数据结构与算法

第三章　栈和队列

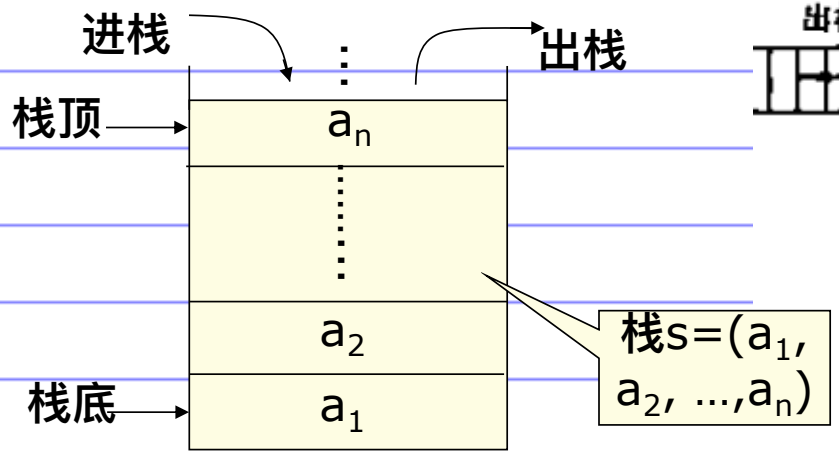
栈
栈的应用举例
队列

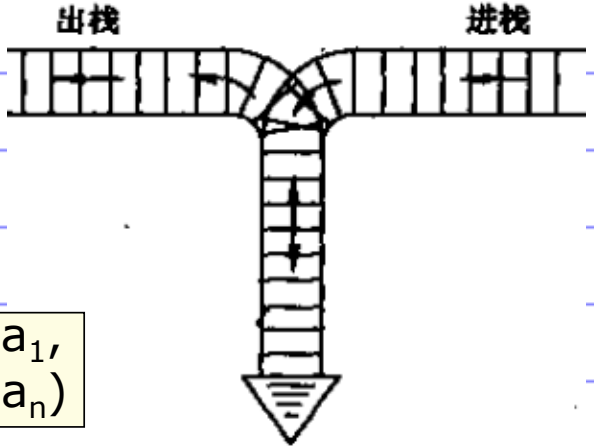
前言

- 栈和队列是两种特殊的线性表，其特殊性在于栈和队列的基本操作是线性表的子集，它们是操作受限的线性表，可称为限定性的DS.

第一节 栈

- 栈(stack)
  - 限定仅在表尾进行插入或删除操作的线性表
    - 表尾—栈顶
    - 表头—栈底
    - 不含元素的空表称空栈
  - 特点：先进后出（FILO）或后进先出（LIFO）





栈的基本操作

- 教材P44-45关于栈的抽象描述

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 FALSE。

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。

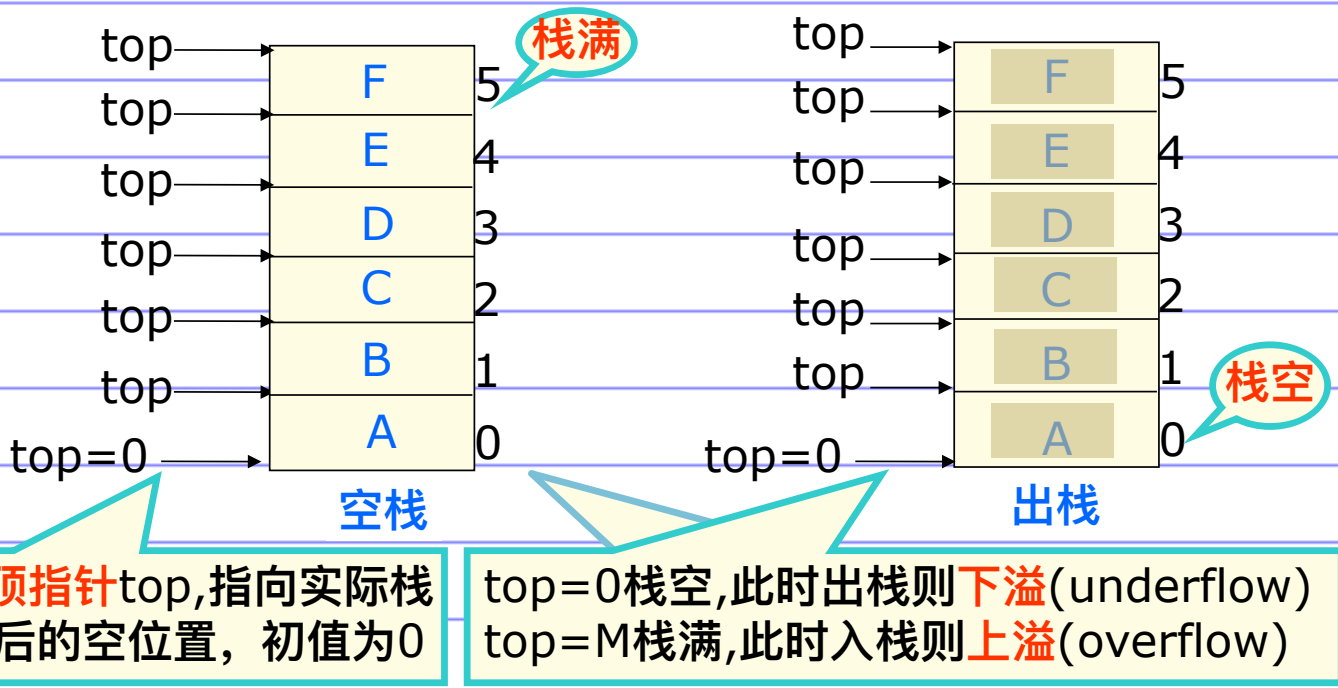
Push(&S, e)
初始条件：栈 S 已存在。
操作结果：插入元素 e 为新的栈顶元素。

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。

StackTraverse(S, visit())
初始条件：栈 S 已存在且非空。
操作结果：从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失效。

栈的顺序存储结构

- 顺序栈—实现：一维数组s[M]
- 当堆栈中数据全部弹出后，在内存中的是什么？



栈的顺序存储表示

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
Typedef struct {
    SElemType *base;  //在栈构造之前和销毁之后，base值为NULL
    SElemType *top;   //栈顶指针
    int stacksize;
} SqStack;
```

栈的基本操作

- Status InitStack(SqStack &S);
- Status DestroyStack(SqStack &S);
- Status ClearStack(SqStack &S);
- Status StackEmpty(SqStack S);
- int StackLength(SqStack S);
- Status GetTop(SqStack S, SElemtype &e);
- Status Push(SqStack &S, SElemType e);
- Status Pop(SqStack &S, SElemType &e);
- Status StackTraverse(SqStack S, Status(*visit)());

顺序栈算法

```
Status InitStack(SqStack &S) {
  S.base =
   (SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
  if(!S.base) exit(OVERFLOW);
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

顺序栈算法

- 入栈算法、出栈算法

```
Status  push(SqStack &S, SElemType e) {
   if(S.top – S.base >= S.stacksize) {
      S.base = (SElemType *)realloc(S.base,
            (S.stacksize+STACKINCREMENT)*siezeof(SElemType));
      if(!S.base) exit(OVERFLOW);
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
   }
   *S.top++ = e;
   return OK;
}
```

顺序栈算法

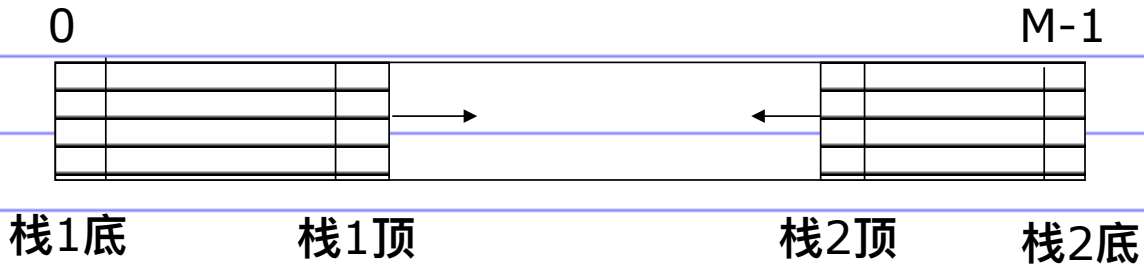
```
Status Pop(SqStack &S, SElemType &e) {
    if(S.top==S.base) return ERROR;
    e = *--S.top;
    return OK;
}
```

```
Status GetTop(SqStack S, SElemType &e) {
    if(S.top==S.base) return ERROR;
    e = *(S.top-1);
    return OK;
}
```

- 用数组实现的顺序栈的评价
  - 栈的容量在使用前难以估计
  - 操作简便

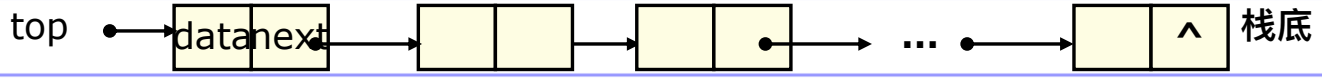
共享顺序栈

- 顺序栈所需容量在使用前难以估计，而且有些CPU内存有限，可供堆栈使用的空间有限，如单片机。常在一个程序中将两个堆栈使用的空间放在一起。



栈的链式存储结构

- 若一个程序中要使用多于两个的栈，则可以采用链表作为存储结构，这种存储结构通常称为链栈(linked-stack).



- 结点定义

```
typedef  struct tagLinkedStack
{
   int data;
   struct tagLinkedStack *next;
} LinkedStack;
```

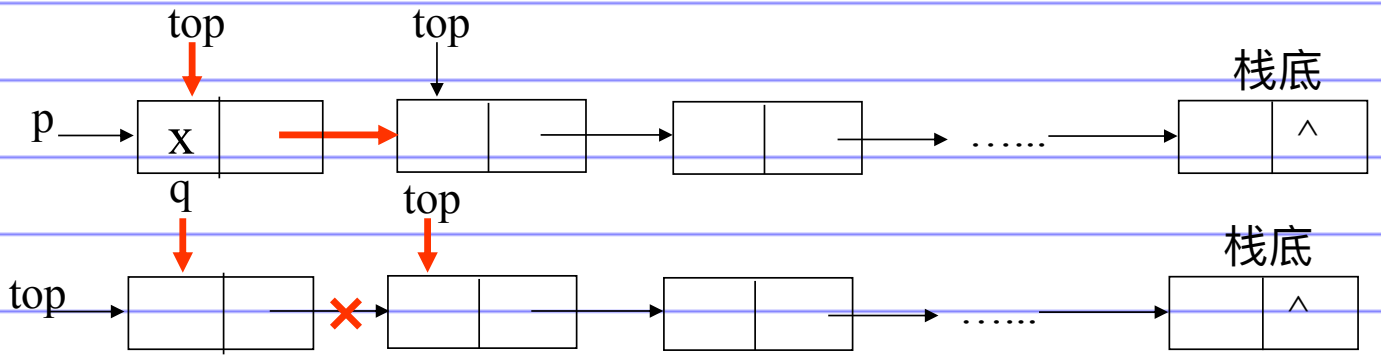
- 入栈算法
- 出栈算法

链栈算法

- 链栈通常都在链表头端操作。在尾端操作呢？

```
LinkedStack *LSPush(
   LinkedStack *top, int x)
{ LinkedStack *p;
   p =(LinkedStack *)
            malloc(sizeof(LinkedStack));
   p->data=x;
   p->next=top;
   top=p;
   return(p);
}
```

```
LinkedStack *LSPop(
   LinkedStack *top,int *x)
{  LinkedStack *q;
   if (top) {
      q=top;
      *x=top->data;
      top=top->next;
      free(q); }
   return(top);
}
```

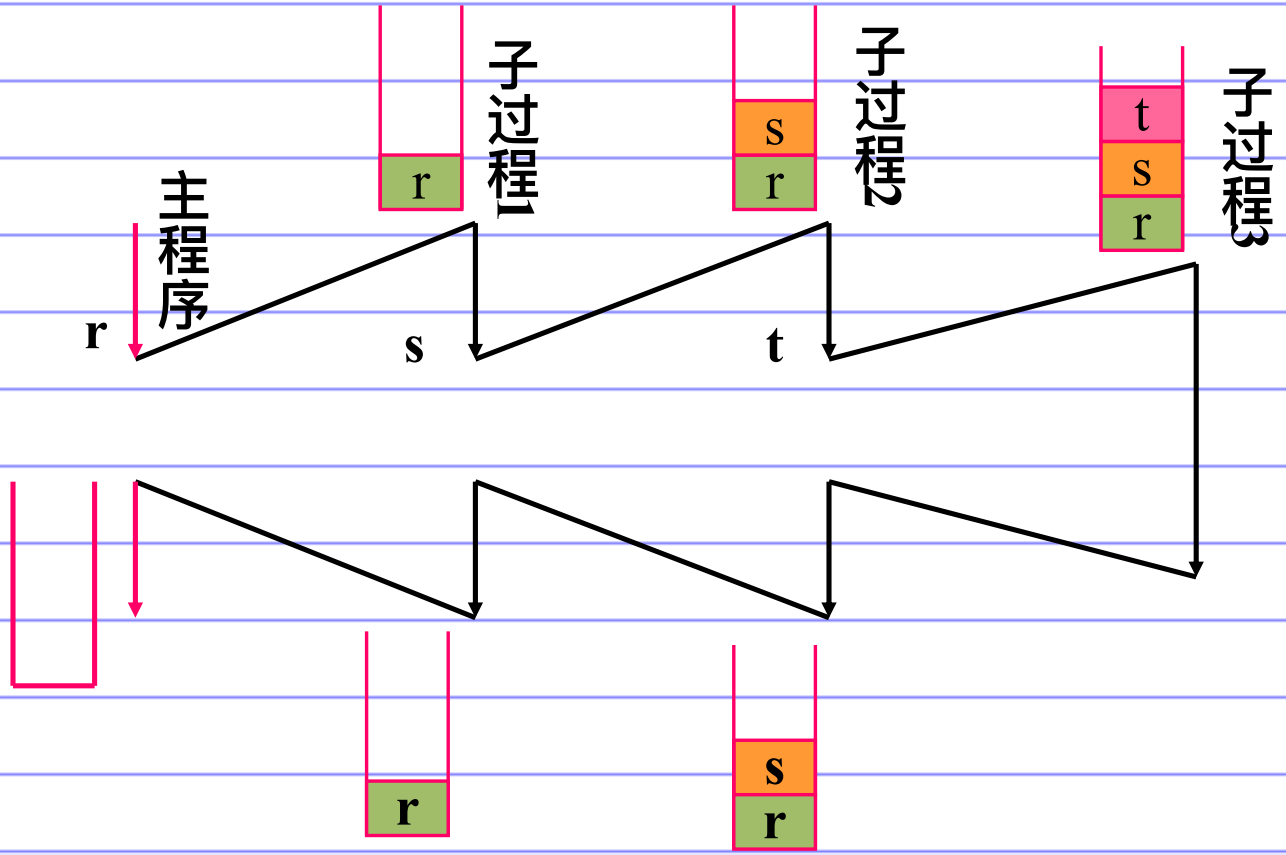


第二节 栈的应用举例

- 过程的嵌套调用
- 数制转换
- 括号的匹配
- 回文游戏
- 走迷宫
- 地图着色问题
- 表达式求值

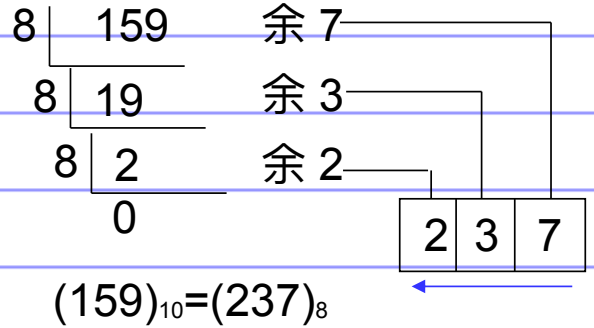
过程的嵌套调用

- 子程序调用-调用时入栈，返回时出栈

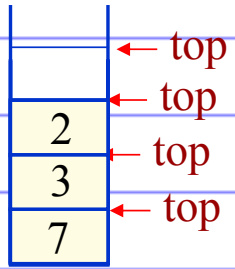


数制转换

- 把十进制数159转换成八进制数



$(159)_{10}=(237)_8$




```
void conversion (){
  // 构造空栈
  InitStack(S);
  scanf ("%d",N);
  while (N) {
    Push(S, N % 8);
    N = N/8;
  }
  while (! StackEmpty) {
    Pop(S,e);
    printf ("%d", e );
  }
} // conversion
```

括号的匹配

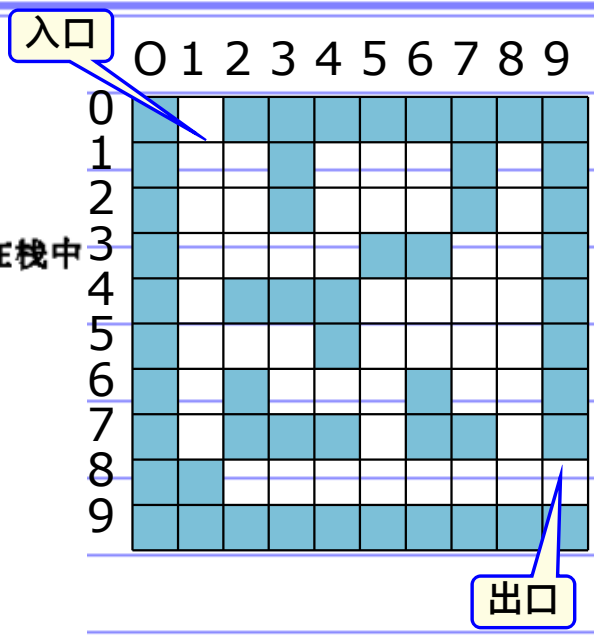
- 考虑下列括号序列
  - [([][])]
  - 要求成对出现,但可以嵌套
- 进一步考虑算术表达式中的括号，只允许大(花)括号中套中括号、中括号中套小括号
  - {[()]}
- 考虑字符串，在JavaScript脚本语言中，字符串是可以用双引号””或单引号’’包括的一个串。用双引号括起的串中可以有单引号，而用单引号括起的串中可以有双引号。

回文游戏

- 顺读与逆读字符串一样(不含空格)
- 例如：
  - ini
  - madam im adam
- 算法
  - 读入字符串
  - 去掉空格（原串）
  - 压入栈
  - 原串字符与出栈字符依次比较
    - 若不等，非回文
    - 若直到栈空都相等，回文

走迷宫

```
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经探索,
        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);
```

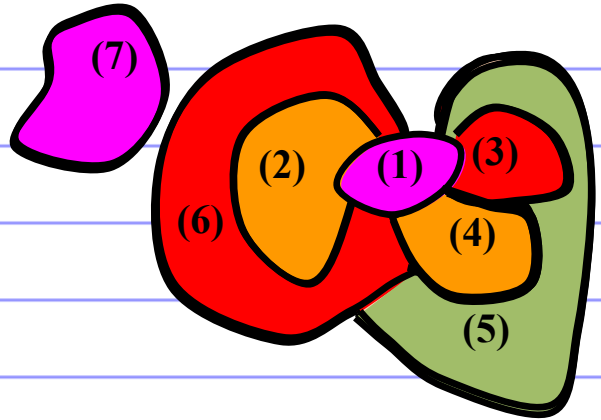


地图着色问题

- 地图着色问题

邻接矩阵R[7][7]

	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0



1	2	3	4	5	6	7
1	2	3	2	4	3	1

已经证明，地图可以只用四种颜色着色

1# 紫色
2# 黄色
3# 红色
4# 绿色

表达式求值

- 运算规则：
  - 从左到右
  - 先乘除、后加减
  - 先括号内、后括号外
- 中缀表达式

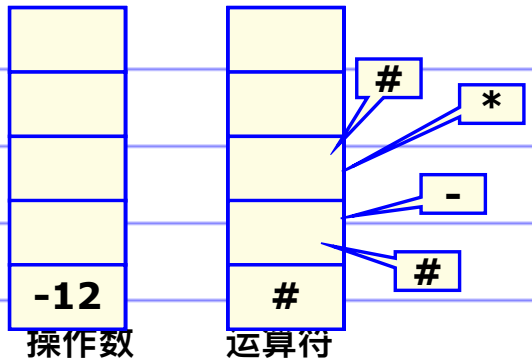
  a*b+c

  a+b*c

  a+(b*c+d)/e
- 操作数栈和运算符栈
- 例　计算 2+4-3*6
- 算法见教材P53.

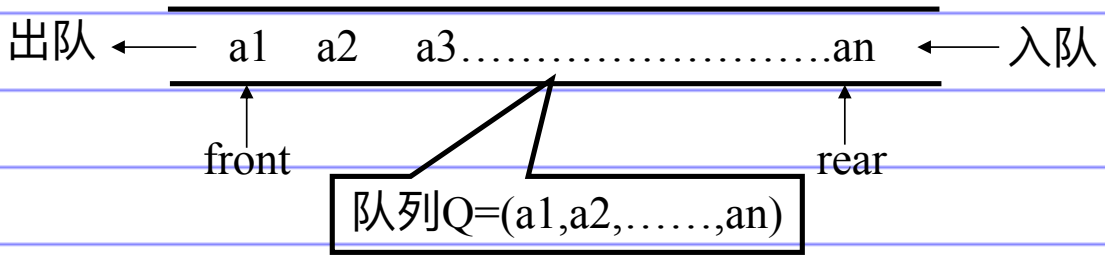
算符间的优先关系

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



第三节 队列

- 队列的定义及特点
  - 定义：队列是限定只能在表的一端进行插入，在表的另一端进行删除的线性表
    - 队尾(rear)——允许插入的一端
    - 队头(front)——允许删除的一端
  - 队列特点：先进先出(FIFO)



队列的抽象描述

ADT Queue {

数据对象：D = { a_i | $a_i \in$ ElemSet, i = 1, 2, …, n, n ≥ 0 }

数据关系：R1 = { <a_{i-1}, a_i> | a_{i-1}, $a_i \in$ D, i = 2, …, n }

约定其中 a_1 端为队列头，a_n 端为队列尾。

基本操作：

InitQueue(&Q)

操作结果：构造一个空队列 Q。

DestroyQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回 TRUE，否则 FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q, &e)

初始条件：Q 为非空队列。

操作结果：用 e 返回 Q 的队头元素。

EnQueue(&Q, e)

初始条件：队列 Q 已存在。

操作结果：插入元素 e 为 Q 的新的队尾元素。

DeQueue(&Q, &e)

初始条件：Q 为非空队列。

操作结果：删除 Q 的队头元素，并用 e 返回其值。

QueueTraverse(Q, visit())

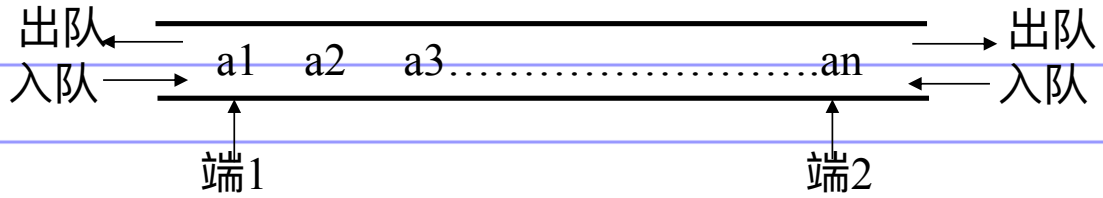
初始条件：Q 已存在且非空。

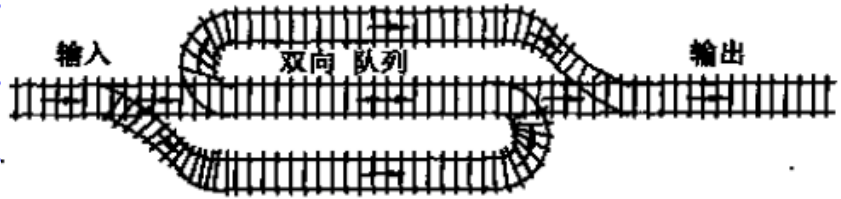
操作结果：从队头到队尾，依次对 Q 的每个数据元素调用函数 visit()。一旦 visit() 失败，则操作失败。

} ADT Queue

双端队列

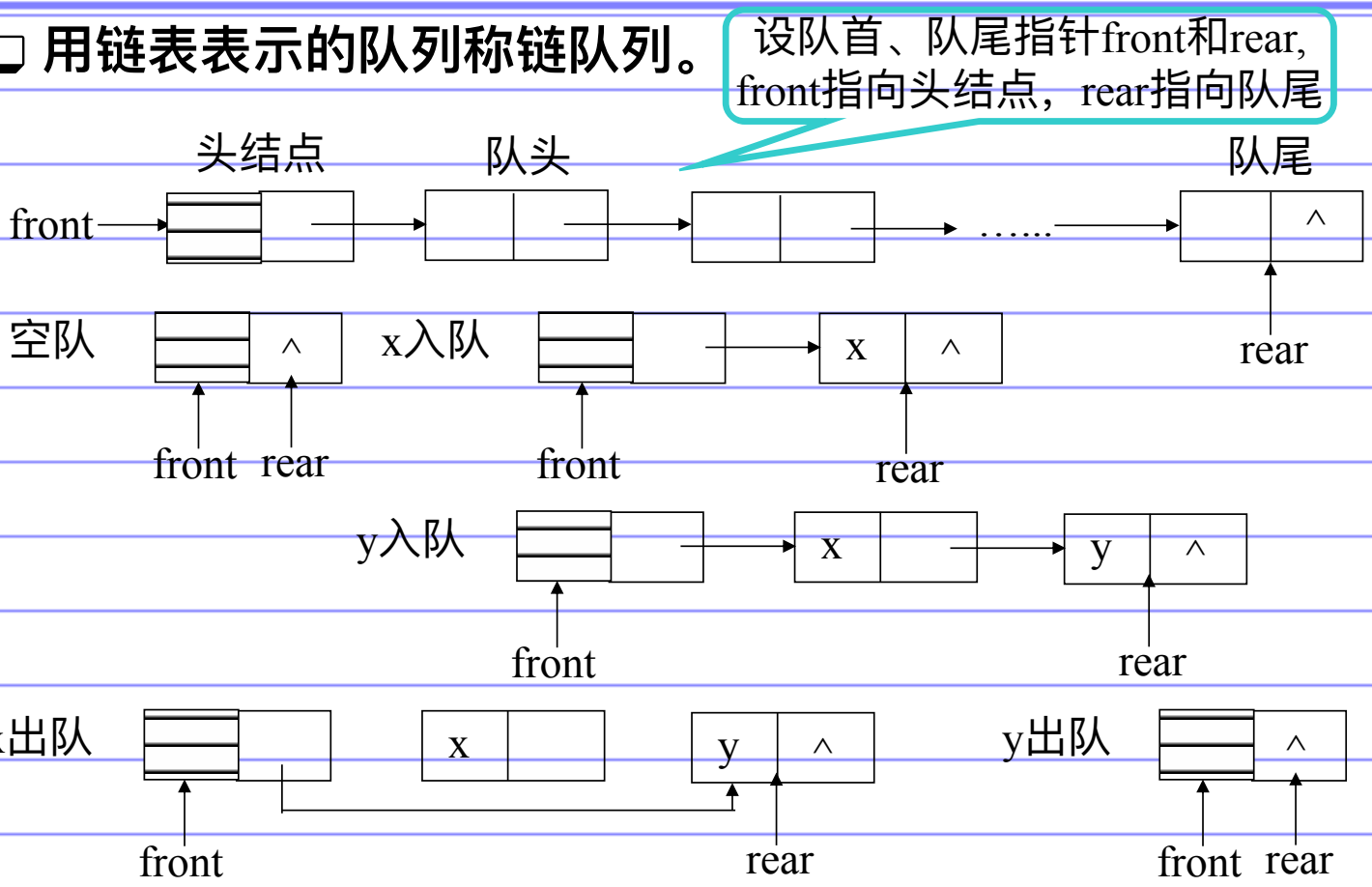
- 定义：限定插入和删除操作可在表的两端进行的线性表。



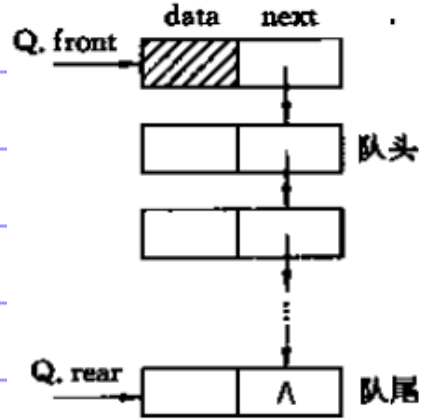


链队列

- 用链表表示的队列称链队列。



链队列基本操作




```
// 队列的链式存储结构
typedef struct QNode {
    QElemType       data;
    struct QNode    * next;
}QNode,  * QueuePtr;
typedef struct {
    QueuePtr   front; // 队头指针
    QueuePtr   rear;  // 队尾指针
}LinkQueue;
```

```
// 基本操作
Status InitQueue (LinkQueue &Q)
    // 构造一个空队列 Q
Status DestroyQueue (LinkQueue &Q)
    // 销毁队列 Q, Q 不再存在
Status ClearQueue (LinkQueue &Q)
    // 将 Q 清为空队列
Status QueueEmpty (LinkQueue Q)
    // 若队列 Q 为空队列, 则返回 TRUE,
    // 否则返回 FALSE
int QueueLength (LinkQueue Q)
    // 返回 Q 的元素个数, 即为队列的长度
Status GetHead (LinkQueue Q, QElemType &e)
    // 若队列不空, 则用 e 返回 Q 的队头元素,
    // 并返回 OK; 否则返回 ERROR
Status EnQueue (LinkQueue &Q, QElemType e)
    // 插入元素 e 为 Q 的新的队尾元素
Status DeQueue (LinkQueue &Q, QElemType &e)
    // 若队列不空, 则删除 Q 的队头元素, 用 e
    // 返回其值, 并返回 OK; 否则返回 ERROR
Status QueueTraverse(LinkQueue Q, visit())
    // 从队头到队尾依次对队列 Q 中每个元素
    // 调用函数 visit()。
    // 一旦 visit 失败, 则操作失败。
```

链队列基本操作算法实现

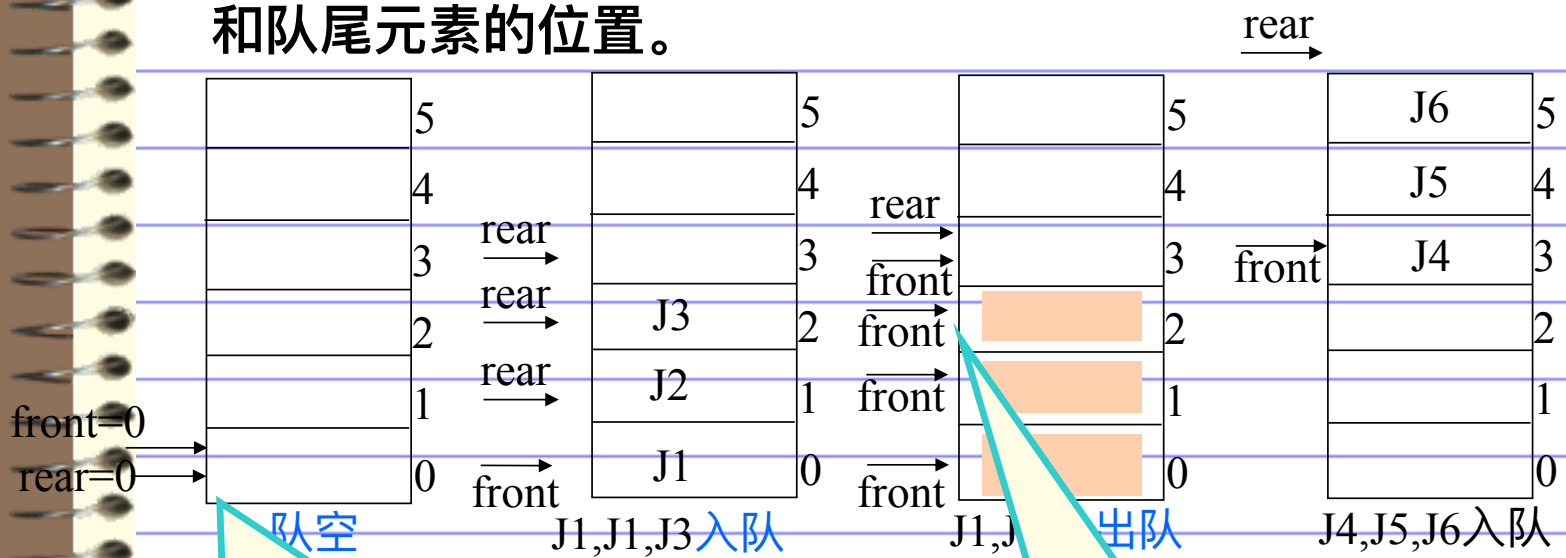
```
//基本操作的算法实现
Status InitQueue (LinkQueue &Q)
{   // 构造一个空队列 Q
    Q.front = Q.rear =
     (QueuePtr)malloc(sizeof(QNode));
    //存储分配失败
    if (! Q.front) exit (OVERFLOW);
    Q.front->next = NULL;
    return OK;
}

Status DestroyQueue (LinkQueue &Q)
{   // 销毁队列 Q
    while (Q.front) {
      Q.rear = Q.front->next;
      free (Q.front);
      Q.front = Q.rear;
    }
    return OK;
}
```

```
Status EnQueue (LinkQueue &Q, QElemType e)
{   // 插入元素 e 为 Q 的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    //存储分配失败
    if (! p)  exit (OVERFLOW);
    p->data = e;     p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}
Status DeQueue (LinkQueue &Q, QElemType &e)
{   // 若队列不空,则删除 Q 的队头元素,用 e
    // 返回其值,并返回 OK;否则返回 ERROR
    if (Q.front == Q.rear) return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next = p->next;
    if (Q.rear == p) Q.rear = Q.front;
    free (p);
    return OK;
}
```

队列的顺序表示

- 队列作为线性表，也可以用顺序存储结构存储。除了可以用一组地址连续的存储单元依次存放从队头到队尾的元素之外，还需附设两个指针front和rear分别指向队头元素和队尾元素的位置。



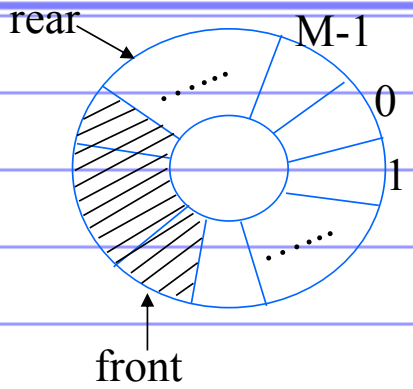
设两个指针front,rear,约定：
rear指示队尾元素的下一个位置；
front指示队头元素
初值front=rear=0

空队列条件：front==rear
入队列：sq[rear++]=x;
出队列：x=sq[front++];

顺序队列存在的问题

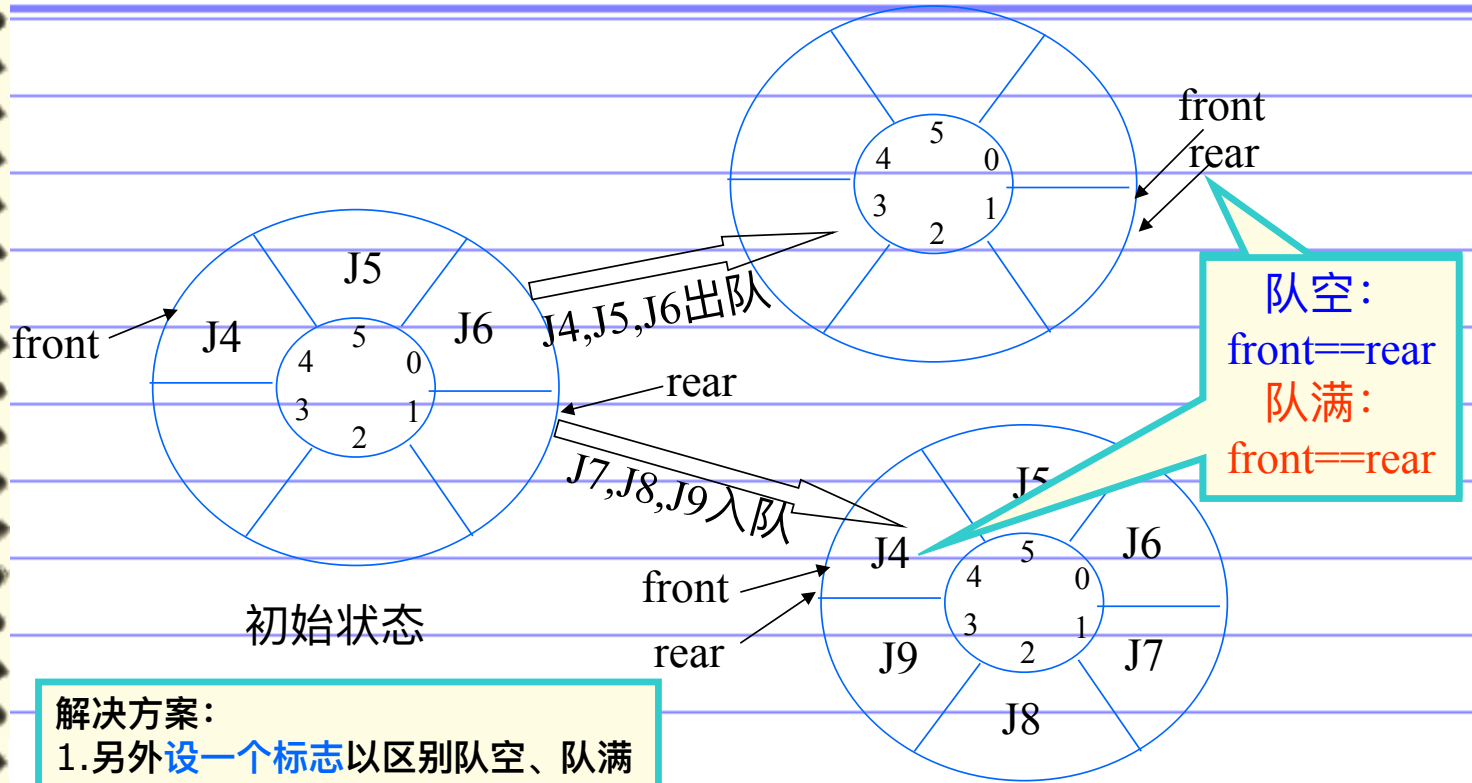
- 设数组维数为M，则-
  - 当front=0,rear=M时，当再次有元素入队时发生溢出——真溢出
  - 当front≠0,rear=M时，当再次有元素入队时发生溢出——假溢出
- 解决方案
  - 队首固定，每次出队剩余元素向下移动——浪费时间
  - 循环队列
    - 基本思想：把队列设想成环形，让sq[0]接在sq[M-1]之后，若rear==M，则令rear=0;
    - 重复利用已经用过的空间

循环队列



- 基本思想：把队列sq设想成环形，让sq[0]接在sq[M-1]之后，若rear+1==M，则令rear=0;
- 实现：利用“模”运算
  - 入队：　rear=(rear+1)%M;　　sq[rear]=x;
  - 出队：　front=(front+1)%M;　　x=sq[front];
- 队满、队空判定条件

循环队列插入与删除



解决方案：

1.另外设一个标志以区别队空、队满

2.少用一个元素空间：

队空：front==rear

队满：(rear+1)%M==front

循环队列的顺序存储

```
#define MAXQSIZE 100
Typedef struct {
   QElemType *base;
   int front;
   int rear;
}SqQueue;
Status  InitQueue(SqQueue &Q) {
   Q.base = (QElemType *)malloc(MAXQSIZE*sizeof(QElemType));
   if(!Q.base) exit(OVERFLOW);
   Q.front = Q.rear = 0;
   return OK;
}
```

循环队列插入与删除

```
Status EnQueue(SqQueue &Q, QElemType e) {
    if((Q.rear+1)%MAXSIZE)== Q.front) return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1)%MAXQSIZE;
    return OK;
} //入队
Status DeQueue(SqQueue &Q, QElemType &e) {
    if(Q.front == Q.rear) return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1)%MAXQSIZE;
    return OK;
} //出队
```