**西 安 邮 电 大 学**

（计算机学院）

课内实验报告

**课程名称： 数据结构**

**实验名称：二叉树的建立、遍历、结点数统计、高度计算，**

**哈夫曼树的构建，哈夫曼编码、译码**

**专业名称： 数据科学与大数据技术**

**班 级： 大数据2001**

**学生姓名： 陈雨涵**

**学 号： 04204021**

**指导教师： 初建玮**

**实验日期： 2021年11月3日**

**实验题目**

1. **实验目的**

**通过二叉树的建立、遍历、结点数的统计、高度的计算以及哈夫曼编码等实验项目，巩固第六章所学的知识。通过树相关应用算法的实现，训练学生设计算法以及解决实际问题的能力。通过二叉树的创建，遍历，进一步训练学生对递归算法的理解和编写能力，引导、激发学生对本课程学习的积极性。**

1. **实验内容**

**1. 二叉树的建立、遍历、结点数的统计、高度计算**

**输入二叉树的扩展先序序列，以二叉链表作为存储结构，建立二叉树。1.输出这棵二叉树的先序、中序和后序遍历序列（后序遍历用非递归算法）；2.分别统计这颗二叉树中叶子结点，度为1，为2的结点个数。3.计算这棵树的高度并输出**

**2. 哈夫曼树的构建，哈夫曼编码、译码**

**已知某通信报文中字符出现的频率，（1）构造一棵哈夫曼树，依次给出各字符编码结果 （2）对给定字符串进行编码；（3）对给定编码串进行译码**

1. **实验方案设计**

**1. 二叉树的建立、遍历、结点数的统计、高度计算**

**二叉树的先序遍历和中序遍历我使用的是书上提到的递归算法，比较简单，在此不再赘述。**

**对于二叉树的后序遍历，题目中要求使用非递归算法。解决该问题的方法有多种，这里我使用的是“栈+标志位”的思路，即每个结点入栈时加上一个标志位flag同时入栈，进左子树访问时置flag=0，进右子树访问时置flag=1，当从子树返回时，通过判断flag的值决定下一步的动作。**

**叶子结点，度为1和2的结点数的统计同样可以采用递归的思路，如果根节点为空，则返回0；如果根节点的左右子树均为空，叶子结点数+1；度为1和2的情况相同。**

**在求树的高度时也是用递归实现，先递归遍历左子树求高度hl，再递归遍历右子树求高度hr，取hl和hr中较大的那个值，树的高度h=(hl > hr ? hl :hr) + 1;**

**2. 哈夫曼树的构建，哈夫曼编码、译码**

**哈夫曼树是一棵二叉树，对于n个叶子的哈夫曼树，它共有2n-1个结点，因此可以存储在一个大小为2n-1的一维数组中。在哈夫曼编译码的实际应用中，我们既需要从根结点出发走一条从根到叶子的路径，又需要从叶子结点出发走一条从叶子到根的路径，所以每个结点既需要孩子的信息，又需要双亲的信息，因此每个结点可以设置成三叉链表结 点结构 (Weight, Parent, Lchild, Rchild)。静态三叉链表数组中，前n个元素存储叶子结点，后n-1个元素存储不断生成的新结点，即度为2的分支结点，最后一个元素将是哈夫曼树的根节点。**

**建立哈夫曼树可分为以下三步：**

**(1)初始化所有结点。构造n个根结点，权值置为Wi，孩子和双亲指针置为0。这样做即将数组前n个元素视为根结点。**

**(2)初始化后n-1个元素，各域均置为0。**

**(3)构造新结点，构建哈夫曼树。在数组已有结点中选双亲为0且权值最小的两个结点，构造新结点，新结点下标为数组中已有结点的下一个位置，其权值为选取的两权值最小结点的权值之和，其左、右孩子分别为两权值最小的根节点，并将两权值最小结点的双亲改为指向新结点。重复此过程n-1次。**

**哈夫曼编码可分为以下三步：**

**(1)从叶子结点开始，沿结点的双亲链追溯到根结点，追溯过程中，每上升一层，则经过了一条分支，便可得到一位哈夫曼编码值，左分支为0，右分支为1。**

**(2)产生哈夫曼编码串时，使用一个临时数组cd，每位编码从后向前逐位放入cd中，由start指针控制存放的次序。**

**(3)到达根结点时，一个叶子的编码构造完成，此时将cd数组中以start开始的串复制到动态申请的编码串空间。**

**哈夫曼译码的主要思路是：从哈夫曼树的根出发，根据每一位编码的0或1确定进入左子树或右子树，直至到达叶子结点，即可得到一个对应的字符。重复此过程，直至编码串处理结束。**

1. **实验程序的功能**

**1. 二叉树的建立、遍历、结点数的统计、高度计算**

/\* 二叉链表结点 \*/

typedef *struct* *Node* {

*char* data;              /\* 二叉链表的结点信息 \*/

*struct* *Node*\* LChild;    /\* 结点的左子树 \*/

*struct* *Node*\* RChild;    /\* 结点的右子树 \*/

} *BiTNode*, \* *BiTree*;

/\* 栈结构 \*/

typedef *struct* *elem* {

*BiTNode* node;    /\* 栈中元素的结点信息 \*/

*int* flag;        /\* 标记位，进左子树为0，进右子树为1 \*/

} *ElemType*;

typedef *struct* *stack* {

*ElemType* elem[MAXSIZE];    /\* 顺序栈数组 \*/

*int* top;                   /\* 栈顶指针 \*/

} *SeqStack*;

/\* 自定义函数声明 \*/

*BiTNode*\* CreateBiTree(*BiTNode*\* *root*);   /\* 扩展先序序列建立二叉树 \*/

*void* PreOrder(*BiTree* *root*);             /\* 递归先序遍历二叉树 \*/

*void* InOrder(*BiTree* *root*);              /\* 递归中序遍历二叉树 \*/

*void* PostOrder(*BiTree* *root*);            /\* 非递归后序遍历二叉树 \*/

*SeqStack*\* InitStack(*void*);              /\* 初始化栈 \*/

*int* Empty(*SeqStack*\* *s*);                 /\* 判空栈 \*/

*int* Push(*SeqStack*\* *s*, *ElemType* *x*);      /\* 入栈 \*/

*int* Pop(*SeqStack*\* *s*, *ElemType*\* *x*);      /\* 出栈 \*/

*ElemType*\* GetTop(*SeqStack*\* *s*);          /\* 取栈顶元素 \*/

*int* Leafjishu(*BiTree* *root*);           /\* 统计二叉树中叶子结点的个数 \*/

void Yi(BTree T) //统计度为1的结点数

void Er(BTree T) //统计度为2的结点数

*int* BiTreeDeepth(*BiTree* *root*);          /\* 计算二叉树的高度 \*/

**2. 哈夫曼树的构建，哈夫曼编码、译码**

/\* 哈夫曼树结构 \*/

typedef *struct* *node* {

*int* weight;

*int* parent;

*int* LChild;

*int* RChild;

*char* data;

} *HTNode*, *HuffmanTree*[M + 1];

/\* 编码结构 \*/

typedef *char*\* *HuffmanCode*[N + 1];

/\* 自定义函数声明 \*/

*void* select(*HuffmanTree* *ht*, *int* *n*, *int*\* *s1*, *int*\* *s2*);

*void* CreateHuffmanTree(*HuffmanTree* *ht*, *int* *w*[], *int* *n*);

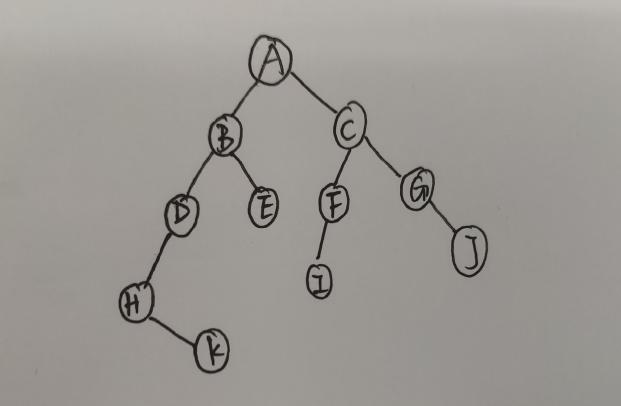
*void* CreateHuffmanCode(*HuffmanTree* *ht*, *HuffmanCode* *hc*, *int* *n*);

*void* HuffmanDeCode(*HuffmanTree* *ht*, *char* *ch*[]);

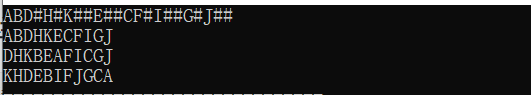
1. **测试数据及运行结果**

**1.  二叉树的建立、遍历、结点数的统计、高度计算**

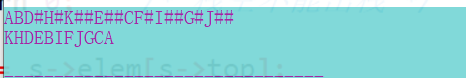
**第一组：**



**二叉树的创建和递归遍历**

****

**非递归后序遍历**

****

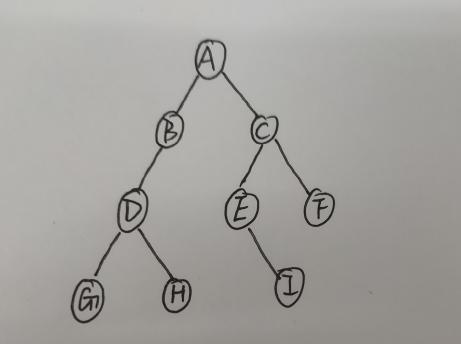
**结点个数**

****

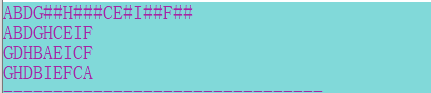
**二叉树的高度**

****

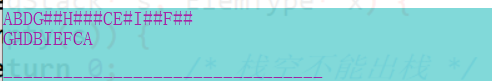
**第二组：**



**二叉树的创建和递归遍历**

****

**非递归后序遍历**

****

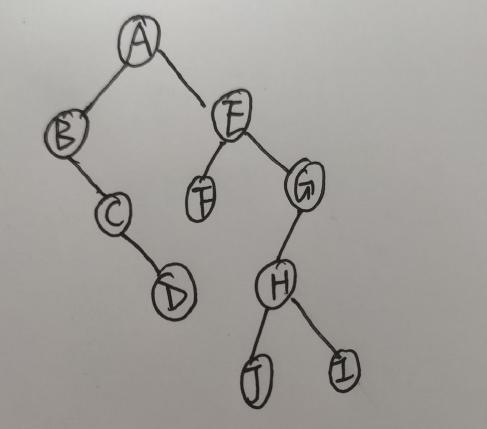
**结点个数**

****

**二叉树的高度**

****

**第三组：**



**二叉树的创建和递归遍历**

****

**非递归后序遍历**

****

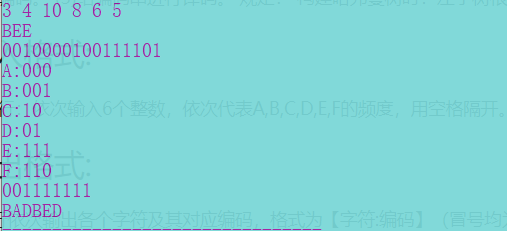
**结点个数**

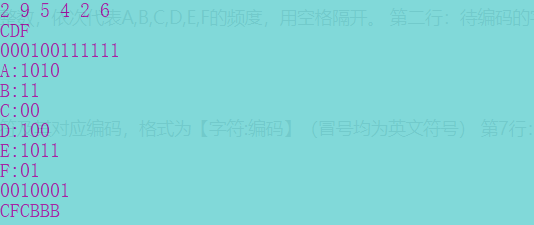
****

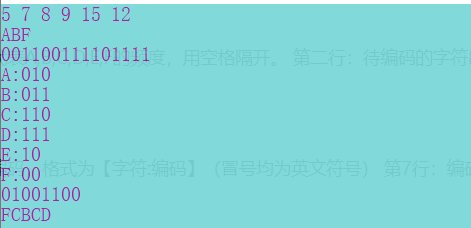
**二叉树的高度**

****

**2. 哈夫曼树的构建，哈夫曼编码、译码**

****

****

****

1. **实验总结**

**本次实验的内容是二叉树的建立、遍历和结点数的统计以及哈夫曼编码。第1道实验题中，二叉树的前序遍历和中序遍历我是采用的递归算法，比较简单，没有遇到什么问题；后序遍历引入了一个标志位采用非递归算法，用到了栈这种数据结构。但在编写代码的过程中，我犯了一个很低级的错误——忘记在主函数中初始化栈，导致了一系列bug，并且花费了很长时间去调试。第2道实验题，我通过阅读书上的代码，自己完成了select函数的编写，也没有遇到什么大问题。在本次实验中，我巩固和加深了对树的理解，提高了综合运用本课程所学知识解决实际问题的能力。我会利用课余时间通过阅读别人发布的相关博客等途径去了解其他的思路。**

**优点：**可以清楚地得到二叉树的三种遍历序列以及得到树的高度。

**对设计及调试过程的心得体会；**

设计代码时提前思考代码的逻辑结构，考虑算法的时间复杂度以及空间复杂度，思考算法是否还能改进会提高效率。算法是可以改进的，要勤于思考，善于发现问题并解决问题。

1. **附录**

**1. 二叉树的建立、遍历和结点数的统计**

**二叉树的创建和递归遍历**

#include <stdio.h>  
#include <stdlib.h>  
#define MAXSIZE 100005  
  
*/\* 二叉链表结点 \*/***typedef struct** Node {  
 **char** data; */\* 二叉链表的结点信息 \*/* **struct** Node\* LChild; */\* 结点的左子树 \*/* **struct** Node\* RChild; */\* 结点的右子树 \*/*} BiTNode, \* BiTree;  
  
*/\* 栈结构 \*/***typedef struct** elem {  
 BiTNode node; */\* 栈中元素的结点信息 \*/* **int** flag; */\* 标记位，进左子树为0，进右子树为1 \*/*} ElemType;  
  
**typedef struct** stack {  
 ElemType elem[MAXSIZE]; */\* 顺序栈数组 \*/* **int** top; */\* 栈顶指针 \*/*} SeqStack;  
  
*/\* 自定义函数声明 \*/*BiTNode\* CreateBiTree(BiTNode\* root); */\* 扩展先序序列建立二叉树 \*/***void** PreOrder(BiTree root); */\* 递归先序遍历二叉树 \*/***void** InOrder(BiTree root); */\* 递归中序遍历二叉树 \*/***void** PostOrder(BiTree root); */\* 非递归后序遍历二叉树 \*/*SeqStack\* InitStack(**void**); */\* 初始化栈 \*/***int** Empty(SeqStack\* s); */\* 判空栈 \*/***int** Push(SeqStack\* s, ElemType x); */\* 入栈 \*/***int** Pop(SeqStack\* s, ElemType\* x); */\* 出栈 \*/*ElemType\* GetTop(SeqStack\* s); */\* 取栈顶元素 \*/  
  
/\* 主函数 \*/***int** main(**void**) {  
 BiTNode\* root = CreateBiTree(root);  
 PreOrder(root);  
 printf("\n");  
 InOrder(root);  
 printf("\n");  
 PostOrder(root);  
 **return** 0;  
}  
  
*/\* 自定义函数 \*/*BiTNode\* CreateBiTree(BiTNode\* root) {  
 **char** ch;  
 ch = getchar();  
 **if** (ch == '#') {  
 **return** NULL;  
 } **else** {  
 root = (BiTNode\*)malloc(**sizeof**(BiTNode));  
 root->data = ch;  
 root->LChild = CreateBiTree(root->LChild);  
 root->RChild = CreateBiTree(root->RChild);  
 }  
 **return** root;  
}  
  
**void** PreOrder(BiTree root) {  
 **if** (root) {  
 printf("%c", root->data);  
 PreOrder(root->LChild);  
 PreOrder(root->RChild);  
 }  
}  
  
**void** InOrder(BiTree root) {  
 **if** (root) {  
 InOrder(root->LChild);  
 printf("%c", root->data);  
 InOrder(root->RChild);  
 }  
}  
  
**void** PostOrder(BiTree root) {  
 SeqStack\* s = InitStack();  
 **while** (root || !Empty(s)) {  
 **while** (root) {  
 ElemType temp;  
 temp.node = \* root;  
 temp.flag = 0;  
 Push(s, temp);  
 root = root->LChild;  
 }  
 **while** (!Empty(s) && GetTop(s)->flag == 1) {  
 ElemType\* x = GetTop(s);  
 Pop(s, x);  
 printf("%c", x->node.data);  
 }  
 **if** (!Empty(s)) {  
 root = GetTop(s)->node.RChild;  
 GetTop(s)->flag = 1;  
 }  
 }  
}  
  
SeqStack\* InitStack(**void**) {  
 SeqStack\* s;  
 s = (SeqStack\*)malloc(**sizeof**(SeqStack));  
 s->top = -1;  
 **return** s;  
}  
  
**int** Empty(SeqStack\* s) {  
 **if** (s->top == -1) {  
 **return** 1;  
 } **else** {  
 **return** 0;  
 }  
}  
  
**int** Push(SeqStack\* s, ElemType x) {  
 **if** (s->top == MAXSIZE - 1) {  
 **return** 0; */\* 栈满不能入栈 \*/* } **else** {  
 s->top++;  
 s->elem[s->top] = x;  
 **return** 1;  
 }  
}  
  
**int** Pop(SeqStack\* s, ElemType\* x) {  
 **if** (Empty(s)) {  
 **return** 0; */\* 栈空不能出栈 \*/* } **else** {  
 \* x = s->elem[s->top];  
 s->top--;  
 **return** 1;  
 }  
}  
  
ElemType\* GetTop(SeqStack\* s) {  
 **if** (Empty(s)) {  
 **return** NULL; */\* 栈空 \*/* } **else** {  
 **return** &(s->elem[s->top]);  
 }  
}

**非递归后序遍历**

#include <stdio.h>  
#include <stdlib.h>  
#define MAXSIZE 100005  
  
*/\* 二叉链表结点 \*/***typedef struct** Node {  
 **char** data; */\* 二叉链表的结点信息 \*/* **struct** Node\* LChild; */\* 结点的左子树 \*/* **struct** Node\* RChild; */\* 结点的右子树 \*/*} BiTNode, \* BiTree;  
  
*/\* 栈结构 \*/***typedef struct** elem {  
 BiTNode node; */\* 栈中元素的结点信息 \*/* **int** flag; */\* 标记位，进左子树为0，进右子树为1 \*/*} ElemType;  
  
**typedef struct** stack {  
 ElemType elem[MAXSIZE]; */\* 顺序栈数组 \*/* **int** top; */\* 栈顶指针 \*/*} SeqStack;  
  
*/\* 自定义函数声明 \*/*BiTNode\* CreateBiTree(BiTNode\* root); */\* 扩展先序序列建立二叉树 \*/***void** PostOrder(BiTree root); */\* 非递归后序遍历二叉树 \*/*SeqStack\* InitStack(**void**); */\* 初始化栈 \*/***int** Empty(SeqStack\* s); */\* 判空栈 \*/***int** Push(SeqStack\* s, ElemType x); */\* 入栈 \*/***int** Pop(SeqStack\* s, ElemType\* x); */\* 出栈 \*/*ElemType\* GetTop(SeqStack\* s); */\* 取栈顶元素 \*/  
  
/\* 主函数 \*/***int** main(**void**) {  
 BiTNode\* root = CreateBiTree(root);  
 PostOrder(root);  
 printf("\n");  
 **return** 0;  
}  
  
*/\* 自定义函数 \*/*BiTNode\* CreateBiTree(BiTNode\* root) {  
 **char** ch;  
 ch = getchar();  
 **if** (ch == '#') {  
 **return** NULL;  
 } **else** {  
 root = (BiTNode\*)malloc(**sizeof**(BiTNode));  
 root->data = ch;  
 root->LChild = CreateBiTree(root->LChild);  
 root->RChild = CreateBiTree(root->RChild);  
 }  
 **return** root;  
}  
  
  
**void** PostOrder(BiTree root) {  
 SeqStack\* s = InitStack();  
 **while** (root || !Empty(s)) {  
 **while** (root) {  
 ElemType temp;  
 temp.node = \* root;  
 temp.flag = 0;  
 Push(s, temp);  
 root = root->LChild;  
 }  
 **while** (!Empty(s) && GetTop(s)->flag == 1) {  
 ElemType\* x = GetTop(s);  
 Pop(s, x);  
 printf("%c", x->node.data);  
 }  
 **if** (!Empty(s)) {  
 root = GetTop(s)->node.RChild;  
 GetTop(s)->flag = 1;  
 }  
 }  
}  
  
SeqStack\* InitStack(**void**) {  
 SeqStack\* s;  
 s = (SeqStack\*)malloc(**sizeof**(SeqStack));  
 s->top = -1;  
 **return** s;  
}  
  
**int** Empty(SeqStack\* s) {  
 **if** (s->top == -1) {  
 **return** 1;  
 } **else** {  
 **return** 0;  
 }  
}  
  
**int** Push(SeqStack\* s, ElemType x) {  
 **if** (s->top == MAXSIZE - 1) {  
 **return** 0; */\* 栈满不能入栈 \*/* } **else** {  
 s->top++;  
 s->elem[s->top] = x;  
 **return** 1;  
 }  
}  
  
**int** Pop(SeqStack\* s, ElemType\* x) {  
 **if** (Empty(s)) {  
 **return** 0; */\* 栈空不能出栈 \*/* } **else** {  
 \* x = s->elem[s->top];  
 s->top--;  
 **return** 1;  
 }  
}  
  
ElemType\* GetTop(SeqStack\* s) {  
 **if** (Empty(s)) {  
 **return** NULL; */\* 栈空 \*/* } **else** {  
 **return** &(s->elem[s->top]);  
 }  
}

**结点个数**

#include <stdio.h>  
#include <stdlib.h>  
  
**typedef struct** Node {  
 **char** data; */\* 二叉链表的结点信息 \*/* **struct** Node\* Ld; */\* 结点的左子树 \*/* **struct** Node\* Rd; */\* 结点的右子树 \*/*} BNode, \* BTree;  
**char** chu[60];  
**int** qizhi;  
**int** yi=0;  
**int** er=0;  
BNode\* CreateBiTree(BNode\* root) {  
 **char** ch;  
 ch = getchar();  
 **if** (ch == '#') {  
 **return** NULL;  
 } **else** {  
 root = (BNode\*)malloc(**sizeof**(BNode));  
 root->data = ch;  
 root->Ld = CreateBiTree(root->Ld);  
 root->Rd = CreateBiTree(root->Rd);  
 }  
 **return** root;  
}  
**int** leafjishu(BTree T, **int** count) {  
 **if** (T) {  
 **if** (T->Ld == NULL && T->Rd == NULL) {  
 count++;  
 chu[qizhi++] = T->data;  
 }  
 count = leafjishu(T->Ld, count);  
 count = leafjishu(T->Rd, count);  
 }  
 **return** count;  
}  
**void** Yi(BTree T){  
 **if**(T!=NULL)  
 {  
 Yi(T->Ld);  
 Yi(T->Rd);  
 **if**((T->Ld==NULL && T->Rd!=NULL)||(T->Ld!=NULL && T->Rd==NULL))  
 {  
 yi++; //统计度为1的结点数目  
 }  
 }  
}  
**void** Er(BTree T){  
 **if**(T!=NULL)  
 {  
 Er(T->Ld);  
 Er(T->Rd);  
 **if**(T->Ld!=NULL && T->Rd!=NULL)  
 {  
 er++; //统计度为2的结点数目  
 }  
 }  
}  
**int** main() {  
 BTree T = CreateBiTree(T);  
 **int** ct = 0,  
 count= leafjishu(T, ct);  
 Yi(T);  
 Er(T);  
 printf("%d %d %d\n", count, yi, er);  
 **int** i;  
 **for** (i = 0; i < count; i++) {  
 printf("%c", chu[i]);  
 }  
 **return** 0;  
}

**二叉树的高度**

#include <stdio.h>  
#include <stdlib.h>  
#define MAXSIZE 100005  
  
*/\* 二叉链表结点 \*/***typedef struct** Node {  
 **char** data; */\* 二叉链表的结点信息 \*/* **struct** Node\* LChild; */\* 结点的左子树 \*/* **struct** Node\* RChild; */\* 结点的右子树 \*/*} BiTNode, \* BiTree;  
  
*/\* 栈结构 \*/***typedef struct** elem {  
 BiTNode node; */\* 栈中元素的结点信息 \*/* **int** flag; */\* 标记位，进左子树为0，进右子树为1 \*/*} ElemType;  
  
**typedef struct** stack {  
 ElemType elem[MAXSIZE]; */\* 顺序栈数组 \*/* **int** top; */\* 栈顶指针 \*/*} SeqStack;  
  
*/\* 自定义函数声明 \*/*BiTNode\* CreateBiTree(BiTNode\* root); */\* 扩展先序序列建立二叉树 \*/***int** BiTreeDeepth(BiTree root); */\* 计算二叉树的高度 \*/  
  
/\* 主函数 \*/***int** main(**void**) {  
 BiTNode\* root = CreateBiTree(root);  
 printf("%d\n", BiTreeDeepth(root));  
 **return** 0;  
}  
  
*/\* 自定义函数 \*/*BiTNode\* CreateBiTree(BiTNode\* root) {  
 **char** ch;  
 ch = getchar();  
 **if** (ch == '#') {  
 **return** NULL;  
 } **else** {  
 root = (BiTNode\*)malloc(**sizeof**(BiTNode));  
 root->data = ch;  
 root->LChild = CreateBiTree(root->LChild);  
 root->RChild = CreateBiTree(root->RChild);  
 }  
 **return** root;  
}  
  
**int** BiTreeDeepth(BiTree root) {  
 **if** (!root) {  
 **return** 0;  
 } **else** {  
 **if** (BiTreeDeepth(root->LChild) > BiTreeDeepth(root->RChild)) {  
 **return** 1 + BiTreeDeepth(root->LChild);  
 } **else** {  
 **return** 1 + BiTreeDeepth(root->RChild);  
 }  
 }  
}

**2. 哈夫曼编码**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#define N 100  
#define M 2 \* N - 1  
  
*/\* 哈夫曼树结构 \*/***typedef struct** node {  
 **int** weight;  
 **int** parent;  
 **int** LChild;  
 **int** RChild;  
 **char** data;  
} HTNode, HuffmanTree[M + 1];  
  
*/\* 编码结构 \*/***typedef char**\* HuffmanCode[N + 1];  
  
*/\* 自定义函数声明 \*/***void** select(HuffmanTree ht, **int** n, **int**\* s1, **int**\* s2);  
**void** CreateHuffmanTree(HuffmanTree ht, **int** w[], **int** n);  
**void** CreateHuffmanCode(HuffmanTree ht, HuffmanCode hc, **int** n);  
**void** HuffmanDeCode(HuffmanTree ht, **char** ch[]);  
  
*/\* 主函数 \*/***int** main(**void**) {  
 HuffmanTree HT;  
 **int** w[7];  
 **for** (**int** i = 1; i <= 6; i++) {  
 scanf("%d", &w[i]);  
 }  
 **char** ch1[10];  
 **char** ch2[100];  
 scanf("%s", ch1);  
 scanf("%s", ch2);  
 CreateHuffmanTree(HT, w, 6);  
 HuffmanCode HC;  
 CreateHuffmanCode(HT, HC, 6);  
 **for** (**int** i = 1; i <= 6; i++) {  
 printf("%c:%s\n", 'A' + i - 1, HC[i]);  
 }  
 **for** (**int** i = 0; i < strlen(ch1); i++) {  
 **switch** (ch1[i]) {  
 **case** 'A':  
 printf("%s", HC[1]);  
 **break**;  
 **case** 'B':  
 printf("%s", HC[2]);  
 **break**;  
 **case** 'C':  
 printf("%s", HC[3]);  
 **break**;  
 **case** 'D':  
 printf("%s", HC[4]);  
 **break**;  
 **case** 'E':  
 printf("%s", HC[5]);  
 **break**;  
 **case** 'F':  
 printf("%s", HC[6]);  
 **break**;  
 }  
 }  
 printf("\n");  
 HuffmanDeCode(HT, ch2);  
 **return** 0;  
}  
  
*/\* 自定义函数 \*/***void** select(HuffmanTree ht, **int** n, **int**\* s1, **int**\* s2) {  
 */\* 找最小、次小权值对应的结点 \*/* **int** temp;  
 **for** (**int** i = 1; i <= n; i++) {  
 **if** (ht[i].parent == 0) {  
 temp = i;  
 **break**;  
 }  
 }  
 **for** (**int** i = 1; i <= n; i++) {  
 **if** (ht[i].parent == 0) {  
 **if** (ht[i].weight < ht[temp].weight) {  
 temp = i;  
 }  
 }  
 }  
 \* s1 = temp;  
 **for** (**int** i = 1; i <= n; i++) {  
 **if** (ht[i].parent == 0 && i != (\* s1)) {  
 temp = i;  
 **break**;  
 }  
 }  
 **for** (**int** i = 1; i <= n; i++) {  
 **if** (ht[i].parent == 0 && i != (\* s1)) {  
 **if** (ht[i].weight < ht[temp].weight) {  
 temp = i;  
 }  
 }  
 }  
 \* s2 = temp;  
}  
  
**void** CreateHuffmanTree(HuffmanTree ht, **int** w[], **int** n) {  
 **int** m = 2 \* n - 1;  
 */\* 初始化前n个元素成为根节点 \*/* **for** (**int** i = 1; i <= n; i++) {  
 ht[i].weight = w[i];  
 ht[i].data = 'A' + i - 1;  
 ht[i].parent = 0;  
 ht[i].LChild = 0;  
 ht[i].RChild = 0;  
 }  
 */\* 初始化后n-1个空元素 \*/* **for** (**int** i = n + 1; i <= m; i++) {  
 ht[i].weight = 0;  
 ht[i].parent = 0;  
 ht[i].LChild = 0;  
 ht[i].RChild = 0;  
 }  
 */\* 从第n+1个元素开始构造新结点 \*/* **for** (**int** i = n + 1; i <= m; i++) {  
 **int** s1, s2;  
 */\* 在ht的前i-1项中选双亲为0且权值最小的两结点s1, s2 \*/* select(ht, i - 1, &s1, &s2);  
 */\* 建新结点，赋权值 \*/* ht[i].weight = ht[s1].weight + ht[s2].weight;  
 */\* 赋新结点左、右孩子指针 \*/* ht[i].LChild = s1;  
 ht[i].RChild = s2;  
 */\* 改s1, s2的双亲指针 \*/* ht[s1].parent = i;  
 ht[s2].parent = i;  
 }  
}  
  
**void** CreateHuffmanCode(HuffmanTree ht, HuffmanCode hc, **int** n) {  
 */\* 从叶子结点到根节点，逆向求哈夫曼编码 \*/* **char**\* code;  
 **int** start;  
 code = (**char**\*)malloc(**sizeof**(**char**) \* n);  
 code[n - 1] = '\0';  
 **for** (**int** i = 1; i <= n; i++) {  
 start = n - 1;  
 **int** c = i;  
 **int** p = ht[i].parent;  
 **while** (p) {  
 --start;  
 **if** (ht[p].LChild == c) {  
 code[start] = '0';  
 } **else** {  
 code[start] = '1';  
 }  
 c = p;  
 p = ht[p].parent;  
 }  
 hc[i] = (**char**\*)malloc(**sizeof**(**char**) \* (n - start));  
 strcpy(hc[i], &code[start]);  
 }  
 free(code);  
}  
  
**void** HuffmanDeCode(HuffmanTree ht, **char** ch[]) {  
 */\* 哈夫曼编码的译码 \*/* **int** m = 11;  
 **for** (**int** i = 0; i < strlen(ch); i++) {  
 **if** (ch[i] == '0') {  
 m = ht[m].LChild;  
 } **else if** (ch[i] == '1') {  
 m = ht[m].RChild;  
 }  
 **if** (ht[m].LChild == 0) {  
 printf("%c", ht[m].data);  
 m = 11;  
 }  
 }  
}