**3.2队列（Queue）**

**3.2.1 队列的逻辑结构**

**1.队列的定义**

**队列是一种先进先出（first in first out）或后进后出（last in last out）的线性表。**

**队列是限定只能在队尾（rear）插入，称为“入队”，在队头（front）删除称为“出队”。不考虑队列“加塞”和“中途退出”情况。如果有这样的考虑，那是队列的进一步使用了。**

**显然它与栈不同,栈的插入和删除都在栈顶一头进行。**

**队列操作反映“先来先得到服务，后来后得到服务”的原则。**

**日常生活中的队例：排队买饭、上车等。**

**●通常队列表示如下：**

****

**出队入队**

**2. 队列的基本操作**

**（1）置空队:初始化一个空队**

**（2）入队：在队尾插入一个新元素**

**（3）出队：在对头删除一个元素**

**（4）取队头元素：取对头元素的值，与出队操作不同，对头元素不变**

**（5）判别队为空吗？是返回1，否则返回0**

**3.2.2队列的顺序存储**

**与线性表和栈类似，队列也有顺序存储和链式存储两种方式。**

**1. 顺序队列**

**用一维数组存储。**

**定义：**

**#define MAXSIZE 100**

**typedef int ElemType;**

**typedef struct Node**

**{**

**ElemType data[MAXSIZE];**

**int front,rear;**

**}SeqQueue;**

**SeqQueue Q;**

**顺序队列头尾指针所指安排的三种情形：**

1. **front和rear：若队列非空，front指向队头元素的位置，rear指向队尾元素的下一个元素的位置。如：**

**8**

**6**

**3**

**↑front ↑rear**

**若Q为空队初始化约定：Q.front=Q.rear=-1**

**↑front ↑rear**

**空队列**

**6**

**3**

**8**

**↑front↑rear**

**注意：8,3,6视为不在队列中**

**队列元素个数= Q.rear -Q.front**

**当空队列时：队列元素个数= Q.rear -Q.front=0（即Q.rear=Q.front）**

**说明：大部分教材都用这种定义**

**(2)有的书上将队列指针定义成下列情形也可以：**

**front和rear：若队列非空，front指向队头元素的前一个位置，rear指向队尾元素的位置。如：**

**8**

**6**

**3**

**↑front ↑rear**

**当空队列时：队列元素个数= Q.rear -Q.front=0（即Q.rear=Q.front）**

**(3)将队列指针定义成下列情形对空队列遭遇尴尬（不用）：**

**front和rear：若队列非空，front和rear分别指向队头元素和队尾元素的位置。如：**

**8**

**6**

**3**

**↑front ↑rear**

**说明：这样规定，队列元素个数= Q.rear-Q.front+1，当队列为空时（元素个数为0）Q.rear -Q.front+1=0**

**（即front=rear+1），导致front在rear之后，矛盾（所以不用这种定义）。**

**即空队列**

**↑rear ↑front**

**●入队操作**

**例如：将元素5加入队中**

**Q.data[Q.rear++]=5;//注意溢出**

**入队前：**

**3**

**6**

**8**

**↑front ↑rear**

**入队后：**

**3**

**5**

**6**

**8**

**↑front ↑rear**

**队满时不能再入队,否则溢出：rear=MAXSIZE+1**

**3**

**5**

**6**

**8**

**↑front ↑rear**

**void EnSeqQueue(SeqQueue Q,ElemType x)//入队**

**{**

**if(Q.rear==MAXSIZE+1)**

**{printf(“队满”);exit(0);}**

**Q.data[Q.rear++]=x;**

**}**

**●出队操作**

**x=Q.data[Q.front++];**

**出队前：**

**3**

**5**

**6**

**8**

**↑front ↑rear**

**出队后：**

**3**

**5**

**6**

**8**

**↑front ↑rear**

**注：元素8已不在队列中**

**队空时不能再进行出队操作：**

**front=rear=-1**

**↑front ↑rear**

**ElemType DeSeqQueue(SeqQueue Q)//出队**

**{**

**if(Q.rear==Q.front&&Q.front==-1)**

**{printf(“队空”);exit(0);}**

**return Q.data[Q.front++];**

**}**

**队列的真假队满现象：**

1. **队列的“真”满**

**3**

**5**

**6**

**8**

**↑front ↑rear**

1. **队列的“假”满**

**3**

**5**

**6**

**↑front ↑rear**

**此时，要想再入队可以随时将队列前移，但这要解决判断问题和大量数据移动。**

**克服这种情况的办法就是采用循环队列。**

**2.循环队列**

**将队列的数据区看成首尾相接的循环结构。这种循环结构不是物理层面上的实现，而是通过数学中的取模（MOD或%）运算使队列在逻辑上实现循环。**

1. **队列为空**

**front=rear=-1**

rear

front

**（2）将“A”，“B” 和 “C”分别入队**

front

B

rear

A

C

**(3)继续入队直到队满**

**(rear+1)%MAXSIZE=front**

**说明：队列为“空”和“满”时均有rear=front,靠此无法分清这两种情况，所以为“空”时用rear=front来判断，为“满”时靠(rear+1)%MAXSIZE=front，**

**即队尾再入队一个元素赶上队头元素。**

front

B

rear

A

C

D

E

F

G

H

**（4）“A”和“B”出队**

front

rear

C

D

E

F

G

**综合上述情况，循环队列情况如下：**

1. **对头和队尾front和rear的初始值均为-1**
2. **空队列front==rear为真**
3. **入队rear=(rear+1)%MAXSIZE**
4. **出队front=(front+1)%MAXSIZE**
5. **队满(rear+1)%MAXSIZE==front为真**
6. **队列长度=(rear-front+MAXSIZE)% MAXSIZE**

**队列长度=MAXSIZE-1**

**3.循环队列的入队和出队操作**

**（1）入队**

**void EnCirQueue(SeqQueue Q,ElemType x)**

**{**

**if((Q.rear+1)%MAXSIZE==Q.front)**

**{printf(“队满!\n”);exit(0);}**

**else**

**{**

**Q.rear=(Q.rear+1)%MAXSIZE;**

**Q.data[Q.rear]=x;**

**}**

**}**

**（2）出队**

**ElemType DeCirQueue(SeqQueue Q)**

**{**

**if(Q.front==-1)**

**{printf(“队空！\n”);exit(0);}**

**else**

**{**

**if(Q.front==Q.rear)Q.front=Q.rear=-1;**

**//出队后队空**

**else if(Q.front==MAXSIZE-1)Q.front=0;**

**//重新开始**

**else Q.front++;**

**}**

**}**

**3.2.3队列的链式存储结构**

**1.链队存储**

**由于链队列的入队限制在“队尾”，而出队列限制在“对头”，所以链队列需要设立两个指针，即队头指针（front）和队尾指针（rear），为了体现一体性，将二者封装在一个结构体内。**

**为操作方便，给链队附加一个头结点。**

**请注意以下定义较复杂一些。**

**定义：**

**typedef char ElemType;**

**typedef struct Node //链队列结点类型**

**{**

**ElemType data;**

**struct Node \*next;**

**}LQNode ,\*LinkQNode;**

**typedef struct //头和尾封装在一起的链队列类型**

**{**

**LQNode \*front,\*rear;**

**}LQueue,\*LinkQueue;**

**LinkQueue Q;**

**链队的两种情况：**

**●空链队（只有一个专用的“头结点”）**

Ø

Q🡪front

Q🡪rear

**●非空链队（一般情形）**

Ø







Q🡪front

Q🡪rear

**2.链队的基本操作**

**（1）链队列的初始化**

Ø

Q🡪front

Q🡪rear

**LinkedQueue \*LinkQueueInit( )**

**{**

**Q=(LinkQueue)malloc(sizeof(LQueue));**

**//申请头尾指针结点**

**Q🡪front=(LinkQNode)malloc(sizeof(LQNode));**

**//申请队列指针结点**

**Q🡪front🡪next=NULL;**

**Q🡪rear=Q🡪front;**

**return Q;**

**}**

1. **入队**

**根据队列特点和操作的要求，入队的结点只能加入队尾。**

**下列要入队的结点E加入队列中，即E成为新的队尾结点。**

Ø

H

K

E

Q🡪front

Q🡪rear

s

Ø

H

K

E

Q🡪front

Q🡪rear

s

**void EnLinkQueue(LinkQueue Q,ElemType x)**

**{**

**LinkLQNode s;**

**s=(LinkQNode)malloc(sizeof(LQNode));//申请入队结点**

**if(s==NULL){printf(“申请失败！”);exit(0);}**

**s🡪data=x;**

**s🡪next=NULL;**

**Q🡪rear🡪next=s;//入队**

**Q🡪rear=s;//指向新的队尾**

**}**

1. **出队**

**根据队列特点和操作的要求，出队只能在队头进行。出队后可能出现两种情况，队空和非空队。**

**●出队后为非空队(s🡪next!=NULL):**

**出队前：**

Ø

H

K

E

Q🡪front

Q🡪rear

s

**Q🡪front🡪next=s🡪next;**

**x=s🡪data;**

**free(s);**

**出队后：**

Ø

H

K

E

Q🡪front

Q🡪rear

s

Ø

K

E

Q🡪front

Q🡪rear

**●出队后为空队(s🡪next==NULL)**

Ø

H

Q🡪front

Q🡪rear

s

**Q🡪front🡪next=s🡪next;// Q🡪front🡪next=NULL;**

**Q🡪rear=Q🡪front;**

**x=s🡪data;**

**free(s);**

Ø

Q🡪front

Q🡪rear

**ElemType DeLinkQueue(LinkQueue Q)**

**{**

**ElemType x;**

**LQNode s;**

**if(Q🡪front==Q🡪rear)**

**{printf(“队空,不能出队！”);exit(0);}**

**s=Q🡪front🡪next;**

**Q🡪front🡪next=s🡪next;**

**if(s🡪next==NULL)Q🡪rear=Q🡪front;//只有一个结点可删**

**x=s🡪data;**

**free(s);**

**return x;**

**}**

**P3-3.c 链队列的创建、遍历、入队和出队**

**#include<stdio.h>**

**typedef char DataType;**

**typedef struct node**

**{**

**DataType data;**

**struct node \*next;**

**}LinkList;**

**typedef struct**

**{**

**LinkList \*rear;**

**}LinkQueue;**

**LinkQueue \*InitQueue( )//置空队**

**{**

**LinkQueue \*q;**

**q=(LinkQueue \*)malloc(sizeof(LinkQueue));**

**// 申请链队的尾指针结点**

**q->rear=(LinkList \*)malloc(sizeof(LinkList));// 申请头结点**

**q->rear->next=q->rear;// 头结点的尾指针指向自身**

**return q;**

**}**

**int IsEmpty(LinkQueue \*q)//判空队**

**{**

**return q->rear==q->rear->next;**

**}**

**void EnQueue(LinkQueue \*q,DataType value)//入队**

**{**

**LinkList \*s;**

**s=(LinkList \*)malloc(sizeof(LinkList));// 申请新结点**

**s->data=value;**

**s->next=q->rear->next;**

**q->rear->next=s;**

**q->rear=s;**

**}**

**void DeQueue(LinkQueue \*q)//出队**

**{**

**DataType x;**

**if(IsEmpty(q))**

**{**

**printf("队是空的！\n");**

**return;**

**}**

**x=q->rear->next->next->data;**

**q->rear->next->next=q->rear->next->next->next;**

**printf("出队的元素是:%c\n",x);**

**}**

**void VisitQueue(LinkQueue \*q)//遍历队列（单链表）**

**{**

**LinkList \*s=q->rear->next->next;**

**while(s!=NULL&&s!=q->rear->next)**

**{**

**printf("%2c",s->data);**

**s=s->next;**

**}**

**}**

**void main( )**

**{**

**DataType da;**

**LinkQueue \*p;**

**p=InitQueue( );//置空队**

**if(IsEmpty(p))printf("\n\n经测试队是空的！\n\n");**

**else printf("经测试队是非空的！\n\n");**

**printf("输入将要首次入队的元素字符串（换行作为结束）：");**

**scanf("%c",&da);**

**while(da!='\n')**

**{**

**EnQueue(p,da);**

**scanf("%c",&da);**

**}**

**printf("首次生成的队列是：");**

**VisitQueue(p);printf("\n\n");**

**printf("输入一个要入队的元素：");scanf("%c",&da);**

**EnQueue(p,da);**

**printf("入队后的新队列是：");**

**VisitQueue(p);printf("\n\n");**

**DeQueue(p);**

**printf("出队后的新队列是：");**

**VisitQueue(p);printf("\n\n");**

**}**

**3.3栈和队列的应用**

**●算术表达式的计算问题**

1. **将中缀表达式变为后缀表达式**

**一般的算术表达式运算符位于两个操作数的中间，称为中缀表达式，如：(5+3)\*2-8/4**

**用中缀算术运算式求值是我们通常的做法，因为要考虑算符的优先级和括号，在机器上实现却不容易。于是可以考虑在机器上实现求值更方便的方法，这就是后缀表达式（即把运算符放在操作数的后面），被称为后序(逆)波兰式。**

**将中缀表达式自左至右变为后缀表达式，方法如下：**

**先设立一个“运算符栈”和一个“操作数和运算符队列”。**

**●当读到操作数时，直接送入队列中；**

**●当读到当前运算符（不妨称为oper）时，若oper比当前栈顶的运算符优先级还高将其进栈，否则将**

**所有优先级等于或高于oper的运算符弹出，送到队列中，oper进栈；**

**●当读到运算符左括号时，将其直接进栈；**

**●当读到运算符右括号时，将靠近栈顶的第一个左**

**括号上面的运算符全部弹出，送到队列，再丢弃左括号；**

**●当读到表达式结束符时（用“=”表示），将栈中所有的运算符弹出，送到队列。**

**例如：中缀表达式A+B\*C-D/E变为后缀表达式ABC\*+DE/-**

**运算符的优先级：**

**一级：\* /**

**二级：+ -**

**三级：=**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **步骤** | **输入字符** | **栈**  **（操作符）** | **队列**  **（后缀表达式）** | **操作说明** |
| **1** | **A+B\*C-D/E=** |  | **A** | **“A”入队** |
| **2** | **A+B\*C-D/E=** | **+** | **A** | **“+”入栈** |
| **3** | **A+B\*C-D/E=** | **+** | **AB** | **“B”入队** |
| **4** | **A+B\*C-D/E=** | **+\*** | **AB** | **“\*”入栈** |
| **5** | **A+B\*C-D/E=** | **+\*** | **ABC** | **“C”入队** |
| **6** | **A+B\*C-D/E=** | **+**  **-** | **ABC\***  **ABC\*+** | **“\*”出栈后入队，因为“-”优先级低于“\*”；**  **“+”出栈后入队，因为“-”优先级等于“+”；**  **“-”入栈** |
| **7** | **A+B\*C-D/E=** | **-** | **ABC\*+D** | **“D”入队** |
| **8** | **A+B\*C-D/E=** | **-/** | **ABC\*+D** | **“/”入栈** |
| **9** | **A+B\*C-D/E=** | **-/** | **ABC\*+DE** | **“E”入队** |
| **10** | **A+B\*C-D/E=** | **-**  **=** | **ABC\*+DE/**  **ABC\*+DE/-**  **ABC\*+DE/-** | **“/”出栈后入队；**  **“-”出栈后入队；**  **“=”入栈；结束** |

**2．中缀表达式的计算程序**

**设立两个栈：运算符栈(SeqStack)和操作数栈(\_SeqStack),并在前者中事先存入一个结束符“=”。**

**自左至右顺序读入运算式字符串：**

**a.若输入的是操作数，将其入操作数栈；**

**b. 若输入的是运算符，则：**

**●输入的运算符的优先级高于运算符栈顶算符的优先级，则将该运算符入运算符栈；**

**●输入的运算符的优先级低于或等于运算符栈顶算符的优先级，则从操作数栈连续退出两个操作数，并从运算符栈退出一个运算符，同时进行相应的运算，并将运算结果存入操作数栈。对输入的运算符重新进行b过程；**

**c.若读入的是“=”，且运算符栈中栈顶也是“=”，处理结束，最后的结果放在操作数的栈顶。**

**运算符优先级：**

**一级：\*（乘） /（除）**

**二级：+（加） -（减）**

**三级：=（等号）**

|  |  |  |  |
| --- | --- | --- | --- |
| **表达式** | **运算符栈** | **操作数栈** | **操作说明** |
| **=5+3\*2-8/4=** | **=** |  | **开始，“=”入栈** |
| **=5+3\*2-8/4=** | **=** | **5** | **“5”入栈** |
| **=5+3\*2-8/4=** | **=+** | **5** | **“+”优先级高于“=”，“+”入栈** |
| **=5+3\*2-8/4=** | **=+** | **5，3** | **“3”入栈** |
| **=5+3\*2-8/4=** | **=+\*** | **5,3** | **“\*”优先级高于“+”，“\*”入栈** |
| **=5+3\*2-8/4=** | **=+\*** | **5，3，2** | **“2”入栈** |
| **=5+3\*2-8/4=** | **=+**  **=**  **=-** | **5 ，6**  **11**  **11** | **“-”优先级低于“\*”，计算3\*2=6，“\*”出栈，**  **“-”优先级等于“+”，计算5+6=11，“+”出栈**  **“-”优先级高于“=”，**  **“-”入栈** |
| **=5+3\*2-8/4=** | **=-** | **11，8** | **“8”入栈** |
| **=5+3\*2-8/4=** | **=-/** | **11，8** | **“/”优先级高于“-”，**  **“/”入栈** |
| **=5+3\*2-8/4=** | **=-/** | **11，8，4** | **“4”入栈** |
| **=5+3\*2-8/4=** | **=-**  **=** | **11，2**  **9** | **“=”优先级低于“/”，**  **计算8/4=2，“/”出栈**  **“=”优先级低于“-”，计算11-2=9，结束** |

**例p3-4.c中缀表达式的求值（栈的使用）**

**只能计算加(+)、减(-)、乘(\*)、除(/)、求余数(%)和乘方(^),不支持括号；操作数只限一位数。**

**算符的优先级由高到低是：**

**一级：^（乘方）**

**二级：\*（乘） /（除） %（求余数）**

**三级：+（加） -（减）**

**四级：=（等号）**

**#define MAXSIZE 100**

**#include<stdio.h>**

**#include<math.h>**

**typedef char DataType;**

**typedef struct node**

**{**

**DataType data[MAXSIZE+1];**

**int top;**

**}SeqStack;// 运算符栈**

**typedef int \_DataType;**

**typedef struct \_node**

**{**

**\_DataType data[MAXSIZE];**

**int top;**

**}\_SeqStack;// 操作数栈**

**void InitStack(SeqStack \*s) //运算符栈初始化**

**{**

**s🡪top=0;**

**}**

**void \_InitStack(\_SeqStack \*s) //操作数栈初始化**

**{**

**s🡪top=0;**

**}**

**void Push(SeqStack \*s,DataType x)//运算符入栈**

**{**

**if(s🡪top==MAXSIZE)printf("栈上溢!\n");**

**else s🡪data[++s🡪top]=x;**

**}**

**void \_Push(\_SeqStack \*s,\_DataType x)//操作数入栈**

**{**

**if(s🡪top==MAXSIZE)printf("栈上溢!\n");**

**else s🡪data[++s🡪top]=x;**

**}**

**int IsEmpty(SeqStack \*s)//运算符栈判空(yes=1,no=0)**

**{**

**return s🡪top==0;**

**}**

**int \_IsEmpty(\_SeqStack \*s)//操作数判栈空(yes=1,no=0)**

**{**

**return s🡪top==0;**

**}**

**DataType GetTop(SeqStack \*s)//取运算符栈顶元素**

**{**

**if(IsEmpty(s)){printf("栈空!\n");return NULL;}**

**else return s🡪data[s🡪top];**

**}**

**\_DataType \_GetTop(\_SeqStack \*s)//取操作数栈顶元素**

**{**

**if(\_IsEmpty(s))**

**{printf("栈空!\n");return NULL;}**

**else return s🡪data[s🡪top];**

**}**

**DataType Pop(SeqStack \*s)//运算符出栈**

**{**

**if(IsEmpty(s))**

**{printf("栈下溢!\n");return NULL;}**

**else return s🡪data[s🡪top--];**

**}**

**\_DataType \_Pop(\_SeqStack \*s)//操作数出栈**

**{**

**if(\_IsEmpty(s))**

**{printf("栈下溢!\n");return NULL;}**

**else return s🡪data[s🡪top--];**

**}**

**int precede(char oper) //运算符的优先级**

**{**

**switch(oper)**

**{**

**case '=': return 0;**

**case '+':**

**case '-': return 1;**

**case '\*':**

**case '/':**

**case '%': return 2;**

**case '^': return 3;**

**}**

**}**

**int operate(int x,char oper,int y)//计算结果**

**{**

**switch(oper)**

**{**

**case '+':return x+y;**

**case '-':return x-y;**

**case '\*':return x\*y;**

**case '/':if(y!=0)return x/y;**

**else printf("除数分母为0！\n");**

**case '%':return x%y;**

**case '^':return pow(x,y);**

**}**

**}**

**int calc()//表达式计算**

**{**

**SeqStack oper\_s;\_SeqStack num\_s;**

**char ch,op;int x,y;**

**InitStack(&oper\_s);Push(&oper\_s,'=');**

**\_InitStack(&num\_s);**

**printf("输入表达式,=作为结束: ");**

**scanf("%c",&ch);**

**while(ch!='='||GetTop(&oper\_s)!='=')**

**if(ch>='0'&&ch<='9')**

**{**

**\_Push(&num\_s,ch-'0');**

**scanf("%c",&ch);**

**}**

**else if(precede(ch)>precede(GetTop(&oper\_s)))**

**{**

**Push(&oper\_s,ch);**

**scanf("%c",&ch);**

**}**

**else**

**{**

**op=Pop(&oper\_s);**

**y=\_Pop(&num\_s);**

**x=\_Pop(&num\_s);**

**\_Push(&num\_s,operate(x,op,y));**

**}**

**return \_GetTop(&num\_s);**

**}**

**void main()**

**{**

**printf("表达式的结果是：%d\n",calc());**

**}**

**述评：操作数为一位容易计算，如：5=‘5’-‘0’**

**若操作数为多位，可以通过循环解决，如：**

**123=“123”=（‘1’-‘0’）\*100+（‘2’-‘0’）\*10+**

**（‘3’-‘0’）**