调用函数(*visit)
{
   for (int curPosition = front; curPosition != rear;
      curPosition = (curPosition + 1) % maxSize){
(*visit)(elem[curPosition]);
   }
}
```

3.2.3 循环队列——部分成员函数的实现

```
template<class ElemType>
StatusCode SqQueue<ElemType>::OutQueue(ElemType &e)
// 操作结果: 如果队列非空, 那么删除队头元素, 并用e
// 返回其值, 返回SUCCESS,否则返回UNDER_FLOW
{
    if (!Empty()) {              // 队列非空
        e = elem[front];         // 用e返回队头元素
        front = (front + 1) % maxSize;// 指向下一元素
        return SUCCESS;
    }
    else {                       // 队列为空
        return UNDER_FLOW;
    }
}
```

3.2.3 循环队列——部分成员函数的实现

```
template<class ElemType>
StatusCode SqQueue<ElemType>::InQueue(const ElemType &e)
// 操作结果: 如果队列已满, 返回OVER_FLOW,
// 否则插入元素e为新的队尾, 返回SUCCESS
{
    if (Full()) {                  // 队列已满
        return OVER_FLOW;
    }
    else {                         // 队列未满, 入队成功
        elem[rear] = e;            // 插入e为新队尾
        rear = (rear + 1) % maxSize;// rear指向新队尾,return
SUCCESS;
    }
}
```

3.2.3 循环队列——练习

利用两个栈S1、S2模拟一个队列（如客户队列）时，如何用栈的运算实现队列的插入、删除运算，请简述算法思想。

【解答】队列的插入：当需要作入队列操作时，用S1存放输入的元素，可用Push操作实现。

队列的删除：需要作出队操作时，可到S2栈中去取，如果栈S2为空，则将S1中元素全部送入到S2中，然后再从S2中输出元素。

3.2.3 循环队列——练习

设栈S和队列Q的初始状态均为空，元素abcdefg依次进入栈S。若每个元素出栈后立即进入队列Q，且7个元素出队的顺序是bdcfeag，则栈S的容量至少是 （ C ）

A：1　　B：2　　C：3　　D：4

某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作，则不可能得到的顺序是（ C ）

A：bacde　B：dbace　C：dbcae　D：ecbad

3.3 实例研究——表达式求值

- 表达式求值是高级语言编译中的一个基本问题，是栈的典型应用实例。
- 一个表达式由操作数(亦称运算对象)、操作符（亦称运算符）和分界符组成。

3.3 实例研究——表达式求值

- 算术表达式有三种表示
  - 中缀(infix)表示
    <操作数> <操作符> <操作数>，如 A+B;
  - 前缀(prefix)表示
    <操作符> <操作数> <操作数>，如 +AB;
  - 后缀(postfix)表示
    <操作数> <操作数> <操作符>，如 AB+;

3.3 实例研究——表达式求值

操作符的优先级——对于任意两个操作符theta1和theta2的优先关系如下:

1. theta1 < theta2: theta1优先级低于theta2。
2. theta1 = theta2: theta1优先级等于theta2。
3. theta1 > theta2: theta1优先级高于theta2。
4. theta1 e theta2: theta1与theta2不允许相继出现。

3.3 实例研究——表达式求值

theta1 \ theta2	'+'	'-'	'*'	'/'	' ('	') '	'='
'+'	>	>	<	<	<	>	>
'-'	>	>	<	<	<	>	>
'*'	>	>	>	>	<	>	>
'/'	>	>	>	>	<	>	>
' ('	<	<	<	<	<	=	e
') '	>	>	>	>	e	>	>
'='	<	<	<	<	<	e	=

3.3 实例研究——表达式求值

算符优先法

- 算符优先法就是根据上面的运算符优先关系来实现对表达式进行编译执行
- 任何表达式都可看成由操作数（operand）、操作符（operator）和界限符（delimiter）组成
- 为简单起见，只讨论算要四则运算符（“+”、“-”、“*”、“/”），操作数为常数，界限符为左右圆括和等号（“（”，“）”，“=”）；

3.3 实例研究——表达式求值

表达式的中缀表示

- 优先级高的先计算
- 优先级相同的自左向右计算
- 当使用括号时从最内层括号开始计算

3.3 实例研究——表达式求值

中缀算术表达式求值算法之一

- 使用两个栈，操作符栈optr（operator），操作数栈opnd（operand）
- 对中缀表达式求值的一般规则
  1. 在optr栈中压入一个‘=’ 。
  2. 从输入流获取一字符ch。
  3. 取出optr的栈顶optrTop 。
  4. 当optrTop != ’=’或ch != ’=’ 时，循环执行以下工作，否则结束算法。此时在opnd栈的栈顶得到运算结果。

3.3 实例研究——表达式求值

while(optrTop!='=' ||ch != '='){

① 若ch不是操作符，则将字符放回输入流(cin.putback)，读操作数operand并进opnd栈，并读入下一字符送入ch;

② 若ch是操作符，将比较ch的优先级和optrTop的优先级：

1、若optrTop<ch，则ch进optr栈，从中缀表达式取下一字符送入ch;

2、若optrTop>ch，则从opnd栈退出a2和a1，从optr栈退出θ，形成运算指令 (a1) θ (a2)，结果进opnd栈;

3、若optrTop=ch(此处特指optrTop与ch的优先级相等)且ch == ')'，此时optrTop为'('，则从optr栈退出栈顶的‘(’，对消括号，然后从中缀表达式取下一字符送入ch ;

4、 若optrTop e ch，则出现表达式错误，终止执行。

③ 取出optr的栈顶optrTop。

}

3.3 实例研究——表达式求值

中缀表达式中操作符的优先级的另一个表示方法

- Isp叫做栈内(in stack priority)优先数，实际为表达式中左操作符的优先数。
- Icp叫做栈外(in coming priority)优先数，实际为表达式中右操作符的优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的'='号与输入流最后的'='号配对时。

操作符 ch	'='	'('	'*', '/', '%'	'+', '-'	')'
Isp (栈内)	0	1	5	3	6
Icp (栈外)	0	6	4	2	1

3.3 实例研究——表达式求值

中缀算术表达式求值的另一种算法

- 使用两个栈，操作符栈optr（operator），操作数栈opnd（operand），
- 对中缀表达式求值的一般规则：
  1. 在optr栈中压入一个‘=’ 。
  2. 从输入流获取一字符ch。
  3. 取出optr的栈顶optrTop 。
  4. 当optrTop != ’=’或ch != ’=’ 时，循环执行以下工作，否则结束算法。此时在opnd栈的栈顶得到运算结果。

3.3 实例研究——表达式求值

while(optrTop != '=' || ch != '='){

① 若ch不是操作符，则将字符放回输入流(cin.putback)，读操作数operand并进opnd栈，并读入下一字符送入ch;

② 若ch是操作符，比较Icp(ch)的优先级和Isp(optrTop)的优先级：

1、若Isp(optrTop)<Icp(ch)，则ch进optr栈，从中缀表达式取下一字符送入ch;

2、若Isp(optrTop)>Icp(ch)，则从opnd栈退出a2和a1，从optr栈退出θ，形成运算指令 (a1) θ (a2)，结果进opnd栈;

3、若Isp(optrTop)==Icp(ch)且ch == ')'，则从optr栈退出栈顶的'('，对消括号，然后从中缀表达式取下一字符送入ch 。

③取出optr的栈顶optrTop。

}

3.3 实例研究——表达式求值

表达式3*2^(4+2*2-6*3)-5在求值过程中，当扫描到6时，对象栈和运算符栈为（D），其中^表示乘幂

A. 3,2,4,2,2; *^(+*-

B. 3,2,8; *^(+*-

C. 3,2,4,2,2; *^(-

D. 3,2,8; *^(-

实例研究——表达式求值(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。

① 运算数：直接输出；

② 左括号：压入堆栈；

③ 右括号：将栈顶的运算符弹出并输出，直到遇到左括号（出栈，不输出）；

④ 运算符：

- 若优先级大于栈顶运算符时，则把它压栈；
- 若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，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；

⑤ 若各对象处理完毕，则把堆栈中存留的运算符一并输出。

实例研究——表达式求值(后缀表达式)

中缀表达式转换为后缀表达式： 2*(9+6/3-5)+4

步骤	待处理表达式	堆栈状态（底←→顶）	输出状态
1	2*（9+6/3-5）+4		
2	*（9+6/3-5）+4		2
3	（9+6/3-5）+4	*	2
4	9+6/3-5）+4	* （	2
5	+6/3-5）+4	* （	2 9
6	6/3-5）+4	* （ +	2 9
7	/3-5）+4	* （ +	2 9 6
8	3-5）+4	* （ + /	2 9 6
9	-5）+4	* （ + /	2 9 6 3
10	5）+4	* （ -	2 9 6 3 / +
11	）+4	* （ -	2 9 6 3 / + 5
12	+4	*	2 9 6 3 / + 5 -
13	4	+	2 9 6 3 / + 5 - *
14		+	2 9 6 3 / + 5 - * 4
15			2 9 6 3 / + 5 - * 4 +

实例研究——表达式求值(后缀表达式)

已知操作符包括'+'、'-'、'*'、'/'、'('和')'。将中缀表达式a+b-a*((c+d)/e-f)+g转换为等价的后缀表达式ab+acd+e/f-*-g+时，用栈来存放暂时还不能确定运算次序的操作符，若栈初始时为空，则转换过程中同时保存在栈中的操作符的最大个数是

A. 5　　B. 7　　C. 8　　D. 11

答案：A

扫描项	栈	后缀表达式
a+b	+	ab
-	-	ab+
a*	-*	ab+a
((c+d	-*((+	ab+acd
)	-*(	ab+acd+
/e	-*(/	ab+acd+e
-	-*(-	ab+acd+e/
f)	-*	ab+acd+e/f-
+	+	ab+acd+e/f-*-
g	+	ab+acd+e/f-*-g
		ab+acd+e/f-*-g+

实例研究——表达式求值(后缀表达式)

表达式a*(b+c)-d的后缀表达式是（B）

A. abcd*+-　　B. abc+*d-

C. abc*+d-　　D. -+*abcd

表达式a-b*c/d+e/f扫描到d时，运算符栈的栈顶元素是（B）

A. *　　B. /　　C. -　　D. -/