数据结构

实验报告（四）

树和二叉树

学号：3020205015

姓名：石云天

班级：智能机器平台 2 班

日期：2022.11.13

目　录

一、实验内容描述

本次实验主题是树和二叉树，主要包含两个内容：二叉树的基本函数和基础算法、哈夫曼树的构造和编码。下面是详细的实验内容：

（1）实现树的结点 TreeNode（支持线索二叉树）和树 MyTree（支持线索二叉树），采用二叉链表存储，完成以下功能：

- 树结点的初始化和销毁
- 树结点的打印
- 树初始化，初始化一棵空树
- 树初始化，根据二叉树的先序序列，生成二叉树的二叉链表存储,使用@表示 NULL
- 树的复制构造函数,复制参数中的树
- 树销毁，支持普通二叉树与线索二叉树
- 先序遍历二叉树并打印，仅支持普通二叉树，不考虑线索化，对结点的访问操作为打印该结点
- 中序遍历二叉树并打印，支持普通二叉树与线索二叉树，对结点的访问操作为打印该结点
- 后序遍历二叉树并打印，仅支持普通二叉树，不考虑线索化，对结点的访问操作为打印该结点
- 定位二叉树中的结点，在树中找到值为 v 的结点则返回该结点，否则返回 NULL，支持普通二叉树与线索二叉树
- 计算二叉树的叶子结点数，仅支持普通二叉树，不考虑线索化
- 计算二叉树的深度，仅支持普通二叉树，不考虑线索化
- 当前树是否是线索二叉树，是线索二叉树返回 true，否则 false
- 为二叉树生成中序线索二叉树
- 寻找中序线索二叉树中某结点的前驱结点，仅支持线索二叉树
- 寻找中序线索二叉树中某结点的后继结点，仅支持线索二叉树

（2）实现霍夫曼树 HuffmanTree，输出对应的霍夫曼编码

- 树初始化 HuffmanTree，根据输入创建一棵霍夫曼树，第一个参数为结点个数，第二个参数为结点数组，结点值为结点重要度，越大代表越重要，要求树构建时偏小的值放入左子树，偏大的值放入右子树
- 树销毁
- 输出霍夫曼编码，格式：结点值：编码，结点排序递减
- 其他必要的函数

二、实验步骤

（1）根据上课所讲，回顾数和二叉树的基本概念，充分认识到递归在树及其相关算法中的重要意义。因为其非线性的数据结构，除非是满二叉树，否则一般选取链式存储结构（二叉链表），并总结所有关于树的操作，构造抽象数据类型。
（2）仔细阅读实验要求，二叉树和哈夫曼编码的各个操作的主要算法。对于递归过程还应该考虑其非递归算法（利用栈实现）。
（3）利用 CodeBlocks 编译器，配置环境，基于 C++语言将算法用程序实现。
（4）编译运行程序，使用样例进行程序测试，观察所编程序是否实现要求的功能。
（5）考察算法的时间复杂度和空间复杂度，评价算法的优劣，进一步优化程序。
（6）撰写实验报告，进行实验总结与反思。

三、程序设计

（一）抽象数据类型 ADT

二叉树的抽象数据类型：

ADT MyTree{

数据对象：D=$\{a_i|a_i \in char, i = 1,2,3\ldots\}$

数据关系：二叉树的每个结点有唯一的前驱结点（除了根结点），可以有多个后继结点。数据之间是一种非线性关系。每个结点的度不超过 2，并且具有左右顺序。

基本操作：

void InitTree(Tree &*T*)

操作结果：二叉树的初始化函数

void JudgeThreadTree(Tree *T*,*int* &*e*)

操作结果：判断树是普通二叉树还是线索二叉树

int PreCreateTree (Tree &*T*,*char a*[],*int* &*i*,*int m*)

操作结果：先序生成一棵二叉树

void InVisit (Tree &T)

操作结果：中序遍历二叉树

void PreVisit (Tree &*T*)

操作结果：先序遍历二叉树

void PostVisit (Tree &*T*)

操作结果：后序遍历二叉树

void LeafNumber(Tree *T*,*int* &*num*)

操作结果：计算叶子结点个数

int GetHeight(Tree *T*)

操作结果：计算二叉树的高度

void CopyTree(Tree *T*,Tree &*M*)

操作结果：复制一棵二叉树

void CreateTreadTree(Tree &*T*, tNode* &*pre*)

操作结果：二叉树的中序线索化

int FindFrontNode(Tree *T*,tNode* *p*,tNode* &*q*)

操作结果：寻找线索二叉树的前驱结点

int FindNextNode(Tree *T*,tNode* *p*,tNode* &*q*)

操作结果：寻找线索二叉树的后继结点

void DeleteTree(Tree &*T*)

操作结果：销毁二叉树（对于普通二叉树和线索二叉树都适用）

tNode* FindGivenNode *(Tree T,char v)*

操作结果：寻找值为 v 的结点位置（对于普通二叉树和线索二叉树都适用）

}**ADT MyTree**;

哈夫曼树的抽象数据类型：

ADT HuffmanTree{

数据对象：D={$a_i|a_i \in int, i = 1,2,3\ldots$}

数据关系：数据结点之间的关系同树一样，但是哈夫曼树使得树的带权路径长度最小，得到的二叉树称为最优二叉树。

基本操作：

int CreateHuffmanTree(HTree &*H*,*int num*,*int a*[])

操作结果：构造一棵哈夫曼树

int DeleteHuffmanTree(HTree &*H*)

操作结果：销毁一颗哈夫曼树

void HuffmanCode(List &*C*,HTree *H*,*int x*)

操作结果：创建哈夫曼编码表

}**ADT HuffmanTree;**

（二）数据存储结构

在本次实验中共使用到三种数据数据存储结构：

（1）二叉树的数据结构

对于二叉树这种数据结构，由于需要同时表达普通二叉树和线索二叉树两种结构，所以在二叉链表的基础上添加两个 tag 位，具体如下：

```
typedef struct tNode{
    char data;
    int ltag;
    int rtag;
    struct tNode * lchild;
    struct tNode * rchild;
}tNode,*Tree;
```

（2）哈夫曼树的数据结构

对于哈夫曼树的结构，由于根据叶子结点的个数可以计算出所有结点的个数，因此可以采用一种静态的多维数组结构。数组元素应当包括数据、双亲、左右孩子结点信息。具体如下：

```
typedef struct tNode{
    int data;
    int parent;
    int lchild;
    int rchild;
}tNode,*HTree;
```

（3）哈夫曼编码表的数据结构

对于哈夫曼编码表，由于元素个数是固定的，因此可以创建一个静态的数组结果来存储。每个数组元素应该包含标号、数据、编码的字符数组、以及编码位数。具体如下：

```
typedef struct CodeList{
    int pos;
    int data;
    char code[1000];
    int num;
}CodeNode,*List;
```

（三）算法简述

下面分别对二叉树的构建与基础算法、线索二叉树的构建与遍历算法、哈夫曼树的构建算法与哈夫曼编码表的构建算法的算法思想进行介绍。

一、二叉树的构建与基础算法

由于二叉树的定义即为递归建立的，因此二叉树的建立、二叉树的三种遍历算法（前序、中序、后序）都是通过递归调用实现的。计算二叉树的叶子结点个数、计算二叉树的高度、销毁二叉树、寻找特定结点所在位置等算法都是二叉树遍历算法的扩展应用，也都是通过递归调用实现的。整个递归过程比较简单，首先描述根结点的操作，再递归调用左、右孩子即可。算法细节体现在代码附录中，此处不再过多赘述。

二、线索二叉树的构建与遍历算法

（1）二叉树的中序线索化

首先建立头结点，头结点的 ltag 置为 0，并将其 lchild 指向二叉树的根结点；rtag 置为 1，并将其 rchild 指向自身。然后进行遍历，准备 T 指针和 pre 指针，同样采用递归调用的方式，先调用左孩子，然后处理根结点：将所有的空指针都链接到前驱结点或后继结点，并相应的修改 ltag 和 rtag 的值，最后再调用右孩子。在上述递归过程结束后，还需将最后一个结点的右孩子链接到头结点上，从而实现双向线索化。

（2）线索二叉树的遍历

实验要求获取所需结点的前驱结点和后继结点，这里以获取其前驱结点为例进行说明。如果有线索，那么获取线索二叉树结点的前驱结点是很容易的，直接寻找其左孩子即可得到。如果没有线索，由于线索二叉树是中序建立，因此可以从该结点的左孩子开始向右寻找，直至出现新的有线索的结点。根据中序的规则，这个结点的右孩子一定指向所给的结点，即该结点是给定结点的前驱结点。此处还需考虑边界问题，需要判断该结点是否为第一个结点（左孩子指向为 NULL），如果是则输出“NO FRONT NODE”。获取后继结点与获取前驱结点的算法思想完全相同，不再过多赘述。

三、哈夫曼的构建算法

由于前文已经介绍了哈夫曼树的数据结构，故此处简要介绍如何构建一棵哈夫曼树：

（1）初始化数据结构

若已知有 n 个叶子结点，则哈夫曼树将会有 $2n-1$ 个结点。对于前 n 个结点，将其分别初始化为相应的权值。同时对于所有结点的双亲结点与左右孩子结点均初始化为-1。

（2）选取元素构建哈夫曼编码表

每次取出两个双亲结点数值为-1 且数值最小的结点，将这两个结点求和后作为新的结点添加到表中，并相应修改其双亲结点和左右孩子结点（对于数组，其下标即可发挥指针的作用），直至该表全部填满则停止。

四、哈夫曼编码表的构建算法

在构建出一棵哈夫曼树的基础上，我们需要构建出对应结点的哈夫曼编码表。根据实验输出要求，首先将各个初始结点按照降序排列进存储的静态表中。然后对于每一个结点，使用两个 int 型变量维护该元素当前的下标和其双亲结点的数组下标。对于每个结点，寻找其双亲结点，如果该结点与双亲结点的左孩子匹配，则编码为 0；如果该结点与双亲结点的右孩子匹配，则编码为 1。循环往复，直至找到根结点为止。注意此时是从叶子结点逆序溯源到根结点，是逆序的哈夫曼编码。因此需要对该编码数组再进行逆序输出，最后得到正确的哈夫曼编码。

（四）程序代码

为保证实验报告的清晰和可读性，将源代码以**附录形式**附于文末。

四、调试分析

（一）调试过程和主要错误

在程序编写完成后，使用 CodeBlocks 编译并运行。发现初代程序主要存在以下两方面问题：

（1）MyTree 程序调试过程中出现错误

在构建线索二叉树的过程中遇到了问题：构造过程中需要初始化一个头结点，但是这个结点并不存储任何的信息和权值，因此我最初想初始化一个头结点，随后记录其地址，在线索二叉树构建完成后再释放该结点，但是程序一直报错 Segmentation Fault。这说明程序出现了空指针访问元素的情况，再反复调试过程中也没能解决。随后通过查阅资料，我发现头结点其实并不需要释放，而是可以形成类似于双向链表的数据结构。头结点的 ltag 置为 0，指向根结点，

rtag 置为 1，指向最后一个结点。与此同时，最后一个结点的 rchild 指针指向头结点，实现了一种类似于循环链表的连接。在修改后，就可以顺利运行程序。并且这种建立线索二叉树的算法同时也为判断一棵树是否为线索二叉树提供了一种快速的判断方法——考察头结点的 ltag 值。若 T->ltag 等于 1，即为线索二叉树，若等于 0 就是普通二叉树。

（2）HuffmanTree 程序调试过程中出现问题

由于二叉树的左右结点是有序的，习惯上，我们将较小的结点作为左孩子，将较大的结点作为右孩子，但是并没有充分考虑到结点相等时的排序稳定性问题。这是我们在算法设计过程中没有考虑到的方面。因此重新审视输出问题，发现实验要求中也并未给出有关的要求，所以严格意义上来说两种输出结果（等值分支分别在上层左、右结点上）都是可以接受的。但是为了保证输出的规范性，在编程过程中还是应确保排序的稳定性。此问题只在构建过程中发现两个相同权值的结点时才会出现。

（二）时间复杂度

对于二叉树遍历算法，每个结点至多被访问一次，因此其时间复杂度为 O(n)，而二叉树删除算法、计算叶子结点个数、计算二叉树高度等算法在本质上均为二叉树遍历算法的变形应用，因此时间复杂度也为 O(n)。

哈夫曼树的构建算法首先需要对 $2n-1$ 个结点进行初始化，再对后$n-1$ 个结点进行填充，在填充过程中，需要访问前面所有结点以查找最小的两个元素，此为双层循环嵌套，故其时间复杂度为 O(n^2)。对于哈夫曼编码表的构建算法，首先设其有 n 个叶子结点，故哈夫曼树的平均高度可认为是 $\log(2n-1)$，由于根结点需要到达每个叶子结点一次，因此可以认为该算法的时间复杂度为 $O(n\log)$ 。

五、程序测试

在完成全部程序编写后，输入测试样例进行测试，所得结果均满足要求。树和二叉树基本算法测试结果见图 1，哈夫曼树及哈夫曼编码算法测试结果见图 2。

```
输入先序生成树的字符序列:
ABC!!DE!G!!F!!!
先序遍历二叉树:
A B C D E G F
中序遍历二叉树:
C B E G D F A
后序遍历二叉树:
C G E F D B A
复制二叉树

对复制后的二叉树进行先序遍历:
A B C D E G F
计算叶子节点个数:
3
计算树的深度:
5
判断是否为一颗线索二叉树(是则返回1):
0
寻找值为E的节点是否存在:
存在，地址为: 0xfc1b60
对二叉树进行中序线索化
判断是否为一颗线索二叉树(是则返回1):
1
输入一个字符 查找其前驱节点和后序节点:
G
前驱节点是:  E
后继节点是:  D

Process returned 0 (0x0)   execution time : 33.535 s
Press any key to continue.
```

图 1 树和二叉树基本算法测试结果

```
6
7 11 2 5 10 8
11 : 10
10 : 01
8 : 00
7 : 111
5 : 1101
2 : 1100

Process returned 0 (0x0)   execution time : 25.218 s
Press any key to continue.
```

图 2 哈夫曼树及哈夫曼编码算法测试结果

六、实验总结

通过这次试验，我发现我对树和二叉树这一部分的理解不够深入全面，需要不断巩固学习，加深理解。树与前面所学的数据结构不同，其为非线性结构，在实现过程中，为同时表达普通二叉树与线索二叉树，需要在普通二叉链表的基础上增设 ltag 与 rtag 两个标志位。同时。在编程过程中需要完成某些特定目标时，我不能很快的想出其对应的操作，需要课下不断练习以熟能生巧，还可以多查阅一些资料以开阔自己的思路。在本次实验中，我编写并实现了（线索）二叉树以及哈夫曼树的各项基本操作，同时利用以上数据结构实现了输出哈夫曼编码算法，在此过程中不断调试，寻找问题，并不断简化代码，提升函数执行速度。除此之外，在本次实验过程中，编写、调试程序花费了很长时间：首先是递归算法的编写，由于之前对于递归算法的思想理解不够深刻，起初使用起来较为生疏，错误频出，经过不断练习，我发现只要理清各部分之间的关系就可以很轻松的完成，但是递归算法在运行时并不会提高效率，因此需要考虑一些非递归的实现方式，需要后续进一步深入思考。其次是对于空指针的理解更加深刻，我最初以为 Segmentation Fault 类型报错是因为数组越界，经过不断查阅资料发现是因为调用了空指针，空指针不会指向任何实体，因此在程序编写过程中需要格外注意各指针变量指向的变化，在 delete 操作完成后，最好在后面加一行将指针置为 NULL 的代码，这可以有效避免调用空指针的错误。在本次实验后，我还需要精益求精，不断改进程序，优化函数性能，实现预期目标与功能所需。

附录 1：程序源代码：MyTree

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

//利用二叉链表进行存储
//添加两个 tag 位以同时支持二叉树和二叉线索树，有线索时置 1，无线索时置 0
typedef struct tNode{
    char data;
    int ltag;
    int rtag;
    struct tNode * lchild;
    struct tNode * rchild;
}tNode,*Tree;

//判断一棵树是否为线索二叉树，若为线索二叉树，则 e=1，反之 e=0
void JudgeThreadTree(Tree T,int &e){
    e = T->rtag;
}

//树初始化 构造一棵空树
void InitTree(Tree &T){
    T = NULL;
}

//先序生成一颗二叉树（递归），空结点用!表示
int PreCreateTree(Tree &T,char a[],int &i,int m){
    if(a[i]!='!'){
        T = new tNode;
        T->data = a[i];
        T->ltag = 0;
        T->rtag = 0;
        i++;
        PreCreateTree(T->lchild,a,i,m);
        PreCreateTree(T->rchild,a,i,m);
    }
```

```
    else{
        T = NULL;
        i++;
    }
    if(i > m-1) return 0;
}

void Visit(Tree T){
    if(T->data != '!') cout << T->data << ' ';
    //此判断语句无实际意义
}

//先序遍历二叉树(递归)
void PreVisit(Tree &T){
    if(T){
        Visit(T);
        PreVisit(T->lchild);
        PreVisit(T->rchild);
    }
}

//中序遍历二叉树（递归）
void InVisit(Tree &T){
    if(T){
        InVisit(T->lchild);
        Visit(T);
        InVisit(T->rchild);
    }
}

//后序遍历二叉树（递归）
void PostVisit(Tree &T){
    if(T){
        PostVisit(T->lchild);
        PostVisit(T->rchild);
```

```
        Visit(T);
    }
}

//查找对应值为 v 的结点（面向普通二叉树）
void FindNode(Tree T,char &e,char v){
    if(T != NULL){
        if(T->data == v) e = 1;
        FindNode(T->lchild,e,v);
        FindNode(T->rchild,e,v);
    }
}

//求叶子结点的个数（递归）
void LeafNumber(Tree T,int &num){
    if(T){
        if((T->lchild == NULL) && (T->rchild == NULL)) num++;
        LeafNumber(T->lchild,num);
        LeafNumber(T->rchild,num);
    }
}

//求二叉树的高度（递归）
int GetHeight(Tree T){
    if(T == NULL) return 0;
    else{
        int l = GetHeight(T->lchild);
        int r = GetHeight(T->rchild);
        return (l>r) ? l+1 : r+1;
    }
}

//复制一颗二叉树（递归）
void CopyTree(Tree T,Tree &M){
    if(T){
```

```
        M = new tNode;
        M->data = T->data;
        M->ltag = M->rtag = 0;
        M->lchild = NULL;
        M->rchild = NULL;
        CopyTree(T->lchild,M->lchild);
        CopyTree(T->rchild,M->rchild);
    }
}

//实现二叉树的中序线索化，使用前需要初始化一个开始结点
void CreateTreadTree(Tree &T, tNode* &pre){
    if(T){
        CreateTreadTree(T->lchild,pre);
        if(!T->lchild){
            T->lchild = pre;
            T->ltag = 1;
        }
        if(!pre->rchild){
            pre->rchild = T;
            pre->rtag = 1;
        }
        pre = T;
        CreateTreadTree(T->rchild,pre);
    }
}

void CreateTreadTree2(Tree &ThreadT,Tree T){
    ThreadT = new tNode;
    ThreadT->ltag = 0;
    ThreadT->rtag = 1;
    ThreadT->rchild = ThreadT;
    ThreadT->lchild = T;
    tNode* pre = ThreadT;
```

```
    CreateTreadTree(T,pre);

    pre->rchild = ThreadT;
    pre->rtag = 1;
    ThreadT->rchild = pre;
}

//寻找线索二叉树的前驱结点，返回前驱结点的指针
int FindFrontNode(Tree T,tNode* p,tNode* &q){
    if(p->ltag == 1) q = p->lchild;
    else{
        tNode* tem = p->lchild;
        while(tem->rtag == 0){
            tem = tem->rchild;
        }
        q = tem;
    }
    if(q->data == NULL) cout << "NO Front Node!" << endl;


}

//寻找线索二叉树的后继结点 返回后继结点的指针
int FindNextNode(Tree T,tNode* p,tNode* &q){

    if(p->rtag == 1) q = p->rchild;
    else{
        tNode* tem = p->rchild;
        while(tem->ltag == 0){
            tem = tem->lchild;
        }
        q = tem;
    }
    if(q->data == NULL) cout << "NO Next Node!" << endl;
}
```

```
//销毁二叉树（递归），面向普通二叉树和线索二叉树
void DeleteTree(Tree &T){
    int e;
    JudgeThreadTree(T,e);
    //面向普通二叉树
    if(!e){
        if(T){
            DeleteTree(T->lchild);
            DeleteTree(T->rchild);
            delete T;
        }
    }
    //面向线索二叉树(中序生成)
    else{
        tNode* p = T;
        while(p->lchild != NULL) p = p->lchild;
        tNode* q;
        while(FindNextNode(T,p,q)){
            delete p;
            p = q;
        }
        delete p;
        p = NULL;
    }
}

//查找对应值为 v 的结点（递归）,返回对应结点的指针（此处不考虑有多个 v 的情况）
void FindNode(Tree T,tNode* &poi,char v){
    if(T){
        if(T->data == v) poi = T;
        FindNode(T->lchild,poi,v);
        FindNode(T->rchild,poi,v);
    }
}
```

```
void FindNode2(Tree T,tNode* &poi,char v){
    int e = 0;
    JudgeThreadTree(T,e);
    //面向普通的二叉树
    if(!e){
        FindNode(T,poi,v);
    }
    //面向中序生成的线索二叉树
    else{
        tNode* p = T->lchild;
        while(p != T){
            while(p->ltag == 0) p = p->lchild;
            if(p->data == v) poi = p;
            while(p->rtag == 1 && p->rchild != T){
                p = p->rchild;
                if(p->data == v) poi = p;
            }
            p = p->rchild;
        }
    }
}

tNode* FindGivenNode(Tree T,char v){
    tNode* poi = NULL;
    FindNode2(T,poi,v);
    return poi;
}

int main(){
    char a[100];
    cout << "输入先序生成树的字符序列: " << endl;
    cin >> a;
    int m = 0;//统计字符串的长度
    for(int i = 0;a[i] != '\0';i++){
```

```
    m++;
}
int k = 0;
Tree T;
Tree M;
PreCreateTree(T,a,k,m);//先序生成树 T

cout << "先序遍历二叉树：" << endl;
PreVisit(T);
cout << endl;
cout << "中序遍历二叉树：" << endl;
InVisit(T);
cout << endl;
cout << "后序遍历二叉树：" << endl;
PostVisit(T);
cout << endl;
cout << "复制二叉树" << endl;
CopyTree(T,M);
cout << endl;
cout << "对复制后的二叉树进行先序遍历：" << endl;
PreVisit(M);
cout << endl;

int x;
cout << "计算叶子结点个数：" << endl;
LeafNumber(T,x);
cout << x << endl;
cout << "计算树的深度：" << endl;
cout << GetHeight(T) << endl;

int e=0;
cout << "判断是否为一颗线索二叉树(是则返回 1)：" << endl;
JudgeThreadTree(T,e);
cout << e << endl;
```

```
    cout << "寻找值为 E 的结点是否存在：" << endl;
    if(FindGivenNode(T,'E'))  cout << "存在，地址为：" <<
FindGivenNode(T,'E') << endl;
    else cout << "不存在" << endl;

    tNode* G;
    cout << "对二叉树进行中序线索化" << endl;
    CreateTreadTree2(G,T);

    cout << "判断是否为一颗线索二叉树(是则返回 1)：" << endl;
    JudgeThreadTree(G,e);
    cout << e << endl;

    cout << "输入一个字符 查找其前驱结点和后序结点：" << endl;
    char t;
    cin>>t;

    tNode* given = FindGivenNode(G,t);
    tNode* fron;
    tNode* nex;
    FindFrontNode(G,given,fron);
    FindNextNode(G,given,nex);
    cout << "前驱结点是：  " << fron->data << endl;
    cout << "后继结点是：  " << nex->data << endl;
}
```

附录 2：程序源代码：HuffmanTree

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

//哈夫曼树的数据结构
typedef struct tNode{
    int data;
    int parent;
    int lchild;
    int rchild;
}tNode,*HTree;

//创建哈夫曼树
int CreateHuffmanTree(HTree &H,int num,int a[]){
    if(num == 0){
        cout << "Empty Tree" << endl;
        return 0;
    }
    int sum = 2 * num -1;
    H = new tNode[sum];//利用静态存储哈夫曼树

    //初始化存储结构
    for(int i = 0;i < num;i++){
        H[i].data = a[i];
        H[i].parent = -1;
        H[i].lchild = -1;
        H[i].rchild = -1;
    }
    for(int i = num;i < sum;i++){
        H[i].data = 0;
        H[i].parent = -1;
        H[i].lchild = -1;
        H[i].rchild = -1;
    }
```

```
    //构造哈夫曼树，若涉及到树的结点个数小于 2，那么程序报错终止
    int x = num;
    while(x < sum){
        int m1=1000000;//代表稍大一点的数
        int m2=100000;//代表稍小一点的数
        int p1,p2;
        for(int i = 0;i < x;i++){
            if(H[i].parent == -1){
                if(H[i].data <= m2){
                    p1 = p2;
                    m1 = m2;
                    m2 = H[i].data;
                    p2 = i;
                }
                else if(H[i].data < m1){
                    p1 = i;
                    m1 = H[i].data;
                }
            }
        }
        H[x].data = m1 + m2;
        H[x].lchild = p2;
        H[x].rchild = p1;
        H[p1].parent = x;
        H[p2].parent = x;
        x++;
    }
}

//销毁哈夫曼树，即释放存储哈夫曼树所占用的空间
int DeleteHuffmanTree(HTree &H){
    delete H;
}
```

```
//生成哈夫曼编码，创建编码表结构
typedef struct CodeList{
    int pos;
    int data;
    char code[1000];
    int num;//表示编码位数
}CodeNode,*List;

void HuffmanCode(List &C,HTree H,int x){

    C=new CodeNode[x];
    //初始化编码表，以 data 为关键字实现编码表的降序排列
    for(int i = 0;i < x;i++){
        C[i].pos = i;
        C[i].data = H[i].data;
    }
    //获取每个元素的哈夫曼编码（顺序相反）
    for(int i = 0;i < x;i++){
        int temp = C[i].pos;
        int pre;
        C[i].num = 0;
        while(H[temp].parent != -1){
            pre = temp;
            temp = H[temp].parent;
            if(H[temp].lchild == pre){
                C[i].code[C[i].num] = '0';
                C[i].num++;
            }
            else if(H[temp].rchild == pre){
                C[i].code[C[i].num] = '1';
                C[i].num++;
            }
        }
       //逆序输出，得到顺序正确的哈夫曼编码
        for(int l = 0;l < C[i].num/2;l++){
```

```
            char tempchar = C[i].code[l];
            C[i].code[l] = C[i].code[C[i].num-1-l];
            C[i].code[C[i].num-1-l] = tempchar;
        }
    }
}

int main(){
    HTree H;
    int num;
    cin >> num;
    int a[num];
    for(int i = 0;i < num;i++){
        cin >> a[i];
    }
    sort(a,a+num,greater<int>());//建立哈夫曼树
    CreateHuffmanTree(H,num,a);//实现哈夫曼编码
    List C;//建立编码表
    HuffmanCode(C,H,num);

    for(int i = 0;i < num;i++){
        cout << C[i].data << " " << ":" << " ";
        for(int j=0;j<C[i].num;j++)  cout << C[i].code[j];
        cout << endl;
    }
}
```