

# 《数据结构》上机报告

2020 年 11 月 4 日

姓名：马家昱 学号：1950509 班级：计科 1 班 得分：300

实验题目	二叉树	
问题描述	二叉树是 $n$ 个有限元素的集合，该集合或者为空、或者由一个称为根（root）的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成，是有序树。当集合为空时，称该二叉树为空二叉树。在二叉树中，一个元素也称作一个结点。	
基本要求	1. (p1) 实现二叉树的创建与遍历。使用先序序列创建二叉树，并实现先序、中序、后序、层次遍历，并且使用逆中序输出二叉树的树形图。 2. (p2) 判断两棵二叉树是否同构，并计算其深度。 3. (p3) 深入理解二叉树的非递归遍历，具体为操作规律与遍历序列的关系，重构一棵二叉树。 4. 对 127 个 ASCII 码实现赫夫曼编码。	
	已完成基本内容（序号）：	1, 2, 3, 4
选做要求	1. 程序增添适当的注释，程序的书写采用缩进格式。 2. 程序拥有一定的健壮性，对非法的输入有一定的报错信息。	
	已完成选做内容（序号）	1, 2
数据结构设计	<p>本实验所有题目均采用二叉树的数据结构。</p> <p>在二叉树的创建与遍历与非递归遍历中，二叉树均采用三个指针域的链式表示，即数据域、左孩子域和右孩子域。</p> <p>二叉树的同构中，使用静态数组存储二叉树，每个节点包括数据，左孩子与右孩子在数组中的相对位置。</p> <p>赫夫曼编码的应用中，使用静态数组存储二叉树，每个节点包括权重(weight)、双亲（parent）、左孩子与右孩子。</p> <p>二叉树为有限个二叉节点的集合。</p>	

## 1. 二叉树的创建与遍历

二叉树链表储存的基本结构

```
class BiTree { //二叉树的链存储结构
public:
    char data; //数据域
    BiTree* lchild; //左孩子
    BiTree* rchild; //右孩子
public:
    BiTree();
};
```

```
BiTree::BiTree() { //初始化函数
    data = '\0';
    lchild = NULL;
    rchild = NULL;
}
```

先序创建二叉树 BiTree\* preCreate()

```
BiTree* preCreate() { //先序递归建立二叉树
    BiTree* T;
    char elem;
    cin >> elem;
    if (elem == '#') //节点为空
        T = NULL;
    else {
        T = (BiTree*)malloc(sizeof(BiTree));
        T->data = elem;

        T->lchild = preCreate(); //递归建立左子树
        T->rchild = preCreate(); //递归建立右子树
    }
    return T;
}
```

先序、中序与后序遍历

```

void preOrderTraverse(BiTree* T) { //先序遍历
    if (T) {
        cout << T->data;
        preOrderTraverse(T->lchild);
        preOrderTraverse(T->rchild);
    }
}

```

```

void inOrderTraverse(BiTree* T) { //中序遍历
    if (T) {
        inOrderTraverse(T->lchild);
        cout << T->data;
        inOrderTraverse(T->rchild);
    }
}

```

```

void postOrderTraverse(BiTree* T) { //后序遍历
    if (T) {
        postOrderTraverse(T->lchild);
        postOrderTraverse(T->rchild);
        cout << T->data;
    }
}

```

使用队列进行层次遍历

```

void levelOrderTraverse(BiTree* T) { //层次遍历
    BiTree* p;
    queue<BiTree*> Q;
    if (T) {
        Q.push(T);
        while (!Q.empty()) {
            p = Q.front();
            cout << p->data;
            Q.pop();
            if (p->lchild)
                Q.push(p->lchild);
            if (p->rchild)
                Q.push(p->rchild);
        }
    }
}

```

递归进行逆中序遍历，打印树形图

```

void prtbtree(BiTree* p, int cur)//逆中序遍历, 逆时针旋转90度输出二叉树
{
    if (p)
    {
        prtbtree(p->rchild, cur + 1);
        for (int i = 0; i < cur; i++)
            cout << " ";
        cout << p->data;
        cout << '\n';
        prtbtree(p->lchild, cur + 1);
    }
}

```

## 2. 二叉树的同构

序列化二叉树节点类

```

class BiTreeNode { //序列化二叉树类
public:
    char data;
    int lchild;
    int rchild;
};

```

建立一个序列化二叉树, 并返回其根节点在数组中的位置

```

int createTree(BiTreeNode T[]) { //根据输入建立一个二叉树, 返回其根节点
    int root;
    char lc, rc;
    int searchRoot[MAX_NODE_NUM] = { 0 };
    int N;
    cin >> N;
    if (N == 0)
        return null;
    for (int i = 0; i < N; i++) {
        cin >> T[i].data >> lc >> rc;

        if (lc != '-') {
            T[i].lchild = lc - '0';
            searchRoot[T[i].lchild] = 1;
        }
        else T[i].lchild = null;

        if (rc != '-') {
            T[i].rchild = rc - '0';
            searchRoot[T[i].rchild] = 1;
        }
        else T[i].rchild = null; //节点为空
    }

    for (int i = 0; i < N; i++) //当一个节点既不是某个节点的左节点, 也不是某个节点的右节点时, 其为根节点
        if (searchRoot[i] == 0) {
            root = i;
            break;
        }
    return root;
}

```

使用递归判断两颗二叉树是否同构

```

bool isomorphic(BiTreeNode T1[], int root1, BiTreeNode T2[], int root2) { //递归判断是否同构
    if (root1 == -1 && root2 == -1)
        return true;
    if ((root1 != -1 && root2 == -1) || (root1 == -1 && root2 != -1))
        return false;
    if (T1[root1].data != T2[root2].data)
        return false;
    else {
        if (isomorphic(T1, T1[root1].lchild, T2, T2[root2].lchild) && isomorphic(T1, T1[root1].rchild, T2, T2[root2].rchild)) //左右子树交换前是否相同
            return true;
        else if (isomorphic(T1, T1[root1].lchild, T2, T2[root2].rchild) && isomorphic(T1, T1[root1].rchild, T2, T2[root2].lchild)) //左右子树交换后是否相同
            return true;
        else return false;
    }
}

```

递归求其深度

```

int BiTreeDepth(BiTreeNode T[], int root) { //递归搜索树的深度
    int depthl, depthr;
    if (root != -1) {
        depthl = BiTreeDepth(T, T[root].lchild);
        depthr = BiTreeDepth(T, T[root].rchild);
        if (depthl >= depthr)
            return (depthl + 1);
        else return (depthr + 1);
    }
    else return 0;
}

```

### 3. 二叉树的非递归遍历

三种 push 操作，均为向树中增加新节点

```

void pushFisrt(char elem, BiTree* &T, BiTree* &p) { //由于没有头结点，所以第一个节点要特殊处理
    T->data = elem;
    S.push(T);
    p = T;
}

```

```

void pushCmd1(char elem, BiTree* &p) { //两种push操作的区别在于上一个操作时push还是pop
    BiTree* newNode = (BiTree*)malloc(sizeof(BiTree));
    newNode->data = elem;
    p->lchild = newNode;
    S.push(newNode);
    p = p->lchild;
}

```

```

void pushCmd2(char elem, BiTree* &p) {
    BiTree* newNode = (BiTree*)malloc(sizeof(BiTree));
    newNode->data = elem;
    p->rchild = newNode;
    S.push(newNode);
    p = p->rchild;
}

```

两种 pop 操作

```

void popCmd1(BiTree* &p) { //两种pop操作的区别在于上一个操作时push还是pop
    p->lchild = NULL;
    p = S.top();
    S.pop();
}

```

```

void popCmd2(BiTree* &p) {
    p->rchild = NULL;
    p = S.top();
    S.pop();
}

```

#### 4. 赫夫曼编码

```
typedef struct {
    int wight;
    int parent;
    int lChild;
    int rchild;
}HTNode,*HuffmanTree;

typedef char** HuffmanCode;
```

```
HuffmanTree HT = NULL;
```

```
HuffmanCode HC;
```

```
int w[128];
```

选择权重最小的两棵树，并合并成一个树，其权重为两棵树权重之和

```
void select(HuffmanTree HT, int n, int& s1, int& s2) {
    int least=9999;
    int preleast=9999;
    int p = 0;
    int q = 0;
    for (int i = 1; i <= n; i++)
        if (HT[i].parent == 0) {
            if (HT[i].wight <= least) {
                preleast = least;
                q = p;
                least = HT[i].wight;
                p = i;
            }
            else if (HT[i].wight > least&& HT[i].wight <= preleast) {
                preleast = HT[i].wight;
                q = i;
            }
        }
    s1 = p;
    s2 = q;
}
```

生成赫夫曼编码

	<pre> void HuffmanCoding(HuffmanTree&amp; HT, HuffmanCode&amp; HC, int w[], int n) {     if (n &lt;= 1)         return;     int m = 2 * n - 1;     int i = 0;     HuffmanTree p = NULL;     HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));     for (p = HT; i &lt;= n; ++i, ++p)         *p = { w[i], 0, 0, 0 };     for (; i &lt;= m; ++i, ++p)         *p = { 0, 0, 0, 0 };     for (int i = n + 1; i &lt;= m; i++) {         int s1, s2;         select(HT, i - 1, s1, s2);         HT[s1].parent = HT[s2].parent = i;         HT[i].lChild = s1;         HT[i].rChild = s2;         HT[i].wight = HT[s1].wight + HT[s2].wight;     }     HC = (HuffmanCode)malloc((n + 1) * sizeof(char*));     char* cd = (char*)malloc(n * sizeof(char));     cd[n - 1] = '\0';     for (int i = 1; i &lt;= n; i++) {         int start = n - 1;         for (int c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {             if (HT[f].lChild == c)                 cd[--start] = '0';             else cd[--start] = '1';             HC[i] = (char*)malloc((n - start) * sizeof(char));             strcpy(HC[i], &amp;cd[start]);         }     } } </pre>
开发环境	Windows 10 Microsoft visual studio 2019

(运行结果截图)

1. 二叉树的创建与遍历

```
Microsoft Visual Studio 调试控制台
abc##d##ef###
abcdef
cbdafe
cdbfea
abecdf
    e
      f
a      d
    b  c
```

2. 二叉树的同构

```
Microsoft Visual Studio 调试控制台
6
A 1 4
B 2 3
C - -
D - -
E 5 -
F - -
6
B 4 3
F - -
A 5 0
C - -
D - -
E 1 -
Yes
3
3
```

3. 二叉树的非递归遍历

```
Microsoft Visual Studio 调试控制台
6
push a
push b
push c
pop
pop
push d
pop
pop
push e
push f
pop
pop
cdbfea
```

调试分析



#### 4. 赫夫曼编吗

选择Microsoft Visual Studio 调试控制台

```
0's HuffmanCode is:0000001
1's HuffmanCode is:0001000
2's HuffmanCode is:0001001
3's HuffmanCode is:0010001
4's HuffmanCode is:0011000
5's HuffmanCode is:0011001
6's HuffmanCode is:0100001
7's HuffmanCode is:0101000
8's HuffmanCode is:0101001
9's HuffmanCode is:0110001
:'s HuffmanCode is:0111000
;s HuffmanCode is:0111001
<'s HuffmanCode is:1000001
=s HuffmanCode is:1001000
>'s HuffmanCode is:1001001
?'s HuffmanCode is:1010001
@s HuffmanCode is:1010110
A's HuffmanCode is:1010111
B's HuffmanCode is:1011001
C's HuffmanCode is:1011010
D's HuffmanCode is:1011011
E's HuffmanCode is:1011101
F's HuffmanCode is:1011110
G's HuffmanCode is:1011111
H's HuffmanCode is:1100001
I's HuffmanCode is:1100010
J's HuffmanCode is:1100011
K's HuffmanCode is:1100101
L's HuffmanCode is:1100110
M's HuffmanCode is:1100111
```

#### 赫夫曼编码:

#### 5. 赫夫曼编码

```
typedef struct {
    int wight;
    int parent;
    int lChild;
    int rchild;
}HTNode,*HuffmanTree;

typedef char** HuffmanCode;

HuffmanTree HT = NULL;
HuffmanCode HC;
int w[128];
```

选择权重最小的两棵树，并合并成一个树，其权重为两棵树权重之和

```

void select(HuffmanTree HT, int n, int& s1, int& s2) {
    int least=9999;
    int preleast=9999;
    int p = 0;
    int q = 0;
    for (int i = 1; i <= n; i++)
        if (HT[i].parent == 0) {
            if (HT[i].wight <= least) {
                preleast = least;
                q = p;
                least = HT[i].wight;
                p = i;
            }
            else if (HT[i].wight > least&& HT[i].wight <= preleast) {
                preleast = HT[i].wight;
                q = i;
            }
        }
    s1 = p;
    s2 = q;
}

```

生成赫夫曼编码

```

void HuffmanCoding(HuffmanTree& HT, HuffmanCode& HC, int w[], int n) {
    if (n <= 1)
        return;
    int m = 2 * n - 1;
    int i = 0;
    HuffmanTree p = NULL;
    HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));
    for (p = HT; i <= n; ++i, ++p)
        *p = { w[i], 0, 0, 0 };
    for (; i <= m; ++i, ++p)
        *p = { 0, 0, 0, 0 };
    for (int i = n + 1; i <= m; i++) {
        int s1, s2;
        select(HT, i - 1, s1, s2);
        HT[s1].parent = HT[s2].parent = i;
        HT[i].lChild = s1;
        HT[i].rChild = s2;
        HT[i].wight = HT[s1].wight + HT[s2].wight;
    }
    HC = (HuffmanCode)malloc((n + 1) * sizeof(char*));
    char* cd = (char*)malloc(n * sizeof(char));
    cd[n - 1] = '\0';
    for (int i = 1; i <= n; i++) {
        int start = n - 1;
        for (int c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
            if (HT[f].lChild == c)
                cd[--start] = '0';
            else cd[--start] = '1';
            HC[i] = (char*)malloc((n - start) * sizeof(char));
            strcpy(HC[i], &cd[start]);
        }
    }
}

```

将一段文本转化为赫夫曼编码

使用了 STL 标准库中的 map 型数据结构，字符作为 key 值，二进制编码作为 value 值。

```

for (int i = 1; i <= 127; i++) {
    char c = i;
    cout << c << "s HuffmanCode is:" << HC[i] << endl;
    codeMap.insert(pair<char, string>(c, HC[i]));
}
cout << "请输入你想要转换为二进制编码的文本: " << endl;
char S[100];
cin >> S;
for (int i=0;;i++) {
    if (S[i] == '\0')
        break;
    cout << codeMap.at(S[i]);
}

```

将一段二进制编码转化为文本：

从树的根节点开始寻找，若为 0，则向左孩子走，反之向右孩子走，一直走到叶子

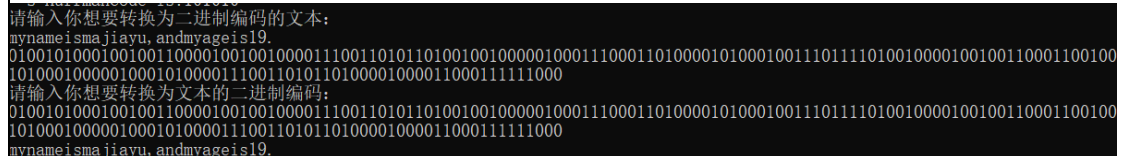
节点为止。

```
cout << "请输入你想要转换为文本的二进制编码：" << endl;
cin >> S;
int temp = 253;
for (int i = 0;; i++) {
    if (S[i] == '\0')
        break;

    if (S[i] == '0')
        temp = HT[temp].lChild;
    else temp = HT[temp].rchild;

    if (HT[temp].lChild == 0 && HT[temp].rchild == 0) {
        char c = temp;
        cout << c;
        temp = 253;
    }
}
```

压缩与解压结果如下：



```
请输入你想要转换为二进制编码的文本:
wnameismajiayu, andmyageis19.
010010100010010011000010010010000111001101011010010010000010001110001101000010100010011101111010010000100100110001100100
101000100000100010100001110011010110100001000011000111111000
请输入你想要转换为文本的二进制编码:
010010100010010011000010010010000111001101011010010010000010001110001101000010100010011101111010010000100100110001100100
101000100000100010100001110011010110100001000011000111111000
wnameismajiayu, andmyageis19.
```

心得体会：

通过此次与二叉树相关的一系列操作，我首先对递归这一常见而重要的算法有了更深入的理解，同时对递归的使用也更加得心应手。无论是二叉树的创建，求深度，判断同构或是线索化，均使用到了递归这一重要的方法。其次，我对二叉树这种相对陌生的结构逐渐熟悉。由于之前没有应用过二叉树，因此在学习理论知识过程中存在一定困难。然而完成这些实验后，在课堂上没有吸收的知识再次被复习和巩固。