# 实验一

## 1、实验目的

（1） 掌握线性表的顺序存储结构和链式存储结构；

（2） 掌握线性表和链表（循环链表）的基本操作，包括创建、查找、插入、删除等；

（3） 明确线性表的顺序存储结构和链式存储结构特点及其应用。

## 2、实验内容（题目）

### 约瑟夫环

Joseph 问题：已知 n 个人（以编号 1，2，3，_，代表）围坐成一圈，现在从编号为 k 的人开始报数，疏导 m 的人出列；他的下一个人又从 1 开始报数，数到 m 的人出列；依次重复下去，指到所有人全部出列。

例如：当 n=8 m=4 k=3，出列顺序为 6，2，7，4，3，5，1，8。

## 3、实验环境

（1） 软件系统

操作系统：macOS

编程环境：coderunner

（2） 硬件系统

CPU：2 GHz 四核 Intel Core i5

内存：16 GB 3733 MHz LPDDR4X 硬盘：Macintosh HD

## 4、程序代码（以源程序形式提供）见附件

## 5、实验分析

从以下方面分析：

1、正确性（输入的测试数据和测试结果，附测算截屏）

（1） 顺序表方法

```cpp
#include <iostream>
#include <cmath>
#include <stdlib.h>
using namespace std;

int total[1000];

int del(int total[1000],int location1,int totalnum)
{
    int i ;
    for(i = location1; i < totalnum; i++)//遍历顺序表 计数
    {
        total[i] = total[i+1];
    }
    return 0;
}

int Joseph(int total[1000], double location, double totalnum, double d
{
    int count = 0;
    int totalnum1 = totalnum;
    int location1 = location-1;
    for(count = 0; count < totalnum; count++)
    {
        if(location1 > totalnum1 - 1)
```

```
请输入总人数 Totalnum: 8
请输入开始的位置 Location: 4
请输入想要删除的序数 Deletenum: 3
6 ->1 ->4 ->8 ->5 ->3 ->7 ->2 ->
```

（2） 单链表方法

```cpp
//
//  约瑟夫环-单链表实现.cpp
//  Josephu
//
//  Created by 邵柏豪 on 2020/11/19.
//

#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;

typedef struct LNode{
    int num;
    struct LNode *next;
}LNode;
typedef LNode *Linklist;

void Create(Linklist &L, int totalnum)//构造链表
{
    Linklist p, q;
    p = L;
    int m = 1;
    for(int i = 0; i < totalnum; i++)
    {
```

```
请输入总人数 Totalnum: 8
请输入开始的位置 Location: 4
请输入想要删除的序数 Deletenum: 3
6 -> 1 -> 4 -> 8 -> 5 -> 3 -> 7 -> 2 ->
```

2、可读性（主要在源代码中体现）

代码注释：函数主体注释每个循环的意义；

写代码格式：空格、空行、缩紧、括号对齐；

3、健壮性（容错性，主要在源代码中体现，在此简要说明）

运行函数主体之前，检验 totalnum location deletenum 输入值的合法性，保证输入的变量符合题干中的客观含义，并符合函数内部对变量的定义；

如果输入变量超出范围，程序会要求用户再次输入直到输入变量是合法的；

4、时间和空间复杂度（针对核心算法函数分析）

（1）顺序表实现

假设给定的数列为 1,2,3,...,n-1,n 的形式；

从第 k 人开始报数到 m，假设每次报数到 m 的人的编号为 m+k-1，然后这个人出列。剩下的人的编号分别是 m+k、m+k+1、...n-1、n、1、2、...、m+k-2；设定数列 1,2,3,...,n-1,n 根据约瑟夫规则剩下的数字为与 m，n 有关的方程式：f(n,m)；设定 f'(n-1,m)为数列 m+k,m+k-1,...n-1,n,1,2,...,m+k-2，根据约瑟夫规则运行，最后剩下的数字。

可以看出，f(n,m),f'(m,n)表示的最后数字对应的序列式不一样的第一个是以升序排列，

而第二个是先升序，后降序。

可以根映射将在序列二中的数字转化为对应序列一种的形式，

m+k -> 1 m+k+1 -> 2

...

n-1 -> n-m-1

0 -> n-m

...

m+k-2 -> n-1

正向的转换方程式为 p(x) = (x + n - m) % n，x 属于序列二；逆向的转换方程

式为 p'(x) = (x + m) % n，x 属于序列一；

f'(n-1,m)表示的第二个序列的最后一个数和序列一经过 p'(x)转后，序列的最后一个数字相同： f'(n-1,m) = [f(n-1,m) + m] % n。

综上所述：

当 n=1 时，f(n,m) = 0；

当 n>1 时，f(n,m) = [f(n-1,m)+m] % n（递归函数）；时间复杂度为 O(n*n*m)、空间复杂度为 O(n)。

（2）单链表实现使用循环链表的方法，因为这种方法在删除一个节点后，对于其他节点的位置改动不用太大。

这是一种很浪费时间的方法，每次都删除第 m 个数字（注意题意包含摸的概念），也就是说，每次删除，都要用 O(m)的时间，一共有 n 个数字，想要剩下一个，其余都要删除，那么就得用(n-1)*O(m)的时间，时间复杂度为 O(m*n)，空间复杂度为 O(n)。

# 基于栈的中缀算术表达式求值

## 1.实验目的

1.掌握栈的基本操作算法的实现，包括栈的初始化、进栈、出栈、取栈顶元素等。

2.掌握利用栈实现中缀表达式求值的算法。

## 2.实验内容

输入一个中缀算术表达式，求解表达式得值。

运算符包括“+”、“-”、“*”、“/”、“(” “)” “=” 参加运算的数为 double 类型且为正数。

（要求：直接使用中缀算术表达式进行计算，不能转化为后缀或前缀表达式再进行计算，只考虑二元运算即可）

## 3.实验环境

（1） 软件系统

操作系统：macOS

编程环境：coderunner

（2） 硬件系统

CPU：2 GHz 四核 Intel Core i5 内存：16 GB 3733 MHz LPDDR4X 硬盘：Macintosh HD

## 4.程序代码

（见附件）

## 5.算法分析

1. 正确性

```
                    break;

                case '=': //OPTR的栈顶元素是“(”且ch是“)”，括号内容已经运算完
                    Pop(OPTR, x);
                    cin >> ch; //弹出OPTR栈顶的“(”，读入下一字符ch
                    break;
            }
        }
    }
    return GetTop(OPND); //OPND栈顶元素即为表达式求值结果
}

int main()
{
    while (1)
    {
        char ch;
        cin >> ch;
        if(ch=='=')break;
        double res = EvaluateExpression(ch);//表达式求值
        cout <<setiosflags(ios::fixed)<<setprecision(2)<< res <<endl;
    }
    return 0;
}
```

```
1+2*3-2/2=
6.00
```

2. 可读性

   代码注释：函数主体注释每个循环的意义；

   写代码格式：空格、空行、缩紧、括号对齐；

3. 健壮性（容错性）

   对各项输入违法判错，保证输入格式具有其相应的数学意义；

4. 时间复杂度

此程序的运算时间主要用在字符串扫描和算符优先权的比较上。把#看作运算符，操作数与运算符的个数相同。

最坏情况下优先级比较是 n/2 次，即运算顺序完全是逆序的,此时，每个字符扫描一遍是 O(n)的，时间复杂度是 O(n*n)。

# 基于字符串模式匹配算法的病毒感染检测问题

## 1.实验目的

1.掌握字符串的顺序存储表示方法。
2.掌握字符串模式匹配 BF 算法和 KMP 算法的实现。

## 2.实验内容

医学研究者最近发现了某些新病毒,通过对这些病毒的分析,得知它们的 DNA 序列都是环状的。现在研究者已收集了大量的病毒 DNA 和人的 DNA 数据,想快速检测出这些人是否感染了相应的病毒。为了方便研究,研究者将人的 DNA 和病毒 DNA 均表示成由一些字母组成的字符序列,然后检测某种病毒 DNA 序列是否在患者的 DNA 序列中出现过。如果出现过,则此人感染了该病毒,否则没有感染。

例如,假设病毒的 DNA 序列为 baa，患者 1 的 DNA 序列为 aaabbba，则感染;患者 2 的 DNA 序列为 babbba,则未感染。(注意,人的 DNA 序列是线性的,而病毒的 DNA 序列是环状的。)

输入要求

多组数据,每组数据有 1 行,为序列 A 和 B,A 对应病毒的 DNA 序列,B 对应人的 DNA 序列。A 和 B 都为“0”时输入结束。

输入样例

abbab abbabaab
baa cacdvcabacsd
abc def
0 0

输出样例

YES
YES
NO

## 3.实验环境

### （1） 软件系统

操作系统：macOS

编程环境：coderunner

### （2） 硬件系统

CPU：2 GHz 四核 Intel Core i5 内存：16

GB 3733 MHz LPDDR4X 硬盘：

Macintosh HD

## 4.程序代码

（见附件）

## 5.算法分析

### 1.正确性

程序可以得到正确的结果；

```cpp
#include<stdio.h>
#include<iostream>
#include<algorithm>
#include<cmath>
#include<ctime>
#include<string>
using namespace std;

bool Find(string Peo_DNA, string Vir_DNA, int i)
{
    int len_Peo = Peo_DNA.length();
    int len_Vir = Vir_DNA.length();
    int num = 0, j = 0;

    for (j = 0; j<2*len_Vir && i+j<len_Peo; j++)
    {
        if (Peo_DNA[i+j] == Vir_DNA[j])
        {
            num++;
        }
        else
        {
            num = 0;
        }
        if (num == len_Vir)
```

```
请输入病毒和人的DNA序列: abbab abbabaab
YES
请输入病毒和人的DNA序列: baa cacdvcabacsd
NO
请输入病毒和人的DNA序列:
```

### 2.可读性

代码注释：函数主体注释每个循环的意义；

写代码格式：空格、空行、缩紧、括号对齐；

### 3.健壮性（容错性）

对各项输入违法判错，保证输入格式具有其相应的数学意义；

### 4.时间复杂度

时间复杂度：0（m+n）空间复杂度：O（n）

KMP 算法是 BF 算法的改进，其最大的优势就是将 BF 算法 O(m*n)的时间复杂度降低为

O（m+n）的时间复杂度。

其中，降低时间复杂度最关键的步骤是避免病毒序列（模式串）中指针的回溯。

在 BF 算法中，一旦主串与模式串中正在比较的字符不相等，模式串中指针后退重新开始匹配，主串的下一个字符重新与模式串中的第一个字符开始匹配，时间复杂度较高。

而在 KMP 算法不需回溯模式串指针，而是利用已经得到的“部分匹配”结果将模式串向右“滑动”尽可能远的一段距离，再继续进行比较。

在 KMP 算法中，求 next 函数的时间复杂度为 O（m）。求得 next 函数后，匹配可以如下进行：

（1） 两指针指向的字符相等时，两指针都后移一位；

（2） 不相等时，主串指针不动，模式串指针退到 next[j]的位置再比较，若相等，则两指针各自增 1，否则 j 再退到下一个 next 位置。

依此类推，最后会有两种可能：一是 j 退到某个 next 值时字符比

较相等，两指针增 1，继续匹配；

另一种情况是匹配失败，模式串需要右移一个位置，从主串的下一个字符开始重新进行匹配，复杂度为 O（n）。

因此整体的时间复杂度为 O(m+n)。

# 基于哈夫曼树的数据压缩算法

## 1.实验目的

1.掌握哈夫曼树的构造算法
2.掌握哈夫曼编码的构造算法

## 2.实验内容

输入一串字符串，根据给定的字符串中字符出现的频率建立相应的哈夫曼树，构造哈夫曼编码表，在此基础上可以对压缩文件进行压缩（即编码），同时可以对压缩后的二进制编码文件进行解压（即解码）

实验要求：

1.输入说明:多组数据，每组数据 1 行，为一个字符串(只考虑 26 个小写英文字母即可)。档输入字符串为“0”时，输入结束。

2.输出说明:每组数据输出 2n+3 行（n 为输入串中字符类别的个数)。第 1 行为统计出来的字符出现频率（只输出存在的字符，格式为：字符：频度)，每两组字符之间用一个空格分隔，字符按照 ASCII 码从小到大的顺序排列。第 2 行到第 2n 行为哈夫曼树的存储结构的终态（一行当中的数据用空格分隔)。第 2n+1 行为每个字符的哈夫曼编码（只输出存在的字符，格式为：字符：编码)，每两组字符之间用一个空格分隔，字符按照 ASCII 码从小到大的顺序排列。第 2n+2 行为编码后的字符串，第 2n+3 行为解码后的字符串（与输入的字符串相同)

## 3.实验环境

（1） 软件系统

操作系统：macOS

编程环境：coderunner

（2） 硬件系统

CPU：2 GHz 四核 Intel Core i5 内存：16

GB 3733 MHz LPDDR4X 硬盘：

Macintosh HD

## 4.程序代码

（代码见附件）设计思路：

①建立一个结构体，其中包含了 weight:结点的权值，parent:结点的双亲，lchild:结点的左孩子以及 rchild:结点的右孩子；

②统计出来的字符出现频率（只输出存在的字符，格式为：字符：频度）。

③输出哈夫曼树的存储结构的终态，每一行有五个数字，其中第一个为结点序号，第二个为结点的值，第三个为结点的父节点的序号，第四个为结点左孩子的序号，第五个为结点右孩子的序号。

④设计一个算法，求出每个节点的哈夫曼编码。

⑤根据每个结点的编码，写出字符串的编码形式。

⑥设计一个算法，进行解码。

⑦构造一个哈夫曼树。

【核心算法设计】（1）构造哈夫曼树

```
void Select(HuffmanTree HT,int len,int &s1,int &s2)
{
//在构造哈夫曼树的时候，需要选出两个最小的结点循环遍历找出一个最小值，将其赋值给s1。
用 temp 将 s1 的值保存下来。
将 s1 的值改成最大值。
循环遍历找出一个最小值，将其赋值给 s2。
恢复 s1 的值。
}
```

（2） 初始化哈夫曼树

```
void CreatHuffmanTree(HuffmanTree &HT,int n,map<char,int>&maps)
{
初始化哈夫曼树
通过 n-1 次的选择、删除、合并来创建哈夫曼树
}
```

（3） 逆向解码

```
void CreatHuffmanCode(HuffmanTree HT,HuffmanCode &HC,int n)
{
从叶子到根逆向求每个字符的哈夫曼编码，存储在编码表 HC 中
结点 c 是 f 的左孩子，则生成代码 0 结点 c 是 f 的右孩子，则生
成代码 1
}
```

## 5.算法分析

（1）正确性

测试用例

aaaaaaabbbbbccdddd

aabccc

0 a:7 b:5 c:2 d:4

1 7 7 0 0

2 5 6 0 0

3 2 5 0 0
4 4 5 0 0
5 6 6 3 4
6 11 7 2 5
7 18 0 1 6
A:0 b:10 c:110 d:111
00000001010101010110110111111111111
aaaaaaabbbbbccdddd a:2 b:1 c:3
1 2 4 0 0
2 1 4 0 0
3 3 5 0 0
4 3 5 2 1
5 6 0 3 4
a:11 b:10 c:0
111110000
aabccc

```cpp
//基于哈夫曼树的数据压缩方法
#include<iostream>
#include<map>
#include<string>
```

```
aaaaaaabbbbbccdddd

aabccc

0a:7 b:5 c:2 d:4
1 7 7 0 0
2 5 6 0 0
3 2 5 0 0
4 4 5 0 0
5 6 6 3 4
6 11 7 2 5
7 18 0 1 6
a:0 b:10 c:110 d:111
00000001010101010110110111111111111
aaaaaaabbbbbccdddd
a:2 b:1 c:3
1 2 4 0 0
2 1 4 0 0
3 3 5 0 0
4 3 5 2 1
5 6 0 3 4
a:11 b:10 c:0
111110000
aabccc
```

（2）　可读性代码注释：函数主体注释每个循环的意义；

写代码格式：空格、空行、缩紧、括号对齐；

（3）　健壮性（容错性）

对各项输入违法判错，保证输入格式具有其相应的数学意义；

（4）　时间复杂度时间复杂度为 O(n^2)

因为在 CreatHuffmanTree（）函数中调用了 Select（）函数，里面每个都包含了一层 for 循环（n*n）,使得最终的时间复杂度为 O(n^2)。