第3章

栈和队列

本章要点

- ◇ 栈
- ◇ 栈的应用举例
- ◇ 队列
- ◇ 队列的应用举例
- ◇ 递归

本章学习目标

- ◇ 理解栈的定义及其基本运算
- ◇ 掌握顺序栈和链栈的各种操作实现
- ◇ 理解队列的定义及其基本运算
- ◇ 掌握循环队列和链队列的各种操作实现
- ◇ 学会利用栈和队列解决一些问题

3.1 栈

栈和队列是在程序设计中广泛使用的两种重要的数据结构。由于从数据结构角度看，栈和队列是操作受限的线性表，因此，也可以将它们称为限定性的线性表结构。

3.1.1 栈的定义与基本操作

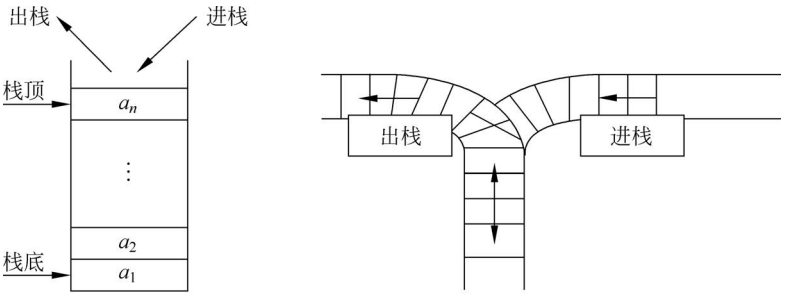
在日常生活中，我们会发现有许多这样的趣事。例如，把许多书籍依次放进一个大小相当的箱子中，当我们在取书时，就得先把后放进里面的书取走，才能拿到先放入的被压在最底层的书；又如一叠洗净的盘子，洗的时候总是将盘子逐个叠放在已洗好的盘子上面，而用的时候则是从上往下逐个取用，即后洗好的盘子比先洗好的盘子先被使用。这种后进先出的结构称为栈。

1. 栈的定义

栈(stack)是一种仅允许在一端进行插入和删除运算的线性表。栈中允许进行插入和删除的那一端，称为**栈顶**(top)。栈顶的第1个元素称为**栈顶元素**。栈中不可以进行插入和删除的那一端(线性表的表头)，称为**栈底**(bottom)。在一个栈中插入新元素，即把新元素放到当前栈顶元素的上面，使其成为新的栈顶元素，这一操作称为**进栈**、**入栈**或**压栈**(push)。从一个栈中删除一个元素，即把栈顶元素删除掉，使其下面的元素成为新的栈顶元素，称为**出栈**或**退栈**(pop)。例如，在栈 $S=(a_1,a_2,\cdots,a_n)$ 中，a_1 称为栈底元素，a_n 称为

栈顶元素。进栈顺序为 $a_1, a_2, \cdots, a_n$，如图 3.1(a)所示，而出栈顺序为 $a_n, a_{n-1}, \cdots, a_2, a_1$。

注意：由于栈的插入和删除操作只能在栈顶一端进行，后进栈的元素必定先出栈，所以栈又称为**后进先出**(Last In First Out)的线性表(简称为 LIFO 结构)。它的这个特点可用图 3.1(b)所示的铁路调度站形象地表示。



(a) 栈的示意图　　(b) 用铁路调度站表示栈的“后进先出”的特点

图 3.1　栈的图示

思考：① 栈是什么？它与一般线性表有何不同？

② 一个栈的输入序列是 12345，若在入栈的过程中允许出栈，则栈的输出序列 43512 有可能实现吗？12345 的输出呢？

讨论：有无通用的判别原则？

有！若输入序列是…，P_j，…，P_k，…，P_i，…($P_j < P_k < P_i$)，则一定不存在输出序列…，P_i，…，P_j，…，P_k，…

2. 栈的基本操作

定义在栈上的基本操作有：

(1) InitStack(S)：构造一个空栈 S。

(2) ClearStack(S)：清除栈 S 中的所有元素。

(3) StackEmpty(S)：判断栈 S 是否为空，若为空，则返回 TRUE；否则返回 FALSE。

(4) GetTop(S)：返回 S 的栈顶元素，但不移动栈顶指针。

(5) Push(S, x)：插入元素 x 作为新的栈顶元素(入栈操作)。

(6) Pop(S)：删除 S 的栈顶元素并返回其值(出栈操作)。

由于栈是运算受限的线性表，因此线性表的存储结构对栈也同样适用。与线性表相似，栈也有两种存储表示方法，即顺序存储和链式存储两种结构。顺序存储的栈称为**顺序栈**，链式存储的栈称为**链栈**。

3.1.2　顺序栈的存储结构和操作的实现

1. 顺序栈存储结构的定义

顺序栈利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素。在 C 语言中，可以用一维数组描述顺序栈中数据元素的存储区域，并预设一个数组的最大空间。栈底

设置在 0 下标端,栈顶随着插入和删除元素而变化,即入栈的动作使地址向上增长(称为“向上增长”的栈),可用一个整型变量 top 来指示栈顶的位置。为此,顺序栈存储结构的描述如下:

```
#define Maxsize 100          /* 设顺序栈的最大长度为 100,可依实现情况而修改 */
typedef int datatype;
typedef struct
{
  datatype stack[Maxsize];
  int top;                                   /* 栈顶指针 */
}SeqStack;                                   /* 顺序栈类型定义 */
SeqStack *S;                                 /* S 为顺序栈类型变量的指针 */
```

由于 C 语言中数组下标是从 0 开始的,即 S -> stack[0]是栈底元素,而栈顶指针 S -> top 是正向增长的,即进栈时栈顶指针 S -> top 加 1,然后把新元素放在 top 所指的空单元内,退栈时 S -> top 减 1,因此 S -> top 等于 -1(或 S -> top 小于 0)表示栈空,S -> top 等于 maxsize-1 表示栈满。由此可知,对顺序栈进行插入和删除运算相当于是在顺序表的表尾进行的,其时间复杂度为 $O(1)$。一个栈的几种状态以及在这些状态下栈顶指针 top 和栈中元素之间的关系如图 3.2 所示。

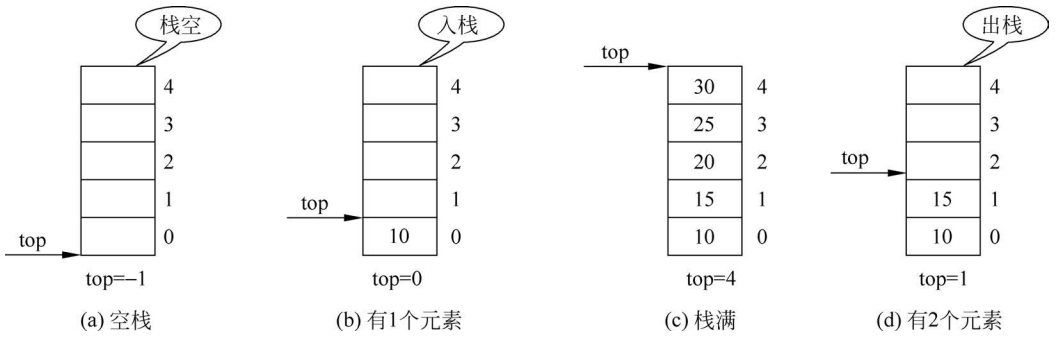


图 3.2 栈顶指针和栈中元素之间的关系

通过分析,我们可以得出以下结论:

(1) 若 top=-1,则表示栈空。

如果使栈顶指针 top 指向待入栈元素的位置,则入栈、出栈时,top 应该如何变化?栈空、栈满的条件又是什么?

(2) 若 top=maxsize-1,则表示栈满。

如果使栈顶指针 top 指向待入栈元素的位置,则入栈、出栈时,top 应该如何变化?栈空、栈满的条件又是什么?

2. 顺序栈的基本操作

由于顺序栈的插入和删除只在栈顶进行,因此顺序栈的基本操作比顺序表简单得多。值得一提的是:在做入栈操作前,首先要判定栈是否满;在做出栈操作前,又得先判定栈是否空。

1) 构造一个空栈

```
SeqStack *InitStack()
{ SeqStack *S;
  S=(SeqStack *)malloc(sizeof(SeqStack));
  if(!S)
    {printf("空间不足");
```

```
      return NULL;}
   else
     {S->top = -1;
      return S;}
}
```

2）取栈顶元素

```
datatype GetTop(SeqStack *S)
  {if (S->top == -1)
     {printf("\n 栈是空的!");
      return FALSE;}
    else
     return S->stack[S->top];
}
```

3）入栈

```
SeqStack *Push(SeqStack *S,datatype x)
{if(S->top == Maxsize-1)
    {printf("\n 栈是满的!");
      return NULL; }
    else
        { S->top++;
          S->stack[S->top] = x;
          return s;}
}
```

4）出栈

```
datatype Pop( SeqStack *S)
  {if(S->top == -1)
   {printf("\nThe sequence stack is empty!");
    return FALSE;}
   S->top--;
   return S->stack[S->top+1];
}
```

5）判别空栈

```
int  StackEmpty(SeqStack  *S)
 {if(S->top == -1)
    return TRUE;
 else
    return FALSE;
}
```

例 3.1　若增加 main 函数以及 display 函数，则可以调试上述各种栈的基本操作算法。

```
#define Maxsize 50
typedef int datatype;
typedef struct
   {datatype stack[Maxsize];
   int top;
   }SeqStack;
void display(SeqStack *S)
 {int t;
  t = S->top;
```

```
  if(S->top == -1)
    printf("the stack is empty!\n");
  else
    while(t!= -1)
      {t--;
       printf("%d->",S->stack[t]);}
      }
  main()
 {int a[6] = {3,7,4,12,31,15},i;
  SeqStack *p;
  p = InitStack();
  for(i = 0;i<6;i++) Push(p,a[i]);
  printf("output the stack values: ");
  display(p);
  printf("\n");
  printf("the stacktop value is: %d\n",GetTop(p));
  Push(p,100);
  printf("output the stack values:");
  display(p);
  printf("\n");
  printf("the stacktop value is: %d\n",GetTop(p));
  Pop(p);Pop(p);
  printf("the stacktop value is: %d\n",GetTop(p));
  printf("Pop the stack value :");
  while(!StackEmpty(p))
  printf("%4d",Pop(p));
  printf("\n");
}
```

运行结果如下：

```
output the stack values: 15->31->12->4->7->3->
the stacktop value is: 15
output the stack values: 100->15->31->12->4->7->3->
the stacktop value is: 100
the stacktop value is: 31
Pop the stack value : 31  12  4  7  3
```

思考：

① 顺序表和顺序栈的操作有何区别？

② 为什么要设计栈，它有何用途？

③ 这里定义的入栈操作是：栈顶指针加 1，然后入栈；当然也可以定义先入栈，然后栈顶指针加 1。同样出栈操作也可以栈顶指针减 1，然后出栈。

讨论：什么是栈的溢出？

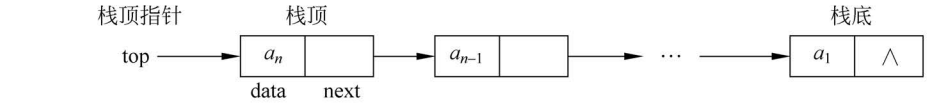
答：对于顺序栈，入栈时必须先判断栈是否满，栈满的条件是 S -> top==maxsize-1。栈满时不能入栈，否则会产生错误，这种现象称为上溢。

出栈时必须先判断栈是否空，栈空的条件是 S -> top==-1。栈空时不能出栈，否则会产生错误，这种现象称为下溢。

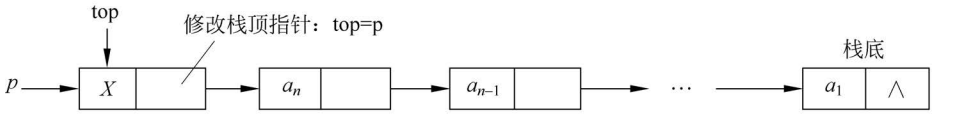
3.1.3 链栈的存储结构和操作的实现

栈的链式存储结构与线性表的链式存储结构相同，是通过由结点构成的单链表实现的。

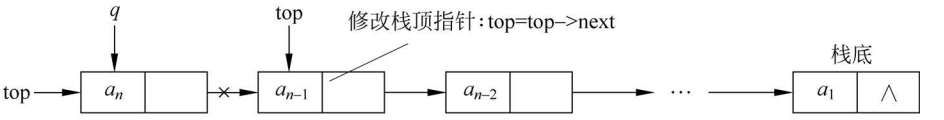
为了操作方便，这里采用没有头结点的单链表。此时栈顶为单链表的第 1 个结点，整个单链表称为**链栈**。链栈的表示如图 3.3(a)所示。



(a) 链栈示意图



(b) 入栈示意图



(c) 出栈示意图

图 3.3　链栈的图示

链栈的类型定义如下：

```
Typedef struct node
{
  datatype data;                          /*数据域*/;
  struct node * next;                     /*指针域*/;
}LinkStack;                               /*链栈结点类型*/
LinkStack   *top;                         /*top 为栈顶指针变量*/
```

top 为栈顶指针，它唯一地确定一个栈。栈空时 top＝NULL。因为链栈是动态分配结点空间的，所以操作时无须考虑上溢问题。由于链栈的入栈、出栈操作限定在栈顶方向进行，其时间复杂度为 $O(1)$，因此没有必要附加一个头结点。

下面是链栈的部分基本操作的实现，其余的操作请读者自行完成。

1）判别空栈

```
int  StackEmpty(LinkStack *top)
{return (top? 0:1);
}
```

2）取栈顶元素

```
datatype GetTop(LinkStack *top)
{
  if(!top) {printf("\n 链栈是空的!"); return FALSE;}
  return (top->data);
}
```

3）入栈

```
LinkStack* Push((LinkStack *top,datatype x)
{
  LinkStack *p;
  p=( Linkstack *)malloc(sizeof(LinkStack));   /*分配空间*/
```

```
    p->data = x;                                    /* 设置新结点的值 */
    p->next = top;                                  /* 将新元素插入栈中 */
    top = p;                                        /* 修改栈顶指针 */
    return top;
    }
```

4) 出栈

```
LinkStack *Pop( LinkStack *top)
{ LinkStack *q;
  if(!top) {printf("\n链栈是空的!");return NULL;}    /* 判栈是否空 */
  q = top;                                          /* 指向被删除的结点 */
  top = top->next;                                  /* 修改栈顶指针 */
  free(q);
  return top;
}
```

读者可以仿照顺序栈的方法,上机调试链栈的各种基本操作的算法。

说明:

① 链栈不必设头结点,因为栈顶(表头)操作频繁。

② 链栈一般不会出现栈满情况,除非空间不足,导致 malloc 分配失败。

③ 链栈的入栈、出栈操作就是栈顶的插入与删除操作,修改指针即可完成。

④ 链栈的优点:可使多个栈共享空间;在栈中元素变化的数量较大,且存在多个栈的情况下,链栈是栈的首选存储方式。

3.2 栈的应用

由于栈的操作具有后进先出的特点,因此栈成为了程序设计中的有用工具。反之,从本节所举例子中可发现,凡问题求解具有后进先出的天然特性,其求解过程中也必须利用栈。

3.2.1 数制转换

例 3.2 将十进制整数转换成二至九的任一进制数输出。由计算机基础知识可知,把一个十进制整数 N 转换成任一种 r 进制数得到的一个 r 进制整数,转换的方法是采用逐次除以基数 r 取余法。

```
8 | 4327      余数
8 | 540  ..... 7
8 | 67   ..... 4
8 | 8    ..... 3
8 | 1    ..... 0
    0    ..... 1
```

图 3.4 十进制数 4327 转换为八进制数的过程

将一个十进制数 4327 转换成八进制数$(10347)_8$,其过程如图 3.4 所示。

在十进制整数 N 转换为 r 进制数的过程中,由低到高依次得到 r 进制数中的每一位数字,而输出时又需要由高到低依次输出每一位,恰好与计算过程相反,输出的过程符合“后进先出”的栈的特性。因此,可在转换过程中每得到一位 r 进制数就进栈保存,转换完毕后依次出栈正好是转换结果。算法思路如下:

(1) 若 $N\neq 0$,则将 $N\%r$ 压入栈 s 中。

(2) 用 N/r 代替 N。

(3) 若 $N>0$,则重复步骤(1)、(2);若 $N=0$,则将栈 a 的内容依次出栈。

下面给出完整的 C 语言程序。

```
#define Maxsize 100
#include<stdio.h>
typedef int datatype;
typedef struct
  {int stack[Maxsize];
   int top;
  }SeqStack;

SeqStack *InitStack()
{ SeqStack *S;
  S=(SeqStack *)malloc(sizeof(SeqStack));
  if(!S)
    {printf("空间不足"); return NULL;}
  else
    {S->top=0;
     return S;
     }
}

SeqStack *push(SeqStack *S,int x)
{  if (S->top==Maxsize)
    {printf("the stack is overflow!\n");
     return NULL;
    }
   else
    {S->stack[S->top]=x;
     S->top++;
     return s;
    }
}

int  StackEmpty(SeqStack  *S)
{ if(S->top==0)
    return 1;
  else
    return 0;
}

int pop(SeqStack *S)
  {int y;
   if(S->top==0)
     {printf("the stack is empty!\n"); return FALSE;}
  else
   {S->top--;
    y=S->stack[S->top];
    return y;
  }
}

void conversion(int N, int r)
{ int x=N,y=r;
```

```
  SeqStack *S;                              /*定义一个顺序栈*/
  s=InitStack();                            /*初始化栈*/
  while(N!=0)                               /*由低到高求出 r 进制数的每一位并入栈*/
       { push(s, N %r );
         N=N/r;
       }
  printf("\n十进制数%d所对应的%d进制数是:"x,y);
  while(!StackEmpty(s))                     /*由高到低输出每一位 r 进制数*/
    printf("%d",pop(s));
  printf("\n");
}

main()
{int n,r;
 printf("请输入任意一个十进制整数及其所需转换的二至九间的任一进制数:\n");
 scanf("%d%d",&n,&r);
 conversion(n,r);
}
```

3.2.2 括号匹配问题

例 3.3 设一个表达式中可以包含 3 种括号：小括号、中括号和大括号，各种括号之间允许任意嵌套，如小括号内可以嵌套中括号、大括号，但是不能交叉。举例如下：

([]{}) 正确的

([()]) 正确的

{([])} 正确的

{[(])} 不正确的

{(}[]) 不正确的

如何检验一个表达式的括号是否匹配呢？大家知道，当自左向右扫描一个表达式时，凡是遇到一个左括号都期待有一个右括号与之匹配。

按照括号正确匹配的规则，在自左向右扫描一个表达式时，后遇到的左括号比先遇到的左括号更加期待有一个右括号与之匹配。因为可能会连续遇到多个左括号，且它们都期待寻求匹配的右括号，所以必须将遇到的左括号存放好。又因为后遇到的左括号的期待程度高于先前遇到的左括号的期待程度，所以应该将所遇到的左括号存放于一个栈中。这样，当遇到一个右括号时，就查看栈顶结点，如果它们匹配，则删除栈顶结点；如果不匹配，则说明表达式中括号是不匹配的。如果扫描完整个表达式后，这个栈是空的，则说明表达式中的括号是匹配的，否则说明表达式中的括号是不匹配的。算法如下：

```
int match(char c[])
 { int i=0;
   SeqStack *S;
   S=InitStack();
   while(c[i]!='#')
   {
     switch(c[i])
     {
       case '{':
       case '[':
```

```
        case '(': Push(S,c[i]);break;
        case '}': if(!StackEmpty(s)&& GetTop(S) == '{')
                    {Pop(S);break;}
                  else return FALSE;
        case ']': if(!StackEmpty(s)&& GetTop(S) == '['  )
                    {Pop(S);break;}
                  else return FALSE;
        case ')': if(!StackEmpty(s)&& GetTop(S) == '(' )
                    {Pop(S);break;}
                  else return FALSE;
      }
     i++;
    }
    return (StackEmpty(S));      /* 栈空则匹配,否则不匹配 */
}
```

3.2.3　子程序的调用

例 3.4　在计算机程序中,程序调用与返回处理是利用栈来实现的。某个程序要去调用子程序(或子函数)之前,先将该调用指令的下一条指令的地址保存到栈中,然后才转而去执行子程序(或子函数),当子程序(或子函数)执行完后要从栈中取出返回地址,从断点处继续往下执行。如图 3.5 所示,主程序中的 r 处调用子程序 1,先将该断点地址 r 入栈;子程序 1 中的 s 处调用子程序 2,首先又将 s 压入栈;子程序 2 中的 t 处调用子程序 3,又得先将 t 入栈保存……当子程序 3 调用结束时,就从栈中弹出返回地址 t,回到子程序 2。以此类推,再从栈中弹出返回地址 s,从子程序 2 返回子程序 1,然后继续从栈中弹出返回地址 r,从子程序 1 返回主程序,直到整个程序结束。

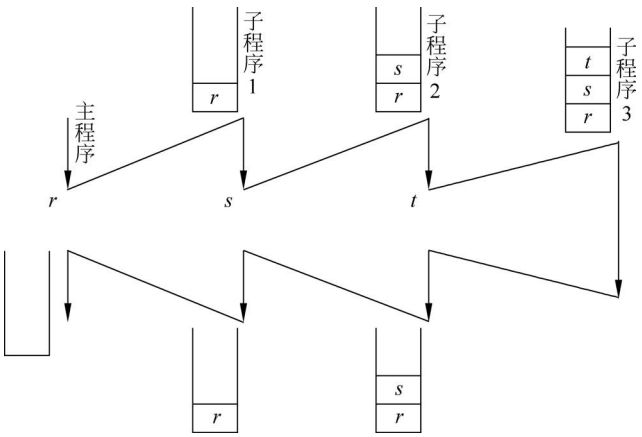


图 3.5　栈在子程序嵌套调用中的应用

栈在程序设计中的另一个重要应用就是递归的实现。一个递归函数的运行过程类似于多个函数的嵌套调用,只是主调函数和被调函数都是同一个函数。为了保证递归函数的正确运行,系统需要设立一个“递归工作栈”,在整个递归函数运行期间都要使用它。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。

思考：要求用递归的方式来求解某个数 n(不妨设 $n=4$)的阶乘，请描述该递归工作栈的数据如何变化。

3.2.4 利用一个顺序栈逆置一个带头结点的单链表

例 3.5 已知 head 是带头结点的单链表$(a_1,a_2,\cdots,a_n)$(其中 $n\geqslant 0$)，有关说明如下：

```
typedef int datatype;
#include <stdio.h>
typedef struct node
      {datatype data;
       struct node *next;
      }linklist;
linklist *head;
```

请设计一个算法，利用一个顺序栈使上述单链表实现逆置，即利用一个顺序栈将单链表$(a_1,a_2,\cdots,a_n)$(其中 $n\geqslant 0$)逆置为$(a_n,a_{n-1},\cdots,a_1)$，如图 3.6 所示。

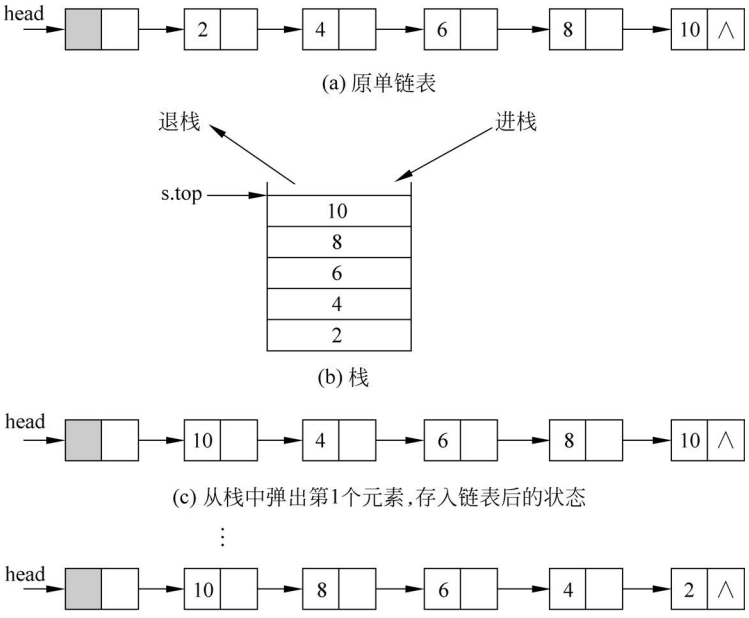


图 3.6 利用一个栈逆置单链表

解题思路(用顺序栈实现)：

(1) 建立一个带头结点的单链表 head。

(2) 输出该单链表。

(3) 建立一个空栈 s(顺序栈)。

(4) 依次将单链表的数据入栈。

(5) 依次将单链表的数据出栈，并逐个将出栈的数据存入单链表的数据域(自前向后)。

(6) 输出单链表。

程序如下(采用顺序栈实现)：

```
#include <stdio.h>                          /*利用顺序栈逆置单链表*/
#include <iostream.h>
#include <malloc.h>
#define maxsize 100                         /*栈的最大元素数为 100*/
typedef int datatype;
typedef struct node                         /*定义单链表结点类型*/
  {datatype data;
   struct node *next;
  }linklist;
linklist *head;                             /*定义单链表的头指针*/
typedef struct                              /*定义顺序栈*/
  {datatype d[maxsize];
   int top;
  }seqstack;
seqstack s;                                 /*定义顺序栈 s,s 是结构体变量,且 s 是全局变量*/
linklist *creatlist( )                      /*建立单链表*/
 {linklist *p,*q;
  int n=0;
  p=q=(struct node *)malloc(sizeof(linklist));
  head=p;
  p->next=0;                                /*头结点的数据域不存放任何数据*/
  p=(struct node *)malloc(sizeof(linklist));
  scanf("%d",&p->data);
  while(p->data!=-1)                        /*输入 -1 表示链表结束*/
      {n=n+1;
       q->next=p;
       q=p;
       p=(struct node *)malloc(sizeof(linklist));
       scanf("%d",&p->data);
      }
  q->next=0;
  return (head);
 }

void print(linklist *head)                  /*输出单链表*/
  {linklist *p;
  p=head->next;
  if (p==0) printf("This is an empty list.\n");
    else
        {do  {printf("%6d",p->data);
              p=p->next;
             }while(p!=0);
        printf("\n");
        }
  }

seqstack initstack()                        /*构造一个空栈 s*/
{s.top=-1;
 return s;                                  /*返回结构体变量 s 的首址*/
}
int push(seqstack *s,datatype x)            /*入栈,此处 s 是指向顺序栈的指针*/
{if((*s).top==maxsize-1)                    /*(*s).top 即为 s->top*/
  {printf("栈已满,不能入栈!\n");
   return 0;
```

```
  }
else
 {
   (*s).top++;                          /*栈顶指针上移*/
   (*s).d[(*s).top]=x;                  /*将 x 存入栈中*/
 }
}

datatype pop(seqstack *s)               /*出栈,此处 s 是指向顺序栈的指针*/
{datatype y;
 if((*s).top==-1)
   {printf("栈为空,无法出栈! \n");
    return 0;
 }
else {y=(*s).d[(*s).top];               /*栈顶元素出栈,存入 y 中*/
      (*s).top--;                       /*栈顶指针下移*/
      return y;
      }
}

int stackempty(seqstack s)              /*判栈空,此处 s 是结构体变量*/
{
 return s.top==-1;
     }

int stackfull(seqstack s)               /*判栈满,此处 s 是结构体变量*/
{
 return s.top==maxsize-1;
}

linklist *backlinklist(linklist *head) /*利用顺序栈 s 逆置单链表 head*/
  {linklist *p;
   p=head->next;
   initstack();
   while(p)
    {push(&s,p->data);                  /*单链表的数据依次入栈 s*/
     p=p->next;
    }
  p=head->next;
  while(!stackempty(s))
    {p->data=pop(&s);                   /*数据出栈依次存入单链表的数据域*/
     p=p->next;
    }
   return (head);
  }

void main()
{
 linklist *head;
 head=createlist();
 print(head);
 head=backlinklist(head);
 print(head);
}
```

此算法的时间复杂度为 $O(n)$,算法的空间复杂度也是 $O(n)$。

思考：在上述问题中，能否将链表结点中的地址入栈，来实现一个单链表的逆置？如果要求使用链栈来实现单链表的逆置，那么程序应该怎么修改？

3.3 队 列

3.3.1 队列的定义与基本操作

1. 队列的定义

队列(queue)也是线性表的一种特例，它是一种限定在表的一端进行插入而在另一端进行删除的线性表。与栈相反，队列遵循先进先出(First In First Out，FIFO)的原则。允许删除的一端，称为队头(front)，允许插入的一端，称为队尾(rear)。向队列中插入新元素，称为入队(或进队)，新元素入队后，就成为新的队尾元素；从队列中删除元素，称为出队(或退队)，元素离队后，其后继元素就成为新的队头元素。

队列的例子在日常生活中随处可见，它反映了“先来先服务”的原则，例如，排队购物、食堂买饭等，新到的人排在队尾(入队)，站在队头的人被服务完后离开(出队)，当最后一个人离开后，队列为空。在队列 $Q=(a_1,a_2,\cdots,a_n)$ 中，a_1 称为队头元素，a_n 称为队尾元素。

队列中的元素是按照 $a_1,a_2,\cdots,a_n$ 的顺序进入的，退出队列也只能按照这个次序依次退出，也就是说，只有在 $a_1,a_2,\cdots,a_{n-1}$ 都离开队列之后 a_n 才能退出队列，如图 3.7 所示。这和日常生活中的排队是一致的，最早进入队列的元素最早离开。

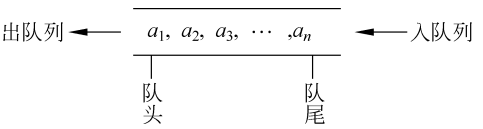


图 3.7 队列示意图

注意：只能从队尾插入元素，从队头删除元素。

队列在程序设计中经常使用，一个最典型的例子就是操作系统中的作业排队。在允许多道程序运行的计算机系统中，同时有几个作业运行。如果运行的结果都需要经过通道输出，那么就要按请求输出的先后次序排队。每当通道传输完毕可以接受新的输出任务时，队头的作业先从队列中退出做输出操作；凡是申请输出的作业都从队尾进入队列。

讨论：为什么要设计队列？它有什么独特的用途？

① 离散事件的模拟(模拟事件发生的先后顺序，如 CPU 芯片中的指令译码队列)。

② 操作系统中的作业调度(一个 CPU 执行多个作业)。

③ 简化程序设计。

2. 队列的基本操作

队列的基本操作主要有如下几种：

(1) InitQueue(Q)：构造一个空队列 Q。

(2) QueueEmpty(Q)：判断队列是否为空。

(3) QueueLength(Q)：求队列的长度。

(4) GetHead(Q)：返回 Q 的队头元素，不改变队列状态。

(5) EnQueue(Q,x)：插入元素 x 作为 Q 的新的队尾元素。

(6) DeQueue(Q)：删除 Q 的队头元素。

(7) ClearQueue(Q)：清除队列 Q 中的所有元素。

与线性表类似，队列也有两种存储表示，即顺序队列和链队列。由于链队列相对比较简单，因此，我们先介绍链队列。

3.3.2 链队列的存储结构和操作的实现

1. 链队列的定义

链队列就是用链表表示的队列(如图 3.8 所示)，它是限制仅在表头进行删除和在表尾进行插入的单链表。一个链队列显然需要两个分别指示队头和队尾的指针(分别称为头指针和尾指针)才能唯一确定。与线性表的单链表相似，为了操作方便，给链队列添加一个头结点，并令头指针指向头结点，尾指针指向真正的队尾元素结点。因此，判定链队列为空的条件是队头指针与队尾指针均指向头结点。

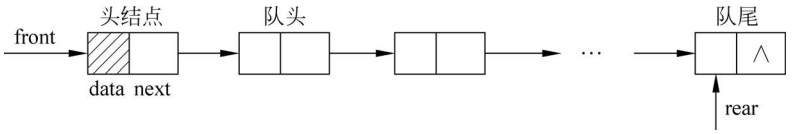


图 3.8 链队列示意图

结点类型定义：

```
typedef Struct Qnode
  {
    datatype  data;                    /* 数据域 */
    Struct  Qnode  * next;             /* 指针域 */
  }Qnode;
```

链队列类型定义：

```
typedef  struct
  {
   Qnode  * front;                     /* 队头指针 */
   Qnode  * rear;                      /* 队尾指针 */
  }LinkQueue;
```

链队列的入队和出队操作即为单链表的插入和删除操作的特殊情况，只是尚需修改尾指针或头结点的指针。各种操作的指针修改情况如图 3.9 所示。

2. 链队列的基本操作

1) 构造一个空队列

```
LinkQueue * InitQueue()                /* 建立一个空的链队列 */
  { LinkQueue  * q;
    Qnode * p;
    q = (LinkQueue * )malloc(sizeof(LinkQueue));  /* 为队列头指针分配空间 */
    p = (Qnode * )malloc(sizeof(Qnode)); /* 为头结点分配空间 */
    p -> next = NULL;                  /* 置头结点的指针域为空 */
    q -> front = q -> rear = p;              /* 队首指针、队尾指针均指向头结点 */
  return q;
}
```

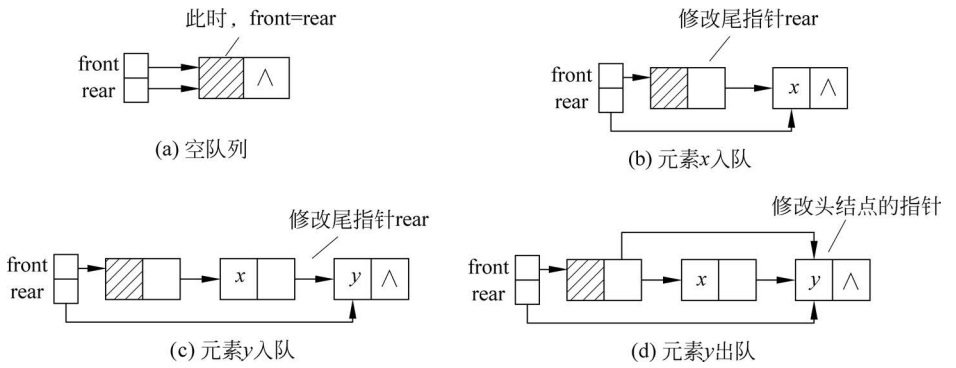


图 3.9　队列运算指针变化情况示意图

2）取队头元素

```
datatype GetHead(LinkQueue *Q)
{ if(Q->.front->next == Q->rear)              /*判断队列是否为空*/
      {printf("\n链队列为空!");return FALSE;}
  return Q->front->next->data;                /*返回队头元素*/
}
```

3）入队

```
void EnQueue(LinkQueue *Q,datatype x)
  { Qnode *p;
    p=(Qnode*)malloc(sizeof(Qnode));          /*为新结点分配空间*/
    p->data=x; p->next=NULL;                  /*设置新结点的值*/
    Q->rear->next=p;                          /*将值为x的元素入队*/
    Q->rear=p;                                /*修改队尾指针*/
}
```

4）出队

```
datatype DeQueue(LinkQueue *Q)
 { Qnode *p;
   datatype x;
   if (Q->front==Q->rear)                     /*判断队列是否为空*/
        {printf("队列为空,无法删除!"); return FALSE;}
   p=Q->front->next;                          /*置p指向队头元素*/
   x=p->data;                                 /*将队头元素值赋给x*/
   Q->front->next=p->next;                    /*出队*/
   if(Q->rear==p) Q->rear=Q->front;           /*若队列为空,则修改队尾指针指向头结点*/
   free(p);                                   /*释放空间*/
   return x;                                  /*返回出队元素的值*/
}
```

注意：删除队头元素算法中存在特殊情况。一般情况下，删除队头元素时仅需修改头结点中的指针，但当队列中最后一个元素被删除后，队列尾指针也丢失了，因此需对队尾指针重新赋值(指向头结点)。

思考：链队列会上溢吗？

3.3.3 顺序队列的存储结构和操作的实现

1. 顺序队列的定义

队列的顺序存储结构称为**顺序队列**。和顺序栈相类似,在队列 Q 的顺序存储结构中,用一组地址连续的存储单元依次存放从队列头到队列尾的元素。但它的顺序存储结构比栈的顺序存储结构稍微复杂一些,除了定义一个一维数组外,还需附设两个指针 front 和 rear 分别指示当前队头元素和队尾元素在数组中的位置。

为了描述方便,这里约定,初始化建空队列时,令 front=rear=0,入队操作的过程为:把新插入的元素放在 rear 所指的空单元内,成为新的队尾元素,尾指针 rear 增 1;出队操作的过程为:每当删除一个队头元素时,头指针 front 增 1。因此,在非空队列中,头指针始终指向队头元素,而尾指针始终指向队尾元素的下一个位置。头、尾指针和队列中元素之间的关系如图 3.10 所示。

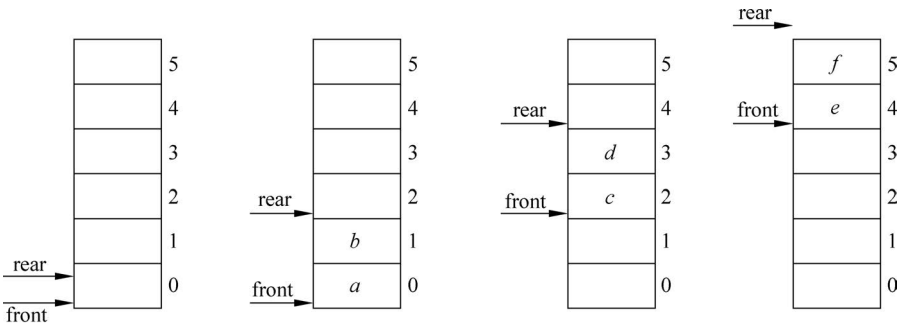


图 3.10 头、尾指针和队列中元素之间的关系

在进行入队操作时会出现如图 3.10(d)所示的情况,由于在进行入队和出队操作时总是使 front 和 rear 的值增加,因此,当进行了若干次入队和出队操作后,队尾指针到了最后,无法插入了,但队列并没有满,即元素的个数少于队列满时的个数 maxsize,这种现象称为**假溢出**。

避免假溢出有两种办法:

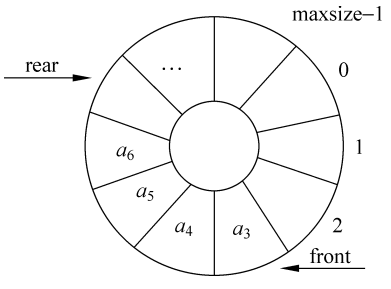


图 3.11 循环队列示意图

(1) 像日常生活中的排队一样,每次一个元素出队,将整个队列向前移动一个位置。

(2) 较为巧妙的办法是,将顺序队列的数据区 data[0～maxsize－1]看成一个首尾相接的圆环,当存到 maxsize－1 时,下一个“地址”就翻转成 0,使 data[0]接在 data[maxsize－1]之后,且头尾指针的关系不变,这种队列称为**循环队列**,如图 3.11 所示。在这里采用循环队列解决假溢出现象。

循环队列的类型定义如下:

```
#define maxsize 100                    /*最大队列长度  */
typedef struct
  {
```

```
    datatype data[MAXSIZE];              /* 存储队列的数据空间 */
    int front;          /* 队头指针,若队列不空,则指向队头元素 */
    int rear;           /* 队尾指针,若队列不空,则指向队尾元素的下一个位置 */
}SqQueue;
```

2. 循环队列的特点

通过对图 3.12 所示的循环队列的几种状态进行分析,可以知道:

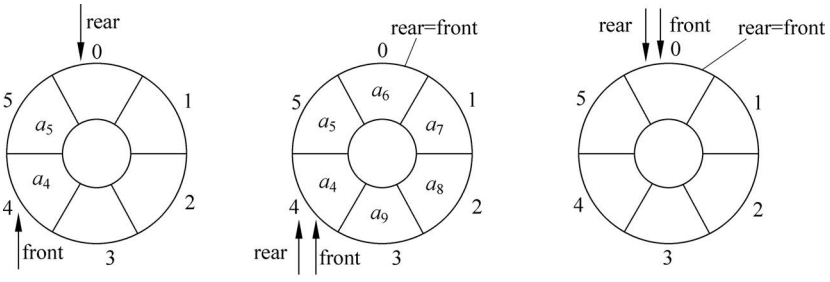
(1) 在对循环队列做入队操作时,尾指针 rear 加 1,但当尾指针指向数组空间的最后一个位置 maxsize 时,若队头元素的前面仍存在空闲的位置,则表明队列未满,下一个存储位置应是下标为 0 的空闲位置,此时应将尾指针置为 0,通过语句 rear=(rear+1)%maxsize 就能实现此操作。这样存储队列的数组就变为首尾相接的一个环,即为循环队列。

(2) 在出队时,队头指针也必须采用取模运算,即 front=(front+1) %maxsize,才能够实现存储空间的首尾相接。

(3) 由于入队时尾指针向前追赶头指针,出队时头指针向前追赶尾指针,故队空和队满时头尾指针均相等。因此,无法通过 front=rear 来判断队列是“空”还是“满”,如图 3.12 所示。对于这个问题有两种处理方法:一是另设一个标志位以区别队列的“空”和“满”;二是少用一个元素的空间,约定以“队头指针在队尾指针的下一位置(指环状的下一位置)上”作为队列“满”的标志,即若数组的大小是 maxsize,则该数组所表示的循环队列最多允许存储 maxsize−1 个结点(注意:rear 所指的单元始终为空,当然也可以使 front 始终指向队头元素的前一个空元素,rear 始终指向队尾元素),如图 3.13 所示。这样,可以得出以下结论:

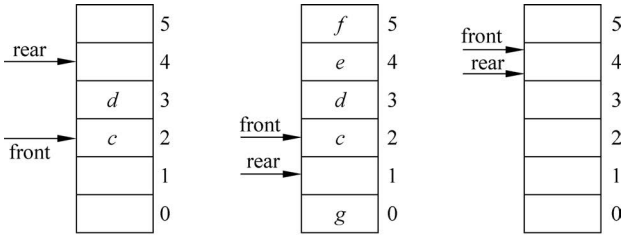
循环队列满的条件:(rear+1)%maxsize==front。

循环队列空的条件:rear==front。



(a) 一般情况　(b) 队满　(c) 队空

图 3.12　循环队列的几种状态表示



(a) 正常情况　(b) 队满　(c) 队空

图 3.13　循环队列操作示意图

3. 循环队列的基本操作

1) 构造空队列

```
SqQueue * InitQueue()
  { SqQueue  * q;
    q = (SqQueue * )malloc(sizeof(SqQueue));      /* 开辟一个足够大的存储队列空间 */
    q -> front = q -> rear = 0;                   /* 将队列头尾指针置为零 */
    return q;                                     /* 返回队列的首地址 */
  }
```

2) 判断队空

```
int QueueEmpty(SqQueue * q)
  {return(q -> front == q -> rear);               /* 如果队列为空返回 1,否则返回 0 */
  }
```

3) 入队

```
int EnQueue(SqQueue * q, datatype x)
  { if((q -> rear + 1) % MAXSIZE == q -> front)   /* 判断队列是否满 */
      {printf("\n 循环队列满!");return FALSE;}   /* 若队列满,则终止 */
    q -> data[q -> rear] = x;                     /* 将元素 x 入队 */
    q -> rear = (q -> rear + 1) % MAXSIZE;        /* 修改队尾指针 */
    return TRUE;
  }
```

4) 出队

```
datatype DeQueue(SqQueue * q)
  { datatype x;
    if (q -> front == q -> rear)                  /* 判断队列是否空 */
      {printf("\n 循环队列空! 不能做删除操作!");
      return FALSE;}                              /* 若队列空,则终止 */
    x = q -> data[ q -> front ];                  /* 将队头元素出队并赋给变量 x */
    q -> front = (q -> front + 1) % MAXSIZE;      /* 修改队列头指针 */
    return x;                                     /* 将被删除元素返回 */
  }
```

例 3.6 若增加 main 函数以及 display 函数,则可以调试上述各种队列的基本操作算法。

```
#define MAXSIZE 20
typedef int datatype;
typedef struct{
    datatype data[MAXSIZE];
    int front;
    int rear;
}SqQueue;

void display(SqQueue * q)                         /* 显示队列中元素的值 */
  { int s;
    s = q -> front;                               /* 利用工作指针 s 来读取队头元素的值 */
    if (q -> front == q -> rear)
      printf("the sqQueue is empty!");            /* 队空 */
    else
     while(s != q -> rear)                        /* 队不空 */
        {printf(" -> %d", q -> data[s]);
```

```
            s=(s+1)%MAXSIZE;                    /*移动工作指针s,准备读取下一个元素*/
          }
  printf("\n");}
}

main()
  {int a[6]={3,7,4,12,31,15},i;
   SqQueue *p;
   p=InitQueue();                               /*初始化一个空队列*/
   for(i=0;i<6;i++) EnQueue(p,a[i]);            /*a[i]入队*/
   printf("output the queue values: ");
   display(p);                                  /*显示队列中的所有元素*/
   printf("\n");
   EnQueue(p,100);EnQueue(p,200);               /*100和200分别入队*/
   printf("output the queue values: ");
   display(p);                                  /*显示队列中的所有元素*/
   printf("\n");
   DeQueue(p);DeQueue(p);                       /*将两个元素出队*/
   printf("output the queue values: ");
   while(!QueueEmpty(p))
    printf("4d",DeQueue(p));
   printf("\n");
}
```

得到的结果如下：

```
output the queue values: ->3->7->4->12->31->15
output the queue values: ->3->7->4->12->31->15->100->200
output the queue values:    4   12   31   15 100 200
```

3.4 队列的应用

队列在算法设计中的应用是非常广泛的。例如，在计算机科学领域中，解决主机与外部设备之间速度不匹配的问题，解决由多用户引起的资源竞争等诸多问题，都需要利用队列来处理。又如，后续内容将会用到的**优先队列**（每个元素都带有一个优先级别，每个元素在队列中的位置是按照其优先级高低来调整的，无论是做插入操作还是删除操作，都确保优先级最高的元素被调整到队首），在操作系统的各种调度算法中应用广泛。在应用程序中，队列通常用来模拟排队情景。

3.4.1 打印杨辉三角形

例3.7 打印杨辉三角形是一个初等数学问题。系数表中的第 i 行有 $i+1$ 个数，除了第1个和最后一个数为1外，其余的数则为上一行中位于其左、右的两数之和，如图3.14所示。

解决此问题的方法很多，如采用一个二维数组。更为直接的方法是用两个一维数组，其中一个存放已经计算得到的第 i 行的值，在输出第 i 行的值的同时计算出第 $i+1$ 行的值，如此写出的算法虽然结构清晰，但需要两个辅助空间，并且这两个数组在计算过程中需相互交换。只用一个数组的空间也可以，但整个算法就不是很清晰了。在此引入“循环队列”，就

可以省去一个数组的辅助空间,而且可以利用队列的操作特点,使程序结构变得清晰。

该算法的基本思想是:如果要计算并输出二项系数表(杨辉三角形)的前 n 行的值,则所设循环队列的最大空间应为 $n+2$。假设队列中已存有第 i 行的值,为了计算方便,在两行之间均加一个 0 作为行间的分隔符,则在计算第 $i+1$ 行之前,头指针正指向第 i 行的 0,而尾元素为第 $i+1$ 行的 0。由此,从左至右输出第 i 行的值,并将计算所得的第 $i+1$ 行的值插入队列。第 i 行元素与第 $i+1$ 行元素的关系如图 3.15 所示。

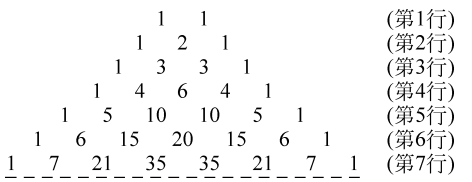


图 3.14 杨辉三角形

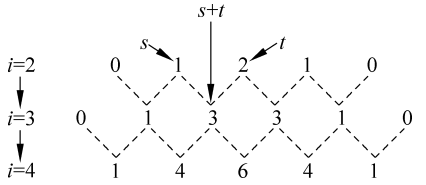


图 3.15 第 $i+1$ 行元素值与第 i 行元素间的关系示意图

假设 $n=4$,$i=3$,则输出第 3 行元素并求解第 4 行元素值的循环执行过程中队列的变化状态如图 3.16 所示。

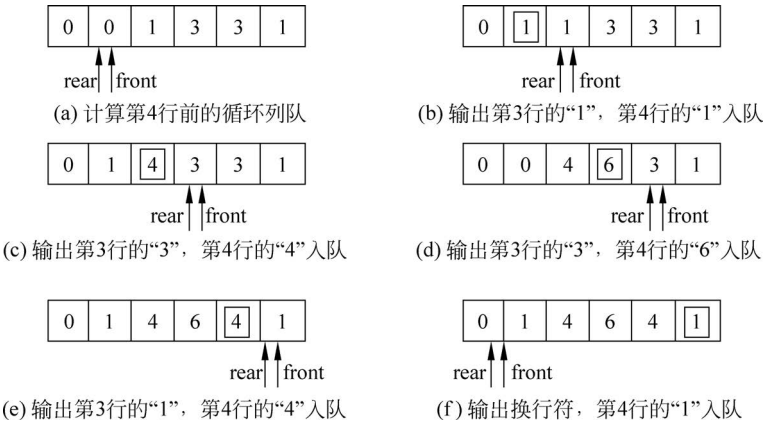


图 3.16 计算二项式系数第 4 行的队列变化状况

输出 $n\leqslant 7$ 时的杨辉三角形的 C 语言程序如下:

```
#define MAXSIZE 10                          /*定义队列的最大长度*/
#include<stdio.h>
typedef int datatype;
typedef struct
{   int data[MAXSIZE];
    int front;
    int rear;
}SqQueue;

SqQueue *InitQueue()                        /*队列的初始化*/
{ SqQueue   *q;
  q=(SqQueue*)malloc(sizeof(SqQueue));
  q->front=q->rear=0;
  return q;
}
```

```
void EnQueue (SqQueue * q, datatype x)     /* 入队 */
{ if((q->rear + 1) % MAXSIZE == q->front)
    {printf("\n 顺序循环队列是满的!");exit(1);}
  q->data[q->rear] = x;
  q->rear = (q->rear + 1) % MAXSIZE;
}

datatype DeQueue (SqQueue * q)             /* 出队 */
{ datatype x;
  if (q->front == q->rear)
    { printf("\n 顺序队列是空的! 不能做删除操作!"); exit(1);}
  x = q->data[q->front];
  q->front = (q->front + 1) % MAXSIZE;
  return x;
}

int QueueEmpty(SqQueue * q)                /* 判断队空 */
{ return(q->front == q->rear);
}

int GetHead(SqQueue * q)                   /* 取队头元素 */
{int e;
  if (q->front == q->rear)
      e = 0;
  else
    e = q->data[q->front];
  return e;
}

void YangHui( int n )                      /* 打印杨辉三角形的前 n 行 */
{ SqQueue  * q;
  int i, j,s,t;
  for(i = 1;i <= n;i++)
  printf("  ");
  printf("1\n");                           /* 在中心位置输出杨辉三角最顶端的 1 */
  q = InitQueue();                         /* 设置容量为 n + 2 的空队列 */
  EnQueue(q,0);                            /* 添加行分隔符 */
  EnQueue(q,1);EnQueue(q,1);               /* 第 1 行的值入队 */
  for(j = 1;j < n;j++)                     /* 利用循环队列输出前 n - 1 行的值 */
    {for(i = 1;i <= n - j;i++)             /* 在输出第 j 行的首元素之间输出 n - j 个空格 */
    printf("  ");
    EnQueue(q,0);                          /* 行分隔符 0 入队 */
   do                                      /* 输出第 j 行并计算第 j + 1 行 */
    {s = DeQueue(q);                       /* 删除队头元素并赋给 s */
     t = GetHead(q);                       /* 取队头元素给 t */
     if(t) printf(" % 5d",t);              /* 若不到行分隔符则输出 t,再输出一个空格 */
       else printf("\n");                  /* 否则输出一个换行符 */
     EnQueue(q,s + t);                     /* 将第 j + 1 行的对应元素 s + t 入队 */
    }while(t!= 0);
  }
DeQueue(q);                                /* 删除第 n 行前的行分隔符 */
printf(" % 3d",DeQueue(q));                /* 输出第 n 行的第 1 个元素 */
while(!QueueEmpty(q))                      /* 输出第 n 行的其余元素 */
    {t = DeQueue(q);
    printf(" % 5d",t);
    }
```

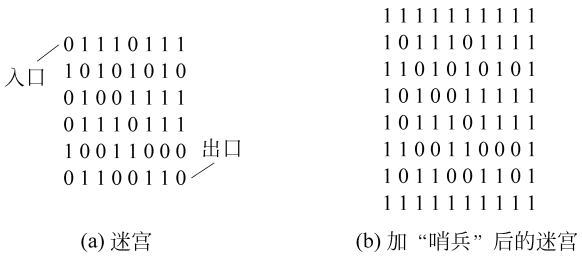
```
}

main()
{int n;
  printf("\n 请输入杨辉三角形的行数:\n");
  scanf(" %d",&n);
  YangHui(n);
}
```

思考：欲输出超过 7 行的杨辉三角形时，应如何修改此程序？

3.4.2 迷宫问题：寻找一条从迷宫入口到出口的最短路径

迷宫问题是实验心理学的一个经典问题，心理学家把一只老鼠从一个无顶盖的迷宫入口处赶进迷宫，在迷宫的出口处设置了一块奶酪，吸引老鼠在迷宫中寻找通路以到达出口。对同一只老鼠重复进行上述实验，一直到老鼠从入口到出口，而不走错一步。老鼠经多次实验终于寻找到走通迷宫的路线。

用计算机来处理迷宫问题的实质是：求出一条从入口到出口的通路，或者得出没有通路的结论。通常采用一种称为回溯法的方法，即不断试探且及时纠正错误的搜索方法，这需要借助“栈”来实现。回溯法在许多书中都有介绍，在此不再赘述。迷宫如图 3.17 所示。



(a) 迷宫　　(b) 加“哨兵”后的迷宫

图 3.17　迷宫示意图

如果在一般走迷宫的方法上，更进一步要求不论试探方位如何，找出一条最短路径，那么又该如何解决呢？其算法的基本思想是：从迷宫的入口[1][1]出发，向四周搜索，记下所有一步能到达的坐标点；然后依次从每一点出发，向四周搜索，记下所有从入口点出发，经过两步可以到达的坐标点。依次进行下去，一直到达迷宫的出口处[m][n]，然后从出口处沿搜索路径回溯直到入口点，这样就找到了从入口到出口的一条最短路径。

我们可以使用数据结构 maze[1..m,1..n]表示迷宫，为了算法方便，在四周加上“哨兵'1'”即变为数组 maze[1..m+1,1..n+1]，如图 3.17(b)所示。用数组 move[8]中的两个域 dx、dy 分别表示 X、Y 方向的移动增量，其值如表 3.1 所示。

表 3.1　X、Y 方向的移动增量表

方向	下标	dx	dy	方向	下标	dx	dy
北	0	−1	0	南	4	1	0
东	1	−1	1	西南	5	1	−1
东	2	0	1	西	6	0	−1
东南	3	1	1	西北	7	−1	−1

由于先到达的点要先向下搜索，故引进一个“先进先出”数据结构——队列——来保存已到达的点的坐标。到达迷宫的出口点(m,n)后，为了能够从出口点沿搜索路径回溯直至入口，对于每一点，在记下点的坐标的同时，还要记下到达该点的前趋点，因此，需用一个结构数组 sq[num]作为队列的存储空间。因为迷宫中每个点至多被访问一次，所以 num 至多等于$m \times n$。sq 的每一个结点有 3 个域，即 x、y 和 pre，其中 x、y 分别为所到达的点的坐标，pre 为前趋点在 sq 中的下标。除 sq 外，还有队头、队尾指针 front 和 rear 用来指向队头和队尾元素。

初始状态是，队列中只有一个元素 sq[0]，记录的是入口点的坐标(1,1)，因为该点是出发点，没有前趋点，所以 pre 域为－1，队头指针 front 和队尾指针 rear 均指向它(sq[0])。此后搜索时都是以 front 所指点为搜索的出发点，当搜索到一个可到达的点时，就将该点的坐标及 front 所指点的位置入队，不但记下到达点的坐标，还记下它的前趋点的下标。若 front 所指点的 8 个方向搜索完毕，则出队，继续对下一点进行搜索。搜索过程中若遇到出口点，则表示成功，搜索结束，打印出迷宫的最短路径，算法结束；如果当前队空，即表示没有搜索点了，说明迷宫没有通路，算法也结束。

程序如下：

```
#include<stdio.h>
#define m 10
#define n 15
#define NUM m*n
typedef struct
  { int x,y;                                  /*x,y为到达点的坐标*/
    int pre;                                  /*pre为(x,y)的前趋点在数组sq中的下标*/
  }sqtype;
int maze[m+1][n+1];
typedef struct
  {int dx;
   int dy;
  }moved;

void shortpath(int maze[m][n],moved move[8])
{ sqtype sq[NUM];
  int front,rear;
  int x,y,i,j,v;
  front=rear=0;
  sq[0].x=1; sq[0].y=1; sq[0].pre=-1;   /*选(1,1)点为入口点入队*/
  maze[1][1]=-1;          /*表示该点搜索过了，所以置成-1。该点是入口点，原值为0*/
  while (front<=rear)                         /*队列不空*/
    {x=sq[front].x; y=sq[front ].y;
     for (v=0;v<8;v++)                        /*循环扫描每个方向，共8个方向*/
     { i=x+move[v].dx;  j=x+move[v].dy;/*选择一个前进方向(i,j)*/
      if (maze[i][j]==0)                      /*如果该方向可走*/
         { rear++;                            /*入队*/
           sq[rear].x=i;  sq[rear].y=j;  sq[rear].pre=front;
           maze[i][j]=-1;                     /*将其赋值为-1，以免重复搜索*/
         }
      if (i==m && j==n)                       /*找到了出口*/
```

```
        { printpath(sq,rear);                    /* 打印迷宫 */
          restore(maze);                         /* 恢复迷宫,使数组 maze 中的 -1 全变成 0 */
          return 1;}
      }
  front++;                                       /* 当前点搜索完,取下一个点搜索 */
    }
 return 0;
}

void printpath(sqtype sq[],int rear)             /* 打印迷宫路径 */
{ int i;
  i = rear;
  do
   {printf(" ( %d, %d)?",sq[i].x , sq[i].y);
    i = sq[i].pre; /* 回溯 */
   } while (i!= -1);
}
```

在此例中,不能采用循环队列,因为在本问题中,队列里保存了探索到的路径序列,如果采用循环队列,则会覆盖先前得到的路径序列。在有些问题中,如持续运行的实时监控系统中,监控系统源源不断地收到监控对象顺序发来的信息,如报警,为了保持报警信息的顺序性,就要按顺序一一保存。这些信息有无穷多个,不可能全部同时驻留内存,可根据实际问题,设计一个适当大的向量空间,用作循环队列,最初收到的报警信息一一入队,当队满之后,又有新的报警到来时,新的报警则覆盖掉了旧的报警,内存中始终保持当前最新的若干条报警,以便满足快速查询需求。

3.5 递　　归

递归是算法设计中最常用的手段,它通常将一个大型复杂的问题转化为一个与原问题相似的规模较小的问题来求解,往往通过少量的语句实现重复的计算,起到事半功倍的作用。

递归是栈的一个重要应用。在设计一些问题的算法时,经常需要将原问题分解为若干子问题求解,而原问题的求解方式与子问题的求解方式相同。因此,如果求解原问题的程序段是一个函数,则可以在函数体内调用函数自身来实现对子问题的求解,这就是一种组织形式——递归。

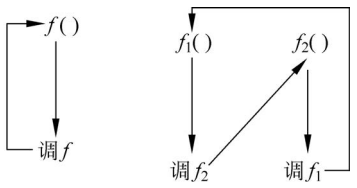
3.5.1 递归的定义与实现

如果一个函数直接调用自己或者通过一系列调用间接地调用自己,则称这一函数是递归定义的。

递归不仅是程序的一种组织形式,更是软件设计中一种重要的方法和技术。由于递归程序通过调用自身来完成与自身要求相同的子问题的求解,因而省略了程序设计中的许多细节操作,简化了程序的设计过程,并在求解许多复杂问题时,采用递归技术更简单、更高效。正因为如此,递归技术也较多地应用于程序的开发中。

1. 递归程序的定义

递归程序直接或间接调用自己。若函数体内直接调用自身，则称为直接递归；若一个函数通过调用其他函数并由其他函数反过来又调用该函数，则称为间接调用。函数直接调用和间接调用如图 3.18 所示。



(a) 直接调用　　(b) 间接调用

图 3.18　函数直接调用和间接调用示意图

下面是两个递归函数的实例：

1）求解整数 $n!$

$$f(n)=\begin{cases}1 & n=1,0\\ nf(n-1) & n>1\end{cases}$$

从函数的定义中看到，为求 $n!$，必须求 $(n-1)!$，而要求 $(n-1)!$，又必须求 $(n-2)!$，依次类推，如图 3.19 所示。

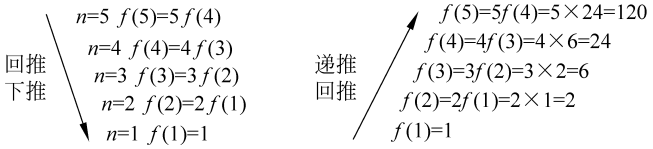


图 3.19　求 $n!$ 的递归求解过程

求解 $n!$ 的递归函数算法如下，这是一个自身调用自身的递归函数。

```
int fact(int n)
  {   int f;
    if(n < 0) printf("n < 0,data error!");
     else
      if(n == 0 || n == 1)
          f = 1;
      else
          f = fact(n - 1) * n;
  return f;
}
```

2）两个函数通过相互调用实现间接递归调用

```
void  p2(int n)
  {if(n > 0)
      if(n % 2 == 1)
          {p2(n - 1);
          printf(" % d\n",n);}
      else
          {printf(" % d\n",n);
          p3(n - 2);}
```

```
  }
void  p3(int n)
  {if(n > 0)
      if(n % 3 == 1)
          { printf(" % d\n",n);
          p2(n - 1);}
      else
          {p3(n - 2);
          printf(" % d\n",n);}
}
```

在以下 3 种情况下,常常要用到递归的方法。

第 1 种情况,定义是递归的。现实中有许多实际问题是递归定义的,这时用递归方法可以使问题的描述大大简化(如上述求 $n!$)。递归函数都有一个终止的条件(如求 $n!$ 时的 $n=0$),它使递归不再执行下去。此外,数学上常用的幂函数、Fibonacci 数列等,它们的定义和计算也都是递归的。

第 2 种情况,数据结构是递归的。某些数据结构是递归的,则它们的操作可递归地描述。例如,链表就是一种递归的数据结构,其结点 node 的定义由数据域 data 和指针域 next 组成,而指针则由 node 定义。对于递归的数据结构采用递归的方法来编写十分方便。

例 3.8 使用递归查找非空不带头结点的单链表的最后一个结点,并输出其数据域的值。

```
void find(linklist L)
    {/ * 输出非空不带头结点的单链表 L 的最后一个结点数据域的值 * /
if(L -> next == NULL)
   printf(" % d\n", L -> data);
 else
   find(L -> next);
}
```

如果 L -> next==NULL,表明 L 已到达单链表的最后一个结点,此时可输出结点数据域的值,否则以 L -> next 为表头指针继续递归执行该项过程。

例 3.9 使用递归非空不带头结点的单链表中查找其数据域的值等于给定值 x 的结点,并输出其值,递归结束条件是 L!=NULL 且 L -> data==x。

```
void findd(linklist L, datatype x)
{/ * 在非空不带头结点的单链表 L 中查找其数据域的值等于 x 的结点并输出其值 * /
 if(L -> data == x)
   printf(" % d\n", L -> data);
 else
   findd(L -> next, x);
}
```

后面将介绍的树结构的定义也是递归的,所以,关于树的一些算法也可以用递归来实现。

第 3 种情况,某些问题自身没有明显的递归结构,但用递归方法求解更简单。一个典型的例子就是 Hanoi 问题(在此不做介绍,有兴趣的读者可以参阅其他参考书)。

2. 递归程序的实现

递归函数类似于函数的多层调用,只是调用者和被调用者是同一个函数。在每次调用时,系统将属于各个递归层次的信息组成一个活动记录(包含本层调用的参数、返回地址、局部变量等信息),并将这个活动记录保存在系统的“递归工作栈”中,每当递归调用一次,新产

生的活动记录入栈，一旦本次调用结束，则将栈顶活动记录出栈，系统根据出栈的返回信息返回本次的调用处，继续向下执行。下面以 4! 为例说明执行递归调用 fact(4)时工作栈的情况，如图 3.20 所示。

函数 fact 执行过程	返回地址
函数 fact(4)被调用，先把返回主调函数的地址入栈，然后再调用 fact(4)	返回主调函数地址
为计算表达式 4fact(3)，先将表达式的地址入栈，然后再调用 fact(3)	表达式 4fact(3)的地址 返回主调函数地址
为计算表达式 3fact(2)，先将表达式的地址入栈，然后再调用 fact(2)	表达式 3fact(2)的地址 表达式 4fact(3)的地址 返回主调函数地址
为计算表达式 2fact(1)，先将表达式的地址入栈，然后再调用 fact(1)	表达式 2fact(1)的地址 表达式 3fact(2)的地址 表达式 4fact(3)的地址 返回主调函数地址
执行 return f 语句，弹出栈顶元素(表达式 2fact(1)的地址)，把 fact(1)的值返回到调用表达式 $f=2\text{fact}(1)$	表达式 3fact(2)的地址 表达式 4fact(3)的地址 返回主调函数地址
把 fact(2)的值返回到调用表达式 $f=3\text{fact}(2)$	表达式 4fact(3)的地址 返回主调函数地址
把 fact(3)的值返回到调用表达式 $f=4\text{fact}(3)$	返回主调函数地址
把 fact(4)的值返回到主调函数，主调函数继续向下执行	栈空

图 3.20　求 4! 递归工作栈示意图

3.5.2　递归消除

在求 $n!$ 的递归算法中，通过图 3.20 的讨论可以看出，递归程序在运行时要花费较多的时间和空间，效率较低。虽然并不是一定要禁止使用递归，但是，为了提高效率，有时需要消去在一个程序中最经常执行部分的递归调用。下面通过几个实例来讨论消除递归的技术。

1. 基于迭代的递归消除

例 3.10 计算幂函数 x^n 的递归 C 语言函数。

```
double power(double x, unsigned n)
{/* 计算幂函数 x^n 的递归函数 */
 if(n == 0)
 return 1.0;
 return power(x, n-1) * x;
}
```

再将递归函数 power( )的递归部分用一个循环来代替,将递归的终止条件作为循环的结束条件。

```
double power(double x, unsigned n)
{/* 计算幂函数 x^n 的递归函数 */
 if(n == 0)
   return 1.0;
 while( --n)
   y *= x;
   return  y;
}
```

函数中 while 语句的循环体执行 $n-1$ 次,时间复杂度是 $O(n)$。

例 3.11 求具有 n 个元素的数组 a 的各元素之和的递归算法。

```
float psum(float a[ ],  int n)
{/* 求数组 a 的各元素之和数 */
 if(n <= 0)
   return 0;
 else
  return psum( a , n-1) + a [n-1];
}
```

再将递归函数 psum( )的递归部分用一个循环来代替,将递归的终止条件作为循环的结束条件。

```
float psum(float a[ ],  int n)
{/* 求数组 a 的各元素之和 */
 float sum = 0;
 for(int i = 0; i < n; i++)
   sum += a[i];
  return sum;
}
```

2. 基于栈的递归消除

很多情况下,一个递归算法无法转化成循环算法,这时,通常引入一个工作栈作为控制机制以消除递归。

例 3.12 求具有 n 个元素的数组 a 的各元素之和,要求用顺序栈消除递归的算法。

```
int psums(int a[ ],int n)
{/* 求数组 a 的各元素之和数 */
 int sum = 0, i , j;
 initstack( );                          /* 初始化栈 s */
 while(i < n-1)
   { push(s, x);                        /* 入栈 */
```

```
    i++;
   }
 while (!stackempty(s))
  { j = pop(s);                              /* 出栈 */
   sum += j; }
  }
```

例 3.13　求具有 n 个元素的数组 a 的最大元素的算法。

递归算法：

```
int maxs(int i)
{/* 此函数返回数组 a 中最大元素的下标值,a[ ]和 n 是全局量 */
 if(i < n - 1)
   { j = maxs(i + 1);
     if(a[i] > a[j])k = i;
     else  k = j;
   }
   else k = n - 1;
   return k;
  }
```

非递归算法(用栈实现)：

```
int maxs1(int i)
{/* 此函数返回数组 a 中最大元素的下标值 */
 int k , j;
 initstack( );                               /* 初始化栈 s */
 while(i < n)
   { push(s, x);                             /* 入栈 */
    i++;
   }
   else k = n - 1;
 while (!stackempty(s))
  { j = pop(s);                              /* 记下出栈元素的下标 */
      if(a[k] < a[j])k = j;
  }
  return k;
  }
```

经过一系列简化得到的非递归算法：

```
int maxs2(int a[ ],int n)
{/* 此函数返回数组 a 中最大元素的下标值 */
 int  i , k = n - 1;
   i = n - 1;
 while(i > 1)
    { i -- ;
      if(a[i] > a[k])k = i;
    }
    return k;
    }
```

基于上述各例,得出基于栈的递归消除的转换规则如下：

(1) 置一个栈 s,开始时为空。

(2) 在被调用函数的入口处设置一个标号,以便返回(隐含在程序中)。

(3) 函数的每一递归调用,用以下与其等价的操作来替换。

① 保留现场：开辟栈顶存储空间，用于保存返回地址、调用层的形式参数和局部变量等信息。

② 准备数据：为被调用函数准备数据，即计算实参的值，并赋给对应的形参。

③ 转入被调用函数执行。

④ 调用返回：若调用函数需要返回值，则从回传变量中取出所要保存的值送到相应的位置。

(4) 对返回语句可用以下几个等价语句来替换。如果栈不空，可依次执行如下操作；否则结束本函数调用，返回。

① 回传数据：若调用函数需要返回值，将其值保存到回传变量中。

② 恢复现场：从栈顶取出返回地址及各变量、形参的值，并出栈。

③ 返回：按返回地址返回。

需要说明的是，按这样的转换方法得到的程序结构一般是比较差的，因而需要重新进行调整。

本章小结

- 栈和队列是两种常见的数据结构，它们都是运算受限制的线性表。栈的插入和删除均在栈顶进行，它的特点是后进先出；队列的插入在队尾进行，删除在队头进行，它的特点是先进先出。在解决具有“后进先出”特点的实际问题时，可以使用“栈”；在解决具有“先进先出”特点的实际问题时，可以使用“队列”。
- 根据存储方式的不同，栈可以分为顺序栈和链栈；而队列也可以分为顺序队列和链队列，但一般情况下使用的顺序队列是循环队列。本章介绍了顺序栈、链栈、链队列和循环队列的各种基本运算，读者应该掌握。
- 读者应该重点领会栈和队列的“溢出”(上溢和下溢)概念及其判别条件，并掌握栈空、栈满、队列空和队列满的正确判别方法，以便及时控制返回。

习 题 3

一、填空题

1. 线性表、栈和队列都是_____结构，可以在线性表的_____位置插入和删除元素；对于栈只能在_____插入和删除元素；对于队列只能在_____插入和在_____删除元素。

2. 栈是一种特殊的线性表，允许插入和删除运算的一端称为_____；不允许插入和删除运算的一端称为_____。

3. _____是被限定为只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表。

4. 在一个循环队列中，队首指针指向队首元素的_____位置。

5. 在具有 n 个单元的循环队列中，队满时共有_____个元素。

6. 向栈中压入元素的操作是先_____，后_____。

7. 从循环队列中删除一个元素时，其操作是先_____，后_____。

8. 在操作序列 push(1)，push(2)，pop()，push(5)，push(7)，pop()，push(6)之后，栈顶元素是_____，栈底元素是_____。

9. 在操作序列 enqueue(1)，enqueue(2)，dequeue()，enqueue(5)，enqueue(7)，dequeue()，enqueue(9)之后，队头元素是_____，队尾元素是_____。

10. 用单链表表示的链式队列的队头在链表的_____位置。

二、选择题

1. 栈中元素的进出原则是(　　)。

A. 先进先出　　B. 后进先出　　C. 栈空则进　　D. 栈满则出

2. 已知一个栈的入栈序列是1，2，3，…，n，其输出序列为 p_1，p_2，p_3，…，p_n，若 $p_1=n$，则 p_i 为(　　)。

A. i　　B. $n=i$　　C. $n-i+1$　　D. 不确定

3. 如果入栈是元素先入栈，然后 ST -> top++，则判定一个栈 ST(最多元素为 m_0)为空的条件是(　　)。

A. ST -> top!=0　　B. ST -> top==0

C. ST -> top!=m_0　　D. ST -> top==m_0

4. 当利用长度为 N 的数组顺序存储一个栈时，假定用 top==N 表示栈空，则向这个栈插入一个元素时，首先应执行(　　)语句修改 top 指针。

A. top++　　B. top--　　C. top　　D. top=0

5. 假定一个链栈的栈顶指针用 top 表示，当 p 所指向的结点进栈时，执行的操作是(　　)。

A. p -> next=top;top=top -> next;　　B. top=p -> p;p -> next=top;

C. p -> next=top -> next;top -> next=p;　　D. p -> next=top;top=p;

6. 判定一个队列 QU(最多元素为 m_0)为满的条件是(　　)。

A. QU -> rear-QU -> front==m_0　　B. QU -> rear-QU -> front -1==m_0

C. QU -> front==QU -> rear　　D. QU -> front==QU -> rear+1

7. 数组 $Q[n]$用来表示一个循环队列，f 为当前队列头元素的前一位置，r 为队尾元素的位置，假定队列中元素的个数小于 n，则计算队列中元素的公式为(　　)。

A. $r-f$　　B. $(n+f-r)\%n$

C. $n+r-f$　　D. $(n+r-f)\%n$

8. 假定一个链队的队首和队尾指针分别为 front 和 rear，则判断队空的条件为(　　)。

A. front==rear　　B. front!=NULL

C. rear!=NULL　　D. front==NULL

9. 假定利用数组 $a[N]$循环顺序存储一个队列，用 f 和 r 分别表示队首和队尾指针，并已知队未空，当进行出队并返回队首元素时所执行的操作为(　　)。

A. return(a[++r%N])　　B. return(a[--r%N])

C. return(a[++f%N])　　D. return(a[f--%N])

10. 从供选择的答案中选出最确切的一项，把相应编号填入对应的栏内。

设有4个数据元素 a_1、a_2、a_3 和 a_4，对它们分别进行栈操作或队操作。在进栈或进队

操作时,按 a_1、a_2、a_3、a_4 次序每次进入一个元素。假设栈或队的初始状态都是空。

现要进行的栈操作是进栈两次,出栈一次,再进栈两次,出栈一次;这时,第 1 次出栈得到的元素是(　　),第 2 次出栈得到的元素是(　　);类似地,考虑对这 4 个数据元素进行的队操作是进队两次,出队一次,再进队两次,出队一次;这时,第 1 次出队得到的元素是(　　),第 2 次出队得到的元素是(　　)。经操作后,最后在栈中或队中的元素还有(　　)个。

A. a_1　　B. a_2　　C. a_3　　D. a_4　　E. 1　　F. 2　　G. 3　　H. 0

三、算法分析题(请写出下列各算法的功能)

1.
```
int M(int x)
(int y;
if(x>100) return(x-10);
else
 {y=M(x+11);
  return(M(y));
 }
}
```

2.
```
void a1(Seqstack S)
{int I, n, a[100];
n=0;
while(!SeqstackEmpty(S)) {n++; Pop(S,a[n]);}
for(I=1; I<=n; I++) Push(S,a[I]);
}
```

3.
```
void a2()
{Queue Q;
 InitQueue(Q);
 Char x='e', y='c';
 EnQueue(Q,'h'); EnQueue(Q,'r'); EnQueue(Q,y);
 x=DeQueue(Q);  EnQueue(Q,x);
  x=DeQueue(Q);  EnQueue(Q,'a');
while(!QueueEmpty(Q))
 { y=DeQueue(Q);
   printf("%c",y)
 }
printf("%c",x);
}
```

四、算法设计题

1. 设单链表中存放着 n 个字符,试设计算法判断字符串是否为中心对称的字符串。例如"abcdedcba"就是中心对称的字符串。

2. 编写一个表达式中开、闭括号是否合法配对的算法。

3. 编号为 1、2、3、4 的 4 列火车通过一个如图 3.1(b)所示的栈式的列车调度站,可能得到的调度结果有哪些?如果有 n 列火车通过调度站,请设计一个算法,输出所有可能的调度结果。

4. 设有两个栈 S_1、S_2 都采用顺序栈方式,并且共享一个存储区[0..maxsize-1],为了尽量利用空间,减少溢出的可能性,可采用栈顶相向、迎面增长的存储方式,试设计入栈、出

栈的算法。

5. 假设用一个单循环链表来表示队列(也称为循环队列),该队列只设一个队尾指针,不设队头指针,试编写相应的入队和出队的算法。

6. 假设将循环队列定义为:以域变量 rear 和 length 分别指示循环队列中队尾元素的位置和内含元素的个数。试给出循环队列的队满条件,并写出相应的入队和出队的算法。

7. 设计算法,判断一个算术表达式中的圆括号是否正确配对。

8. 设单链表中存放有 n 个字符,编写算法,判断该字符串是否有中心对称关系(又称回文),例如 xyzzyx 与 xyzyx 都是中心对称的字符串。