第3章 栈和队列

3.1 栈

```
// c3-1.h 栈的顺序存储结构。在教科书第 46 页(见图 3-1)
#define STACK_INIT_SIZE 10 // 存储空间初始分配量
#define STACK_INCREMENT 2 // 存储空间分配增量
struct SqStack // 顺序栈
{ SElemType * base; // 在栈构造之前和销毁之后,base 的值为 NULL
  SElemType * top; // 栈顶指针
  int stacksize; // 当前已分配的存储空间,以元素为单位
};
```

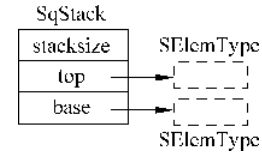


图 3-1 顺序栈存储结构

```
// bo3-1.cpp 顺序栈(存储结构由 c3-1.h 定义)的基本操作(9 个)
void InitStack(SqStack &S)
{ // 构造一个空栈 S。在教科书第 47 页(见图 3-2)
  if(!(S.base = (SElemType * )malloc(STACK_INIT_SIZE * sizeof(SElemType))))
    exit(OVERFLOW); // 动态分配存储空间失败,则退出
  S.top = S.base; // 栈顶指向栈底(空栈)
  S.stacksize = STACK_INIT_SIZE; // 存储空间为初始分配量
}
void DestroyStack(SqStack &S)
{ // 销毁栈 S,S 不再存在(见图 3-3)
  free(S.base); // 释放栈空间
  S.top = S.base = NULL; // 栈顶、栈底指针均为空
  S.stacksize = 0; // 当前已分配的存储空间为 0
}
void ClearStack(SqStack &S)
{ // 把栈 S 置为空栈
  S.top = S.base; // 栈顶指针指向栈底
}
Status StackEmpty(SqStack S)
{ // 若栈 S 为空栈,则返回 TRUE; 否则返回 FALSE
```

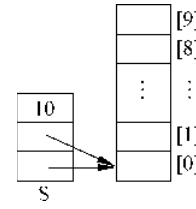


图 3-2 构造一个空的顺序栈 S

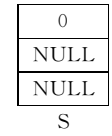
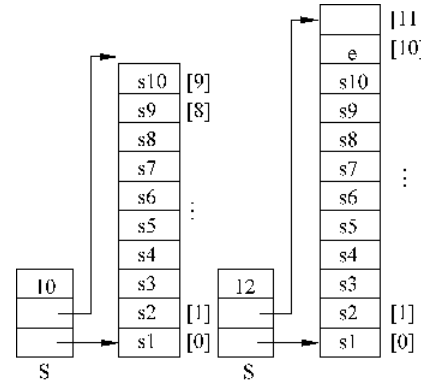


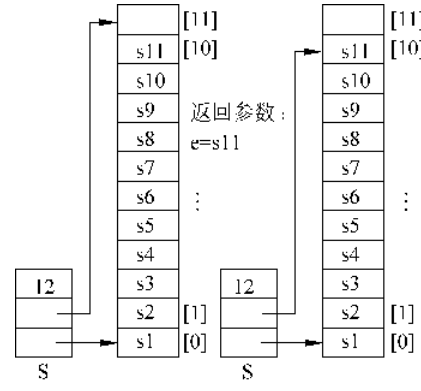
图 3-3 销毁顺序栈 S

```
  if(S.top == S.base) // 空栈条件
    return TRUE;
  else
    return FALSE;
}
int StackLength(SqStack S)
{ // 返回栈 S 的元素个数,即栈的长度
  return S.top - S.base;
}
Status GetTop(SqStack S,SElemType &e) // 在教科书第 47 页
{ // 若栈 S 不空,则用 e 返回 S 的栈顶元素,并返回 OK; 否则返回 ERROR
  if(S.top>S.base) // 栈不空
  { e = *(S.top - 1); // 将栈顶元素赋给 e
    return OK;
  }
  else
    return ERROR;
}
void Push(SqStack &S,SElemType e)
{ // 插入元素 e 为栈 S 新的栈顶元素。
  // 在教科书第 47 页(见图 3-4)
  if(S.top - S.base == S.stacksize) // 栈满
  { S.base = (SElemType * )realloc(S.base,
    (S.stacksize + STACK_INCREMENT) *
    sizeof(SElemType)); // 追加存储空间
    if(!S.base) // 追加存储空间失败,则退出
      exit(OVERFLOW);
    S.top = S.base + S.stacksize; // 修改栈顶指针,指向新的栈顶
    S.stacksize += STACK_INCREMENT; // 更新当前已分配的存储空间
  }
  *(S.top) ++= e; // 将 e 入栈,成为新的栈顶元素,栈顶指针上移 1 个存储单元
}
Status Pop(SqStack &S,SElemType &e) // 在教科书第 47 页
{ // 若栈 S 不空,则删除 S 的栈顶元素,用 e 返回其值,
  // 并返回 OK; 否则返回 ERROR(见图 3-5)
  if(S.top == S.base) // 栈空
    return ERROR;
  e = * -- S.top; // 将栈顶元素赋给 e,栈顶指针下移
                  // 1 个存储单元
  return OK;
}
void StackTraverse(SqStack S,void( * visit)(SElemType))
{ // 从栈底到栈顶依次对栈 S 中每个元素调用函数 visit()
```



(a) 调用Push()之前　(b) 调用Push()之后

图 3-4　调用 Push()示例



(a) 调用Pop()之前　(b) 调用Pop()之后

图 3-5　调用 Pop()示例

```
  while(S.top>S.base) // S.base 指向栈元素
    visit( * S.base ++ ); // 对该栈元素调用 visit(),
  printf("\n");          // 栈底指针上移 1 个存储单元
}

// main3-1.cpp 检验 bo3-1.cpp 的主程序
#include"c1.h"
typedef int SElemType; // 定义栈元素类型,此句要在 c3-1.h 的前面
#include"c3-1.h" // 栈的顺序存储结构
#include"bo3-1.cpp" // 顺序栈存储结构的基本操作(9 个)
#define ElemType SElemType // 将 func2-2.cpp 中的 ElemType 类型定义为 SElemType 类型
#include"func2-2.cpp" // 包括 equal()、comp()、print()、print1()和 print2()函数
void main()
{
  int j;
  SqStack s;
  SElemType e;
  InitStack(s); // 初始化栈 s
  for(j = 1;j<= 12;j ++ )
    Push(s,j); // 将值为 j 的栈元素入栈 s 中
  printf("栈中元素依次为");
  StackTraverse(s,print); // 从栈底到栈顶依次对栈 s 中每个元素调用 print()函数
  Pop(s,e); // 弹出栈顶元素,其值赋给 e
  printf("弹出的栈顶元素 e = %d\n",e);
  printf("栈空否? %d(1:空 0:否),",StackEmpty(s));
  GetTop(s,e); // 将新的栈顶元素赋给 e
  printf("栈顶元素 e = %d,栈的长度为 %d\n",e,StackLength(s));
  ClearStack(s); // 清空栈 s
  printf("清空栈后,栈空否? %d(1:空 0:否)\n",StackEmpty(s));
  DestroyStack(s); // 销毁栈 s
  printf("销毁栈后,s.top = %u,s.base = %u,s.stacksize = %d\n",s.top,s.base,s.stacksize);
}
```

程序运行结果:

```
栈中元素依次为 1 2 3 4 5 6 7 8 9 10 11 12
弹出的栈顶元素 e = 12
栈空否? 0(1:空 0:否),栈顶元素 e = 11,栈的长度为 11
清空栈后,栈空否? 1(1:空 0:否)
销毁栈后,s.top = 0,s.base = 0,s.stacksize = 0
```

3.2　栈的应用举例

3.2.1　数制转换

```
// algo3-1.cpp 调用算法 3.1 的程序
#define N 8 // 定义待转换的进制 N(2～9)
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion() // 算法 3.1
{ // 对于输入的任意一个非负十进制整数,打印输出与其等值的 N 进制数
  SqStack s;
  unsigned n; // 非负整数
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数 n 转换为 %d 进制数,请输入:n(≥0) = ",N);
  scanf("%u",&n); // 输入非负十进制整数 n
  while(n) // 当 n 不等于 0
  { Push(s,n%N); // 入栈 n 除以 N 的余数(N 进制的低位)
    n = n/N;
  }
  while(!StackEmpty(s)) // 当栈不空
  { Pop(s,e); // 弹出栈顶元素且赋值给 e
    printf("%d",e); // 输出 e
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

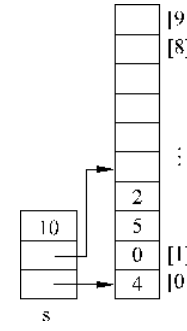


图 3-6　栈 s 在元素最多时的状态

程序运行结果(见图 3-6):

```
将十进制整数 n 转换为 8 进制数,请输入:n(≥0) = 1348 ↙
2504
```

如果将 N 定义为 2,algo3-1.cpp 就是将十进制数转换为二进制数的程序。

程序运行结果:

```
将十进制整数 n 转换为 2 进制数,请输入:n(≥0) = 13 ↙
1101
```

algo3-1.cpp 能不能用于十进制到十六进制的转换呢? 存在一个问题,要将余数 10～

15 转换为 A～F 输出。algo3-2.cpp 实现了十进制到十六进制的转换。

```
// algo3-2.cpp 修改算法 3.1,十进制→十六进制
typedef int SElemType; // 定义栈元素类型为整型
#include"c1.h"
#include"c3-1.h" // 采用顺序栈
#include"bo3-1.cpp" // 利用顺序栈的基本操作
void conversion()
{ // 对于输入的任意一个非负十进制整数,打印输出与其等值的十六进制数
  SqStack s;
  unsigned n; // 非负整数
  SElemType e;
  InitStack(s); // 初始化栈
  printf("将十进制整数 n 转换为十六进制数,请输入:n(≥0) = ");
  scanf("%u",&n); // 输入非负十进制整数 n
  while(n) // 当 n 不等于 0
  { Push(s,n%16); // 入栈 n 除以 16 的余数(十六进制的低位)
    n = n/16;
  }
  while(!StackEmpty(s)) // 当栈不空
  { Pop(s,e); // 弹出栈顶元素且赋值给 e
    if(e<=9)
      printf("%d",e); // 小于等于 9 的余数,直接输出
    else
      printf("%c",e+55); // 大于 9 的余数,输出相应的字符
  }
  printf("\n");
}
void main()
{
  conversion();
}
```

程序运行结果(见图 3-7):

```
将十进制整数 n 转换为十六进制数,请输入:n(≥0) = 164↙
A4
```

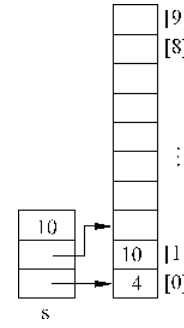


图 3-7 栈 s 在元素最多时的状态

3.2.2 行编辑程序

```
// algo3-3.cpp 行编辑程序,实现算法 3.2
typedef char SElemType; // 定义栈元素类型为字符型
#include"c1.h"
#include"c3-1.h" // 栈的顺序存储结构
```

```
#include"bo3-1.cpp" // 顺序栈存储结构的基本操作(9 个)
FILE * fp;
void copy(SElemType c)
{ // 将字符 c 送至 fp 所指的文件中
  fputc(c,fp);
}
void LineEdit()
{ // 利用字符栈 s,从终端接收一行并送至调用过程的数据区。算法 3.2
  SqStack s;
  char ch;
  InitStack(s); // 初始化栈 s
  printf("请输入一个文本文件,^Z 结束输入:\n");
  ch = getchar(); // 接收字符到 ch
  while(ch! = EOF) // 当全文未结束(EOF 为^Z 键,全文结束符)
  { while(ch! = EOF&&ch! = '\n') // 当全文未结束且未到行末(不是换行符)
      { switch(ch) // 对于当前字符 ch,分情况处理
      { case '#':if (!StackEmpty(s))
                    Pop(s,ch); // 仅当栈非空时弹出栈顶元素,c 可由 ch 替代
                break;
        case '@':ClearStack(s); // 重置 s 为空栈
                break;
        default :Push(s,ch); // 其他字符进栈
      }
      ch = getchar(); // 从终端接收下一个字符到 ch
    } // 到行末或全文结束,退出此层循环
    StackTraverse(s,copy); // 将从栈底到栈顶的栈内字符依次传送至文件(调用 copy()函数)
    fputc('\n',fp); // 向文件输入一个换行符
    ClearStack(s); // 重置 s 为空栈
    if(ch! = EOF) // 全文未结束
      ch = getchar(); // 从终端接收下一个字符到 ch
  }
  DestroyStack(s); // 销毁栈 s
}
void main()
{
  fp = fopen("ed.txt","w");
  // 在当前目录下建立 ed.txt 文件,用于写数据,如已有同名文件则先删除原文件
  if(fp) // 建立文件成功
  { LineEdit(); // 行编辑
    fclose(fp); // 关闭 fp 所指的文件
  }
  else
    printf("建立文件失败! \n");
}
```

程序运行结果(以教科书第 49 页的输入为例)：

```
请输入一个文本文件,^Z 结束输入：
whli##ilr#e(s## *s)↙

  outcha@putchar( *s = # ++ );↙

^Z↙
```

文件 ed.txt 的内容：

```
while( *s)
putchar( *s++ );
```

3.2.3 迷宫求解

```
// algo3-4.cpp 利用栈求解迷宫问题(只输出一个解,算法 3.3)
#include"c1.h"
struct PosType // 迷宫坐标位置类型(见图 3-8)
{ int x; // 行值
  int y; // 列值
};
// 全局变量
PosType begin,end; // 迷宫的入口坐标,出口坐标
PosType direc[4]={{0,1},{1,0},{0,-1},{-1,0}};
// {行增量,列增量},移动方向依次为东南西北
#define MAXLENGTH 25 // 设迷宫的最大行列为 25
typedef int MazeType[MAXLENGTH][MAXLENGTH]; // 迷宫数组类型[行][列]
MazeType m; // 迷宫数组
int x,y; // 迷宫的行数,列数
void Print()
{ // 输出迷宫的解(m 数组)
  int i,j;
  for(i=0;i<x;i++)
  { for(j=0;j<y;j++)
      printf("%3d",m[i][j]);
    printf("\n");
  }
}
void Init()
{ // 设定迷宫布局(墙值为 0,通道值为 1)
  int i,j,x1,y1;
  printf("请输入迷宫的行数,列数(包括外墙):");
  scanf("%d,%d",&x,&y);
```

PosType

x	y

图 3-8 PosType 结构

```
  for(i=0;i<y;i++) // 定义周边值为0(外墙)
  { m[0][i]=0; // 上边
    m[x-1][i]=0; // 下边
  }
  for(i=1;i<x-1;i++)
  { m[i][0]=0; // 左边
    m[i][y-1]=0; // 右边
  }
  for(i=1;i<x-1;i++)
    for(j=1;j<y-1;j++)
      m[i][j]=1; // 定义除外墙,其余都是通道,初值为1
  printf("请输入迷宫内墙单元数:");
  scanf("%d",&j);
  printf("请依次输入迷宫内墙每个单元的行数,列数:\n");
  for(i=1;i<=j;i++)
  { scanf("%d,%d",&x1,&y1);
    m[x1][y1]=0; // 修改内墙的值为0
  }
  printf("迷宫结构如下:\n");
  Print();
  printf("请输入入口的行数,列数:");
  scanf("%d,%d",&begin.x,&begin.y);
  printf("请输入出口的行数,列数:");
  scanf("%d,%d",&end.x,&end.y);
}
int curstep=1; // 当前足迹,初值(在入口处)为1
struct SElemType // 栈的元素类型。在教科书第51页(见图3-9)
{ int ord; // 通道块在路径上的"序号"
  PosType seat; // 通道块在迷宫中的"坐标位置"
  int di; // 从此通道块走向下一通道块的"方向"(0～3表示东～北)
};
#include"c3-1.h" // 采用顺序栈存储结构
#include"bo3-1.cpp" // 采用顺序栈的基本操作函数
// 定义墙元素值为0,可通过路径为1,经试探不可通过路径为-1,通过路径为足迹
Status Pass(PosType b)
{ // 当迷宫m的b点的值为1(可通过路径),返回OK;否则,返回ERROR
  if(m[b.x][b.y]==1)
    return OK;
  else
    return ERROR;
}
void FootPrint(PosType b)
{ // 使迷宫m的b点的值变为足迹(curstep)
  m[b.x][b.y]=curstep;
}
void NextPos(PosType &b,int di)
{ // 根据当前位置b及移动方向di,修改b为下一位置
```

SElemType

ord	seat.x	seat.y	di

图3-9　SElemType结构

```
  b.x += direc[di].x;
  b.y += direc[di].y;
}
void MarkPrint(PosType b)
{ // 使迷宫 m 的 b 点的值变为 -1(标记为经试探由此不能到达终点的路径)
  m[b.x][b.y] = -1;
}
Status MazePath(PosType start,PosType end) // 算法 3.3
{ // 若迷宫 m 中存在从入口 start 到出口 end 的通道,则求得一条存放在栈中(从栈底到栈顶),
  // 并返回 TRUE;否则返回 FALSE
  PosType curpos = start; // 当前位置在入口
  SqStack S; // 顺序栈
  SElemType e; // 栈元素
  InitStack(S); // 初始化栈
  do
  { if(Pass(curpos)) // 当前位置可以通过,即是未曾走到过的通道块
    { FootPrint(curpos); // 留下足迹(使迷宫 m 的当前位置的值为 curstep)
      e.ord = curstep; // 栈元素的序号为当前足迹 curstep
      e.seat = curpos; // 栈元素的位置为当前位置
      e.di = 0; // 从当前位置出发,下一位置是向东
      Push(S,e); // 入栈当前位置及状态
      curstep++; // 足迹加 1
      if(curpos.x == end.x&&curpos.y == end.y) // 到达终点(出口)
        return TRUE;
      NextPos(curpos,e.di); // 由当前位置及移动方向,确定下一个当前位置,仍赋给 curpos
    }
    else // 当前位置不能通过
      { if(!StackEmpty(S)) // 栈不空
      { Pop(S,e); // 退栈到前一位置
        curstep--; // 足迹减 1
        while(e.di == 3&&!StackEmpty(S)) // 前一位置处于最后一个方向(北)且栈不空
        { MarkPrint(e.seat); // 在前一位置留下由其不能到达终点的标记(-1)
          Pop(S,e); // 再退回一步
          curstep--; // 足迹再减 1
        }
        if(e.di<3) // 未到最后一个方向(北)
        { e.di++; // 换下一个方向探索
          Push(S,e); // 入栈该位置的下一个方向
          curstep++; // 足迹加 1
          curpos = e.seat; // 确定当前位置
          NextPos(curpos,e.di); // 由当前位置及移动方向,确定下一个当前位置,仍赋给 curpos
        }
      }
    }
```

```
  }while(!StackEmpty(S));
  return FALSE;
}
void main()
{
  Init(); // 初始化迷宫
  if(MazePath(begin,end)) // 有通路
  { printf("此迷宫从入口到出口的一条路径如下：\n");
    Print(); // 输出此通路(m 数组)
  }
  else
    printf("此迷宫没有从入口到出口的路径\n");
}
```

由于把迷宫数组 m、迷宫的行数 x、列数 y、迷宫的入口坐标 begin 和出口坐标 end 作为全局变量，它们在各函数中都通用，不需要用形参来传递，减少了 Print()函数和 Init()函数中的形参数量。

程序运行结果(以教科书图 3.4 为例)：

```
请输入迷宫的行数,列数(包括外墙)：10,10↙
请输入迷宫内墙单元数：18↙
请依次输入迷宫内墙每个单元的行数,列数：
1,3↙
1,7↙
2,3↙
2,7↙
3,5↙
3,6↙
4,2↙
4,3↙
4,4↙
5,4↙
6,2↙
6,6↙
7,2↙
7,3↙
7,4↙
7,6↙
7,7↙
8,1↙
迷宫结构如下：
  0  0  0  0  0  0  0  0  0  0
  0  1  1  0  1  1  1  0  1  0
  0  1  1  0  1  1  1  0  1  0
  0  1  1  1  1  0  0  1  1  0
  0  1  0  0  0  1  1  1  1  0
```

```
  0   1   1   1   0   1   1   1   1   0
  0   1   0   1   1   1   0   1   1   0
  0   1   0   0   0   1   0   0   1   0
  0   0   1   1   1   1   1   1   1   0
  0   0   0   0   0   0   0   0   0   0
请输入入口的行数,列数：1,1↙
请输入出口的行数,列数：8,8↙
此迷宫从入口到出口的一条路径如下：(见图 3-10)
  0   0   0   0   0   0   0   0   0   0
  0   1   2   0  -1  -1  -1   0   1   0
  0   1   3   0  -1  -1  -1   0   1   0
  0   5   4  -1  -1   0   0   1   1   0
  0   6   0   0   0   1   1   1   1   0
  0   7   8   9   0   1   1   1   1   0
  0   1   0  10  11  12   0   1   1   0
  0   1   0   0   0  13   0   0   1   0
  0   0   1   1   1  14  15  16  17   0
  0   0   0   0   0   0   0   0   0   0
```

图 3-10　到达终点时栈 S 的内容

从入口到出口的一条路径由全局变量 m 数组显示。入口的值为 1，路径的值依次为 2，3，…，17。图 3-10 显示了到达终点时栈的内容。其中栈元素的 di 成员的值本是 0～3，为直观，用东、南、西、北代替。由于有数组 m，栈的主要作用并不是保存路径。它的主要作用是当试探失败时，通过退栈回到前一点，从前一点再继续试探。

这条路径共走了 16 步，从入口(第 1 步)到第 5 步其实只须走 2 步。原因是 func3-1. cpp 中定义的数组 direc[]的移动方向依次为东、南、西、北。程序由入口处先向东试探。既然有路，就不再向南试探了。因此走了弯路。如果将 direc[0] 和 direc[1]的值交换一下，新的移动方向依次为南、东、西、北。这样，程序先向南试探，只在不成功时才向东试探。避免了在此处走弯路，14 步就到了出口。以下是修改了 direc[]的值为“direc[4]={{1,0},{0,1},{0,-1},{-1,0}};”，不改变输入数据的情况下的运行结果。为节约篇幅，略去相同部分，只列出最后结果。

程序运行结果：

```
此迷宫从入口到出口的一条路径如下：
  0   0   0   0   0   0   0   0   0   0
  0   1   1   0   1   1   1   0   1   0
  0   2   1   0   1   1   1   0   1   0
  0   3   1   1   1   0   0   1   1   0
  0   4   0   0   0   1   1   1   1   0
  0   5   6   7   0   1   1   1   1   0
  0  -1   0   8   9  10   0   1   1   0
  0  -1   0   0   0  11   0   0   1   0
  0   0   1   1   1  12  13  14  15   0
  0   0   0   0   0   0   0   0   0   0
```