# 第3章 特殊线性表

CHAPTER

## 3.1 栈

### 3.1.1 栈的定义及其基本操作

**栈**是一种特殊的线性表，它只允许在同一端进行插入和删除操作。允许插入和删除的一端称为**栈顶**，另一端称为**栈底**。处于栈顶位置的元素称为**栈顶元素**。栈中含有元素的个数称为**栈长**，含有 0 个数据元素的栈称为**空栈**。通过栈的定义可以看出，栈的特点是后进先出(LIFO)或称先进后出(FILO)，因此，栈又称为后进先出的线性表。

通过下面例子来说明栈结构的特点，假设有一个很窄的死胡同，胡同里能容纳若干台汽车，胡同的宽度一次只容许一辆车进出。现有 5 台车，编号分别为 A,B,C,D,E。按编号的顺序依次进入死胡同，如图 3.1 所示。此时若编号为 D 的车要退出胡同，必须等编号为 E 的车退出后才可以。若编号为 A 的车要退出胡同，必须等编号为 E,D,C,B 的车依次都退出后才可以。这个死胡同就可以比作一个栈，编号为 A 的车的位置称为栈底，编号为 E 的车的位置称为栈顶，车进出胡同可以看作栈的插入和删除操作，插入和删除操作都在栈顶进行。


图 3.1 死胡同示意图

习惯上，把栈的插入操作称为**入栈**(或称为进栈、压栈)，把栈的删除操作称为**出栈**(或称为退栈、弹栈)。

栈的基本操作主要有以下 7 种：

(1) 初始化操作 initstack(S)，其作用是初始化一个空栈 S。

(2) 求栈长操作 getlen(S)，其作用是返回栈 S 的元素个数，即栈长。

(3) 取栈顶元素操作 gettop(S,x)，其作用是通过 x 带回栈 S 的栈顶元素值。

（4）入栈操作 push(S,x)，其作用是将值为 x 元素压入到栈 S 中，使 x 成为新的栈顶元素。

（5）出栈操作 pop(S,x)，其作用是将非空栈的栈顶元素删除，同时将栈顶元素值赋给 x，新的栈顶元素为栈 S 中原栈顶的下一个元素。

（6）判栈空操作 emptystack(S)，其作用是判断栈 S 是否为空，若栈 S 为空，则返回 1，否则返回 0。

（7）输出栈操作 list(S)，其作用是依次输出栈 S 中的所有元素。

栈是一个线性表，它也有线性表的顺序存储和链式存储两种存储结构，分别称为顺序栈和链栈。下面分别介绍。

### 3.1.2 顺序栈的表示和实现

**1. 顺序栈**

**顺序栈**，即栈的顺序存储结构，利用一组地址连续的存储单元依次存放从栈底到栈顶之间的数据元素，同时利用一个变量记录当前栈顶的位置（下标或指针），称为**栈顶指针**，栈顶指针并不一定是指针变量，也可以是下标变量。为了用 C 语言描述方便，在此约定：用下标变量记录栈顶的位置，栈顶指针始终指向栈顶元素的上一个单元；在初始化栈时，栈顶指针值为 0，表示空栈；在栈中插入新的元素后，栈顶指针增 1；在栈中删除栈顶元素时，栈顶指针减 1。假设用一维数组 S[5]表示一个顺序栈，用变量 top 记录栈顶元素下标，图 3.2 是这个顺序栈的几种状态。

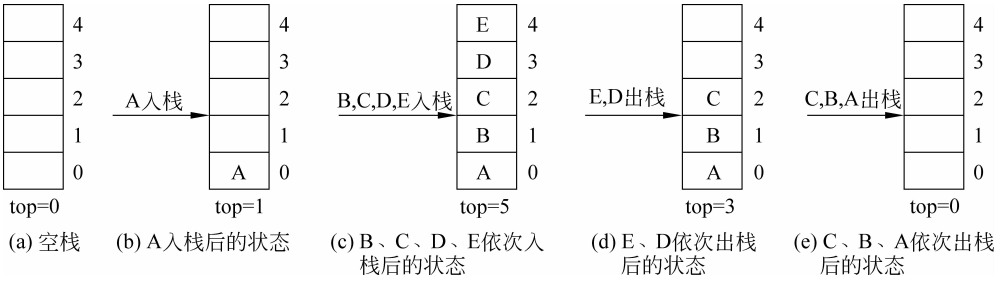


图 3.2 顺序栈 s 的几种状态

图 3.2(a)表示顺序栈为空栈，这也是初始化操作的结果，此时栈顶指针值 top＝0。

图 3.2(b)表示在图 3.2(a)状态下，元素 A 入栈后的状态，此时栈顶指针值 top＝1。

图 3.2(c)表示在图 3.2(b)状态下，元素 B、C、D、E 依次入栈后的状态，此时栈顶指针值 top＝5，表示栈满。

图 3.2(d)表示在图 3.2(c)状态下，元素 E、D 依次出栈后的状态 ，此时栈顶指针值 top＝3。

图 3.2(e)表示在图 3.2(d)状态下，元素 C、B、A 依次出栈后的状态，此时栈顶指针值 top＝0，表示栈空。

由上面的例子可知，在用数组表示栈时，无论数组定义得多么大，栈在使用过程中都有可能会出现栈满的情况，当栈满时，C 语言中定义的数组是无法扩充空间的，因此，在 C 语言中使用动态分配的一维数组，即在初始化时先用函数 malloc()为栈分配一个初始

容量，在操作过程中，若栈的空间不足，再用函数 realloc()重新申请一个足够大的空间。顺序栈的类型定义如下：

```
#define INITSIZE 100     /* 存储空间的初始分配量 */
typedef int ElemType;    /* 在实际应用中，根据需要定义所需的数据类型 */
typedef struct
{ int top;               /* 栈顶指针，也可以定义成地址指针的形式:ElemType *top */
  ElemType *base;        /* 存放元素的动态数组空间 */
  int stacksize;         /* 当前栈空间的大小 */
}sqstack;
```

其中，INITSIZE 是为栈分配的存储空间的初始分配量；base 是栈空间的起始地址，也称为栈底指针，它始终指向栈底的位置；top 是栈顶指针，指向栈顶元素的上一个单元，其初值为 0，作为栈空的标记；stacksize 是当前栈可使用的最大容量。当 top＝＝stacksize 时表示栈满，此时若有元素入栈，需增加存储空间，否则将产生(上)溢出错误。当 top＝＝0 时表示栈空，此时若进行出栈操作，将产生(下)溢出错误。

**2. 基本操作在顺序栈上的实现**

基本操作在顺序栈上的实现如下。

1) 初始化操作(创建一个空栈 S)

算法思路：创建一个空栈，并将栈顶指针 top 初始化为 0。

```
void initstack(sqstack *S)
{ S->base=(ElemType *)malloc(INITSIZE*sizeof(ElemType));
  S->top=0;
  S->stacksize=INITSIZE;
}
```

2) 求栈长操作(返回栈 S 的元素个数，即栈的长度)

```
int getlen(sqstack *S)
{ return (S->top);
}
```

3) 取栈顶元素操作(取出栈 S 的栈顶元素的值)

算法思路：将栈顶元素值存入 e 指向的内存单元，top 值不变。

```
int gettop(sqstack *S,ElemType *e)
{ if(S->top==0) return 0;          /* 栈空，返回 0 */
  *e=S->base[S->top-1];            /* 栈顶元素值存入指针 e 所指向的内存单元 */
  return 1;                        /* 取栈顶成功，返回 1 */
}
```

4) 压栈操作(将值为 x 的数据元素插入到栈 S 中，使之成为新的栈顶元素)

算法思路：将入栈元素 x 存入 top 所指的位置上，然后栈顶指针 top 增 1。

```
int push(sqstack *S,ElemType x)
```

```
{ if(S->top>=S->stacksize)                    /* 若栈满,则增加一个存储单元 */
  { S->base=(ElemType *)realloc(S->base,(S->stacksize+1)*sizeof(ElemType));
    if(!S->base) return 0;                    /* 空间分配不成功,返回 0 */
    S->stacksize++;
  }
  S->base[S->top++]=x;                        /* 插入元素后,栈顶指针上移 */
  return 1;
}
```

5）弹栈操作（取出栈 S 的栈顶元素值，同时栈顶指针下移一格）

算法思路：先将栈顶指针 top 下移一格，再将 top 单元中的元素存入指针 e 所指向的内存单元。

```
int pop(sqstack *S,ElemType *e)
{ if(S->top==0) return 0;                     /* 栈空,返回 0 */
  *e=S->base[--S->top];                       /* 先将指针减 1,再返回栈顶元素值 */
  return 1;                                   /* 弹栈成功,返回 1 */
}
```

6）判栈空操作（判断栈 S 是否为空）

```
int emptystack(sqstack *S)
{ if(S->top==0) return 1;                     /* 栈空,返回 1,否则返回 0 */
  else return 0;
}
```

7）输出栈操作（输出栈 S 自栈顶到栈底的元素值）

```
void list(sqstack *S)
{ int i;
  for(i=S->top-1;i>=0;i--)
    printf("%4d ",S->base[i]);
  printf("\n");
}
```

由于栈结构具有后进先出的固有特性，致使栈成为程序设计中的有用工具。栈在计算机语言处理和将递归算法改为非递归算法等方面起着非常重要的作用。

**【例 3.1】** 编写算法，将十进制正整数 $m$ 转换成 $n(2\leqslant n\leqslant 9)$ 进制数。

算法思路：利用“辗转相除法”，即不断地用 $n$ 去除 $m$，直到 $m$ 为 0 为止，将各次除得的余数倒排。在此，将十进制数 $m$ 除以 $n$ 所得的各位余数入栈，然后依次出栈并输出即可。

```
void conversion(int m,int n)
{ sqstack S;
  initstack(&S);
  while(m!=0)
  { push(&S,m%n); m=m/n;}                     /* 余数入栈,商作为下一个被除数 */
  list(&S);                                   /* 输出 */
}
```

**【例 3.2】** 编写算法，用非递归方法计算 $n!$。

算法思路：将 $n$ 至 1 依次入栈，然后依次出栈并乘到初值为 1 的变量 $f$ 上即可。

```
long fac(int n)
{ long f; ElemType x; sqstack S;
  initstack(&S);
  while(n>0)
  { push(&S,n);n--;}
  f=1;
  while(!emptystack(&S))
  { pop(&S,&x);f*=x;}
  return f;
}
```

**【例 3.3】** 利用一个栈实现下列函数的非递归计算。

$$P_n(x)=\begin{cases}1 & n=0\\ 2x & n=1\\ 2xP_{n-1}(x)-2(n-1)P_{n-2}(x) & n>1\end{cases}$$

算法思路：设置一个栈用于保存 $n$ 和对应的 $P_n(x)$值，栈中相邻元素的 $P_n(x)$有上述关系。然后，边出栈边计算 $P_n(x)$，直到栈空时计算的 $P_n(x)$即为所求。

```
typedef struct
{ int no;                                  /*n值*/
  double val;                              /*P值*/
}ElemType;                                 /*栈中的数据元素为结构体类型 *
double p(int n,double x)
{ double fv1,fv2;
  int i;
  sqstack S;
  initstack(&S);
  for(i=n;i>=2;i--) S.base[S.top++].no=i;
  fv1=1;fv2=2*x;
  while(!emptystack(&S))                   /*若栈不空,则计算各个P值*/
  { S.base[--S.top].val=2*x*fv2-2*(S.base[S.top].no-1)*fv1;
    fv1=fv2;
    fv2=S.base[S.top].val;
  }
  return fv2;                              /*返回最终结果*/
}
```

**【例 3.4】** 设计一个算法，判断一个表达式中括号“(”与“)”、“[”与“]”、“{”与“}”是否匹配。若匹配，则返回 1；否则返回 0。

算法思路：设置一个栈 S，用 i 扫描表达式 exps，当遇到“(”、“[”、“{”时，将其入栈，遇到“}”、“]”、“)”时，判断栈顶是否是相匹配的括号，若不匹配，则表明表达式有误；若表达

式扫描完时 top 的值为 0，则表明表达式括号匹配。

```
typedef char ElemType;                    /* 在此数据元素类型定义为字符型 char */
int match(char *exps)
{ int i=0,nomatch=0;
  sqstack S;
  ElemType x;
  initstack(&S);
  while(!nomatch&&exps[i]!='\0')
  { switch(exps[i])
    { case '(':
      case '[':
      case '{':push(&S,exps[i]);break;
      case ')':gettop(&S,&x);
               if(x=='(') pop(&S,&x);
               else nomatch=1;
               break;
      case ']':gettop(&S,&x);
               if(x=='[') pop(&S,&x);
               else nomatch=1;
               break;
      case '}':gettop(&S,&x);
               if(x=='{') pop(&S,&x);
               else nomatch=1;
    }
    i++;
  }
  if(emptystack(S)&&!nomatch) return 1;
  else return 0;
}
```

## * 3.1.3 链栈的表示和实现

**1. 链栈**

栈的链式存储结构称为**链栈**。链栈实际上是一个仅在表头进行操作的单链表，这种单链表的第 1 个结点称为栈顶结点，最后一个结点称为栈底结点，头指针指向栈顶结点（不带头结点）或头结点（带头结点）。图 3.3 是一个带头结点的链栈示意图。

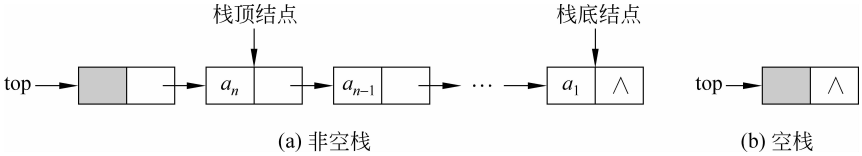


图 3.3 链栈示意图

本节讨论的链栈均指带头结点的链栈。链栈的类型定义如下：

```
typedef int ElemType;
typedef struct node
{ ElemType data;
  struct node *next;
}linkstack;
```

这实质上就是单链表的结点类型定义。

**2. 基本操作在链栈上的实现**

1）初始化操作（创建一个带头结点的空栈 S）

```
linkstack *initstack(void)
{ linkstack *S;
  S=(linkstack *)malloc(sizeof(linkstack));
  S->next=NULL;
  return S;
}
```

2）取栈顶元素操作（取出栈 S 的栈顶元素值）

算法思路：将栈中第 1 个结点的值送到 e 所指向的内存单元，不删除栈顶结点。

```
int gettop(linkstack *S,ElemType *e)
{ if(S->next==NULL) return 0;            /*若栈空,返回 0*/
  *e=S->next->data;                      /*取栈顶成功,返回 1*/
  return 1;
}
```

3）求栈长操作（返回栈 S 的元素个数，即栈的长度）

算法思路：从栈顶开始统计链栈中元素个数，直到栈底为止。

```
int getlen(linkstack *S)
{ linkstack *p; int i;
  p=S->next; i=0;
  while(p!=NULL)
  { i++; p=p->next;}
  return i;
}
```

4）入栈操作（将值为 x 的数据元素插入栈 S 中，使 x 成为新的栈顶元素）

算法思路：先创建一个新结点，其数据域的值为 x，然后将该结点插入到头结点之后作为栈顶结点。

```
int push(linkstack *S,ElemType x)
{ linkstack *p;
  p=(linkstack *)malloc(sizeof(linkstack));
  if(!p) return 0;                       /*若空间分配不成功,返回 0*/
  p->data=x;
```

```
  p->next=S->next;                          /* 插入到头结点之后 */
  S->next=p;
  return 1;
}
```

5）出栈操作（删除栈 S 的栈顶元素）

算法思路：先将栈 S 的栈顶结点的值送到 e 所指向的内存单元，然后删除栈顶结点。

```
int pop(linkstack *S,ElemType *e)
{ linkstack *p;
  if(S->next==NULL) return 0;               /* 栈空，删除失败，返回 0 */
  p=S->next;
  *e=p->data;                               /* 栈顶结点值赋给 e 指针所指向的单元 */
  S->next=p->next;
  free(p);
  return 1;                                 /* 出栈成功，返回 1 */
}
```

6）判栈空操作（判断栈 S 是否为空）

```
int emptystack(linkstack *S)
{ if(S->next==NULL) return 1;               /* 栈空，返回 1，否则返回 0 */
  else return 0;
}
```

7）输出栈（输出栈 S 自栈顶到栈底的元素值）

```
void list(linkstack *S)
{ linkstack *p;
  p=S->next;
  while(p!=NULL)
  { printf("%4d ",p->data);
    p=p->next;
  }
  printf("\n");
}
```

**【例 3.5】** 编写算法，利用栈将带头结点的单链表逆置。

方法一：把单链表 S1 看成一个链栈，把在 S1 中出栈的元素依次入栈 S2，直到 S1 为空为止。此时的 S2 就是 S1 的逆置。

```
void turnlink1(linkstack *S1,linkstack *S2)
{ ElemType x;
  while(S1->next!=NULL)
  { pop(S1,&x);
    push(S2,x);
  }
}
```

方法二：将链表中的元素利用顺序栈交换顺序。

```
void turnlink2(slink *head)
{ sqstack S;
  slink *p;
  initstack(&S);
  p=head->next;
  while(p)
  { push(&S,p->data);p=p->next;}        /*链表中元素依次入栈*/
  p=head->next;
  while(!emptystack(&S))
  { pop(&S,&p->data);p=p->next;}        /*栈中元素依次出栈到链表中对应结点上*/
}
```

当然，该算法也可以使用链栈来实现。

**【例 3.6】** 设计一个算法，判断一个字符串是否对称。若是，则返回 1，否则返回 0。

算法思路：先将长度为 len 的字符串的前半部分(str[0]～str[n/2])入栈，然后用后半部分(str[(len+1)/2]～str[len-1])的字符依次与出栈的元素相比较；不相等时返回 0；若比较完毕且栈为空，则字符串对称。

```
typedef char ElemType;                  /*这里的 ElemType 类型设定为 char*/
int fsame(char *str)
{ ElemType x,same=1;
  int len,i;
  linkstack *S;
  S=initstack();
  for(len=0;str[len]!='\0';len++);      /*计算字符串长*/
  for(i=0;i<len/2;i++)
    push(S,str[i]);                     /*字符串前半部分入栈*/
  for(i=(len+1)/2;i<len;i++)
  { pop(S,&x);
    if(x!=str[i])                       /*对应字符比较*/
    { same=0;break;}
  }
  if(emptystack(S)&&same) return 1;     /*对称返回 1,否则返回 0*/
  else return 0;
}
```

## 3.2　队　　列

### 3.2.1　队列的定义及其基本操作

**队列**也是一种特殊的线性表，它只允许在一端进行插入操作，在另一端进行删除操作。允许插入的一端称为**队尾**，允许删除的一端称为**队头**。新插入的结点只能添加到队

尾，要删除的结点只能是排在队头的结点。在队列中插入结点的操作称为**入队列**，在队列中删除结点的操作称为**出队列**。队列中含有元素的个数称为队列长度，含有 0 个元素的队列称为**空队列**。图 3.4 是一个队列示意图。

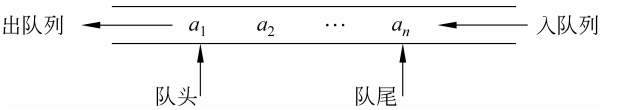


图 3.4　队列示意图

图 3.4 中的数据元素是按照 $a_1$，$a_2$，…，$a_n$ 的顺序进入队列的，出队列的顺序也只能按照这个顺序，也就是说，只有在 $a_1$ 到 $a_{i-1}$（$2 \leqslant i \leqslant n$）都出队列后，$a_i$ 才能出队列。由此可知，队列的特点是先进先出（FIFO）或后进后出（LILO）。因此，队列又称为先进先出的线性表。

队列的基本操作主要有以下 7 种：

（1）初始化操作 initqueue(Q)，其作用是构造一个空队列 Q。

（2）求队列长度操作 getlen(Q)，其作用是返回队列 Q 的元素个数，即队列的长度。

（3）取队头元素操作 getfront(Q,e)，其作用是通过 e 返回队列 Q 的队头元素值。

（4）入队操作 enqueue(Q,x)，其作用是将值为 x 的元素插入到队列 Q 中，使 x 成为新的队尾元素。

（5）出队操作 outqueue(Q,e)，其作用是删除队列 Q 中的队头元素，同时将队头元素值通过 e 带回。原队列中的第 2 个元素成为新的队头元素。

（6）判队空操作 emptyqueue(Q)，其作用是判断队列 Q 是否为空，若队列为空，则返回 1，否则返回 0。

（7）输出队列操作 list(Q)，其作用是从队头到队尾依次输出队列 Q 中的所有元素。

队列也是一个线性表，其存储结构也分为顺序存储和链式存储两种，分别称为顺序队列和链队列。

### 3.2.2　顺序队列的表示和实现

#### 1. 顺序队列和循环队列

**顺序队列**，即队列的顺序存储结构，就是利用一组地址连续的存储单元依次存放从队头到队尾的数据元素，同时利用两个变量分别记录当前队列中队头元素和队尾元素位置，这两个变量分别称为**队头指针**和**队尾指针**。队头指针和队尾指针并不一定是指针变量，也可以是下标变量。在此，用下标变量来描述队列。在初始化空队列时，队头指针和队尾指针的值都为 0；元素入队列后，队尾指针增 1 即可；出队列时，队头指针增 1 即可。因此，在非空队列中，队头指针始终指向队头元素，队尾指针始终指向队尾元素的下一个单元（队尾元素下标＋1 处）。

顺序队列的类型定义如下：

```
#define MAXQSIZE 100
typedef int ElemType;
```