第 3 章　栈 和 队 列

栈和队列是两种限定性的数据结构，在普通线性表中，对于表中元素结点的插入、删除等操作不受元素所在位置的限制，而栈和队列上对于元素结点的插入、删除操作会受位置约束。对栈来说，插入和删除运算均是在栈的尾部进行的，而对队列来说，插入在队列的头部进行，而删除在队列的尾部进行。本章主要介绍栈和队列的定义、表示方法和实现等，最后结合实际应用给出了实例。

3.1　栈

栈是非常重要的线性表，在算法设计中应用非常广泛。

3.1.1　栈的定义

栈(Stack)是仅限定在表的一端进行插入和删除的线性表。对栈来说，允许进行插入和删除的一端称为栈顶(top)，不允许插入和删除的另一端则称为栈底(bottom)。对于栈的操作一般只有进栈和出栈两种。

设给定栈 $S=(a_1,a_2,\cdots,a_n)$，则称 a_1 为栈底元素，a_n 为栈顶元素。栈底元素 a_1 是最先插入(进栈)的元素，又是最后一个被删除(退栈)的元素；栈顶元素 a_n 是最后插入(进栈)的元素，又是最先被删除(退栈)的元素。也就是说，退栈时，最后进栈的元素最先出栈，最先进栈的元素最后出栈。由此可见，栈的修改操作是按“后进先出”的原则进行的，如图 3-1 所示。因此栈又被称为后进先出(Last In First Out，简称 LIFO)的线性表。

1. 对栈的运算

(1) CREAT(S)：建立一个空栈；

(2) PUSH(S,x)：在栈中加入一个新元素 x；

(3) POP(S)：删除栈 S 的栈顶元素；

(4) GETTOP(S)：读栈 S 中的栈顶元素；

(5) EMPTY(S)：测试栈 S 是否为空。

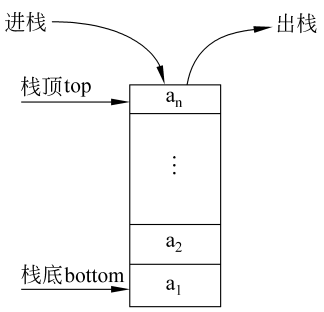


图 3-1　栈的示意图

2. 栈的存储表示方式

1) 静态的数组表示

栈的顺序存储结构是指用一个固定大小的数组来表

示栈，它的好处就是以任何语言处理都相当方便；不利之处就是数组的大小是固定的，而栈本身是变动的。如果进出栈的数据量无法确定，就很难确定数组的大小，要是数组声明得太大就容易造成内存资源浪费，声明太小就会造成栈不够使用。

2）动态的链表表示

使用链表的结构来表示栈，因为链表的声明是动态的，可随时改变链表的长度，这就不会存在静态数组表示时存在的问题了，可以有效地利用内存资源，但缺点是处理复杂。

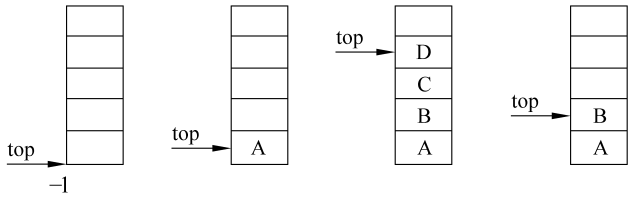
3.1.2　栈的顺序存储结构

由于栈是一种含有特殊约束的线性表，其存储结构都可以是线性表的存储结构。因此，栈也有两种存储表示方法：顺序存储和链式存储。

栈的顺序存储结构简称顺序栈，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，同时附设指针 top 指示栈顶元素在顺序栈中的位置。用一维数组表示栈的顺序存储结构。通常以 top＝0 表示空栈，由于 C 语言中数组的下标约定从 0 开始，因此用 C 作描述语言时，这样设定会带来很大不便，因此用 top＝－1 表示栈空。顺序栈数据结构可表示为：

```
Typedef struct
{
    int  stacksize;
    Selemtype * bottom;
    SelemType * top;
}SqlStack;                          /* 顺序栈类型定义 */
Sqlstack * S;                       /* S 是顺序栈类型指针 */
```

其中，stacksize 是指栈当前可使用的最大容量。栈的初始化操作为按设定的初始分配量进行第一次存储分配。称 top 为栈顶指针，其初值指向栈底，即 top＝－1 可以作为栈空的标记，每当插入新的栈顶元素时，指针 top 增 1，删除栈顶元素时，指针 top 减 1。图 3-2 展示了顺序栈中数据元素和栈顶指针之间的对应关系。



(a) 空栈　(b) 插入A　(c) 插入B，C，D　(d) 删除D，C

图 3-2　栈顶指针和栈中元素之间的关系

用静态数组实现栈结构，栈可表示为如图 3-3 所示的结构。

约定 top 指向真正的栈顶位置下面一个空单元，即新数据将要插入的位置。

以下是数组表示的顺序栈的模块说明。

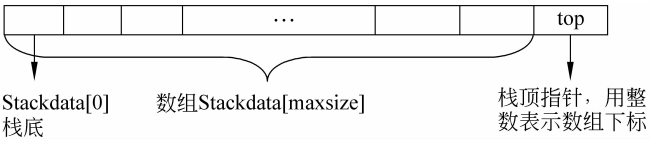


图 3-3 栈的数组表示

```
/* ADT stack 的表示与实现 */
/* 栈的顺序存储表示 */
#define maxsize 64                          /* 栈的最大容量 */
typedef datatype int;                       /* 栈元素的数据类型 */
typedef struct
{
  datatype data[maxsize];
  int top;
} seqstack;                                 /* 顺序栈定义 */
/* 顺序栈的实现 */
seqstack *s;
```

1. 置空栈(栈的初始化)操作

```
initstack(s)
seqstack *s;
  {
   s->top=-1;
   }
```

2. 判栈空操作

```
int empty(s)
seqstack *s;
{
    if(s->top=s->bottom)
        return false;
    else
        return true;
}
```

3. 进栈操作

```
seqstack *Push(s,x)                         /* 将元素 x 插入顺序栈 s 的顶部 */
seqstack *s;
datatype x;
{
    if (s->top==maxsize-1)
    {
```

```
        printf("overflow");
        return NULL;
    }
    else
    {
    s->top++;
    s->data[s->top]=x;
    }
    return s;
}
```

4. 出栈操作

```
Datatype Pop(s,e)                          /*若栈非空,删除栈顶元素,用 e 返回其值*/
seqstack *s;
{
    if (empty(s))
        {printf("underflow");return NULL;/*下溢*/ }
    else
    {
        s->top--;
        e=s->data[s->top+1];
        return(e);
    }
}
```

5. 取栈顶操作

```
Datatype GetTop(s)                         /*取顺序栈 s 的栈顶*/
seqstack *s;
{
    if (empty(s))
    {
      printf("stack is empty");            /*空栈*/
      return NULL;
    }
  else
        return(s->data[s->top]);
}
```

在使用一个数组作为栈的存储结构时,为了保证栈不会溢出,数组中一般都会有剩余的预留存储空间。多个栈共同操作时,如果给每个栈都定义一个数组,容易造成空间的浪费,而且有时会出现一个栈上溢,而另一个栈剩余很多空间的情况。为了合理地使用这些存储单元,可以采用将多个栈存储于同一数组中的方法,即多栈共享空间。

针对两个栈共享一个一维数组 S[0,…,MAXSIZE−1]存储空间的情况,根据栈操作

的特点，可使第一个栈使用数组空间的前面部分，并使栈底在前；而使第二个栈使用数组空间的后面部分，并使栈底在后。其空间分配示意图如图 3-4 所示。

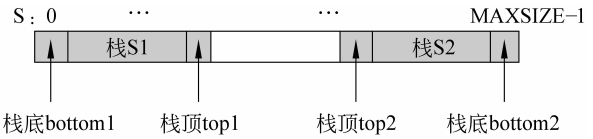


图 3-4　两栈共享空间示意图

这样处理可以使两个栈最大限度地使用数组空间。下面给出这种共享空间存储结构的定义。

```
typedef datatype int;                          /*栈元素的数据类型*/
#define maxsize 64                             /* 栈的最大容量*/
typedef struct
{
  datatype data[maxsize];
  int top1,top2;
}dstack;
```

设栈 s1 和 s2 共享空间 s，s1 从前向后存放，s2 从后向前存放，top1 和 top2 分别是 s1 和 s2 的栈顶指针，则在这种存储结构中初始化、进栈和出栈操作算法如下。

1. 初始化操作

```
InitDstack(dstack *s)
{
    s->top1=-1;
    s->top2=maxsize;
}
```

2. 进栈操作

```
PushDstack(dstack*s,char ch,datatype x)
{                                              /*把数据元素 x 压入栈 s 的左栈或右栈*/
    if (s->top2-s->top1==1) return 0;          /*栈已满*/
    if(ch=="s1")
    {   s->top1=s->top1+1;
        s->data[s->top1]=x;
        return 1;
    }                                          /*进栈 s1*/
    if(ch=="s2")
    {   s->top2=s->top2-1;
        s->data[s->top2]=x;
        return 1;
    }                                          /*进栈 s2*/
}
```

3. 出栈操作

```
popdstack(dstack * s,char ch)
{                                          /* 从栈 S1 或 S2 取出栈顶元素并返回其值 */
    if (ch=="s1")
    {
      if(s->top1<0)
        return NULL;                       /* 栈 s1 已空 */
      else
        {
        s->top1=s->top1-1;
        return(s->data[s->top1]);
        }
    }                                      /* s1 出栈 */
    if(ch=="s2")
    {
    if(s->top2>maxsize-1)
        return NULL;                       /* 栈 s2 已空 */
    else
      {
        s->top2=s->top2+1;
        return (s->data[s->top2]);
      }
    }                                      /* s2 出栈 */
}
```

关于三个以上的栈共享一个数组空间的情况，由于可能需要移动中间某个栈在数组中的相对位置，处理不方便，一般不采用。

3.1.3　栈的链式存储结构

栈的链式存储结构也称做链栈。它是运算受到限制的单链表，其插入和删除操作也仅限于在表头位置上进行。由于只能在链表的头部进行操作，所以不必再附加头结点，链栈的栈顶指针就是链表的头指针。

链栈是单链表的特例，所以其类型和变量的说明和单链表一样。

```
Typedef datatype int;
Typedef struct node
{
datatype data;
struct node   * next;
}linkstack;                                /* 链栈结点类型 */
linkstack * top;
```

top是栈顶指针，它唯一地确定一个链栈。当top＝NULL时，该链栈是空栈。链栈的示意图见图3-5。

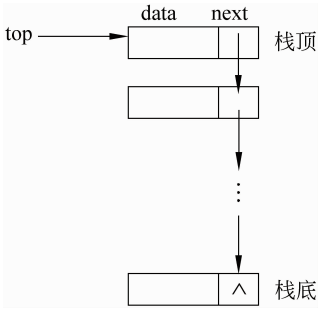


图3-5　链栈示意图

下面给出链栈的进栈和出栈的算法。

1. 进栈操作

当需将一个新元素w插入链栈时，可动态地向系统申请一个结点p的存储空间，将新元素w写入新结点p的数据域，将栈顶指针top的值写入p结点的指针域，使原栈顶结点成为新结点p的直接后继结点，栈顶指针top改为指向p结点。

```
Linkstack pushlinkstack(top,w)
                                        /*将元素 w 插入链栈 top 的栈顶*/
Linkstack *top;
Datatype X;
{
linkstack *p;
p=malloc(sizeof(linkstack));            /*生成新结点*p */
p->data=w;
p->next=top;
return p;                               /*返回新栈顶指针*/
}/*pushlinkstack*/
```

2. 出栈操作

当栈顶元素出栈时，先取出栈顶元素的值，将栈顶指针top指向top结点的直接后继结点，释放原栈顶结点。

```
linkstack poplinkstack(top,x)           /*删除链栈 top 的栈顶结点*/
linkstack top;
datatype *x;                            /*让 x 指向栈顶结点的值，返回新栈指针*/
{
    linkstack *p;
    if (top==NULL)
      {  printf("空栈，下溢");return NULL; }
    else
        {
        *x=top->data;                   /*将栈顶数据存入*x */
        p=top;                          /*保存栈顶结点地址*/
        top=top->next;                  /*删除原栈顶结点*/
        free(p);                        /*释放原栈顶结点*/
        return top;                     /*返回新栈顶指针*/
        }
}/*poplinkstack*/
```

3. 置栈空

```
Void InitStack(LinkStack * S)
{
    S->top=NULL;
}
```

4. 判栈空

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

5. 取栈顶元素

```
datatype StackTop(LinkStack * S)
{
    if(StackEmpty(S))
        Error("Stack is empty.")
    return S->top->data;
```

3.1.4　顺序栈和链式栈的比较

实现顺序栈和链式栈的所有操作的时间相同，但初始化一个顺序栈必须首先声明一个固定长度，这样在栈不满时，就浪费了一部分存储空间，而链式栈无此问题。

可以利用顺序栈的这一特性同时实现两个栈，使用一个数组存储两个栈，每个栈从各自的端点向中间延伸，这样就节约了存储空间。只有当整个向量空间被两个栈占满（即两个栈顶相遇）时，才会发生上溢。因此，两个栈共享一个长度为 m 的向量空间和两个栈分别占用两个长度为 $\llcorner m/2 \lrcorner$ 和 $\ulcorner m/2 \urcorner$ 的向量空间比较，前者发生上溢的概率比后者要小得多，如图 3-6 所示。

图 3-6　用一个数组实现两个栈

3.2　栈的应用

栈是应用非常广泛的一种数据结构，是程序设计中一个重要的工具。

3.2.1　迷宫问题

迷宫问题是一个古老的游戏，要在迷宫中找到出口，通常采用“穷举求解”的方法，经

过一连串的错误尝试才能找到正确的路径。利用栈来解决迷宫问题是一个比较常见的解决方案，同时迷宫问题的求解也是一个非常具有代表性的栈应用实例。

解决迷宫问题，面对的第一个问题是迷宫的表示方法。可以用一个二维数组 maze[max_row][max_col]来表示一个迷宫，其中数组元素 0 代表可通行的路径，1 代表障碍。另外，定义左上角为入口，右下角为出口。图 3-7 表示将一个 5×5 阶的迷宫表示成二维数组方法。

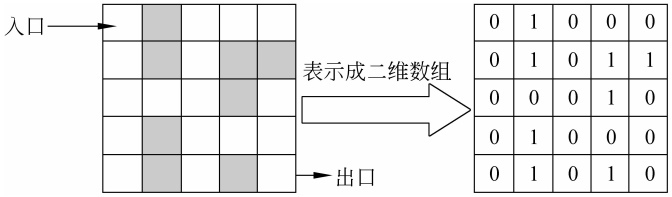


图 3-7　迷宫问题的数组表示

现在的位置为(i,j)，则下一步有上、下、左、右四种走法，如图 3-8 所示。

假设出发顺序是向上、向下、向左、向右，每个结点都是按照此优先顺序来决定下一个结点的方向。用一个数组来存放这四个可能的移动方向，声明如下。

	(i–1, j)	
(i, j–1)	(i, j)	(i, j+1)
	(i+1, j)	

图 3-8　四种可能移动方向

```
Typedef struct
{
int vert;
int horiz;
}offsets;
offsets move[4];
```

其中 move 数组中依据方向的优先顺序依次存入垂直和水平的移动位置，只要以一个整型变量 direct 从 0～3 作为 move 数组的索引值，就可以知道其移动方向，如下所示。

方向索引值(dir)	Move[direct]. vert	Move[direct]. horiz	方向
0	1	0	上
1	−1	0	下
2	0	−1	左
3	0	1	右

因此假设现在的位置在 maze[row][col]，而下一步的位置在 maze[next_row][next_col]，则可利用 direct 变量知道 next_row 和 next_col 两个值：

```
next_row=row+move[direct].vert;
next_col=col+move[direct].horiz;
```

但现在的位置(i,j)走下一步之前，应该事先知道下一步是不是可以走。如果新位置

的数组值为0,就表示可以走,为1就表示不可以走。在判断下一步是不是可以走之前,还要判断是不是已经走出迷宫了。为了方便起见,假设迷宫的四周是被障碍所包围,即将迷宫数组事先用1包围起来。这样对于一个max_row × max_col的迷宫,就需要一个(max_row+2)×(max_col+2)的矩阵来表示迷宫。入口位置在maze[1][1],出口位置在maze[max_row][max_col],迷宫的数组表示如图3-9所示。

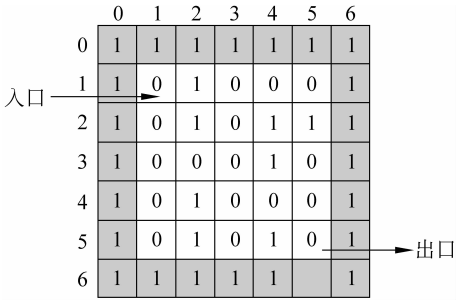


图3-9　迷宫的数组表示

根据上述原则,如果新的位置没有障碍,则将这个位置记录下来,反之,则测试下一个方向,如果每一个方向都走不通,则将此结点删除。由此可见,由起点走到终点是经过一连串的测试错误后,才找到通路的,这种查找方法称为试错法。

而一步步退回的技巧称为回溯,回溯的基本原理是当从某一点开始,找不到一组解时,就回到前一个结点,再重新从这点出发去测试下一个未尝试的可能路径,因此使用回溯一定要记录所走过的路径,才有可能回到前一点,所以需要利用栈来存放所测试过的路径。

在解决迷宫问题时,在一遍遍查找可行路径时,必须用栈的特性记录刚走过的位置,如从(i,j)到(m,n)位置时,就将(i,j)存入栈中,以备没路前进时后退之用;另外,还有一个数据也需要一并存入栈中,就是当从位置(m,n)退回到位置(i,j),要重新查找新路时的新方向d。为了避免由(i,j)到位置(m,n),又由位置(m,n)走回到位置(i,j),凡是走过的结点,其迷宫值由0变为2,在后退时又由2变回0。

迷宫算法如下。

```
void Path ( int maze[ ][col])
{Stack * s;
int  i, j, m, n, d, k, r, c, found;
maze[1][1]=2;                          /* 入口,迷宫值变为2 */
s=NULL;                                /* 将栈置空 */
Push( &s,1,1,2);                       /* 位置和迷宫值进栈 */
found=0;
k=0;
while ( s !=NULL)
{
    Pop(&s, &i, &j, &d);
    while(d<4){                        /* 寻找可走的路径 */
      m=i+move[d].vert;
      n=j+move[d].horiz;
      if(maze[m][n]==0)                /* 可以走 */
      {
        maze[m][n]=2;
```