数据结构与算法 第二次实验

学号：22920212204392 姓名：黄勖

一、 实验目的

1. 了解二叉树的基础实现方法与原理，理解二叉树的基本操作的代码编写方式

2. 学会灵活按照二叉树的存储内容自由编写二叉树的存储结构

3. 在二叉树的基础上进而理解编写哈夫曼树的基础实现方法，理解哈夫曼树的基本操作的代码编写方式

4. 通过实验探索不同类型的树的相似点与区别，发现在操作实现上的不同

二、 实验内容

2-1 设二叉树的存储结构如下：

```
typedef struct BiTNode
{
Type data;  //数据域；Type: 用户定义数据类型
struct BiTNode *Lchild;  //左指针域
struct BiTNode *Rchild;  //右指针域
} BiTNode, *BiTree;
```

实现二叉树的基本操作和遍历操作。

实验内容：

- ✓ **设计二叉树存储方式**

```
7   typedef char Type;
8   //test input:abc##d#e##f##
9
10  typedef struct BiTNode {
11      Type data;  //数据域；Type: 此处用户定义为char类型
12      struct BiTNode *lchild;  //左指针域
13      struct BiTNode *rchild;  //右指针域
14  } BiTNode, *BiTree;
```

- ✓ **创建二叉树**

```
16  void InitBiTree(BiTree &T) {
17      T = NULL;
18  }
```

```
109  void CreateBiTree(BiTree &T) {
110      Type ch;
111      scanf("%c", &ch);
112      if (ch == '#')
113          T = NULL;
114      else {
115          T = (BiTree)malloc(sizeof(BiTNode));
116          if (!T)
117              exit(-1);
118          T->data = ch;
119          CreateBiTree(T->lchild);
120          CreateBiTree(T->rchild);
121      }
122  }
```

✓ **访问节点**

```
20  void Visit(Type e) {
21      if(e==NULL) {
22          return;
23      }
24      printf("%c ", e);
25  }
```

✓ **删除节点**

```
123  void Delete_Node(BiTree &T)  {
124      if(T) {
125          Visit(T->data);
126          cout << "delete?" << endl;
127          char a;
128          cin >> a;
129          if(a=='y') {
130              T->data = NULL;
131              T->lchild = NULL;
132              T->rchild = NULL;
133              return;
134          }
135          Delete_Node(T->lchild);
136          Delete_Node(T->rchild);
137      }
138  }
```

✓ **释放二叉树**

```
27  void DestoryBiTree(BiTree &T) {
28      if (T) {
29          DestoryBiTree(T->lchild);
30          DestoryBiTree(T->rchild);
31          free(T);
32          T = NULL;
33      }
34  }
```

✓ **前序遍历**

```
36  void PreOrderTraverse(BiTree T) {
37      if (T) {
38          Visit(T->data);
39          PreOrderTraverse(T->lchild);
40          PreOrderTraverse(T->rchild);
41      }
42  }
```

✓ **中序遍历**

```
44  void InOrderTraverse(BiTree T) {
45      if (T) {
46          InOrderTraverse(T->lchild);
47          Visit(T->data);
48          InOrderTraverse(T->rchild);
49      }
50  }
```

✓ **后序遍历**

```
52  void PostOrderTraverse(BiTree T) {
53      if (T) {
54          PostOrderTraverse(T->lchild);
55          PostOrderTraverse(T->rchild);
56          Visit(T->data);
57      }
58  }
```

✓ **层序遍历**

```
60  void LevelOrderTraverse(BiTree T) {
61      queue<BiTree> q;
62      BiTree a;
63      if (T) {
64          q.push(T);
65          while (!q.empty()) {
66              a = q.front();
67              q.pop();
68              Visit(a->data);
69              if (a->lchild != NULL)
70                  q.push(a->lchild);
71              if (a->rchild != NULL)
72                  q.push(a->rchild);
73          }
74          cout << endl;
75      }
76  }
```

✓ **判断二叉树是否为空**

```
78  int BiTreeEmpty(BiTree T) {
79      if (T)
80          return 0;
81      else
82          return 1;
83  }
```

✓ **计算二叉树深度**

```
85  int BiTreeDepth(BiTree T) {
86      int i, j;
87      if (!T)
88          return 0;
89      i = BiTreeDepth(T->lchild);
90      j = BiTreeDepth(T->rchild);
91      return i > j? i + 1: j + 1;
92  }
```

✓ **其他的一些小功能（读根、值、设定值）**

```
94   Type Root(BiTree T) {
95       if (BiTreeEmpty(T))
96           return NULL;
97       else
98           return T->data;
99   }
100
101  Type Value(BiTree p) {
102      return p->data;
103  }
104
105  void Assign(BiTree p, Type value) {
106      p->data = value;
107  }
```

✓ **主函数交互**

```
139  int main() {
140      BiTree T;
141      InitBiTree(T);
142      CreateBiTree(T);
143      cout << endl << "前序遍历: " << endl;
144      PreOrderTraverse(T);
145      cout << endl << "中序遍历: " << endl;
146      InOrderTraverse(T);
147      cout << endl << "后序遍历: " << endl;
148      PostOrderTraverse(T);
149      cout << endl << "层序遍历: " << endl;
150      LevelOrderTraverse(T);
151      Delete_Node(T);
152      cout << endl << "前序遍历: " << endl;
153      PreOrderTraverse(T);
154      cout << endl << "中序遍历: " << endl;
155      InOrderTraverse(T);
156      cout << endl << "后序遍历: " << endl;
157      PostOrderTraverse(T);
158      cout << endl << "层序遍历: " << endl;
159      LevelOrderTraverse(T);
160      return 0;
161  }
```

2-2 设计算法实现：

(1) 构造哈夫曼树；

(2) 求解哈夫曼编码。

实验内容：

✓ 哈夫曼结点，放在一个数组中，即 HNodeType HuffNodes[]

```c
typedef struct {        //Huffman树结点结构体
    float weight;       //结点权值，这里是字符出现的频率，及频次/字符种类数
    int parent;         //父结点位置索引，初始-1
    int lchild;         //左孩子位置索引，初始-1
    int rchild;         //右孩子位置索引，初始-1
} HNodeType;
```

✓ 哈夫曼编码结构，也采用顺序存储结构（数组）

```c
typedef struct {        //Huffman编码结构体
    int bit[MAXBIT];    //字符的哈夫曼编码
    int start;          //该编码在数组bit中的开始位置
} HCodeType;
```

✓ 接受字符串

```c
void str_input(char str[]) {
    //输入可包含空格的字符串，输入字符串存放在str中
    gets(str);
}
```

✓ 统计字符频次

```c
int TextStatistics(char text[], char ch[], float weight[]) {
    //统计每种字符的出现频次，返回出现的不同字符的个数
    //出现的字符存放在ch中，对应字符的出现频次存放在weight中
    int text_index = 0;  //text字符串索引
    int ch_index = 0;    //计字符数组增加索引，仅用于出现不同字符时，将该字符加入到ch[]中。仅自增
    int weight_index = 0;//频数更新索引。用于指定weight[]要更新频数的位置
    while(text[text_index]!='\0') {
        //查找ch中，是否存在字符text[text_index]，返回查到的第一个字符的位置
        char* pos = strchr(ch,text[text_index]);
        //如果ch中无该字符。或者ch为空。就将text[text_index]加入到ch中
        if(ch[0]==NULL  || pos == NULL ) {
            //加入到统计字符数组中
            ch[ch_index] = text[text_index];
            //添加一个字符的频数，当所有字符都统计完之后再计算频率
            weight[ch_index] += 1;
            ch_index++;
        }
        //如果字符串中有该字符
        else {
            //找到该字符的索引位置，更新其频数
            weight_index = pos - ch ;
            weight[weight_index] += 1;
        }
        text_index++;
    }
    ch[ch_index] = '\0';//添加结束符
    //根据频数计算频率
    int index=0;
    while(weight[index]!=0) {
        weight[index]/=text_index;
        index++;
    }
    //最终 ch_index的值即为text字符串中不同字符的个数
    return ch_index;
}
```

✓ 找到权值最小的两个结点

```
// 从 HuffNodes[0..range]中，找到最小的结点索引赋给s1,s2 。已经找到过的结点索引被储存在out[]中
void select(HNodeType HuffNodes[],int range,int *s1,int *s2) {
    //先找第一个最小值 。
    float min1 = 5;
    for(int index1=0; index1<=range; index1++) {
        if(HuffNodes[index1].weight < min1 && HuffNodes[index1].parent ==-1) {
            //判断该结点是否被选过。如果该结点parent为0，则其为被选
            min1 = HuffNodes[index1].weight;
            *s1 = index1 ;
        }
    }
    //找第2个最小值
    float min2 = 5;
    for(int index2=0; index2<=range ; index2++) {
        if(HuffNodes[index2].weight < min2 && HuffNodes[index2].parent ==-1 && index2!=*s1) {
            //判断该结点是否被选过。还要判断其是否被s1选了
            min2 = HuffNodes[index2].weight;
            *s2 = index2 ;
        }
    }
}
```

✓ 构造哈夫曼树

```
void HuffmanTree(HNodeType HuffNodes[], float weight[], int n) {
    if(n>MAXLEAF) {
        printf("超出叶结点最大数量!\n");
        return;
    }
    if(n<=1) return;
    int m = 2*n-1;//结点总个数
    int node_index = 0;
    //构造各叶节点
    for(; node_index < n; node_index++) {
        HuffNodes[node_index].weight = weight[node_index];
        HuffNodes[node_index].parent = -1;
        HuffNodes[node_index].lchild = -1;
        HuffNodes[node_index].rchild = -1;
    }
    //构造非叶节点
    for(; node_index<m; node_index++) {
        HuffNodes[node_index].weight = 0;
        HuffNodes[node_index].parent = -1;
        HuffNodes[node_index].lchild = -1;
        HuffNodes[node_index].rchild = -1;
    }
    //构建Huffmantree
    int s1,s2;//最小值索引
    for(int i = n; i < m; i++) {
        select(HuffNodes,i-1,&s1,&s2);
        HuffNodes[s1].parent = i;
        HuffNodes[s2].parent = i;
        HuffNodes[i].lchild = s1;
        HuffNodes[i].rchild = s2;
        HuffNodes[i].weight = HuffNodes[s1].weight + HuffNodes[s2].weight;
    }

}
```

✓ 生成哈夫曼编码

```
113  void HuffmanCode(HNodeType HuffNodes[], HCodeType HuffCodes[], int n) {
114      //生成Huffman编码，Huffman编码存放在HuffCodes中
115      int start;
116      for(int i =0 ; i<n; i++) {
117          start = n-2;
118          for(int c = i , f=HuffNodes[i].parent ; f!=-1; c =f,f=HuffNodes[f].parent) {
119              if(c == HuffNodes[f].lchild) HuffCodes[i].bit[start--]=0;
120              else HuffCodes[i].bit[start--]=1;
121          }
122          HuffCodes[i].start = start+1;
123      }
124  }
```

✓ 遍历哈夫曼树，使用队列递归方式，在本次实验中使用的中序遍历

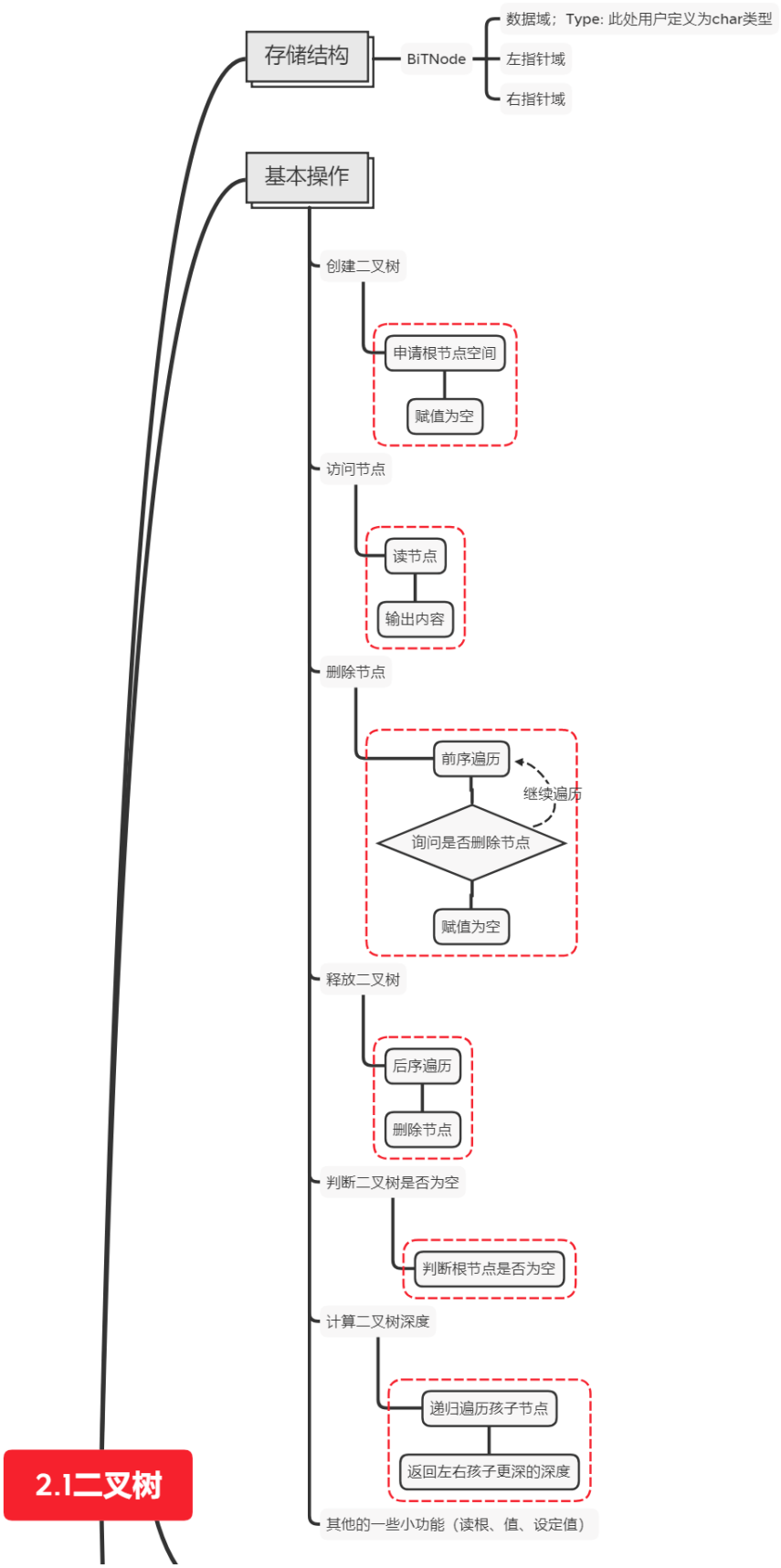
```
125  int MidOrderTraverse(HNodeType HuffNodes[], float result[], int root, int resultIndex) {
126      //Huffman树的中序遍历，遍历结果存放在result中，返回下一个result位置索引
127      //根节点 为root
128      if (root!=-1) {
129          resultIndex = MidOrderTraverse( HuffNodes,result,HuffNodes[root].lchild,resultIndex);
130          result[resultIndex++] = HuffNodes[root].weight;
131          resultIndex = MidOrderTraverse( HuffNodes,result,HuffNodes[root].rchild,resultIndex);
132      }
133      return resultIndex;
134  }
```

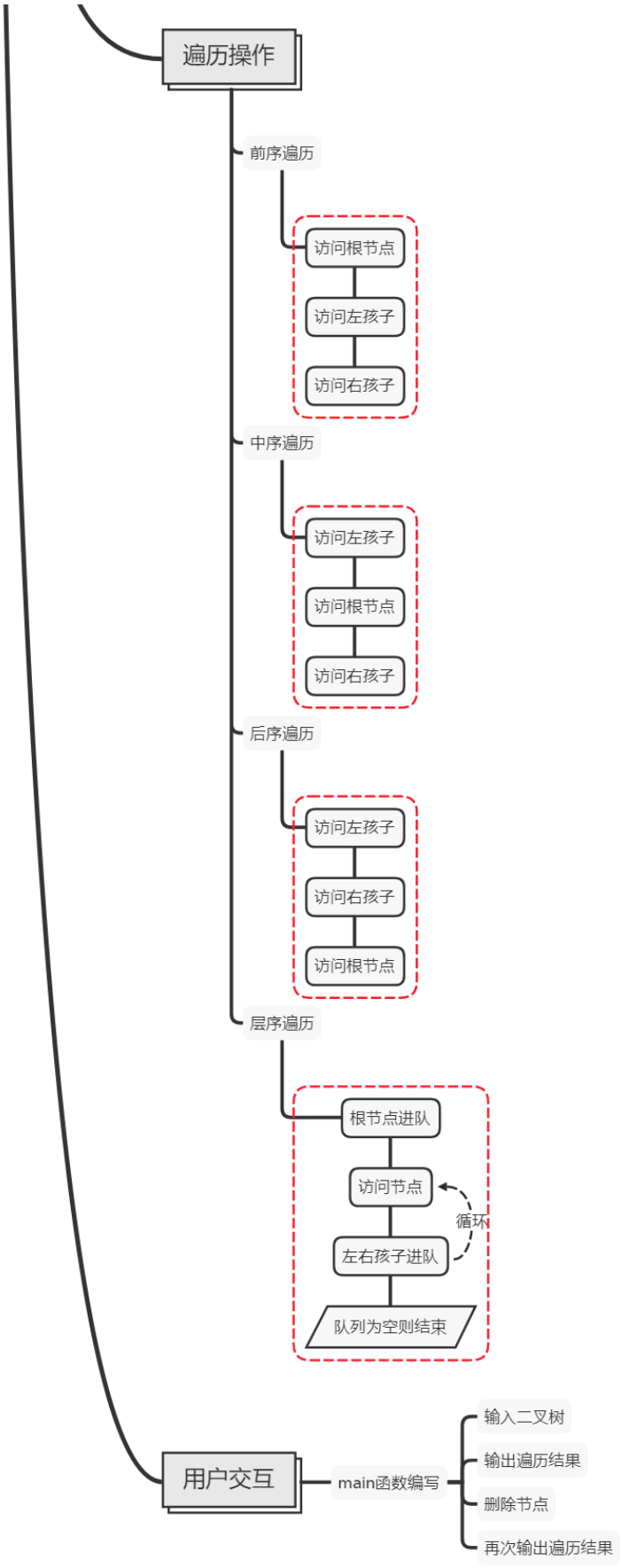
✓ 主函数交互

```
135  int main() {
136      HNodeType HuffNodes[MAXNODE];   // 定义一个结点结构体数组
137      HCodeType HuffCodes[MAXLEAF];   // 定义一个编码结构体数组
138      char text[MAXVALUE+1], ch[MAXLEAF];
139      float weight[MAXLEAF], result[MAXNODE];
140      int i, j, n, resultIndex;
141      str_input(text);
142      //字符总数n
143      n = TextStatistics(text, ch, weight);
144      // 输出哈夫曼编码
145      HuffmanTree(HuffNodes, weight, n);
146      HuffmanCode(HuffNodes, HuffCodes, n);
147      for (i=0; i<n; i++) {
148          printf("%c的Huffman编码是: ", ch[i]);
149          for(j=HuffCodes[i].start; j<n-1; j++)
150              printf("%d", HuffCodes[i].bit[j]);
151          printf("\n");
152      }
153      // 输出Huffman树的中序遍历结果
154      resultIndex = MidOrderTraverse(HuffNodes, result, 2*n-2, 0);
155      printf("\nHuffman树的中序遍历结果是: ");
156      for (i=0; i<resultIndex; i++)
157          if (i < resultIndex-1)
158              printf("%.4f, ", result[i]);
159          else
160              printf("%.4f\n", result[i]);
161  }
```

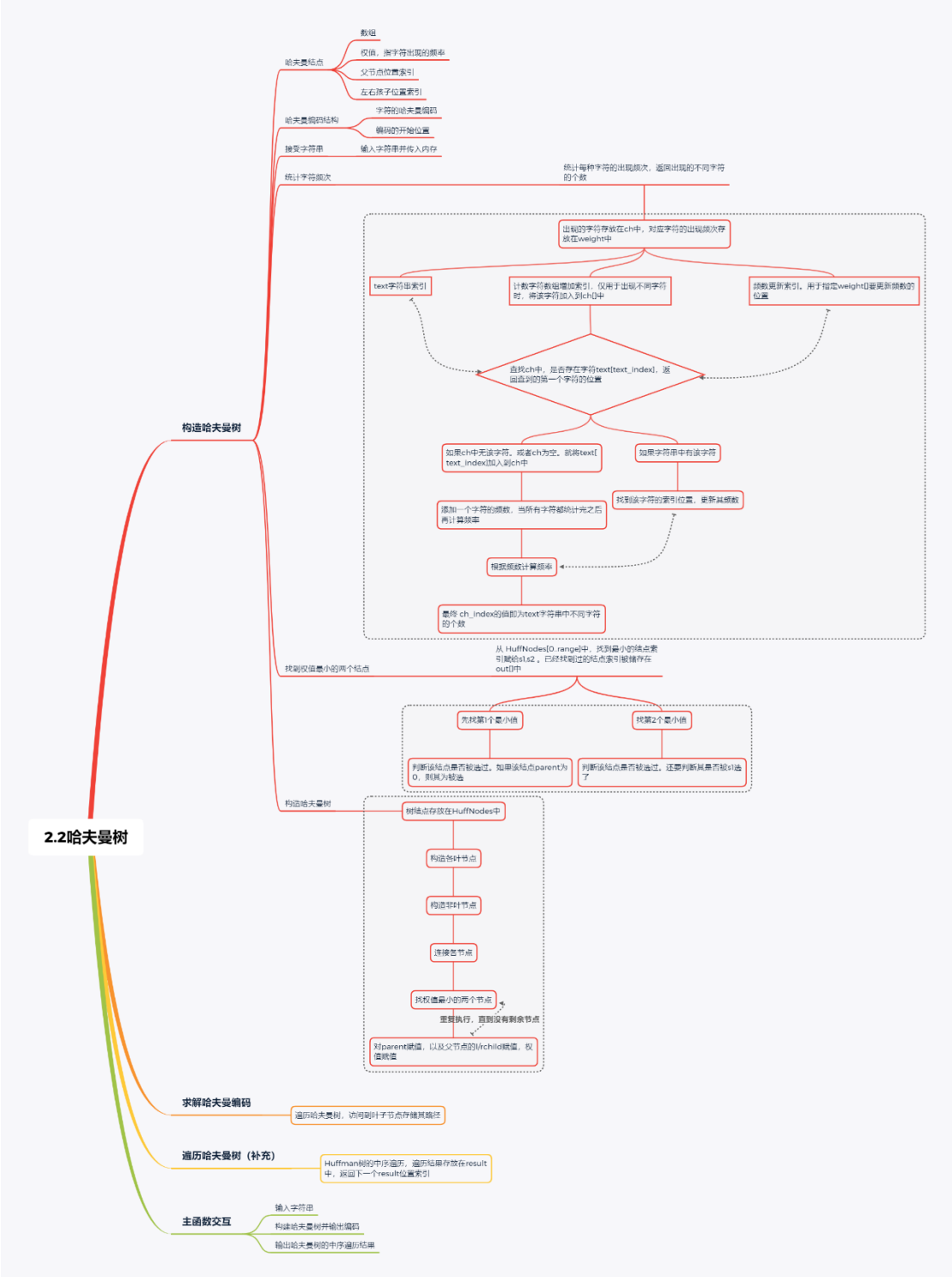
三、 主要算法流程图

2-1 算法流程（图片太小可见目录下的 2-1 二叉树.png）




遍历操作
前序遍历
访问根节点
访问左孩子
访问右孩子
中序遍历
访问左孩子
访问根节点
访问右孩子
后序遍历
访问左孩子
访问右孩子
访问根节点
层序遍历
根节点进队
访问节点
循环
左右孩子进队
队列为空则结束
用户交互
main函数编写
输入二叉树
输出遍历结果
删除节点
再次输出遍历结果

2-2 算法流程



四、 实验结果：

（结合截图说明算法的输入输出）

1、关于 2-1 的输入与输出：

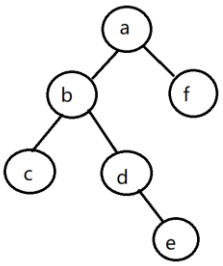
```
E:\大二上\数据结构与算法\实验\实验2_22920212204392_黄勖\2.1.exe
abc##d#e##f##

前序遍历:
a b c d e f
中序遍历:
c b d e a f
后序遍历:
c e d b f a
层序遍历:
a b f c d e
```

```
a delete?
n
b delete?
n
c delete?
n
d delete?
n
e delete?
y
f delete?
n
```

```
前序遍历:
a b c d f
中序遍历:
c b d a f
后序遍历:
c d b f a
层序遍历:
a b f c d
```

在实际运行中，我创建了



这样一个二叉树，并在后续删除了 e，可以看出遍历的结果都是正确的。

2、关于 2-2 的输入与输出

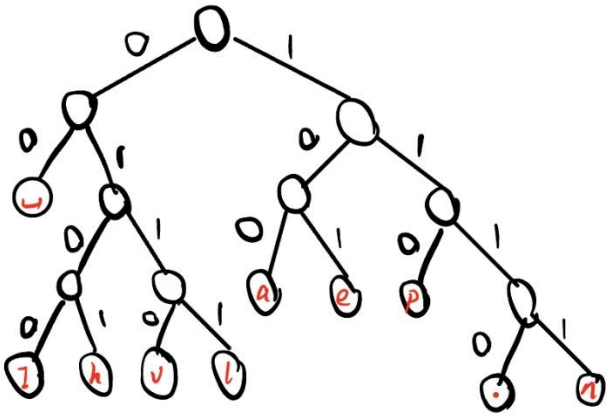
```
E:\大二上\数据结构与算法\实验\实验2_22920212204392_黄勖\2.2.exe
I have an apple pen.
I的Huffman编码是: 0100
 的Huffman编码是: 00
h的Huffman编码是: 0101
a的Huffman编码是: 100
v的Huffman编码是: 0110
e的Huffman编码是: 101
n的Huffman编码是: 1111
p的Huffman编码是: 110
l的Huffman编码是: 0111
. 的Huffman编码是: 1110

Huffman树的中序遍历结果是: 0.2000, 0.4000, 0.0500, 0.1000, 0.0500, 0.2000, 0.0500, 0.1000,
 0.0500, 1.0000, 0.1500, 0.3000, 0.1500, 0.6000, 0.1500, 0.3000, 0.0500, 0.1500, 0.1000

--------------------------------

Process exited after 13.53 seconds with return value 0
请按任意键继续. . .
```

在实际运行中，我创建了



这样的哈夫曼树，并且利用中序遍历得到了每个节点的值（即出现概率）。

五、 实验小结（即总结本次实验所得到的经验与启发等）：

在本次实验中，我尝试具体运用了二叉树，在实体机的实验中我能够更深刻地理解对这一部分数据结构的执行方式与特点，并且在编写代码的过程中，我通过不断的调试去寻找语句之间的问题和不足，在潜移默化中提高了我的代码编写能力，这是一次完成效果良好的实验！