

第 4 章 特殊线性表—栈、队、串

栈、队列、串是常用数据结构。其中栈与队列不仅可直接用于描述问题，而且大量用于算法的实现中。串多用于直接描述非数值的简单信息。

从数据元素间的逻辑关系看，栈、队列与串是线性表，但从操作方式与种类看，它们与线性表有许多不同。因此，若把数据间逻辑关系与相应的操作作为整体看待（即作为抽象数据类型），它们应为新的数据结构。事实上，栈与队列是操作受限的线性表，串是元素受限的线性表。尽管它们与线性表有许多共同点，但也有不少特殊性。本章重点介绍这些特殊性。

§ 4.1 栈

§ 4.1.1 栈的逻辑结构

(一) 基本概念

栈是一种限定仅在表的一端进行插入与删除的线性表。允许进行插入与删除的这一端称为**栈顶**，而另一端称为**栈底**，不含元素的空表称为**空栈**，插入与删除分别称**进栈**与**出栈**。见图 4-0。

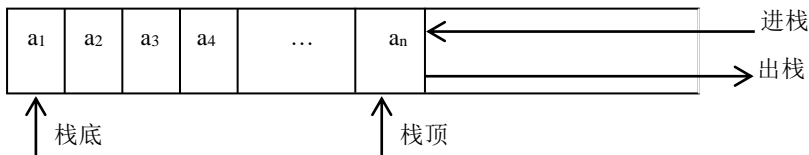


图 4-1 栈示意图

由于插入与删除只能在同一端进行，所以较先进入栈的元素，在进行出栈操作时，较后才能出栈。特别是，最先进栈者，最后才能出栈，而最晚进栈者，必最先出栈。因此，栈也称作**先进后出**（First In Last Out）的线性表，简称 **FILO** 表。

在现实世界中，也有许多符合栈模型的例子。例如，火车扳道站、单车道死胡同，都是栈的例子。在计算机系统中，栈的例子更是多见，例如，记录中断返回地址（断点）

的结构就是栈。在中断发生时，为了处理完中断事件后恢复被中断程序的继续执行，需记下断点。在允许多级中断的情况下，中断处理过程仍可能被其它中断所中断，因此，系统可能同时要保存多个断点。由于中断恢复是先恢复最近被打断的过程，所以断点的保存次序与取出次序正好相反，这正好满足栈的特性。

栈中元素除具有线性关系外，还具有先进后出的特性，所以在应用时，应根据这两点决定是否使用栈。

通过使用栈，可以产生一定规则的排列次序。例如，我们假定，有 n 个元素 $1, 2, \dots, n$ ，它们按 $1, 2, \dots, n$ 的次序进栈， i 进栈时， $1 \sim (i-1)$ 都已进过栈，其中某些因素也可以已经按规则出了栈。每种元素只允许进一次栈。而出栈无具体规定（即可随时出栈），每个元素位于栈顶时就可以出栈了。这样，每种完全进出栈后，出栈次序都是这 n 个元素的一个全排列。全排列最多有 $n!$ 种，那么，在这 $n!$ 种全排列中，有哪些是可能的出栈次序？有哪些是不可能的？例如，对 $n=3$ ，则全排列有 $3!=6$ 种，其中可能的出栈次序是：

1 2 3: 1 进—1 出—2 进—2 出—3 进—3 出
 1 3 2: 1 进—1 出—2 进—3 进—3 出—2 出
 2 1 3: 1 进—2 进—2 出—1 出—3 进—3 出
 2 3 1: 1 进—2 进—2 出—3 进—3 出—1 出
 3 2 1: 1 进—2 进—3 进—3 出—2 出—1 出

不可能的次序是：

3 1 2

从该例看出，在可能的出栈序列中，若对任意元素 k ，若它后面有小于它的元素，则这些小于它的元素必须以“逆序”出现。例如， $n=4$ 时，1423 是不可能的出栈序列，因为 4 后面的 23 是顺序排列。据此，对于一般的 n ，可设计算法，编写一个计算机程序，找出所有的可能出栈次序和不可能出栈次序。具体留做练习。

(二) 栈抽象模型

这里，我们将栈视为一个抽象对象/类（亦称接口），即不考虑它的具体数据结构存储，不考虑基本操作的实现，只考虑它的基本操作的接口（输入/输出）。该抽象对象/类从面向对象观点定义了栈的属性、方法。由于是抽象的，所以，该类无具体对象，只用做派生各种对象/类，例如，下面将介绍的栈的顺序存储和链式存储所对应的类均是这里的抽象类的派生类。

从基本操作上看，栈是线性表的子集，故应当是线性表的父类，这样，线性表类也可以共享栈的方法。

```
template <class TElem>
```

```
//设栈元素可以是任意类型，故使用类模板，用 TElem 代表可变元素类型
```

```
class TStack0
```

```
{
```

```
protected:
```

```

        long len;
public:
    long GetLen(void) {return len; };
    char IsEmpty() {return (len<=0)? 1:0; };
    virtual TElem& Push(TElem &elem)=0;
    virtual TElem& Pop(void)=0;
    virtual TElem& GetTop(void)=0;

    virtual TElem& RollDown()=0;
    virtual TElem& RollUp()=0;
    virtual void Clear()=0;

};

```

下面对该类中主要部分进行说明。

len: 私有数据成员，表示栈中当前元素个数。

GetLen(void): 返回栈长度（元素个数）。有了 len，该函数可直接在此实现。

IsEmpty(): 检查栈是否为空，空时返回逻辑真，否则返回逻辑假。该函数可直接在此实现。

Push(TElem &elem): 进栈操作，将元素 elem 压入栈。返回栈顶元素的引用。溢出时触发异常。

Pop(): 出栈操作，将栈顶元素摘下，并返回其值。下溢（栈空）时触发异常。

GetTop(void): 与 Pop()类似，只是不摘下元素。

RollDown(): 将栈中元素循环向下（栈底方向）下推一个位置，栈底元素到达栈顶。

RollUp(): 将栈中元素循环向上（栈顶方向）上推一个位置，栈顶元素到达栈底。

§ 4.1.2 栈的顺序存贮结构

(一) 存贮方法

栈是一种特殊的线性结构，故它可按线性结构的存贮方式存贮，即顺序与链式。本节先介绍顺序方式。

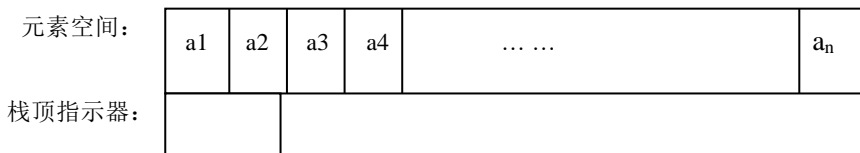


图 4-2 栈顺序存储结构
假设栈底在地址小的一端

仍然用一维数组模拟连续的存贮空间（在此是存放栈元素的空间）。由于插入与删除都是针对表的一端进行的，所以设置一个量指示插入/删除端的位置，一般称该量为**栈顶指针**（指示器）。不过，在栈底固定的顺序存储下，栈顶指示器与栈长度存在对应关系，它们二者可以互相推出。这种用一维数组与栈顶指针表示栈的结构如图 4-0 所示。

(二) 顺序栈类

根据上面的讨论，顺序结构的栈的类定义如下。该类是前面介绍的抽象类 TStack0 的派生类。下面就新出现的成分说明。

room: 指向一块 TElem 类型的动态空间，充当元素类型为 TElem 的一维数组，用于存储栈元素。

size: 记录 room 所指空间的大小。

elemBuff: 元素缓冲区，主要用于临时存放从栈中摘取下的元素，作为函数的返回值引用。

long ResizeRoom(long newSize): 重新分配一块大小为 newSize 的空间，并复制原数据。

CopyRoom(TElem *objRoom, long n1, TElem *srcRoom, long n2): 用于复制数据。

```
template <class TElem>
class TStackSqu : public TStack0<TElem>
{
protected:
    TElem *room;
    TElem elemBuff;
    long size;

    long CopyRoom(TElem *objRoom, long n1, TElem *srcRoom, long n2);
    long ResizeRoom(long newSize);
public:
    TStackSqu();
    TStackSqu(long mSize);
    ~TStackSqu();

    virtual TElem& Push(TElem &elem);
    virtual TElem& Pop(void);
    virtual TElem& GetTop(void);

    virtual TElem& RollDown();
```

```
virtual TElem& RollUp();  
virtual void    Clear();  
  
virtual void Print();  
  
};
```

(三) 基本操作的实现

下面的程序给出了栈的连续存贮结构下的基本操作的实现。由于较简单，这里就不给出说明了。

```
template <class TElem>  
TStackSqu<TElem>::TStackSqu() //构造函数，实现初始化  
{  
    room=NULL; //未分配空间，置相应的标志  
    size=0;  
    len=0;  
}  
  
template <class TElem> //另一构造函数，实现初始化和空间分配  
TStackSqu<TElem>::TStackSqu(long mSize)  
{  
    len=0;  
    room=new TElem[mSize]; //分配 m 个元素的空间  
    if (room==NULL) throw TExcepComm(4);  
    size=mSize; //记下空间大小  
}  
  
template <class TElem>  
TStackSqu<TElem>::~~TStackSqu()  
{ //析构函数，释放在构造函数中所申请的空间  
    if (room!=NULL) delete[] room;  
}  
  
template <class TElem>  
TElem& TStackSqu<TElem>::Push(TElem &elem)  
{ //进栈操作，将 elem 压入栈，并返回栈顶元素
```

```
    if (len>=size-1)
    { //栈空间不足时，调用 ResizeRoom 重新分配空间
        long ret=ResizeRoom(size+10);
        if (ret<0) throw TExcepComm(3);
    }
    room[len]=elem; //elem 进栈
    len++;
    return room[len-1];
}

template <class TElem>
TElem& TStackSqu<TElem>::Pop(void)
{ //出栈操作，将当前栈顶元素摘除，并将其作为返回值
    if (len<=0) throw TExcepComm(3); //栈空时触发异常
    len--; //栈长度减一，表示当前栈顶元素不被认为属于栈，即相当于被摘除
    return room[len]; //返回当前栈顶元素
};

template <class TElem>
TElem& TStackSqu<TElem>::GetTop(void)
{ //返回栈顶元素
    return room[len-1];
}

template <class TElem>
void TStackSqu<TElem>::Clear()
{ //清空栈，只需将长度标识置为 0
    len=0;
}

template <class TElem>
TElem& TStackSqu<TElem>::RollDown()
{ //栈元素下滚一位
    TElem x;
    x=room[0];
    for (long i=1; i<len; i++)
        room[i-1]=room[i];
    room[len-1]=x;
```

```
        return room[len-1];
    }

    template <class TElem>
    long TStackSqu< TElem>::
    CopyRoom(TElem *objRoom, long n1, TElem *srcRoom, long n2)
    { //复制，将 srcRoom 中内容(n2 个元素)全部复制到 objRoom
      //返回实际复制元素数目
      long i, k;

      k=n2;
      if (k>n1) k=n1; //源数据个数(n2)大于目标空间(n1)时，只复制 n1 个

      for (i=0; i<k; i++)
          objRoom[i] = srcRoom[i]; //复制
      return k;
    };

    template <class TElem>
    long TStackSqu< TElem>::ResizeRoom(long newSize)
    { //重新分配空间，新空间大小为 newSize
      long i, k;
      TElem *pE;

      pE= new TElem[newSize]; //申请一块新空间
      if (pE==NULL) throw TExcepComm(2);

      len = CopyRoom(pE, newSize, room, len); //调用 CopyRoom 将原空间的内容复制到新空间
      if (room!=NULL) delete room; //删除原空间
      room = pE; //令 room 存储新空间的地址
      size=newSize;

      return len;
    };
};
```

§ 4.1.3 栈的链式存储结构

(一) 存储方法

与线性表类似，栈也可采用链式存储结构。线性表采用链式存储，主要有两个目的：避免插入/删除操作带来的移动元素问题与存储区的预申请问题。对于栈，由于插入与删除只在栈顶进行，故不存在第一个问题。所以栈采用链式存储结构主要目的是避免存储区的预申请问题。

我们这里仍然采用单链表存储栈元素。单链表的结点类型同前面介绍的链式线性表的结点元素。

与线性链表类似，链式栈也需要一个描述结点，它作为整个栈的代表。它的主要数据项为指向栈顶元素的指针。另外可设一个存储当前栈中元素个数的量。链式栈中的存储栈元素的结点的结构与线性链表相同。图 4-0 是链式栈的示意。

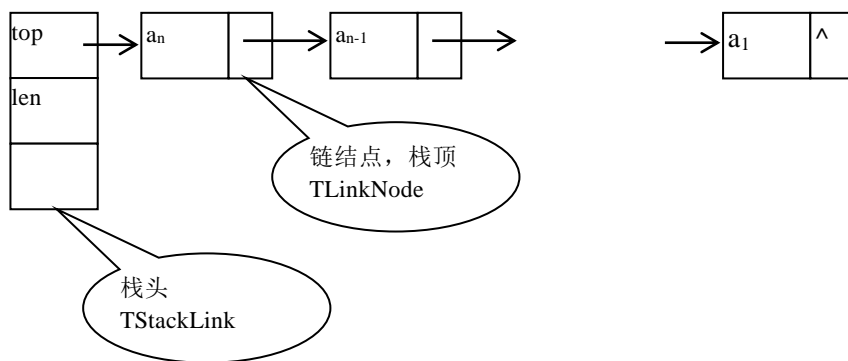


图 4-3 链式栈基本形式

(二) 面向对象描述

我们这里将首结点的地址和链表中当前结点个数等有关链表的信息，看做一个类中的数据成员，并针对其设置有关操作，构成一个类 `TStackLink`。在定义该类前，需定义链表中结点的类型 `TLinkNode`：

```
template <class TElem>
struct TLinkNode
{
    TElem info; //栈元素内容，为可变类型 TElem
    TLinkNode *next; //链指针
};
```


类 TStackLink 从抽象类 TStack0 继承而来，其定义如下：

```
template <class TElem>
class TStackLink : public TStack0<TElem>
{
    protected:
        TLinkNode<TElem> *top; //栈顶结点指针
        TElem elemBuff; //元素内容缓冲区
    public:
        TStackLink();
        ~TStackLink();

        virtual TElem& Push(TElem &elem);
        virtual TElem& Pop(void);
        virtual TElem& GetTop(void);

        virtual TElem& RollDown();
        virtual TElem& RollUp();
        virtual void    Clear();

    //Newly added functions
        virtual TLinkNode<TElem>* PushNode(TLinkNode<TElem> *pNode);
        virtual TLinkNode<TElem>* PopNode(void);
        virtual const TLinkNode<TElem>* GetTopNode(void);

        virtual void Print();
};
```

TStackLink 类从数据结构的角度看，相当于图 3.3 所示的栈头结点。

该类仍然是从栈抽象类 TStack0 继承而来。除了包含了 TStack0 的全部成员外，还增加了一套针对链指针的操作。这些操作实质上是很直接的、很基本的，是实现其他操作的重要基础，另一方面，由于链表本身的特殊性，也直接需要将这些有关链的操作作为基本操作。

它的几个重要的数据成员为：

top----栈顶指针，指向栈中第一个元素结点。

len----表长度，即栈中结点的数目。

至于成员函数的说明，在下面给出。

(三) 基本操作实现

1. 初始化：通过构造函数实现，只简单地将 `top` 置为空，表示无结点存在。析构函数的任务是释放所有结点（通过调用 `Clear()`）。

```
template <class TElem>
TStackLink<TElem>::TStackLink()
{
    top = NULL;
    len=0;
}

template <class TElem>
TStackLink<TElem>::~~TStackLink()
{
    Clear(); //成员函数 Clear()的功能是释放链中所有结点
}
```

2. 进栈 `Push(TElem &elem)`：先生成一个链结点，将 `elem` 值赋予其 `info`，然后调用 `PushNode` 将该结点压入栈，返回栈顶元素的内容的引用。

```
template <class TElem>
TElem& TStackLink<TElem>::Push(TElem &elem)
{
    TLinkNode<TElem> *pNode;

    pNode = new(nothrow) TLinkNode<TElem>; //申请一个新结点
    if (pNode==NULL) throw TExcepComm(4);

    pNode->info = elem ;//TElem must support "="
    PushNode(pNode); //调用成员函数将新结点压入栈

    return pNode->info;
}
```

3. 出栈 `Pop()`：先调用 `PopNode` 将栈顶结点取出（同时将结点值存于 `elemBuff`），然后释放该结点，返回刚刚弹出的元素的内容的引用。

```
template <class TElem>
TElem& TStackLink<TElem>::Pop(void)
{

```

```
TLinkNode<TElem> *pNode;

pNode = PopNode(); //调用成员函数从栈中弹出一个结点，并将值存入 elemBuff
delete pNode;
return elemBuff;
}
```

4. 读栈 GetTop(): 返回栈顶元素的内容的引用。

```
template <class TElem>
TElem& TStackLink<TElem>::GetTop(void)
{
    return top->info;
}
```

5. 结点进栈 PushNode(TLinkNode<TElem> *pNode): 将 pNode 所指结点压入栈顶，返回该结点的地址。

```
template <class TElem>
TLinkNode<TElem>* TStackLink<TElem>::PushNode(TLinkNode<TElem> *pNode)
{
    pNode->next = top;
    top = pNode;
    len++;
    return pNode;
}
```

6. 结点出栈 PopNode(void): 将栈顶结点摘下并返回其地址。

```
template <class TElem>
TLinkNode<TElem> *TStackLink<TElem>::PopNode(void)
{
    TLinkNode<TElem> *pNode;

    if (top==NULL) throw TExcepComm(5);
    elemBuff = top->info; //TElem must support "="
    pNode = top;
    top = top->next; //将第一个结点从链中摘除
    len--;
    return pNode;
}
```

7. 读栈顶结点 GetTopNode(void): 将栈顶结点地址返回。


```
template <class TElem>
const TLinkNode<TElem>* TStackLink<TElem>::GetTopNode(void)
{
    return top;
}
```

8. Clear(): 释放链中所有结点

```
template <class TElem>
void TStackLink<TElem>::Clear()
{
    TLinkNode<TElem> *p, *q;

    p=top; //p 指向第一个结点
    while (p!=NULL)
    {
        q=p;
        p=p->next; //令 p 指向下一个结点
        delete q; //释放原 p 所指结点
    }

    top = NULL; //所有结点都已释放，故应置 top 为空
    len=0;
}
```

注意, "delete p"并不自动将 p 置为空, 所以, 若上面的程序中, 没有 "top=NULL", 则如果再次所用对应的对象, 会发生错误!

§ 4.1.4 多栈共享存贮空间*

(一) 存储方法

栈采用连续存贮结构时, 要预分配足够大的存贮空间。若系统中使用 n 个栈, 每个栈的最大尺寸为 $n_i (i=1, 2, \dots, n)$, 则 n 个栈要预分配的总存贮容量为

$$\sum_{i=1}^n n_i$$

如果已知这 n 个栈在任何时刻的总的存贮占用量均不会超过 N , 且有

$$N < \sum_{i=1}^n n_i$$

则可考虑令这 n 个栈共享一块大小为 N 的存贮区，以节省存贮空间(图 4-0)。

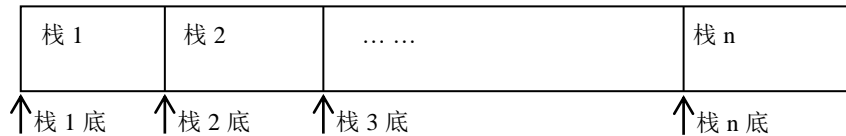


图 4-4 n 栈共享存贮空间

在这种情况下，除第一个栈外，其它栈的栈底位置均不固定，当某栈溢出时（即栈顶与下一个栈的栈底重叠时），应该移动若干个栈，为该栈让出一些空间，这个过程称为栈移动过程 MoveStack。

关于 MoveStack 的算法，是多栈共享存贮空间问题中的关键。它的设计目标应能使系统运行期间内的总的平均元素移动量最小。最直观的目标是，使一次 MoveStack 的执行所引起的移动量最小。然而，由于各个栈在独立工作，它们的进栈出栈操作是随机的，所以某次 MoverStack 的移动量最小并不能保证总的平均移动量最小。

由于 n 栈共享存贮区，且它们的栈底位置不固定，所以需有 n 个栈顶指针和 n 个栈底指针，下面给出这种结构的 C++ 语言描述（假定 n 个栈的元素类型均相同）。

```
template <class TElem> //设栈元素类型为可变类型 TElem
class TStacks
{
    long numStacks; //栈的个数
    long totSize; // 总栈空间尺寸
    TElem *room;    //栈空间，动态空间，做一维数组用
    long *top,      //栈顶指针数组
        *bot; //栈底指针数组

public:
    TStacks(long mNumStacks, long mTotSize);
    ~TStacks();
    //下面是其他成员函数，此略

};
```

我们将这 n 个栈（序号为 $0 \sim \text{numStacks}-1$ ）按固定次序（即相对位置固定）分配在

一维数组 `room` 中。初始时, 令它们各占有 $\left\lfloor \frac{totSize}{numStacks} \right\rfloor$ 个元素的空间 (最后一个栈要多占有一些), 即令

$$top[i]=bot[i]= \left\lfloor \frac{totSize}{numStacks} \right\rfloor * i$$

这里, $i=0, 1, \dots, numStacks-1$ 。

为处理方便, 我们增设一个虚栈, 它的序号为 `numStacks`, 它的底指针定义为

$$bot[numStacks-1]=totSize$$

这种分配可用图 4-0 所示。

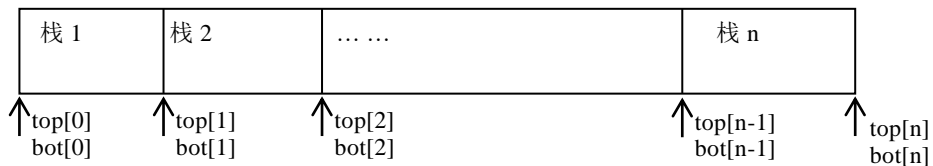


图 4-5 n 栈共享存贮空间初始空间分配

(二) 算法考虑

下面用伪码给出 `n` 栈共享空间的主要算法。

1. 初始化栈

```
Init(s)
{
    for(i=0; i<numStacks; i++)
        s->top[i] = s->bot[i] = i*totSize/numStacks;
    s->bot[numStacks]=totSize;
}
```

2. 进栈

设 `s` 为 `TStacks` 的一个实例的指针, 将元素 `x` 放到 `i` 号栈中的过程为

```
Push ( i, x)
{
    if (s->top[i]+1 == s->bot[i+1])
    {
        ret = MoveStacks(s, i,1); //i 号栈满,调用 MoveStacks 为 i 号栈让出 1 个位置
```

```

    if (ret < 0) 触发异常;    //栈真满, 无空闲空间可利用
}
s->top[i]++;
s->room[s->top[i]] = x;    //进栈
}

```

3. 出栈

设 s 为 TSstacks 的一个实例的指针, 将 i 号栈顶的元素弹出返回的过程为

Pop(i)

```

{
    if (s->top[i]==s->bot[i]) 触发栈空异常;
    x = s->room[s->top[i]];
    s->top[i]--;
    return x;
}

```

4. 栈移动

相比这下, 栈移动过程 MoveStack(s, i, m)较复杂, 它的功能是为 i 号栈腾出 m 个空位。下面是它的一种实现算法。这里, 用 n 表示 numStacks. 为了使读者了解另外一种算法描述方法, 这里采用自然语言流程描述法。

1. [向右搜索空闲区] 对 $r=i+1, i+2, \dots, n-1$, 计算各栈的空闲空间, 并累计到 rs 中, 一旦 $rs \geq m$, 则停止搜索, 并记下当前 r 值。

2. [向左搜索空闲区] 对 $l=i-1, i-2, \dots, 0$, 计算各栈的空闲空间, 并累计到 ls 中, 一旦 $ls \geq m$, 则停止搜索, 并记下当前 l 值。

3. [r 与 l 是否存在?] 若 $ls+rs < m$, 则表明总存贮区中已无满足要求的空闲区, 不能移出空位, 失败返回, 否则, 转下面操作。

4. [确定移量] 若 $rs \geq m$ (r 存在), 计算在 i 栈右方让出空位的元素移动量

$$rm = s \rightarrow top[r] - s \rightarrow top[i]$$

否则(r 不存在)令 $rm=0$ 。若 $ls \geq m$, 计算在 i 栈左边让出空位的元素移动量

$$lm = s \rightarrow top[i] - s \rightarrow bot[l+1]$$

否则令 $lm=0$ 。

5. [确定移动方向] 若 rm 与 lm 均为 0, 则应两边移, 转 8; 若 $lm \leq 0$, 或 $lm > 0$ 但 $lm > rm$, 则表明应向右移, 转 7, (否则应向左移, 下滑到 6)。

6. [左移] 将序号为 $s \rightarrow bot[l+1] \sim s \rightarrow top[i]$ 的数组元素依次向左移 m 位置, 并修改相应的栈顶与栈底指针, 转 9。

7. [右移] 将序号为 $s \rightarrow top[r] \sim s \rightarrow top[i]+1$ 的数组元素依次向右移 m 位置, 并修改相应的栈顶与栈底指针。转 9。

8. [两边移] 根据均衡原则, 分头向左右两移动若干元素 (总量为 m 个)。

9. [结束] (成功) 返回。

(三) 两栈共享存贮空间

对于两栈共享存贮空间，可以特殊处理，以避免移动元素。方法是将两栈的栈底分别固定在存贮区的两端，两栈指针相向增长（见图 4-0）。



图 4-6 两栈共享存贮空间

§ 4.2 队列

§ 4.2.1 队列的逻辑结构

(一) 基本概念

队列(Queue)，简称为队，是一种限定仅分别在表的两端进行插入与删除的线性表。允许插入的一端称**队尾**，允许删除的一端称**队头**。插入与删除操作分别称为**入队**与**出队**。见图 4-0。

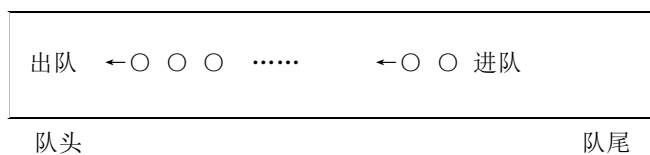


图 4-7 队列示意图

队是一种先进先出 FIFO (First In First Out) 的结构，元素出队的次序与进队次序相同。

现实世界中许多问题（关系）可用队列描述，例如，对顾客服务的部门的工作往往是按排队方式进行的，这类系统称作排队系统。在计算机算法实现中，也经常使用队列记录需按 FIFO 方式处理的数据。

(二) 队列抽象模型

这里讨论队列的面向对象抽象模型。抽象模型主要通过基本操作定义。队列的基本操作主要包括进队、出队。下面是描述队列的抽象模型的类 `TQueue0`，它规定了通用操作接口（基本操作），是各种存储结构的队列的基础。

```
template <class TElem>
class TQueue0
{
protected:
    long len;
public:
    long GetLen(void) {return len; };
    char IsEmpty() {return (len<=0)? 1:0; };
    virtual TElem& QPush(TElem &elem)=0;
    virtual TElem& QPop(void)=0;
    virtual TElem& GetHead(void)=0;

    virtual TElem& RollDown()=0;
    virtual TElem& RollUp()=0;
    virtual void Clear()=0;
};
```

下面对该类中主要部分进行说明。

len: 私有数据成员，表示队中当前元素个数。

GetLen(void): 返回队长度（元素个数）。该函数可直接在此实现。

IsEmpty(): 检查队是否为空，空时返回逻辑 `True`，否则返回 `False`。该函数可直接在此实现。

QPush(TElem &elem): 进队。将元素 `elem` 加入队中，返回其在队中值的引用。溢出时触发异常。

QPop(): 出队。将队头元素摘下，并返回其值的引用。下溢（队空）时触发异常。

GetHead(void): 与 `QPop()` 类似，只是不摘下元素。

RollDown(): 将队中元素循环向下（队尾底方向）推一个位置，队尾元素到达队头。

RollUp(): 将队中元素循环向上（队头方向）推一个位置，队头元素到达队尾。

§ 4.2.2 队列的顺序存储结构

(一) 非循环队列

1. 存储方法

与栈类似，队列也有相应的顺序存储结构。同样用一维数组做为队元素存储空间。

另设两个指示器，分别指示队头与队尾（分别称队头与队尾指针/指示器），其结构如图 4-0 所示。

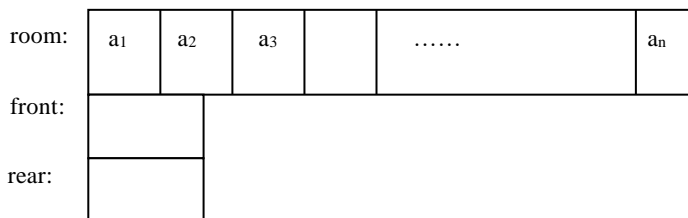


图 4-8 顺序存储结构的队列

room: 指向元素存储空间，作为一维数组使用。

front: 队头指示器。为了方便，令 front 指向队头元素的前一个位置（空位）。

rear: 队尾指示器。这里令 rear 指向队尾元素位置。

空队时，front=0, rear=0

依这种处理方法，room 的第一个元素未被使用，队元素从第二个元素起存放。

这里，由于我们不循环使用队空间，故将这种情况的队列称为非循环队列，这主要是针对下节将要介绍的循环队列而言的。

这种结构与栈的顺序存储结构类似。下面先讨论针对这种结构的基本操作的实现。

2. 进队

进队是将元素添加到队尾，故需访问队尾指针 rear。将元素 x 进队 Q 的操作为：

```
if (队满) 触发异常;
Q->rear++;
Q->room[Q->rear] = x;
```

3. 出队

出队是将当前队中头元素摘取并返回其：

```
if (队空) 触发异常;
Q->front++;
return Q->room[Q->front];
```

4. 队空/满问题

对于这种操作方法，队列变空时有下列关系成立

$$Q \rightarrow \text{front} == Q \rightarrow \text{rear}$$

显然初始状态也满足此式。

由于进队时是每次只对 rear 进行加 1（而没有减少 rear 的操作），故必有一次使 rear 超过 room 的空间大小界限，此时，再按该方法就不能进行进队操作了，这种情况我们称为“队满”。

分析上面的进队与出队算法，由于进队时 `rear` 增加，但出队时 `rear` 也不会减小，所以，当进过 $N-1$ 个元素(设 N 为队空间大小)后，不论此前是否出过队，`rear` 也会越界。这种队满，我们称为“假满”。事实上，如果前面已有元素出了队，则还有空闲区可用，只是由于算法的限制，不能继续执行进队操作罢了。

例如，若 `room` 有 5 个空间，若先进 3 个元素（占到 `room[1]~room[3]`），然后两次出队（剩余元素留在 `room[3]` 中），然后再进一个（当前元素在 `room[3]~room[4]`），此时，若按上面的算法再进队，就会发现已不能执行了。

这种问题的根源是出队操作所腾出的空位不能重复利用。这是一个十分严重的问题，若不解决，无论分配多大的空间，也不能保证不发生溢出。所以，这种队列无实用价值。

解决该方法一般有两种：

- 发生假满时，移动队中元素，在尾部让出空位。但这种方法效率差；
- 循环队列法。这是一种更好的方法，我们在下面介绍。

(二) 循环队列

1. 基本思想

所谓循环队列，就是指将队列的存储区域视作循环结构，即将存储区的第 1 个元素视为最后一个元素的后继，同样将最后元素视为第 1 个元素的前趋，队列的头尾指针的移动均可回绕，按这种方式处理的队列，称为循环队列。

显然，在循环队列中，元素出队后空出的位置可被重复使用，又免去了移动元素的工作。

下面考虑循环队列的实现。假定队空间的下标范围为 $0 \sim N-1$ ，要实现循环队，只要实现头尾指针回绕即可，其它与上面介绍的非循环队列相同。

根据模运算特点，将头尾指针加 1 后按 N 取模即可实现尾指针的回绕。另一种方法是，对头尾指针加 1 后检查其是否大于 $(N-1)$ ，若是则置其为 0。

为了处理方便，我们仍然假定：

- 队头指示器 `front`：指向队中第 1 个元素的前一个位置。
- 队尾指示器 `rear`：指向队尾元素所在位置。

这样的处理，队中有一个元素的空间不被利用。在这种情况下，队空与队满的条件分别为：

- 队空：`rear == front`
- 队满：`(rear+1) MOD N == front`

试想，如果也让 `front` 直接指向队头元素位置（而不是前个位置），判别队空与满就没这样简单了。

如果设置队长度 `len`，队满和空的判别，可直接使用 `len`，而不用上述方法。

2. 实现

循环队的具体的存储用类 TQueueSqu 描述如下。

```
template <class TElem>
class TQueueSqu : public TQueue0<TElem>
{
    protected:
        TElem *room; //队空间首地址，当一维数组用
        TElem elemBuff;
        long front, rear, //队头与尾指示器
            size; //队空间的大小

        long CopyRoom(TElem *objRoom, long n1, TElem *srcRoom, long n2);
        long ResizeRoom(long newSize); //这两个函数与线性表中相同
    public:
        TQueueSqu();
        TQueueSqu(long mSize);
        ~TQueueSqu();

        virtual TElem& QPush(TElem &elem);
        virtual TElem& QPop(void);
        virtual TElem& GetHead(void);

        virtual TElem& RollDown();
        virtual TElem& RollUp();
        virtual void    Clear();

        virtual void Print();
};
```

下面是各主要操作的实现程序。注意，我们这里用队列长度判别队空与满。

```
template <class TElem>
TQueueSqu<TElem>::TQueueSqu()
{ //初始化，队列也未分配空间
    room=NULL;
    front=0;
    rear=0;
    size=0;
    len=0;
}
```

```
template <class TElem>
TQueueSqu<TElem>::TQueueSqu(long mSize)
{ //初始化，并分配队空间
    len=0;
    front=0;
    rear=0;
    room=new(nothrow) TElem[mSize]; //分配 mSize 个 TElem 型元素空间
    if (room==NULL) throw TExcepComm(4);
    size=mSize;
}

template <class TElem>
TQueueSqu<TElem>::~~TQueueSqu()
{ //析构函数，释放已分配的队空间
    if (room!=NULL) delete[] room;
}

template <class TElem>
TElem& TQueueSqu<TElem>::QPush(TElem &elem)
{ //将 elem 进队,并返回该元素在队中的引用
    if (len>=size-1)
    { //若队空间不足，则重新分配
        long ret=ResizeRoom(size+10);
        if (ret<0) throw TExcepComm(3);
    }
    rear++;
    if (rear>=size) rear=0; //rear 加过尾后回绕，实现“循环”
    room[rear]=elem; //TElem must support "="
    len++;
    return room[rear];
}

template <class TElem>
TElem& TQueueSqu<TElem>::QPop(void)
{ //出队。返回队头元素的引用。
    //注意，应及时将引用的值保存，否则以后的进队/出队操作会破坏该元素
    if (len<=0) throw TExcepComm(3);
```

```

len--;
front++;
if (front>=size) front=0; //front 过尾后回绕, 实现“循环”
return room[front];
};

template <class TElem>
TElem& TQueueSqu<TElem>::GetHead(void)
{ //返回队头元素的引用
return room[(front+1) % size];
}

```

§ 4.2.3 队列的链式存储结构

(一) 存储方法

与栈类似, 若队中元素个数变化范围很大, 则可采用链式结构 (动态式)。这里, 链式队列的元素采用线性链表存储。该链表的描述结点应至少含两个指针: 队头元素指针与队尾元素指针, 分别令其指向队中首元素与尾元素对应的结点。这种队列如图 4-0 所示。

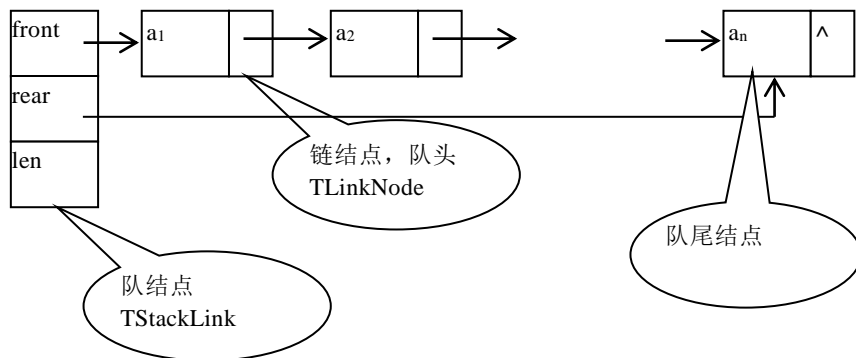


图 4-9 链式队列基本形式

(二) 面向对象描述

设立一个类 `TQueueLink`, 其从 `TQueue0` 派生而来。该类的数据部分描述了图 4-0 所示的“队结点”。至于 `TLinkNode`, 仍同前。

```

template <class TElem>
struct TLinkNode //链结点类型

```

```

{
    TElem info;
    TLinkNode *next; //Note: not necessary for <TElem> to follow TLinkNode here
};

template <class TElem>
class TQueueLink : public TQueue0<TElem> //队类
{
    protected:
        TLinkNode<TElem> *front, *rear; //头与尾指针
        TElem elemBuff;
    public:
        TQueueLink();
        ~TQueueLink();

        virtual TElem& QPush(TElem &elem);
        virtual TElem& QPop(void);
        virtual TElem& GetHead(void);

        virtual TElem& RollDown();
        virtual TElem& RollUp();
        virtual void Clear();

    //Newly added functions
        virtual TLinkNode<TElem>* QPushNode(TLinkNode<TElem> *pNode);
        virtual TLinkNode<TElem>* QPopNode(void);
        virtual const TLinkNode<TElem>* GetHeadNode(void);

        virtual void Print();
};

```

(三) 基本操作实现

这里将各主要基本操作的程序实现列出。

```

template <class TElem>
TQueueLink<TElem>::TQueueLink()
{ //构造函数，实现初始化

```

```
front = NULL;
rear = NULL;
len = 0;
}

template <class TElem>
TQueueLink<TElem>::~TQueueLink()
{ //析构函数，调用 Clear()释放链中所有元素结点
    Clear();
}

template <class TElem>
TElem& TQueueLink<TElem>::QPush(TElem &elem)
{ //进队。元素值 elem 进队，并返回它在队中的值的引用
    TLinkNode<TElem> *pNode;

    pNode = new(nothrow) TLinkNode<TElem>; //申请链结点
    if (pNode==NULL) throw TExcepComm(4);

    pNode->info = elem ;//TElem must support "="
    QPushNode(pNode); //调用 QPushNode，将新生成的链结点推入队

    return pNode->info;
}

template <class TElem>
TElem& TQueueLink<TElem>::QPop(void)
{ //出队。返回队头元素值的引用。注意，在下次调用该函数前要将返回值复制。
    TLinkNode<TElem> *pNode;

    pNode = QPopNode(); //调用 QPopNode，将队头元素摘下
    elemBuff = pNode ->info; //TElem must support "="
    delete pNode;
    return elemBuff; //返回队头元素值的引用
}

template <class TElem>
TElem& TQueueLink<TElem>::GetHead(void)
```



```

{ //返回队头元素的值的引用
    if (front==NULL) throw TExcepComm(2);
    return front->info;
}


template <class TElem>
TLinkNode<TElem>* TQueueLink<TElem>::
    QPushNode(TLinkNode<TElem> *pNode)
{ //结点进队。将 pNode 所指结点进到队尾
    pNode->next = NULL;
    if (rear!=NULL) rear->next = pNode; //队不空时，新结点插入到最后一个结点的后面
    else front = pNode; //队空时，新结点作为队中第一个结点
    rear = pNode; //令 rear 指向新的队尾
    len++;
    return front; //返回队头结点
}

template <class TElem>
TLinkNode<TElem> *TQueueLink<TElem>::QPopNode(void)
{ //结点出队。将队头结点摘除并返回其指针
    TLinkNode<TElem> *pNode;

    if (front==NULL) throw TExcepComm(5); //队空时抛掷异常
    elemBuff = front->info; //TElem must support "="
    pNode = front;
    front = front->next; //将队头元素摘下
    if (front==NULL) rear=NULL; //出队后，若队变空，则 rear 也要置为空
    len--;
    return pNode; //返回所摘下的元素
}

template <class TElem>
const TLinkNode<TElem>* TQueueLink<TElem>::GetHeadNode(void)
{ //返回队头元素指针
    return front;
}

```

 在 QPushNode()中, 在新结点进队前, 要特别注意队是否为空! 为空时相当于在空链中插入结点, 要特殊处理! 类似地, 在 QPopNode()中, 出队后要特别注意队是否已变空, 若已变空, 则 rear 指针要置为空。

§ 4.3 串

现实世界中的实体有多种形式, 如数值、文字符号序列、图形、图象、声音等。其中, 文字(符号)序列称为字符串, 简称串。串在现实世界中是屡见不鲜的。如人名、产品名、事物名称、车牌号、文献、源程序等。在计算机中, 串被认为是一种特殊线性表——元素为文字符号的线性表, 由于现实问题中经常使用串, 所以对串应提供灵活多样的基本操作, 选择合适的存贮结构。

§ 4.3.1 串的逻辑结构

(一) 基本概念

串是由零个或多个字符构成的有限序列, 一般记为

$$s = "a_1a_2 \cdots a_n" \quad (n \geq 0)$$

其中, s 代表**串的名称**, 用双引号括起来的部分(不含该双引号本身)为**串值**(即字符序列), 每个 a_i 为一个字符。字符是计算机可识别的任意符号(字母、数字及其它符号)。串值中字符个数(即 n)称为**串长**, 长度为 0 的串称为**空串**。

串中任意个连续字符构成的部分称为该串的**子串**, 包含子串的串相对该子串称为**主串**。串中某字符在串中出现的序号为该字符在串中的**字符位置**, 子串中第 1 个字符在串中的序号称为该子串在该串中的**位置**。

下面是几个例子:

$s_1 = "ab123"$ //长度为 5 的串

$s_2 = "100"$ //长度为 3 的串

$s_3 = " "$ //含两个空格字符的串, 长度为 2

$s_4 = ""$ //空串, 长度为 0

显然, 若某串的长度为 n , 则在该串中, 长度为 1 的子串的个数为 n 个, 长度为 2 的子串的个数为 $n-1$ 个, ..., 长度为 i 的子串的个数为 $n-(i-1)$ 个, 即 $n-i+1$ 个, 所以, 长度为 n 的串中子串总数(包括空串与自身)为


$$1 + \sum_{i=1}^n (n-i+1) = 1 + \frac{n(n+1)}{2}$$

(二) 串的比较运算的定义


在实际应用中，常需要论及串之间的大小关系。那么，串之间的大小关系是如何定义的呢？首先应定义单个字符间的大小关系。


1. 字符的大小

在计算机中，每个字符都有一个唯一的数值表示——字符编码（字符内码），字符间的大小关系就定义为它们的代码之间的大小关系。字符编码可有多种，对英文字母和符号，ASCII 码是最常见的一种。本教程就用字符的 ASCII 码之间的大小关系代表相应字符的大小关系。例如，字符 A 与 B 的 ASCII 码分别为 65 与 66（十进制），则说 'A' < 'B'。

 ASCII (American Standard code for Information Interchange) 是美国信息交换标准编码，长度为 8 位二进制符号，所以最多能编 $2^8=256$ 个不同的字符。但 ASCII 中规定，最高位为 1 的码代表一些特殊的字符（或命令），所以，ASCII 有效的字符编码为 128 个，其包括英文大小写字母，数字 0~9 及其他一些常用符号（计算机键盘上的字符键）。在字符和数字中，空格编码最小，其次是数字（按 0~9 的次序）、大写字母（按 A-Z 的次序）、小写字母（按 a-z 的次序）等。

对汉字，它们的大小关系也按编码大小确定。汉字的编码也有几种，如大陆用的国标码 (GB2312)，台湾地区用的 Big5 等。

 GB2312（国标码）是针对汉字的 16 位二进制编码，所以最多能编 $2^{16}=65536$ 个不同的码，则足以代表新华字典中的所有汉字。GB2312 将汉字分为两级编码，分别称一级字库和二级字库。一级字库包括了日常用的大多数汉字，它中的汉字的国标码之间的大小关系，与对应的汉语拼音构成的串的大小关系一致（新华字典中，排在较前面的字，它的国标码也较小），而二级字库中的汉字码之间的大小关系，与对应汉字的笔划数目的大小关系一致。

 要在一个系统中能混合所用多种文字，则需要一种统一的编码系统，它将各种文字的基本字符在同一空间内编码。Unicode 就是这样一种编码。

2. 字符串间的大小关系

有了字符大小的定义，就可以考虑串的大小关系了。设 X 与 Y 是两个串：

$$X = "x_1x_2 \cdots x_m"$$

$$Y = "y_1y_2 \cdots y_n"$$

则它们之间的大小关系定义如下：

① 当 $m=n$ 且 $x_1=y_1, \cdots, x_m=y_m$ ，则称 $X=Y$ 。

② 当下列条件之一成立时，称 $X < Y$ ：

i) $m < n$ ，且 $x_i=y_i, i=1, 2, \cdots, m$

ii) 存在某一个 $k \leq \min(m, n)$ ，使得 $x_i=y_i, i=1, 2, \cdots, k-1, x_k < y_k$ 。

③ 不属于情况①与②时，称 $X > Y$ 。

这种方法定义的串的大小关系，也称**字典序**（英文词典中单词的排列次序就是此定义下的非递减次序）。

下面给出几个例子说明此定义。

"abac"与"abac": 相等(=)

"we"与"web": 前者小于后者(情况 i)

"mouth"与"move": 前者小于后者(情况 ii)

(三) 串的基本操作

在实际应用中,对串的操作是多种多样的,所以,为了处理方便,为串引入的基本操作也应很多。下面介绍几种常见的。这里,用 s, s_1, \dots 表示串对象。

- **s.StrChar(long idx):** 返回 s 中序号为 idx 的字符。
- **s.StrCopy(maxNum):** 将 s 的值复制一份,返回其引用(或地址), $maxNum$ 表示最多复制的字符数目,省缺时表示“全部”。
- **s.StrCompare(s1, mode):** 串比较函数,比较 s 与 s_1 之间的大小关系,若 $s < s_1$, 返回负数;若 $s == s_1$ 则返回 0;若 $s > s_1$ 则返回正数。 $mode$ 为比较/匹配模式,为下列两种之一:

大小写敏感方式(Sensitive)、大小写不敏感方式(NonSensitive)。

$mode$ 省缺值为“大小写敏感”(下同)。

- **s.StrLen():** 长度函数。返回值为 s 的长度。
- **s.StrConcat(s1, maxNum):** 连接函数。将 s_1 的值连接到 s 的尾部,使 s 的值变为原 s 与 s_1 首尾相接的结果。 $maxNum$ 表示最多复制的字符数目,省缺时表示“全部”。
- **s.StrCharAt(ch, mode, sn):** 字符位置函数。设字符 ch 可能在 s 中出现若干次,则该函数返回 ch 的第 sn 次出现的位置(序号)。 sn 为正数时表示“正向数”($sn=1$ 表示第一个字符); sn 为负数时表示“倒数”($sn=-1$ 表示最后字符)。 $mode$ 为针对 ch 的比较/匹配模式。 sn 缺省值为 1。例如, $s="daefdza2"$, 则 $s.StrCharAr('a', Sensitive, -2)$ 的返回值为 6(倒数第二个 a 的序号为 6)。

• **s.StrStr(s1, mode, sn):** 子串位置函数。返回子串 s_1 在 s 中第 sn 次出现的出现位置(序号)。 sn 含义同上。 $mode$ 同前,是针对在 s 中查找 s_1 的比较/匹配的模式。

• **s.StrSearch(s1, mode, sn, sn2):** 搜索子串函数。在 s 中按 $mode$ 方式,从 s 中的第 sn 起查找 s_1 的第 sn_2 个出现,返回出现位置。这里, sn 的含义与前面的 sn 类似,当 $sn > 0$ 时搜索从左到右,否则从右到左。 sn 省缺值为 1。 $mode$ 同前,是针对于在 s 中查找 s_1 的比较/匹配 s_1 的模式。

• **s.StrSub(sn, num):** 求子串函数。返回 s 中第 sn 字符起的 num 个字符构成的子串的引用(或地址)。 sn 的含义同上。 $num > 0$ 时表示从“左向右”截取(即从 sn 所指起,从序号小的方向向大的方向数 num 字符,作为子串), $num < 0$ 表示“从右向左”截取(即从 sn 所指起,从序号大的方向向小的方向数 num 字符,作为子串)。 $num=0$ 时表示从 sn 起的“全部”,此时,若 $sn > 0$ 则 num 相当于正 0(+0)(从左向右的全部),否则相当负 0(-0)(从右向左的全部)。 num 省缺值为 +0 或 -0(根据 sn 的正负决定)。 sn 省缺值为 1。

• **s.StrReplace(s1, s2, mode, sn, num):** 子串替换函数。在 s 中,用 s_2 的值替 s_1 的从

第 sn 个出现起的 num 个出现。返回实际替换掉的 $s2$ 的个数。 sn 与 num 含义同前。 $mode$ 同前，是针对于在 s 中查找 $s1$ 的比较/匹配 $s1$ 的模式。它们的缺省值也同前。

• **s.StrInsert($s1, sn$)**: 插入函数。在 s 中的第 sn 个字符位置前，插入 $s1$ ，返回 s 引用。 sn 含义同前。

• **s.StrDelete(sn, num)**: 删除函数。将 s 中从 sn 起的 num 个字符删除掉，并返回被删除字符所构成的串。它们的缺省值也同前。

• **s.StrDelete(sel)**: 删除函数。将 s 中的由下标选择器 sel 所指出的各字符删除，并返回被删除字符所构成的串。

• **s.StrTrimL()**: 删除 s 中打头空格，返回 s 。

• **s.StrTrimR()**: 删除 s 中尾随空格，返回 s 。

• **s.StrTrimLR()**: 删除 s 中打头和尾随空格，返回 s 。

• **s.StrTrimAll()**: 删除 s 中所有空格，返回 s 。

• **s.ToUpper()**: 转化为大写。将 s 中所有小写字母转化为大写。返回 s 。

• **s.ToLower()**: 转化为小写。将 s 中所有大写字母转化为小写。返回 s 。

• **s.ToInteger()**: 转化为整数。将 s 转化为整数，若 s 中打头的为非数字（正负号也认作数字），则转化结果为 0，否则，从头起搜索数字，将最左连续出现的所有数字作为整体转化为整数值。返回转化结果。

• **s.ToFloat()**: 转化为浮点数。将 s 转化为浮点数，若 s 中打头的为非数字（正负号及小数点也认作数字），则转化结果为 0，否则，从头起搜索数字，将最左连续出现的所有合法数字作为浮点数转化为整数值。返回转化结果。

• **s.ToString($number, wd, dec$)**: 将 $number$ 所表示的数值转化为字符串，作为 s 的新值。返回 s 。这里， wd 表示转化后的总串长， dec 表示转化后的串中，原 $number$ 的小数部分保留的位数。 dec 大于实际小数位数时，右边补 0，小于时从右边截断。 dec 省缺值为 0， wd 省缺值为 $number$ 的实际长度。

在这些基本运算中，有些并不是最基本的，它们可用其它操作实现。但为了使用方便，也常常将它们列在基本运算中。

§ 4.3.2 串的存贮结构

与线性表类似，串也有两种基本存贮结构：连续与链式。但考虑到存贮效率与算法实现的方便性，串多采用连续存贮结构。

(一) 连续存贮结构

串值存贮在一块连续的存贮区中，这块连续的存贮区一般为动态区，以适合串长度变化范围大的情况。另外，为访问方便，为每个串设立一个描述结点，记录串值存贮区首址与串长度，这种结构类似于线性表的连续结构。

(二) 链式存贮结构

解决可变长串的另一方法是采用链式存贮结构。有两种具体方案：

i)非压缩形式：一个链表结点只存贮一个字符。为了操作方便，描述结点设一个指向首元素结点的指针和一个指向尾元素结点的指针，同时仍设串长度值字段。这种结构仍类似于线性表的链式存储。

这种结构的优点是操作方便，但存贮利用很低。

ii)压缩形式：为了提高存贮利用率，可令一个链结点存贮多个字符。这实质上是一种连续与链式相结合的结构。这种结构对应的操作的实现也很复杂。对改变串长度的操作，可能涉及链结点的增加与删除问题。若要仅允许使不满的结点出现在最后一个结点上，则势必要经常移动字符。但若允许不满结点出现在其它位置，则造成表示的不一致，相当于链表中存贮了多个串，也会使操作实现复杂化。

(三) 串的基本操作实现

本节以连续存贮结构为例，说明定长串的的实现方法。串类的数据部分定义为：

```
class TStr
{
    unsigned char *room;
    long size; //串空间尺寸
    long len; //串当前长
public:
    TStr(long mSize=30);
    TStr(TStr& s);

    ... ..
}
```

至于类 TStr 的完整定义与实现，留作练习。下面将介绍其中的一个很重要的操作----StrStr 的实现。该操作也叫**串模式匹配**。

给定两个串 T 与 P：

$$T = "t_0t_1 \dots t_{n-1}"$$
$$P = "p_0p_1 \dots p_{m-1}"$$

其中 $0 < m \leq n$ ，我们将在 T 中寻找最先出现的一个与 P 相同的子串的过程称为串模式匹配，下称 T 为目标串（主串），P 称为模式串（子串）。前面介绍的 StrStr() 函数就属于串模式匹配。

串模式匹配在符号处理中占有十分重要的地位，它是基于规则匹配的逻辑推理（如 Prolog 语言的目标执行过程、专家系统中的推理机）的基础，所以它的效率显得十分重

要。这里先介绍一种简单的匹配算法，它的时间复杂度较高，然后介绍它的一种改进算法。

1. 简单串模式匹配算法

这是一种带回溯的匹配算法。首先，将子串 P 视为从主串 T 中第 1 个字符起与 T 对齐，从头起依次比较对应字符，若全部对应相等，则表明已找到匹配，终止，否则，将 P 视为从 T 中第 2 个字符起与 T 对齐，再从头比较对应字符，过程与前类似，如此进行，直至某次找到了匹配（成功），或某次 T 中无足够的剩余字符与 P 对齐（不能匹配，失败）。

实现时，设三个指示器 i , j , ii ，它们的含意是：

i ——指向主串 T 中当前参加比较的字符。起始时，指向 T 的首字符，此后，每比较一次，后移一步，一趟匹配失败时，回溯到该趟比较起点的下一位置。

j ——指向子串 P 中当前参加比较的字符。起始时，指向 T 的首字符，每比较一次，后移一步，一趟匹配失败时，回溯到 P 的首字符处。

ii ——记录每趟比较的在主串 T 中的起点，每趟比较后，后移一步。

具体算法的实现，见下面的程序。

```
long TStr::StrStr(TStr& s)
```

```
{//子串匹配。在对象串中寻找子串 s 的第 1 次出现的出现位置序号，无出现时返回-1
```

```
long i, j, ii;
```

```
i=j=ii=0;
```

```
if (len < s.len) return -1; //若子串长度大于主串长度，则不能匹配
```

```
do
```

```
{
```

```
if (room[i] == s.room[j]) //当前字符相等时推进
```

```
{ i++; j++; }
```

```
else
```

```
{//当前字符不等时回溯
```

```
ii++;
```

```
i = ii;
```

```
j=0;
```

```
if (len - ii < s.len) return -1;
```

```
//如果主串中后面尚未比较的的字符个数小于子串长度，则不能匹配，失败返回
```

```
}
```

```
} while (j<s.len);
```

```
return ii; //成功（子串中全部字符都与主串中当前对应比较的相等，则匹配），返回子串位置
```

```
}
```

下面分析一下此算法的时间复杂度。显然，此算法的主要工作是 `do` 循环，它的循

环次数与目标串及模式串相关，所以时间复杂度是个随机量。最理想的情况是第一趟就得到匹配，此时的循环次数为 n （即子串 s 的长度），而最坏的情况是不存在匹配，且每趟匹配过程都是在比较完子串 s 中最后一个字符时才发现不能匹配，此种情况的循环次数是 $(m-n+1) \cdot n$ ，这里 m 为主串的长度。显然， n 越大，此式的值越小。当 $n \ll m$ 时，此式约等于 $m \cdot n$ 。

2. 无回溯的匹配算法

在上面介绍的匹配算法中，某趟匹配失败时，下一趟的匹配相当于将子串 P 后移一位后从头与主串中对应字符比较，即相当于 i 指示器回溯到上趟（最近失败的一趟）匹配的起点的下一个位置，这样，主串中每个字符都要与子串中第 1 个字符对应一次，向后比较，因此，主串中每个字符参加比较的次数最多可达 n 次（ n 为子串长度），因此时间复杂度为 $O(n \cdot m)$ 。

那么，能否使目标串中每个字符只参加一次比较呢？也就是说能否不回溯 i 指示器？回答是肯定的。这个问题是由 D.E.Knoth 与 V.R.Pratt 和 J.H.Morris 同时解决的，所以有的文献也称这种思想的串匹配算法为 KMP 算法。

i 指示器之所以可以不再回溯，是因为重新开始一趟匹配时，可以利用上趟匹配的结果。一般而言，要找到一个无回溯匹配法，关键问题是，在匹配过程中，一旦发现 p_j 与 t_i 不等，应能确定出从 p 中的哪个字符起与 t_i （或 t_{i+1} ）对齐继续往下比较，我们记这个字符为 p_k ，显然 $k < j$ ，且对不同的 i ， k 也不同。Knoth 等人发现这个 k 值仅仅依赖于子串 P 的前 i 个字符的构成，而与主串 T 无关，这是个令人鼓舞的发现，它是实现无回溯的关键。

我们用一个函数 $next$ 表示 j 与 k 的这种对应关系，它是个仅与子串有关的函数，它的含意是，当在匹配过程中一旦发现一对不等的字符（设为 p_j 与 t_i ），则若 $next(j)=k(k \geq 0)$ ，则下次的比较应从 p 中的 p_k 起与 T 中的 t_i 对齐往下进行；若 $next(j)<0$ ，则 p 中任何字符不必再与 t_i 比较，下次比较从 p 的首字符起与 t_{i+1} 对齐往下进行。

$next(j)$ 的取值，应首先保证使匹配过程无遗漏（即不放过任何可能的匹配），其次应使 p 串向后滑动尽可能大的距离。函数 $next()$ 称为失败函数。

下面假定对给定的子串，它的 $next$ 函数已确立，则串模式匹配算法可描述为下列程序的形式，这就是 KMP 算法的基本部分。程序中，用一维数组 $next[]$ 表示失败函数 $next$ 。程序中的 i 变量只增不减，它的初值为 0，在循环中一直保持 $i < n$ ，所以 i 加 1 的语句 $i++$ 至多执行 n 次，又 $next[j] < j$ ，且 $1 \leq j \leq m$ ，故语句 $j=next[j]$ 至多执行 m 次，由于循环中只存在分别含 $i++$ 与 $j=next[j]$ 的两个互斥分支，故循环次数不超过 $MAX(m,n)$ ，即此算法的时间复杂度为 $O(MAX(m,n))$ ，通常 $m < n$ ，故时间复杂度为 $O(n)$ 。

```
int StrStr(TStr T, TStr P)
{
    int i, j;
    int *next;
```



```
if (P.len > T.len) return -1;
next = BuildNext(P); //为 P 串构造失败函数对应的数组 next

i = j = 0; //假定串中首字符的下标为 0
do
{
    if (T.room[i] == P.room[j])
        { i++; j++; }
    else
    {
        if (next[j] > 0) j = next[j]-1; //下次，子串 P 的位置 j 与主串中当前位置 i 对齐比较
        else { j = 0; i++; } //下次，子串的首字符与主串的当前比较位置的下个位置对齐比较
        if (T.len - i < p.len - j) return -1; //不能继续匹配，失败返回
    }
} while (j < P.len);
return i-P.len; //成功，返回出现位置
}
```

关于失败函数的构造，这里就不讨论了。

本章小结

本章重点介绍了三种特殊线性表----栈、队列、串。

栈是先进后出(FILO)结构，属于操作受限的线性表，它限定元素的进出只在表同一端进行。在实际中，凡是对元素的保存次序与使用次序相反的情况，都可以使用栈。栈的主要基本操作是进栈和出栈。栈的存储结构及实现都与线性表类似。

队列是先进先出(FIFO)结构，也属于操作受限的线性表，它限定元素的进出只能分别在表的两端进行。在实际中，凡是对元素的保存次序与使用相同的情况，都可以使用队列。队列的主要基本操作是进队和出队。对的存储结构及实现也都与线性表类似。

串是内容受限的线性表，它限定表中元素是字符。串一般都采用顺序存储结构。提供合理、灵活、方便的基本操作，对串尤为重要。

习 题

1 设有 4 个元素 1、2、3、4 依次进栈，而栈的出栈操作可随时进行（进出栈可任意交错进行，但要保证进栈次序不破坏 1、2、3、4 的相对次序），请写出所有的不可能的出栈次序和所有的可能出栈次序。

2 证明，若借助栈由输入序列 $(1, 2, \dots, n)$ ，得到输出序列 (p_1, p_2, \dots, p_n) ，那么，在输出序列中，不可能出现这种情况：存在 $i < j < k$ ，使 $p_j < p_k < p_i$ 。这里 (p_1, p_2, \dots, p_n) 是 $(1, 2, \dots, n)$ 的一个全排列，每个元素只能按 $1, 2, \dots, n$ 的次序进一次栈。

3 根据习题 4.2，编写程序，打印出与进栈序列 $(1, 2, \dots, n)$ 对应各种可能的出栈序列。

4 假定链式队列分别为下列三种情况，请写出相应的进队和出队程序。i) 带头结点链表；ii) 循环链表；iii) 带头结点的循环链表。

5 实现类 TStr(包括所列出的各个操作)。