

栈和队列是两种特殊的线性表。从数据逻辑结构角度看,栈和队列的元素均呈现一种线性关系;从运算的角度看,栈和队列是操作受限的线性表。本章介绍栈和队列的概念、存储结构和基本运算的实现算法。

3.1 栈

3.1.1 栈的基本概念



栈是一种特殊的线性表,其特殊性体现在元素插入和删除运算上,它的插入和删除运算仅限定在表的某一端进行,不能在表中间和另一端进行。允许进行插入和删除的一端称为**栈顶**,另一端称为**栈底**。栈的插入操作称为**进栈**(或**入栈**),删除操作称为**出栈**(或**退栈**)。处于栈顶位置的数据元素称为**栈顶元素**。不含任何数据元素的栈称为**空栈**。

正是这种受限的元素插入和删除运算,使得栈表现出**先进后出**或者**后进先出**的特点。举一个例子进行说明,假设有一个很窄的死胡同,胡同里能容纳若干人,但每次只能容许一个人进出。现有 5 个人,分别编号为①~⑤,按编号的顺序依次进入此死胡同,如图 3.1(a)所示。此时若编号为④的人要退出死胡同,必须等⑤退出后才可以。若①要退出,则必须等到⑤、④、③、②依次都退出后才行,如图 3.1(b)所示。这里人进出死胡同的原则是先进去的后出来。



图 3.1 死胡同示意图

在该例中,死胡同就看作是一个栈,栈顶相当于死胡同口,栈底相当于死胡同的另一端,进、出死胡同可看作进栈、出栈操作。插入栈的示意图如图 3.2 所示。

栈的基本运算主要包括以下 6 种。

- (1) 初始化栈 $\text{InitStack}(\text{st})$: 建立一个空栈 st 。
- (2) 销毁栈 $\text{DestroyStack}(\text{st})$: 释放栈 st 占用的内存空间。
- (3) 进栈 $\text{Push}(\text{st}, x)$: 将元素 x 插入栈 st 中,使 x 成为栈 st 的栈顶元素。
- (4) 出栈 $\text{Pop}(\text{st}, x)$: 当栈 st 不空时,将栈顶元素赋给 x ,并从栈中删除当前栈顶元素。
- (5) 取栈顶元素 $\text{GetTop}(\text{st}, x)$: 若栈 st 不空,取栈顶元素 x 并返回 1; 否则返回 0。
- (6) 判断栈空 $\text{StackEmpty}(\text{st})$: 判断栈 st 是否为空栈。

包含基本运算的栈如图 3.3 所示,其中, $\text{op}_1 \sim \text{op}_6$ 表示上述 6 个基本运算。

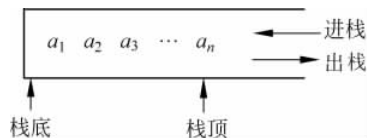


图 3.2 栈的示意图

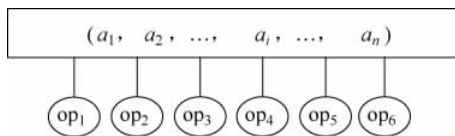


图 3.3 包含基本运算的栈

【例 3.1】 设一个栈的输入序列为 a, b, c, d , 借助一个栈(假设栈大小足够大)所得到的出栈序列不可能是_____。

- A. a, b, c, d B. b, d, c, a C. a, c, d, b D. d, a, b, c

解: a, b, c, d 序列经过栈的情况如图 3.4 所示, 根据栈的特点, 很容易得出 d, a, b, c 是不可能的, 因为 d 先出栈, 说明 a, b, c 均已在栈中, 按照进栈顺序, 从栈顶到栈底的顺序应为 c, b, a , 出栈的顺序只能是 d, c, b, a 。所以不可能的出栈序列是 D。

【例 3.2】 已知一个栈的进栈序列是 $1, 2, 3, \dots, n$, 其出栈序列是 $p_1, p_2, \dots, p_n$, 若 $p_1 = n$, 则 p_i 的值为_____。

- A. i B. $n-i$ C. $n-i+1$ D. 不确定

解: $p_1 = n$, 则出栈序列是唯一的, 即为 $n, n-1, \dots, 2, 1$, 由此推出 $p_i = n-i+1$ 。本题答案为 C。

【例 3.3】 元素 a, b, c, d, e 依次进入初始为空的栈中, 假设栈大小足够大。若元素进栈后可停留、可立即出栈, 直到所有的元素都出栈, 则所有可能的出栈序列中, 以元素 d 开头的出栈序列个数是_____。

- A. 3 B. 4 C. 5 D. 6

解: 若元素 d 第一个出栈, a, b, c 均在栈中, 从栈顶到栈底的顺序应为 c, b, a , 如图 3.5 所示, 此后合法的栈操作如下。

- (1) e 进栈, e 出栈, c 出栈, b 出栈, a 出栈, 得到的出栈序列 $decba$ 。
- (2) c 出栈, e 进栈, e 出栈, b 出栈, a 出栈, 得到的出栈序列 $dceba$ 。
- (3) c 出栈, b 出栈, e 进栈, e 出栈, a 出栈, 得到的出栈序列 $dcbea$ 。

(4) c 出栈, b 出栈, a 出栈, e 进栈, e 出栈, 得到的出栈序列 $dcbae$ 。

以元素 d 开头的出栈序列个数为 4, 本题答案为 B。

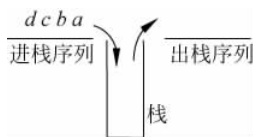


图 3.4 序列经过一个栈的情况

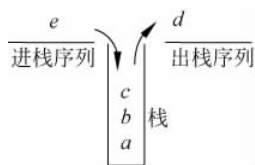


图 3.5 元素出栈的情况

3.1.2 栈的顺序存储结构

栈是一种特殊的线性表, 和线性表存储结构类似, 栈也有两种存储结构: 顺序存储结构和链式存储结构。

栈的顺序存储结构称为**顺序栈**。顺序栈通常由一个一维数组 `data` 和一个记录栈顶元素位置的变量 `top` 组成。习惯上将栈底放在数组下标小的那端, 栈顶元素由栈顶指针 `top` 所指向。顺序栈类型声明如下。

```
#define MaxSize 100 //顺序栈的初始分配空间大小
typedef struct
{
    ElemType data[MaxSize]; //保存栈中元素, 这里假设 ElemType 为 char 类型
    int top; //栈顶指针
} SqStack;
```

在上述顺序栈定义中, `ElemType` 为栈元素的数据类型, `MaxSize` 为一个常量, 表示 `data` 数组中最多可放的元素个数, `data` 元素的下标范围为 $0 \sim \text{MaxSize}-1$ 。当 $\text{top} = -1$ 时表示栈空; 当 $\text{top} = \text{MaxSize}-1$ 时表示栈满。

图 3.6 说明了顺序栈 `st` 的几种状态(假设 $\text{MaxSize} = 5$)。图 3.6(a) 表示顺序栈为栈空, 这也是初始化运算得到的结果。此时栈顶指针 $\text{top} = -1$ 。如果做出栈运算, 则会“下溢出”。

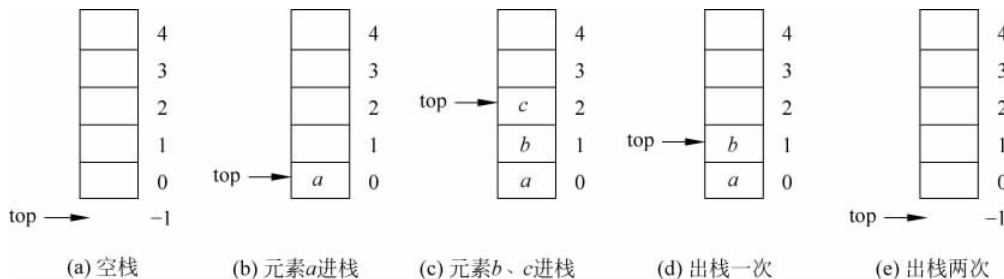


图 3.6 顺序栈的几种状态

图 3.6(b) 表示栈中只含一个元素 a , 在图 3.6(a) 的基础上用进栈运算 `Push(st, 'a')`, 可以得到这种状态。此时栈顶指针 $\text{top} = 0$ 。

图 3.6(c)表示在图 3.6(b)基础上又有两个元素 b 、 c 先后进栈,此时栈顶指针 $\text{top}=2$ 。

图 3.6(d)表示在图 3.6(c)状态下执行一次 $\text{Pop}(\text{st}, x)$ 运算得到。此时栈顶指针 $\text{top}=1$ 。故 b 为当前的栈顶元素。

图 3.6(e)表示在图 3.6(d)状态下执行两次 $\text{Pop}(\text{st}, x)$ 运算得到。此时栈顶指针 $\text{top}=-1$,又变成栈空状态。

归纳起来,对于顺序栈 st ,其初始时置 $\text{st.top}=-1$,它的 4 个要素如下。

- (1) 栈空条件: $\text{st.top}=-1$ 。
- (2) 栈满条件: $\text{st.top}=\text{MaxSize}-1$ 。
- (3) 元素 x 进栈操作: $\text{st.top}++$; 将元素 x 放在 $\text{st.data}[\text{st.top}]$ 中。
- (4) 出栈元素 x 操作: 取出栈元素 $x=\text{st.data}[\text{st.top}]$; $\text{st.top}--$ 。

顺序栈的基本运算算法如下。

1. 初始化栈运算算法

其主要操作是设定栈顶指针 top 为 -1 ,对应的算法如下。

```
void InitStack(SqStack &st)           //st 为引用型参数
{
    st.top = -1;
}
```

2. 销毁栈运算算法

这里顺序栈的内存空间是由系统自动分配的,在不再需要时由系统自动释放其空间。对应的算法如下。

```
void DestroyStack(SqStack st)
{ }
```

3. 进栈运算算法

其主要操作是: 栈顶指针加 1,将进栈元素放在栈顶处。对应的算法如下。

```
int Push(SqStack &st, ElemType x)
{
    if (st.top == MaxSize - 1)           //栈满上溢出返回 0
        return 0;
    else
    {
        st.top++;
        st.data[st.top] = x;
        return 1;                       //成功进栈返回 1
    }
}
```

4. 出栈运算算法

其主要操作是: 先将栈顶元素取出,然后将栈顶指针减 1。对应的算法如下。

```
int Pop(SqStack &st, ElemType &x)       //x 为引用型参数
```

```

{   if (st.top == -1)                //栈空返回 0
        return 0;
    else
    {   x = st.data[st.top];
        st.top--;
        return 1;                    //成功出栈返回 1
    }
}

```

5. 取栈顶元素运算算法

其主要操作是：将栈顶指针 top 处的元素取出赋给变量 x , top 保持不动。对应的算法如下。

```

int GetTop(SqStack st, ElemType &x)    //x 为引用型参数
{   if (st.top == -1)                //栈空返回 0
        return 0;
    else
    {   x = st.data[st.top];
        return 1;                    //成功取栈顶元素返回 1
    }
}

```

6. 判断栈空运算算法

其主要操作是：若栈为空 ($\text{top} == -1$) 则返回值 1, 否则返回值 0。对应的算法如下。

```

int StackEmpty(SqStack st)
{   if (st.top == -1) return 1;      //栈空返回 1
    else return 0;                   //栈不空返回 0
}

```

提示：将顺序栈类型声明和栈基本运算函数存放在 `SqStack.cpp` 文件中。

当顺序栈的基本运算函数设计好后, 给出以下程序调用这些基本运算函数, 读者可以对照程序执行结果进行分析, 进一步体会顺序栈的各种运算的实现过程。

```

#include "SqStack.cpp"                //包含前面的顺序栈基本运算函数
void main()
{   SqStack st;                      //定义一个顺序栈 st
    ElemType e;
    printf("初始化栈 st\n");
    InitStack(st);
    printf("栈 %s\n", (StackEmpty(st) == 1?"空":"不空"));
    printf("a 进栈\n"); Push(st, 'a');
    printf("b 进栈\n"); Push(st, 'b');
    printf("c 进栈\n"); Push(st, 'c');
    printf("d 进栈\n"); Push(st, 'd');
    printf("栈 %s\n", (StackEmpty(st) == 1?"空":"不空"));
}

```

```

GetTop(st,e);
printf("栈顶元素:%c\n",e);
printf("出栈次序:");
while (!StackEmpty(st))           //栈不空循环
{   Pop(st,e);                     //出栈元素 e 并输出
    printf("%c ",e);
}
printf("\n");
DestroyStack(st);
}

```

```

初始化栈st
栈空
a进栈
b进栈
c进栈
d进栈
栈不空
栈顶元素:d
出栈次序:d c b a

```

图 3.7 程序执行结果

上述程序的执行结果如图 3.7 所示。

【例 3.4】 若一个栈用数组 $\text{data}[1..n]$ 存储 (n 为一个大于 2 的正整数), 初始栈顶指针 top 为 $n+1$, 则以下元素 x 进栈的正确操作是_____。

- A. $\text{top}++$; $\text{data}[\text{top}]=x$; B. $\text{data}[\text{top}]=x$; $\text{top}++$;
 C. $\text{top}--$; $\text{data}[\text{top}]=x$; D. $\text{data}[\text{top}]=x$; $\text{top}--$;

解: 栈元素是向一端伸展的, 即从栈底向栈顶方向伸展。这里初始栈顶指针 top 为 $n+1$, 说明 $\text{data}[n]$ 端作为栈底, 在进栈时 top 应递减, 由于不存在 $\text{data}[n+1]$ 的元素, 所以在进栈时应先将 top 递减, 再将 x 放在 top 处。本题答案为 C。

3.1.3 栈的链式存储结构

栈的链式存储结构是采用某种链表结构, 栈的链式存储结构简称为链栈。这里采用单链表作为链栈, 如图 3.8 所示, 该单链表是不带头结点的, 通过首结点指针 ls 唯一标识整个单链表。同时, 单链表的首结点就是链栈的栈顶结点 (图中首结点值为 a_n 表示最近进栈的元素是 a_n), 所以 ls 也作为栈顶指针, 栈中的其他结点通过 next 域链接起来, 栈底结点的 next 域为 NULL 。因链栈本身没有容量限制, 所以不必考虑栈满的情况, 这是链栈相比顺序栈的一个优点。

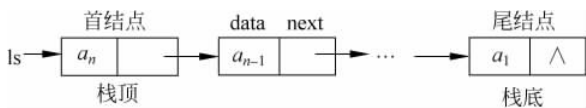


图 3.8 链栈示意图

链栈的类型定义如下。

```

typedef struct node
{   ElemType data;           //存储结点数据, 这里假设 ElemType 为 char
    struct node * next;      //指针域
} LinkStack;

```

归纳起来, 链栈 ls 初始时 $\text{ls}=\text{NULL}$, 其 4 个要素如下。

- (1) 栈空条件: $ls == \text{NULL}$ 。
- (2) 栈满条件: 不考虑。
- (3) 元素 x 进栈操作: 创建存放元素 x 的结点 p , 将其插入到栈顶位置上。
- (4) 出栈元素 x 操作: 置 x 为栈顶结点的 data 域, 并删除该结点。

链栈的基本运算算法如下。

1. 初始化栈运算算法

其主要操作是: 置 s 为 NULL 标识栈为空栈。对应的算法如下。

```
void InitStack(LinkStack * &ls)           //ls 为引用型参数
{
    ls = NULL;
}
```

2. 销毁栈运算算法

链栈的所有结点空间都是通过 malloc 函数分配的, 在不再需要时需通过 free 函数释放所有结点的空间, 与单链表销毁算法类似, 只是这里链栈是不带头结点的。对应的算法如下。

```
void DestroyStack(LinkStack * &ls)
{
    LinkStack * pre = ls, * p;
    if (pre == NULL) return;           //考虑空栈的情况
    p = pre->next;
    while (p != NULL)
    {
        free(pre);                     //释放 pre 结点
        pre = p; p = p->next;           //pre、p 同步后移
    }
    free(pre);                          //释放尾结点
}
```

3. 进栈运算算法

其主要操作是: 先创建一个新结点 p , 其 data 域值为 x ; 然后将该结点插入到开头作为新的栈顶结点。对应的算法如下。

```
int Push(LinkStack * &ls, ElemType x)    //ls 为引用型参数
{
    LinkStack * p;
    p = (LinkStack *) malloc(sizeof(LinkStack));
    p->data = x;                          //创建结点 p 用于存放 x
    p->next = ls;                          //插入 p 结点作为栈顶结点
    ls = p;
    return 1;
}
```

4. 出栈运算算法

其主要操作是: 将栈顶结点(即 ls 所指结点)的 data 域值赋给 x , 然后删除该栈顶结点。对应的算法如下。

```

int Pop(LinkStack * &ls, ElemType &x)           //ls 为引用型参数
{
    LinkStack * p;
    if (ls == NULL)                             //栈空,下溢出返回 0
        return 0;
    else                                         //栈不空时出栈元素 x 并返回 1
    {
        p = ls;                                //p 指向栈顶结点
        x = p->data;                            //取栈顶元素 x
        ls = p->next;                          //删除结点 p
        free(p);                               //释放 p 结点
        return 1;
    }
}

```

说明：尽管链栈操作时不考虑上溢出,但仍然需要考虑出栈操作时的下溢出。

5. 取栈顶元素运算算法

其主要操作是：将栈顶结点(即 ls 所指结点)的 data 域值赋给 x。对应的算法如下。

```

int GetTop(LinkStack * ls, ElemType &x)
{
    if (ls == NULL)                             //栈空,下溢出时返回 0
        return 0;
    else                                         //栈不空,取栈顶元素 x 并返回 1
    {
        x = ls->data;
        return 1;
    }
}

```

6. 判断栈空运算算法

其主要操作是：若栈为空(即 ls == NULL)则返回值 1,否则返回值 0。对应的算法如下。

```

int StackEmpty(LinkStack * ls)
{
    if (ls == NULL) return 1;
    else return 0;
}

```

提示：将链栈结点类型声明和链栈基本运算函数存放在 LinkStack.cpp 文件中。

当链栈的基本运算函数设计好后,给出以下程序调用这些基本运算函数,读者可以对照程序执行结果进行分析,进一步体会顺序栈的各种运算的实现过程。

```

#include "LinkStack.cpp"                       //包含前面的链栈基本运算函数
void main()
{
    ElemType e;
    LinkStack * st;                            //定义一个链栈 st
    printf("初始化栈 st\n");
    InitStack(st);
    printf("栈%s\n", (StackEmpty(st) == 1 ? "空" : "不空"));
    printf("a 进栈\n"); Push(st, 'a');
    printf("b 进栈\n"); Push(st, 'b');
    printf("c 进栈\n"); Push(st, 'c');
}

```

```

printf("d 进栈\n"); Push(st, 'd');
printf("栈%s\n", (StackEmpty(st) == 1 ? "空" : "不空"));
GetTop(st, e);
printf("栈顶元素:%c\n", e);
printf("出栈次序:");
while (!StackEmpty(st))           // 栈不空循环
{
    Pop(st, e);                   // 出栈元素 e 并输出
    printf("%c ", e);
}
printf("\n");
DestroyStack(st);
}

```

上述程序的执行结果如图 3.7 所示。

【例 3.5】 以下各链表均不带有头结点,其中最不适合用作链栈的链表是_____。

- A. 只有表尾指针没有表头指针的循环单链表
- B. 只有表头指针没有表尾指针的循环单链表
- C. 只有表头指针没有表尾指针的循环双链表
- D. 只有表尾指针没有表头指针的循环双链表

解: 由于各链表均不带有头结点,这里表头指针就是首结点指针。采用一种链表作为链栈时,通常将链表首结点处作为栈顶。一个链表适不适合作为链栈,就看它是否能够高效地实现栈的基本运算,而栈的主要操作是进栈和出栈。

考虑选项 A,只有表尾指针没有表头指针的循环单链表,假设表尾指针为 rear,它指向循环单链表的尾结点,如图 3.9 所示。元素 x 进栈的操作是:创建存放 x 的结点 p ,将结点 p 插入在 rear 结点之后作为栈顶结点,不改变 rear;出栈的操作是:删除 rear 结点之后的结点。两种操作的时间复杂度均为 $O(1)$ 。



图 3.9 只有表尾指针没有表头指针的循环单链表

考虑选项 B,只有表头指针没有表尾指针的循环单链表,假设表头指针为 first,它指向循环单链表的首结点,如图 3.10 所示。元素 x 进栈的操作是:创建存放 x 的结点 p ,将结点 p 插入在 first 结点之前,即 $p \rightarrow \text{next} = \text{first}$, $\text{first} = p$,但还要将其改变为循环单链表,而尾结点需要遍历所有结点找到,遍历的时间为 $O(n)$,所以进栈操作的时间复杂度为 $O(n)$;出栈的操作是:删除 first 结点,删除后仍然需要将其改变为循环单链表,所以出栈操作的时间复杂度为 $O(n)$ 。两种操作的时间复杂度均为 $O(n)$ 。

考虑选项 C 和 D,由于都是循环双链表,可以通过表头结点直接找到尾结点,在插入和删除后改为循环双链表的时间均为 $O(1)$,所以它们的进栈和出栈操作的时间复杂度都是 $O(1)$ 。

比较起来,只有表头指针没有表尾指针的循环单链表作为链栈时,进栈和出栈操作的时间

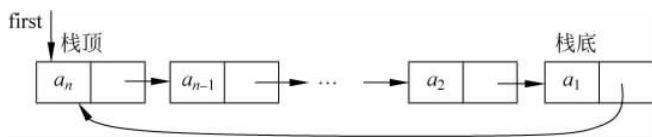


图 3.10 只有表头指针没有表尾指针的循环单链表

性能最差。本题答案为 B。

3.1.4 栈的应用示例



在较复杂的数据处理过程中,通常需要保存多个临时产生的数据,如果先产生的数据后进行处理,那么需要用栈来保存这些数据。本节通过几个经典示例说明栈的应用方法,并且算法中均采用顺序栈,实际上采用链栈完全相同。

【例 3.6】 假设以 I 和 O 分别表示进栈和出栈操作,栈的初态和终态均为空,进栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列。

(1) 下面所示的序列中哪些是合法的?

A. IOIOIOO B. IOOIOIO C. IIIIOIO D. IIIOOIOO

(2) 通过对(1)的分析,设计一个算法利用链栈的基本运算判定操作序列 str 是否合法。若合法返回 1; 否则返回 0。

解: (1) A、D 均合法,而 B、C 不合法。因为在 B 中,先进栈一次,立即出栈两次,这会造成栈下溢。在 C 中共进栈 5 次,出栈 3 次,栈的终态不为空。

归纳起来,这样的操作序列是合法的,当且仅当其中所有 I 的个数与 O 的个数相等,而且任何前缀中 I 的个数大于或等于 O 的个数。

(2) 本例用一个顺序栈 st 来判断操作序列是否合法,其中, str 为存放操作序列的字符数组, n 为该数组的元素个数。对应的算法如下。

```
#include "Sqstack.cpp"                                //包含前面的顺序栈基本运算函数
int Judge(char str[], int n)
{
    int i; ElemType x;
    SqStack st;                                         //定义一个顺序栈 st
    InitStack(st);                                     //栈初始化
    for (i=0; i<n; i++)                                //遍历 str 的所有字符
    {
        if (str[i] == 'I')                            //为'I'时进栈
            Push(st, str[i]);
        else if (str[i] == 'O')                        //为'O'时出栈
        {
            if (!Pop(st, x))                            //若栈下溢出,则返回 0
                return 0;
            DestroyStack(st);
        }
    }
    if (StackEmpty(st))                                //栈为空时返回 1; 否则返回 0
    {
        DestroyStack(st);
        return 1;
    }
}
```

```

    }
    else
    {
        DestroyStack(st);
        return 0;
    }
}

```

说明：本例的另一种解法是用 cnt 累计 I 与 O 的个数差, cnt 从 0 开始, 循环遍历 str, 遇到 I 增 1, 遇到 O 减 1。cnt 小于 0 时返回 0。循环结束后 cnt 为 0 则返回 1。

【例 3.7】 回文指的是一个字符串从前面读和从后面读都一样, 如 "abcba"、"123454321" 都是回文。设计一个算法利用栈判断一个字符串是否为回文。

解：由于回文是从前到后以及从后到前都是一样的, 所以只要将待判断的字符串颠倒, 然后与原字符串相比较, 就可以决定是否回文了。

将字符串 str 从头到尾的各个字符依次进栈到顺序栈 st 中, 由于栈的特点是后进先出, 则从栈顶到栈底的各个字符正好与字符串 str 从尾到头的各个字符相同; 然后将字符串 str 从头到尾的各个字符, 依次与从栈顶到栈底的各个字符相比较, 如果两者不相同, 则表明 str 不是回文, 在相同时继续比较; 如果相应字符全部匹配, 则说明 str 是回文。对应的算法如下。

```

#include "SqStack.cpp"                //包含前面的顺序栈基本运算函数
int Palindrome(char str[], int n)
{
    SqStack st;                       //定义一个顺序栈 st
    InitStack(st);                   //栈初始化
    int i;
    char ch;
    for (i=0; i<n; i++)               //所有字符依次进栈
        Push(st, str[i]);
    i=0;                             //从头开始遍历 str
    while (!StackEmpty(st))           //栈不空循环
    {
        Pop(st, ch);                 //出栈元素 ch
        if (ch!=str[i++])             //两字符不相同返回 0
        {
            DestroyStack(st);
            return 0;
        }
    }
    DestroyStack(st);                 //销毁栈 st
    return 1;                         //所有相应字符都相同时返回 1
}

```

【例 3.8】 设计一个算法, 判断一个可能含有小括号("("与")")、中括号("[与]")和大括号("{与}")的表达式中各类括号是否匹配。若匹配, 则返回 1; 否则返回 0。

解：设置一个顺序栈 st, 定义一个整型 flag 变量(初始为 1)。用 i 扫描表达式 exp, 当 $i < n$ 并且 flag=1 时循环: 当遇到左括号("(" "[" "{") 时, 将其进栈; 遇到 ")" "]" "}" 时, 出栈字符 ch, 若出栈失败(下溢出)或者 ch 不匹配, 则置 flag=0 退出循环, 否则直到 exp 扫描完毕为止。若栈空并且 flag 为 1 则返回 1, 否则返回 0。



例如,对于表达式" $[()]$ ",其括号不匹配,匹配过程如图 3.11(a)所示;对于表达式" $[(())]$ ",其括号是匹配的,匹配过程如图 3.11(b)所示。

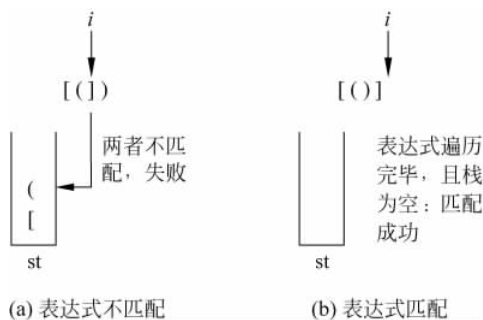


图 3.11 判断表达式括号是否匹配

对应的算法如下。

```
#include "SqStack.cpp"
int Match(char exp[], int n)
{
    SqStack st;
    InitStack(st);
    int flag=1, i=0;
    char ch;
    while (i < n && flag==1)
    {
        switch(exp[i])
        {
            case '(': case '[': case '{':
                Push(st, exp[i]); break;
            case ')':
                if (!Pop(st, ch) || ch != '(')
                    flag=0;
                break;
            case ']':
                if (!Pop(st, ch) || ch != '[')
                    flag=0;
                break;
            case '}':
                if (!Pop(st, ch) || ch != '{')
                    flag=0;
                break;
        }
        i++;
    }
    if (StackEmpty(st) && flag==1)
    {
        DestroyStack(st);
        return 1;
    }
    else
    {
        DestroyStack(st);
        return 0;
    }
}
```

//包含前面的顺序栈基本运算函数
//exp 存放表达式
//定义一个顺序栈 st
//栈初始化
//遍历表达式 exp
//各种左括号进栈
//判断栈顶是否为 '('
//出栈操作失败或者不匹配
//判断栈顶是否为 '['
//出栈操作失败或者不匹配
//判断栈顶是否为 '{'
//出栈操作失败或者不匹配
//栈空且符号匹配则返回 1
//销毁栈 st
//销毁栈 st
//否则返回 0

```

    }
}

```

【例 3.9】 设计一个算法将一个十进制正整数转换为相应的二进制数。

解：将十进制正整数转换成二进制数通常采用除 2 取余数法(称为辗转相除法)。在转换过程中,二进制数是从低位到高位次序得到的,这和通常的从高位到低位输出二进制的次序相反。为此设计一个栈,用于暂时存放每次得到的余数,当转换过程结束时,退栈所有元素便得到从高位到低位的二进制数。如图 3.12 所示是十进制数 12 转换为二进制数 1100 的过程。

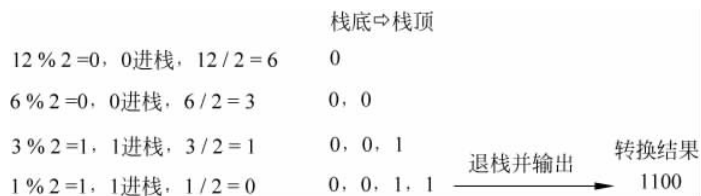


图 3.12 12 转换为二进制数的过程

对应的算法如下。

```

#include "SqStack.cpp"                //包含前面的顺序栈基本运算函数
void trans(int d, char b[])           //b 用于存放 d 转换成的二进制数的字符串
{
    SqStack st;                       //定义一个顺序栈 st
    InitStack(st);                   //栈初始化
    char ch;
    int i = 0;
    while (d != 0)
    {
        ch = '0' + d % 2;            //求余数并转换为字符
        Push(st, ch);                //字符 ch 进栈
        d /= 2;                      //继续求更高位
    }
    while (!StackEmpty(st))           //出栈并存放在数组 b 中
    {
        Pop(st, ch);
        b[i] = ch;
        i++;
    }
    b[i] = '\0';                     //加入字符串结束标志
    DestroyStack(st);                 //销毁栈 st
}

```

设计如下主函数。

```

void main()
{
    int d;
    char str[MaxSize];
    do                                //保证输入一个正整数
    {
        printf("输入一个正整数:");
        scanf("%d", &d);
    } while (d < 0);
}

```

```

trans(d, str);
printf("对应的二进制数: %s\n", str);
}

```

本程序的一次执行结果如下。

输入一个正整数: 120 ✓
对应的二进制数: 11111000



3.2 队列

3.2.1 队列的基本概念

队列(简称队)也是一种运算受限的线性表,在这种特殊的线性表上,插入限定在表的某一端进行,删除限定在表的另一端进行。队列的插入操作称为进队,删除操作称为出队。允许插入的一端称为队尾,允许删除的一端称为队头。新插入的元素只能添加到队尾,被删除的只能是排在队头的元素。

正是这种受限的元素插入和删除运算,使得队列表现出先进先出或者后进后出的特点。举一个例子进行说明,如图 3.13 所示是顾客①~⑤排队买车票的情况,排队围栏里能容纳若干人,但每次只能容许一个人进出,进入排队队列只能从进口进,某顾客买完车票后只能从出口出。从中看到,顾客进入排队队列的顺序是①~⑤,那么买票并离开队列的顺序也只能是①~⑤。也就是说,顾客进出排队队列的原则是先进去的先出来。

归纳起来,一个队列的示意图如图 3.14 所示。



图 3.13 顾客排队买车票

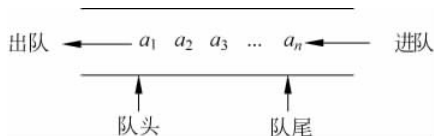


图 3.14 队列的示意图

队列的基本运算如下。

- (1) 初始化队列 $\text{InitQueue}(\text{qu})$: 建立一个空队 qu 。
- (2) 销毁队 $\text{DestroyQueue}(\text{qu})$: 释放队列 qu 占用的内存空间。
- (3) 进队 $\text{EnQueue}(\text{qu}, x)$: 将 x 插入到队列 qu 的队尾。
- (4) 出队 $\text{DeQueue}(\text{qu}, x)$: 将队列 qu 的队头元素出队并赋给 x 。
- (5) 取队头元素 $\text{GetHead}(\text{qu}, x)$: 取出队列 qu 的队头元素并赋给 x , 但该元素不出队。
- (6) 判断队空 $\text{QueueEmpty}(\text{qu})$: 判断队列 qu 是否为空。

包含基本运算的队列如图 3.15 所示,其中, $\text{op}_1 \sim$

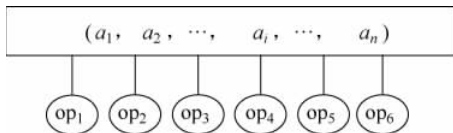


图 3.15 包含基本运算的队列

op₆ 表示上述 6 个基本运算。

【例 3.10】 以下属于队列的基本运算的是_____。

- A. 对队列中的元素排序 B. 取出最近进队的元素
C. 在队列中某元素之前插入元素 D. 删除队头元素

解: 删除队头元素即出队,属队列的一种基本运算,其他均不是队列的基本运算。本题答案为 D。

【例 3.11】 设栈 S 和队列 Q 的初始状态均为空,元素 $a、b、c、d、e、f、g$ 依次进入栈 S。若每个元素出栈后立即进入队列 Q,且 7 个元素出列的顺序是 $b、d、c、f、e、a、g$,则栈 S 的容量至少是_____。

- A. 1 B. 2 C. 3 D. 4

解: 由于队列不改变进、出队元素的次序,即 $a_1、a_2、\dots、a_n$ 依次进入一个队列,出队序列只有 $a_1、a_2、\dots、a_n$ 一种,所以本题变为通过一个栈将 $a、b、c、d、e、f、g$ 序列变为 $b、d、c、f、e、a、g$ 序列时栈空间至少多大。其过程如表 3.1 所示,从中可以看到,栈中最多有三个元素,即栈大小至少为 3。本题答案为 C。

表 3.1 由 $a、b、c、d、e、f、g$ 序列通过一个栈得到 $b、d、c、f、e、a、g$ 序列的过程

操 作	栈 S	出栈序列
a 进栈	a	
b 进栈	ab	
b 出栈	a	b
c 进栈	ac	b
d 进栈	acd	b
d 出栈	ac	bd
c 出栈	a	bdc
e 进栈	ae	bdc
f 进栈	$ae f$	bdc
f 出栈	ae	$bdc f$
e 出栈	a	$bdc fe$
a 出栈		$bdc fea$
g 进栈	g	$bdc fea$
g 出栈		$bdc fea g$

3.2.2 队列的顺序存储结构

与栈类似,队列通常有两种存储结构,即顺序存储结构和链式存储结构。队列的顺序存储结构简称为**顺序队**,它由一个一维数组(用于存储队列中元素)及两个分别指示队头和队尾的变量组成(因为队列不同于栈,栈只要一端发生改变,而队列的两端都可能发生改变),这两个变量分别称为“队头指针”和“队尾指针”。通常约定队尾指针指示队尾元素的当前位置,队头指针指示队头元素的前一个位置。



顺序队的类型声明如下。

```
#define MaxSize 20                                //指定队列的容量
typedef struct
{
    ElemType data[MaxSize];                        //保存队中元素,这里假设 ElemType 为 char 类型
    int front, rear;                                //队头和队尾指针
} SqQueue;
```

顺序队的定义为一个结构类型,该类型变量有三个域: data、front、rear。其中, data 为存储队列中元素的一维数组。队头指针 front 和队尾指针 rear 定义为整型变量,实际取值范围是 $0 \sim \text{MaxSize}-1$ 。

图 3.16 表示了顺序队列 sq(假设 $\text{MaxSize}=5$)的几种状态。

图 3.16(a)表示队列的初始状态, $\text{sq.rear}=\text{sq.front}=-1$ 。

图 3.16(b)表示元素 a 进队后, $\text{sq.rear}=0, \text{sq.front}=-1$ 。

图 3.16(c)表示元素 b、c、d、e 依次进队后, $\text{sq.rear}=4, \text{sq.front}=-1$ 。

图 3.16(d)表示元素 a、b 出队后, $\text{sq.rear}=4, \text{sq.front}=1$ 。

图 3.16(e)表示元素 c、d、e 出队后, $\text{sq.rear}=\text{sq.front}=4$ 。

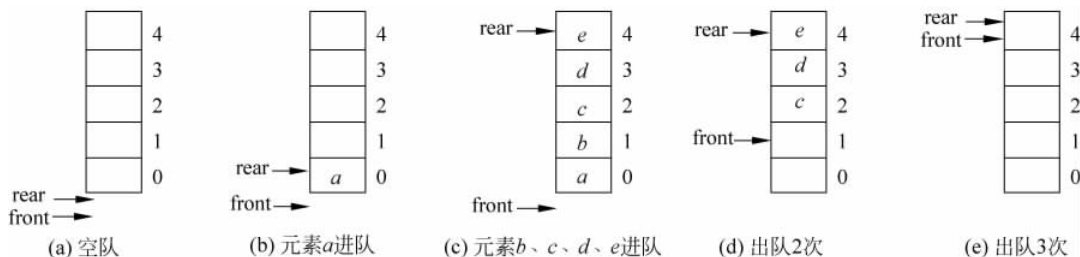


图 3.16 顺序队的几种状态

从图 3.16 中可以看到,在队列刚建立时,先对它进行初始化,令 $\text{front}=\text{rear}=-1$ 。每当进队一个元素时,让队尾指针 rear 增 1,再将新元素放在 rear 所指位置,也就是说元素进队只会引起 rear 的变化,而不会导致 front 的变化,而且 rear 指示刚进队的元素(队尾元素)位置。队头指针 front 则不同,每当出队一个元素时,让队头指针 front 增 1,再把 front 所指位置上的元素取出,也就是说元素出队只会引起 front 的变化,而不会导致 rear 的变化,而且 front 所指示的元素已出队了,它实际指示的是当前队列中队头元素的前一位置。

从图 3.16 中可以看到,图 3.16(a)和图 3.16(e)都是队空的情况,均满足 $\text{front}=\text{rear}$ 的条件,所以可以将 $\text{front}=\text{rear}$ 作为队空的条件。那么队满的条件如何设置呢?受顺序栈的启发,似乎很容易得到队满的条件为 $\text{rear}=\text{MaxSize}-1$ 。显然这里有问题,因为图 3.16(d)和图 3.16(e)都满足这个“队满”的条件,而实际上队列并没有满,这种因为队满条件设置不合理而导致的“溢出”称为假溢出,也就是说这种“溢出”并不是真正的溢出,尽管队满条件成立了,但队列中还有多个存放元素的空位置。

为了能够充分地使用数组中的存储空间,可以把数组的前端和后端连接起来,形成一个环

形的表,即把存储队列元素的表从逻辑上看成一个环,这个环形的表叫作循环队列或环形队列。图 3.17 表示了循环队列 sq 的几种状态。

图 3.17(a)表示队列的初始状态, $\text{sq.rear}=\text{sq.front}=0$ 。

图 3.17(b)表示元素 a 进队后, $\text{sq.rear}=1, \text{sq.front}=0$ 。

图 3.17(c)表示元素 b, c, d, e 依次进队后, $\text{sq.rear}=0, \text{sq.front}=0$ 。

图 3.17(d)表示元素 a, b 出队后, $\text{sq.rear}=0, \text{sq.front}=2$ 。

图 3.17(e)表示元素 c, d, e 出队后, $\text{sq.rear}=\text{sq.front}=0$ 。

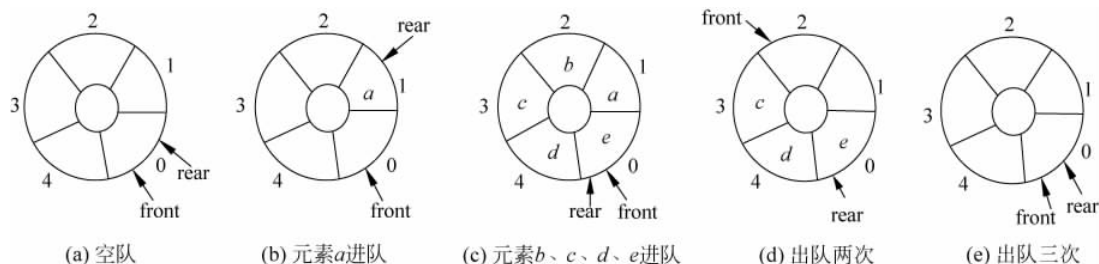


图 3.17 循环队列的几种状态

循环队列首尾相连,当队头指针 $\text{front}=\text{MaxSize}-1$ 后,再前进一个位置就自动到 0,这可以利用除法取余的运算(MOD, C/C++ 语言中运算符为%)来实现。所以队头队尾指针进 1 的操作如下。

队头指针进 1: $\text{front}=(\text{front}+1) \bmod \text{MaxSize}$

队尾指针进 1: $\text{rear}=(\text{rear}+1) \bmod \text{MaxSize}$

循环队列的队头指针和队尾指针初始化时都置为 0: $\text{front}=\text{rear}=0$ 。在队尾插入新元素和删除队头元素时,相关指针都循环进 1。当它们进到 $\text{MaxSize}-1$ 时,并不表示队空间满了,只要有需要,利用 MOD 运算可以前进到数组的 0 号位置。

如果循环队列读取元素的速度快于存入元素的速度,队头指针会很快追上队尾指针,一旦到达 $\text{front}=\text{rear}$ 时,则队列变成空队列。反之,如果队列存入元素的速度快于读取元素的速度,队尾指针很快就会赶上队头指针,一旦队列满就不能再加入新元素了。

在循环队列中仍不能区分队空和队满,因为图 3.17(a)、图 3.17(c)和图 3.17(e)都满足条件 $\text{front}=\text{rear}$,而图 3.17(a)和图 3.17(e)为队空,图 3.17(c)为队满。那么如何解决这一问题呢? 仍设置队空条件为 $\text{front}=\text{rear}$,将队满条件设置为 $(\text{rear}+1) \bmod \text{MaxSize}=\text{front}$,也就是说,当 rear 指到 front 的前一位置时就认为队列满了,显然在这样设置的队满条件下,队满条件成立时队中还有一个空闲单元,也就是说这样的队中最多只能进队 $\text{MaxSize}-1$ 个元素。

说明: 上述设置循环队列队满条件的方法不是最理想的,因为队中最多只能放入 $\text{MaxSize}-1$ 个元素,但它是一种最简单的方法,后面例 3.15 就在这个基础上进行了改进,读者可以体会两种方法的差异。

归纳起来,上述设置的循环队列 sq 的 4 个要素如下。

- (1) 队空条件: $\text{sq.front} == \text{sq.rear}$ 。
- (2) 队满条件: $(\text{sq.rear} + 1) \% \text{MaxSize} == \text{sq.front}$ 。
- (3) 进队操作: sq.rear 循环进 1; 元素进队。
- (4) 出队操作: sq.front 循环进 1; 元素出队。

循环队列的基本运算算法如下。

1. 初始化队列运算算法

其主要操作是: 设置 $\text{sq.front} = \text{sq.rear} = 0$ 。对应的算法如下。

```
void InitQueue(SqQueue &sq)           //sq 为引用型参数
{
    sq.rear = sq.front = 0;           //指针初始化
}
```



2. 销毁队列运算算法

这里顺序队的内存空间是由系统自动分配的,在不再需要时由系统自动释放其空间。对应的算法如下。

```
void DestroyQueue(SqQueue sq)
{ }
```

3. 进队运算算法

其主要操作是: 先判断队列是否已满,若不满,让队尾指针循环进 1,在该位置存放 x 。对应的算法如下。

```
int EnQueue(SqQueue &sq, ElemType x)
{
    if ((sq.rear + 1) \% MaxSize == sq.front) //队满上溢出
        return 0;
    sq.rear = (sq.rear + 1) \% MaxSize;      //队尾循环进 1
    sq.data[sq.rear] = x;
    return 1;
}
```

4. 出队运算算法

其主要操作是: 先判断队列是否已空,若不空,让队头指针循环进 1,将该位置的元素值赋给 x 。对应的算法如下。

```
int DeQueue(SqQueue &sq, ElemType &x) //x 为引用型参数
{
    if (sq.rear == sq.front)           //队空下溢出
        return 0;
    sq.front = (sq.front + 1) \% MaxSize; //队头循环进 1
    x = sq.data[sq.front];
    return 1;
}
```

5. 取队头元素运算算法

其主要操作是：先判断队列是否已空，若不空，将队头指针前一个位置的元素值赋给 x 。对应的算法如下。

```
int GetHead(SqQueue sq, ElemType &x)           //x 为引用型参数
{
    if (sq.rear == sq.front)                    //队空下溢出
        return 0;
    x = sq.data[(sq.front + 1) % MaxSize];
    return 1;
}
```

6. 判断队空运算算法

其主要操作是：若队列为空，则返回 1；否则返回 0。对应的算法如下。

```
int QueueEmpty(SqQueue sq)
{
    if (sq.rear == sq.front) return 1;
    else return 0;
}
```

提示：将顺序队类型声明及其基本运算函数存放在 SqQueue.cpp 文件中。

当顺序队的基本运算函数设计好后，给出以下程序调用这些基本运算函数，读者可以对照程序执行结果进行分析，进一步体会顺序队的各种运算的实现过程。

```
#include "SqQueue.cpp"                        //包含前面的顺序队基本运算函数
void main()
{
    SqQueue sq;                               //定义一个顺序队 sq
    ElemType e;
    printf("初始化队列\n");
    InitQueue(sq);
    printf("队%s\n", (QueueEmpty(sq) == 1 ? "空" : "不空"));
    printf("a 进队\n"); EnQueue(sq, 'a');
    printf("b 进队\n"); EnQueue(sq, 'b');
    printf("c 进队\n"); EnQueue(sq, 'c');
    printf("d 进队\n"); EnQueue(sq, 'd');
    printf("队%s\n", (QueueEmpty(sq) == 1 ? "空" : "不空"));
    GetHead(sq, e);
    printf("队头元素：%c\n", e);
    printf("出队次序：");
    while (!QueueEmpty(sq))                  //队不空循环
    {
        DeQueue(sq, e);                      //出队元素 e
        printf("%c ", e);                    //输出元素 e
    }
    printf("\n");
    DestroyQueue(sq);                         //销毁顺序队 sq
}
```

上述程序的执行结果如图 3.18 所示。

说明：顺序队有循环队列和非循环队列两种，前者把存储队列元素的表从逻辑上看成一个环，从而新进队的元素可以覆盖已出队元素的空间，提高存储空间利用率。但有些情况下要利用所有进队的元素求解时，只能采用非循环队列。

【例 3.12】 若用一个大小为 6 的数组来实现循环队列，队头指针 front 指向队列中队头元素的前一个位置，队尾指针 rear 指向队尾元素的位置。若当前 rear 和 front 的值分别为 0 和 3，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为_____。

A. 1 和 5

B. 2 和 4

C. 4 和 2

D. 5 和 1

解：当前有 rear=0，进队两个元素后，rear 循环递增 2，rear=2；当前有 front=3，出队一个元素后，front 循环递增 1，front=4。本题答案为 B。

【例 3.13】 对于循环队列，写出求队列中元素个数的公式，并编写相应的算法。

解：循环队列中队头、队尾指针变化主要有如图 3.19 所示的两种情况，归纳起来，循环队列元素个数的计算公式如下。

$$(\text{rear} - \text{front} + \text{MaxSize}) \% \text{MaxSize}$$

对应的算法如下。

```
int Count(SqQueue sq)
{
    return (sq.rear - sq.front + MaxSize) % MaxSize;
}
```

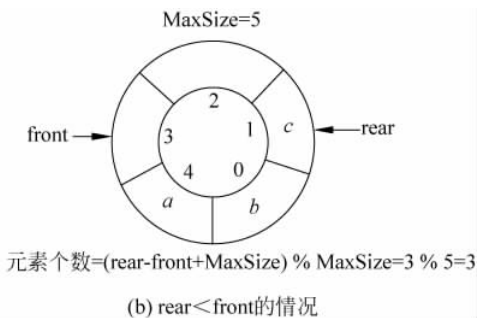
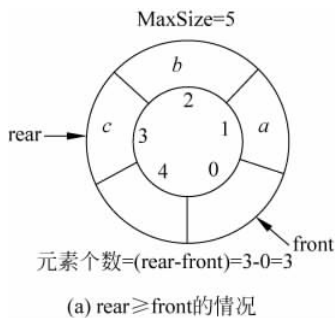


图 3.19 求循环队中元素个数的两种情况

【例 3.14】 已知循环队列存储在一维数组 $A[0..n-1]$ 中，且队列非空时 front 和 rear 分别指向队头元素和队尾元素。若初始时队列空，且要求第一个进入队列的元素存储在 $A[0]$ 处，则初始时 front 和 rear 的值分别是_____。

A. 0, 0

B. 0, n-1

C. n-1, 0

D. n-1, n-1

初始化队列
队空
a 进队
b 进队
c 进队
d 进队
队不空
队头元素: a
出队次序: a b c d



图 3.18 程序的执行结果

解: 在循环队列中,进队操作是队尾指针 rear 循环加 1,再在该处放置进队的元素,本题要求第一个进入队列的元素存储在 $A[0]$ 处,则 rear 应为 $n-1$,因为这样有 $(rear+1)\%n=0$ 。而队头指针 front 指向队头元素,此时队头位置为 0,所以 front 的初值为 0。本题答案为 B。

提示: 在一般的数据结构教科书中,循环队列的队头指针 front 设计为指向队列中队头元素的前一个位置,而队尾指针 rear 指向队尾元素的位置,本题的 front 和 rear 有所不同。

【例 3.15】 如果用一个大小为 MaxSize 的数组表示环形队列,该队列只有一个队头指针 front,不设队尾指针 rear,而改置一个计数器 count,用以记录队列中的元素个数。

(1) 队列中最多能容纳多少个元素?

(2) 设计实现队列基本运算的算法。

解: 依题意,设计队列的类型如下。

```
typedef struct
{
    ElemType data[MaxSize];           //存放队列中的元素
    int front;                         //队头指针
    int count;                         //队列中元素个数
} SQueue;
```

(1) 队列中最多可容纳 MaxSize 个元素,因为这里不需要空出一个位置以区分队列空和队列满的情况。

(2) 队列 sq 为空的条件是: $sq.count==0$; 队列为满的条件是: $sq.count==MaxSize$ 。在队头指针 sq.front 和队中元素个数 sq.count 已知时,计算队尾元素的位置的公式是:

$$\text{队尾元素位置} = (sq.front + sq.count) \% MaxSize$$

在这种队列上实现队列的基本运算算法如下。

```
//----- 队初始化算法 -----
void InitQueue(SQueue &qu)
{
    qu.front = qu.count = 0;
}
//----- 销毁队列算法 -----
void DestroyQueue(SQueue sq)
{
}
//----- 元素进队算法 -----
int EnQueue(SQueue &sq, ElemType x)
{
    if (sq.count == MaxSize) return 0;           //队满
    sq.count++;                                  //队中元素个数增 1
    sq.data[(sq.front + sq.count) \% MaxSize] = x;
    return 1;
}
//----- 出队元素算法 -----
int DeQueue(SQueue &sq, ElemType &x)
{
    if (sq.count == 0) return 0;                 //队空
    sq.count--;                                  //队中元素个数减 1
    sq.front = (sq.front + 1) \% MaxSize;
    x = sq.data[sq.front];
    return 1;
}
//----- 取队头元素算法 -----
```

```

int GetHead(SQueue sq, ElemType &x)
{
    if (sq.count==0) return 0;           //队空
    x=sq.data[(sq.front+1)%MaxSize];
    return 1;
}

//-----判队空算法-----
int QueueEmpty(SQueue sq)
{
    if (sq.count==0) return 1;           //队空返回 1
    else return 0;                       //队不空返回 0
}

```

3.2.3 队列的链式存储结构

队列的链式存储结构简称为**链队**,它实际上是一个同时带有队头指针 front 和队尾指针 rear 的单链表。队头指针指向队头结点,队尾指针指向队尾结点即单链表的最后一个结点,并将队头和队尾指针结合起来构成链队结点,如图 3.20 所示。

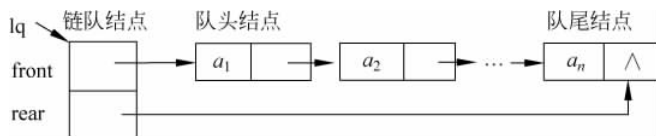


图 3.20 链队示意图

其中,链队的数据结点类型声明如下。

```

typedef struct QNode
{
    ElemType data;           //存放队中元素
    struct QNode * next;     //指向下一个结点的指针
} QType;                    //链队中数据结点的类型

```

链队结点的类型声明如下。

```

typedef struct qptr
{
    QType * front;           //队头指针
    QType * rear;            //队尾指针
} LinkQueue;                //链队结点类型

```

在这样的链队中,队空的条件是 $lq \rightarrow front == NULL$ 或 $lq \rightarrow rear == NULL$ (这里采用 $lq \rightarrow front == NULL$ 的队空条件)。一般情况下,链队是不会出现队满的情况的。归纳起来,链队 lq 的 4 个要素如下。

- (1) 队空条件: $lq \rightarrow front == NULL$ 。
- (2) 队满条件: 不考虑(因为每个结点是动态分配的)。
- (3) 进队操作: 创建结点 p , 将其插入到队尾, 并由 $lq \rightarrow rear$ 指向它。
- (4) 出队操作: 删除队头结点。

在链队上实现队列基本运算算法如下。

1. 初始化队列运算算法

其主要操作是：创建链队结点，并置该结点的 rear 和 front 均为 NULL。对应的算法如下。

```
void InitQueue(LinkQueue * &lq)           //lq 为引用型参数
{
    lq=(LinkQueue *) malloc(sizeof(LinkQueue));
    lq->rear=lq->front=NULL;             //初始时队头和队尾指针均为空
}
```

2. 销毁队列运算算法

链队的所有结点空间都是通过 malloc 函数分配的,在不再需要时需通过 free 函数释放所有结点的空间。在销毁队列 lq 时,先像释放单链表一样释放队中所有数据结点,然后释放链队结点 lq。对应的算法如下。

```

void DestroyQueue(LinkQueue * &lq)
{
    QType * pre=lq->front, * p;
    if (pre!=NULL) //非空队的情况
    {
        if (pre==lq->rear) //只有一个数据结点的情况
            free(pre); //释放 pre 结点
        else //有两个或多个数据结点的情况
        {
            p=pre->next;
            while (p!=NULL)
            {
                free(pre); //释放 pre 结点
                pre=p; p=p->next; //pre,p 同步后移
            }
            free(pre); //释放尾结点
        }
    }
    free(lq); //释放链队结点
}

```

3. 进队运算算法

其主要操作是：创建一个新结点 s ，将其链接到链队的末尾，并由 rear 指向它。对应的算法如下。

```
int EnQueue(LinkQueue * &lq, ElemType x) //lq 为引用型参数
{
    QType * s;
    s=(QType *)malloc(sizeof(QType)); //创建新结点 s,插入到链队的末尾
    s->data=x;s->next=NULL;
    if (lq->front==NULL) //原队为空队的情况
        lq->rear=lq->front=s; //front 和 rear 均指向 s 结点
    else //原队不为空队的情况
    {
        lq->rear->next=s; //将结点 s 链到队尾
        lq->rear=s; //rear 指向它
    }
    return 1;
}
```

4. 出队运算算法

其主要操作是：将 front 指向结点的 data 域值赋给 x ，并删除该结点。对应的算法如下。

```
int DeQueue(LinkQueue * &lq, ElemType &x)    //lq, x 均为引用型参数
{
    QType * p;
    if (lq->front == NULL)                    //原队为空队的情况
        return 0;
    p=lq->front;                               //p 指向队头结点
    x=p->data;                                 //取队头元素值
    if (lq->rear == lq->front)                  //若原队列中只有一个结点,删除后队列变空
        lq->rear=lq->front=NULL;
    else                                       //原队有两个或以上结点的情况
        lq->front=lq->front->next;
    free(p);
    return 1;
}
```

5. 取队头元素运算算法

其主要操作是：将 front 指向结点的 data 域值赋给 x 。对应的算法如下。

```
int GetHead(LinkQueue * lq, ElemType &x)    //x 为引用型参数
{
    if (lq->front == NULL)                    //原队为队空的情况
        return 0;
    x=lq->front->data;
    return 1;
}
```

6. 判断队空运算算法

其主要操作是：若链队为空，则返回 1；否则返回 0。对应的算法如下。

```
int QueueEmpty(LinkQueue * lq)
{
    if (lq->front == NULL) return 1;          //队空返回 1
    else return 0;                            //队不空返回 0
}
```

提示：将链队结点类型声明及其基本运算函数存放在 LinkQueue.cpp 文件中。

当链队的基本运算函数设计好后，给出以下程序调用这些基本运算函数，读者可以对照程序执行结果进行分析，进一步体会链队的各种运算的实现过程。

```
#include "LinkQueue.cpp"                    //包含前面的链队基本运算函数
void main()
{
    LinkQueue * lq;                          //定义一个链队 lq
    ElemType e;
    printf("初始化队列\n");
    InitQueue(lq);
    printf("队%s\n", (QueueEmpty(lq) == 1 ? "空" : "不空"));
    printf("a 进队\n"); EnQueue(lq, 'a');
    printf("b 进队\n"); EnQueue(lq, 'b');
```

```

printf("c 进队\n"); EnQueue(lq, 'c');
printf("d 进队\n"); EnQueue(lq, 'd');
printf("队%s\n", (QueueEmpty(lq) == 1 ? "空" : "不空"));
GetHead(lq, e);
printf("队头元素: %c\n", e);
printf("出队次序:");
while (!QueueEmpty(lq))           // 队不空循环
{
    DeQueue(lq, e);                // 出队元素 e
    printf("%c ", e);              // 输出元素 e
}
printf("\n");
DestroyQueue(lq);
}

```

上述程序的执行结果如图 3.18 所示。

【例 3.16】 若使用不带头结点的循环链表来表示队列, lq 是这样的链表中尾结点指针, 如图 3.21 所示。试基于此结构给出队列的相关运算算法。

解: 这里使用的循环链表不带头结点, lq 始终指向队尾结点, $lq \rightarrow next$ 即为队头结点。当 $lq == NULL$ 时队列为空, $lq \rightarrow next == lq$ 表示队列中只有一

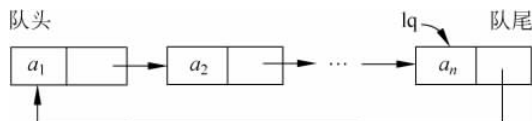


图 3.21 用循环单链表表示链队

个结点。队列的相关运算算法如下。

```

typedef struct node
{
    ElemType data;           // 数据域
    struct node * next;      // 指针域
} QNode;                    // 链队中数据结点类型

// --- 初始化队列运算算法 ---
void InitQueue(QNode * &lq)
{
    lq = NULL;
}

// --- 销毁链队 ---
void DestroyQueue(QNode * &lq)
{
    QNode * pre, * p;
    if (lq != NULL)
    {
        if (lq->next == lq)           // 原队中只有一个结点
            free(lq);
        else                           // 原队中有两个或以上的结点
        {
            pre = lq;
            p = pre->next;
            while (p != lq)
            {
                free(pre);
                pre = p;
            }
        }
    }
}

```

```

        p=p->next;                //pre 和 p 同步后移
    }
    free(pre);                    //释放最后一个结点
}
}

//-----进队运算算法-----
void EnQueue(QNode * &lq, ElemType x)
{
    QNode * s;
    s=(QNode *)malloc(sizeof(QNode)); //创建存放 x 的结点 s
    s->data=x;                        //原为空队
    if (lq==NULL)
    {
        lq=s;
        lq->next=lq;                //构成循环单链表
    }
    else                             //原队不空,结点 s 插到队尾,并由 lq 指向它
    {
        s->next=lq->next;
        lq->next=s;
        lq=s;                       //lq 指向结点 s
    }
}

//-----出队运算算法-----
int DeQueue(QNode * &lq, ElemType &x)
{
    QNode * s;
    if (lq==NULL) return 0;          //原为队空
    if (lq->next==lq)                //原队只有一个结点
    {
        x=lq->data;
        free(lq);
        lq=NULL;
    }
    else                             //原队有两个或以上的结点,删除队头结点
    {
        s=lq->next;                  //将结点 lq 之后的结点 s 删除
        x=s->data;
        lq->next=s->next;
        free(s);
    }
    return 1;
}

//-----取队头元素运算算法-----
int GetHead(QNode * lq, ElemType &x)
{
    if (lq==NULL) return 0;          //原为队空
    x=lq->next->data;
    return 1;
}

//-----判断队空运算算法-----
int QueueEmpty(QNode * lq)

```

```

{   if (lq == NULL) return 1;           //队空返回 1
    else return 0;                     //队不空返回 0
}

```

【例 3.17】 以下各种不带头结点的链表中,最不适合用作链队的是_____。

- A. 只带队首指针的非循环双链表 B. 只带队首指针的循环双链表
C. 只带队尾指针的循环双链表 D. 只带队尾指针的循环单链表

解: 队列最基本的运算是进队和出队。链队的队头和队尾分别在链表的前、后端,即进队在链尾进行,出队在链首进行。

对于选项 A 的只带队首指针的非循环双链表,在末尾插入一个结点(进队)的时间复杂度为 $O(n)$,其他选项的链表完成同样操作的时间复杂度均为 $O(1)$,所以相比较而言,选项 A 的存储结构最不适合用作链队。本题答案为 A。

3.2.4 队列的应用示例

在较复杂的数据处理过程中,通常需要保存多个临时产生的数据,如果先产生的数据先进行处理,那么需要用队列来保存这些数据。下面通过一个典型示例说明队列的应用。



【例 3.18】 设计一个程序,反映病人到医院看病、排队看医生的过程。

解: (1) 设计存储结构。

病人排队看医生采用先到先看的方式,所以要用到一个队列。由于病人人数具有较大的不确定性,这里采用一个带头结点的单链表作为队列的存储结构。为了简单,病人通过其姓名来唯一标识。例如,有 Smith、John 和 Mary 三个病人依次排队的病人队列如图 3.22 所示。

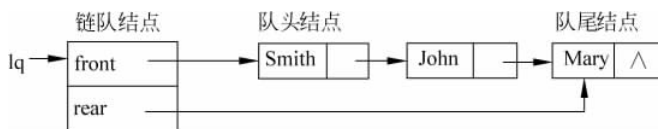


图 3.22 病人队列

病人链队结点类型如下。

```

typedef struct
{   QType * front;           //指向队头病人结点
    QType * rear;           //指向队尾病人结点
} LQueue;                   //病人链队类型

```

病人链队中的结点类型如下。

```

typedef struct Lnode
{   char data[10];           //存放患者姓名
    struct Lnode * next;     //指针域
} QType;                   //链队中结点类型

```

(2) 设计运算算法。

在病人队列设计好后,设计相关的基本运算算法,如队列初始化、进队和出队等,这些算法如下。

```
//——初始化队列运算算法——
void InitQueue(LQueue * &lq)
{   lq=(LQueue *)malloc(sizeof(LQueue));
    lq->rear=lq->front=NULL;           //初始时队头和队尾指针均为空
}

//———销毁链队———
void DestroyQueue(LQueue * &lq)
{   QType * pre=lq->front, * p;
    if (pre!=NULL)                     //非空队的情况
    {   if (pre==lq->rear)               //只有一个数据结点的情况
        free(pre);                     //释放 pre 结点
        else                             //有两个或多个数据结点的情况
        {   p=pre->next;
            while (p!=NULL)
            {   free(pre);               //释放 pre 结点
                pre=p; p=p->next;        //pre、p 同步后移
            }
        }
        free(pre);                     //释放尾结点
    }
    free(lq);                           //释放链队结点
}

//———进队运算算法———
void EnQueue(LQueue * &lq, char x[])
{   QType * s;
    s=(QType *)malloc(sizeof(QType)); //创建新结点,插入到链队的末尾
    strcpy(s->data, x); s->next=NULL;
    if (lq->front==NULL)                //原队为空队的情况
        lq->rear=lq->front=s;           //front 和 rear 均指向 s 结点
    else                                 //原队不为空队的情况
    {   lq->rear->next=s;                //将结点 s 链到队尾
        lq->rear=s;                     //rear 指向结点 s
    }
}

//———出队运算算法———
int DeQueue(LQueue * &lq, char x[])
{   QType * p;
    if (lq->front==NULL)                 //原队为空队的情况
        return 0;
    p=lq->front;                          //p 指向队头结点
    strcpy(x, p->data);                   //取队头元素值
    if (lq->rear==lq->front)              //若原队列中只有一个结点,删除后队列变空
        lq->rear=lq->front=NULL;
```

```

        else
            lq->front=lq->front->next;
        free(p);
        return 1;
    }

//-----判断队空运算算法-----
int QueueEmpty(LQueue * lq)
{
    if (lq->front==NULL) return 1;
    else return 0;
}

//-----输出队中所有元素的算法-----
int DispQueue(LQueue * lq)
{
    QType * p;
    if (QueueEmpty(lq)) return 0;
    else
    {
        p=lq->front;
        while (p!=NULL)
        {
            printf("%s ",p->data);
            p=p->next;
        }
        printf("\n");
        return 1;
    }
}

```

(3) 设计主函数。

然后设计如下主函数通过简单的提示性菜单方式来操作各个功能。

```

void main()
{
    int sel,flag=1;
    char name[10];
    LQueue * lq;
    InitQueue(lq);
    while (flag==1)
    {
        printf("1:排队 2:看医生 3:查看排队 0:下班 请选择:");
        scanf("%d",&sel);
        switch(sel)
        {
            case 0:
                if (!QueueEmpty(lq))
                    printf(" >>请排队的患者明天就医\n");
                DestroyQueue(lq);
                flag=0;
                break;
            case 1:
                printf(" >>输入患者姓名:");
                scanf("%s",name);
                EnQueue(lq,name);

```

```

        break;
    case 2:                                     //一个病人看医生
        if (!DeQueue(lq, name))
            printf(" >>没有排队的患者\n");
        else
            printf(" >>患者%s 看医生\n", name);
        break;
    case 3:                                     //查看目前病人排队情况
        printf(" >>排队患者:");
        if (!DispQueue(lq))
            printf(" >>没有排队的患者\n");
        break;
    }
}
}

```

(4) 执行结果。

本程序的一次执行结果如图 3.23 所示。

```

1:排队 2:看医生 3:查看排队 0:下班 请选择:1✓
    >>输入患者姓名:Smith✓
1:排队 2:看医生 3:查看排队 0:下班 请选择:1✓
    >>输入患者姓名:John✓
1:排队 2:看医生 3:查看排队 0:下班 请选择:3✓
    >>排队患者: Smith John
1:排队 2:看医生 3:查看排队 0:下班 请选择:1✓
    >>输入患者姓名:Mary✓
1:排队 2:看医生 3:查看排队 0:下班 请选择:2✓
    >>患者Smith看医生
1:排队 2:看医生 3:查看排队 0:下班 请选择:2✓
    >>患者John看医生
1:排队 2:看医生 3:查看排队 0:下班 请选择:0✓
    >>请排队的患者明天就医

```

图 3.23 看病程序的一次执行结果

小结

(1) 栈和队列的共同点是,它们的数据元素都呈线性关系,且只允许在端点处插入和删除元素。

(2) 栈是一种“后进先出”或者“先进后出”的数据结构,只能在一端进行元素的插入和删除。

(3) 栈可以采用顺序栈和链栈两类存储结构。

(4) n 个不同元素的进栈顺序和出栈顺序不一定相同。

- (5) 在顺序栈中,通常用栈顶指针指向栈顶元素,栈顶指针类型为 int 类型。
- (6) 在顺序栈中,进栈和出栈操作不涉及栈中其他元素的移动。
- (7) 无论是顺序栈还是链栈,进栈和出栈运算的时间复杂度均为 $O(1)$ 。
- (8) 队列是一种“先进先出”或者“后进后出”的数据结构,只能从一端插入元素,另一端删除元素。
- (9) 队列可以采用顺序队和链队两类存储结构。
- (10) n 个元素进队的顺序和出队顺序总是一致的。
- (11) 顺序队中的元素个数可以由队头指针和队尾指针计算出来。
- (12) 循环队列也是一种顺序队,是通过逻辑方法使其首尾相连,解决非循环队列的假溢出现象。
- (13) 无论是顺序队还是链队,进队和出队运算的时间复杂度均为 $O(1)$ 。
- (14) 在算法设计中通常用栈或者队列保存临时数据,如果先保存的元素先处理,采用队列;如果后保存的元素先处理,采用栈。

练习题 3

1. 单项选择题

- (1) 栈的“先进后出”特性是指()。
 - A. 最后进栈的元素总是最先出栈
 - B. 当同时进行进栈和出栈操作时,总是进栈优先
 - C. 每当有出栈操作时,总要先进行一次进栈操作
 - D. 每次出栈的元素总是最先进栈的元素
- (2) 设一个栈的进栈序列是 a, b, c, d (即元素 $a \sim d$ 依次通过该栈),则借助该栈所得到的输出序列不可能是()。
 - A. $abcd$
 - B. $dcba$
 - C. $acdb$
 - D. $dabc$
- (3) 一个栈的进栈序列是 a, b, c, d, e ,则栈的不可能的输出序列是()。
 - A. $edcba$
 - B. $decba$
 - C. $dceab$
 - D. $abcde$
- (4) 已知一个栈的进栈序列是 $1, 2, 3, \dots, n$,其输出序列的第一个元素是 $i (1 \leq i \leq n)$,则第 $j (1 \leq j \leq n)$ 个出栈元素是()。
 - A. i
 - B. $n-i$
 - C. $j-i+1$
 - D. 不确定
- (5) 设顺序栈 st 的栈顶指针 top 的初始值为 -1 ,栈空间大小为 MaxSize,则判定 st 栈为栈空的条件为()。
 - A. $st.top == -1$
 - B. $st.top != -1$
 - C. $st.top == \text{MaxSize}$
 - D. $st.top == \text{MaxSize}$
- (6) 设顺序栈 st 的栈顶指针 top 的初始值为 -1 ,栈空间大小为 MaxSize,则判定 st 栈为栈满的条件是()。

- A. qu. rear==qu. front

B. $(qu.rear+1)\%MaxSize==qu.front+1$

C. $(qu.rear+1)\%MaxSize==qu.front$

D. $qu.rear==qu.front$

(17) 设循环队列中数组的下标是 $0\sim N-1$, 其头尾指针分别为 f 和 r (队头指针 f 指向队首元素的前一位置, 队尾指针 r 指向队尾元素的位置), 则其元素个数为()。

A. $r-f$

B. $r-f-1$

C. $(r-f)\%N+1$

D. $(r-f+N)\%N$

(18) 一个循环队列中用 $data[0..n-1]$ 数组保存队中元素, 另设置一个队尾指针 $rear$ 和一个记录队中实际元素个数的变量 $count$, 则该队中最多可以存放的元素个数是()。

A. $n-1$

B. n

C. $(rear+n)\%n$

D. $(n-rear)\%n$

(19) 设栈 S 和队列 Q 的初始状态为空, 元素 $e_1\sim e_6$ 依次通过栈 S , 一个元素出栈后即进队列 Q , 若 6 个元素出队的序列是 $e_2, e_4, e_3, e_6, e_5, e_1$, 则栈 S 的容量至少应该是()。

A. 5

B. 4

C. 3

D. 2

(20) 与顺序队相比, 链队的()。

A. 优点是可实现无限长队列

B. 优点是进队和出队时间性能更好

C. 缺点是不能进行顺序访问

D. 缺点是不能根据队首和队尾指针计算队的长度

2. 填空题

(1) 栈是一种特殊的线性表, 允许插入和删除运算的一端称为(①)。不允许插入和删除运算的一端称为(②)。

(2) 若栈空间大小为 n , 则最多的连续进栈操作的次数为()。

(3) 一个栈的输入序列是 12345, 输出序列为 12345, 其进栈出栈的操作为()。

(4) 有 5 个元素, 其进栈次序为 a, b, c, d, e , 在各种可能的出栈次序中, 以元素 c, d 最先出栈(即 c 第一个且 d 第二个出栈)的次序有()。

(5) 顺序栈用 $data[0..n-1]$ 存储数据, 栈顶指针为 top , 其初始值为 0, 则元素 x 进栈的操作是()。

(6) 顺序栈用 $data[0..n-1]$ 存储数据, 栈顶指针为 top , 其初始值为 0, 则出栈元素 x 的操作是()。

(7) () 是被限定为只能在表的一端进行插入运算, 在表的另一端进行删除运算的线性表。

(8) 设有数组 $A[0..m]$ 作为循环队列的存储空间, $front$ 为队头指针(它指向队首元素的前一位置), $rear$ 为队尾指针(它指向队尾元素的位置), 则元素 x 执行进队的操作是()。

(9) 设有数组 $A[0..m]$ 作为循环队列的存储空间, $front$ 为队头指针(它指向队首元素的前一位置), $rear$ 为队尾指针(它指向队尾元素的位置), 则元素出队并保存到 x 中的操作是()。

(10) 设循环队列的大小为 70, 队头指针 $front$ 指向队首元素的前一位置, 队尾指针 $rear$ 指向队尾元素位置。现经过一系列进队和出队操作后, 有 $front=20, rear=11$, 则队列中的元素个数是()。

3. 简答题

- (1) 简要说明线性表、栈与队的异同点。
- (2) 当用一维数组实现顺序栈时, 为什么一般将栈底设置在数组的一端, 而不是设置在中间?
- (3) 在以下几种存储结构中, 哪个最适合用作链栈?
 - ① 带头结点的单链表;
 - ② 不带头结点的循环单链表;
 - ③ 带头结点的双链表;
- (4) 在循环队列中插入和删除元素时, 是否需要移动队中元素?
- (5) 顺序队的“假溢出”是怎样产生的? 如何判断循环队列是空还是满?

4. 算法设计题

- (1) 设计一个算法, 利用一个顺序栈将字符数组 $a[0..n-1]$ 的所有元素逆置。
- (2) 设计一个算法, 将一个十进制正整数 d 转换为相应的八进制数。
- (3) 设计一个算法, 利用栈的基本运算返回给定栈中的栈底元素, 要求仍保持栈中元素次序不变。这里只能使用栈 st 的基本运算来完成, 不能直接用 $st.data[0]$ 来得到栈底元素。
- (4) 设计一个算法, 利用循环队列的基本运算和例 3.13 求循环队列元素个数的算法, 求指定循环队列中的队尾元素, 要求队列中元素次序不改变, 算法的空间复杂度为 $O(1)$ 。
- (5) 对于循环队列, 假设队中所有元素为数字字符, 利用循环队列的基本运算和例 3.13 求循环队列元素个数的算法, 删除其中所有奇数字符元素。

上机实验题 3

1. 基础实验题

- (1) 设计字符顺序栈的基本运算程序, 并用相关数据进行测试。
- (2) 设计字符链栈的基本运算程序, 并用相关数据进行测试。
- (3) 设计字符循环队列的基本运算程序, 并用相关数据进行测试。
- (4) 设计字符链队的基本运算程序, 并用相关数据进行测试。

2. 应用实验题

- (1) 设计一个算法, 利用顺序栈的基本运算删除栈 st 中所有值为 e 的元素(这样的元素可能有多个), 并且保持其他元素次序不变。并用相关数据进行测试。
- (2) 设计一个算法, 利用顺序栈的基本运算求栈中从栈顶到栈底的第 k 个元素, 要求仍保持栈中元素次序不变。并用相关数据进行测试。

(3) 设进栈序列是 $1, 2, \dots, n$ (n 为一个大于 2 的正整数), 编写一个程序判断通过一个栈能否得到由 $a[0..n-1]$ 指定出栈序列, 如果能够得到指定的出栈序列, 请给出栈操作的步骤。并用相关数据进行测试。

(4) 设计一个算法, 利用循环队列的基本运算和例 3.13 求循环队列元素个数的算法, 删除指定队列中的队尾元素, 要求算法的空间复杂度为 $O(1)$ 。

(5) 设计一个循环队列, 用 $front$ 和 $rear$ 分别作为队头和队尾指针, 另外用一个标志 tag 标识队列可能空 ($tag=0$) 或可能满 ($tag=1$), 这样加上 $front==rear$ 可以作为队空或队满的条件。要求设计队列的相关基本运算算法。

(6) 用循环队列求解约瑟夫问题: 设有 n 个人站成一圈, 其编号为 $1 \sim n$ 。从编号为 1 的人开始按顺时针方向 $1, 2, \dots$ 循环报数, 数到 m 的人出列, 然后从出列者的下一个人重新开始报数, 数到 m 的人又出列, 如此重复进行, 直到 n 个人都出列为止。要求输出这 n 个人的出列顺序。并用相关数据进行测试。

哈佛图书馆的二十条训言

1. 此刻打盹, 你将做梦; 而此刻学习, 你将圆梦。
2. 我荒废的今日, 正是昨日殒身之人祈求的明日。
3. 觉得为时已晚的时候, 恰恰是最早的时候。
4. 勿将今日之事拖到明日。
5. 学习时的苦痛是暂时的, 未学到的痛苦是终生的。
6. 学习这件事, 不是缺乏时间, 而是缺乏努力。
7. 幸福或许不排名次, 但成功必排名次。
8. 学习并不是人生的全部。但既然连人生的一部分——学习也无法征服, 还能做什么呢?
9. 请享受无法回避的痛苦。
10. 只有比别人更早、更勤奋地努力, 才能尝到成功的滋味。
11. 谁也不能随随便便成功, 它来自彻底的自我管理和毅力。
12. 时间在流逝。
13. 现在流的口水, 将成为明天的眼泪。
14. 狗一样地学, 绅士一样地玩。
15. 今天不走, 明天要跑。
16. 投资未来的人, 是忠于现实的人。
17. 受教育程度代表收入。
18. 一天过完, 不会再来。
19. 即使现在, 对手也在不停地翻动书页。
20. 没有艰辛, 便无所获。