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

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被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

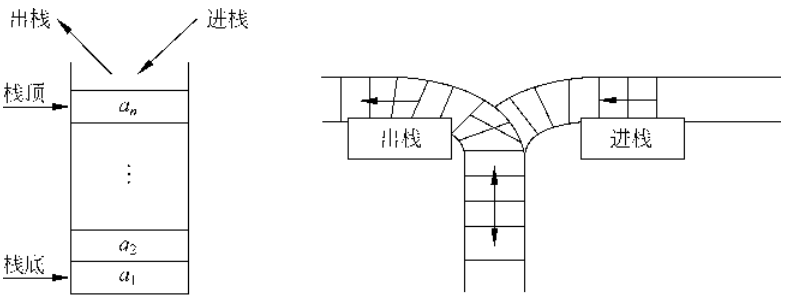
在日常生活中，我们会发现有许多这样的趣事：例如，把许多书籍依次放进一个大小相当的箱子中，当我们在取书时，就得先把后放进里面的书取走，才能拿到先放入的被压在最底层的书；又如一叠洗净的盘子，洗的时候总是将盘子逐个叠放在已洗好的盘子上面，而用的时候则是从上往下逐个取用，即后洗好的盘子比先洗好的盘子先被使用。这种后进先出的结构称为栈。

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)。栈顶的第一个元素称为**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$\cdots,P_j,\cdots,P_k,\cdots,P_i,\cdots(P_j<P_k<P_i)$，则一定不存在这样的输出序列$\cdots,P_i,\cdots,P_j,\cdots,P_k,\cdots$

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,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。

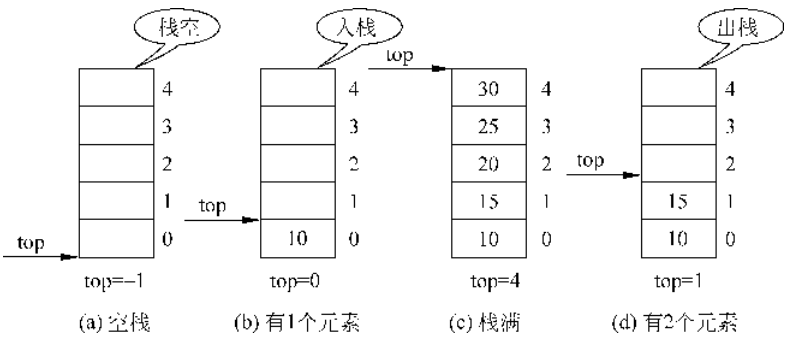


图3.2 栈顶指针和栈中元素之间的关系

通过分析,我们可以得出以下结论:

(1) 若top=-1,则表示栈空;

(2) 若top=maxsize-1,则表示栈满。

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 我们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一个结点，整个单链表称为**链栈**。链栈的表示如图 3.3(a)所示。

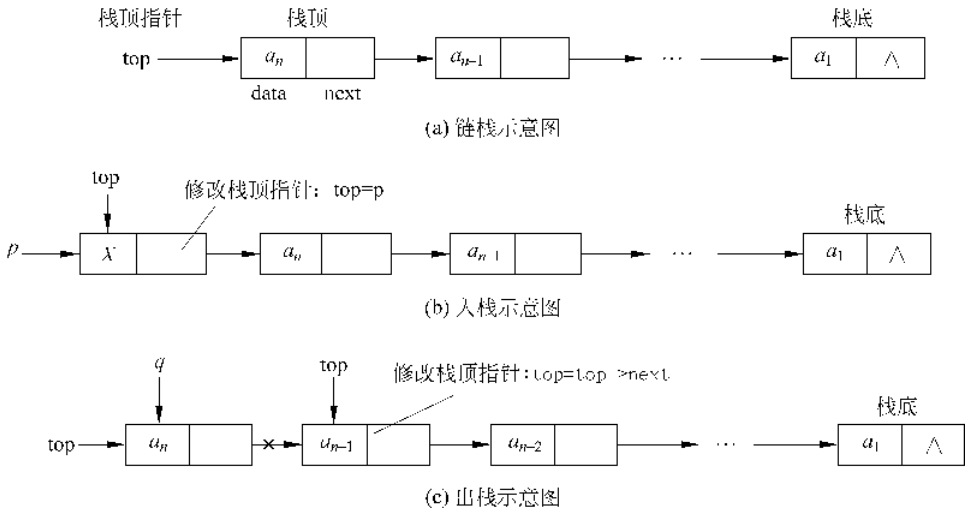


图 3.3 链栈的图示

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 链栈的优点是,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,其求解过程中也必然需要利用栈。

3.2.1 数制转换

例 3.2 将十进制整数转换成二至九之间的任一进制数输出。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。

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。

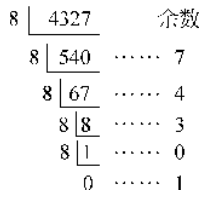


图 3.4　十进制数 4327 转换为八进制数的过程

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三种括号：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(!StackeEmpty(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。依次类推，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。

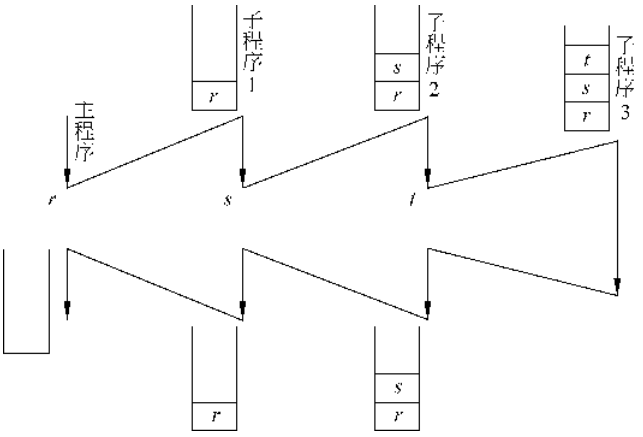


图3.5 栈在子程序嵌套调用中的应用

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。

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。