
实验课程	数据结构实验	成绩	
实验项目	实验二　二叉树的操作与实现	指导老师	

一、实验目的：

1、领会二叉链存储结构和掌握二叉树中的各种基本运算算法设计；

2、领会线索二叉树的构造过程以及构造二叉树的算法设计；

3、领会哈夫曼树的构造过程以及哈夫曼编码的生成过程；

4、掌握二叉树遍历算法的应用，熟练使用先序、中序、后序 3 种递归遍历算法

二、使用仪器、器材

微机一台

操作系统：WinXP

编程软件：C/C++编程软件

三、实验内容及原理

1、教材 P247 实验题 1：实现二叉树的各种基本运算的算法

编写一个程序 btree.cpp，实现二叉树的基本运算，并在此基础上设计一个程序 exp7-1.cpp 完成以下功能。

（1）由图 7.33 所示的二叉树创建对应的二叉链存储结构 b，该二叉树的括号表示串为“A(B(D,E(H(J,K(L,M(,N))))),C(F,G(,I)))”。

（2）输出二叉树 b。

（3）输出‘H’结点的左、右孩子结点值。

（4）输出二叉树 b 的高度。

（5）释放二叉树 b。

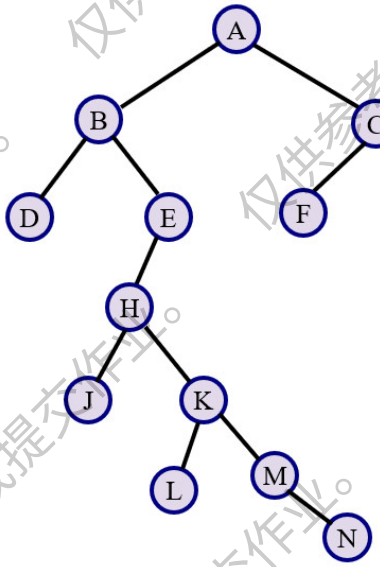


图 7.33 一棵二叉树

2、教材 P248 实验题 3：由遍历序列构造二叉树

编写一个程序 exp7-3.cpp，实现由先序序列和中序序列以及由中序序列和后序序列构造一棵二叉树的功能（二叉树种的每个结点值为单个字符），要求以括号表示和凹入表示法输出该二叉树，并用先序遍历序列“ABDEHJKLMNCFGI”和中序遍历序列“DBJHLKMNEAFCGI”以及由中序遍历序列“DBJHLKMNEAFCGI”和后序遍历序列“DJLNMKHEBFIGCA”进行验证。

3、教材 P248 实验题 5：构造哈夫曼树生成哈夫曼编码

编写一个程序 exp7-5.cpp，构造一棵哈夫曼树，输出对应的哈夫曼编码和平均查找长度，并对下表（表 7.8）所示的数据进行验证。

表 7.8 单词及出现的频度

单词	The	of	a	to	and	in	that	he	is	at	on	for	His	are	be
频度	1192	677	541	518	462	450	242	195	190	181	174	157	138	124	123

4、教材 P248 实验题 8：简单算术表达式二叉树的构建和求值

编写一个程序 exp7-8.cpp，先用二叉树来表示一个简单算术表达式，树的每一个结点包括一个运算符或运算数。在简单算术表达式中只包含+、-、*、/和一位正整数且格式正确（不包括括号），并且要按照先乘除后加减的原则构造二叉树，图 7.34 所示为“1+2*3-4/5”代数表达式对应的二叉树，然后由对应的二叉树计算该表达式的值。

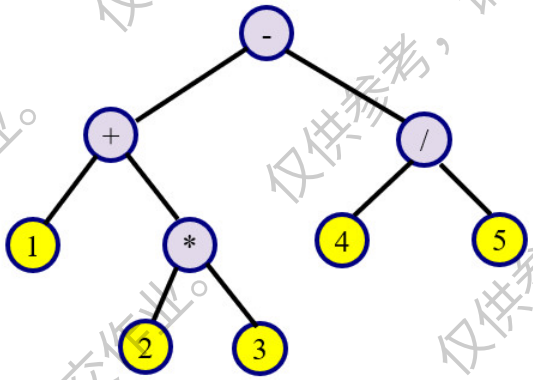


图 7.34 二叉树表示简单算术表达式

5、教材 P249 实验题 9：用二叉树表示家谱关系并实现各种查找功能

编写一个程序 exp7-9.cpp，采用一棵二叉树表示一个家谱关系（由若干家谱记录构成，每个家谱记录由父亲、母亲和儿子的姓名构成，其中姓名是关键字），要求程序具有以下功能。

（1） 文件操作功能：家谱记录的输入，家谱记录的输出，清除全部文件记录和将家谱记录存盘。要求在输入家谱记录时按从祖先到子孙的顺序输入，第一个家谱记录的父亲域为所有人的祖先。

（2） 家谱操作功能：用括号表示法输出家谱二叉树，查找某人的所有儿子，查找某人的所有祖先（这里的祖先是指所设计的二叉树结构中某结点的所有祖先结点）。

四、实验过程原始数据记录

第 1 题：

```
//-------------------------btree.h-----------------------
#pragma once

#include <iostream>

// 数据域元素类型
#define ElemType char
// 用广义表创建二叉树最大高度
#define MaxHeight 10

// 二叉树节点定义
typedef struct BiTNode
{
    ElemType data;                  // 数据域
```

```
    struct BiTNode* lchild, * rchild;  // 左、右孩子指针
}BiTNode, * BiTree;

// 二叉树操作
void InitBiTree(BiTree& T);     // 初始化二叉树
void DestroyBiTree(BiTree& T); // 销毁二叉树
void PrintBiTree(BiTree T);     // 打印二叉树
void Visit(BiTNode*);           // 访问（打印）节点值
int BiTreeHeight(BiTree T);     // 获取二叉树高度
bool FindTNode(BiTree T, ElemType e);           // 在树中查找
void CreateBiTree(const char str[], BiTree& T); // 创建二叉树

//-------------------------btree.cpp-----------------------
#include <iostream>
#include "btree.h"
using namespace std;

// 初始化二叉树
void InitBiTree(BiTree& T)
{
    T = new BiTNode;
    T->lchild = NULL;
    T->rchild = NULL;
}

// 销毁二叉树
void DestroyBiTree(BiTree& T)
{
    // 本算法递归直到叶子节点
    // 然后删除叶子节点返回父节点

    if (!T)  // 递归终止条件
        return;
    // 有孩子才继续递归深入，没有代表目前节点为叶子，直接释放
    if (T->lchild || T->rchild)
    {
        DestroyBiTree(T->lchild);
        DestroyBiTree(T->rchild);
    }
    delete T;
    T = NULL;
}
```

```
// 在树中查找
bool FindTNode(BiTree T, ElemType e)
{
    if (!T)  // 递归终止条件
        return false;

    // 如果找到啦，就输出左右孩子然后返回真
    if (T->data == e)
    {
        cout << "    " << e << "节点的";
        if (!T->lchild)
            cout << "左孩子为空";
        else
            cout << "左孩子为 " << T->lchild->data;
        if (!T->rchild)
            cout << " , 右孩子为空";
        else
            cout << " , 右孩子为 " << T->rchild->data;
        return true;
    }
    // 如果没有找到，看看这个节点有没有孩子，只要有一个孩子就去它孩子里边找找看
    else if (T->lchild || T->rchild)
    {
        // 如果在左孩子里找到啦就返回true
        if (FindTNode(T->lchild, e))
            return true;
        // 如果在右孩子里找到啦就返回true
        else if (FindTNode(T->rchild, e))
            return true;
    }
    // 如果很不幸，在这个节点的左右孩子里都找不到~~只能返回失败了
    else
        return false;
}

// 获取二叉树高度
int BiTreeHeight(BiTree T)
{
    // 空节点高度为0
    if (!T)
        return 0;
    // 非叶子节点才继续深入（减少递归层次）
    if (T->lchild || T->rchild)
    {
```

```
        int LHeight = BiTreeHeight(T->lchild);
        int RHeight = BiTreeHeight(T->rchild);
        return max(LHeight, RHeight) + 1;
    }
    // 叶子节点高度为1
    else
        return 1;
}

// 创建二叉树(非递归)
void CreateBiTree(const char str[], BiTree& T)
{
    int i = 0;                  // 用于遍历字符串的变量
    int flag = 2;               // tag=0,创建左孩子; tag=1，创建右孩子
    int len = strlen(str);      // 求字符串的长度
    int top = -1;               // 父栈栈顶位置
    BiTNode* s = T;             // s是用来开拓空间的指针
    BiTNode* ParentStack[MaxHeight]{ NULL };// 用于保存孩子节点的父节点地址

   T->data = str[i++];                      // 给根节点赋值

    for (; i < len; i++)
    {
        switch (str[i])
        {
        case '(':           // 扫描到（表示即将插入一个左孩子
            flag = 0;       // 设置标志
            ParentStack[++top] = s;    // 刚刚新建的孩子要当父亲，将其入栈
            break;
        case')':      // 扫描到）说明该父节点孩子已经建立完毕，父指针无用
            top--;    //   栈顶--
            break;
        case',':
            flag = 1;
            break;
        default: // 当扫描到字母
            // 创建孩子节点
            s = new BiTNode;
            s->data = str[i];
            s->lchild = NULL;
            s->rchild = NULL;

            if (flag == 0)// 插入左孩子
                ParentStack[top]->lchild = s;
```

```
            else            // 插入右孩子
                ParentStack[top]->rchild = s;
            break;
        }
    }
}

// 打印二叉树(先序/广义表)
void PrintBiTree(BiTree T)
{
    if (!T)  // 递归终止条件
        return;

    Visit(T);

    // 只有在左右孩子至少有一个存在时才访问
    if (T->lchild || T->rchild)
    {
        cout << "(";      // 要访问孩子时添加（
        PrintBiTree(T->lchild);   // 递归调用
        cout << ",";      // 访问完毕左孩子要访问右孩子时添加,
        PrintBiTree(T->rchild);   // 递归调用
        cout << ")";      // 访问完毕左右孩子添加）
    }
}

// 访问（打印）节点值
void Visit(BiTNode* node)
{
    if (!node)
        return;
    cout << node->data;
}

//-------------------------exp7-1.cpp----------------------
#include <iostream>
#include "btree.h"
using namespace std;

int main()
{
    BiTree b;
    InitBiTree(b);
    cout << "（1）由图7.33所示的二叉树创建对应的二叉链存储结构b" << endl;
```

```
    CreateBiTree("A(B(D,E(H(J,K(L,M(,N))))),C(F,G(,I)))", b);

    cout << endl << "（2）输出二叉树b" << endl;
    cout << "    ";
    PrintBiTree(b);

    cout << endl;
    cout << endl << "（3）输出‘H’结点的左、右孩子结点值" << endl;
    FindTNode(b, 'H');
    cout << endl;

    cout << endl << "（4）输出二叉树b的高度：" << BiTreeHeight(b) << endl;
    cout << endl << "（5）释放二叉树b" << endl;
    DestroyBiTree(b);

    return 0;
}
```

第 2 题：

```
#include <iostream>
using namespace std;


// 数据域元素类型
#define ElemType char

// 二叉树节点定义
typedef struct BiTNode
{
    ElemType data;                  // 数据域
    struct BiTNode* lchild, * rchild;  // 左、右孩子指针
}BiTNode, * BiTree;


// 二叉树操作
void InitBiTree(BiTree& T);         // 初始化二叉树
void DestroyBiTree(BiTree& T);      // 销毁二叉树
BiTree Pre_InCreateBT(const char Pre[], const char In[], int len);  // 先序、中序序列创建二叉树
BiTree In_PostCreateBT(const char Post[], const char In[], int len);// 后序、中序序列创建二叉树
void PrintBiTree(BiTree T);             // 打印二叉树（括号表示法）
void PrintBiTree(BiTree T, int depth); // 打印二叉树（凹入表示法）
```

```
int main()
{
    int len = strlen("ABDEHJKLMNCFGI");
    cout << "用先序“ABDEHJKLMNCFGI”和中序“DBJHLKMNEAFCGI”构造二叉树b1" << endl;
    BiTree b1 = Pre_InCreateBT("ABDEHJKLMNCFGI", "DBJHLKMNEAFCGI", len);

    cout << endl << "由中序“DBJHLKMNEAFCGI”和后序“DJLNMKHEBFIGCA“构造二叉树b2" << endl;
    BiTree b2 = In_PostCreateBT("ABDEHJKLMNCFGI", "DBJHLKMNEAFCGI", len);

    cout << endl << "括号表示法表示b1" << endl;
    PrintBiTree(b1);
    cout << endl;
    cout << "凹入表示法表示b1" << endl;
    PrintBiTree(b1, 0);

    cout << endl << "括号表示法表示b2" << endl;
    PrintBiTree(b2);
    cout << endl;
    cout << "凹入表示法表示b2" << endl;
    PrintBiTree(b2, 0);

    DestroyBiTree(b1);
    DestroyBiTree(b2);

    return 0;
}

// 初始化二叉树
void InitBiTree(BiTree& T)
{
    T = new BiTNode;
    T->lchild = NULL;
    T->rchild = NULL;
}

// 销毁二叉树
void DestroyBiTree(BiTree& T)
{
    // 本算法递归直到叶子节点
    // 然后删除叶子节点返回父节点

    if (!T)  // 递归终止条件
        return;
    // 有孩子才继续递归深入，没有代表目前节点为叶子，直接释放
```

```
    if (T->lchild || T->rchild)
    {
        DestroyBiTree(T->lchild);
        DestroyBiTree(T->rchild);
    }
    delete T;
    T = NULL;
}

// 先序、中序序列创建二叉树
BiTree Pre_InCreateBT(const char Pre[], const char In[], int len)
{
    // 递归终止条件
    if (len <= 0)
        return NULL;
    // 表示”根节点“在中序序列的位置
    int i = 0;
    // 在中序序列寻找”根节点“
    for (; i < len; i++)
        if (*Pre == In[i])
            break;
    // 创建节点空间，进行递归建立二叉树
    BiTree T = new BiTNode;
    T->data = *Pre;
    T->lchild = Pre_InCreateBT(Pre + 1, In, i);
    T->rchild = Pre_InCreateBT(Pre + 1 + i, In + i + 1, len - i - 1);
    return T;
}

// 后序、中序序列创建二叉树
BiTree In_PostCreateBT(const char Post[], const char In[], int len)
{
    // 递归终止条件
    if (len <= 0)
        return NULL;
    // 表示”根节点“在中序序列的位置
    int i = 0;
    // 在中序序列寻找”根节点“
    for (; i < len; i++)
        if (Post[len - 1] == In[i])
            break;
    // 创建节点空间，进行递归建立二叉树
    BiTree T = new BiTNode;
    T->data = Post[len - 1];
```

```
    T->lchild = Pre_InCreateBT(Post, In, i);
    T->rchild = Pre_InCreateBT(Post + i, In + i + 1, len - i - 1);
    return T;
}

// 打印二叉树(广义表表示法)
void PrintBiTree(BiTree T)
{
    if (!T) // 递归终止条件
        return;

    cout << T->data;

    // 只有在左右孩子至少有一个存在时才访问
    if (T->lchild || T->rchild)
    {
        cout << "(";      // 要访问孩子时添加（
        PrintBiTree(T->lchild);    // 递归调用
        cout << ",";      // 访问完毕左孩子要访问右孩子时添加,
        PrintBiTree(T->rchild);    // 递归调用
        cout << ")";      // 访问完毕左右孩子添加）
    }
}

// 打印二叉树(凹入表示法)
void PrintBiTree(BiTree T, int depth)
{
    // 递归终止条件
    if (!T)
        return;
    // 打印凹入的符号，depth为凹入深度
    for (int i = 0; i < depth; i++)
        cout << "***";
    cout << T->data;
    cout << endl;
    // 非叶子节点才需要递归深入，减少递归深度
    if (T->lchild || T->rchild)
    {
        PrintBiTree(T->lchild, depth + 1);
        PrintBiTree(T->rchild, depth + 1);
    }
}
```

第 3 题：

```
#include <iostream>
using namespace std;

#pragma warning(disable:4996)

// 哈夫曼树数据域字符长度
#define StrLen 5

 // 哈夫曼树定义
typedef struct
{
    char data[StrLen];              // 数据域
    int weight;                     // 结点权值
    int parent, lchild, rchild;     // 结点的父结点、左孩子、右孩子下标
}HTNode, * HuffmanTree;

// 所用接口声明
void CreateHT(HuffmanTree HT, int Weight[], int n);  // 构造哈夫曼树
void PrintHT(HuffmanTree HT, int n);                 // 打印哈夫曼树数组
void HFCodeing(HuffmanTree HT, char* HFCode[], int n);// 生成哈夫曼编码
void PrintHTCode(char* HFCode[], HuffmanTree HT, int n);// 打印哈夫曼编码
int NodeWPL(HuffmanTree HT, int i);                  // 求该结点的带权路径值，权*根结点到该结点路径长
int GetWPL(HuffmanTree HT, int n);                   // 求哈夫曼树的平均查找长度WPL


int main()
{
    int n = 15;
    char** HFCode = new char* [n];
    int Weight[] = { 1192,677,541,518,462,450,242,195,190,181,174,157,138,124,123 };
    const char* data[15] =
{ "The","of","a","to","and","in","that","he","is","at","on","for","His","are",  "be" };
    // 申请哈夫曼树的空间
    HuffmanTree HT = new HTNode[2 * n - 1];
    // 结点数据域赋值
    for (int i = 0; i < n; i++)
        strcpy_s(HT[i].data, data[i]);

    cout << "1.根据表格数据构造哈夫曼树如下：" << endl;
    CreateHT(HT, Weight, n);
```

```
    PrintHT(HT, n);

    cout << "2.输出对应的哈夫曼编码如下：" << endl;
    HFCodeing(HT, HFCode, n);
    PrintHTCode(HFCode, HT, n);

    cout << endl << "3.输出平均查找长度：" << GetWPL(HT, n) << endl;
    return 0;
}

// 构造哈夫曼树
// 哈夫曼树、权值、节点数量
void CreateHT(HuffmanTree HT, int Weight[], int n)
{
    int i1, i2;              // i1,i2表示最小权值的两个结点下标
    int min1, min2;  // min1,min2表示两个最小权值
    // 初始化哈夫曼树
    for (int i = 0; i < 2 * n - 1; i++)
    {
        HT[i].lchild = -1;
        HT[i].rchild = -1;
        HT[i].parent = -1;
    }

    // 初始结点按给定权值赋值
    for (int i = 0; i < n; i++)
        HT[i].weight = Weight[i];

    // k实际上为选取两个节点组成新树的根储存在数组的下标
    for (int k = n; k < 2 * n - 1; k++)
    {
        // 寻找没有父结点的两个权值最小的结点下标
        // 每次查找重置下标；min1=min2=int类型下最大值
        i1 = i2 = 0;
        min1 = min2 = 2147483647;
        // i是一个遍历哈夫曼数组的下标变量，此循环中k为循环次数
        for (int i = 0; i < k; i++)
        {
            // 先判断是否有父结点
            if (HT[i].parent == -1)
            {
                if (HT[i].weight < min1)
                {
                    swap(min1, min2);
```

```
                    swap(i1, i2);
                    min1 = HT[i].weight;
                    i1 = i;
                }
                else if (HT[i].weight < min2)
                {
                    min2 = HT[i].weight;
                    i2 = i;
                }
            }
        }
        // 找到两个最小下标后，组成新树
        // 新树的权值=两最小结点权值和
        HT[k].weight = min1 + min2;
        //-------父亲有了孩子~~-------
        HT[k].lchild = i1;
        HT[k].rchild = i2;
        //-------孩子有了父亲~~-------
        HT[i1].parent = k;
        HT[i2].parent = k;
    }
}

// 打印哈夫曼树数组
void PrintHT(HuffmanTree HT, int n)
{
    cout << "weight" << '\t';
    cout << "parent" << '\t' << "lchild" << '\t' << "rchild" << '\t' << "data" << endl;
    for (int i = 0; i < 2 * n - 1; i++)
    {
        cout << HT[i].weight << '\t';
        cout << HT[i].parent << '\t';
        cout << HT[i].lchild << '\t';
        cout << HT[i].rchild << '\t';
        if (i < n)
            cout << HT[i].data << endl;
        else
            cout << "空" << endl;
    }
    cout << endl;
}
```

第 4 题：

```
#include <iostream>
using namespace std;

// 数据域元素类型
#define ElemType char

// 二叉树节点定义
typedef struct BiTNode
{
    ElemType data;                    // 数据域
    struct BiTNode* lchild, * rchild;  // 左、右孩子指针
}BiTNode, * BiTree;

// 二叉树操作
BiTree CreateBiTree(string str);          // 创建二叉树
void DestroyBiTree(BiTree& T);            // 销毁二叉树
void PrintBiTree(BiTree T, int depth);    // 打印二叉树(凹入表示法)
double CalculareBiTree(BiTree T);         // 计算

int main()
{
    string s = "1+2*3-4/5";
    BiTree B;
    cout << "1.根据" << s << "构造二叉树" << endl;
    B = CreateBiTree(s);
    cout << endl << "2.凹入表示法输出二叉树" << endl;
    PrintBiTree(B, 0);
    cout << endl << "3.根据二叉树计算表达式值为：" << CalculareBiTree(B) << endl;
    DestroyBiTree(B);
    return 0;
}

// 创建二叉树
BiTree CreateBiTree(string str)
{
    // 申请新建结点空间
    BiTNode* b = new BiTNode;
    // 确定表达式长度（最大循环次数）
    int len = str.length();
    // 表达式空，返回NULL
```

```
    if (!len)
        return NULL;
    /*
    Tips1：从表达式尾部开始查找
        观察题目给出的图可以知道，表达式后面的符号在树的上层
        建树是从根往下建立，所以从表达式尾部开始查找
    Tips2：先找+-
        因为计算树表达式时利用先序遍历，首先访问根，然后左、右
        根据递归原理，递归到最深才返回
        运算法则先* / 再+-，所以* /在最深，+-在上层
    */
    // 先从字符串尾部开始往回寻找+ -
    for (int i = len - 1; i >= 0; i--)
    {
        if (str[i] == '+' || str[i] == '-')
        {
            // 符号作为根结点
            b->data = str[i];
            // 符号左边表达式进入递归继续创建
            b->lchild = CreateBiTree(str.substr(0, i));
            // 符号右边表达式进入递归继续创建
            b->rchild = CreateBiTree(str.substr(i + 1));
            return b;
        }
    }
    // 如果没有找到+-；就字符串尾部开始往回寻找* /
    for (int i = len - 1; i >= 0; i--)
    {
        if (str[i] == '*' || str[i] == '/')
        {
            // 符号作为根结点
            b->data = str[i];
            // 符号左边表达式进入递归继续创建
            b->lchild = CreateBiTree(str.substr(0, i));
            // 符号右边表达式进入递归继续创建
            b->rchild = CreateBiTree(str.substr(i + 1));
            return b;
        }
    }
    // 最后如果+-/*都没有找到，只剩下数字了
    // 数字作为叶子结点，直接赋值返回
    b->data = str[0];
    b->lchild = NULL;
    b->rchild = NULL;
```

```
    return b;
}

// 销毁二叉树
void DestroyBiTree(BiTree& T)
{
    // 本算法递归直到叶子节点
    // 然后删除叶子节点返回父节点

    if (!T) // 递归终止条件
        return;
    // 有孩子才继续递归深入，没有代表目前节点为叶子，直接释放
    if (T->lchild || T->rchild)
    {
        DestroyBiTree(T->lchild);
        DestroyBiTree(T->rchild);
    }
    delete T;
    T = NULL;
}

// 打印二叉树(凹入表示法)
void PrintBiTree(BiTree T, int depth)
{
    // 递归终止条件
    if (!T)
        return;
    // 打印凹入的符号，depth为凹入深度
    for (int i = 0; i < depth; i++)
        cout << "###";
    cout << T->data;
    cout << endl;
    // 非叶子节点才需要递归深入，减少递归深度
    if (T->lchild || T->rchild)
    {
        PrintBiTree(T->lchild, depth + 1);
        PrintBiTree(T->rchild, depth + 1);
    }
}

// 计算
double CalculareBiTree(BiTree T)
{
    if (!T)
```

```
        return -1;
    switch (T->data)
    {
    case '+':
        return CalculareBiTree(T->lchild) + CalculareBiTree(T->rchild);
    case '-':
        return CalculareBiTree(T->lchild) - CalculareBiTree(T->rchild);
    case '*':
        return CalculareBiTree(T->lchild) * CalculareBiTree(T->rchild);
    case '/':
        return CalculareBiTree(T->lchild) / CalculareBiTree(T->rchild);
    default:
        return T->data - '0';
    }

}
```

第 5 题：

```
#include <iostream>
#include <fstream>
using namespace std;

// 数据域元素类型
#define ElemType string

// 二叉树节点定义
typedef struct GenNode
{
    ElemType data;                                  // 数据域
    struct GenNode* lchild, * rchild;  // 左、右孩子指针
}GenNode, * GenTree;

// 全局变量(文件读写)
ifstream ifs;
ofstream ofs;
string p1, p2, p3;


// 显示菜单
void ShowMainMenu()
{
    cout << "**********************" << endl;
```

```
    cout << "******二叉树家谱******" << endl;
    cout << "**********************" << endl;
    cout << "** (1)文件操作功能  **" << endl;
    cout << "** (2)家谱操作功能  **" << endl;
    cout << "**********************" << endl;
    cout << "请输入：";
}
void ShowFMenu()
{
    cout << "1.家谱记录的输入" << endl;
    cout << "2.家谱记录的输出" << endl;
    cout << "3.清除全部文件记录" << endl;
    cout << "4.将家谱记录存盘" << endl;
    cout << "请输入：";
}
void ShowSMenu()
{
    cout << "1.括号表示法输出家谱" << endl;
    cout << "2.查找某人的所有儿子" << endl;
    cout << "3.查找某人的所有祖先" << endl;
    cout << "请输入：";
}

// 输出家谱
void PrintGen(GenTree Gen)
{
    if (!Gen)
    {
        cout << "空！" << endl;
        return;
    }
    // 有妻子
    if (Gen->lchild!=NULL)
    {
        cout << Gen->data << '\t';
        GenNode* p = Gen->lchild; //妻子
        while (p)
        {
            cout << p->data << '\t';
            p = p->rchild; // 孩子
        }
    }
    cout << endl;
    // 孩子当家作主，递归
```

```
    if (!Gen->lchild)
        return;
    GenNode* q = Gen->lchild->rchild;
    while (q)
    {
        PrintGen(q);
        q = q->rchild;
    }
}

// 查找
GenNode* Find(GenTree Gen, string name)
{
    if (!Gen)    // 递归终止条件
        return NULL;

    if (Gen->data == name)
        return Gen;

   // 只有在左右孩子至少有一个存在时才访问
    if (Gen->lchild || Gen->rchild)
    {
        Find(Gen->lchild,name);    // 递归调用
        Find(Gen->rchild,name);    // 递归调用
    }
}

// 创建家庭
void CreateFamliy(GenNode*& F)
{
    // 父、母、孩子
    // 父左子为妻子，右子为孩子
    cout << "输入父母、孩子名字，空格分隔：";
    cin >> p1 >> p2 >> p3;
    GenTree father = new GenNode;
    GenNode* mother = new GenNode;
    GenNode* child = new GenNode;
    father->data = p1;
    mother->data = p2;
    child->data = p3;
    father->lchild = mother;
    father->rchild = NULL;
    mother->lchild = NULL;
    mother->rchild = child;
```

```
    child->lchild = NULL;
    child->rchild = NULL;
    f = father;
}

// 创建族谱
void CreateGen(GenTree Gen)
{
    // 祖先结点，父结点
    GenNode* famliy, * father;
    CreateFamliy(famliy);
    father = Find(Gen, famliy->data);
    // 没有找到
    if (!father)
    {
        cout << "没有祖先~" << endl;
        return;
    }
    // 如果没有左孩子，说明是没有老婆孩子
    // 下面添加老婆孩子
    if (!father->lchild)
        father->lchild = famliy->lchild;
    // 有老婆孩子，加二胎
    else
    {
        GenNode* p = father->lchild;  // 妻子结点
        while (p->rchild)
            p = p->rchild;    // 一直找孩子
        p->rchild = famliy->lchild->rchild; // 孩子
    }
}

// 清空
void DestoryGen(GenTree Gen)
{
    if (!Gen)
        return;
    DestoryGen(Gen->lchild);
    DestoryGen(Gen->rchild);
    delete Gen;
    Gen = NULL;
}

// 文件保存
```

```
void SaveGen(GenTree Gen)
{
    if (!Gen)
    {
        ofs << "*" << " ";
        return;
    }
    ofs.open("data.txt", ios::app);
    if (!ofs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }
    ofs << Gen->data << " ";
    ofs.close();
    SaveGen(Gen->lchild);
    SaveGen(Gen->rchild);
}

// 括号表示法打印
void BraPrintGen(GenTree Gen)
{
    if (!Gen)    // 递归终止条件
        return;

    cout << Gen->data;

    // 只有在左右孩子至少有一个存在时才访问
    if (Gen->lchild || Gen->rchild)
    {
        cout << "(";      // 要访问孩子时添加（
        BraPrintGen(Gen->lchild); // 递归调用
        cout << ",";      // 访问完毕左孩子要访问右孩子时添加,
        BraPrintGen(Gen->rchild); // 递归调用
        cout << ")";      // 访问完毕左右孩子添加）
    }
}

// 找到所有的孩子
bool ShowALLSon(GenTree Gen, string name)
{
    GenNode* q;
    GenNode* p = Find(Gen, name);
    if (!p)
```

```
    {
        cout << "查无此人" << endl;
        return false;
    }
    // 母亲
    if (!Gen->lchild)
    {
        q = p->rchild;
        cout << Gen->data << "母亲孩子是：";
        while (q)
        {
            cout << q->data << " ";
            q = q->rchild;
        }
        return true;
    }
    // 父亲
    p = p->lchild->rchild;
    cout << Gen->data << "父亲孩子是：";
    while (p)
    {
        cout << p->data << " ";
        p = p->rchild;
    }
    return true;
}

// 找到孩子所有祖先
bool ShowALLFather(GenTree Gen, string name)
{
    // 家谱空
    if (!Gen)
        return false;
    // 递归终止，这个路径递归找到了这个孩子
    if (Gen->data == name)
        return true;
    // 证明这个路径上的所有结点都是其祖先
    if (ShowALLFather(Gen->lchild,name)|| ShowALLFather(Gen->rchild, name))
    {
        cout << Gen->data << " ";
        return true;
    }
    return false;
}
```

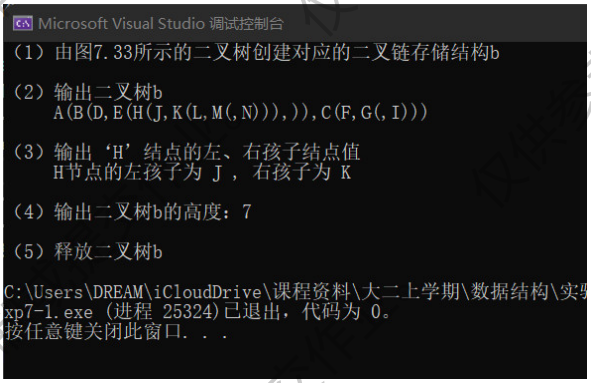
```
int main()
{
    string name;
    GenTree Gen = NULL;   // 总家谱
    int flag = -1;
    int val1, val2, val3;
    while (true)
    {
        cout << endl;
        ShowMainMenu();
        cin >> val1;
        switch (val1)
        {
        case 1:
            ShowFMenu();
            cin >> val2;
            switch (val2)
            {
            // 家谱记录的输入
            case 1:
                if (flag == -1)
                {
                    CreateFamliy(Gen);
                    flag = 1;
                }
                else
                    CreateGen(Gen);
                break;

            // 家谱记录的输出
            case 2:
                PrintGen(Gen);
                break;

            // 清除全部文件记录
            case 3:
                DestoryGen(Gen);
                break;

            // 将家谱记录存盘
            case 4:
                SaveGen(Gen);
                break;
```

```
            default:
                break;
            }
            break;
        case 2:
            ShowSMenu();
            cin >> val3;
            switch (val3)
            {
            // 括号表示法输出家谱
            case 1:
                BraPrintGen(Gen);
                break;
            // 查找某人的所有儿子
            case 2:
                cout << endl << "请输入查找人姓名：";
                cin >> name;
                ShowALLSon(Gen, name);
                break;

            case 3:
                cout << endl << "请输入查找人姓名：";
                cin >> name;
                ShowALLFather(Gen, name);
                break;
            }
            break;
        default:
            return 0;
        }
    }

    return 0;

}
```

五、实验结果及分析

1、

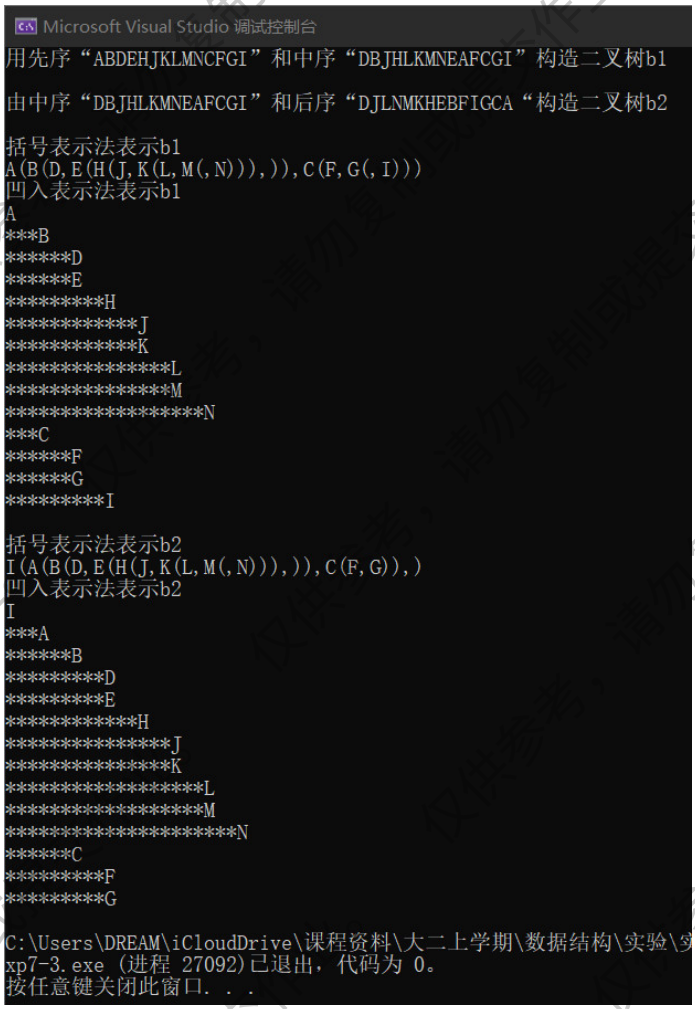
运行结果：



运行结果正确

2、

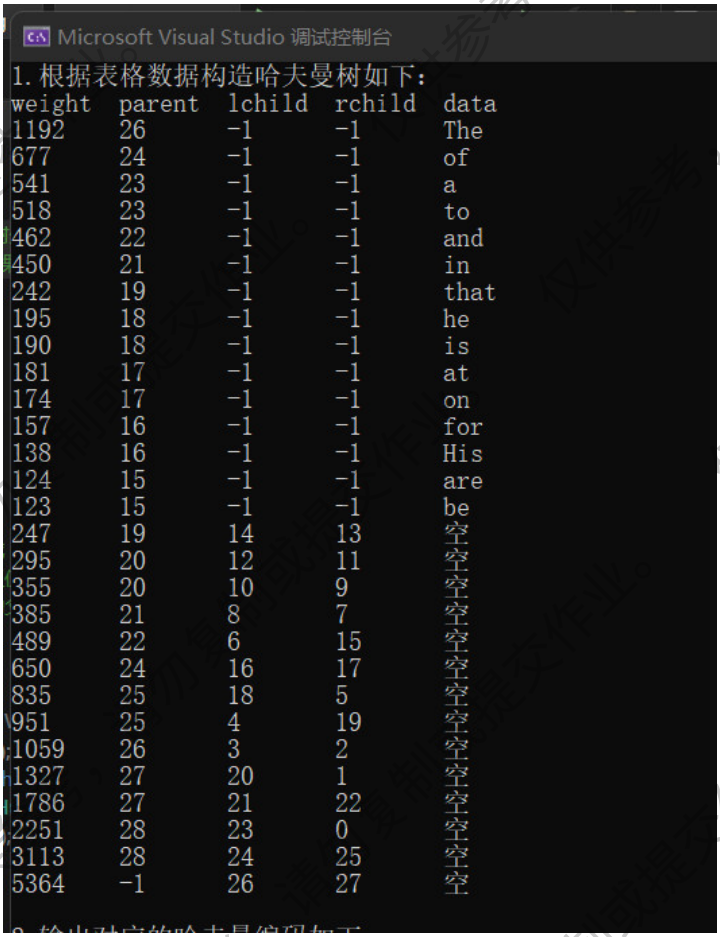
运行结果：



运行结果正确

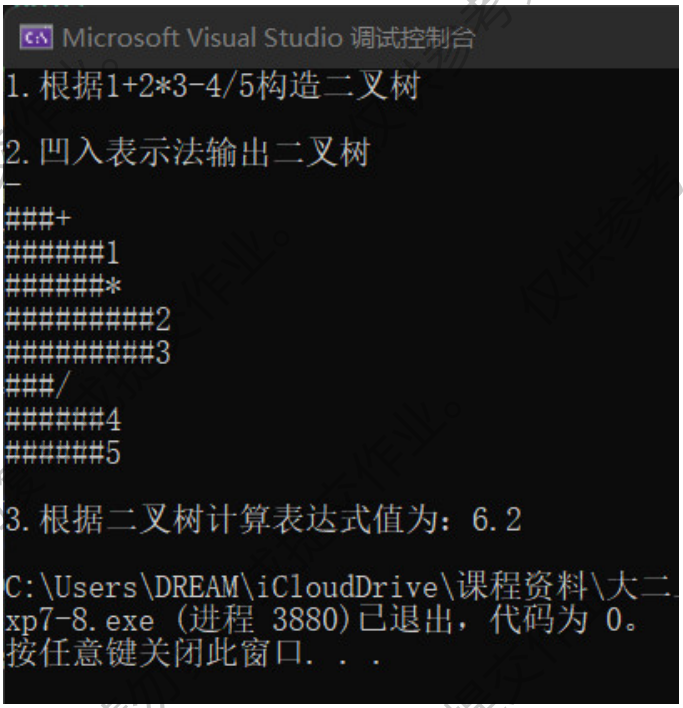
3、

运行结果：



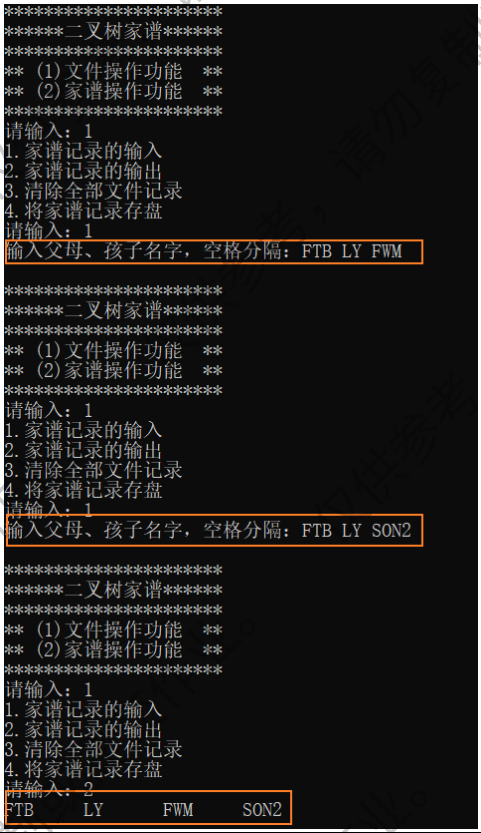
运行结果正确

4、
运行结果：



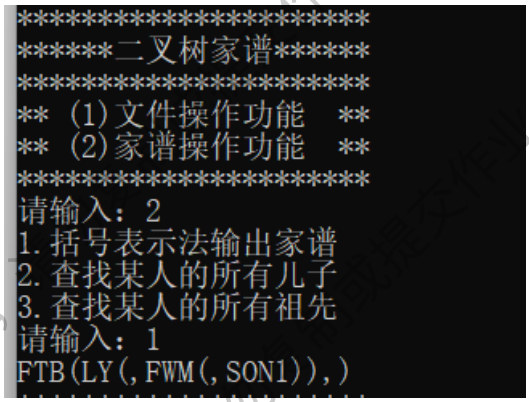
运行结果正确

5、
运行结果：
输入输出

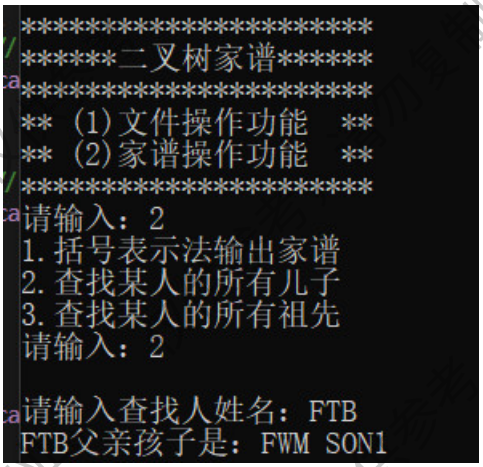


文件保存

括号表示



输出所有孩子



输出所有祖先

```
**********************
******二叉树家谱******
**********************
** (1)文件操作功能   **
** (2)家谱操作功能   **
**********************
请输入：2
1.括号表示法输出家谱
2.查找某人的所有儿子
3.查找某人的所有祖先
请输入：3

请输入查找人姓名：SON1
FWM LY FTB
```

运行结果正确

实验体会与心得：

本次实验是关于二叉树的一些实验，二叉树是一种递归定义的结构，所以有关二叉树了一些操作大量使用递归完成。本次实验的难度较大，主要难点在于递归的大量使用，只有深刻理解的递归的含义，才能够写出相关函数操作，使我我认识到了递归的魅力。