



## 教学上机实验报告

课程名称： 数据结构

任课教师姓名： 贾盼盼

学生学号： 312105010207

学生姓名： 刘晨阳

学生专业班级： 计算 2106

2021 ~ 2022 学年 第一学期

## 河南理工大学

### 教学上机实验报告评价分值标准

序号	评价指标	分值	评价等级及参考分值					评价分
			优	良	中	合格	差	
1	实验报告内容完整充实	10	10	8	7	6	3	
2	实验内容书写规范、字迹工整认真	10	10	8	7	6	3	
3	实验过程叙述详细、概念正确，语言表达准确，结构严谨，调理清楚，逻辑性强，自己努力完成，没有抄袭。	30	30	26	23	20	10	
4	对实验过程中存在的问题分析详细透彻、深刻、全面、规范、，结合实验内容，有自己的个人见解和想法，并能结合该实验提出相关问题，给出解决方法。	30	30	26	23	20	10	
5	实验结果、分析和结论正确无误	20	20	17	15	13	6	

总得分		
<div>签名（签章）：</div> <div>日期：    年   月   日</div>		

目录

1.实验一 .....	3
2.实验二 .....	9
3.实验三 .....	11
4.实验四 .....	14
5.实验五 .....	17
6.实验六 .....	22
7.实验七 .....	30
8.实验八 .....	39

1.实验一

<div>河南理工大学教学上机实验报告</div> <div>上机时间   2022 年 9 月 21   日</div>
---

**实验题目：**

写出顺序表的初始化，按位取元素，插入，删除，遍历，有序表的合并及就地逆置算法。

参照附件中程序，实现程序中部分功能（7、8、9）。

**实验目的和要求：**

掌握顺序表的相关算法及应用，了解顺序表在操作时的时效性，要特别注意算法的健壮性

**实验过程：****程序主要功能源代码：**

```
int main()
{
    SqList L;
    int i,res,temp,a,b,c,e,choose,j;
    int max=0;int index=0;
    cout<<"1. 建立顺序表\n";
    cout<<"2. 输入数据\n";
    cout<<"3. 查找\n";
    cout<<"4. 插入\n";
    cout<<"5. 删除\n";
    cout<<"6. 输出数据\n";
    cout<<"7.最大数及位置\n";
    cout<<"8.按元素值删除\n";
    cout<<"9.逆置\n";
    cout<<"0. 退出\n\n";

    choose=-1;
    while(choose!=0)
    {
        cout<<"请选择:";
        cin>>choose;
        switch(choose)
        {
            case 1:
                if(InitList_Sq(L))                //创建顺序表
                    cout<<"成功建立顺序表\n\n";
```

```

else
    cout<<"顺序表建立失败\n\n";
break;
case 2:                                     //输入 10 个数
    cout<<"请输入 10 个数:\n";
    for(i=0;i<10;i++)
        cin>>L.elem[i];
    L.length=10;
    cout<<endl;
    break;
case 3:                                     //顺序表的查找
    cout<<"请输入所要查找的数:";
    cin>>e;                                //输入 e, 代表所要查找的数值
    temp=LocateElem_Sq(L,e);
    if(temp!=0)
        cout<<e<<" 是第 "<<temp<<"个数.\n\n";
    else
        cout<<"查找失败! 没有这样的数\n\n";
    break;
case 4:                                     //顺序表的插入
    cout<<"请输入两个数, 分别代表插入的位置和插入数值:";
    cin>>a>>b;                            //输入 a 和 b, a 代表插入的位置, b 代表插入的数值
    if(ListInsert_Sq(L,a,b))
        cout<<"插入成功.\n\n";
    else
        cout<<"I 插入失败.\n\n";
    break;
case 5:                                     //顺序表的删除
    cout<<"请输入所要插入的数:";
    cin>>c;                                //输入 c, 代表要删除数的位置
    if(ListDelete_Sq(L,c,res))
        cout<<"删除成功.\n 被删除的数是:"<<res<<endl<<endl;
    else
        cout<<"删除失败.\n\n";
    break;
case 6:                                     //顺序表的输出
    cout<<"当前顺序表为:\n";
    for(i=0;i<L.length;i++)
        cout<<L.elem[i]<<" ";
    cout<<endl<<endl;
    break;
case 7:
    cout<<"查找最大数及其位置";

    for(i=0;i<10;i++)

```

```

        {
            if(L.elem[i]>max)
            {
                max=L.elem[i];
                index=i;
            }
        }
        printf("最大数为: %d\n 最大数下标为: %d\n",max,index);
        break;
    case 8:
        cout<<"请输入你要删除的元素\n";
        int e; int index_8;
        scanf("%d",&e);
        index_8=LocateElem_Sq(L,e);
        ListDelete_Sq(L,index_8,e);
        cout<<"删除元素成功\n";
        break;
    case 9:
        if(L.length%2==0)
        {
            for(i=L.length-1,j=0;j<(L.length/2)-1,i>=L.length/2;j++,i--)
            {
                temp=L.elem[j];
                L.elem[j]=L.elem[i];
                L.elem[i]=temp;
            }
        }
        else
        {
            for(i=L.length-1,j=0;j<(L.length-1)/2,i>(L.length-1)/2;j++,i--)
            {
                temp=L.elem[j];
                L.elem[j]=L.elem[i];
                L.elem[i]=temp;
            }
        }

        cout<<"逆置元素为\n";
        for(j=0;j<L.length;j++)
        {
            printf("%d ",L.elem[j]);
        }
        cout<<"\n";
        cout<<"逆置元素完成";break;
    }
}

```

```
}  
    return 0;  
}
```

实验主要是对功能 7, 8, 9 进行补充

实验结果:

D:\IDE\_code\_practice\c\_practice(visualc++2010)\cpp\_project\Debug\cpp\_project.exe

```
1. 建立顺序表  
2. 输入数据  
3. 查找  
4. 插入  
5. 删除  
6. 输出数据  
7. 最大数及位置  
8. 按元素值删除  
9. 逆置  
0. 退出
```

请选择:1  
成功建立顺序表

请选择:2  
请输入10个数:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

请选择:7  
查找最大数及其位置最大数为: 10  
最大数下标为: 9

请选择:8  
请输入你要删除的元素

8  
删除元素成功

请选择:9  
逆置元素为

10 9 7 6 5 4 3 2 1  
逆置元素完成请选择:

实验分析:

该实验主要是对利用顺序表的存储结构对数据进行操作

程序前面有很多写好的操作函数，实验主要是对顺序表 7, 8, 9 功能操作进行补充，完成顺序表的查找最大元素，按照元素值进行删除以及对元素进行逆置。

在运行程序的时候要首先选择 1 进行一个顺序表的创建，然后再选择 2 对顺序表中元素进行输入，不能直接选择其他功能进行使用，否则会报错。

在实现删除元素的时候，每删除一个元素要记得对顺序表长度进行减一的操作，在实现逆置操作的时候，要根据顺序表的具体长度来设置循环次数，这里我采用的是对顺序表长度进行一个奇偶数的判断，从而对不同情况进行顺序表的逆置。

实验成绩：

日期：\_\_\_\_年\_\_月\_\_日



## 2.实验二

### 河南理工大学教学上机实验报告

上机时间 2021 年 9 月 26 日

#### 实验题目：

写出算法用链表实现一元多项式的加法与乘法运算

#### 实验目的和要求：

**实验内容：**写出算法用链表实现一元多项式的加法与乘法运算。

**实验要求：**使学生掌握在链表上实现按序插入，删除结点的算法，按要求格式实现在一元多项式的内容的输出。

#### 实验过程：

```
p3=p1;
p1=p1->next;

// _____;
// _____;
r = p2;
p2=p2->next;
// _____;
```

补充代码，用链表实现一元多项式的加法与乘法运算

运行结果：

```
请输入有关多项式pa系数和指数的数据文件名称（文件名+“.txt”，如PolyA.txt）：
PolyA.txt
请输入有关多项式pb系数和指数的数据文件名称（文件名+“.txt”，如PolyB.txt）：
PolyB.txt
多项式Pa和Pb相加后的结果是：
(7) * x^0 + (11) * x^1 + (22) * x^7 + (5) * x^17
请按任意键继续. . .
```

实验结果：

用链表实现了一元多项式的加法与乘法运算，分为指数相同和指数不同的情况，用链表进行操作，实现多项式的运算

实验分析：

该程序要将多项式的相关信息 txt 文件与源程序文件放在同一目录下，才能读取到相关信息。其次本程序是采用链表的形式，使用 p1, p2, p3 分别指向第一个多项式结点和第二个多项式结点以及当前结点的位置，如果指数相同则将对对应系数相加，并同时当前结构体指针向下移动一个结点，依次完成多项式的相加。

实验成绩：

日期：\_\_\_\_年\_\_\_\_月\_\_\_\_日

### 3.实验三

#### 河南理工大学教学上机实验报告

上机时间 2021 年 10 月 19 日

实验题目：

堆栈和队列基本操作实验

实验目的和要求：

实验内容：

- 1、从键盘上输入一个表达式，试编写算法实现对该表达式的求值。
- 2、借助循环队列实现舞伴问题。

**实验要求：**掌握链堆栈和循环队列的相关算法，并能使用其解决实际问题。

## 实验过程:

## 实验 1:

```

11  if (!in(ch)) {
        Push(OPND, ch);
        cin >> ch;
    } //ch不是运算符则进OPND栈
else
    switch (Precede(GetTop(OPTR), ch)) //比较OPTR的栈顶元素和ch的优先级
    {
        case '<':
            Push(OPTR, ch);
            cin >> ch; //当前字符ch压入OPTR栈，读入下一字符ch
            break;
        case '=':
            Pop(OPTR, theta); //_____ (1) _____; //弹出OPTR栈顶的运算符
            Pop(OPND, a); //_____ (2) _____;
            Pop(OPND, b); //_____ (3) _____; //弹出OPND栈顶的两个运算数
            Push(OPND, Operate(a, theta, b)); //_____ (4) _____; //将运算结果压入OPND栈
            break;
        case '>': //OPTR的栈顶元素是“(”且ch是“)”
            Pop(OPTR, x);
            cin >> ch; //弹出OPTR栈顶的“(”，读入下一字符ch
            break;
    }
}

```

## 实验 2:

```

        EnQueue(Mdancers, p); //插入男队
    }
    cout << "The dancing partners are:" << endl;
    while (!QueueEmpty(Fdancers) && !QueueEmpty(Mdancers)) { //依次输出男女舞伴的姓名
        DeQueue(Fdancers, p); //_____ (1) _____; //女士出队
        cout << p.name << " "; //输出出队女士姓名
        EnQueue(Mdancers, p); //_____ (2) _____; //男士出队
        cout << p.name << endl; //输出出队男士姓名
    }
    if (!QueueEmpty(Fdancers)) { //女士队列非空，输出队头女士的姓名

}

int EnQueue(SqQueue &Q, Person e) { //插入元素e为Q的新的队尾元素
    if ((Q.rear + 1) % MAXQSIZE == Q.front) //尾指针在循环意义上加1后等于头指针，表明队满
        return ERROR;
    Q.base[Q.rear] = e; //新元素插入队尾
    Q.rear = (Q.rear + 1) % MAXQSIZE; //_____ (3) _____; //队尾指针加1
    return OK;
}

```

实验结果：

运行结果：

实验 1:

0-9以内的多项式计算

1. 计算

0. 退出

选择: 1

请输入要计算的表达式（操作数和结果都在0-9的范围内，以#结束）：

6\*9

#

计算结果为54

0-9以内的多项式计算

1. 计算

0. 退出

选择:

实验 2:

The dancing partners are:

鄢雪棱 鄢雪棱

高欣雅 高欣雅

张甜源 张甜源

许丹丹 许丹丹

冉小溪 冉小溪

周如意 周如意

胡思琪 胡思琪

蒋雅琪 蒋雅琪

The first woman to get a partner is: 金城武

请按任意键继续. . .

实验分析:

该程序要将多项式的相关信息 txt 文件与源程序文件放在同一目录下，才能读取到相关信息。其次本程序是采用链堆栈和循环队列的形式，对程序目的进行了功能实现。两个链栈，分别存储表达式中的 运算数 OPND 和 运算符 OPTR，根据运算符的优先级依次进行入栈和出栈从而实现表达式的计算

实验成绩：

日期：\_\_\_\_年\_\_\_\_月\_\_\_\_日

#### 4.实验四

### 河南理工大学教学上机实验报告

上机时间 2021 年 10 月 26 日

实验题目：

用定长存储方式存储字符串数据，并在该模式下实现简单模式匹配、KMP 模式匹配算法。

实验目的和要求：

实验内容：

用定长存储方式存储字符串数据，并在该模式下实现简单模式匹配、KMP 模式匹配算法。

**实验要求：**掌握模式匹配的相关算法，并能使用其解决实际问题。

实验过程：

实验：

BF 算法：

```
//算法4.1 BF算法
int Index(SString S, SString T, int pos)
{
    //返回模式T在主串S中第pos个字符之后第s一次出现的位置。若不存在，则返回值为0
    //其中，T非空， $1 \leq pos \leq StrLength(S)$ 
    int i = pos;
    int j = 1;
    while (i <= S[0] && j <= T[0])
    {
        if (S[i] == T[j]) { i++; j++; }
        else { i = i - j + 2; j = 1; }
    }
    if (j > T[0]) return i - T[0];
    else return 0;
} //Index
```

计算 next 数组：

```
void get_next(SString T, int next[])
{ //求模式串T的next函数值并存入数组next
    int i = 1, j = 0;
    next[1] = 0;
    while (i < T[0])
    {
        if (j == 0 || T[i] == T[j])
        {
            i++; j++; next[i] = j;
        }
        else j = next[j];
    }
} //get_next
```

KMP 算法：

```

int Index_KMP(SSString S, SString T, int pos, int next[])
{
    // 利用模式串T的next函数求T在主串S中第pos个字符之后的位置的KMP算法
    //其中, T非空,  $1 \leq \text{pos} \leq \text{StrLength}(S)$ 
    int i = pos, j = 1;
    while (i <= S[0] && j <= T[0])
    {
        if (j == 0 || S[i] == T[j])
        {
            i++; j++;
        }
        else j = next[j];
    }
    if (j > T[0]) return i - T[0];
    else return 0;
} //Index_KMP

```

实验结果:

运行结果:

```

int main()
{
    SString S;
    StrAssign(S, "aaabbaba");
    SString T;
    StrAssign(T, "abb");
    // BF算法测试
    cout << "主串和子串在第" << Index(S, T, 1) << "个字符处首次匹配\n";
    // KMP算法测试
    int* p = new int[T[0] + 1]; // 生成T的next数组
    get_next(T, p);
    cout << "主串和子串在第" << Index_KMP(S, T, 1, p) << "个字符处首次匹配\n";
    system("pause");
    return 0;
}

```

D:\IDE\_code\_practice\c\_practice(visualc++2010)\cpp\_project\Debug\cpp\_project.exe  
 主串和子串在第3个字符处首次匹配  
 主串和子串在第3个字符处首次匹配  
 请按任意键继续. . .

0 %

动窗口

名称

值



**实验分析：****对于 BF 算法：**

采用 SString 数据类型，数组第一个位置存放字符串的长度，采用 while 循环的方式，只要索引不超过字符串 S 和 T 的长度，就符合循环条件，如果 S 和 T 的这位字符相同，就继续往下进行，如果不相同，则 S 串指针回溯为  $i-j+2$  而 T 串指针回溯为 1，从头开始，一直进行下去，直到循环结束，如果 j 超过 T 的最大长度，则说明找到，如果没有超过则说明没有找到

**对于 next 的计算**

采用 SString 数据类型，采用 while 循环，只要指针不超过 T 串的最大长度，就符合循环条件，当 i 等于 1 时， $next[j]=0$ 。当 i 不等于 1 时，判断 T 串从 1 开始的值是否等于 T 串与 S 串不相等时候的值，如果相等，则  $next[i]=j$  如果不相等，则指针进行回溯，j 等于  $next[j]$

**KMP 算法：**

采用 next 数组的方式，对两个串依次进行对比，当两个串都没有比较到串尾的时候，继续往后比较，如果匹配成功，则指针均继续往下进行，如果匹配失败，则 T 串的指针回溯，此时他的指针等于  $next[j]$  的值，循环结束之后，如果 T 串指针长度大于 T 串的长度，则位置即为 S 串指针减去 T 串的长度，否则返回 0

**实验成绩：**

日期：\_\_\_\_年\_\_\_\_月\_\_\_\_日

## 5.实验五

### 河南理工大学教学上机实验报告

上机时间 2022 年 11 月 09 日

**实验题目：**

设计实现 Huffman 树的创建算法，并利用 Huffman 树进行编码和译码。

**实验目的和要求：**

**实验目的：**熟练掌握二叉树的基本操作操作，应用二叉树的结构解决实际问题，熟练掌握 Huffman 树和 Huffman

编码的构造方法。

**实验要求：** 1.在“Huffman 编码”基础上的实现译码操作如输入“10010”译出相应的字符。 2.选做：(1)增加统计原文中各字符权值的功能,(2)将原文进行编码。

**实验过程：**

**程序主要功能源代码：**

//算法 5.11 根据赫夫曼树求赫夫曼编码

```
#include<iostream>
```

```
#include<cstring>
```

```
using namespace std;
```

```
typedef struct
```

```
{
```

```
    int weight;
```

```
    int parent,lchild,rchild;
```

```
}HTNode,*HuffmanTree;
```

```
typedef char **HuffmanCode;
```

```
void Select(HuffmanTree HT,int len,int &s1,int &s2)
```

```
{
```

```
    int i,min1=INT_MAX,min2=INT_MAX;//先赋予最大值
```

```
    for(i=1;i<=len;i++)
```

```
    {
```

```
        if(HT[i].weight<min1&&HT[i].parent==0)
```

```
        {
```

```
            min1=HT[i].weight;
```

```
            s1=i;
```

```
        }
```

```
    }
```

```
    int temp=HT[s1].weight;//将原值存放起来，然后先赋予最大值，防止 s1 被重复选择
```

```
    HT[s1].weight=INT_MAX;
```

```
    for(i=1;i<=len;i++)
```

```
    {
```

```
        if(HT[i].weight<min2&&HT[i].parent==0)
```

```
        {
```

```
            min2=HT[i].weight;
```

```
            s2=i;
```

```
        }
```

```
    }
```

```
    HT[s1].weight=temp;//恢复原来的值
```

```
}
```

//用算法 5.10 构造赫夫曼树

```

void CreatHuffmanTree(HuffmanTree &HT,int n)
{
    //构造赫夫曼树 HT
    int m,s1,s2,i;
    if(n<=1) return;
    m=2*n-1;
    HT=new HTNode[m+1];      //0 号单元未用，所以需要动态分配 m+1 个单元，HT[m]表示根结点
    for(i=1;i<=m;++i)        //将 1~m 号单元中的双亲、左孩子，右孩子的下标都初始化为 0
        { HT[i].parent=0; HT[i].lchild=0; HT[i].rchild=0; }

    cout<<"请输入叶子结点的权值：\n";
    for(i=1;i<=n;++i)        //输入前 n 个单元中叶子结点的权值
        cin>>HT[i].weight;
    /*-----初始化工作结束，下面开始创建赫夫曼树-----*/
    for(i=n+1;i<=m;++i)
    { //通过 n-1 次的选择、删除、合并来创建赫夫曼树
        Select(HT,i-1,s1,s2);
        //在 HT[k](1≤k≤i-1)中选择两个其双亲域为 0 且权值最小的结点，
        // 并返回它们在 HT 中的序号 s1 和 s2
        HT[s1].parent=i;
        HT[s2].parent=i;
        //得到新结点 i，从森林中删除 s1，s2，将 s1 和 s2 的双亲域由 0 改为 i
        HT[i].lchild=s1;
        HT[i].rchild=s2;          //s1,s2 分别作为 i 的左右孩子
        HT[i].weight=HT[s1].weight+HT[s2].weight;    //i 的权值为左右孩子权值之和
    }
}

// CreatHuffmanTree

void CreatHuffmanCode(HuffmanTree HT,HuffmanCode &HC,int n)
{
    //从叶子到根逆向求每个字符的赫夫曼编码，存储在编码表 HC 中
    int i,start,c,f;
    HC=new char*[n+1];          //分配 n 个字符编码的头指针矢量
    char *cd=new char[n];        //分配临时存放编码的动态数组空间
    cd[n-1]='\0';               //编码结束符
    for(i=1;i<=n;++i)
    {
        //逐个字符求赫夫曼编码
        start=n-1;              //start 开始时指向最后，即编码结束符位置
        c=i;
        f=HT[i].parent;          //f 指向结点 c 的双亲结点
        while(f!=0)
        {
            //从叶子结点开始向上回溯，直到根结点
            //回溯一次 start 向前指一个位置
            --start;
        }
    }
}

```

```

        if(HT[f].lchild==c)
            cd[start]='0';           //结点 c 是 f 的左孩子，则生成代码 0
        else
            cd[start]='1';           //结点 c 是 f 的右孩子，则生成代码 1
        c=f;
        f=HT[f].parent;             //继续向上回溯
    }
    HC[i]=new char[n-start];         //求出第 i 个字符的编码
    strcpy(HC[i], &cd[start]);      //为第 i 个字符编码分配空间
    delete cd;                      //将求得的编码从临时空间 cd 复制到 HC 的当前行中
    delete cd;                      //释放临时空间
}                                   // CreatHuffmanCode

void show(HuffmanTree HT,HuffmanCode HC,int n)
{
    for(int i=1;i<=n;i++)
        cout<<HT[i].weight<<"编码为"<<HC[i]<<endl;
}

/*void decode(HuffmanTree &HT,char *ch,int n)//依次读入电文，根据哈夫曼树译码
{
    int i,j=0;
    char b[100];
    char endflag='#';    //电文结束标志取#
    i=2*n-1;             //从根结点开始往下搜索
    getchar();
    cout<<("输入发送的编码(以'#'为结束标志): ");
    gets(b);
    cout<<"译码后的字符为";
    while(b[j] != endflag)
    {
        //填写完整

    }
    cout<<endl;
}

//decode
*/

int main()
{
    HuffmanTree HT;
    HuffmanCode HC;
    int n;
    cout<<"请输入待编码的字符(叶子结点)的个数: \n";
    cin>>n;

```


```

    getchar();
    char *p=new char[n];//?
    printf("请输入待编码的字符(中间不加空格)");
    for(int i=1;i<=n;i++)
        cin>>p[i];

    CreatHuffmanTree(HT,n);
    CreatHuffmanCode(HT,HC,n);
    show(HT,HC,n);
    //decode(HT,p,n);
return 1;
}

```

### 实验结果：



Microsoft Visual Studio 调试控制台

请输入待编码的字符(叶子结点)的个数:  
5  
请输入待编码的字符(中间不加空格)abcde  
请输入叶子结点的权值:  
123  
456  
789  
22  
0  
123编码为001  
456编码为01  
789编码为1  
22编码为0001  
0编码为0000

D:\IDE\_code\_practice\c\_practice (vusual studio2019) \c语言\Solution1\Project1\Debug\Project1.exe (进程 31440) 已退出, 什  
码为 1。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口。...

### 实验分析：

首先构造两个结构体分别存储结点的字符及权值、哈夫曼编码值, 然后读取前  $n$  个结点的字符及权值, 建立哈夫曼树, 读取前  $n$  个结点的字符及权值, 建立哈夫曼树, 读入电文, 最后根据哈夫曼树译码。本次实验更好的了解了哈夫曼夫的构造与算法

### 实验成绩：

日期：\_\_\_\_年\_\_\_\_月\_\_\_\_日

## 6.实验六

### 河南理工大学教学上机实验报告

上机时间 2022 年 11 月 14 日

实验题目：

图结构基本算法的实现

实验目的和要求：

**实验目的：**掌握图的存储结构及深度优先、广度优先遍历算法,理解图在实际应用中的经典算法如：最小生成树算法、最短路径算法、拓扑排序算法等。

**实验内容：**实现邻接链表和逆邻接链表两种求顶点入度的算法，并在拓扑排序算法及求关键路径算法中应用。

实验过程：

**程序主要功能源代码：**

实验一：拓扑排序及关键路径：

```
int CreateUDG(ALGraph& G) {
    //创建有向图 G 的邻接表、逆邻接表
    int i, k;

    cout << "请输入总顶点数，总边数，以空格隔开:";
    cin >> G.vexnum >> G.arcnum;           //输入总顶点数，总边数
    cout << endl;

    cout << "输入点的名称，如 a" << endl;

    for (i = 0; i < G.vexnum; ++i) {        //输入各点，构造表头结点表
        cout << "请输入第" << (i + 1) << "个点的名称:";
        cin >> G.vertices[i].data;          //输入顶点值
        G.converse_vertices[i].data = G.vertices[i].data;
        //初始化表头结点的指针域为 NULL
        G.vertices[i].firstarc = NULL;
        G.converse_vertices[i].firstarc = NULL;
    } //for
    cout << endl;
```

```

cout << "输入边依附的顶点及其权值，如 a b 3" << endl;

for (k = 0; k < G.arcnum; ++k) {                                //输入各边，构造邻接表
    VerTexType v1, v2;
    int i, j, w;
    cout << "请输入第" << (k + 1) << "条边依附的顶点及其权值:";
    cin >> v1 >> v2 >> w;                                     //输入一条边依附的两个顶点
    i = LocateVex(G, v1);  j = LocateVex(G, v2);
    //确定 v1 和 v2 在 G 中位置，即顶点在 G.vertices 中的序号

    ArcNode* p1 = new ArcNode;                                //生成一个新的边结点*p1
    p1->adjvex = j;                                           //邻接点序号为 j
    p1->nextarc = G.vertices[i].firstarc;  G.vertices[i].firstarc = p1;
    p1->weight = w;
    //将新结点*p1 插入顶点 vi 的边表头部

    ArcNode* p2 = new ArcNode;                                //生成一个新的边结点*p1
    p2->adjvex = i;                                           //逆邻接点序号为 i
    p2->nextarc = G.converse_vertices[j].firstarc;  G.converse_vertices[j].firstarc = p2;
    p2->weight = w;
    //将新结点*p1 插入顶点 vi 的边表头部
} //for
return OK;
} //CreateUDG

void FindInDegree(ALGraph G) {
    //求出各顶点的入度存入数组 indegree 中
    int i, count;

    for (i = 0; i < G.vexnum; i++) {
        count = 0;
        ArcNode* p = G.converse_vertices[i].firstarc;
        if (p) {
            while (p) {
                p = p->nextarc;
                count++;
            }
        } //if
        indegree[i] = count;
    } //for
} //FindInDegree

/*void FindInDegree(ALGraph G)
{ // 正邻接表求顶点的入度
    int i;
    ArcNode *p;

```

```

for(i=0;i<G.vexnum;i++)
    indegree[i]=0; // 赋初值
for(i=0;i<G.vexnum;i++)
{
    p=G.vertices[i].firstarc;
    while(p)
    {
        indegree[p->adjvex]++;
        p=p->nextarc;
    }
}
}
*/

int TopologicalOrder(ALGraph G, int topo[]) {
    //有向图 G 采用邻接表存储结构
    //若 G 无回路，则生成 G 的一个拓扑序列 topo[]并返回 OK，否则 ERROR
    int i, m;
    FindInDegree(G); //求出各顶点的入度存入数组 indegree 中
    InitStack(S); //栈 S 初始化为空
    for (i = 0; i < G.vexnum; ++i)
        if (!indegree[i]) Push(S, i); //入度为 0 者进栈
    m = 0; //对输出顶点计数，初始为 0
    while (!StackEmpty(S)) { //栈 S 非空
        Pop(S, i); //将栈顶顶点 vi 出栈
        topo[m] = i; //将 vi 保存在拓扑序列数组 topo 中
        ++m; //对输出顶点计数
        ArcNode* p = G.vertices[i].firstarc; //p 指向 vi 的第一个邻接点
        while (p) {
            int k = p->adjvex; //vk 为 vi 的邻接点
            --indegree[k]; //vi 的每个邻接点的入度减 1
            if (indegree[k] == 0) Push(S, k); //若入度减为 0，则入栈
            p = p->nextarc; //p 指向顶点 vi 下一个邻接结点
        } //while
    } //while

    if (m < G.vexnum) return ERROR; //该有向图有回路
    else return OK;
} //TopologicalOrder

int CriticalPath(ALGraph G) {
    //G 为邻接表存储的有向网，输出 G 的各项关键活动
    int n, i, k, j, e, l;
    if (!TopologicalOrder(G, topo)) return ERROR;
    //调用拓扑排序算法，使拓扑序列保存在 topo 中，若调用失败，则存在有向环，返回 ERROR

```



```

n = G.vexnum; //n 为顶点个数
for (i = 0; i < n; i++) //给每个事件的最早发生时间置初值 0
    ve[i] = 0;

/*————按拓扑次序求每个事件的最早发生时间————*/
for (i = 0; i < n; i++) {
    k = topo[i]; //取得拓扑序列中的顶点序号 k
    ArcNode* p = G.vertices[k].firstarc; //p 指向 k 的第一个邻接顶点
    while (p != NULL) { //依次更新 k 的所有邻接顶点的最早发生时间
        j = p->adjvex; //j 为邻接顶点的序号
        if (ve[j] < ve[k] + p->weight) //更新顶点 j 的最早发生时间 ve[j]
            ve[j] = ve[k] + p->weight;
        p = p->nextarc; //p 指向 k 的下一个邻接顶点
    } //while
} //for

for (i = 0; i < n; i++) //给每个事件的最迟发生时间置初值 ve[n-1]
    vl[i] = ve[n - 1];

/*————按逆拓扑次序求每个事件的最迟发生时间————*/
for (i = n - 1; i >= 0; i--) {
    k = topo[i]; //取得拓扑序列中的顶点序号 k
    ArcNode* p = G.vertices[k].firstarc; //p 指向 k 的第一个邻接顶点
    while (p != NULL) { //根据 k 的邻接点，更新 k 的最迟发生时间
        j = p->adjvex; //j 为邻接顶点的序号
        if (vl[k] > vl[j] - p->weight) //更新顶点 k 的最迟发生时间 vl[k]
            vl[k] = vl[j] - p->weight;
        p = p->nextarc; //p 指向 k 的下一个邻接顶点
    } //while
} //for

/*————判断每一活动是否为关键活动————*/
cout << endl;
cout << "关键活动路径为:";
for (i = 0; i < n; i++) { //每次循环针对 vi 为活动开始点的所有活动
    ArcNode* p = G.vertices[i].firstarc; //p 指向 i 的第一个邻接顶点
    while (p != NULL) {
        j = p->adjvex; //j 为 i 的邻接顶点的序号
        e = ve[i];
        l = vl[j] - p->weight;
        if (e == l) //若为关键活动，则输出<vi, vj>
            cout << G.vertices[i].data << "-->" << G.vertices[j].data << " ";
        p = p->nextarc; //p 指向 i 的下一个邻接顶点
    } //while
}

```

```

    } //for
    return OK;
} //CriticalPath

int main() {
    cout << "*****算法 6.13 关键路径算法*****" << endl << endl;
    ALGraph G;
    CreateUDG(G);
    int* topo = new int[G.vexnum];

    cout << endl;
    cout << "有向图创建完成!" << endl << endl;

    if (!CriticalPath(G))
        cout << "网中存在环，无法进行拓扑排序！" << endl << endl;
    cout << endl;
    return OK;
} //main

```

实验二：Floyed 算法：

```

64 void ShortestPath_Floyed(ALGraph G) {
65     //用Floyd算法求有向网G中各对顶点i和j之间的最短路径
66     int i, j, k;
67     for (i = 0; i < G.vexnum; ++i) //各对结点之间初始已知路径及距离
68         for (j = 0; j < G.vexnum; ++j) {
69             D[i][j] = G.arcs[i][j];
70             if (D[i][j] < MaxInt && i != j) Path[i][j] = i; //如果i和j之间有弧，则将j的前驱置为
71             else Path[i][j] = -1; //如果i和j之间无弧，则将j的前驱置为-1
72         } //for
73     for (k = 0; k < G.vexnum; ++k)
74         for (i = 0; i < G.vexnum; ++i)
75             for (j = 0; j < G.vexnum; ++j)
76                 if (D[i][k] + D[k][j] < D[i][j]) { //从i经k到j的一条路径更短
77                     D[i][j] = D[i][k] + D[k][j];
78                     Path[i][j] = Path[i][k];
79                 } //if
80 } //ShortestPath_Floyed
81

```

实验三：Kruskal

```

94
95 void MiniSpanTree_Kruskal(AMGraph G) {
96     //无向网G以邻接矩阵形式存储，构造G的最小生成树T，输出T的各条边
97     int i, j, v1, v2, vs1, vs2;
98     Sort(G); //将数组Edge中的元素按权值从小到大排序
99     for (i = 0; i < G.vexnum; ++i) //辅助数组，表示各顶点自成一个连通分量
100         Vexset[i] = i;
101     for (i = 0; i < G.arcnum; ++i) {
102         //依次查看排好序的数组Edge中的边是否在同一连通分量上
103         v1 = LocateVex(G, Edge[i].Head); //v1为边的始点Head的下标
104         v2 = LocateVex(G, Edge[i].Tail); //v2为边的终点Tail的下标
105         vs1 = Vexset[v1]; //获取边Edge[i]的始点所在的连通分量vs1
106         vs2 = Vexset[v2]; //获取边Edge[i]的终点所在的连通分量vs2
107         if (vs1 != vs2) { //边的两个顶点分属不同的连通分量
108             cout << Edge[i].Head << "-->" << Edge[i].Tail << endl; //输出此边
109             for (j = 0; j < G.vexnum; ++j) //合并vs1和vs2两个分量，即两个集合统一编号
110                 if (Vexset[j] == vs2) Vexset[j] = vs1; //集合编号为vs2的都改为vs1
111         } //if
112     } //for
113 } //MiniSpanTree_Kruskal
114

```

#### 实验四：prim

```

76 void MiniSpanTree_Prim(AMGraph G, VerTexType u) {
77     //无向网G以邻接矩阵形式存储，从顶点u出发构造G的最小生成树T，输出T的各条边
78     int k, j, i;
79     VerTexType u0, v0;
80     k = LocateVex(G, u); //k为顶点u的下标
81     for (j = 0; j < G.vexnum; ++j) { //对V-U的每一个顶点vi，初始化closedge[i]
82         if (j != k) {
83             closedge[j].adjvex = u;
84             closedge[j].lowcost = G.arcs[k][j]; //{adjvex, lowcost}
85         } //if
86     } //for
87     closedge[k].lowcost = 0; //初始，U = {u}
88     for (i = 1; i < G.vexnum; ++i) { //选择其余n-1个顶点，生成n-1条边(n= G.v
89         k = Min(G); //求出T的下一个结点：第k个顶点，closedge[k]中存有当前最小边
90         u0 = closedge[k].adjvex; //u0为最小边的一个顶点，u0 ∈ U
91         v0 = G.vexs[k]; //v0为最小边的另一个顶点，v0 ∈ V-U
92         cout << "边 " << u0 << "-->" << v0 << endl; //输出当前的最小边(u0, v0)
93         closedge[k].lowcost = 0; //第k个顶点并入U集
94         for (j = 0; j < G.vexnum; ++j) {
95             if (G.arcs[k][j] < closedge[j].lowcost) { //新顶点并入U后重新选择最小边
96                 closedge[j].adjvex = G.vexs[k];
97                 closedge[j].lowcost = G.arcs[k][j];
98             } //if
99         } //for
100     } //for
101 } //MiniSpanTree_Prim

```

实验结果：

实验一：拓扑排序及关键路径：

```

        p = p->nextarc;          //p指向i的下一个邻接顶点
    } //while
    //for
    return OK;
} //CriticalPath

int main() {
    cout << "*****算法6.13 关键路径算法*****";
    ALGraph G;
    CreateUDG(G);
    int* topo = new int[G.vexnum];

    cout << endl;
    cout << "有向图创建完成!" << endl << endl;

    if (!CriticalPath(G))
        cout << "网中存在环, 无法进行拓扑排序!" << endl;
    cout << endl;
    return OK;
} //main

```

Microsoft Visual Studio 调试控制台

\*\*\*\*\*算法6.13 关键路径算法\*\*\*\*\*

请输入总顶点数, 总边数, 以空格隔开: 4 3

输入点的名称, 如a

请输入第1个点的名称: a

请输入第2个点的名称: b

请输入第3个点的名称: c

请输入第4个点的名称: d

输入边依附的顶点及其权值, 如a b 3

请输入第1条边依附的顶点及其权值: a b 2

请输入第2条边依附的顶点及其权值: b c 2

请输入第3条边依附的顶点及其权值: c d 3

有向图创建完成!

关键活动路径为: a-->b b-->c c-->d

## 实验二:

### 最短路径 Floyd

Microsoft Visual Studio 调试控制台

\*\*\*\*\*算法6.11 弗洛伊德算法\*\*\*\*\*

请输入总顶点数, 总边数, 以空格隔开: 4 4

输入点的名称, 如a

请输入第1个点的名称: a

请输入第2个点的名称: b

请输入第3个点的名称: c

请输入第4个点的名称: d

输入边依附的顶点及权值, 如a b 3

请输入第1条边依附的顶点及权值: a b 1

请输入第2条边依附的顶点及权值: b c 2

请输入第3条边依附的顶点及权值: c d 2

请输入第4条边依附的顶点及权值: a d 4

有向图G创建完成!

请依次输入路径的起点与终点的名称: a d

a-->d

最短路径的长度为: 4

D:\IDE\_code\_practice\c\_practice (visual studio2019) \c语言\Solution1\Project1\Debug\Project1.exe (进程 24852) 已退出, 码为 0。

要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。

按任意键关闭此窗口...

## 实验三: kruskal:

\*\*\*\*\*算法6.9 克鲁斯卡尔算法\*\*\*\*\*

请输入总顶点数，总边数，以空格隔开:4 3

输入点的名称，如a  
 请输入第1个点的名称:a  
 请输入第2个点的名称:b  
 请输入第3个点的名称:c  
 请输入第4个点的名称:d

输入边依附的顶点及权值，如a b 6  
 请输入第1条边依附的顶点及权值:a b 1  
 请输入第2条边依附的顶点及权值:b c 2  
 请输入第3条边依附的顶点及权值: c d 2

\*\*\*\*\*无向图G创建完成!\*\*\*\*\*

a-->b  
 b-->c  
 c-->d

D:\IDE\_code\_practice\c\_practice (vvisual studio2019) \c语言\Solution1\Project1\Debug\Project1.exe (进程 31120)已退出，代码为 0。

要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。

按任意键关闭此窗口...

#### 实验四：Prim:

\*\*\*\*\*算法6.8 普里姆算法\*\*\*\*\*

请输入总顶点数，总边数，以空格隔开:4 3

输入点的名称，如a  
 请输入第1个点的名称:a  
 请输入第2个点的名称:b  
 请输入第3个点的名称:c  
 请输入第4个点的名称:d

输入边依附的顶点及权值，如a b 5  
 请输入第1条边依附的顶点及权值:a b 1  
 请输入第2条边依附的顶点及权值:b c 2  
 请输入第3条边依附的顶点及权值:c d 3

无向图G创建完成!

\*\*\*\*\*利用普里姆算法构造最小生成树结果:\*\*\*\*\*

边 a-->b  
 边 b-->c  
 边 c-->d

D:\IDE\_code\_practice\c\_practice (vvisual studio2019) \c语言\Solution1\Project1\Debug\Project1.exe (进程 32484)已退出，代码为 0。

要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。

按任意键关闭此窗口...

#### 实验分析:

##### 1. 拓扑排序及关键路径

###### (1) 拓扑排序

求出各顶点的入度 4 存入数组 indegree[i] 使入度为 0 的顶点入栈，若不空则一直循环让顶点出栈且保存在拓扑数组里，让每个顶点连接点的入度减一，变成 0 了就入栈。如果输出的数量小于顶点个数，说明有环。

###### (2) 关键路径

对图中顶点排序，按拓扑序列求出每个时间最早发生的时间 ve[i]，按逆拓扑序列求出每个事件发生的最迟时间 vl[i]。如果 ve[i]==vl[i] 的活动 a[i]就是关键路径

##### 2. 最短路径 Floyd

算法实现：先初始化所有最短路径长度， $D[i][j]=G.arcs[i][j]$  然后进行  $n$  次比较和更新。

每次找  $i, j$  之间最短的中间点，寻找最小的路径

### 3. 最小生成树 Kruskal

算法实现：将所有边的信息按权值从小到大，每次循环从排好序的边找出一条，判断他们是不是再一个 Vexset 中在就舍去大的边，若不是，合并这两个连通分量。

### 4. 最小生成树 Prim

算法实现：初始化所有顶点，对其余的每一个顶点  $v$ ，将  $closedge[j]$  均初始化到  $u$  的边信息，循环  $n-1$  次，每次选出最小的  $closedge[k]$  输出此边，将  $k$  加入，更新剩余的最小边信息，再次循环。

实验成绩：

日期：\_\_\_\_\_年\_\_\_\_月\_\_\_\_日

## 7.实验七

### 河南理工大学教学上机实验报告

上机时间 2022 年 11 月 21 日

实验题目：

- 1.编写程序实现有序表二分查找的递归算法；
- 2.用非递归的方法实现二叉排序树上的查找（并求出查找成功时关键字所在的层次）。

实验目的和要求：

**实验目的：** 1.掌握顺序表的查找方法，尤其是二分查找方法。 2.掌握二叉排序树的建立及查找过程，理解二叉排序树查找过程及插入和删除算法。

**实验要求：** 1.编写程序实现有序表二分查找的递归算法； 2.用非递归的方法实现二叉排序树上的查找（并求出查找成功时关键字所在的层次）。

实验过程：

程序主要功能源代码：

### 实验一：实现有序表二分查找的递归算法

### 核心源代码：

```
int Search_Bin(SSTable ST, int key, int low, int high) {  
    // 递归折半查找  
  
    int mid;  
    if (low <= high)  
    {  
        mid = (low + high) / 2;  
        if (key == ST.R[mid].key) return mid;  
        else if (key < ST.R[mid].key)  
        {  
            high = mid - 1;  
            Search_Bin(ST, key, low, high);  
        }  
        else  
        {  
            low = mid + 1;  
            Search_Bin(ST, key, low, high);  
        }  
    }  
}  
} // Search_Bin
```

## 实验二：

## 非递归的方法实现二叉排序树上的查找

### 核心源代码：

//算法 7.4 二叉排序树的递归查找

```
BSTree SearchBST(BSTree T,int key) {
```

```
//在根指针 T 所指二叉排序树中递归地查找某关键字等于 key 的数据元素
```

//若查找成功, 则返回指向该数据元素结点的指针, 否则返回空指针

```
if(!T|| key==T->data.key) return T; //查找结束
```

```
else if (key<T->data.key) return SearchBST(T->lchild,key); //在左子树中继续查找
```

```
else return SearchBST(T->rchild,key);           //在右子树中继续查找
```

```
} // SearchBST
```

//非递归实现查找成功时返回指向该数据元素结点的指针和所在的层次

```
BSTree Searchlev(BSTree T,char key,int &lev) {
```

```
    //统计查找次数
```

```
    BSTree p = T; //p 为二叉树中工作指针
```

```
    lev++;
```

```
    while(T)
```

```
    {
```

```
        // cout<<lev;
```

```
        if(key==T->data.key) return T;
```

```
        if(key<T->data.key){
```

```
            T=T->lchild; //在左子树查找
```

```
            lev++;
```

```
        } else{
```

```
            T=T->rchild; //在右子树查找
```

```
            lev++;
```

```
        }
```

```
        // cout<<lev;
```

```
    }
```

```
    // cout<<"end";
```

```
    // return T;
```



```
return NULL;
```

```
}// Searchlev
```

//算法 7.5 二叉排序树的插入

```
void InsertBST(BSTree &T,ElemType e ) {
```

```
    //当二叉排序树 T 中不存在关键字等于 e.key 的数据元素时， 则插入该元素
```

```
    if(!T) {                                //找到插入位置， 递归结束
```

```
        BSTree S = new BSTNode;              //生成新结点*S
```

```
        S->data = e;                          //新结点*S 的数据域置为 e
```

```
        S->lchild = S->rchild = NULL; //新结点*S 作为叶子结点
```

```
        T = S;                               //把新结点*S 链接到已找到的插入位置
```

```
    }
```

```
    else if (e.key< T->data.key)
```

```
        InsertBST(T->lchild, e );            //将*S 插入左子树
```

```
    else if (e.key> T->data.key)
```

```
        InsertBST(T->rchild, e);             //将*S 插入右子树
```

```
}// InsertBST
```

## //算法 7.6 二叉排序树的创建

```
void CreateBST(BSTree &T){
```

```
    //依次读入一个关键字为 key 的结点，将此结点插入二叉排序树 T 中
```

```
    T=NULL;
```

```
    ElemType e;
```

```
    cin>>e.key;        //???
```

```
    while(e.key!=ENDFLAG){    //ENDFLAG 为自定义常量，作为输入结束标志
```

```
        InsertBST(T, e);        //将此结点插入二叉排序树 T 中
```

```
        cin>>e.key;        //???
```

```
    }//while
```

```
}//CreatBST
```

```
void DeleteBST(BSTree &T,int key){
```

```
    //从二叉排序树 T 中删除关键字等于 key 的结点
```

```
    BSTree p=T;BSTree f=NULL;        //初始化
```

```
    BSTree q;
```

```
    BSTree s;
```

```
    /*-----下面的 while 循环从根开始查找关键字等于 key 的结点*p-----*/
```

```
    while(p){
```

```
        if (p->data.key == key) break;        //找到关键字等于 key 的结点*p，结束循环
```

```
        f=p;        //f 为*p 的双亲结点
```

```
        if (p->data.key> key)  p=p->lchild;    //在*p 的左子树中继续查找
```

```

else p=p->rchild;                //在*p 的右子树中继续查找

} //while

if(!p) return;                  //找不到被删结点则返回

/*—考虑三种情况实现 p 所指子树内部的处理： *p 左右子树均不空、无右子树、无左子树—*/

if ((p->lchild)&& (p->rchild)) {    //被删结点*p 左右子树均不空

    q = p;

    s = p->lchild;

    while (s->rchild)              //在*p 的左子树中继续查找其前驱结点，即最
    右下结点

        {q = s; s = s->rchild;}    //向右到尽头

    p->data = s->data;              //s 指向被删结点的“前驱”

    if(q!=p){

        q->rchild = s->lchild;      //重接*q 的右子树

    }

    else q->lchild = s->lchild;      //重接*q 的左子树

    delete s;

} //if

else{

    if(!p->rchild) {                //被删结点*p 无右子树， 只需重接其左子树

        q = p; p = p->lchild;

    } //else if

    else if(!p->lchild) {           //被删结点*p 无左子树， 只需重接其右子树

```

```

        q = p; p = p->rchild;

    }//else if

    /*—————将 p 所指的子树挂接到其双亲结点*f 相应的位置—————*/

    if(!f) T=p;                //被删结点为根结点

    else if (q==f->lchild) f->lchild = p;    //挂接到*f 的左子树位置

    else f->rchild = p;          //挂接到*f 的右子树位置

    delete q;

}

} //DeleteBST

```

//算法 7.7 二叉排序树的删除

//中序遍历

```
void InOrderTraverse(BSTree &T)
```

```

{
    if(T)
    {
        InOrderTraverse(T->lchild);

        cout<<T->data.key<<" ";

        InOrderTraverse(T->rchild);
    }
}

```

```
char predt=0;

int judgeBST(BSTree T)
{
    int b1, b2;

    if(T == NULL)
    {
        return 1;
    }
    else
    {
        b1 = judgeBST(T->lchild);

        if(b1 == 0 || predt > T->data.key) return 0;

        predt = T->data.key;

        b2 = judgeBST(T->rchild);

        return b2;
    }
}
```

**实验结果：**

**实验一结果：**

请依次输入两个关键字（用回车隔开）： 30


45

找到30位置为30

找到45位置为45

D:\IDE\_code\_practice\c\_practice (visual studio2019) \c语言\Sc  
码为 0.

**实验二结果：**

 D:\IDE\_code\_practice\c\_practice(visualc++2010)\cpp\_project\Debug\cpp\_projec

请输入若干整数，用空格分开，以-1结束输入

5 8 9 4 7 3 6 -1

当前有序二叉树中序遍历结果为

3 4 5 6 7 8 9

请输入待查找关键字（整数）

5

找到关键字5在1层

请输入待删除的关键字

3

当前有序二叉树中序遍历结果为

4 5 6 7 8 9 是一棵排序树

请按任意键继续. . .

**实验分析：****二分查找：**

主要使用循环或递归在每次比较后将查找空间划分为两半，依次进行查找。

**7.4-7.7：**

1. 从表的一端开始逐个将记录的关键字和给定 K 值进行比较，若某个记录的关键字和给定 K 值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败
2. 根结点就是第一次进行比较的中间位置的记录
3. 排在中间位置前面的作为左子树的结点，排在中间位置后面的作为右子树的结点
4. 在 BST 树中插入一个新结点 x 时，若 BST 树为空，则新结点 x 为插入后 BST 树的根结点；否则，将结点 x 的关键字与根结点 T 的关键字进行比较，若相等，不需要插入，若  $x.key < T \rightarrow key$  :结点 x 插入到 T 的左子树中，若  $x.key > T \rightarrow key$  :结点 x 插入到 T 的右子树中

**实验成绩：**

日期：\_\_\_\_年\_\_\_\_月\_\_\_\_日

## 8.实验八

### 河南理工大学教学上机实验报告

上机时间 2022 年 11 月 28 日

#### 实验题目：

1.对所讲过算法深入理解，应用随机函数和时间函数比较各种排序的运行时间。

2.实现双向冒泡排序(相邻两趟排序向相反方向冒泡)

#### 实验目的和要求：

熟悉多种排序算法，理解每种排序算法思想，掌握排序算法的基本设计方法，掌握排序算法时间复杂度和空间复杂度的分析方法。

#### 实验过程：

##### 程序主要功能源代码：

##### 1.直接插入排序

```
//-----直接插入排序-----  
void InsertSort(KeyType R[],int n)//传入一个关键字数组，及其关键字个数  
{  
    int i,j;  
    KeyType tmp;  
    for (i=1;i<n;i++)//依次将各个关键字插入到序列中去  
    {  
        tmp=R[i];  
        j=i-1;           //从右向左在有序区 R[0..i-1]中找 R[i]的插入位置  
        while (j>=0 && tmp<R[j])  
        /*都是在一个数组（KeyType）里面进行操作的，已经排好的序列是从小到大排序的，所以再  
        向其中插入的话，是从右往左比较的，只要遇见比他的大的数据，就继续往里面走就行。遇到比他还  
        小的话，就可以停了*/  
        {  
            R[j+1]=R[j];    //将关键字大于 R[i]的元素后移  
            j--;  
        }  
        R[j+1]=tmp;        //在 j+1 处插入 R[i]  
    }  
}
```

## 2.折半插入排序

```
//-----折半插入排序-----
void InsertSort1(KeyType R[],int n)
{
    int i,j,low,high,mid;
    int tmp;
    for (i=1;i<n;i++)
    {
        tmp=R[i];                //将 R[i]保存到 tmp 中
        low=0;high=i-1;
        while (low<=high)        //在 R[low..high]中折半查找有序插入的位置
        {
            mid=(low+high)/2;    //取中间位置
            if (tmp<R[mid])
                high=mid-1;      //插入点在左半区
            else
                low=mid+1;        //插入点在右半区
        }
        for (j=i-1;j>=high+1;j--) //元素后移
            R[j+1]=R[j];
        R[high+1]=tmp;           //插入
    }
}
```

## 3.希尔排序

```
//-----希尔排序算法-----
void ShellSort(KeyType R[],int n)
{
    int i,j,gap;
    KeyType tmp;
    gap=n/2;                //增量置初值
    while (gap>0)
    {
        for (i=gap;i<n;i++) //对所有相隔 gap 位置的所有元素组采用直接插入排序
        {
            tmp=R[i];
            j=i-gap;
            while (j>=0 && tmp<R[j])//对相隔 gap 位置的元素组进行排序
            {
                R[j+gap]=R[j];
                j=j-gap;
            }
            R[j+gap]=tmp;
        }
        gap=gap/2; //减小增量
    }
}
```

## 4.冒泡排序

```
//-----冒泡排序算法-----
```



```

void BubbleSort(KeyType R[],int n)
{
    int i,j;
    bool exchange;
    KeyType tmp;
    for (i=0;i<n-1;i++)
    {
        exchange=false;
        for (j=0;j<n-i-1;j++)    //比较,找出最大关键字的元素
            if (R[j]>R[j+1])
            {
                tmp=R[j];    //R[j]与 R[j+1]进行交换,将最大关键字元素前移
                R[j]=R[j+1];
                R[j+1]=tmp;
                exchange=true;
            }
        if (!exchange)    //本趟没有发生交换,中途结束算法
            return;
    }
}

```

## 5.快速排序

```

//-----快速排序算法-----
void QuickSort(KeyType R[],int s,int t)
{
    int i=s,j=t;
    KeyType tmp;
    if (s<t)    //区间内至少存在两个元素的情况
    {
        tmp=R[s];    //用区间的第 1 个元素作为基准
        while (i!=j)    //从区间两端交替向中间扫描,直至 i=j 为止
        {
            while (j>i && R[j]>=tmp)
                j--;    //从右向左扫描,找第 1 个小于 tmp 的 R[j]
            R[i]=R[j];    //找到这样的 R[j],R[i]和 R[j]交换
            while (i<j && R[i]<=tmp)
                i++;    //从左向右扫描,找第 1 个大于 tmp 的元素 R[i]
            R[j]=R[i];    //找到这样的 R[i],R[i]和 R[j]交换
        }
        R[i]=tmp;
        QuickSort(R,s,i-1); //对左区间递归排序
        QuickSort(R,i+1,t); //对右区间递归排序
    }
}

```

## 6.直接选择排序

```

//-----直接选择排序

```

```

void SelectSort(KeyType R[],int n)
{
    int i,j,k;
    KeyType tmp;
    for (i=0;i<n-1;i++)           //做第 i 趟排序
    {
        k=i;
        for (j=i+1;j<n;j++)       //在当前无序区 R[i..n-1]中选 key 最小的 R[k]
            if (R[j]<R[k])
                k=j;              //k 记下目前找到的最小关键字所在的位置
        if (k!=i)                 //交换 R[i]和 R[k]
        {
            tmp=R[i];
            R[i]=R[k];
            R[k]=tmp;
        }
    }
}

```

## 7.堆排序算法

```

//-----堆排序算法-----
void sift(KeyType R[],int low,int high)
{
    int i=low,j=2*i;              //R[j]是 R[i]的左孩子
    KeyType tmp=R[i];
    while (j<=high)
    {
        if (j<high && R[j]<R[j+1]) //若右孩子较大,把 j 指向右孩子
            j++;
        if (tmp<R[j])
        {
            R[i]=R[j];           //将 R[j]调整到双亲节点位置上
            i=j;                  //修改 i 和 j 值,以便继续向下筛选
            j=2*i;
        }
        else break;              //筛选结束
    }
    R[i]=tmp;                     //被筛选节点的值放入最终位置
}

```

## 8.二路归并排序算法

```

//-----二路归并排序算法-----
void Merge(KeyType R[],int low,int mid,int high)
{
    KeyType *R1;
    int i=low,j=mid+1,k=0; //k 是 R1 的下标,i、j 分别为第 1、2 段的下标
    R1=(KeyType *)malloc((high-low+1)*sizeof(KeyType)); //动态分配空间
    while (i<=mid && j<=high) //在第 1 段和第 2 段均未扫描完时循环
    {
        if (R[i]<=R[j]) //将第 1 段中的元素放入 R1 中
        {
            R1[k]=R[i];
            i++;k++;
        }
    }
}

```

```

    else //将第 2 段中的元素放入 R1 中
    {
        R1[k]=R[j];
        j++;k++;
    }
    while (i<=mid) //将第 1 段余下部分复制到 R1
    {
        R1[k]=R[i];
        i++;k++;
    }
    while (j<=high) //将第 2 段余下部分复制到 R1
    {
        R1[k]=R[j];
        j++;k++;
    }
    for (k=0,i=low;i<=high;k++,i++) //将 R1 复制回 R 中
        R[i]=R1[k];
    free(R1);
}

```

#### 实验结果：

```

D:\IDE_code_practice\vs_code\c++_files>cd "d:\IDE_code_practice
_practice\vs_code\c++_files\"code
随机产生50000个1-99的正整数,各种排序方法的比较
-----
排序方法          用时          结果验证
-----
直接插入排序      0秒820毫秒     正确
折半插入排序      0秒690毫秒     正确
希尔排序          0秒5毫秒       正确
冒泡排序          3秒867毫秒     正确
快速排序          0秒11毫秒      正确
直接选择排序      1秒243毫秒     正确
堆 排 序          0秒5毫秒       正确
二路归并排序      0秒7毫秒       正确
-----

D:\IDE_code_practice\vs_code\c++_files>

```

#### 实验分析：

通过随机产生的数字，对各大排序算法的时间进行测试，可以发现希尔排序和堆排序、快速排序以及二路归并排序算法比较优越，其次是折半插入排序、直接插入排序、直接选择排序以及冒泡排序。

而对于空间复杂度来说，堆排序为  $O(n^2)$ ，归并排序为  $O(n\log n)$ ，这三者虽然时间复杂度较低，但空间复杂度不如其他算法。

实验成绩：

日期：\_\_\_\_年\_\_月\_\_日