第四章 栈和队列

数据结构课程

裘宗燕

栈和队列是计算机科学和程序设计中应用非常广范的两种数据结构，主要用于临时性地缓存数据元素（把一些数据保存起来供后面的计算中使用）。栈也常称为堆栈。

栈和队列可以方便地用线性表的基本结构实现，因此不少教科书把栈和队列看作是特殊的操作受限的线性表。

从抽象数据类型的角度看，栈和队列是与线性表完全不同的数据结构，因为它们具有不同的操作集合。

栈和队列都提供了元素存入、访问和删除操作，以及几个辅助操作，如创建、判断空等。

这两种结构的特点在于访问和删除操作：任何时刻能访问、删除的元素是默认的且唯一确定的。当时保存的其他元素都不能访问、删除。存入和删除操作将改变能访问的元素。

栈和队列给元素使用规定了顺序，两者的不同也就在这里。

4.1 栈及其基本运算

栈（Stack）是一种容器，可存入数据元素、访问元素、删除元素。在这里没有位置的概念。

栈保证任何时刻可访问、删除的元素都是前面最后存入栈里的那个元素。因此确定了一种访问顺序。

基本操作是封闭集合（与表不同），通常包括:

- 创建空栈
- 判断栈是否为空
- 向栈中插入（或称推入/压入，push）一个元素
- 从栈中删除（或称弹出，pop）一个元素
- 取当前（最新）元素的值（并不删除）

抽象数据类型（代数描述）

集合运算

元素和栈: $e, e_i \in D \qquad s, s_i \in Stack$

$$
\begin{array}{lcl}
newstack & : & \rightarrow Stack \\
push & : & D \times Stack \rightarrow Stack \\
top & : & Stack \rightarrow D \\
pop & : & Stack \rightarrow Stack \\
empty & : & Stack \rightarrow Boolean
\end{array}
$$

代数法则

$$
\begin{array}{lcl}
top(push(e,s)) & = & e \\
pop(push(e,s)) & = & s \\
empty(newstack()) & = & \text{true} \\
empty(push(e,s)) & = & \text{false}
\end{array}
$$

限制

$$
\begin{array}{lcl}
top(newstack()) & = & \text{error} \\
pop(newstack()) & = & \text{error}
\end{array}
$$

满足这组规范的系统（无论抽象的或具体的）都是栈（的模型）

满足这组规范的程序都是栈的实现（无论采用什么技术）

栈的特性:

- 保证任何时刻访问或删除的元素，一定是当时栈里所有元素中最后存入的那个元素（最新元素）
- 这种性质称为后进先出（Last In First Out, LIFO）
- 栈确定了元素的访问顺序，是一种与“时间”有关的结构

栈可看作（可实现为）只在一端插入和删除的表

因此有人把栈称为后进先出（LIFO）表。

进行插入或删除操作的一端称为栈顶，另一端称为栈底。见图。

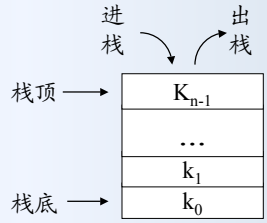


图4.1 栈

4.2 栈的实现

- 4.2.1 顺序表示

  采用类似线性表的结构实现栈

- 4.2.2 链接表示

  采用类似链接表的结构实现栈

4.2.1 顺序表示

顺序表示的栈数据结构的类型定义:

```
#define MAXNUM  100    /* 栈的最大容量*/

typedef int DataType;        /* 栈元素的数据类型* /    typedef
typedef struct  SeqStack  { /* 顺序栈类型定义 */
   int     t;                /*栈顶*/
   DataType  s[MAXNUM];
} SeqStack, *PSeqStack;

PSeqStack   psstack;          /*指向顺序栈的指针*/
```

完全可以采用动态顺序表的方式（技术）实现栈，创建时确定初始大小，必要时进行扩充。

顺序表示的栈基本运算的实现

- 算法4.1 创建空栈

  PSeqStack createEmptyStack_seq(void)
- 算法4.2 判断栈是否为空栈

  int isEmptyStack_seq(PSeqStack psstack)
- 算法4.3 进栈（压入）

  void push_seq(PSeqStack psstack, DataType x)
- 算法4.4 出栈（弹出）

  void pop_seq(PSeqStack psstack)
- 算法4.5 取栈顶元素

  DataType top_seq(PSeqStack psstack)

设 st 是个栈，现在计划在 st.s 里保存数据元素，用 st.t 记录元素的有关信息，支持栈的各种操作。问题:

怎样保证栈的各操作形成一个有机整体，共同实现了一种良好结构的栈？ 因为这些操作都是独立的函数，互不依赖，执行中可以按任何顺序调用。

需要确定一套所有操作都必须遵守的行为准则。

实现一种数据结构时，相关操作都必须遵守的一套准则称为这种数据结构数据不变式（Data Invariant）:

- 创建操作建立的结构应满足不变式（合法的初始状态）
- 其他操作都保证：只要操作前结构处于合法状态（满足不变式），操作结束时结构仍将处于合法状态
- 这样，无论执行多少次操作，顺序如何，数据结构总处于合法的状态（可以用归纳法证明）


```
typedef struct  SeqStack  { /* 顺序栈类型定义 */
   int     t;    DataType  s[MAXNUM];
} SeqStack, *PSeqStack;
```

对栈的顺序实现，我们确定如下不变式:

- 栈中所有元素在数组 s 下标较小的一端（低端）连续存放，按进栈顺序排列，最新元素的位置最高（下标最大）
- t 的值总是 s 里位置最高的元素的下标

也可以采用其他不变式，同样能得到完好的实现。例如:

- 栈中元素在 s 的低端连续存放， 按进栈顺序排列，…
- t 的值总是 s 里元素的个数

请自己考虑如何按此不变式实现栈，并比较两种实现

还可以提出其他不变式，产生顺序栈的不同实现


```
/*创建一个空栈*/
PSeqStack  createEmptyStack_seq( void ) {
   PSeqStack psstack = (PSeqStack)malloc(sizeof(SeqStack));
   if (psstack==NULL)
      printf("Out of space!! \n");
   else
      psstack->t = -1; /* 注意不变式：t = 栈顶元素下标 */
   return psstack;
}

/*判断psstack所指的栈是否为空栈*/
int  isEmptyStack_seq( PSeqStack psstack ) {
   return psstack->t == -1;
}
```

```
void  push_seq( PSeqStack psstack, DataType x ) {
    if ( psstack->t >= MAXNUM - 1)
        printf( "Stack Overflow! \n" );
    else {
        psstack->t++;  psstack->s[psstack->t] = x;
    }
}

void  pop_seq( PSeqStack psstack ) {
    if (psstack->t == -1 ) printf( "Stack Underflow!\n" );
    else psstack->t--;
}

/* 当栈psstack不为空栈时，求栈顶元素的值 */
DataType top_seq( PSeqStack psstack ) {
    return psstack->s[psstack->t];
}
```

请注意不变式的问题

4.2.2 链接表示

链接表示的栈数据结构的类型定义:

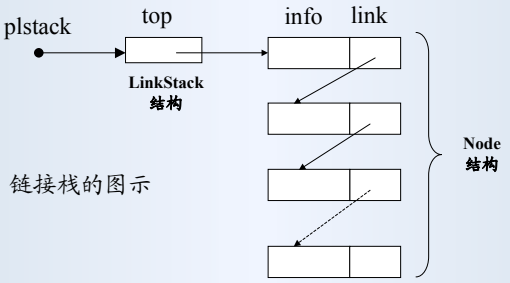
```
typedef  int  DataType;

typedef struct Node  Node, *pNode;
struct  Node  { /* 单链表结点结构 */
    DataType   info;
    pNode       link;
};

typedef struct LinkStack {   /* 链接栈类型定义 */
    pNode  top;          /* 指向栈顶结点 */
} LinkStack,  *PLinkStack;
```



```
PLinkstack  plstack;   /*变量声明*/
```



基本结构与链接表类似

链接表示栈基本运算的实现

- 算法4.6 创建空栈
  PLinkStack createEmptyStack_link(void)
- 算法4.7 判断栈是否为空栈
  int isEmptyStack_link(PLinkStack plstack)
- 算法4.8 进栈
  void push_link(PLinkStack plstack, DataType x)
- 算法4.9 出栈
  void pop_link(PLinkStack plstack)
- 算法4.10 取栈顶元素
  DataType top_link(PLinkStack plstack)


```
/* 创建空的链接栈，返回指向空链接栈的指针*/
PLinkStack  createEmptyStack_link(void) {
   PLinkStack plstack =
                (PLinkStack )malloc(sizeof(LinkStack));
   if (plstack != NULL)
      plstack->top = NULL;
   else
      printf("Out of space! \n");
   return plstack;
}

/* 判链接栈是否为空栈 */
int  isEmptyStack_link( PLinkStack plstack ) {
   return plstack->top == NULL;
}
```



```
void push_link( PLinkStack ps, DataType x ) {
    PNode  p = (PNode)malloc( sizeof( Node ) );
    if ( p == NULL )  printf("Out of space!\n");
    else {
        p->info = x; /* 存入数据 */
        p->link = ps->top; ps->top = p; /* 链接 */
    }
}
void  pop_link( PLinkStack ps ) {
    if( isEmptyStack_link( ps ) ) printf( "Empty stack pop.\n" );
    else {
        PNode p = ps->top;
        ps->top = p->link; /* 摘除 */
        free(p); /* 释放 */
    }
}
```

```
/* 对非空栈求栈顶元素值 */
DataType top_link( PLinkStack ps ) {
   return ps->top->info;
}
```

栈的另一种常见设计是不提供 top 操作，让 pop 操作在删除栈顶元素的同时返回它。例如:

- 进栈
  int push (stack, data_item)
- 出栈
  DataType pop (stack)

这种设计的优点是能更好反应栈的特性。缺点是不容易选择栈空时的返回值（返回什么？）

4.3 栈的应用*

- 4.3.1 栈与递归
  - 递归
  - 递归函数到非递归函数的转换*
  - 简化的背包问题*
- 4.3.2 迷宫问题
- 4.3.3 表达式计算
  - 后缀表达式的求值
  - 中缀表达式到后缀表达式的转换

栈有非常多的应用

后面的许多算法里需要都用栈作为辅助结构

4.3.1 栈与递归

递归定义和递归结构

如果一个定义或结构（如 C 函数，数据结构）的某个（某些）部分具有与整体同样的结构，则称其为递归的定义或结构

递归定义中的递归部分必须比整体简单，这样最后才能有终结点（称为递归定义的出口）；递归结构中也必须由基本结构直接组成的部分。否则就是无限递归，不是良好定义。

例:

- 递归定义的C函数（执行流程的一部分通过自身定义）
- 无头结点单链表（非空时，去掉一结点后还是同样结构）

具体实例

例: 简单表达式

a）常数、变量是表达式;

b）若e1, e2是表达式，op为运算符，则

e1 op e2, op e1, (e1) 也是表达式。

例: 阶乘函数 n!

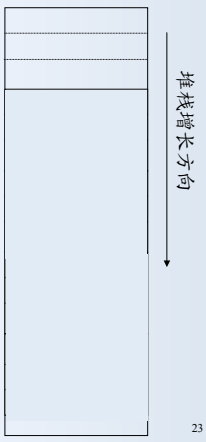
$$fact(n) = \begin{cases} 1 & n=0 \\ n * fact(n-1) & n>0 \end{cases}$$

递归算法主要适用于要解决的问题、要计算的函数、或者要处理的数据具有递归性质的情况

例: 阶乘的递归计算

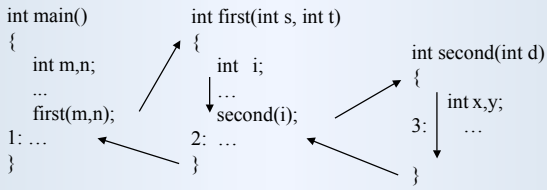
```
int fact (int n) {
   if (n == 0)
      return 1;
   else
      return  n*fact(n-1);
};                              r2

int main( ) {
   int n;
   printf("%8d%8d\n",
           n, fact(3));
}                               r1
```



二. 递归函数到非递归函数的转换

函数嵌套调用，按“后调用先返回”的规则进行。



与函数（过程）有关的数据总是“后进先出”，正好是栈的特征，适合采用栈保存中间的数据。

函数调用的前序和后序动作

调用前:

1. 为被调用函数的局部变量分配存储区;
   （活动记录，数据区，堆栈帧）
2. 将所有实参和返回地址存入活动记录;
   （实参和形参的结合，传值。）
3. 将控制转到被调用函数入口。

调用后（返回）:

1. 将被调用函数的计算结果存入指定位置;
2. 释放被调用函数的存储区;
3. 按以前保存的返回地址将控制转回调用函数。

递归定义的函数每次递归函数调用，都将自动执行这些动作。

要把这种函数变换成非递归的，就需要自己做这些事情，用栈保存中间的信息

算法4.12 阶乘的非递归计算

```
int  norecfact (int n) { /*自己管理栈，模拟函数调用过程 */
    int res = 1;
    pSeqStack st = createEmptyStack_seq();
    while (n > 0) {
        push_seq(st, n);   -- n;
    }
    while ( !isEmptyStack_seq(st) ) {
        res *= top_seq(st);
        pop_seq(st);
    }
    free(st);  return(res);
}
```

这里并没有严格地按规矩翻译，只保存了必要信息。例如这里的计算结果没有进栈。

三．简化的背包问题

背包里可放重量为s的物品，现有n件物品，重量分别为w_1, w_2, …, w_n。问能否从中选出若干件的重量之和正好是s。若存在则称此背包问题有解，否则无解。

背包问题表示:

int knap(int s, int n)　其中 s >= 0, n >= 1

如果有解:

1. 如果选的物品中不包含w_n;
   knap(s, n-1) 的解就是knap(s, n)的解;
2. 如果选的物品中包含w_n;
   knap(s - w_n, n-1)的解就是knap(s, n)的解.

问题显然具有递归性质，将原问题的求解归结为较少物品或者较少总重量的同样问题的求解。

背包问题可递归定义如下:

$$knap(s, n) = \begin{cases} 1 & s = 0 \\ 0 & s < 0 \\ 0 & s > 0 \text{ , } n < 1 \\ knap(s, n-1) \text{ 或 } knap(s-w_n, n-1) & \text{当 } s>0, n>=1 \end{cases}$$

1 表示有解；0 表示无解

前三种情况可以直接知道有没有解

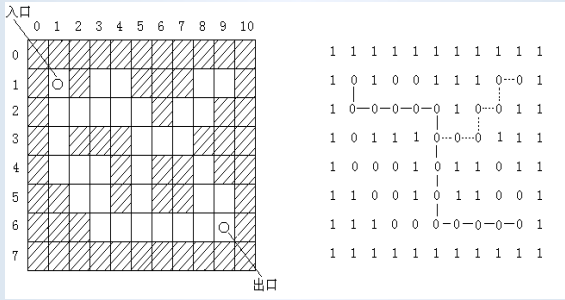
后两种情况，把问题归结到简单一些的问题

算法4.13 背包问题的递归算法

```
int w[] = { … … };  /* 全局数组，n件物品的重量 */
int knap(int s, int n) {
    if ( s == 0) return 1;
    else if (s < 0 || s > 0 && n < 1) return 0;
    else if (knap(s - w[n-1], n-1) == 1) {
        printf("n = %d, w[%d]=%d\n", n, n-1, w[n-1]);
        return 1;
    }
    else  return (knap(s, n-1));
}
```

通过引入堆栈，也可以把这个算法转换为非递归算法，但得到的算法复杂且难以理解（见书）。今天由于语言实现技术的发展，人工去做这种转换的需要也越来越少了。

4.3.2 迷宫问题



(a) 迷宫的图形表示　　(b) 迷宫的二维数组表示

探索到出口的路径，具有递归性质：

- 如果当前位置是出口，问题已解决
- 如果从当前位置已无路可走，探索失败
- 向可行的方向走一步，从那里出发探索到出口的路径

本问题的特点：

- 在每个位置可能有多个可行选择，有分支，需要逐一试探
- 只需要找到一条路径（而不是所有可能路径）

要解决这个问题，需要：

- 为问题找一种数据表示
- 一种确定可行方向的方式
- 防止出现在兜圈子的情况（设法记录已试探过的位置）

问题表示

用整数矩阵（二维数组）表示迷宫。

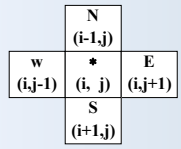
初始时，通路上的点用0表示，非通路点用1。

入口和出口都是数组下标对。

为避免陷入无限循环，在探索中把试探过的点标记为2。

方向处理：找一种方便形式，表示从一个位置出发的可能试探位置

迷宫中任一位置（i, j）有4个可能方向



用数组 dirs 表示可能方向（4个）。数组元素是计算4个方向下一点的偏移值，便于计算各方向的下一位置

	i增值	j增值	方向
0	0	1	E
1	1	0	S
2	0	-1	W
3	-1	0	N

```
int dirs[4][2] = {0,1,1,0,0,-1,-1,0};
```


可以写出一个简单的递归算法。框架：

```
int mazepath(int x1, int y1) {
    mark ( x1, y1 );              /* 留下标记 */
    if ( end(x1, y1) ) return 1;  /* 已是出口 */
    for ( i = 0; i < 3; ++i ) {
        if ( ok( x1+dirs[i][0], y1+dirs[i][1] )
             && mazepath(x1+dirs[i][0], y1+dirs[i][1]) )
            return 1;
    }
    return 0;
}
```

其中x1+dirs[i][0], y1+dirs[i][1] 求出下一探索位置

实际程序可能还需要考虑如何打印路径等等问题

下面考虑非递归的算法，即教科书上的算法

求迷宫中一条路径的方法（回溯法）

从入口出发，采用试探方法，搜索到目标点（出口）的路径

遇到出口则成功结束

遇到分支点时选一个方向向前探索。这时需记录当时的分支点和在这里已试探过的分支（和尚未试探过的分支）

若遇到死路（所有方向都不能走或已试探过），就退回前一分支点，换一方向再探索。直到找到目标，或者所有可能通路都探索到为止。这类方法称为回溯法。

- 每次回退（回溯）时总是去考虑最近记录的那个分支点，如果最近分支点已经没有其他选择，就把它删除
- 记录和删除具有后进先出性质，可以用栈保存分支点信息
- 遇到分支点将相关信息压入栈，删除分支点时将它弹出

迷宫问题算法框架（用一个栈记录搜索中需保存的信息）：

```
入口点相关信息（位置和方向）入栈;
while（栈不空）{
    取栈顶元素作为当前点和方向（并弹出栈顶）;
    while（当前点还存在未试探的方向）{
        求下一点的位置 (g, h);
        if（maze[g][h]是出口）{
            输出路径并 return;
        }
        if (maze[g][h] == 0) {  /*可前进*/
            当前点和方向入栈;
            标记maze[g][h];
            把 (g, h) 作为当前点;
        }
    }
}
```


栈中元素需要记录走过的位置和已经做过的方向选择。包括三项，分别记录位置的行、列坐标，及在该位置已选的方向，4 个方向编码为0、1、2、3（directon 数组的下标值），记录已试探过的方向的最大下标

下面算法使用顺序栈，栈元素类型：

```
typedef struct  NodeMaze {
    int  x, y, d;  /* 当前位置(x,y)和已试探方向的最大下标 d */
}  DataType;
```

什么样的位置入栈？可以选择的方式：

1，所有经过位置都入栈（下面算法采用）

2，还有没有试探的分支的位置

算法4.15　求迷宫中一条路径的非递归算法

```
/* 在迷宫maze[M][N]中求maze[x1][y1]到maze[x2][y2]的路径 */
void mazePath(int maze[][N], int x1, int y1, int x2, int y2) {
    int i, j, k, g, h;    PSeqStack st = createEmptyStack_seq( );
    maze[x1][y1] = 2;                    /* 从入口开始,作标记 */
    pushtostack(st, x1, y1, -1);         /* 入口点进栈 */
    while (!isEmptyStack_seq(st)) { /* 走不通时,一步步回退 */
        DataType ele = top_seq(st);  /* 取前一分支点 */
        pop_seq(st); i = ele.x; j = ele.y;
        for (k = ele.d + 1; k <= 3; ++k) {  /* 依次检查还没有试探的方向 */
            g = i + dirs[k][0]; h = j + dirs[k][1]; /* 算出下一点 */
            if ( g == x2 && h == y2 )        /*到达出口,打印路径 */
            {   printpath(st);  free(st);  return;   }
            if (maze[g][h] == 0) {           /* 遇到一个没试探过的新点 */
                pushtostack(st, i, j, k);    /* 这个新点进栈 */
                maze[g][h] = 2; i = g; j = h; k = -1; /* 标记，作为新的当前点 */
            }
        }
    }
    free(st); printf("No path found.\n");  /* 找不到路径 */
}
```



```
/* 两个辅助函数 */
void pushtostack(PSeqStack st, int x, int y, int d) {
    DataType element;
    element.x = x; element.y = y; element.d = d;
    push_seq(st, element);
}

void printpath(PSeqStack st) {
    DataType element;
    printf("The revers path is:\n");    /* 打印路径上各点 */
    while(!isEmptyStack_seq(st)) {
        element = top_seq(st);
        pop_seq(st);
        printf("the node is: %d %d \n", element.x, element.y);
    }
}
```


迷宫问题是一大类问题的代表。这类问题的基本特征是:

- 存在一组可能的状态（位置、情况等）
- 存在一个初始状态 s_0
- 有一个或者多个结束状态（或存在一种判断结束的方法）
- 对每个状态 s，neighbor(s) 表示与 s 相邻的状态（一步可达）
- 有一个判断函数 valid(s) 判断 s 是否为合法的可行状态
- 问题是：找出从 s_0 到某个（或全部）结束状态的路径；或者是从 s_0 出发，设法找到一个/全部解（一个/全部结束状态）

这类路径搜索问题，都可以用递归的方法求解；也可以借助于一个栈，通过回溯法求解。这类问题也被称为搜索问题。

其他例子如：八皇后问题，骑士周游问题等。实际中的例子包括许多调度问题（例如背包问题），定理证明等

4.3.3 表达式求值

中缀表达形式：二元运算符位于运算对象之间，中缀表达式

后缀表达形式：运算符位于运算对象之后，后缀表达式

例：中缀表达式：3 * (7 + 3 * 6 / 2 – 5)

转换成等价的后缀表达式：3 7 3 6 * 2 / + 5 - *

后缀形式的特点：不需要定义运算符优先级，不需要括号，就可唯一确定表达式计算顺序（只需知道每个运算符的元数）

与后缀形式对应的还有一种前缀表达形式，其中的运算符位于对应的运算对象之前。也有上面性质

为简单，下面假定表达式里的运算对象都是整数；运算符都是二元运算符。（更复杂的情况也可处理，需要记录信息）

计算后缀表达式:

例：3 7 3 6 * 2 / + 5 - *

方法：用一个运算对象栈

反复读入运算对象和运算符:

- 遇到运算对象：入栈
- 遇到运算符：弹出两个运算对象，执行运算，结果入栈。

读完整个表达式时，如栈里只有一个元素，那就是计算结果

如果读入中需要用运算对象时遇到栈空，或者读完表达式时栈内元素不止一个，都是表达式结构错误

后缀表达式求值的算法框架:

```
创建空堆栈 st;
while ( 读入表达式下一元素到 tk 成功 ) {
    if ( tk 是数字序列 )
        将 tk 转换为整数存入 st;
    else {
        从 st 弹出一个元素到 x; 弹出另一元素到 y;
        根据 tk 算出 x 与 y 运算的结果放在 z;
        将 z 压入栈 st;
    }
}
检查 st，若只有一个元素就打印结果，否则报错
```

tk 可以用 C 字符数组。为简单，要求输入表达式各项间有空格分隔（读入下一个单词。没有也可以处理，但复杂些）

中缀表达式转换为后缀表达式:

例: 3 * (7 + 3 * 6 / 2 – 5)

→ 3 7 3 6 * 2 / + 5 - *

方法: 用一个运算符栈。需要运算符表记录运算符的优先级

算法梗概: 用一个运算符栈。逐个读入单词，其中:

- 遇运算对象时直接输出
- 遇运算符 o，与栈顶运算符 o' 比较，若 o' 优先级不低于 o 则弹出 o' 输出（如此反复，直至至栈顶运算符优先级低于 o 的优先级）。最后把o 压入
- 遇左括号时将其进栈
- 遇到右括号时逐个弹出运算符输出，直至遇左括号时弹出
- 表达式结束时逐个弹出运算符输出

中缀表达式求值:

运算对象栈 dst 和运算符栈 ost。逐个读入单词，其中:

- 遇运算对象，直接压入 dst
- 遇运算符 o，与 ost 栈顶 o' 比较。o' 优先级不低于 o 时弹出 o'，从 dst 弹出运算对象，计算结果压入 dst。反复至栈顶运算符优先级低于o 时将运算符 o 压入 ost
- 遇左括号，直接进栈 ost
- 遇右括号，逐个弹出运算符，从 dst 弹出运算对象，计算并将结果压入 dst，直至把对应左括号弹出
- 遇表达式结束，逐个弹出运算符并进行计算，直至处理完全部运算符
- 若 ost 只有一个元素则完成并输出，否则表达式有错。

4.4 队列（Queue）

队列支持的访问顺序是先进先出（FIFO），被访问（删除）的总是当时队列里的元素中最早存入的那个元素

队列基本操作也是封闭集合（与栈类似）

通常包括:

- 创建空队列
- 判断队列是否为空
- 一个元素进入队列（入队，enqueue）
- 查看队头元素（front）;
- 删除队头元素（出队，dequeue）。

抽象数据类型（代数描述）

集合 元素和队列: $e, e_i \in D \quad q, q_i \in Queue$

运算

$$
\begin{array}{lcl}
newqueue & : & \rightarrow Queue \\
enqueue & : & D \times Queue \rightarrow Queue \\
front & : & Queue \rightarrow D \\
dequeue & : & Queue \rightarrow Queue \\
empty & : & Queue \rightarrow Boolean
\end{array}
$$

满足这组规范的系统（抽象或具体）都是队列的模型

满足这组规范的程序都是队列的实现

代数法则

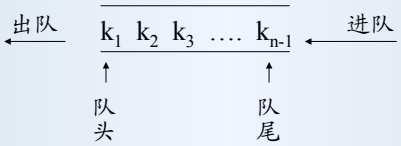
$$
\begin{array}{lcl}
empty(newqueue()) & = & \text{true} \\
empty(enqueue(e,q)) & = & \text{false} \\
dequeue(enqueue(e,q)) & = & empty(q)\ ?\ newqueue()\ :\ enqueue(e, dequeue(q)) \\
front(enqueue(e,q)) & = & empty(q)\ ?\ e\ :\ front(q)
\end{array}
$$

限制

$$
\begin{array}{lcl}
dequeue(newqueue()) & = & \text{error} \\
front(newqueue()) & = & \text{error}
\end{array}
$$

队列可看作（实现为）一种特殊线性表:

- 所有插入都在一端进行，所有删除都在另一端进行
- 删除的一端称队列头，插入的一 端叫队列尾
- 有人把队列称作先进先出表



队列操作也是“封闭”集合，基本操作:

- 创建空队列

  Queue createEmptyQueue (void)

- 判断是否为空队列

  int isEmptyQueue (Queue qu)

- 向队列中插入一个元素

  void enQueue (Queue qu, DataType x)

- 从队列中删除一个元素

  void deQueue (Queue qu)

- 求队列头部元素的值

  DataType frontQueue (Queue qu)

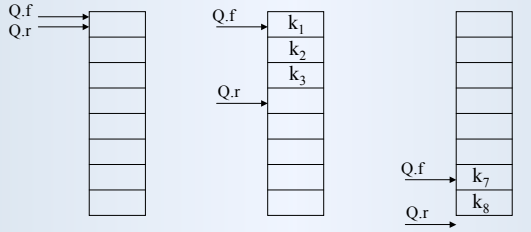
4.5 队列的实现

- 4.5.1 顺序表示
- 4.5.2 链接表示

4.5.1 队列的顺序表示

```
#define  MAXN  …  /* 队列中最大元素个数 */
typedef struct  SeqQueue { /* 顺序队列类型定义 */
   int  f, r;  /* 队列头尾下标 */
   DataType  q[MAXN];
} SeqQueue,  *PSeqQueue;
```


(1) 简单实现有问题:

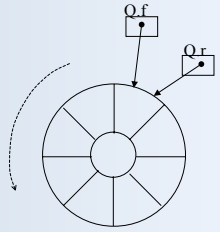


空队列

再存入就导致队列数组越界

实现方式不合适!

(2) "环形队列"实现（把数组看成环形）



图(a) 队列空

入队时，先存入，后移位

数据不变式:

Q.r是最后元素之后空位的下标

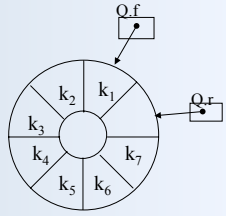
Q.f是首元素的下标

[Q.f, Q.r) 是队列中所有元素（看作按照环形排列时）

当 Q.f == Q.r 时队空

队列满如何判断?

条件不能与队列空判断相同



队列满的状态

判断条件:
(Q.r + 1) mod MAXN == Q.f

入队或者出队时的下标更新语句

Q.f = (Q.f + 1) % MAXN

Q.r = (Q.r + 1) % MAXN

保证更新后的下标仍具有合适的值

环形队列中实现队列基本运算的算法

- 算法4.16 创建一个空队列
  PSeqQueue createEmptyQueue_seq(void)
- 算法4.17 判断队列是否为空队列
  int isEmptyQueue_seq(PSeqQueue paqu)
- 算法4.18 进队列运算
  void enQueue_seq(PSeqQueue paqu, DataType x）
- 算法4.19 出队列运算
  void deQueue_seq(PSeqQueue paqu)
- 算法4.20 取队列头元素的值
  DataType frontQueue_seq(PSeqQueue paqu)


```
/*创建一个空队列*/
PSeqQueue  createEmptyQueue_seq( void ) {
   PSeqQueue paqu =
                (PSeqQueue)malloc(sizeof(struct SeqQueue));
   if (paqu==NULL) printf("Out of space!! \n");
   else paqu->f = paqu->r = 0;
   return paqu;
}
/*判队列是否为空队列*/
int  isEmptyQueue_seq( PSeqQueue paqu ) {
   return paqu->f == paqu->r;
}
/* 对非空队列,求队列头部元素 */
DataType  frontQueue_seq( PSeqQueue paqu ) {
   return paqu->q[paqu->f];
}
```

```
/* 在队列中插入一元素x */
void  enQueue_seq( PSeqQueue paqu, DataType x ) {
   if ( (paqu->r + 1) % MAXN == paqu->f )
      printf( "Full queue.\n" );
   else {
      paqu->q[paqu->r] = x;
      paqu->r = (paqu->r + 1) % MAXN;
   }
}
/* 删除队列头部元素 */
void  deQueue_seq( PSeqQueue paqu ) {
   if ( paqu->f == paqu->r )
       printf( "Empty Queue.\n" );
   else
       paqu->f = (paqu->f + 1) % MAXN;
}
```

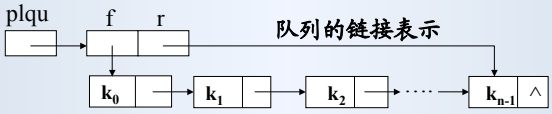

按照前面这种设计，由于需要区分队列空和队列满，在大小为 MAXN 的队列里最多存放 MAXN-1 个元素。

其他可能的设计

- 如果希望充分利用队列表示的数组空间，也可以增加一个元素个数记录 n，根据 n 的值区分空和满
- 修改设计时需要修改数据不变式，操作的实现要满足修改后的数据不变式
- 增加元素个数记录，也增加了入队、出队操作时需要维护的数据成分的工作
- 也可以考虑增加 n 并去掉队尾指标 r 的方式（请考虑这一设计带来哪方面的损失）

4.5.2 队列的链接表示

```
typedef  struct Node  *pNode;
typedef  struct Node {
    DataType    info;
    pNode       link;
} Node;
typedef struct LinkQueue {
    PNode       f, r;
} LinkQueue, *PLinkQueue;
```



链接队列中实现队列基本运算的算法

- 算法4.21 创建一个空队列

  PLinkQueue createEmptyQueue_link( )
- 算法4.22 判断队列是否为空队列

  int isEmptyQueue_link(PLinkQueue plqu)
- 算法4.25 取队列头部元素的值

  DataType frontQueue_link(PLinkQueue plqu)
- 算法4.23 进队列运算

  void enQueue_link(PLinkQueue plqu, DataType x)
- 算法4.24 出队列运算

  void deQueue_link(PLinkQueue plqu)


```
/*创建一个空队列*/
PLinkQueue  createEmptyQueue_link( ) {
   PLinkQueue plqu =
                (PLinkQueue )malloc(sizeof(struct LinkQueue));
   if (plqu != NULL) plqu->f = plqu->r = NULL;
   else printf("Out of space!! \n");
   return (plqu);
}
/*判断链接表示队列是否为空队列*/
int  isEmptyQueue_link( PLinkQueue plqu ) {
   return (plqu->f == NULL);
}
/* 在非空队列中求队头元素 */
DataType  frontQueue_link( PLinkQueue plqu ) {
   return (plqu->f->info);
}
```



```
void  enQueue_link( PLinkQueue plqu, DataType x) {
   PNode p = (PNode ) malloc( sizeof( struct Node ) );
   if ( p == NULL  ) printf("Out of space!");
   else {
      p->info = x;    p->link = NULL;
      if (plqu->f == NULL)  plqu->f = p;
      else  plqu->r->link = p;
      plqu->r = p;
   }
}
void  deQueue_link( PLinkQueue plqu ) {
   PNode   p;
   if ( plqu->f == NULL )  printf( "Empty queue.\n " );
   else {
      p = plqu->f;  plqu->f = plqu->f->link;  free(p);
   }
}
```

4.6 队列的应用——农夫过河问题

农夫过河问题:

- 农夫带一只狼、一只羊和一棵白菜在河的南岸
- 需要全部转运到北岸
- 一条小船只能容下农夫和一件物品
- 只有农夫能撑船
- 问怎么才能安全地过河？（问题：当农夫不在时狼会吃羊，羊也会吃白菜）

也是典型的在状态空间里搜索路径的问题:

- 一组状态（例如农夫和羊在南，狼和白菜在北）
- 从一状态可合法到达另几个状态（如农夫自己过河，从 s 可达 neighbor(s) 中任何一个状态）
- 有些状态不合法（或者不安全，valid(s) 问题。如农夫在北岸，其他东西在南岸）
- 有一个初始状态（都在河的南岸）
- 结束状态集（这里只有一个，都在河的北岸）
- 问题：找一条路径（相邻状态间转移合法），从开始状态到达某结束状态，途中不经过不安全状态

这种问题求解的搜索过程可采用两种不同策略:

- 深度优先(depth_first)搜索。每个时刻探索一条路径，一直向前探试，直到走不通时回溯
- 广度优先(breadth_first)搜索。在所有可能路径上齐头并进，同时探索所有可能性

实现这两种策略所依赖的数据结构正好是栈和队列

前面迷宫问题是搜索问题，用栈实现深度优先搜索

农夫过河问题也可以用深度优先搜索，用栈实现

下面算法采用广度优先搜索，用队列实现

问题表示:

需表示问题中的状态，其中的信息包括农夫等（人和物）位于南岸/北岸（各有两种可能）

可用整数数组。书上采用位向量，用 4 个二进制位的0/1 情况表示状态（易见，共 $2^4 = 16$ 种可能状态）

位向量从低位到高位分别表示羊、白菜、狼和农夫

0000（即整数0）表示都在南岸，目标状态 1111（即整数15）表示都到了北岸

定义如下枚举符:

```
enum { GOAT = 1, CABBAGE = 2, WOLF = 4, FARMER = 8 };
```

借助于位向量和按位运算符，容易描述过河动作

良好的问题表示，可能得到简单和/或高效的程序实现


```
/*求人或物的当前状态的函数（南：0；北：1）*/
int goat(int st) {    return (st & GOAT) != 0; }
int cabbage(int st) {    return (st & CABBAGE) != 0 ; }
int wolf(int st) {    return (st & WOLF) != 0; }
int farmer(int st) {    return (st & FARMER) != 0; }
/*安全性判断函数，若状态安全则返回 1*/
int safe(int st) {
   if ((goat(st) == cabbage(st)) &&  (goat(st) != farmer(st)) )
      return 0; /* 羊吃白菜 */
   if ((goat(st) == wolf(st)) &&  (goat(st) != farmer(st)))
      return 0; /* 狼吃羊 */
   /* 其他状态是安全的 */
   return 1;
}
```


算法梗概

- 队列 moveTo 记录可达但尚未向前探试的状态
- 数组 route 记录到各状态的路径，其中route[i] 记录到状态 i 的路径上的前一状态，-1 表示未访问（前一状态未知）

算法:

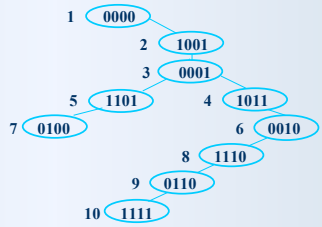
```
初始状态放入队列; 所有状态标记为未访问;
while (!isEmptyQueue_seq(moveTo)&&(route[15] == -1)) {
    从 moveTo 取出一个状态;
    试探所有由这个状态出发可能到达的状态
        if (能到达的状态安全且未访问过) {
            记录到它的路径;
            放入队列;
        }
}
```

```
void farmerProblem( ) {
   int mover, i, st, st_new;    int route[16];
   PSeqQueue states = createEmptyQueue_seq( );
   enQueue_seq(states, 0x00);
   for (i = 0; i < 16; i++) route[i] = -1;    route[0] = 0;

   while (!isEmptyQueue_seq(states) && (route[15] == -1)) {
      st = frontQueue_seq(states); deQueue_seq(states);
      for (mover = 1; mover <= 8; mover <<= 1)
         if (farmer(st) == (0 != (st & mover))) { /* 与农夫同岸？ */
            st_new = st^(FARMER | mover); /* 求过河后状态 */
            if (route[st_new] == -1 && safe(st_new))
               /* 新状态没到过而且安全，记录路径并入队列 */
               route[st_new] = st;    enQueue_seq(states, st_new);
         }
   }
   … … /* 打印输出 */
}
```


Path: 15，6，14，2，11，1，9，0
从初始状态0到最终状态15的动作序列为：

1(0-9) 农夫把羊带到北岸
2(9-1) 农夫独自回到南岸
3(1-11)农夫把白菜带到北岸
4(11-2)农夫带着羊返回南岸
6(2-14)农夫把狼带到北岸
8(14-6)农夫独自返回南岸
9(6-15)农夫把羊带到北岸



广度优先搜索得到的过河流程

关于状态空间搜索问题

- 两种基本方式：深度优先搜索和宽度优先搜索
- 深度优先：在途径的每个分支点选一个分支前进，保留其他可能分支的信息，遇到死路时回溯
  - 用栈保留路途上的分支信息，可保证上述性质
  - 可能比较高效，需要保存的信息与路径长度成正比
  - 可能陷入无穷路径而无法得到解
- 宽度优先：按与初始点的距离，在所有可能路径上一步步齐头并进，在每个点探索所有下一步可能到达的点
  - 用队列保存还有探索可能的所有点，可保证上述性质
  - 可以保证找到的解是距离初始点最近的解
  - 若路径长为 L，点的平均分支数为k，要保存的信息与L^k成正比，可能膨胀很快（指数爆炸）

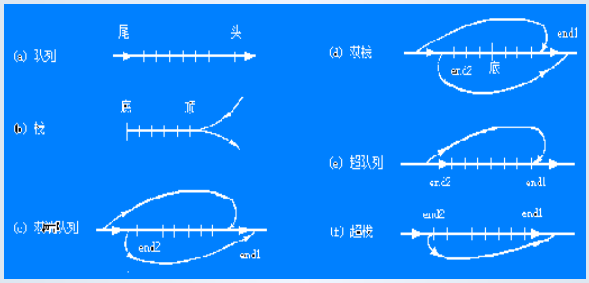
4.7 与栈、队列或表有关的结构

双端队列（deque）： 所有插入和删除在两端进行。两端点分别记作end1和end2。

双栈： 一种受限双端队列，规定从end1插入的元素只能从end1端删除，而从end2插入的元素只能从end2端删除。实际上是两个底部相连的栈。

超队列： 一种输出受限的双端队列，允许在两端插入，限制在一端(例如end1)删除。它好象一种特殊的队列，允许有的最新插入的元素最先删除。

超栈： 一种输入受限的双端队列，允许在两端删除，限制在一端（如end2）插入。它可以看成对栈溢出时的一种特殊处理，即当栈溢出时，可将栈中保存最久（end1端）的元素删除。



小　结

本章主要讨论栈和队列，其基本概念、存储结构、基本运算的实现及一些应用实例。(都可看作受限的表)

栈的插入和删除在同一端进行的，用顺序存储结构时，要注意栈满、栈空条件；用链式存储结构时要注意链的方向。

队列插入在一端进行，而删除在另一端。根据这一特点，使用顺序存储结构时需要用环形队列，解决假溢出问题，但要特别注意队列满和队列空的条件及描述。

递归是计算机科学中的一个根本性问题，本章的讨论仅限于它和栈的关系，作为栈应用的例子，可用栈将递归函数转换成非递归函数，对此我们应有一定的了解。

回溯法是最基本的算法设计技术之一，通过探试确定问题的解序列。本章作为栈的应用对回溯算法做了简单介绍。

下面章节中还可以看到栈和队列的许多应用。