第 3 章

栈 与 队 列

现代计算机编译器领域最杰出的女科学家、图灵奖历史上第一位女性获奖者 Frances E. Allen，作为编译程序组织（compiler organization）和优化算法（optimization algorithms）的先驱，对计算世界做出了开创性的贡献。她说：“编程如登山一样充满挑战。”

的确如此。算法设计与编程本身就是一块充满挑战的神奇之地。如果说具有既定方案的编程需要的是经验、理论和技术能力，那当身处一个全新领域并且面向没有任何既定解决方案的问题时，寻找解决思路与在迷宫中寻找出路无异。需要依靠既有的技能、直觉、经验与规则，在充满障碍与未知的时空中，一步一步地探索走向出口的路。每行进一步，都需要明白自己当下的位置和方向；每一次选择交叉路口，都需要规则与经验；而每一次走到山穷水尽时，都不要忘记，退即是进，予即是得。

无论是现在的学习之路，还是我们之前以及未来的人生之旅，某些时候与迷宫寻路都异曲同工。只要不忘初衷，树立正确的人生观，明白自己当下所走的每一步路，终将寻到一条解决问题的路径。这大概也是探索一些新的未知领域和人生的共同的迷人和奇妙之处。

在计算机领域，迷宫求解离不开栈。本章将对栈以及队列展开阐述，包括它们的逻辑描述、存储实现以及应用。栈和队列作为两种操作受限的线性表，可以分别用于表达式求值、递归问题求解以及银行排队模拟等多个问题中。

3.1 问题的提出

无论是栈还是队列，它们本身具备线性表的逻辑特征。但是对于某些基本操作，如插入和删除，并不像线性表那样，只要位置合法就允许操作，栈和队列会对插入和删除的位置施加一定的限制。本章详细阐述操作受限的线性表即栈和队列的逻辑结构、存储结构和基本操作的实现，并分别讨论如何用栈和队列解决实际问题。

首先，通过下面两个问题进行分析。

问题 1：在高级程序设计 C 语言中，有多种表达式的计算。为了改变表达式中的运算顺序，往往需要添加圆括号，另外数组元素的表示中含有方括号。C 语言的编译器在对表达式进行编译时，一项重要的工作就是检查表达式中的括号是否匹配，例如：(［］())或［(［］［］)］等为正确的格式，［(］)或(()()均为不正确的格式。

编译器是如何检测表达式中括号是否匹配的呢？通常是从左向右对表达式逐个扫描，并对出现的每一个括号进行如下判断：

(1) 如果当前是左括号，则直接保存到左括号序列中。

(2) 如果当前是右括号，则与左括号序列中的最后一个左括号进行比较。即：

① 如果左括号序列中的最后一个左括号是与它不同类型的左括号，则匹配失败，例如：[( ]。

② 如果左括号序列中的最后一个左括号是与它同类型的左括号，则这一对括号匹配成功，并将左括号序列中的最后一个左括号删除。

(3) 如果最后一个括号匹配成功，并且左括号序列中没有剩余的左括号，即括号匹配成功，例如：[([ ] [ ])]。

(4) 如果最后一个括号匹配成功，但左括号序列中还有剩余的左括号，则匹配失败，如[( ) [ ]。

从上述的括号匹配过程中，不难看出，需要对括号组成的序列做如下操作：

(1) 凡是左括号则插入到左括号序列的最后一个左括号之后，简称“进”。

(2) 凡是右括号则查看左括号序列的最后一个左括号，简称“看”。

(3) 凡是右括号与左括号序列的最后一个左括号类型相同则删除，简称“出”。

问题 2：现在各个医院为了更好地满足病人的就医服务，对医院的挂号、分诊、看病、交钱和取药等各个环节实行计算机管理。病人挂完号之后，到对应的科室分诊受理台扫描二维码，即可在滚动的显示屏上显示就诊信息，各个诊室正在看病的排序号以及目前等候的人数。

在病人排队看病过程中，每个病人的挂号数据包括病人的就诊卡号、病人的姓名、专家的姓名和顺序号等，是一个结构体类型的数据元素。

为了让已经挂号的病人及时了解所挂的号在哪个诊室，目前已经看到了第几号，需对挂号候诊的病人做如下处理：

(1) 将已经挂号的病人按选择的专家分别排队。

(2) 正在看病的病人是每个队的队头，简称“看”。

(3) 病人看完病之后走出诊室，简称“出”。

(4) 新挂号的病人排在对应的队尾，简称“进”。

上述问题中的左括号组成的序列和排队就诊的病人组成的序列同属线性结构，都是线性表。与第 2 章的线性表进行对比，不同之处在于：括号匹配中对左括号的插入(进)、对左括号的删除(出)以及查看左括号序列的最后一个括号(看)都限定在左括号序列的同一端进行。医院的挂号分诊处理对候诊病人的插入(进)限定在候诊病人序列的尾部进行；对候诊病人的删除(出)以及查看正在看病的病人(看)则限定在候诊病人序列的首部进行。我们把插入与删除操作只能在一端进行的线性表称为栈，而把插入操作在一端进行、删除操作在另一端进行的线性表称为队列。它们与第 2 章阐述的线性表的插入与删除操作的对比如表 3-1 所示。

表 3-1 插入与删除的对比

操　作	数据结构		
	线性表	栈	队　列
插入	Insert(L, i, x),1≤i≤n+1	Insert(S, n+1, x)	Insert(Q, n+1, x)
删除	Delete(L, i),1≤i≤n	Delete(S, n)	Delete(Q, 1)

其中 n 为线性表的长度。

3.2 栈

3.2.1 栈的定义

栈是一种特殊的线性表，限定插入和删除操作只能在一端进行，具有后进先出（Last In First Out，LIFO）的特点。其中栈顶（top）是允许插入和删除的一端；栈底（bottom）则是不允许插入和删除的一端。

栈结构的示意图如图 3-1 所示。

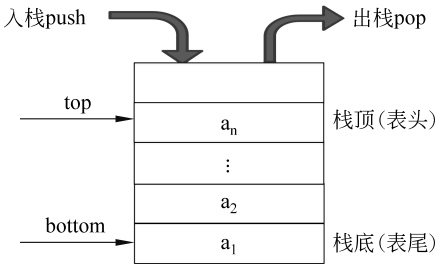


图 3-1 栈的示意图

栈的抽象数据类型描述如下。

```
ADT Stack
{ 数据对象：D={ a_i | a_i∈ElemSet, i=1,2,…,n,  n≥0 }
  数据关系：R={ < a_i-1, a_i> | a_i-1, a_i∈D, i=2,…,n },约定 a_n端为栈顶，a_1端为栈底
  基本操作：
    InitStack(&S);
    DestroyStack(&S);
    StackEmpty(S);
    GetTop(S, &e);
    ClearStack(&S);
    StackLength(S)
    Push(&S, e);
    Pop(&S, &e);
    StackTravers(S, visit());
}ADT Stack
```

(1) 初始化操作：InitStack(&S)；
操作结果：构造一个空栈 S。
(2) 销毁栈结构：DestroyStack(&S)；
已知条件：栈存在。
操作结果：栈 S 被销毁。
(3) 判断栈是否为空：StackEmpty(S)；
已知条件：栈存在。
操作结果：若栈 S 为空栈，则返回 TRUE，否则返回 FALSE。
(4) 获取栈顶：GetTop(S, &e)；
已知条件：栈存在并且非空。
操作结果：用 e 返回 S 的栈顶元素。
(5) 清空栈：ClearStack(&S)；
已知条件：栈存在。
操作结果：将 S 清为空栈。
(6) 求栈的长度：StackLength(S)；
已知条件：栈存在。
操作结果：返回 S 的元素个数，即栈的长度。
(7) 进栈：Push(&S, e)；
初始条件：栈 S 已存在。
操作结果：插入元素 e 为新的栈顶元素。
(8) 出栈：Pop(&S, &e)；
初始条件：栈存在并且非空。
操作结果：用 e 返回删除的栈顶元素。
(9) 遍历：StackTravers(S)；
初始条件：栈存在。
操作结果：访问栈中的全部元素。

3.2.2　栈的顺序存储结构

栈的存储结构主要有顺序栈和链栈两种。栈的顺序存储是指用一片连续的空间存放栈的数据元素，称为顺序栈。

由于对顺序栈的操作限制在栈顶，因此需要已知栈顶的位置。为了防止溢出，需要已知栈的容量。存储数据元素需要一片连续的空间，因此顺序栈是一个结构体类型的变量，用 C 语言描述顺序栈类型有两种方法。

1. 定义顺序栈数据类型方法一

```
#define  MAX  100
typedef struct stack
{   SElemType data[MAX];            //SElemType 是数据元素类型,data 是一维数组
```

```
    int top;                          //指示栈顶的位置
    int stackSize;                    //栈的容量
}SqStack;
```

例如：

```
SqStack S;
```

顺序栈 S 的内存分配示意图如图 3-2 所示。

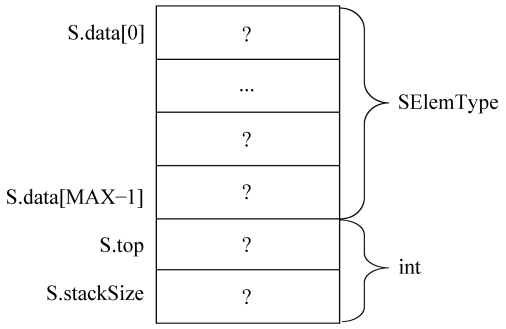


图 3-2　方法一对应的顺序栈内存分配示意图

其中 SqStack 是一个顺序栈类型，用它可以定义顺序栈类型的变量 S，用于存放栈中的数据元素、栈顶的位置和栈的容量。约定栈空时 S.top＝－1。

2. 定义顺序栈数据类型方法二

```
typedef struct stack
{   SElemType *data;                  //data是一个指针变量,存放一片连续空间的首地址
    int top;
    int stackSize;
}SqStack;
```

例如：

```
SqStack S;
```

顺序栈 S 的内存分配示意图如图 3-3 所示。

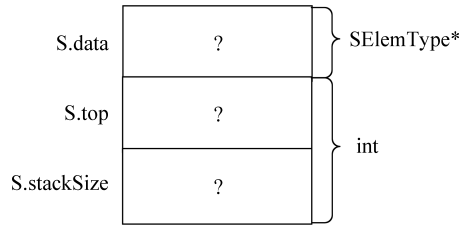


图 3-3　方法二对应的顺序栈内存分配示意图

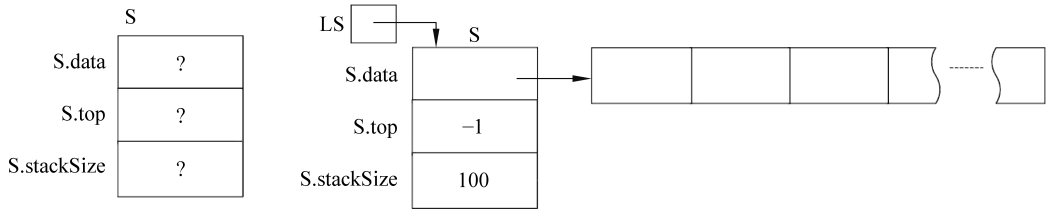
这两种数据类型的区别在于：前者本身包含一个数据成员是数组；后者包含的是一个指针变量，用于指向一片连续的存储空间。连续空间的申请由初始化操作完成。

3.2.3　顺序栈的基本操作实现

下面以顺序栈的第二种说明为例，讲述其基本操作的实现。清楚起见，不妨设栈中的数据元素为整型数据。

1. 顺序栈的初始化操作

分析：初始化操作是给顺序栈类型变量 S 的三个成员赋初值。第一个成员的值是动态申请得到的一片连续空间的首地址；第二个成员的值为－1（表示栈空）；第三个成员的值是申请空间的容量，如图 3-4 所示。



(a) 初始化前　　(b) 初始化后

图 3-4　顺序栈的初始化

初始化操作使得 S 的成员值被改变，申请一片连续的空间需要知道空间的大小。对应的函数应该有两个形参，算法如下。

【算法 3.1】

```
int initSqStack(SqStack * LS,int max)
{   LS->data=( SElemType *)malloc(max*sizeof(SElemType));
    if(LS->data ==NULL){printf("空间申请失败!\n");exit(0);}
    LS->top=-1;   LS->stackSize=max;
    return 1;
}
```

例如：

```
SqStack  S; if(initSqStack(&S,100))printf("初始化成功!\n");
```

2. 顺序栈的判断栈空操作

分析：要想判断栈是否为空，只要判断 S.top 是否为－1 即可，对应的算法如下。

【算法 3.2】

```
int EmptySqStack(SqStack S)
{   if(S.top==-1) return 1;          //栈空
    else return 0;
}
```

例如：

```
if(EmptySqStack(S))printf("栈空!\n");
```

等价于：

```
if(S.top==-1)printf("栈空!\n");
```

3. 顺序栈的获取栈顶元素操作

分析：如果栈为空，不存在栈顶元素；否则栈顶元素的下标是 S.top，栈顶元素是 S.data[S.top]。对应的算法如下。

【算法 3.3】

```
int GetTopSqStack(SqStack S, int *e)
{   if(S.top==-1) return 0;        //栈空
    *e=S.data[S.top];
    return 1;
}
```

例如：

```
int x;
if(GetTopSqStack(S,&x))printf("当前的栈顶是%d\n",x);
else printf("栈空\n");
```

请读者思考，能否直接写成：

```
printf("%d",S.data[S.top]);
```

4. 顺序栈的求长度操作

分析：如果栈为空，将不存在栈顶元素；否则当前栈顶元素的下标是 S.top，S.top＋1 即是栈的长度。对应的算法如下。

【算法 3.4】

```
int LengthSqStack(SqStack S)
{   if(S.top==-1)return 0;
    return S.top+1;
}
```

例如：

```
printf("栈的长度为%d\n", LengthSqStack(S));
```

5. 顺序栈的进栈操作

分析：进栈需要考虑栈满的情况。如果栈不满，则把进栈的元素 e 存放到 S.data[＋＋S.top]中。进栈操作使 S.top 的值加 1，栈 S 发生了变化，如图 3-5 所示。对应的算法如下。

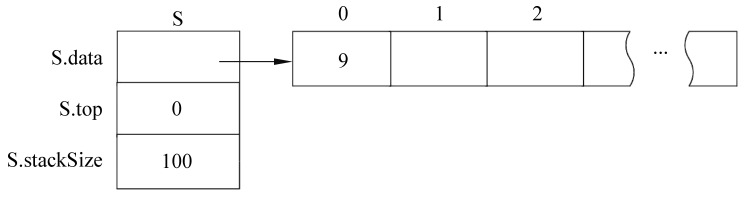
【算法 3.5】

```
int PushSqStack(SqStack *LS,int e)
{   if(LS->stackSize ==LS->top+1)return 0;
```

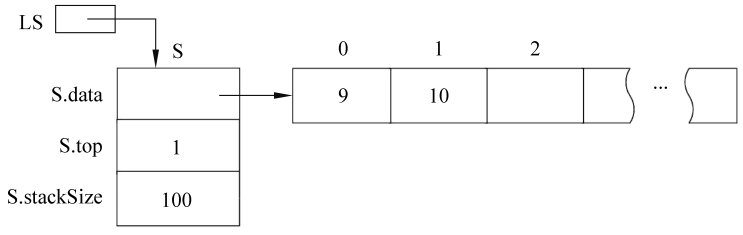
```
    LS->data[++LS->top]=e;
    return 1;
}
```

例如：

```
int x=10; if(PushSqStack(&S,x)==0)printf("栈满!\n");
```



(a) 进栈前

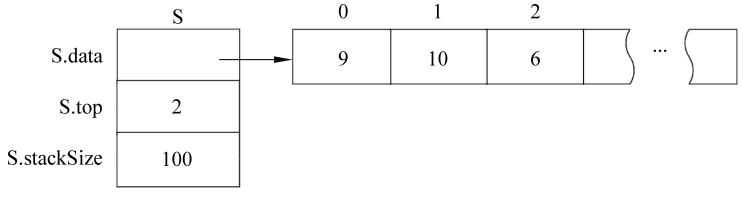


(b) 进栈后

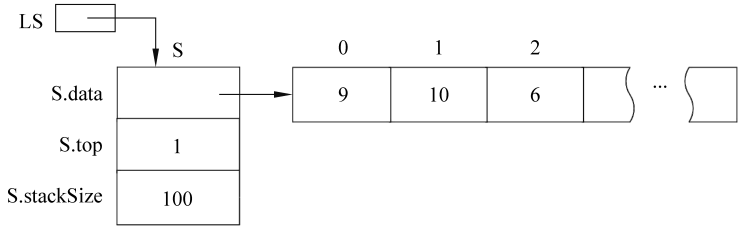
图 3-5　顺序栈的进栈示意图

6. 顺序栈的出栈操作

分析：出栈需要考虑栈为空的情况。如果栈不为空，则将栈顶元素删除并返回。出栈操作使 S.top 的值减 1，栈 S 发生了变化，如图 3-6 所示。对应的算法如下。



(a) 出栈前



(b) 出栈后

图 3-6　顺序栈的出栈示意图

【算法 3.6】

```
int PopSqStack(SqStack *LS,int *e)
{   if(LS->top==-1)return 0;      //栈空
    *e=LS->data[LS->top--];  return 1;
}
```

例如：

```
int x;
if(PopSqStack(&S,&x)==0)printf("栈空!\n");
else  printf("删除的栈顶元素是%d\n",x);
```

7. 顺序栈的遍历操作

分析：遍历需要考虑栈是否为空。如果不为空，则依次从栈顶到栈底访问每个元素。对应的算法如下。

【算法 3.7】

```
int TraversSqStack(SqStack S)
{   int k;
    if(S.top==-1){printf("栈空!\n");return 0;}
    for(k=S.top; k>=0; k--)printf("%5d",S.data[k]);
    printf("\n");
    return 1;
}
```

例如：

```
TraversSqStack(S);
```

有了上述的基本操作，很容易得到其他的操作。如要创建顺序栈，可通过调用初始化操作和进栈操作实现。对应的算法如下。

【算法 3.8】

```
void createSqStack(SqStack *LS,int max)
{   int x,yn;
    initSqStack(LS, max);
    do{   printf("请输入进栈的数据:");  scanf("%d",&x);
          PushSqStack(LS,x);
          printf("继续吗？yes=1,no=0:");
          scanf("%d",&yn);
    } while(yn==1);
}
```

例如：

```
SqStack S; createSqStack(&S,50);
```

3.2.4　栈的链式存储结构

由于栈的操作限定在栈顶，通常采用不带头结点的单向链表存储栈中的数据元素，称为链栈。链栈的存储结构示意图如图 3-7 所示。

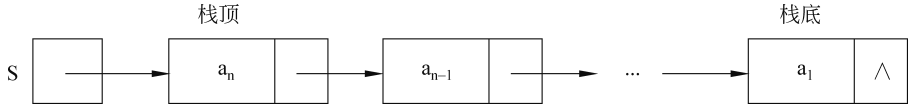


图 3-7　链栈的示意图

链栈的数据类型与线性链表的数据类型描述相同。即：

```
typedef struct snode
{   SElemType data;
    struct snode * next;
}SNode, * LinkStack;
```

3.2.5　链栈的基本操作实现

假定 SElemType 表示整型，有关链栈的基本操作实现如下。

1. 链栈的初始化操作

分析：初始化操作是创建一个不带头结点的空链栈，如图 3-8 所示。

(a) 初始化前　　(b) 初始化后

图 3-8　初始化链栈的示意图

对应的算法如下。

【算法 3.9】

```
void InitLinkStack(LinkStack * LS)
{ * LS=NULL; }
```

例如：

```
LinkStack S;  initLinkStack(&S);
```

等价于：

```
LinkStack S=NULL;
```

说明：在实际编程中，对一些只用一条语句即可完成的操作，如判断栈是否为空以及链栈的初始化等，可以不用编写函数，直接写相应的语句即可。

2. 链栈的判断栈空操作

分析：要确定栈是否为空，只要判断 S 是否为 NULL 即可。对应的算法如下。

【算法 3.10】

```
int EmptyLinkStack(LinkStack S)
{   if(S==NULL)return 1;
    else return 0;
}
```

例如：

```
if(EmptyLinkStack(S))printf("栈空!\n");
```

等价于：

```
if(S==NULL)printf("栈空!\n");
```

3. 链栈的获取栈顶元素操作

分析：因为链栈的头指针指向栈顶，所以栈顶元素是 S->data。如果栈为空，则不存在栈顶元素。对应的算法如下。

【算法 3.11】

```
int GetTopLinkStack(LinkStack S, int *e)
{   if(S==NULL) return 0;          //栈空
    *e=S->data;  return 1;
}
```

例如：

```
int x;
if(GetTopLinkStack (S,&x)==0)printf("栈空!\n");
else printf("当前的栈顶是%d\n",x);
```

4. 链栈的求长度操作

分析：当栈不为空时，依次寻找后继结点，并用累加器求和，得到的结果即是栈的长度。对应的算法如下。

【算法 3.12】

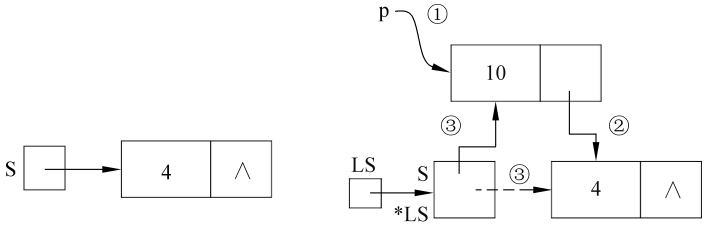
```
int LengthLinkStack(LinkStack S)
{   int n=0;
    while(S){ n++; S=S->next; }
    return n;
}
```

例如：

```
printf("栈的长度为%d\n", LengthLinkStack(S));
```

5. 链栈的进栈操作

分析：进栈的元素将成为新的栈顶元素，并且头指针会记录进栈的元素，它的值也会改变，如图 3-9 所示。



(a) 进栈前　　(b) 进栈后

图 3-9　链栈的进栈示意图

进栈的算法如下。

【算法 3.13】

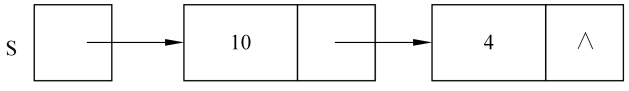
```
int PushLinkStack(LinkStack *LS,int e)
{   LinkStack p=(LinkStack)malloc(sizeof(SNode));  //①
    if(p==NULL) return 0;
    p->data=e; p->next=*LS;                        //②将 p 插入到原来的第 1 个结点之前
    *LS=p;                                         //③
    return 1;
}
```

例如：

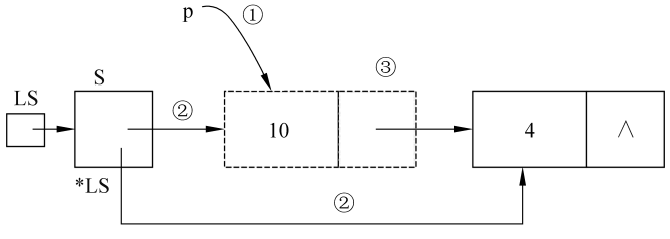
```
int x=10; PushLinkStack(&S,x);
```

6. 链栈的出栈操作

分析：出栈需要考虑栈为空的情况。如果栈不为空，则将栈顶元素删除并返回，头指针指向原来的第二个结点，它的值也会改变，如图 3-10 所示。



(a) 出栈前



(b) 出栈后

图 3-10　链栈的出栈示意图

出栈的算法如下。

【算法 3.14】

```
int PopLinkStack(LinkStack *LS,int *e)
{   LinkStack p=*LS;                          //①p 记录栈顶元素
    if(*LS==NULL)return 0;                    //栈空
    *LS=(*LS)->next;                          //②头指针指向原来的第 2 个结点
    *e=p->data; free(p);                      //③释放被删除结点的存储空间
    return 1;
}
```

例如：

```
int x;
if(PopLinkStack (&S,&x)==0)printf("栈空!\n");
else printf("删除的栈顶元素是%d\n",x);
```

7. 链栈的遍历操作

分析：遍历需要考虑栈是否为空。如果不为空，则依次从栈顶到栈底访问每个元素。对应的算法如下。

【算法 3.15】

```
int TraversLinkStack(LinkStack S)
{   if(S ==NULL){printf("栈空!\n"); return 0;}
    while(S){printf("%5d ",S->data); S=S->next;}
    printf("\n");
    return 1;
}
```

例如：

```
TraversLinkStack(S);
```

有了上述基本操作，很容易得到其他的操作。如要创建链栈，可通过调用初始化操作和进栈操作完成。对应的算法如下。

【算法 3.16】

```
void createLinkStack(LinkStack *LS)
{   int x,yn;
    initLinkStack(LS);
    do{   printf("请输入进栈的数据:");
          scanf("%d",&x);
          PushLinkStack(LS,x);
          printf("继续吗? yes=1,no=0:");
          scanf("%d",&yn);
    }while(yn==1);
}
```

例如：

```
LinkStack S;  createLinkStack(&S);
```

有时为了快速得到链栈中的元素个数，可以用图 3-11 所示的链栈表示。

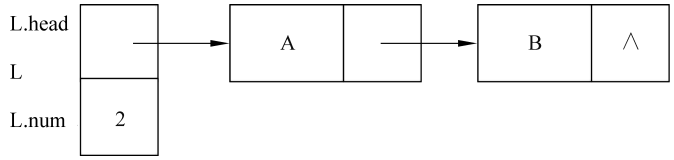


图 3-11　带数据个数的链栈示意图

对应图 3-11 所示链栈的数据类型为：

```
typedef struct node
{   char data;
    struct node * next;
}LNode, * LinkPtr;
typedef struct
{   LinkPtr head;                          //链栈的头指针，存放栈顶元素结点的地址
    int num;                               //存放数据元素个数
}LinkStack;
```

例如：

```
LinkStack L;
```

L 中的成员 L.head 指向链栈的栈顶，L.num 存放的是链栈中的数据元素个数。请读者自行完成对应上述链栈的初始化、进栈和出栈操作的实现。

3.2.6　栈的两种存储结构的区别

顺序栈使用一组连续的空间依序存储各个数据元素，用一个栈顶指针记录当前栈顶元素的位置，并通过栈顶指针判断栈空和栈满以及进栈元素和出栈元素的位置。

链栈是一个不带头结点的单向链表，头指针指向栈顶元素结点，进栈和出栈只需改变头指针。

3.2.7　案例实现：基于栈的括号匹配

括号匹配中对括号的处理符合栈的特点。下面用链栈来解决 3.1 节提出的问题 1 中的括号存储以及其上所需的操作实现。针对实际问题，可以对上述要用到的函数做适当的修改。

由于 C 语言中的表达式只有圆括号和方括号，匹配算法相对简单。为了更好地掌握用栈来判断表达式中括号的匹配问题，不妨假设表达式中允许有圆括号、方括号和花括号。

源程序如下。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//链栈的数据类型描述
typedef struct snode
{   char data;
    struct snode * next;
}SNode, * LinkStack;
//函数的声明
void initLinkStack(LinkStack * );                    //初始化栈
void LinkStackPush(LinkStack * ,char);               //进栈
int LinkStackPop(LinkStack * );                      //出栈
int LinkStackGetTop(LinkStack ,char * );             //获取栈顶元素
void matching(char * );                              //括号匹配
//主函数
void main()
{   char str[80];
    printf("请输入表达式:\n"); gets(str);   //读表达式,允许出现圆括号、方括号和花括号
    matching(str);
}
//有关链栈操作的函数定义省略
......
void matching(char str[])                            //括号匹配
{   LinkStack S; int k, flag=1; char e;
    initLinkStack(&S);                               //创建空栈
    for(k=0; str[k]!='\0' && flag; k++)
    {   if(str[k]!='('&& str[k]!=')'&& str[k]!='['&& str[k]!=']'&&
                    str[k]!='{'&&str[k]!='}')
                    continue;                        //非括号的处理
        switch(str[k])                               //对括号进行配对处理
        {   case '(': case '[': case '{':            //遇左括号进栈
                    PushLinkStack(&S,str[k]); break;
            case ')': //遇右圆括号
                    if(S!=NULL)
                    {   GetTopLinkStack(S,&e);       //获取栈顶元素
                        if(e=='(') PopLinkStack(&S);
                                                     //栈顶是左圆括号,匹配成功
                        else flag=0;                 //栈顶不是左圆括号,匹配失败
                    }
                    else flag=0;                     //栈空,匹配失败
                    break;
            case ']': //遇右方括号
                    if(S!=NULL)
```

```
                {   GetTopLinkStack(S,&e);    //获取栈顶元素
                    if(e=='[')PopLinkStack(&S);
                                              //栈顶是左方括号,匹配成功
                    else flag=0;              //栈顶不是左方括号,匹配失败
                }
                else flag=0;                  //栈空,匹配失败
                break;
        case '}': //遇右花括号
                if(S!=NULL)
                {   GetTopLinkStack(S,&e);    //获取栈顶元素
                    if(e=='{')PopLinkStack(&S);
                                              //栈顶是左花括号,匹配成功
                    else flag=0;              //栈顶不是左花括号,匹配失败
                }
                else flag=0;                  //栈空,匹配失败
                break;
      }//switch
   }//for
   if(flag==1 && S==NULL)printf("括号匹配!\n");
   else printf("括号不匹配!\n");
}
```

3.3　栈的应用

3.3.1　表达式求值

C语言有着丰富的表达式,那么C的编译器是如何处理表达式的呢?本节主要讨论C的简单算术表达式求值。

1. 算术表达式的形式

数学上的算术表达式通常包括操作数和运算符。

- 操作数:简单变量或表达式,用s1、s2表示。
- 运算符:+、-、*、/、(、),用op表示。

通常的算术表达式形式(本书中也称算术表达式)即为数学表达式形式,如3*(5-2)+7。由于运算符的优先级不同,因此求值不一定能够按照从左到右的顺序执行。例如,上述表达式求值顺序是先做减法,接着做乘法,最后做加法。如果能将算术表达式转换成易于从左到右的顺序执行,即可大大提高计算机的执行效率。

算术表达式除了数学上的表达形式外,还有如下三种表达形式。

(1) 中缀表达式(运算符位于两个操作数之间):s1 **op** s2。

(2) 前缀表达式(运算符位于两个操作数之前):**op** s1 s2。

(3) 后缀表达式(运算符位于两个操作数之后):s1 s2 **op**。

以算术表达式3*(5-2)+7为例,下面给出求算术表达式的其他三种表达形式的步

骤。依次处理算术表达式中级别较低的运算符，为了从形式上明确表示运算符的两个操作对象，约定当操作对象是表达式时，用一对花括号将其括起来，结果如下。

• 中缀表达式的处理顺序。

①处理'+'：{3*(5-2)}+7；②处理'*'：3*{5-2}+7；③处理'-'：3*5-2+7。

• 前缀表达式的处理顺序。

①处理'+'：+{3*(5-2)}7；②处理'*'：+*3{(5-2)}7；③处理'-'：+*3-527。

• 后缀表达式的处理顺序。

①处理'+'：{3*(5-2)}7+；②处理'*'：3{(5-2)}*7+；③处理'-'：352-*7+。

不难看出：三种表达式的操作数顺序相同，但运算符顺序不一。其中，中缀表达式丢失了算术表达式中的括号信息，致使运算符的运算顺序不能确定，计算会出现二义性；前缀表达式中的运算规则是连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式，由于运算符的顺序与计算顺序不一致，因此需多次扫描前缀式，才能完成表达式的计算，效率低；后缀表达式中的运算规则是连续出现的两个操作数和在它们之后且紧靠它们的运算符构成一个最小表达式，由于运算符的顺序与计算顺序一致，因此只需一次扫描后缀式，即可完成表达式的计算，效率高。

2. 后缀表达式求值

有了后缀表达式，如何求值呢？

求值过程：后缀表达式是一个字符串，为了方便处理，以'#'结束。用一个栈（假定数据元素类型为整型）来存放操作数和中间的计算结果。对后缀表达式从左向右依次扫描，若是操作数，则将字符转换成整数进栈；若是运算符，则连续出栈两次，第一次出栈的元素是第二个操作数，第二次出栈的元素是第一个操作数，根据当前的运算符做相应的运算，并将计算结果进栈，直到遇到'#'为止。此时栈中只剩下一个元素，即最后的运算结果，出栈即可。

以后缀表达式“3 5 2 - * 7 + #”为例，求值过程如表 3-2 所示。

表 3-2 后缀表达式 3 5 2 - * 7 + # 的求值过程

对后缀表达式“352-*7+#”依次从左向右处理	操作数栈
(1)遇到'3'，将'3'转换成整数 3，进栈	3
(2)遇到'5'，将'5'转换成整数 5，进栈	5 3
(3)遇到'2'，将'2'转换成整数 2，进栈	2 5 3
(4)遇到'-'，从操作数栈连续出栈两次，计算 5-2，将计算结果 3 进栈	3 3
(5)遇到'*'，从操作数栈连续出栈两次，计算 3*3，将计算结果 9 进栈	9
(6)遇到'7'，将'7'转换成整数 7，进栈	7 9
(7)遇到'+'，从操作数栈连续出栈两次，计算 9+7，将计算结果 16 进栈	16
(8)遇到'#'，计算结果出栈，操作数栈为空，算法结束	

对应的算法如下。

【算法 3.17】

```
int suffix_value( char a[] )                          //a 指向后缀表达式
{   int i=0,x1,x2; result; STACK s; init_stack(s); //初始化一个空栈
    while(a[i] !='#')
    {   switch(a[i])
        {   case '+':  x2=pop(s); x1=pop(s); push(s,x1+x2);break;
            case '-':  x2=pop(s); x1=pop(s); push(s,x1-x2);break;
            case '*':  x2=pop(s); x1=pop(s); push(s,x1*x2);break;
            case '/':  x2=pop(s);x1=pop(s);
                       if(x2!=0) push(s, x1/x2);
                       else {printf("分母为 0!\n"); return;}
                       break;
            default:  push(s,a[i]-48);              //将字符转换成整数
        }//switch
        i++;
    } //处理下一个 a[i]
    result=pop(s); return result;
} //suffix_value
```

3. 将算术表达式转换为后缀表达式

为了便于将算术表达式转换成后缀表达式，不妨在算术表达式的末尾增加一个字符'#'，在算术运算符中增加一个'#'运算符。

用一个字符栈来存放运算符。先用'#'初始化字符栈，再对表达式字符串中的每一个字符从左到右依次做如下处理。

(1) 如果当前字符是操作数，则将其存放到后缀表达式数组。

(2) 如果当前字符是运算符，则考虑它是否进栈。

设当前运算符为 op，则

(1) 当 op ＝＝ '('时，op 直接进栈。

(2) 当 op ＝＝ ')'时，栈顶运算符依次出栈，并依次将其按顺序存放到后缀表达式数组，直到遇到'('为止。注意：'('只出栈，不存放到后缀表达式数组。

(3) 当 op 的优先级高于栈顶运算符的优先级时，op 进栈；否则，栈顶的运算符依次出栈，存放到后缀表达式数组，直到栈顶运算符的优先级低于 op，op 进栈。

(4) 当 op＝＝'#'时，栈顶运算符依次出栈，存放到后缀表达式数组，直到栈顶运算符为'#'，算法结束。

设运算符的优先级顺序为＃，(，＋或－，＊或/，从左到右由低到高。

算术表达式 3＊(5－2)＋7 转换成后缀表达式的过程如表 3-3 所示。

表 3-3　算术表达式转换成后缀表达式的过程

对算术表达式“3＊(5－2)＋7＃”从左到右依序处理	运算符栈(顶----底)	后缀表达式
(1)遇到字符'3'：将'3'存入后缀表达式数组	＃	“3”
(2)遇到字符'＊'：'＊'的优先级高于栈顶'＃','＊'进运算符栈	＊ ＃	
(3)遇到字符'('：'('进运算符栈	(＊ ＃	
(4)遇到字符'5'：将'5'存入后缀表达式数组		“35”
(5)遇到字符'－'：'－'的优先级高于栈顶'(','－'进运算符栈	－ (＊ ＃	
(6)遇到字符'2'：将'2'存入后缀表达式数组		“352”
(7)遇到字符')'：将运算符栈的栈顶运算符依次出栈，存入后缀表达式数组，直至遇到'(','('只出栈，不存入后缀表达式数组	＊ ＃	“352－”
(8)遇到字符'＋'：'＋'的优先级低于栈顶'＊','＊'出栈，存入后缀表达式数组；'＋'的优先级高于栈顶'＃','＋'进运算符栈	＋ ＃	“352－＊”
(9)遇到字符'7'：'7'存入后缀表达式数组		“352－＊7”
(10)遇到字符'＃'：将运算符栈的栈顶运算符依次出栈，存入后缀表达式数组，直至栈空，算法结束		“352－＊7＋＃”

判断运算符优先级的函数如下。

```
int prior(char a)                                    //返回运算符 a 的优先级
{   if(a=='*'||a=='/') return 4;
    else if(a=='+'||a=='-') return 3;
    else if(a=='(') return 2;
    else if(a=='#') return 1;
    else return 0;
}
```

将算术表达式转换为后缀表达式的算法如下。

【算法 3.18】

```
void  Transformation(char a[],char suff[])
{   //a 指向算术表达式,以"#"结束,栈用于存放运算符
    //将 a 指向的算术表达式转换为由 suff 指向的后缀表达式
    int i = 0,k=0,n; char ch;
    LinkStack s;
    init_Stack(s); push(s,'#');
    n=strlen(a); a[n]='#'; a[n+1]='\0';              //在表达式的末尾添加一个#
    while (a[i]!='\0')
    {   if (a[i]>='0' && a[i]<='9')suff[k++]=a[i];  //是操作数,直接存入后缀表达式
        else                                         //是运算符
            switch ( a[i] )
            {   case '(':  push(s,a[i]); break;      //进栈
                case ')':  //将左圆括号之上的运算符依次出栈并发送到后缀表达式,左圆
```

```
                        //括号只出栈
                        ch = pop(s);
                        while ( ch!='(')
                        { suff[k++]=ch; ch = pop(s);}
                        break;
    /*比较表达式当前的运算符 a[i]和栈顶运算符 ch 的优先级,如果 a[i]高于 ch,a[i]进栈;反之,栈内高于 a[i]的运算符依次出栈并发往后缀表达式,直到栈顶运算符优先级低,再将 a[i]进栈*/
            default: ch=gettop(s);
                     while(prior(ch)>=prior(a[i]))
                     { suff[k++]=ch; ch=pop(s); ch=gettop(s); }
                     if(a[i]!='#')push(s,a[i]);
        }//end_swicth
      i++;
    }//end_while
    suff[k]='\0';                                    //保证 suff 存放的是字符串
} //Transformation
```

以上算法仅适用于操作数是个位数。要计算任意的实数,需要解决如下问题。

(1) 后缀表达式中的操作数与操作数之间如何隔开?

(2) 操作数栈的元素类型是什么?

(3) 如何将一个数字串转换为一个实数?

(4) 操作数为负数时,如何处理?

例如,算术表达式为-3+(-15.7+9)*4.25+7/8.2。

(1) 先处理负数的情况。

原则:第1个字符为'-',前面加0;'('之后是'-',在'('之后加0。

算术表达式变为0-3+(0-15.7+9)*4.25+7/8.2。

(2) 在操作数与操作数之间加空格。

后缀表达式为0 3 - 0 15.7 -9 + 4.25 * +7 8.2/+。

请读者将上述的有关算法进行修改,使其可以计算任意实数的算术表达式。

4. 算术表达式直接求值

前面介绍的是先将算术表达式转换成后缀表达式,再根据后缀表达式求值。也可由算术表达式直接求值。

算法的主要步骤如下。

(1) 创建两个栈,一个是运算符栈(初始化时,将'#'进栈),另一个是操作数和中间结果栈。

(2) 对算术表达式从左向右依次扫描。

① 如果算术表达式的当前字符是操作数,则将算术表达式的当前字符转换成整数进操作数栈。

② 如果算术表达式的当前字符是运算符,则与运算符栈的栈顶运算符进行比较。