



中南大學  
CENTRAL SOUTH UNIVERSITY

## 高级程序设计 实验报告

学生姓名	王璵
学 号	8208190406
专业班级	计科 1902 班
指导教师	黎方正
学 院	计算机学院
完成时间	2021.1.29

计算机学院

2020 年 12 月

## 目录

实验（一） 单链表、队列、二叉树查找和排序算法的实践.....	3
<1>系统描述： .....	3
<2>功能模块结构 .....	3
<3>主要模块的算法说明： .....	6
<4>运行结果.....	8
<5>课程设计总结： .....	10
<6>参考文献.....	11
<7>附录.....	11
实验（二） 栈与二叉树遍历的实践 .....	31
<1>系统描述： .....	31
<2>功能模块结构： .....	31
<3>主要模块的算法说明： .....	33
<4>运行结果.....	34
<5>课程设计总结： .....	35
<6>参考文献.....	36
<7>附录.....	36
实验（三） 哈夫曼树编译码的实践.....	50
<1>系统描述： .....	50
<2>功能模块结构： .....	51
<3>主要模块的算法说明： .....	52
<4>运行结果.....	53
<5>课程设计总结： .....	53
<6>参考文献.....	54
<7>附录.....	54
实验（四） 图的最小生成树和最短路径的实践.....	58
<1>系统描述： .....	58
<2>功能模块结构： .....	58
<3>主要模块的算法说明： .....	62
<4>运行结果.....	63
<5>课程设计总结： .....	65
<6>参考文献.....	65
<7>附录.....	65
实验（五） 四则运算表达式的求值系统设计（四选一选做部分） .....	73
<1>系统描述： .....	73
<2>功能模块结构： .....	73
<3>主要模块的算法说明： .....	73
<4>运行结果.....	74
<5>课程设计总结： .....	78
<6>参考文献.....	78
<7>附录.....	79

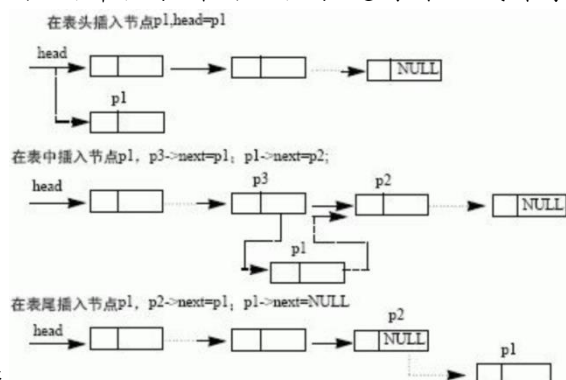
# 实验（一） 单链表、队列、二叉树查找和排序算法的实践

## <1> 系统描述：

- (1) 编写函数，实现输入一组元素，建立一个带头结点的单链表；对该链表进行非递减排序；实现在非递减有序链表中删除值为  $x$  的结点；
- (2) 编写函数，采用链式存储和顺序存储实现队列的初始化、入队、出队操作；
- (3) 编写函数，建立有序表，利用二叉排序树的插入算法建立二叉排序树；在以上二叉排序树中删除某一指定关键字元素；采用折半查找实现某一已知的关键字的查找(采用顺序表存储结构)
- (4) 选用 1-3 的数据结构，编写程序实现下述五种算法：简单插入排序，冒泡排序，快速排序，归并排序，堆排序。

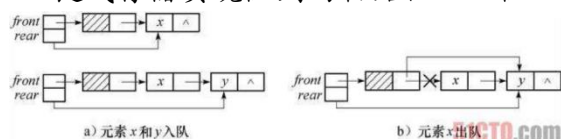
## <2> 功能模块结构：

- 1、建立一个带头结点的单链表并对该链表进行非递减排序和实现在非递减有序链表中

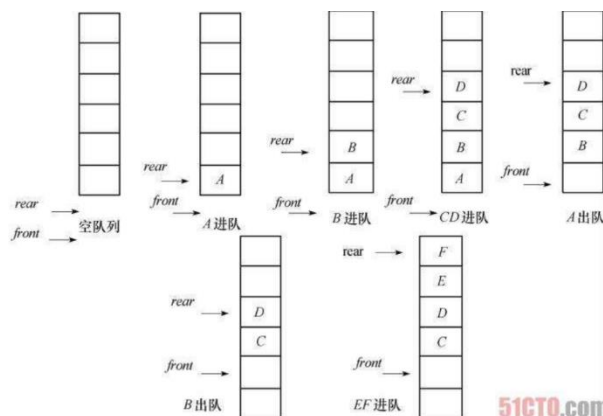


删除值为  $x$  的结点

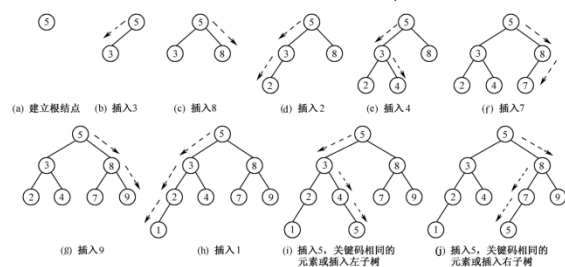
- 2、链式存储实现队列的初始化、入队、出队操作



顺序存储实现队列的初始化、入队、出队操作

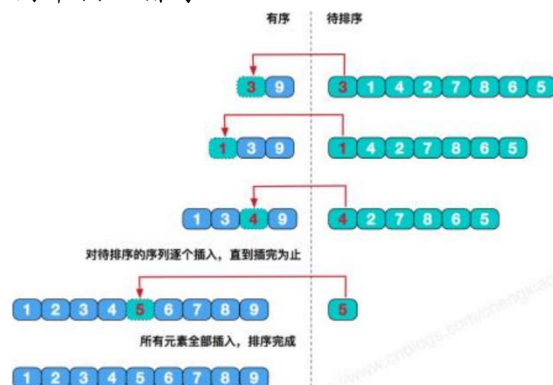


3、建立二叉排序树并在以上二叉排序树中删除某一指定关键字元素；采用折半查找实现某一已知关键字的查找(采用顺序表存储结构)

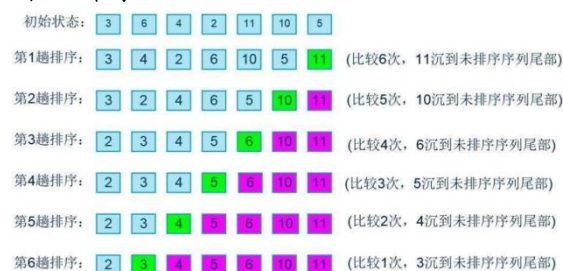


4、几种排序算法的实现

简单插入排序



冒泡排序



快速排序

## 快速排序

初始

{49 38 65 97 76 13 27 49}

一次划分之后

{27 38 13} 49 {76 97 65 49}

序列左继续排序

{13} 27 {38} 49 {76 97 65 49}

(结束)

(结束)

序列右继续排序

{49 65} 76 {97}

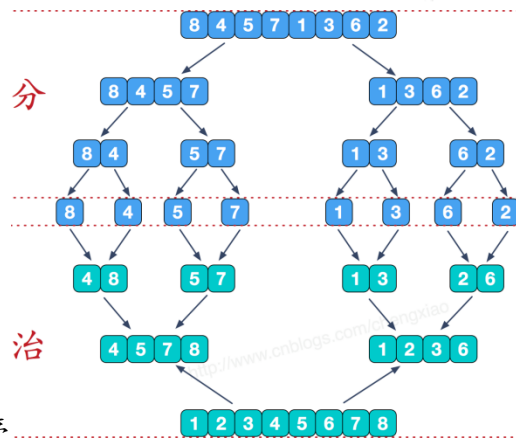
(结束)

49 {65}

(结束)

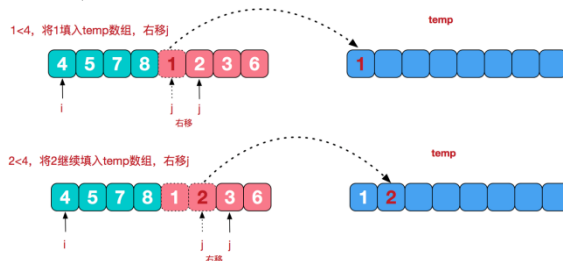
有序序列

{13 27 38 49 49 65 76 97}



归并排序

我们需要将两个已经有序的子序列合并成一个有序序列。如下图所示，以此类推，直到被填满。



堆排序

假设存在以下数组

0 1 2 3 4  
3 6 8 5 7

。每次新插入的数据都与其父结点进行比较，如果插入的数比父结点大，则与父结点交换，否则一直向上交换，直到小于等于父结点，或者来到了顶端，如下图所示。插入6的时候，6大于他的父结点3，则交换；此时，保证了0~1位

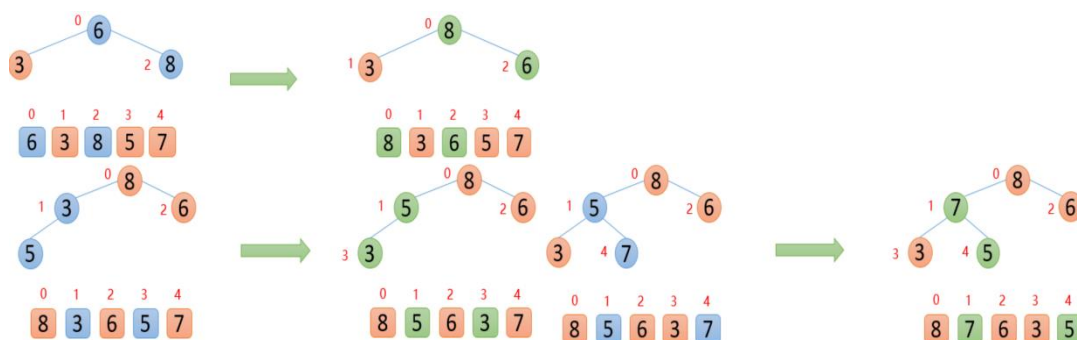


置是大根堆结构。其余同理。



0 1 2 3 4  
3 6 8 5 7

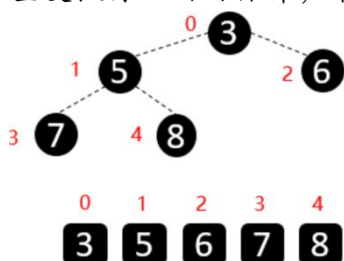
0 1 2 3 4  
6 3 8 5 7



此时，我们已经得到一个大根堆，下面将顶端的数与最后一位数交换，然后将剩余的数再构造一个大根堆。



重复执行上面的操作，最终会得到有序数组。



### <3> 主要模块的算法说明：

(1) 单向链表是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的；其中每个结点都有指针成员变量指向列表中的下一个结点。

在选择头插法和尾插法的时候要注意在合并链表的时候，这里的前提条件是这两个链表要顺序递增，所以我是使用的是尾插法那么如果我选择头插法的时候就要注意了，它是倒着插进去的，比如你输入1，2，3，4，5，而链表的结果则是5，4，3，2，1，所以如果在这个条件下还要执行合并算法的话，前提条件便是顺序递减。

(2) 队列无疑是数据结构中十分重要的一部分，它采用先进先出的模式，可以分为链式队列和顺序队列，在本次程序中，除了实现了初始化，入队和出队的基本要求，我自己也额外使用了删除结点、判断队列长度、判断队列是否为空等拓展功能

这个关于队列的基本功能比较简单，使用链式操作时，用 front 与 rear 的两个指针分别指向队列的头和尾，在进行相应的操作时，指针进行前后移动的操作，使用顺序存

储时，则使用两个整形变量来进行标注队列的队头和队尾，在进行相应的操作时，实现+1的操作

入队是将数据放在队尾指针或者数组，同时队尾+1，出队使用的是 printf 对头的的数据，同时队头-1。删除则是用该结点的前一个指针指向该结点的后一个结点实现对当前结点的删除；判断队列长度则是从队头到队尾的一次遍历指导指向空指针。

(3) 编写函数，建立有序表，利用二叉排序树的插入算法建立二叉排序树；在以上二叉排序树中删除某一指定关键字元素；采用折半查找实现某一已知关键字的查找(采用顺序表存储结构)

要找的关键字要么在左子树上，要么在右子树上，要么在根结点上。二叉排序树的定义可以知道，根结点中的关键字将所有关键字分成了两部分，即大于根结点中关键字的部分和小于根结点中关键字的部分。可以将待查关键字先和根结点中的关键字比较，如果相等则查找成功；如果小于则到左子树中去查找，无须考虑右子树中的关键字；如果大于则到右子树中去查找，无须考虑左子树中的关键字。如果来到当前树的子树根，则重复上述过程；如果来到了结点的空指针域，则说明查找失败。

(4) 选择类排序的主要动作是选择”，简单选择排序采用最简单的选择方式，从头至尾顺序扫描序列，找出最小的一个关键字，和第一个关键字交换，接着从剩下的关键字中继续这种选择和交换，最终使序列有序。

冒泡排序是通过一系列的“交换”动作完成的。首先第一个关键字和第二个关键字比较，如果第一个大，则二者交换，否则不交换；然后第二个关键字和第三个关键字比较，如果第二一直按这种方式进行下去，最终最大的那个关键字被交换到了最后··一趟起泡排序完成。经过多趟这样的排序，最终使整个序列有序。

快速排序也是“交换”类的排序，它通过多次划分操作实现排序。以升序为例，其执行流程可以概括为：每一趟选择当前左右子序列中的一个关键字作为枢轴，将子序列中比枢轴小的移到前面，大的一道后面；当本趟所有子序列都被枢轴以上述规则划分完毕后会得到新的一组更短的子序列，它们成为下一趟划分的初始序列集。

归并排序可以看作一个分而治之的过程：先将整个序列分为两半，对每一半分别进行归并排序，将得到两个有序序列，然后将这两个序列归并成一个序列即可。假设待排序列存在数组 A 中，用 low 和 high 两个整型变量代表需要进行归并排序的范围。

根据堆的定义知道，代表堆一的这棵完全二叉树的根结点的值是最大（或最小）的，因此将一个无序序列调整为一个堆，就可以找出这个序列的最大（或最小）值，然后将找出的这个值交换到序列的最后（或最前），这样，有序序列关键字增加 1 个，无序序列中关键字减少 1 个，对新的无序序列重复这样的操作，就实现了排序。

在这个实验中，我没有采用一个个键入的方式，而是采用随机产生数字的方法进行排序，分别进行 100 个，200 个，300 个，1000 个，2000 个数字的比较，我觉得这

样可以更好的体现程序的随机性与完整性，也可以直观的看到各个排序算法的优良性，也是自己的一个亮点

## <4>运行结果

1、建立一个带头结点的单链表并对该链表进行非递减排序和实现在非递减有序链表中

删除值为 x 的结点

```
选择1:头插还是2:尾插
1
结点个数:5
输入结点的数字:1
输入结点的数字:2
输入结点的数字:3
输入结点的数字:4
输入结点的数字:5
选择: 1. 删除结点 2. 查询操作 3. 打印链表 4. 排序链表 5. 退出
2
查询数字:3
数字在第3个结点
```

```
选择1:头插还是2:尾插
1
结点个数:5
输入结点的数字:1
输入结点的数字:2
输入结点的数字:3
输入结点的数字:4
输入结点的数字:5
选择: 1. 删除结点 2. 查询操作 3. 打印链表 4. 排序链表 5. 退出
3
54321
```

2、链式存储实现队列的初始化、入队、出队操作

```
Microsoft Visual Studio 调试控制台
Create empty queue successfully!
Is it empty?1(1:Yes 0:NO)
The length of queue is 0
After insert three element(-5,5,10), the length of queue is 3
Is it empty?0(1:YES 0:NO)
The element of queue are
-5 5 10
the head element is -5
After clear queue, q.front=00C1D588, q.rear=00C1D588, q.front->next=00000000
After destory queue, q.front=00000000, q.rear=00000000
```

顺序存储实现队列的初始化、入队、出队操作



```

c# Microsoft Visual Studio 调试控制台
队列为空!
请输入要入队的个数
4
输入你要入队的4个数据
45
67
23
14
请输入要出队的个数
1
输出队列中的1个元素:
45
队列中元素如下:
67 23 14

```

3、建立二叉排序树并在以上二叉排序树中删除某一指定关键字元素；采用折半查找实现某一已知的关键字的查找(采用顺序表存储结构)

```

c# Microsoft Visual Studio 调试控制台
9 8 7 6 5 4 3 21 14 99 777 666 55 44 33 22
3 4 5 6 7 8 9 14 21 33 44 55 99 666 777
3
查找成功!
3 4 5 6 7 8 9 21 33 44 55 99 666 777

```

4、几种排序算法的实现

简单插入排序

```

c# Microsoft Visual Studio 调试控制台
1
2
3
4
5
6
7
8
9
10

```

冒泡排序

```

c# Microsoft Visual Studio 调试控制台
2500
900
543
532
76
56
43
35
34
32
3
2
-58
-70
-234

```

快速排序

```

c# Microsoft Visual Studio 调试控制台
请输入需排序的数组的个数: 5
请输入数组元素: 12
56
78
33
55
排序结果: 12 33 55 56 78

```

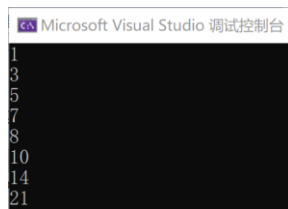
归并排序

```

c# Microsoft Visual Studio 调试控制台
2 5 8 8 66 33 2 12 0 56 20
0 2 2 5 8 8 12 20 33 56 66

```

堆排序



## <5>课程设计总结:

首先，之前学习数据结构的时候并没有重视数据结构在之后编程学习过程中的重要性，因此上个学期的学习更多注重的是对数据结构的理解和纸上演算而很少关注数据结构的具体代码实现。这个学期我们拥有了更丰富的代码经验和能力因此，这个学期实现数据结构的代码再合适不过了。

链表是数据结构中十分重要的一个编程实现部分，在排序，随机存储等方面都发挥着举足轻重的作用，无论是单向链表还是双向链表抑或是循环链表，都是我们应当掌握的重中之重，编程过程中我遇到了比较大的问题，在前面的难点分析中我已经说过。通过这次编程学习，我有所感悟：知识点的学习与巩固，通过课本只是一个手段之一，也绝对不是最有效的手段，最有效的手段对于编程学习来说永远都是实践，只有你遇到了BUG然后花了很长时间去查资料DEBUG才会对这个知识点掌握牢固。

在顺序队中，通常让队尾指针 rear 指向刚进队元素位置，让队首指针 front 指向刚出队的元素位置。因此，元素进队的时候，rear 要向后移动；元素出队的时候，front 也要向后移动。这样经过一系列的出队和进队操作以后，两个指针最终会达到数组末端 `maxSize-1` 处。虽然队中已经没有元素，但仍然无法让元素进队，这就是所谓的“假溢出”。要解决这个问题，可以把数组弄成一个环，让 rear 和 front 沿着环走，这样就永远不会出现两者来到数组尽头无法继续往下走的情况，这样就产生了循环队列。循环队列是改进的顺序队列。

本次实验主要进行了链表、队列、二叉树等数据结构的复习运用，以及多种常用排序算法的实现。他们都是十分重点典型的数据结构，在各个方面都有着广泛的应用，因此，应该熟练地掌握。

在本次编程中我深深体会到对程序提前进行设计规划整理出大概思路以及细心分析编写的重要性。因为本次实验的函数均较复杂和冗长，因此，我们需在编程前对程序有个较为清晰的规划，在正式编写的时候也要细心，否则，这种较大的程序不易检查和修改。

链表、队列、二叉树的一系列操作虽然在数据结构中已经学习过，但因其较为复杂，有一定难度，因此长时间未复习过后有一定的遗忘，查阅了一些资料，这是需要改进复习的。但几个排序算法已经写过多次，对于其基本思想已基本掌握。

下次实验我们将进入另一个十分重要的数据结构——栈，并对二叉树更深一步的学习。在实验前先好好复习，以求在正式实验时更加的高效、准确率高。

## <6> 参考文献

《数据结构》 清华大学出版社

## <7> 附录

1、建立一个带头结点的单链表并对该链表进行非递减排序和实现在非递减有序链表中删除值为 x 的结点

```
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
    int data;
    struct node* next;
}Linklist;

Linklist* create();//创建带头结点的链表
Linklist* head_insert(Linklist* head, int number);//头插法插入结点
Linklist* nail_insert(Linklist* head, int number);//尾插法插入结点
Linklist* research(Linklist* head, int number);//查询操作
Linklist* delete1(Linklist* head, int number);//删除操作
Linklist* sort(Linklist* head, int n);//排序链表
Linklist* display(Linklist* head);//打印链表

int main()
{
    Linklist* head;
    int n, t, number, i, n1, a;
    head = create();
    done1:printf("选择1:头插还是2:尾插\n");
```

```

scanf_s("%d", &n);
switch (n)
{
case 1:
    printf("结点个数:");
    scanf_s("%d", &t);
    for (i = 0; i < t; i++)
    {
        printf("\n输入结点的数字:");
        scanf_s("%d", &number);
        head = head_insert(head, number);
    }

    break;
case 2:
    printf("结点个数:");
    scanf_s("%d", &t);
    for (i = 0; i < t; i++)
    {
        printf("\n输入结点的数字:");
        scanf_s("%d", &number);
        head = nail_insert(head, number);
    }
    break;
default:
    printf("你输入的数字不符合，请重新输入\n");
    goto done1;
}
done: printf("\n选择： 1.删除结点 2.查询操作 3.打印链表 4.排序链表 5.退出\n");
scanf_s("%d", &n1);
switch (n1)
{
case 1:
    printf("请输入您要删除的数字:");
    scanf_s("%d", &number);
    head = delete1(head, number);
    printf("完成");
    goto done;

```

```

    case 2:
        printf("查询数字:");
        scanf_s("%d", &a);
        research(head, a);
    goto done;
    case 3:
        head = display(head);
        goto done;
    case 4:
        sort(head, n);
        goto done;
    case 5: return 0;
    default:
        printf("\n你输入的数字不符合，请重新输入");
        goto done;
}
return 0;
}

```

```

Linklist* create()//创建带头结点的链表
{
    Linklist* head;
    head = (Linklist*)malloc(sizeof(Linklist));
    head->next = NULL;
    return head;
}

```

```

Linklist* head_insert(Linklist* head, int number)//头插法插入结点
{
    Linklist* p, * t;
    t = head;
    p = (Linklist*)malloc(sizeof(Linklist));
    p->data = number;
    p->next = t->next;
    t->next = p;
    return head;
}

```

```

Linklist* tail_insert(Linklist* head, int number)//尾插法插入结点

```

```

{
    Linklist* p, * t;
    t = head;
    p = (Linklist*)malloc(sizeof(Linklist));
    p->data = number;
    while (t->next)
    {
        t = t->next;
    }
    t->next = p;
    p->next = NULL;
    return head;
}

```

Linklist\* research(Linklist\* head, int number)// 查询操作

```

{
    Linklist* t;
    int n = 1;
    t = head->next;// 调试过程中这里出现了错误
    while ((t->data != number) && (t != NULL))
    {
        t = t->next;
        n++;
    }
    if (t == NULL)
    {
        printf("\n没有该结点\n");
        return NULL;
    }

    else
    {
        printf("\n数字在第%d个结点\n", n);
    }
    return NULL;
}

```

Linklist\* delete1(Linklist\* head, int number)// 删除操作

```

{

```

```

Linklist* t, *p;
p = head;
t = p->next; // 问题出现在这
while (t != NULL && t->data != number)
{
    p = t;
    t = t->next;
}
if (t == NULL)
{
    printf("\n没有该结点\n");
    return head;
}
else
{
    p->next = t->next;
    free(t);
}
return head;
}

```

```

Linklist* display(Linklist* head) // 打印链表
{
    Linklist* t;
    t = head->next;
    if (t == NULL)
    {
        printf("the list is empty");
        return NULL;
    }
    while (t)
    {
        printf("%d", t->data);
        t = t->next;
    }
    printf("\n");
    return NULL;
}

```

```

Linklist* sort(Linklist* head, int n)//排序链表
{
    Linklist* min, * comp, * before;
    for (int i = 0; i < n; i++)
    {
        min = head->next;
        before = head;
        comp = min->next;
        while (comp != NULL) {
            if (comp->data < min->data) {
                min = comp;
            }
            comp = comp->next;
        }
        while (before->next != min) {
            before = before->next;
        }
        before->next = min->next;
        min->next = head->next;
        head->next = min;
    }
    return NULL;
}

```

2、链式存储实现队列的初始化、入队、出队操作

```

#include <iostream>
#include <cstdlib>
#include <string>
#include <iomanip>
using namespace std;
#define OK 1
#define OVERFLOW -1
#define TRUE 1
#define FALSE 0
#define ERROR 0
typedef int QElemType;
typedef int Status;
typedef struct QNode
{
    QElemType data;

```



```

        QNode* next;
    }*QueuePtr;

struct LinkQueue
{
    QueuePtr front, rear;
};

Status InitQueue(LinkQueue& Q)
{
    if (!(Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode))))
        exit(OVERFLOW);
    Q.front->next = NULL;
    return OK;
}

Status DestoryQueue(LinkQueue& Q)
{
    while (Q.front)
    {
        Q.rear = Q.front->next;
        free(Q.front);
        Q.front = NULL;
        Q.front = Q.rear;
    }
    return OK;
}

Status ClearQueue(LinkQueue& Q)
{
    QueuePtr q, p;
    Q.rear = Q.front;
    p = Q.front->next;
    Q.front->next = NULL;
    while (p)
    {
        q = p->next;
        free(p);
        p = q;
    }
}

```

```

    }
    return OK;
}

```

```

Status QueueEmpty(LinkQueue Q)
{
    if (Q.front->next == NULL)
        return TRUE;
    else
        return FALSE;
}

```

```

Status QueueLength(LinkQueue Q)
{
    int i = 0;
    QueuePtr p;
    p = Q.front->next;
    while (p)
    {
        ++i;
        p = p->next;
    }
    return i;
}

```

```

Status GetHead(LinkQueue Q, QElemType& e)
{
    QueuePtr p;
    if (Q.front == Q.rear)
        return ERROR;
    p = Q.front->next;
    e = p->data;
    return OK;
}

```

```

Status EnQueue(LinkQueue& Q, QElemType e)
{
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));
}

```

```

        if (!p) return OVERFLOW;
        p->data = e;
        Q.rear->next = p;
        Q.rear = p;
        p->next = NULL;
        return OK;
    }

```

```

Status DeQueue(LinkQueue& Q, QElemType& e)
{
    QueuePtr p;
    if (Q.front == Q.rear)
        return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next = p->next;
    free(p);
    return OK;
}

```

```

Status QueueTraverse(LinkQueue Q, void(*visit)(QElemType))
{
    QueuePtr q;
    q = Q.front->next;
    while (q)
    {
        visit(q->data);
        q = q->next;
    }
    cout << endl;
    return OK;
}

```

```

void visit(QElemType e)
{
    cout << setw(5) << e;
}

```

```

int main()

```

```

{
    int i;
    QElemType e;
    LinkQueue q;
    i = InitQueue(q);
    if (i)
        cout << "Create empty queue successfully!" << endl;
    cout << "Is it empty?" << QueueEmpty(q) << "(1:Yes 0:NO)" << endl;
    cout << "The length of queue is " << QueueLength(q) << endl;
    EnQueue(q, -5);
    EnQueue(q, 5);
    EnQueue(q, 10);
    cout << "After insert three element(-5,5,10),the length of queue is " <<
    QueueLength(q) << endl;
    cout << "Is it empty?" << QueueEmpty(q) << "(1:YES 0:NO)" << endl;
    cout << "The element of queue are " << endl;
    QueueTraverse(q, visit);
    i = GetHead(q, e);
    if (i == OK)
        cout << "the head element is " << e << endl;
    ClearQueue(q);
    cout << "After clear queue,q.front=" << q.front << ",q.rear=" << q.rear << ",
    q.front->next=" << q.front->next << endl;
    DestoryQueue(q);
    cout << "After destory queue,q.front=" << q.front << ",q.rear=" << q.rear << endl;
    return 0;
}

```

顺序存储实现队列的初始化、入队、出队操作

```
#include<iostream>
```

```
#define maxsize 100
```

```
using namespace std;
```

```
typedef struct queue
```

```
{
```

```
    int data[maxsize];
```

```
    int head;
```

```
    int tail;
```

```
}Queue;
```

```

void init(Queue* l)
{
    l->head = -1;
    l->tail = -1;
}

void empty(Queue l)
{
    if (l.head == -1 && l.tail == -1)
        cout << "队列为空!\n";
    else cout << "队列不为空!\n";
}

void inqueue(Queue* l, int i)
{
    if (l->tail + i >= maxsize - 1)
        cout << "要入队的元素过多，请重新输入! ";
    else
    {
        cout << "输入你要入队的" << i << "个数据\n";
        for (int a = 0; a < i; a++)
            cin >> l->data[++l->tail];
    }
}

void outqueue(Queue* l, int j)
{
    if (l->head + j > l->tail)
        cout << "队列中没有足够多元素来输出\n";
    else
    {
        cout << "输出队列中的" << j << "个元素: \n";
        for (int a = 0; a < j; a++)
            cout << l->data[++l->head] << ' ';
        cout << '\n';
    }
}

void print(Queue* l)
{

```

```

        cout << "队列中元素如下: \n";
        for (; l->head < l->tail;
                cout << l->data[++l->head] << ' ');
        cout << '\n';
    }

```

```

int main()
{
    int i, j;
    Queue k;
    init(&k);
    empty(k);
    cout << "请输入要入队的个数\n";
    cin >> i;
    inqueue(&k, i);
    cout << "请输入要出队的个数\n";
    cin >> j;
    outqueue(&k, j);
    print(&k);
    return 0;
}

```

3、建立二叉排序树并在以上二叉排序树中删除某一指定关键字元素；采用折半查找实现某一已知关键字的查找(采用顺序表存储结构)

```

#include <iostream>
using namespace std;
typedef int KeyType;
typedef int InfoType;
typedef struct
{
    KeyType key;
    InfoType otherinfo;
}ElemType;
typedef struct BSTNode
{
    ElemType data;
    struct BSTNode* lchild, * rchild;
}BSTNode, * BST;
void InsertBST(BST& T, ElemType e)
{

```

```

    BSTNode* S;
    if (!T)
    {
        S = new BSTNode;
        S->data = e;
        S->lchild = S->rchild = NULL;
        T = S;
    }
    else if (e.key < T->data.key) InsertBST(T->lchild, e);
    else if (e.key > T->data.key) InsertBST(T->rchild, e);
}

void CreateBST(BST& T)
{
    ElemType e;
    T = NULL;
    cin >> e.key;
    for (int i = 1; i <= 15; i++)
    {
        InsertBST(T, e);
        cin >> e.key;
    }
}

void SearchBST(BST& T, KeyType key)
{
    if (T && key == T->data.key) cout << "查找成功! " << endl;
    else if (T && key < T->data.key) return SearchBST(T->lchild, key);
    else if (T && key > T->data.key) return SearchBST(T->rchild, key);
    else cout << "查找失败" << endl;
}

void DeleteBST(BST& T, KeyType key)
{
    BST P, F, Q, S;
    P = T; F = NULL;
    while (P)
    {
        if (P->data.key == key) break;
        F = P;
        if (P->data.key > key) P = P->lchild;
        else P = P->rchild;
    }
}

```

```

    }
    if (!P) return;
    Q = P;
    if ((P->lchild) && (P->rchild))
    {
        S = P->lchild;
        while (S->rchild)
        {
            Q = S;
            S = S->rchild;
        }
        P->data = S->data;
        if (Q != P) Q->rchild = S->lchild;
        else Q->lchild = S->lchild;
        delete S;
        return;
    }
    else if (!P->rchild)
    {
        P = P->lchild;
    }
    else if (!P->lchild)
    {
        P = P->rchild;
    }
    if (!F) T = P;
    else if (Q == F->lchild) F->lchild = P;
    else F->rchild = P;
    delete Q;
}

void Show(BST T)
{
    if (T)
    {
        Show(T->lchild);
        cout << T->data.key << " ";
        Show(T->rchild);
    }
}

```



```

int main()
{
    BST T;
    int key1, key2;
    CreateBST(T);
    Show(T);
    cout << endl;
    cin >> key1;
    SearchBST(T, key1);
    DeleteBST(T, 14);
    Show(T);
}

```

#### 4、几种排序算法的实现

##### 简单插入排序

```

#include <iostream>
using namespace std;

```

```

void insertsort(int* p);

```

```

int main()
{
    int a[10] = { 6,2,4,7,5,8,9,10,3,1 };
    insertsort(a);
    for (int i = 0; i < 10; i++)
    {
        cout << a[i] << endl;
    }
    return 0;
}

```

```

void insertsort(int* p)
{
    int temp;
    for (int i = 1; i < 10; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (*(p + i) < *(p + j))

```

```

        {
            temp = *(p + i);
            *(p + i) = *(p + j);
            *(p + j) = temp;
        }
    }
}

```

冒泡排序

```

#include <iostream>
using namespace std;
int main()
{
    int a[] = { 900, 2, 3, -58, 34, 76, 32, 43, 56, -70, 35, -234, 532, 543, 2500 };
    int n; // 存放数组a中元素的个数
    int i; // 比较的轮数
    int j; // 每轮比较的次数
    int buf; // 交换数据时用于存放中间数据
    n = sizeof(a) / sizeof(a[0]); /*a[0]是int型, 占4字节, 所以总的字节数除以4等于元素的个数*/
    for (i = 0; i < n - 1; ++i) // 比较n-1轮
    {
        for (j = 0; j < n - 1 - i; ++j) // 每轮比较n-1-i次,
        {
            if (a[j] < a[j + 1])
            {
                buf = a[j];
                a[j] = a[j + 1];
                a[j + 1] = buf;
            }
        }
    }
    for (i = 0; i < n; ++i)
    {
        cout << a[i] << endl;
    }
    cout << endl;
    return 0;
}

```

## 快速排序

```
#include <iostream>
```

```
using namespace std;
```

```
void quick_sort(int* a, int left, int right) //left和right为索引值
```

```
{
```

```
    int temp; //存储每次选定的基准数（从最左侧选基准数）
```

```
    int t;
```

```
    int initial = left;
```

```
    int end = right;
```

```
    temp = a[left];
```

```
    if (left > right) //因为在递归过程中有减1加1的情况，当其越界时，直接return，  
    不返回任何值，即结束当前程序块
```

```
        return;
```

```
    while (left != right) //此时左右index在移动中，若left==right,则跳出循环，将基  
    数归位
```

```
    {
```

```
        while (a[right] >= temp && left < right) //直到找到小于基准数的值为准  
            right--;
```

```
        while (a[left] <= temp && left < right)  
            left++;
```

```
        if (left < right) //交换左右两侧值，当left=right时，跳出外层while循环  
        {
```

```
            t = a[right];
```

```
            a[right] = a[left];
```

```
            a[left] = t;
```

```
        }
```

```
    }
```

```
    a[initial] = a[left];
```

```
    a[left] = temp; //基数归位
```

```
    quick_sort(a, initial, left - 1); //此时left=right
```

```
    quick_sort(a, left + 1, end);
```

```
}
```

```
int main()
```

```
{
```

```
    int a[20], n;
```

```
    cout << "请输入需排序的数组的个数: ";
```

```
    cin >> n;
```

```

    cout << "请输入数组元素: ";
    for (int i = 0; i < n; i++)
        cin >> a[i];
    quick_sort(a, 0, n - 1);
    cout << "排序结果: ";
    for (int j = 0; j < n; j++)
        cout << a[j] << " ";
    cout << endl;
    return 0;
}
归并排序
#include<iostream>
using namespace std;

void Merge(int* a, int p, int q, int r)
{
    int n1 = q - p + 1;    //左部分的元素个数
    int n2 = r - q;        //同上
    int i, j, k;
    int* L = new int[n1 + 1];
    int* R = new int[n2 + 1];
    for (i = 0; i < n1; i++)
        L[i] = a[p + i];
    for (j = 0; j < n2; j++)
        R[j] = a[q + j + 1];
    L[n1] = 11111111;
    R[n2] = 11111111;
    for (i = 0, j = 0, k = p; k <= r; k++)
    {
        if (L[i] <= R[j])
            a[k] = L[i++];
        else
            a[k] = R[j++];
    }
    delete[] L;
    delete[] R;
}

void MergeSort(int* a, int l, int r)
{

```

```

        if (l < r)
        {
            int m = (l + r) / 2;
            MergeSort(a, l, m);
            MergeSort(a, m + 1, r);
            Merge(a, l, m, r);
        }
    }

int main()
{
    int i;
    int a[11] = { 2,5,8,8,66,33,2,12,0,56,20 };
    for (i = 0; i < 11; i++)
        cout << a[i] << " ";
    cout << endl;
    MergeSort(a, 0, 10);
    for (i = 0; i < 11; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}

```

堆排序

```

#include<iostream>
#include<vector>
using namespace std;
void adjust(vector<int>& arr, int len, int index)
{
    int left = 2 * index + 1; // index的左子节点
    int right = 2 * index + 2; // index的右子节点

    int maxIdx = index;
    if (left < len && arr[left] > arr[maxIdx])    maxIdx = left;
    if (right < len && arr[right] > arr[maxIdx])    maxIdx = right;

    if (maxIdx != index)
    {
        swap(arr[maxIdx], arr[index]);
    }
}

```

```

        adjust(arr, len, maxIdx);
    }

}

void heapSort(vector<int>& arr, int size)
{
    // 构建大根堆（从最后一个非叶子节点向上）
    for (int i = size / 2 - 1; i >= 0; i--)
    {
        adjust(arr, size, i);
    }

    // 调整大根堆
    for (int i = size - 1; i >= 1; i--)
    {
        swap(arr[0], arr[i]);    // 将当前最大的放置到数组末尾
        adjust(arr, i, 0);       // 将未完成排序的部分继续进行堆排序
    }
}

int main()
{
    vector<int> arr = { 8, 1, 14, 3, 21, 5, 7, 10 };
    heapSort(arr, arr.size());
    for (int i = 0; i < arr.size(); i++)
    {
        cout << arr[i] << endl;
    }
    return 0;
}

```

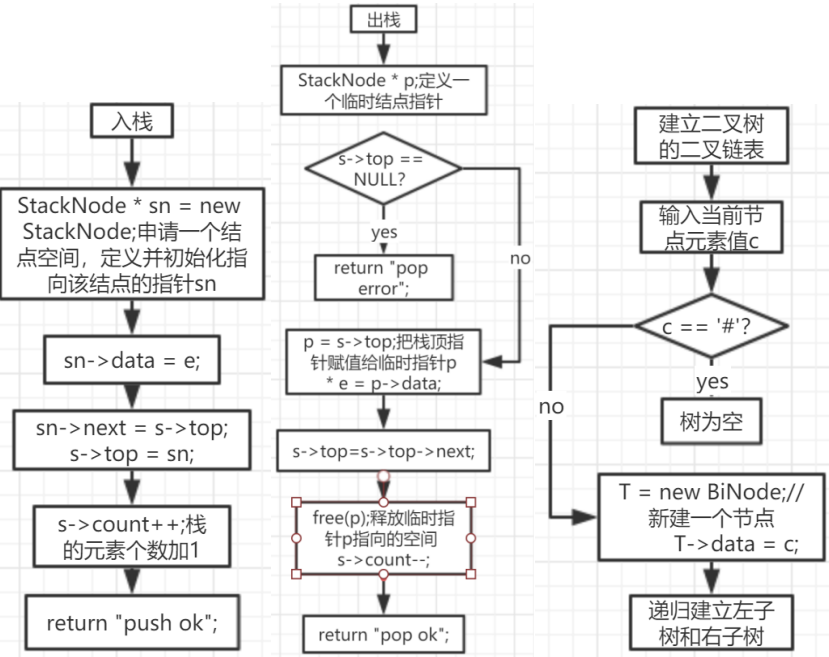
# 实验（二） 栈与二叉树遍历的实践

## <1> 系统描述：

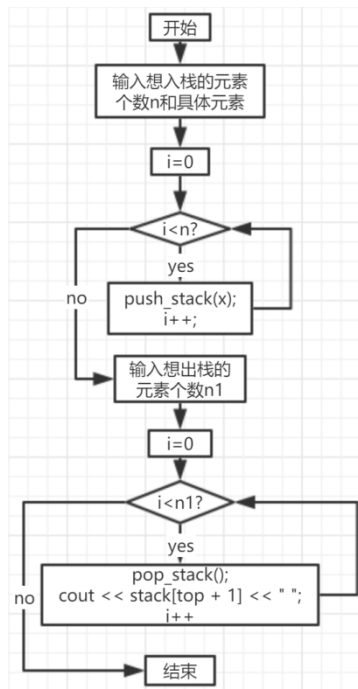
- (1) 编写函数，分别采用链式存储和顺序存储实现栈的初始化、入栈、出栈操作； 3. 编写函数，采用链式存储实现队列的初始化、入队、出队操作
- (2) 编写函数，建立二叉树的二叉链表；实现二叉树的前中后序的递归和非递归遍历算法。

## <2> 功能模块结构：

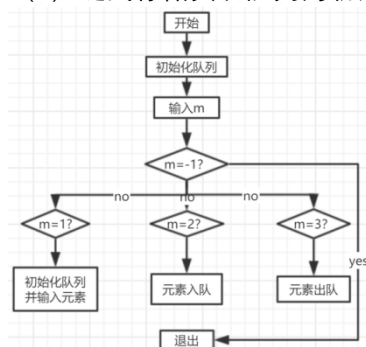
1、(1) 链式存储



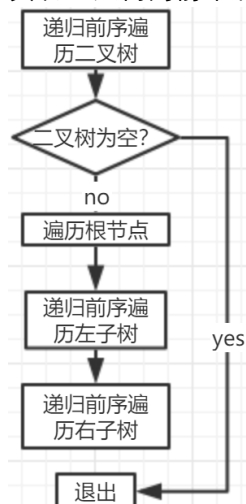
顺序存储



(2) 链式存储实现队列的初始化、入队、出队操作

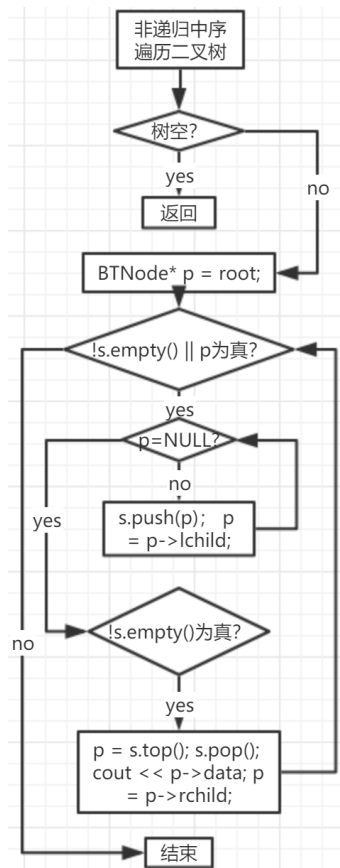


实现二叉树的前中后序的递归遍历

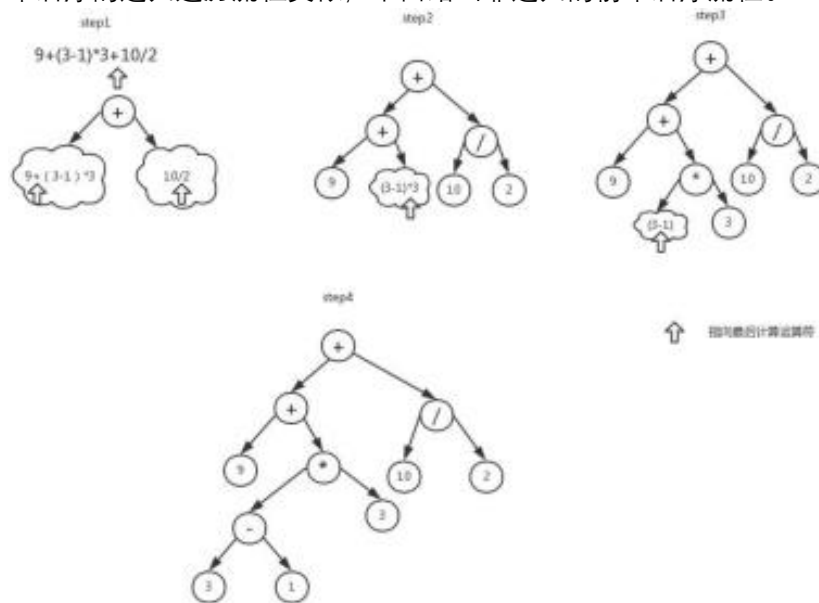


(3) 建立二叉树的二叉链表





中后序的递归遍历流程类似，下面给出非递归的前中后序流程。



### <3>主要模块的算法说明：

先序遍历和中序遍历类似，中序非递归遍历过程如下：

循环执行如下操作：如果栈顶结点左孩子存在，则左孩子入栈；如果栈顶结点左孩子不存在，则出栈并输出栈顶结点，然后检查其右孩子是否存在，如果存在，则右孩子

入栈。当栈空时算法结束。

逆后序遍历序列和先序遍历序列有一定的联系，逆后序遍历序列只不过是先序遍历过程中对左右子树遍历顺序交换所得到的结果。非递归先序遍历算法中对左右子树的遍历顺序交换就可以得到逆后序遍历序列，然后将逆后序遍历序列逆序就得到了后序遍历序列。因此我们需要两个栈，一个栈用来辅助做逆后序遍历（将先序遍历的左、右子树遍历顺序交换的遍历方式称为逆后序遍历）并将遍历序列压入另一个栈 stack2,然后将 stack2 中的元素全部出栈，所得到的序列即为后序遍历序列。

## <4>运行结果

### 1、(1) 链式存储

```
Microsoft Visual Studio 调试控制台
A push ok 1
B push ok 2
pop ok B 1
pop ok A 0
pop error
```

顺序存储

```
Microsoft Visual Studio 调试控制台
请输入你想入站的元素个数: 4
入站的元素依次为:
1 2 3 4
请输入你想出站的元素个数: 2
出站的元素依次为:
4 3
```

### (2) 链式存储实现队列的初始化、入队、出队操作

```
Microsoft Visual Studio 调试控制台
所有操作如下：
(1) 采用链式存储实现队列的初始化操作
(2) 采用链式存储实现队列的入队操作
(3) 采用链式存储实现队列的出队操作
(-1) 退出
请选择：1
请输入元素，以-2 结束
1 2 3 4 -2
1 2 3 4
所有操作如下：
(1) 采用链式存储实现队列的初始化操作
(2) 采用链式存储实现队列的入队操作
(3) 采用链式存储实现队列的出队操作
(-1) 退出
请选择：2
请输入要入队的元素：
5
所有操作如下：
(1) 采用链式存储实现队列的初始化操作
(2) 采用链式存储实现队列的入队操作
(3) 采用链式存储实现队列的出队操作
(-1) 退出
请选择：3
1
所有操作如下：
(1) 采用链式存储实现队列的初始化操作
(2) 采用链式存储实现队列的入队操作
(3) 采用链式存储实现队列的出队操作
(-1) 退出
请选择：-1
```

### (1) 建立二叉树的二叉链表并实现二叉树的前中后序的递归和非递归遍历

```
1: 递归法建立二叉树
2: 循环法建立二叉树
3: 二叉树的先序遍历
4: 二叉树的中序遍历
5: 二叉树的后序遍历
请输入你要执行操作对应的序号：
1
请输入字符串：
ABC..D..E.FG...
3
先序遍历的结果：
ABCDEFG
4
中序遍历的结果：
CBDAEGF
5
后序遍历的结果：
CDBGFEA
```

## 2、四则运算表达式的求值系统设计

```
C:\Users\PRO\Documents\未命名1.exe
1 四则运算表达式求值
0 退出
请输入欲执行功能对应的数字： 1
请输入四则运算表达式，以=结尾：
1+1*5-3+7=
计算结果为：
1+1*5-3+7=10.000000
请按任意键继续. . .
```

## <5>课程设计总结：

(1)采用链式存储实现栈的初始化、入栈、出栈操作时元素入栈和出栈时栈顶指针的改写有些许难度，需要一些严谨的考虑。入栈时，必须先将top 指针原来的值赋给新节点的 next 之后才能改变 top 的值，否则若 top 的值先被覆盖则不能形成链式结构。关

键代码为 `sn->next = s->top; s->top = sn;` 顺序很重要。出栈时，考虑必须全面，如果是空栈，直接返回出栈失败；否则关键代码为 `s->top=s->top->next;`。

(2) 怎样在同一个程序中实现队列的多个操作？一开始我很茫然，可查阅大量资料后发现其实很简单，是自己想复杂了。只需写一个 `switch` 来选择，用输入的数来代替操作并做一些提醒用户的说明即可。因为有些操作是要以另一些操作为前提的，而且并非运行一次就必须要求实现所有操作，用 `switch` 就可以完全按照自己的意愿，并使用循环，在想要的时候退出即可。

(3) 对二叉树的非递归遍历不是很熟，有些遗忘，不知如何实现。查阅资料，借助了栈这一数据结构成功实现了运行。

(4) 本次实验主要进行栈与二叉树遍历的实践。我通过上机实践，对栈与二叉树遍历的基本概念及其不同的实现方法的理论得到进一步掌握，并对在不同存储结构上实现不同的运算方式和技巧有了体会。补足了自己在对栈以及二叉树这两种重要数据结构的认知，并对其上的常用操作有了进一步的认识。

(5) 下次实验我们将会对哈夫曼树的基本概念及其不同的实现方法的理论进行掌握，并对在不同存储结构上实现不同的运算方式和技巧进行体会。哈夫曼树是一种特殊的树，是本次实验更进一步的延申，吸取本次实验的经验，下次实验希望更高效地完成。

## <6> 参考文献

《数据结构》清华大学出版社

## <7> 附录

链式存储

编写入栈出栈函数并编数据来调用以测试是否正确输出。

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
typedef string status;
```

```
//栈的结点，包含 data，和指向下一个结点的指针 next
```

```
typedef struct StackNode
```

```
{
```

```
    char data;
```

```
    struct StackNode* next;
```

```
}StackNode;
```

//栈的链式存储结构，包含指向栈顶的指针 top，和栈中元素个数 count

//top 指向最新入栈的结点，出栈的时候也是 top 指向的元素先出

```
typedef struct LinkStack
```

```
{
```

```
    StackNode * top;
```

```
    int count;
```

```
}LinkStack;
```

//申请内存，初始化，top 指向 NULL，表示初始化为空栈

```
LinkStack * InitLinkStack(LinkStack * s)
```

```
{
```

```
    s = new LinkStack;
```

```
    s->top = NULL;
```

```
    s->count = 0;
```

```
    return s;
```

```
}
```

//入栈

```
status Push(LinkStack * s, char e)
```

```
{
```

StackNode \* sn = new StackNode; // 申请一个结点空间，定义并初始化指向该结点的指针 sn

```
sn->data = e; // 把入栈的元素赋值给新结点的 data
```

```
sn->next = s->top; // 把新结点的 next 指向上一个结点
```

```
s->top = sn; // 把 top 指针指向新结点，栈顶元素是新入栈的元素
```

```
s->count++; // 栈的元素个数加 1
```

```
return "push ok"; // 返回入栈成功
```

```
}
```

//出栈

```
status Pop(LinkStack * s, char* e)
```

```
{
```

```
StackNode * p; // 定义一个临时结点指针
```

```
if (s->top == NULL) return "pop error"; // 如果是空栈，直接返回出栈失败
```

```
p = s->top; // 把栈顶指针赋值给临时指针 p
```

```
*e = p->data; // 把栈顶元素赋值给 e 指向的变量，即主调函数中需要被修改的
```

变量

```

        if (p->next == NULL)//如果出栈的是最后一个元素
            s->top = NULL;//该元素出栈后，栈为空
        else
            s->top = s->top->next;//否则 top 指向出栈结点的下一个结点
        //这个 if else 也可以直接写成 s->top=s->top->next;因为如果出栈的是最后一个元素，top->next 本来就等于 NULL
        free(p);//释放临时指针 p 指向的空间
        s->count--;//栈的元素减 1
        return "pop ok";//返回出栈成功
    }

```

```

int main()
{
    LinkStack * p = NULL;
    p = InitLinkStack(p);
    //入栈
    cout << "A" << " " << Push(p, 'A') << " ";//入栈 A
    cout << p->count << endl;//入栈 A 后有 1 个元素
    cout << "B" << " " << Push(p, 'B') << " ";//入栈 B
    cout << p->count << endl;//入栈 B 后有 2 个元素
    //出栈
    char e;
    char* pe = &e;
    cout << Pop(p, pe) << " ";//出栈 B
    cout << e << " " << p->count << endl;//出栈 B 后有 1 个元素
    cout << Pop(p, pe) << " ";
    cout << e << " " << p->count << endl;//出栈 A 后有 0 个元素
    cout << Pop(p, pe) << endl;//栈空，出栈失败
}

```

顺序存储

相当于用数组实现数据的存储，且编写函数来调用，方便简洁。

```

#include <iostream>
using namespace std;
#define stack_size 10
int stack[stack_size];
int top = 0;
void Init_Stack() //初始化顺序栈
{
    top = -1;
}

```

```

}

void push_stack(int x)
{
    if (top == stack_size)
        cout << "栈满! " << endl;
    else
    {
        top++;
        stack[top] = x;
    }
}

void pop_stack()
{
    if (top == -1)
        cout << "栈下溢! " << endl;
    else
    {
        top--;
    }
}

int main()
{
    Init_Stack(); //初始化顺序栈
    cout << "请输入你想入站的元素个数: "; //顺序栈的建立
    int n;
    cin >> n;
    cout << "入站的元素依次为: " << endl;
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        push_stack(x);
    }
    cout << "请输入你想出站的元素个数: "; //顺序栈的出栈操作
    int n1;
    cin >> n1;
    cout << "出站的元素依次为: " << endl;
}

```

```

        for (int i = 0; i < n1; i++)
        {
            pop_stack();
            cout << stack[top + 1] << " ";

        }
        cout << endl;
        return 0;
    }
    (2)

```

链式存储实现队列的初始化、入队、出队操作 #include<iostream>

```

#define OK 1
#define OVERFLOW 2
#define ERROR -1
using namespace std;
typedef int QElemType;
typedef int Status;
typedef struct QNode {
    QElemType data;
    struct QNode* next;
}QNode, * QueuePtr;
typedef struct {
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;
Status InitQueue(LinkQueue& Q) {
    Q.front = (QueuePtr)malloc(sizeof(QNode));
    Q.rear = (QueuePtr)malloc(sizeof(QNode));
    if (!Q.front)exit(OVERFLOW);
    Q.front = Q.rear;
    return OK;
}
Status InitStack(LinkQueue& Q) {
    QueuePtr p;
    int e;
    cout << "请输入元素，以-2 结束" << endl;
    while (cin>>e && e != -2) {
        p = (QueuePtr)malloc(sizeof(QNode));
        p->data = e;
        p->next = NULL;
    }
}

```



```

        Q.rear->next = p;
        Q.rear = p;
    }
    p = Q.front->next;
    while (p != NULL)
    {
        cout << p->data << " ";
        p = p->next;
    }
    cout << endl;
    return OK;
}

void Test()
{
    cout << "所有操作如下: " << endl;
    cout << " (1) 采用链式存储实现队列的初始化操作" << endl;
    cout << " (2) 采用链式存储实现队列的入队操作" << endl;
    cout << " (3) 采用链式存储实现队列的出队操作" << endl;
    cout << " (-1) 退出" << endl;
    cout << "请选择: ";
}

Status DeQueue(LinkQueue& Q) {
    QueuePtr p;
    if (Q.front == Q.rear)
    {
        cout<<"队列为空!!!!!!!"<<endl;
        return ERROR;
    }
    p = Q.front->next;
    cout << p->data << endl;
    Q.front->next = p->next;
    if (Q.rear == p) Q.rear = Q.front;
    free(p);
    return OK;
}

Status EnQueue(LinkQueue& Q) {
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));
    if (!p)exit(OVERFLOW);

```

```

        cout << "请输入要入队的元素: " << endl;
        cin >> p->data;
        Q.rear->next = p;
        p->next = NULL;
        Q.rear = p;
        return OK;
    }
int main()
{
    LinkQueue q;
    InitQueue(q);
    int m;
    do {
        Test();
        cin >> m;
        switch (m) {
            case 1:
                InitStack(q);
                break;
            case 2:
                EnQueue(q);
                break;
            case 3:
                DeQueue(q);
                break;
        }
    } while (m != -1);
    return 0;
}

```

(3) 关键函数的代码如下

建立二叉树的二叉链表

//默认按前序遍历的顺序输入，尾结点用#表示

```

int Create_BiTree(BiPtr& T)
{
    ElemType c;
    //cout << "请输入当前节点元素值: " << endl;
    cin >> c;
    if (c == '#') T = NULL;

```

```

else
{
    T = new BiNode;//新建一个节点
    T->data = c;
    Create_BiTree(T->lchild);
    Create_BiTree(T->rchild);
}
return 0;
}
实现二叉树的前中后序的递归遍历算法

```

```

//递归前序遍历二叉树
int PreOrderTraverse_Recursive(BiPtr T)
{
    if (T)
    {
        cout << T->data << " ";
        PreOrderTraverse_Recursive(T->lchild);
        PreOrderTraverse_Recursive(T->rchild);
    }
    return 0;
}

```

```

//递归中序遍历二叉树
int InOrderTraverse_Recursive(BiPtr T)
{
    if (T)
    {
        InOrderTraverse_Recursive(T->lchild);
        cout << T->data << " ";
        InOrderTraverse_Recursive(T->rchild);
    }
    return 0;
}

```

```

//递归后序遍历二叉树
int PostOrderTraverse_Recursive(BiPtr T)
{

```

```

if (T)
{
    PostOrderTraverse_Recursive(T->lchild);
    PostOrderTraverse_Recursive(T->rchild);
    cout << T->data << " ";
}
return 0;
}
实现二叉树的前中后序的非递归遍历算法
//非递归前序遍历二叉树
int PreOrderTraverse_NonRecursive(BiPtr T)
{
    BiPtr p = T;
    SqStack S;
    InitStack(S);

    while (p || !EmptyStack(S))
    {
        while (p)
        {
            cout << p->data << " ";
            Push(S, p);
            p = p->lchild;
        }
        if (!EmptyStack(S))
        {
            Pop(S, p); // 刚刚访问过的根节点
            p = p->rchild; // 访问该根节点的右子树，并对右子树重复上述过程
        }
    }
    return 0;
}
//非递归中序遍历二叉树
int InOrderTraverse_NonRecursive_1(BiPtr T)
{
    BiPtr p = T;
    SqStack S;
    InitStack(S);

```

```

Push(S, T);
while (!EmptyStack(S))
{
    while (GetTop(S))
    {
        Push(S, p->lchild);
        p = p->lchild; // 不停地“向左下走”
    }
    Pop(S, p); // 前面会“走过头”，也就是说最后栈顶会多出一个空指针，所以弹出多
    入栈的空指针

```

```

    if (!EmptyStack(S))
    {
        Pop(S, p);
        cout << p->data << " ";
        Push(S, p->rchild);
        p = p->rchild;
    }
}
return 0;
}

```

//非递归后序遍历二叉树

/\*

后序遍历是先访问左子树再访问右子树，最后访问根节点

由于返回根节点时有可能从左子树返回的，也可能是从右子树返回的，

要区分这两种情况需借助辅助指针 r，其指向最近访问过的节点

\*/

```

int PostOrderTraverse_NonRecursive(BiPtr T)

```

```

{
    BiPtr p = T, r = nullptr; // 从树根开始, r 用于指向最近访问过的节点
    SqStack S;
    InitStack(S);

    while (p || !EmptyStack(S))
    {
        if (p)
        {
            Push(S, p);

```

```

        p = p->lchild;//走到最左下方
    }
    else//说明已经走到最左下了，要返回到刚刚经过的根节点，即从 null 的左孩子返回到它的双亲节点
    {
        p = GetTop(S);
        if (p->rchild && p->rchild != r)//右孩子尚未访问，且右孩子不为空，于是，访问右孩子
        {
            p = p->rchild;
            Push(S, p);
            p = p->lchild;//走到右孩子的最左下方，即重复最开始的过程
        }
        else//右孩子为空，则直接输出该节点，或者右孩子刚刚被访问过了，同样直接输出该节点
        {
            Pop(S, p);
            cout << p->data << " ";
            r = p;//r 指向刚刚访问过的节点
            p = nullptr;//左右子树都已经处理完，根节点也刚处理完，返回到根节点的双亲，即栈顶元素，下次重新取栈顶元素分析
        }
    }
}
return 0;
}

```

#### 1、四则运算表达式的求值系统设计

```

#include<iostream>
#include<stdlib.h>
#include<string.h>
using namespace std;
typedef struct BTreeNode
{
    char data;
    struct BTreeNode* lchild;
    struct BTreeNode* rchild;
} BTreeNode;

```

```

int calcula(BTNode* T)
{
    if(T==NULL)
        return 0;
    if(T->data <='9'&&T->data >='0')
        return (T->data-'0');
    else
    {
        switch(T->data)          //因为这一步的 T->data 必为运算符，则必有左右
孩子节点且不空
        {
            case'+': return calcula(T->lchild) + calcula(T->rchild); break;
            case'-': return calcula(T->lchild) - calcula(T->rchild); break;
            case'*': return calcula(T->lchild) * calcula(T->rchild); break;
            case'/': return calcula(T->lchild) / calcula(T->rchild); break;
        }
    }
}

char* input_cheak_str()          // 字符串动态输入与检测函数
{
    printf("请输入一个简单算术表达式(一位正整数且只有+*/无括号,输入换行符结
束):\n");
    int ch_num=0;
    char ch,*str;
    str=(char*)malloc(sizeof(char));
    while((ch=getchar())!='\n') //设置按照输入字符数变化的字符数组(内存足够则不
受数组长度影响)
    {
        if(ch_num%2==1)          //下标为奇数，字符应为运算符号
        {
            if(ch!='+' && ch!='-' && ch!='*' && ch!='/')
            {
                printf(" 第 %d 个字符输入不合法!应为 “+*/” 之一
",ch_num+1);
                return '\0';
            }
        }
    }
}

```

```

else //下标为偶数, 字符应为数字
{
    if(!(ch>='0' && ch<='9'))
    {
        printf("第%d 个字符输入不合法!应为 0 至 9 数字之一",ch_num+1);
        return '\0';
    }
    str[ch_num]=ch;
    str=(char*)realloc(str,(++ch_num+1)*sizeof(char)); // ∴ ch_num 为字符数组下标,而 realloc 参数为字符个数
} // ∴ 新开数组长度参数为下标+2,相当于参数为 num++ 后的 num+1
if(str[ch_num-1]=='+' || str[ch_num-1]=='-' || str[ch_num-1]=='*' || str[ch_num-1]=='/')
{
    //若最后一个字符为运算符则输入不合法
    printf("最后一个字符输入不合法!应为数字!",ch_num+1);
    return '\0';
}
str[ch_num]='\0'; //串结尾设置串结束符
return str;
}

```

```

BTNode* creat_tree(char *str)
{
    int itemCount=0,ASCount=0,len=strlen(str),i; //AS 意为 addSub 加减法,前者为加减项计数,后者为加减符号计数,用于数组下标
    BTNode **ASItem,**ASSign,*root,*p; //ASItem 指针数组存放加减项节点指针,ASSign 指针数组存放加减符号节点指针
    ASItem=(BTNode**)malloc((len/2+1)*sizeof(BTNode*));
    ASSign=(BTNode**)malloc((len/2)*sizeof(BTNode*));
    if(str[0]=='\0') //加减符号节点数必为加减项节点数+1.既
itemCount==ASCount+1
        return NULL;
    for(i=0;i<len/2;i++) //指针数组置空
        ASSign[i]=NULL;
    for(i=0;i<len/2+1;i++)

```



```

        ASItem[i]=NULL;
for(i=0;i<len;i++)          //读取 str 字符数组
{
    if(str[i]<='9' && str[i]>='0')
    {
        p=(BTNode*)malloc(sizeof(BTNode));
        p->data=str[i];
        p->lchild=p->rchild=NULL;
    }
    else if(str[i]=='+' || str[i]=='-')
    {
        ASItem[itemCount++]=p;    //将 p 节点放入加减项数组
        p=(BTNode*)malloc(sizeof(BTNode));
        p->data=str[i];
        ASSign[ASCount++]=p;
    } //将加减符号节点指针放入 ASSign 数组,因有符号节点的孩子必不为空
    且创建过程不会访问其孩子节点,故无需置空
    else          //str[i]符号为乘除的情况
    {
        root=(BTNode*)malloc(sizeof(BTNode));
        root->data=str[i];    //将*,/作为数据存入根节点数据域
        root->lchild=p;    //p 一定为数字或*,/节点(都是已构造好的)
        p=(BTNode*)malloc(sizeof(BTNode));
        p->data=str[++i];    //此时 p 为当前节点的下一个节点,此时
        str[++i]必为数字,且下一个访问的 str 必为符号
        p->lchild=p->rchild=NULL;
        root->rchild=p;    //根节点的右孩子连上此节点
        p=root;    //整个根节点构造完毕,传入 p
    }
}
ASItem[itemCount]=p;
ASSign[0]->lchild=ASItem[0];    //第一个符号节点左孩子连第一个项节点
ASSign[0]->rchild=ASItem[1];
for(i=1;i<ASCount;i++)    //以加减法符号节点作为子树根节点,加减法之间的
    项的节点为子树根节点的孩子节点
{
    //加减符号节点数必为加减项节点数+1.既
    itemCount==ASCount+1,这里构造时 ASCount 已自增一次
    ASSign[i]->lchild=ASSign[i-1]; //除第一个节点以外的加减符号节点左孩子

```

都连上一个符号节点

```
        ASSign[i]->rchild=ASItem[i+1]; // 右孩子都连项节点
    }
    return ASSign[ASCount-1];
}

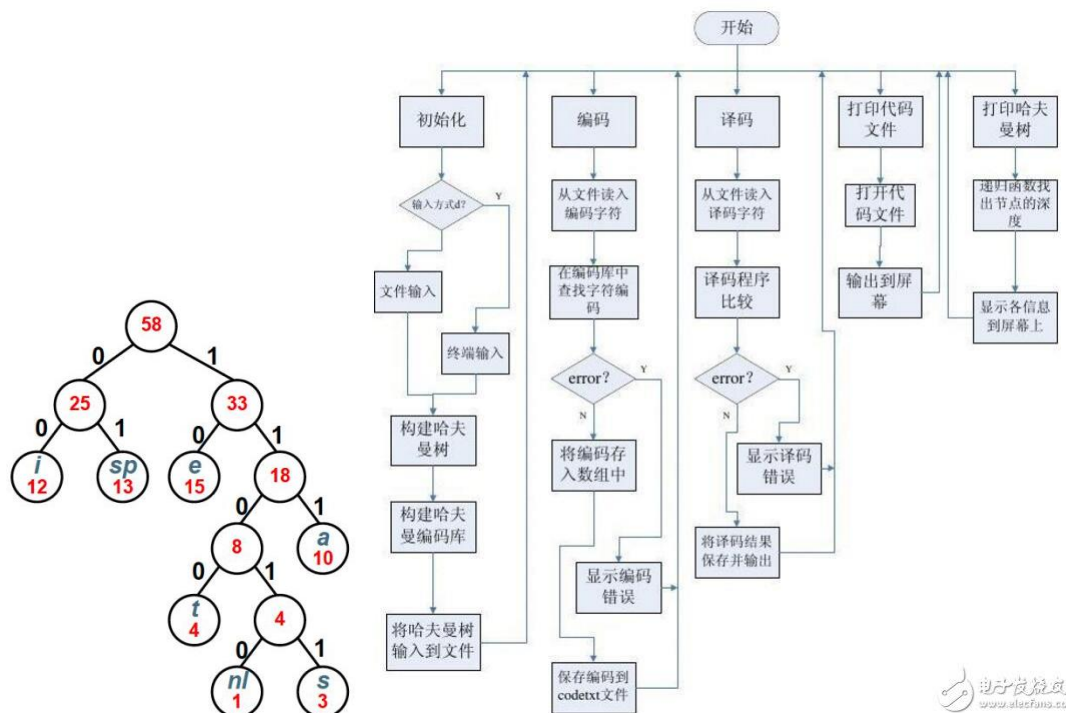
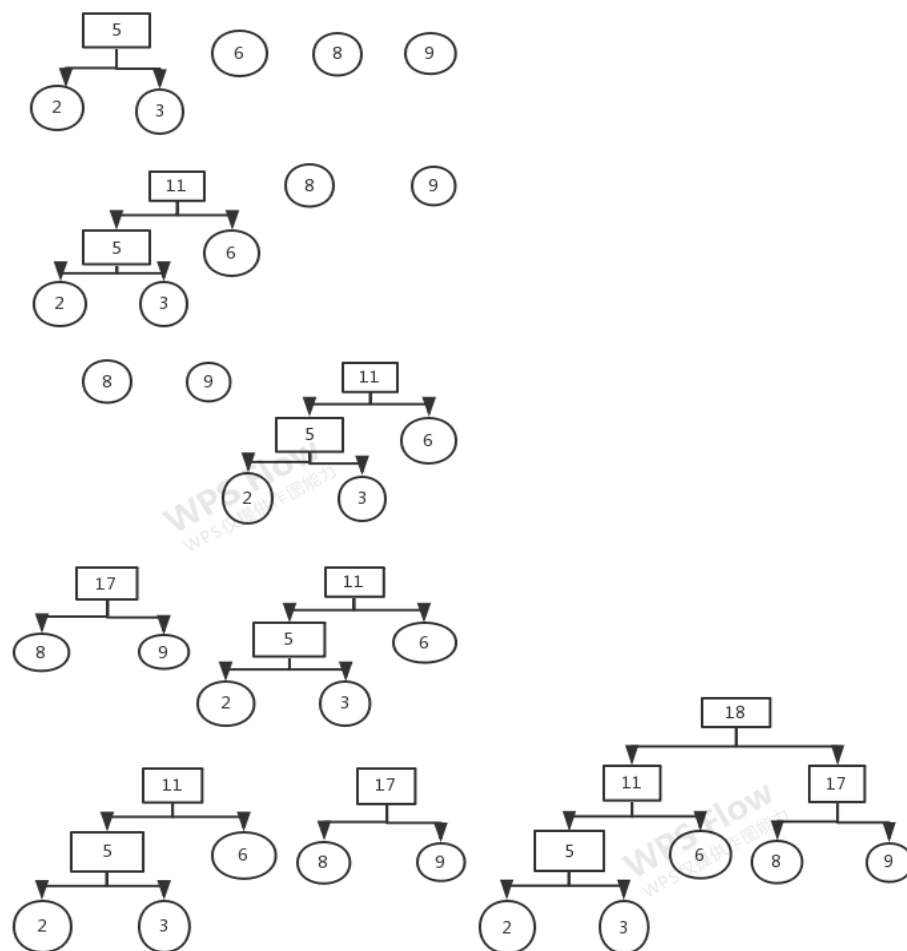
int main ()
{
    printf("%d\n",calcula(creat_tree(input_cheak_str())));
    return 0;
}
```

## 实验（三）哈夫曼树编译码的实践

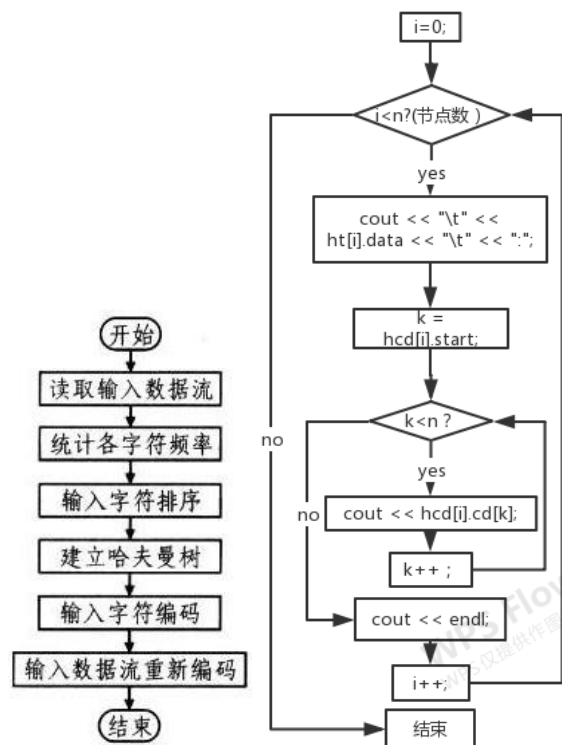
### <1> 系统描述：

- (1) 编写函数，实现创建哈夫曼树的算法；
- (2) 编写函数，实现求哈夫曼编码的算法；
- (3) 根据已知的字符及其权值，建立哈夫曼树，并输出哈夫曼编码

## <2> 功能模块结构:



根据已知的字符及其权值，建立哈夫曼树，并输出哈夫曼编码



### <3> 主要模块的算法说明:

#### (1) 函数实现创建哈夫曼树的算法

下面以 6、3、2、8、9 构造哈夫曼树进行举例说明。

(简而言之，就是按照一个贪心思想和规则进行树的构造，而构造出来的这个树的权值最小！且对于哈夫曼树的每个非叶子节点都有两个孩子，因为哈夫曼树的构造就是自底向上的构造，两两合并。)

1 初始时候各个数直都是一个单节点森林！然后进行排序。

6、3、2、8、9→2、3、6、8、9

2 放入优先队列(自己排序也行)，即每次取两个最小权值顶点，构造父节点。如果队列为空，那么返回节点，并且这个节点为根节点 root。否则继续加入队列进行排序。重复上述操作，直到队列为空。

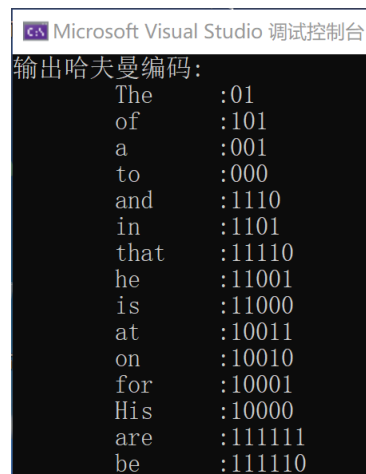
#### (2) 函数实现求哈夫曼编码的算法

先对输入的节点建立哈夫曼树，具体流程如 (1) 所述，再对左右子树进行标号，左子树标 0，右子树标 1。某节点的哈夫曼编码即为从根节点到此节点的路径上的标号序列。

(3) 根据已知的字符及其权值，建立哈夫曼树，并输出哈夫曼编码, 只须将已知的字符和权值输入给每个节点，再依次调用 CreateHT(ht,15);CreateHCode(ht, hcd,

15);Display(ht, hcd, 15);三个函数即可。其中 CreateHT 和 CreateHCode 函数的流程已在 (1) (2) 中给出, Display 函数作用为循环输出每个字符和其对应的哈夫曼编码

#### <4>运行结果



```
Microsoft Visual Studio 调试控制台
输出哈夫曼编码:
The      :01
of       :101
a        :001
to       :000
and      :1110
in       :1101
that     :11110
he       :11001
is       :11000
at       :10011
on       :10010
for      :10001
His      :10000
are      :111111
be       :111110
```

#### <5>课程设计总结:

1、 (1) 怎样将哈夫曼树的构造过程编写成代码?

我们很容易理解哈夫曼树在纸上的构造过程,在二叉树集合中选取两棵根节点权值最小的树作为左右子树构造一棵新的二叉树,且新的二叉树的根节点的权值为其左、右子树上根节点的权值之和;在二叉树集合中删除这两棵树,并将得到的二叉树加入集合中;重复上述步骤,直至二叉树集合中只含一棵树为止。但如何在代码中实现此过程是个难点。我通过查阅资料和对数据结构知识的记忆,知道了,使用结构体数组来存储节点的多个信息,并用循环结构中嵌if语句的结构实现多个节点的权值判断,之后再进行调整地赋值,最终实现了哈夫曼树的创建,具体代码可参照 2、实验方案的 1、 (1)。

2、在哈夫曼编码的求取过程中,如何对先求到的 01 编码进行保留呢?一开始我有点茫然,查阅资料后得知,原来只需简单一个数组即可顺利完成任务。每次求得一个编码后,先对其进行存储,然后移动这个数组的当前操作空间进行下一代编码的求取存储。

3、a 本次实验主要进行了哈夫曼树编译码的实践。通过上机实践，我们哈夫曼树的基本概念及其不同的实现方法的得到进一步掌握，补足了自己对树这种重要数据结构的认知，并对其上的常用操作有了进一步的认识。

下次实验我们将对图进行学习，重点是图的最小生成树和最短路径的实践，这是数在其他数据结构中的应用，是本次实验更进一步的延申。我会吸取本次实验的经验，下次实验希望更高效地完成。

## <6>参考文献

《数据结构》清华大学出版社

## <7>附录

(1) 创建哈夫曼树的算法

其中 HTNode 为树中每个节点的类型，n 为节点数。主函数调用形式为 CreateHT(ht, 15);

```
void CreateHT(HTNode ht[], int n){
    int i, k, lnode, rnode;
    double min1, min2;
    for (i = 0; i < 2 * n - 1; ++i)
        ht[i].parent = ht[i].lchild = ht[i].rchild = -1;
    for (i = n; i < 2 * n - 1; ++i)
    {
        min1 = min2 = inf;
        lnode = rnode = -1;
        for (k = 0; k <= i - 1; ++k)
            if (ht[k].parent == -1)
            {
                if (ht[k].weight < min1)
                {
                    min2 = min1;
                    rnode = lnode;
                    min1 = ht[k].weight;
                    lnode = k;
                }
                else if (ht[k].weight < min2)
                {
                    min2 = ht[k].weight;
                    rnode = k;
                }
            }
        ht[lnode].parent = i;
        ht[rnode].parent = i;
        ht[i].weight = ht[lnode].weight + ht[rnode].weight;
    }
}
```

```

        }
    }
    ht[i].weight = ht[lnode].weight + ht[rnode].weight;
    ht[i].lchild = lnode;
    ht[i].rchild = rnode;
    ht[lnode].parent = i;
    ht[rnode].parent = i;
}
}

```

## (2) 函数实现求哈夫曼编码的算法

其中 HTNode 为树中每个节点的类型，HCode 为存放每个节点哈夫曼编码的结构，n 为节点数。主函数调用形式为 CreateHCode(ht, hcd, 15);

```

void CreateHCode(HTNode ht[], HCode hcd[], int n)
{
    int i, f, c;
    HCode hc;
    for (i = 0; i < n; ++i)
    {
        hc.start = n;
        c = i;
        f = ht[i].parent;
        while (f != -1)
        {
            if (ht[f].lchild == c)
                hc.cd[hc.start--] = '0';
            else
                hc.cd[hc.start--] = '1';
            c = f;
            f = ht[f].parent;
        }
        hc.start++;
        hcd[i] = hc;
    }
}

```

## (3) 根据已知的字符及其权值，建立哈夫曼树，并输出哈夫曼编码

将 (1) (2) 结合起来，写在一个程序中，并补充不要的部分，在主函数中调用他们并再编写一个输出哈夫曼编码的函数，调用它以输出编码。其中，已知的字符及其权值已在主函数中给出。

```

#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
#define inf 999999
#define MAXN 100
typedef struct//树中每个节点的类型
{
    string data;//节点值
    double weight;//权重
    int parent, lchild, rchild;//双亲和左右孩子节点
} HTNode;

```

```

HTNode ht[MAXN];

typedef struct//存放每个节点哈夫曼编码的结构
{
    char cd[MAXN];//存放当前节点的哈夫曼编码
    int start;//存放该节点哈夫曼编码的起始位置，编码为 cd[start]~cd[n]
} HCode;
HCode hcd[MAXN];

void CreateHT(HTNode ht[], int n)//构造哈夫曼树
{
    int i, k, lnode, rnode;
    double min1, min2;
    for (i = 0; i < 2 * n - 1; ++i)
        ht[i].parent = ht[i].lchild = ht[i].rchild = -1;
    for (i = n; i < 2 * n - 1; ++i)
    {
        min1 = min2 = inf;
        lnode = rnode = -1;
        for (k = 0; k <= i - 1; ++k)
            if (ht[k].parent == -1)
            {
                if (ht[k].weight < min1)
                {
                    min2 = min1;
                    rnode = lnode;
                    min1 = ht[k].weight;
                    lnode = k;
                }
                else if (ht[k].weight < min2)
                {
                    min2 = ht[k].weight;
                    rnode = k;
                }
            }
        ht[i].weight = ht[lnode].weight + ht[rnode].weight;
        ht[i].lchild = lnode;
        ht[i].rchild = rnode;
        ht[lnode].parent = i;
        ht[rnode].parent = i;
    }
}

void CreateHCode(HTNode ht[], HCode hcd[], int n)//根据哈夫曼树求哈夫曼编码
{
    int i, f, c;
    HCode hc;
    for (i = 0; i < n; ++i)
    {
        hc.start = n;
        c = i;
    }
}

```



```

        f = ht[i].parent;
        while (f != -1)
        {
            if (ht[f].lchild == c)
                hc.cd[hc.start--] = '0';
            else
                hc.cd[hc.start--] = '1';
            c = f;
            f = ht[f].parent;
        }
        hc.start++;
        hcd[i] = hc;
    }
}

void Display(HTNode ht[], HCode hcd[], int n)
{
    int i, k;
    cout << "输出哈夫曼编码:" << endl;
    for (i = 0; i < n; i++)
    {
        cout << "\t" << ht[i].data << "\t" << ":";
        for (k = hcd[i].start; k <= n; k++)
        {
            cout << hcd[i].cd[k];
        }
        cout << endl;
    }
}

int main()
{
    string str[15] =
    { "The", "of", "a", "to", "and", "in", "that", "he", "is", "at", "on", "for", "His", "are", "be" };
    int frq[15] = { 1192, 677, 541, 518, 462, 450, 242, 195, 190, 181, 174, 157, 138, 124, 123 };
    for (int i = 0; i < 15; i++)
    {
        ht[i].data = str[i];
        ht[i].weight = frq[i];
    }
    CreateHT(ht, 15);
    CreateHCode(ht, hcd, 15);
    Display(ht, hcd, 15);
    return 0;
}

```

## 实验（四）图的最小生成树和最短路径的实践

### <1> 系统描述:

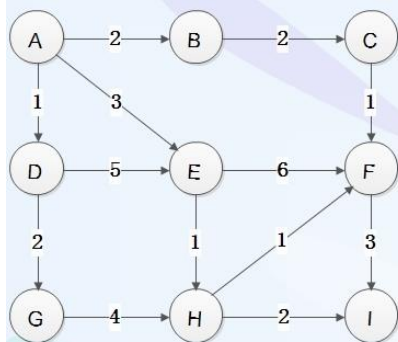
（1）编写用邻接矩阵表示无向带权图时图的基本操作的实现函数，主要包括：①初始化邻接矩阵表示的无向带权；②建立邻接矩阵表示的无向带权图；③输出邻接矩阵表示的无向带权图；编写生成最小生成树的 Prim 算法函数以及输出边集数组的函数

（2）利用邻接矩阵构造有向带权图，并求出某一顶点到其余顶点的最短路径并打印输出；编写求最短路径的 Dijkstra 算法函数，该算法求从顶点 i 到其余顶点的最短路径与最短路径长度；3.编写打印输出从源点到每个顶点的最短路径及长度的函数；

### <2> 功能模块结构:

1、（1）①





```

----- weighted directed matrix -----
---0---1---2---3---4---5---6---7---8---
---A---B---C---D---E---F---G---H---I---
-A|000-002-999-001-003-999-999-999-999-
-B|999-000-002-999-999-999-999-999-999-
-C|999-999-000-999-999-001-999-999-999-
-D|999-999-999-000-005-999-002-999-999-
-E|999-999-999-999-000-006-999-001-999-
-F|999-999-999-999-999-000-999-999-003-
-G|999-999-999-999-999-999-000-004-999-
-H|999-999-999-999-999-001-999-000-002-
-I|999-999-999-999-999-999-999-999-000-
----- weighted directed matrix -----

```

数组下标	0	1	2	3	4	5	6	7	8	初始数据
数组元素	A	B	C	D	E	F	G	H	I	
是否标记数组	0	0	0	0	0	0	0	0	0	
源点到图中各节点的距离数组	0	0	0	0	0	0	0	0	0	

- 1: 源点A到邻接点B,D,E的距离分别是2, 1, 3, 更新到距离数组中(顶点到自身的距离为0)。
- 2: 将源点A加入已标记后, 如下



数组下标	0	1	2	3	4	5	6	7	8	
数组元素	A	B	C	D	E	F	G	H	I	
是否标记数组	1	0	0	0	0	0	0	0	0	
源点A到图中各节点的距离数组	0	2	999	1	3	999	999	999	999	999表示距离无穷大

- 1: 从距离数组中选择一个未标记的最小值1, 对应图中的节点D。
- 2: 将节点D加入已标记。
- 3: 节点D两个邻接点E, G,
  - A到E的距离可表示为: A到D的距离+D到E的距离, 这个值 (5+1=6) 与当前距离数组中已存在的A到E的距离3大, 则不更新A到E的距离, 如下
  - A到G的距离可表示为: A到D的距离+D到G的距离, 这个值 (1+2=3) 与当前距离数组中已存在的A到G的距离999小, 则更新A到G的距离为3, 如下



数组下标	0	1	2	3	4	5	6	7	8	
数组元素	A	B	C	D	E	F	G	H	I	
是否标记数组	1	0	0	1	0	0	0	0	0	
源点A到图中各节点的距离数组	0	2	999	1	3	999	3	999	999	

- 1: 从距离数组中选择一个未标记的最小值2, 对应图中的节点B。
- 2: 将节点B加入已标记。
- 3: 节点B一个邻接点C
  - A到C的距离可表示为: A到B的距离+B到C的距离, 这个值 (2+2=4) 与当前距离数组中已存在的A到C的距离999小, 则更新A到C的距离为4, 如下



数组下标	0	1	2	3	4	5	6	7	8	
数组元素	A	B	C	D	E	F	G	H	I	
是否标记数组	1	1	0	1	0	0	0	0	0	
源点A到图中各节点的距离数组	0	2	4	1	3	999	3	999	999	

- 1: 从距离数组中选择一个未标记的最小值3, 对应图中的节点E。
- 2: 将节点E加入已标记。
- 3: 节点E有两个邻接点F、H,

A到F的距离=A到E的距离+E到F的距离, 这个值(3+6=9)与当前距离数组中已存在的A到F的距离999小, 则更新A到F的距离为9。  
A到H的距离=A到E的距离+E到H的距离, 这个值(3+1=4)与当前距离数组中已存在的A到H的距离999小, 则更新A到H的距离为4, 如下

数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	0	1	1	0	0	0	0
源点A到图中各节点的距离数组	0	2	4	1	3	9	3	4	999

- 1: 从距离数组中选择一个未标记的最小值3, 对应图中的节点G。
- 2: 将节点G加入已标记。
- 3: 节点G有一个邻接点H,

A到H的距离=A到G的距离+G到H的距离, 这个值(3+1=4)与当前距离数组中已存在的A到H的距离4大, 则不更新A到H的距离, 如下

数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	0	1	1	0	1	0	0
源点A到图中各节点的距离数组	0	2	4	1	3	9	3	4	999

- 1: 从距离数组中选择一个未标记的最小值4, 对应图中的节点C。
- 2: 将节点C加入已标记。
- 3: 节点C有一个邻接点F,

A到F的距离=A到C的距离+C到F的距离, 这个值(4+1=5)与当前距离数组中已存在的A到F的距离9小, 则更新A到F的距离为5, 如下

数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	1	1	1	0	1	0	0
源点A到图中各节点的距离数组	0	2	4	1	3	5	3	4	999

- 1: 从距离数组中选择一个未标记的最小值4, 对应图中的节点H。
- 2: 将节点H加入已标记。
- 3: 节点H有两个邻接点F、I,

A到F的距离=A到H的距离+H到F的距离, 这个值(4+1=5)与当前距离数组中已存在的A到F的距离5相等, 则不更新A到F的距离。  
A到I的距离=A到H的距离+H到I的距离, 这个值(4+2=6)与当前距离数组中已存在的A到I的距离999小, 则更新A到I的距离为6, 如下。

数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	1	1	1	0	1	1	0
源点A到图中各节点的距离数组	0	2	4	1	3	5	3	4	6

- 1: 从距离数组中选择一个未标记的最小值5, 对应图中的节点F。
- 2: 将节点F加入已标记。
- 3: 节点F有一个邻接点I,

A到I的距离=A到F的距离+F到I的距离, 这个值(5+3=8)与当前距离数组中已存在的A到I的距离6大, 则不更新A到I的距离, 如下。

数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	1	1	1	1	1	1	0
源点A到图中各节点的距离数组	0	2	4	1	3	5	3	4	6

- 1: 从距离数组中选择一个未标记的最小值6, 对应图中的节点 I。
- 2: 将节点 I 加入已标记。
- 3: 节点 I 没有邻接点, 则不处理。
- 4: 到此为止, 源点A到图中各节点的距离都已求出。



数组下标	0	1	2	3	4	5	6	7	8
数组元素	A	B	C	D	E	F	G	H	I
是否标记数组	1	1	1	1	1	1	1	1	1
源点A到图中各节点的距离数组	0	2	4	1	3	5	3	4	6

### <3>主要模块的算法说明:

1、(1) ①输出矩阵的时候注意最好是输出方正的形状, 即做到真正输出一个矩阵。因此需要采用双层循环进行行列的控制, 且每输完一行一定要换行, 具体关键的代码为:

```
for (i = 0; i < g->vexnum; i++)
{
    for (j = 0; j < g->vexnum; j++)
        cout<<g->arcs[i][j]<<" ";
    cout << endl;
}
```

而且行数、列数是从0开始的, 与实际常用的说法不同, 所以输入的时候需要注意。这是由于初始化邻接矩阵时便是这样的。

②Prim 算法的难点就在于如何找到两个点集之间的最短距离, 解决方法为用循环一一比较各个权值的大小。

(2) 有向图的邻接矩阵与无向图的区别在于它不是对称的, 但两种矩阵的输出函数却是基本相同的。这是因为我们已经在矩阵元素赋值处做了改动。这是一个易于解决但容易困惑的注意点。具体表现为:

无向图的创建邻接矩阵的函数为:

```
void InsertArc(WGraph* g)    /*插入边和权值*/
{
    int i, j, k, w;
    for (k = 0; k < g->arcnum; k++)
    {
        cout << "输入行数、列数、权值(行数、列数从0开始) : " << endl;
        cin>>i>>j>>w;
        g->arcs[i][j] = w;
        g->arcs[j][i] = w;
    }
}
```

```
}
```

其中关键在于  $g->arcs[i][j] = w; g->arcs[j][i] = w;$  即对称的两个元素权值相同。

而有向图的为：

```
for (int i = 0; i < g->n; i++)
{
    for (int j = 0; j < g->n; j++)
    {
        g->edges[i][j] = INF;
    }
}
cout<<"输入边及权(i j weight):"<<endl;
for (int count = 0; count < g->e; count++)
{
    cin>>i>>j>>w;
    g->edges[i][j] = w;
}
```

一条一条边的输入，并不一定对称。

Dijkstra 算法初识也具有一定的难度，但只要搞清楚原理，自然就轻松起来。

其基本思路与 BFS（深度优先搜索）有些类似，但不同的是 BFS 用的是一般队列，也就是严格的先进先出。而 Dijkstra 算法用到的则是优先队列。优先队列中是有序的队列，其顺序取决于规定的规则。比如可以规定值大的在前面，也可以规定值小的在前面。具体的算法过程为：有一个保存到每个点最短路径的数组（如命名为 `shortlen[]`，默认值为无穷，代表无通路）和一个记录点是否被访问过的数组。一开始，从起点开始，用每一个与起点相连的且没有被访问过的点的距离对 `shortlen` 数组进行更新，例如：第  $i$  个点与起始点的距离为  $x$ ， $x$  小于无穷，那么久把 `shortlen[i]` 的值更新为  $x$ 。只要有通路的点，全部放入优先队列然后把这个点记为被访问过。然后就从队列里取出队头，将其当做新的起点，重新进行上面的操作。当队列为空时跳出循环，这个时候的 `shortlen` 数组的值就是起点到各个点的最短距离。

## <4>运行结果

1、 (1) ①

```

Microsoft Visual Studio 调试控制台
读入顶点个数:
3
读入边数:
3
读入顶点信息:
第1个顶点:
1
第2个顶点:
2
第3个顶点:
3
插入边
输入行数、列数、权值(行数、列数从0开始):
0 0 1
输入行数、列数、权值(行数、列数从0开始):
0 1 2
输入行数、列数、权值(行数、列数从0开始):
0 2 3
输出矩阵
1 2 3
2 0 0
3 0 0

```

②

```

Microsoft Visual Studio 调试控制台
8
0 3 2 0 0 0 0 0
3 0 3 6 0 0 0 0
2 3 0 0 5 5 0 0
0 6 0 0 2 0 0 2
0 0 5 2 0 4 3 0
0 0 5 0 4 0 4 0
0 0 0 0 3 4 0 6
0 0 0 2 0 0 6 0
The weight of MST is: 21
The edges of MST are:
(2, 0)
(1, 0)
(4, 2)
(3, 4)
(7, 3)
(6, 4)
(5, 4)

```

(2)

```

Microsoft Visual Studio 调试控制台
create MGraph (邻接矩阵类)
输入顶点数, 边数(n e): 4 4
输入边及权(i j weight):
0 0 1
0 2 3
1 1 2
3 3 5
-----
print MGraph (邻接矩阵类)
1 9999 3 9999
9999 2 9999 9999
9999 9999 9999 9999
9999 9999 9999 5
-----
V0->V2的路径长度为: 3

```



```
Microsoft Visual Studio 调试控制台
输出哈夫曼编码:
The      :01
of       :101
a        :001
to       :000
and      :1110
in       :1101
that     :11110
he       :11001
is       :11000
at       :10011
on       :10010
for      :10001
His      :10000
are      :111111
be       :111110
```

## <5>课程设计总结:

本次实验主要进行图的最小生成树和最短路径的实践。通过这次试验，我对图的基本概念，以及图的最小生成树和最短路径的实现方法的理论得到进一步的掌握，并对在不同存储结构上实现不同的运算方式和技巧得到了更深的体会。这次试验带我们又再一次复习了重要的数据结构——树，并实践了之前只在书本上学习过的应用，拓展了我们的思维，相信对以后的计算机学习有很大的帮助。

下次实验我们将进行最终的系统设计，具体重点为为界面和人机交互的设计。即任选一之前的实际应用：机票订票系统、四则运算表达式求值、哈夫曼编/译码器、校园景点规划和导游，完成界面设计，形成良好人机交互的原型系统。这对我们来说是一个新的挑战，因为我们之前并未学习过这一方面的设计，因此我会先对这一操作进行了解和深入学习，之后入手进行制作和设计。

## <6>参考文献

《数据结构》清华大学出版社

## <7>附录

1、（1）①建立并输出邻接矩阵表示的无向带权图

分别编写不同的函数来完成各个任务，在主函数调用相应的函数即可。由于输入的信息较多，因此需要考虑到用户，在程序中需体现相应的提示输入语句。

```
#include<iostream>
using namespace std;
#define vnum 20
```

```

typedef struct gp
{
    int vexs[vnum];          /*顶点信息*/
    int arcs[vnum][vnum];    /*邻接矩阵*/
    int vexnum, arcnum;      /*顶点数、边数*/
}WGraph;

void CreateGraph(WGraph* g, int v, int e) /*初始化邻接矩阵*/
{
    int i, j;
    g->vexnum = v;
    g->arcnum = e;
    for (i = 0; i < g->vexnum; i++)
        for (j = 0; j < g->vexnum; j++)
            g->arcs[i][j] = 0;
}

void InsertVex(WGraph* g)      /*读入顶点信息*/
{
    int i;
    for (i = 0; i < g->vexnum; i++)
    {
        cout<<"第"<<i + 1<<"个顶点:"<<endl;
        cin>>g->vexs[i];
    }
}

void InsertArc(WGraph* g)      /*插入边和权值*/
{
    int i, j, k, w;
    for (k = 0; k < g->arcnum; k++)
    {
        cout<<"输入行数、列数、权值(行数、列数从0开始)："<<endl;
        cin>>i>>j>>w;
        g->arcs[i][j] = w;
        g->arcs[j][i] = w;
    }
}

void OutGraph(WGraph* g)      /*输出矩阵*/
{
    int i, j;
    for (i = 0; i < g->vexnum; i++)

```

```

    {
        for (j = 0; j < g->vexnum; j++)
            cout<<g->arcs[i][j]<<" ";
        cout << endl;
    }
}
int main()
{
    WGraph g;
    int x, n;
    cout<<"读入顶点个数:"<<endl;
    cin>>x;
    cout<<"读入边数:"<<endl;
    cin>>n;
    CreateGraph(&g, x, n);
    cout<<"读入顶点信息:"<<endl;
    InsertVex(&g);
    cout<<"插入边"<<endl;
    InsertArc(&g);
    cout<<"输出矩阵"<<endl;
    OutGraph(&g);
    return 0;
}

```

②编写生成最小生成树的 Prim 算法函数并输出边集数组

使用贪心算法：

- 1.选择一个初始点 v1;
- 2.为了使得生成树的权极小化，选择 v1 的最小权邻点，并将其连接到 v1;
- 3.选择{v1,v2}的最小权邻点，如 v3，将其连接;
- 4.重复以上步骤，直到所有点都连在一起，得到生成树。

```

#include <iostream>
#include <vector>
#include <limits>
#include <map>
using namespace std;
typedef float Weight;
typedef int Vertex;
typedef pair<Vertex, Vertex> Edge;
typedef vector<Edge> Edges;
typedef map<Vertex, bool> Vertices;

```

```

typedef vector<vector<Weight>> GraphMatrix;
typedef vector<pair<Vertex, Weight>> Labels;
float inf = numeric_limits<float>::max();
// 求 V 的最小权邻点
Vertex nearest_neighbour(Vertices& vertices, Labels& label) {
    Vertex u;
    int n = vertices.size();
    Weight w = inf;
    for (int i = 0; i < n; i++) {
        if (!vertices[i] && label[i].second < w) {
            w = label[i].second;
            u = i;
        }
    }
    return u;
}

pair<Edges, Weight> prim(GraphMatrix& g, Vertex s) {
    int n = g.size();
    Vertices V;
    for (int i = 0; i < n; i++)
        V[i] = false;
    V[s] = true;
    Edges E;
    Weight w = 0; // weight of MST
    Labels L(n);
    for (int i = 0; i < n; i++)
        L[i] = make_pair(s, g[s][i]);
    for (int i = 0; i < n - 1; i++) {
        int u = nearest_neighbour(V, L); // 找到最小权的连接点
        V[u] = true;
        E.push_back(make_pair(u, L[u].first));
        w += g[u][L[u].first];
        for (int j = 0; j < n; j++) {
            if (!V[j] && (g[u][j] < inf) && (g[u][j] < L[j].second))
                L[j] = make_pair(u, g[u][j]);
        }
    }
    return make_pair(E, w);
}

```

```

int main() {
    int numberOfVertices;
    cin >> numberOfVertices;
    GraphMatrix graph(numberOfVertices);
    for (int i = 0; i < numberOfVertices; i++) {
        for (int j = 0; j < numberOfVertices; j++) {
            Weight w;
            cin >> w; // 如果两点没有连通, 则输入 0
            if (w == 0) w = inf;
            graph[i].push_back(w);
        }
    }
    pair<Edges, Weight> MST = prim(graph, 0);
    cout << "The weight of MST is: " << MST.second << endl;
    int n = MST.first.size();
    cout << "The edges of MST are: " << endl;
    for (int i = 0; i < n; i++)
        cout << "(" << MST.first[i].first << ", " << MST.first[i].second << ")" << endl;
    return 0;
}

```

(2) 按题目要求编写的代码如下, 其中, 此代码求的是从  $v_0$  到其余顶点的最短路径。

```

#include <iostream>
using namespace std;
const int maxv = 20;
//INF 定义为无穷大, 表示不连通
const int INF = 9999;
bool visited[maxv];
int u;
/* MGraph 类 (邻接矩阵构建图类) */
class MGraph
{
private:
    typedef struct
    {
        int edges[maxv][maxv];
        int n, e;
    } Graph; // 定义图类型

```

```
Graph* g;//定义图类型变量 g
```

```
public:
```

```
MGraph()//构造函数内，初始化成员 g
```

```
{  
    g = (Graph*)malloc(sizeof(Graph));  
}
```

```
/* 构建图（邻接矩阵） */
```

```
void create_MGraph()
```

```
{  
    cout<<"输入顶点数,边数(n e): ";  
    cin>>g->n>>g->e;  
    int i, j, w;  
    for (int i = 0; i < g->n; i++)  
    {  
        for (int j = 0; j < g->n; j++)  
        {  
            g->edges[i][j] = INF;  
        }  
    }  
    cout<<"输入边及权(i j weight):"<<endl;  
    for (int count = 0; count < g->e; count++)  
    {  
        cin>>i>>j>>w;  
        g->edges[i][j] = w;  
    }  
}
```

```
/* 打印图（邻接矩阵） */
```

```
void print_MGraph()
```

```
{  
    for (int i = 0; i < g->n; i++)  
    {  
        for (int j = 0; j < g->n; j++)  
        {  
            cout<<g->edges[i][j]<<" ";  
        }  
        cout<<endl;  
    }  
}
```

```

/* 输出任意顶点 V0，到其他顶点的最短路径 */
void Dispath(int dist[], int path[], int S[], int v)
{
    int k, d;
    for (int i = 0; i < g->n; i++)
        if (S[i] == 1 && i != v)
        {
            cout<<"V"<<v<<"->V"<<i<<"的路径长度为:"<< dist[i]<<endl;
            k = path[i];
            if (k == -1)
                cout<<"无路径"<<endl;
        }
}

/* Dijkstra 算法，求最短路径 */
void Dijkstra(int v)
{
    int dist[maxv], path[maxv];
    int S[maxv];
    int mindist;
    for (int i = 0; i < g->n; i++)
    {
        dist[i] = g->edges[v][i];
        S[i] = 0;
        if (g->edges[v][i] < INF)
            path[i] = v;
        else
            path[i] = -1;
    }
    S[v] = 1, path[v] = 0;
    for (int i = 0; i < g->n; i++)
    {
        mindist = INF;
        for (int j = 0; j < g->n; j++)
        {
            if (S[j] == 0 && dist[j] < mindist)
            {
                u = j;
                mindist = dist[j];
            }
        }
    }
}

```

```

        }
    }
    S[u] = 1;
    for (int j = 0; j < g->n; j++)
    {
        if (S[j] == 0)
        {
            if (g->edges[u][j] < INF && dist[u] + g->edges[u][j] < dist[j])
            {
                dist[j] = dist[u] + g->edges[u][j];
                path[j] = u;
            }
        }
    }
}

Dispath(dist, path, S, v);
}

/* g->n 为 MGraph 类私有成员
只能通过函数访问，获得图的边数 n */
int MGraph_n()
{
    return g->n;
}

};

int main()
{
    MGraph mgraph; // MGraph 类，实例化对象 mgraph
    // freopen("data.txt", "r", stdin);
    cout << "create MGraph (邻接矩阵类) " << endl;
    mgraph.create_MGraph();
    cout << "-----" << endl;
    cout << "print MGraph (邻接矩阵类) " << endl;
    mgraph.print_MGraph();
    cout << "-----" << endl;
    /* 输出任意顶点 V0，到其他顶点的最短路径 */
    mgraph.Dijkstra(0);
    return 0;
}

```



# 实验（五）四则运算表达式的求值系统设计 （四选一选做部分）

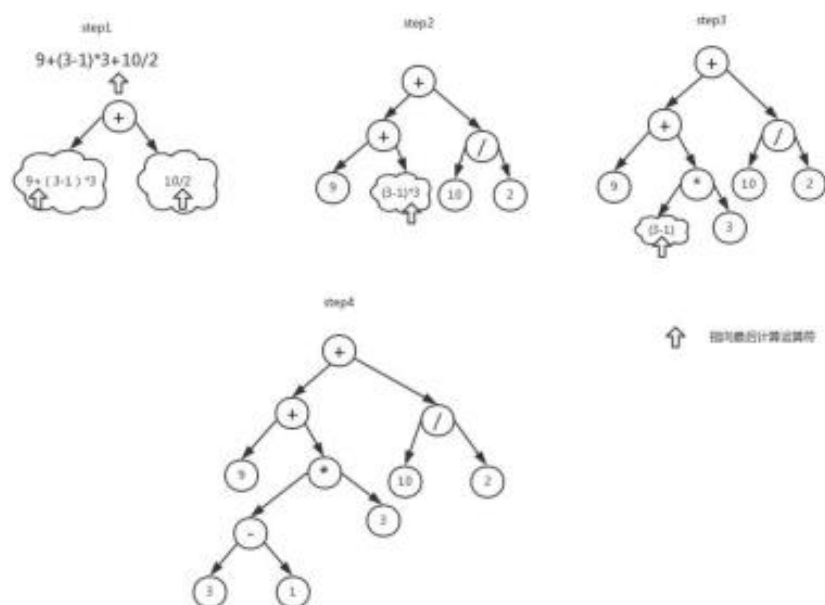
## <1> 系统描述：

（1）结合基本操作，建立表达式二叉树，输出树的前中后序遍历的结果，计算表达式的值。

（2）当用户输入一个合法的算术表达式后，能够返回正确的结果。能够计算的运算符包括：加、减、乘、除、括号；能够计算的操作数要求在实数范围内；对于异常表达式能给出错误提示。

（3）设计并实现人机交互友好的界面或菜单。

## <2> 功能模块结构：

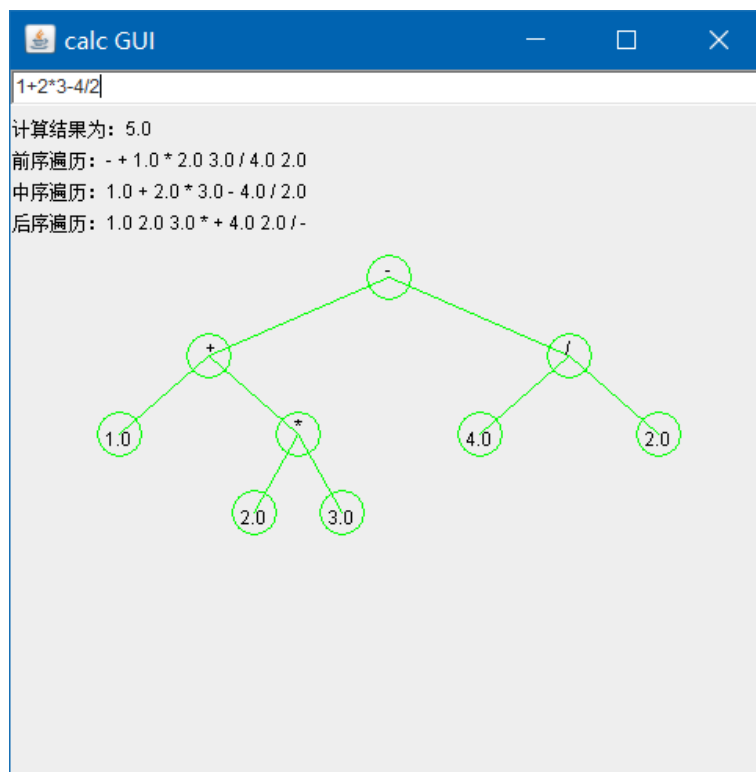


## <3> 主要模块的算法说明：

1、 ops 二维数组存各运算符之间的优先级

- 2、 nodeStack 和 opsStack 是分别存结点和运算符的栈
- 3、 从键盘得到当前输入的数字和运算符，通过运算符的数组下标判断当前运算符与 opsStack 栈顶元素的优先级，当栈顶元素优先级大于当前运算符时，将 nodeStack 中的元素弹出两个分别作为当前运算符结点的左孩子和有孩子进行运算，将得到的新的二叉树的结点存入栈中；当运算符优先级小时，直接将当前运算符压入栈中
- 4、 分别设计前序遍历、中序遍历、后序遍历的函数，实现三种遍历方法

## <4>运行结果



ExprTree

5\*3+2/1

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	=

ExprTree

17.0

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	<div></div>

ExprTree

3(

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	=

ExprTree

Error!

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	=

ExprTree

5/0

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	=

ExprTree

Infinity

7	8	9	+	-
4	5	6	*	/
1	2	3	(	)
D	0	.	C	<div></div>

## <5>课程设计总结：

之前虽然已经学过数据结构的课程，但这次课程设计让我知道了自己学的很浅，基础不牢，没有将理论知识与应用很好的结合起来，通过这次程序设计，我认识到动手能力和逻辑的重要性，当拿到一个问题时，该如何去考虑这个问题，怎么设计这个程序，程序的框架如何，分为哪些模块，各模块又该实现哪些功能，等等。都是我们要去思考的，思路清晰了以后我们写代码会事半功倍。

而在编程过程中，总是会有各种各样的 bug，编译的每一个 error 都很让人抓狂，有时，一些微小的错误却极大地影响了整体的效果。可能只是一个标点符号、一个字母的大小写打错了就导致全篇 error，所以我们在写代码的时候应该注重细节，多细心一些，这个道理不仅仅存在在一个程序中，也渗透在我们生活的方方面面，所以我们也从“debug”的过程参悟了很多人生哲理。

虽然课程设计只有短短几周的时间，但我还是收获颇多，独立思考能力、自学能力、逻辑思考能力都有了长足的进步。本次课设更是给我们的大二上学期画上了一个忙碌又充实的句号也使我受益匪浅。

在这个去实现这些数据结构的算法的过程中，本身便是一种学习新知识的过程。在之前我也尝试过在书本上预习过这些算法，但不得不说的是，很难理解，并且容易忘记。但是，在实践的过程中学习得到的知识，却很难忘记，是一种强化学习的方法。同时，也为了我日后数据结构的学习打下了良好的基础。

其次学习方法的选择：计算机编程知识复杂繁多，以人力却将其全部学完，再进行编程显然是不现实的。所以，面向引用的学习就是一种可取的方法。简而言之，就是需要什么学什么。

三是在过去写程序的过程中，我们往往只被要求了如何实现算法，而忽视了代码的可读性，没有养成添加注释的良好习惯。在本次的高级程序设计实验中，只要代码的量稍微大一点，我们往往就会忘记这一处的代码的作用，或者是某个变量的意义，因此，添加注释便在大型工程项目中显得尤为重要。

我越来越感觉到自己实力的不足，在以后的日子，希望自己真的能够扎实地学习数据结构等专业知识，努力地优化自己的代码运行效率。

## <6>参考文献

《java 语言程序与设计》郭克华

## <7> 附录

代码清单：

```
package textcode;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.TextField;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.util.Stack;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

class TreeNode {
    double value; // 存数值
    char op = 'E'; // 初始设为 E，用来标记是否为数字
    TreeNode lft;
    TreeNode rt;

    TreeNode(double value) {
        this.value = value;
    }

    TreeNode(double value, char op, TreeNode lft, TreeNode rt) {
        this.value = value;
        this.op = op; // 存操作符
        this.lft = lft;
        this.rt = rt;
    }
}
```

```
}
```

```
StringBuilder buf = new StringBuilder();
```

```
public String toString() {  
    buf = new StringBuilder();  
    outPreOrder(this);  
    return buf.toString();  
}
```

```
public String preOrder() { // 前序遍历  
    buf = new StringBuilder();  
    outPreOrder(this);  
    return buf.toString();  
}
```

```
private void outPreOrder(TreeNode node) { // 前序遍历  
    if (node == null) return;  
    if (node.op != 'E')  
        buf.append(node.op); // 操作符  
    else  
        buf.append(node.value); // 数值  
    buf.append(" ");  
    outPreOrder(node.lft);  
    outPreOrder(node.rt);  
}
```

```
public String inOrder() { // 中序  
    buf = new StringBuilder();  
    outInOrder(this);  
    return buf.toString();  
}
```

```
private void outInOrder(TreeNode node) { // 中序遍历
```



```

    if (node == null) return;
    outInOrder(node.lft);
    if (node.op != 'E')
        buf.append(node.op);
    else
        buf.append(node.value);
    buf.append(" ");
    outInOrder(node.rt);
}

```

```

public String postOrder() { // 后序
    buf = new StringBuilder();
    outPostOrder(this);
    return buf.toString();
}

```

```

private void outPostOrder(TreeNode node) { // 后序遍历
    if (node == null) return;
    outPostOrder(node.lft);
    outPostOrder(node.rt);
    if (node.op != 'E')
        buf.append(node.op);
    else
        buf.append(node.value);
    buf.append(" ");
}
}

```

```

class SwingConsole {
    public static void run(final ExprTree f, final int width, final int height) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                f.setTitle(f.getClass().getSimpleName());
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            }
        });
    }
}

```

```

        f.setSize(width, height);
        f.setVisible(true);
    }
});
}
}

/**
 * 图形界面的计算器程序，只能计算加减乘除，
 * 算式中可以有小括号。数字可以是小数
 */
public class ExprTree extends JFrame {

    private JTextField textField;          // 输入文本框
    private String input;                  // 结果

    public ExprTree() {
        input = "";
        Container container = this.getContentPane(); // 计算器
        JPanel panel = new JPanel();
        textField = new JTextField(30);
        textField.setEditable(false);          // 文本框禁止编辑
        textField.setHorizontalAlignment(JTextField.LEFT);
        textField.setPreferredSize(new Dimension(200, 30));
        container.add(textField, BorderLayout.NORTH);

        String[] name = {"7", "8", "9", "+", "-", "4", "5", "6", "*", "/", "1", "2", "3", "(", ")", "D",
            "0", ".", "C", "="};

        panel.setLayout(new GridLayout(4, 5, 1, 1));

        for (String s : name) {
            JButton button = new JButton(s);
            button.addActionListener(new MyActionListener());

```

```

        panel.add(button);
    }
    container.add(panel, BorderLayout.CENTER);
}

```

```

class MyActionListener implements ActionListener {           // 内部类实现按钮响应

```

```

    @Override

```

```

    public void actionPerformed(ActionEvent e) {

```

```

        String actionCommand = e.getActionCommand();

```

```

        switch (actionCommand) {

```

```

            case "C":           // 清除输入

```

```

                input = "";

```

```

                break;

```

```

            case "D":

```

```

                input = input.substring(0, input.length() - 1);

```

```

                break;

```

```

            case "=":

```

```

                textFieldString = input;

```

```

                calcSuccess = true;

```

```

                resultTree = null;

```

```

                try {

```

```

                    resultTree = calc(textFieldString + "#"); // 在输入的表达式后面加上 “#”

```

一起操作

```

                } catch (Exception e1) {

```

```

                    calcSuccess = false;

```

```

                }

```

```

            if (calcSuccess) {

```

```

                input = String.valueOf(resultTree.value); // 成功;计算表达式的值

```

```

            } else {

```

```

                input = "Error!";

```

```

            }

```

```

            ExprTree.this.repaint();

```

```

            break;

```

```

        default:
            input += actionCommand;           // 按下数字
            break;
    }

    textField.setText(input);
}
}

private static final long serialVersionUID = 1L;
private TreeNode resultTree;
private String textFieldString;
private boolean calcSuccess = true;
private char[][] ops = {
    {'>', '>', '<', '<', '<', '>', '>'},
    {'>', '>', '<', '<', '<', '>', '>'},
    {'>', '>', '>', '>', '<', '>', '>'},
    {'>', '>', '>', '>', '<', '>', '>'},
    {'<', '<', '<', '<', '<', '=', 'E'},
    {'E', 'E', 'E', 'E', 'E', 'E', 'E'},
    {'<', '<', '<', '<', '<', 'E', '='},
}; // 用二维数组存各运算符之间的优先级
Stack<TreeNode> nodesStack = new Stack<>(); // 存结点的栈
Stack<Character> opsStack = new Stack<>(); // 存运算符的栈

public void userGUI() { // 画二叉树
    this.setLayout(new BorderLayout());
    TextField tf = new TextField("", 40);
    tf.getText();
    tf.selectAll();
    tf.addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_ENTER) {
                textFieldString = ((TextField) e.getComponent()).getText();
            }
        }
    });
}

```

```

        calcSuccess = true;
        resultTree = null;
        try {
            resultTree = calc(textFieldString + "#");
        } catch (Exception e1) {
            calcSuccess = false;
        }
        ExprTree.this.repaint();
    }
}

});
this.add(tf, BorderLayout.NORTH);
this.setSize(500, 500);
this.setTitle("calc GUI");
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setResizable(true);
this.setVisible(true);
}

```

```

private void drawCircle(Graphics g, Point p, int r) {
    g.drawOval(p.x - r, p.y - r, r * 2, r * 2);
}

```

```

private void drawTree(Graphics g, TreeNode node, Point pme, int width, Point pfather) {
    if (node == null) return;
    g.setColor(Color.GREEN);
    int diameter = 28;
    this.drawCircle(g, pme, diameter / 2);
    g.drawLine(pme.x, pme.y, pfather.x, pfather.y);
    if (node.op != 'E') { // 父亲结点存的是运算符
        g.setColor(Color.BLACK);
        g.drawString(String.valueOf(node.op), pme.x-2, pme.y);
    } else { // 父亲结点存的是数字
        g.setColor(Color.BLACK);
    }
}

```

```

        g.drawString(String.valueOf(node.value), pme.x - diameter / 2 + 5, pme.y + 8);
    }
    int levelHeight = 50;
    drawTree(g, node.lft, new Point(pme.x - width / 2, pme.y + levelHeight), width / 2, pme);
    drawTree(g, node.rt, new Point(pme.x + width / 2, pme.y + levelHeight), width / 2, pme);
}

public TreeNode calc(String inStr) throws Exception {
    opsStack.push('#');
    StringBuilder buf = new StringBuilder();
    int i = 0;
    while (i < inStr.length()) {
        if (Character.isDigit(inStr.charAt(i)) || inStr.charAt(i) == '.') { // number
            buf.delete(0, buf.length()); // 先清零
            while (i < inStr.length() &&
                (Character.isDigit(inStr.charAt(i)) || inStr.charAt(i) == '.')) // 是数字或者小数
                buf.append(inStr.charAt(i++));
            double number = Double.parseDouble(buf.toString()); // 存数字
            nodesStack.push(new TreeNode(number)); // 把数压进栈
        } else if (inStr.charAt(i) == ' ') { // 可能存在输入空格的情况
            i++;
        } else { // operation
            char op = inStr.charAt(i);
            int subNew = getSub(op); // 得到当前运算符的代表数字下标
            boolean goOn = true;
            while (goOn) {
                if (opsStack.isEmpty())
                    throw new Exception("运算符太少!");
                char opFormer = opsStack.peek(); // 获取当前栈顶元素，就是目前栈顶位置的
                // 运算符就
                int subFormer = getSub(opFormer); // 获取当前栈顶元素的运算符的代表数字
                // 下标
                switch (ops[subFormer][subNew]) {

```

```

        case '=':
            goOn = false;
            opsStack.pop(); // 左括号右括号, #和#, 在传表达式进来的时候在最后
            后面加上了一个#, 用来判断是否运算结束
            break;
        case '<':
            goOn = false;
            opsStack.push(op); // 前一个运算符的优先级小于现在这个运算符, 把
            当前运算符压进栈
            break;
        case '>': // 前一个运算符的优先级大于当前的运算符, 把前一个运算符
            弹出栈进行运算, 再把当前运算符压进栈
            goOn = true;
            TreeNode n1 = nodesStack.pop(); // 先弹出来的做右孩子
            TreeNode n0 = nodesStack.pop(); // 后弹出来的做左孩子
            double rs = doOperate(n0.value, n1.value, opFormer); // 对弹出来的两个数
            值和运算符进行运算
            nodesStack.push(new TreeNode(rs, opFormer, n0, n1));
            opsStack.pop();
            break;
        default:
            throw new Exception("没有匹配的操作符: " + op);
    }
}
i++;
}
}
return nodesStack.pop(); // 结点栈弹出头结点
}

```

```

private double doOperate(double n0, double n1, char op) throws Exception { // 对弹出来
    的两个数值和符号进行运算

```

```

    switch (op) {

```

```

        case '+':
            return n0 + n1;
        case '-':
            return n0 - n1;
        case '*':
            return n0 * n1;
        case '/':
            return n0 / n1;
        default:
            throw new Exception("非法操作符: " + op);
    }
}

```

private int getSub(char c) { //用二维数组的数字下标来代表操作符，用数组存储它们的优先关系

```

    switch (c) {
        case '+':
            return 0;
        case '-':
            return 1;
        case '*':
            return 2;
        case '/':
            return 3;
        case '(':
            return 4;
        case ')':
            return 5;
        case '#':
            return 6;
        default:
            return -1;
    }
}

```



```

public void paint(Graphics g) { //画二叉树
    super.paint(g);
    if (calcSuccess) {
        if (resultTree != null) {
            g.drawString("计算结果为: " + resultTree.value, 10, 80);
            g.drawString("前序遍历: " + resultTree.preOrder(), 10, 100);
            g.drawString("中序遍历: " + resultTree.inOrder(), 10, 120);
            g.drawString("后序遍历: " + resultTree.postOrder(), 10, 140);
            int rootBeginX = this.getWidth() / 2;
            int rootBeginY = 170;
            Point p = new Point(rootBeginX, rootBeginY);
            drawTree(g, resultTree, p, this.getWidth() / 2 - 20, p);
        }
    } else {
        g.setColor(Color.RED);
        g.drawString("表达式语法有误! ", 20, 100);
    }
}

public static void main(String[] args) {
    ExprTree gui = new ExprTree();
    SwingConsole.run(gui, 250, 300); //实现画二叉树功能时将这一行注释掉
    //gui.userGUI(); //实现计算器功能时将这一行注释掉, 并将上面的 paint 函数修改名字, 让程序不能调用 paint 函数
}
}

```

