《数据结构与算法实验》第 10 次实验

学院： 专业： 年级：
姓名： 学号： 日期： 2022 年 5 月 26 日

第一部分 验证实验

一、 实验目的

1. 掌握二叉树的线索链表。

二、 实验内容

1. 线索链表（参考课本 p121）

• 按照某种遍历次序对二叉树进行遍历，可以把二叉树中的所有结点排成一个线性序列。在具体应用中，有时需要访问二叉树中的结点在某种遍历序列中的前去和后继，此时，在存储结构中应保存结点在某种遍历序列中的前驱和后继信息。考虑到一个具有 n 个结点的二叉链表，在 2n 个指针域中，只有 n-1 个指针域用来存储孩子结点的地址，存在着 n+1 个空指针域，可以利用这些空指针域存放指向该结点在某种遍历序列中的前驱和后继结点的指针。这些指向前驱和后继结点的指针称为线索，加上线索的二叉树称为线索二叉树。相应地，加上线索的二叉链表称为线索链表。

• 本次验证实验，我们主要验证的是中序线索化链表的相关操作，核心的函数有：建立二叉链表（带线索标志）算法、中序线索化链表算法、中序线索链表构造函数算法、中序线索链表查找后继算法、中序线索链表的遍历算法。

三、 设计编码

1. 线索链表

(1) InThr.h

```
#ifndef INTHR_H
#define INTHR_H

enum flag{Child, Thread};

template <typename T1>
```

```
struct ThrNode
{
    T1 data;
    ThrNode<T1> * lchild, * rchild;
    flag ltag, rtag;
};

template <class DataType>
class InThrBitree
{
    public:
        InThrBitree();                                  //构造函数，建立中序线索链表
        ~InThrBitree();                                 //析构函数，释放结点存储空间
        ThrNode<DataType> * Next(ThrNode<DataType> * p);//查找 p 的后继
        void InOrder();
    private:
        ThrNode<DataType> * root;
        void Release(ThrNode<DataType> *bt);
        void ThrBitree(ThrNode<DataType> * &bt, ThrNode<DataType> * &pre);
        ThrNode<DataType> * Creat(ThrNode<DataType> * bt);
};

#endif
```

(2) InThr. cpp

```
#include <iostream>
#include "InThr.h"
using namespace std;

template<typename T>
InThrBitree<T>::InThrBitree()       //建立二叉树
{
    ThrNode<T> * pre;
    root = NULL;
    root = Creat(root);
    pre = NULL;
    ThrBitree(root, pre);
}

template<typename T>
InThrBitree<T>::~InThrBitree()
{
    Release(root);
```

```
}

template<typename T>
void InThrBitree<T>::Release(ThrNode<T> * bt)
{
    if (bt != NULL)
    {
        if (bt -> ltag == 0)  Release(bt -> lchild);
        if (bt -> rtag == 0)  Release(bt -> rchild);
        delete bt;
    }
}

template <typename T>
ThrNode<T> * InThrBitree<T>::Creat(ThrNode<T> * bt)
{
    char ch;
    cout<<"请输入创建一棵二叉树的结点数据："<<endl;
    cin>>ch;
    if (ch == '#')  return NULL;
    else
    {
        bt = new ThrNode<T>;
        bt -> data = ch;
        bt -> lchild = Creat(bt -> lchild);
        bt -> rchild = Creat(bt -> rchild);
    }
    return bt;
}

template <class DataType>
void InThrBitree<DataType> ::ThrBitree(ThrNode<DataType> *&bt, ThrNode<DataType>
*&pre)
{
     if (bt == NULL) return ;
     ThrBitree(bt -> lchild, pre);
     if (bt -> lchild == NULL)
     {
         bt -> ltag = Thread;
         bt -> lchild = pre;
     }
     if (bt -> rchild == NULL) bt -> rtag = Thread;
     if (pre != NULL && pre -> rtag == 1)  pre -> rchild = bt;
     pre = bt;
```

```
    ThrBitree(bt -> rchild, pre);
}

template <class DataType>
ThrNode<DataType> * InThrBitree<DataType>::Next(ThrNode<DataType> * p)
{
    ThrNode<DataType> * q;
    if (p -> rtag == 1)  q = p -> rchild;
    else
    {
        q = p -> rchild;
        while (q -> ltag == 0)
            q = q -> lchild;
    }
    return q;
}

template <class DataType>
void InThrBitree<DataType>::InOrder( )
{
    ThrNode<DataType> * p;
    if (root == NULL) return;
    p = root;
    while (p -> ltag == 0)  p = p -> lchild;
    cout<<p -> data;
    while (p -> rchild != NULL)
    {
        p = Next(p);
        cout<<p -> data;
    }
}
```

(3) InThr_main.cpp

```
#include <iostream>
#include "InThr.cpp"
using namespace std;

int main()
{
    InThrBitree<char> T;
    cout<<"------中序遍历------ "<<endl;
    T.InOrder();
    cout<<endl;
```

```
    return 0;
}
```

四、 运行与测试

1. 线索链表

```
请输入创建一棵二叉树的结点数据:
AB##C#D##
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
请输入创建一棵二叉树的结点数据:
------中序遍历------
BACD

--------------------------------
Process exited after 12.03 seconds with return value 0
请按任意键继续. . .
```

第二部分 设计实验：二叉表示树

一、问题描述

一个算术表达式可以用二叉树来表示，这样的二叉树称为二叉表示树。二叉表示树具有以下两个特点：

（1） 叶子结点一定是操作数；

（2） 分支结点一定是运算符。

二、基本要求

- 采用二叉链表存储二叉表示树；
- 设计算法将给定的算术表达式转化为二叉表示树；
- 对二叉表示树进行前序、中序和后序遍历，验证表达式的前缀、中缀和后缀形式。

三、算法设计

参考实验书 p215-216

本次算法设计核心思想是把中缀表达式转化为后缀表达式，再用后缀表达式来建立一棵树。在第 6 次实验中，我们进行了实验“表达式求值”，其中就完成了如何将一个带括号的中缀表达式转化为后缀表达式。“中缀转后缀”的具体算法在这里不再赘述，可以参考第 6 次的实验报告。

下面，以课本上的中缀表达式：(A+B)*(C+D*E)为例子来讲解算法过程。这个中缀表达式对应的后缀表达式为 AB+CDE*+*。

我们建立一个栈，里面存放的元素是指向结点的指针。

- 我们依次读入表达式的每个符号。
- 如果符号是操作数，那么就建立一个单结点树并将它推入栈中。
- 如果符号是操作符，那么就从栈中弹出两棵树 T1 和 T2(T1 先弹出) 并形成一棵新的树，该树的根就是操作符，它的左、右儿子分别是 T2 和 T1。然后将指向这颗树的指针压入栈中。
- 重复操作，直到整个表达式都被读完，最后在栈中的指针即为树的根节点指针。

四、设计编码

1. SeqStack.h

```
#ifndef SEQSTACK_H
#define SEQSTACK_H
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;

const int StackSize = 100;

struct BiNode
{
    char data;
    BiNode *lchild,*rchild;
};
```

```
template<class T>
class SeqStack
{
    private:
        T data[StackSize];
        int top;
    public:
        SeqStack();
        ~SeqStack();
        void Push(T x);
        T Pop();
        T GetTop();
        int Empty();
        friend int prevalue(string s,SeqStack<int> &num,SeqStack<char> &sign);
        friend void postfix(string& s,SeqStack<int> &num,SeqStack<char>
&sign,string &posts,int &lp);
        friend int postvalue(string s,int l);
};

class Bitree
{
    public:
        Bitree();
        Bitree(char post[],int lp);
        ~Bitree();
        void PreOrder();
        void InOrder();
        void PostOrder();
    private:
        BiNode *root;
        BiNode *Creat(BiNode *bt);
        void Release(BiNode *bt);
        void PreOrder(BiNode *bt);
        void InOrder(BiNode *bt);
        void PostOrder(BiNode *bt);
};

#endif
```

2. SeqStack. cpp

```
#include <iostream>
#include <cstring>
```

```
#include "SeqStack.h"
using namespace std;

template<class T>
SeqStack<T>::SeqStack()
{
    top = -1;
}

template<class T>
SeqStack<T>::~SeqStack()
{
    //nothing
}

template <class T>
void SeqStack<T>::Push(T x)//get in
{
    if (top==StackSize-1)  throw "上溢";
    top++;
    data[top]=x;
}

template <class T>
T SeqStack<T>::Pop( )
{
    T x;
    if (top==-1)  throw "下溢";
    x=data[top--];
    return x;
}

template <class T>
T SeqStack<T>::GetTop( )
{
    if (top!=-1)  return data[top];
    return 0;
}

int inp(char c)
{

    if(c=='#') return 0;
    if(c=='(') return 1;
```

```
    if(c=='*'||c=='/'||c=='%')  return 5;
    if(c=='+'||c=='-') return 3;
    if(c==')') return 6;
    return 0;
}

int exp(char c)
{
    if(c=='#') return 0;
    if(c=='(') return 6;
    if(c=='*'||c=='/'||c=='%')  return 4;
    if(c=='+'||c=='-') return 2;
    if(c==')') return 1;
    return 0;
}

template <class T>
int SeqStack<T>::Empty( )
{
    if(top==-1)  return 1;
    else  return 0;
}

void postfix(string &s,SeqStack<int> &num,SeqStack<char> &sign,string &posts,int
&lp)
{
    sign.Push('#');
    int l=s.length();
    int k=-1;
    for(int i=0;i<l;i++)
    {
        char c=s[i];
        if(c >= 'A' && c <= 'Z')
        {
            cout<<c;
            k++;
            posts[k]=c;
        }
        else
        {
            char top=sign.GetTop( );
            if(exp(c) > inp(top))  sign.Push(c);
            else if(exp(c) < inp(top))
            {
```

```
                while(exp(c) < inp(top))
                {
                    cout<<sign.GetTop();
                    k++;
                    posts[k] = sign.GetTop();
                    sign.Pop();
                    top = sign.GetTop( );
                }
                if(exp(c) > inp(top))  sign.Push(c);
                else if((exp(c) == inp(top)) && c ==
')')  sign.Pop();
            }
            else if((exp(c)==inp(top))&& c==')')  sign.Pop();
        }
    }
    cout<<endl;
    lp=k;
}

Bitree::Bitree()
{
    root = NULL;
}

/*Bitree::Bitree(BiNode *bt)
{
    root = bt;
}*/

Bitree::Bitree(char post[],int lp)
{
    SeqStack<BiNode *> s;
    for(int i=0;i<=lp;i++)
    {
        char c = post[i];
        if(c >= 'A' && c <= 'Z')
        {
            BiNode *bt = new BiNode;
            bt -> data = c;
            bt -> lchild = NULL;
            bt -> rchild = NULL;
            s.Push(bt);
        }
        else
```

```
        {
            BiNode *bt = new BiNode;
            bt -> data = c;
            bt -> rchild = s.Pop();
            bt -> lchild = s.Pop();
            s.Push(bt);
        }
    }
    root=s.Pop();
}

Bitree::~Bitree()
{
    Release(root);
}

void Bitree::PreOrder()
{
    PreOrder(root);
}

void Bitree::InOrder()
{
    InOrder(root);
}

void Bitree::PostOrder()
{
    PostOrder(root);
}

BiNode *Bitree::Creat(BiNode *bt)
{
    char ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
    if(ch == '#')  return NULL;
    else
    {
        bt = new BiNode;
        bt -> data = ch;
        bt -> lchild = Creat(bt -> lchild);
        bt -> rchild = Creat(bt -> rchild);
        return bt;
```

```
    }
}

void Bitree::Release(BiNode *bt)
{
    if (bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

void Bitree::PreOrder(BiNode *bt)
{
    if(bt == NULL) return;
    else
    {
        cout<<bt->data<<" ";
        PreOrder(bt -> lchild);
        PreOrder(bt -> rchild);
    }
}

void Bitree::InOrder(BiNode *bt)
{
    if (bt==NULL)
    {
        return;
    }
    else {
        InOrder(bt->lchild);
        cout<<bt->data<<" ";
        InOrder(bt->rchild);
    }
}

void Bitree::PostOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        PostOrder(bt -> lchild);
        PostOrder(bt -> rchild);
```

```
        cout<<bt->data<<" ";
    }
}
```

3. SeqStack_main.cpp

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include "SeqStack.cpp"
using namespace std;

int main()
{
    SeqStack<int> num1, num2;
    SeqStack<char> sign1, sign2;
    int lp;
    cout<<"请输入中序表达式，以#结尾："<<endl;
    string pres, posts;
    cin>>pres;
    int l=pres.length();
    cout<<"中缀表达式为：";
    for (int i = 0; i < l-1; i++)  cout<<pres[i];
    cout<<endl;
    cout<<"后缀表达式为：";
    posts = pres;
    postfix(pres, num2, sign2, posts, lp);

    char post[100];
    for (int i = 0; i < lp; i++)  post[i] = posts[i];

    cout<<"根据后序表达式建立一棵二叉表示树:"<<endl;
    Bitree T(post,lp);
    cout<<endl;
    cout<<"-----前序遍历-----"<<endl; T.PreOrder();
    cout<<endl;
    cout<<"-----中序遍历-----"<<endl; T.InOrder();
    cout<<endl;
    cout<<"-----后序遍历-----"<<endl; T.PostOrder();
    cout<<endl;
    return 0;
}
```

五、运行与测试

```
请输入中序表达式，以#结尾：
(A+B)*(C+D*E)#
中缀表达式为：(A+B)*(C+D*E)
后缀表达式为：AB+CDE*+*
根据后序表达式建立一棵二叉表示树：

-----前序遍历-----
□+ A B + C * D E
-----中序遍历-----
A + B □C + D * E
-----后序遍历-----
A B + C D E * + □

--------------------------------
Process exited after 31.38 seconds with return value 0
请按任意键继续. . .
```

六、总结与心得

课上学习了很多种求二叉表示树的方法。如果用中缀表达式去构建二叉表示树，比较难处理各种加括号与去括号，还有如何用括号进行“分叉”。在本次试验中，仅仅了一种最简单的方法，利用以前中缀表达式转后缀表达式的程序，在此基础上，利用简单的栈的知识去构建二叉表示树。

第三部分 综合实验：信号放大器

一、问题描述

天然气经过管道网络从其生产基地输送到消耗地，在传输过程中，其性能的某一个或几个方面可能会有所衰减（例如气压）。为了保证信号衰减不超过容忍值，应在网络中的适位置放置放大器以增加信号（例如电压）使其与源端相同。设计算法确定把信号放大器放在何处，能使所用的放大器数目最少并且保证信号衰减不超过给定的容忍值。

二、基本要求

- 建立模型，设计数据结构；

- 设计算法完成放大器放置；
- 分析算法的时间复杂度。

三、算法设计

参考实验书 p216-217

为了简化问题，假设分布网络是二叉树结构，源端是树的根结点，信号从一个结点流向其孩子结点，树中的每一结点（除了根）表示一个可以用来放置放大器的位置。对于网络中任一结点 i，设 d(i)表示结点 i 与其父结点间的衰减量，D(i)为从结点 i 到结点 i 的子树中任一叶子结点的衰减量的最大值。要计算某结点的 D 值，必须先计算其孩子结点的 D 值，因而必须后序遍历二叉树，当访问一个结点时，计算其 D 值。

四、设计编码

1.Signal.cpp

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
int rrz = 3;

struct element
{
    int D;//从结点 i 到结点 i 的任意子树的衰减量的最大值
    int d;//结点 i 到其父亲的衰减量
    char c;
    bool boost;
};

struct BiNode
{
    element data;
    BiNode *lchild,*rchild;
};

class Bitree
{
    public:
```

```
        Bitree(){root = Creat(root);}
        ~Bitree(){Release(root);}
        void PostOrder(){PostOrder(root);}
    private:
        BiNode *root;
        BiNode *Creat(BiNode *bt);
        void Release(BiNode *bt);
        void PostOrder(BiNode *bt);
};

BiNode *Bitree::Creat(BiNode *bt)
{
    int k;
//  cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>k;
    if(k == -1) return NULL;
    else
    {
        bt = new BiNode;
        bt -> data.d = k;
        cin>>bt -> data.c;
        bt -> lchild = Creat(bt -> lchild);
        bt -> rchild = Creat(bt -> rchild);
        return bt;
    }
}

void Bitree::Release(BiNode *bt)
{
    if(bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

int MAX(int x,int y)
{
    return x>y?x:y;
}

void Bitree::PostOrder(BiNode *bt)
{
```

```
    if(bt == NULL)
    {
        return ;
    }
    else
    {
        BiNode *l = bt -> lchild;
        BiNode *r = bt -> rchild;
        PostOrder(l);
        PostOrder(r);
        bt -> data.D = 0;
        if(l != NULL)
        {
            if(l -> data.D + l -> data.d > rrz)
            {
                l -> data.boost = 1;
                bt -> data.D = l -> data.D;
                cout<<"在"<<l->data.c<<"处放一个放大器"<<endl;
            }
            else  bt -> data.D = max(bt -> data.D, l -> data.D + l -> data.d);
        }
        l = r;
        if(l != NULL)
        {
            if(l -> data.D + l -> data.d > rrz)
            {
                l -> data.boost = 1;
                cout<<"在"<<l->data.c<<"处放一个放大器"<<endl;
                bt -> data.D = max(bt -> data.D, l -> data.D);
            }
            else  bt -> data.D = max(bt -> data.D, l -> data.D + l -> data.d);
        }
        cout<<bt->data.c<<" "<<bt->data.D<<endl;
    }
}

int main()
{
    freopen("a.txt","r",stdin);
    Bitree T;
//  cout<<"请输入容忍值"<<endl;
//  cin>>rrz;
    cout<<"以实验书上 P217 为例，容忍值为 3"<<endl;
    cout<<"输出各个结点到根的衰减量，并记录是否防止放大器。"<<endl;
```

```
    T.PostOrder();
//  T.PostOrder();

    return 0;
}
```

五、运行与测试

```
以实验书上P217为例，容忍值为3
输出各个结点到根的衰减量，并记录是否防止放大器。
H 0
I 0
D 2
E 0
在D处放一个放大器
B 2
J 0
F 2
K 0
G 2
在G处放一个放大器
C 3
在C处放一个放大器
A 3

--------------------------------
Process exited after 0.1857 seconds with return value 0
请按任意键继续. . .
```

第四部分 综合实验：哈夫曼算法应用

一、问题描述

假设某文本文档只包含 26 个英文字母（为了简化问题，设全部为大写字母），应用哈夫曼算法对该文档进行压缩和解压缩操作，使得该文档占用较少的存储空间。

二、基本要求

- 假设文档内容从键盘键入；
- 设计哈夫曼算法存储结构、编码及解码算法；
- 分析算法的时间复杂度和空间复杂度。

三、算法设计

参考实验书 p218

对于给定的文档，首先通过扫描确定文档中出现了哪些英文字母以及出现的次数，以出现的次数作为叶子结点的权值构造哈夫曼树，获得各字符的哈夫曼编码; 然后再扫描遍文档将其进行哈夫曼压缩编码，将文本文档转换为二进制编码；最后将该二进制流进行解码，并与原文档进行对照，以验证算法的正确性。

• 哈夫曼树

给定一组具有确定权值的叶子结点，可以构造出不同的二叉树，将其中带权路径长度最小的二叉树称为哈夫曼树。

由此可见，由相同权值的一组叶子结点构成的二叉树具有不同的形状和不同的带权路径长度，那么如何求得哈夫曼树呢? 根据哈夫曼树的定义，一棵二叉树要使其带权路径长度最小，必须使权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。哈夫曼依据这一特点提出了哈夫曼算法，其基本思想是：

(1)初始化：由给定的 n 个权值{w1,w2,…,wn} 构造 n 棵只有一个根结点的二叉树,从而得到一个二叉树集合 F={T1,T2,…,Tn}。

(2)选取与合并：在 F 中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和。

(3)删除与加人：在 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中。

(4)重复(2)、(3)两步，当集合 F 中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

• 哈夫曼编码

哈夫曼树可用于构造最短的不等长编码方案，具体做法如下：设需要编码的字符集为{d1,d2,…,dn},它们在字符串中出现的频率为{w1,w2,…,wn},以 d1,d2,…,dn 作为叶子结点，w1,w2,…,wn 作为叶子结点的权值，构造一棵哈夫曼编码树，规定哈夫曼编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶子结点所经过的路径组成的 0 和 1 的序列便为该叶子结点对应字符的编码，称为哈夫曼编码。

四、设计编码

1. Huffman.cpp

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
const int MAXINT=9999;

struct element
{
    int weight;                 //权重
    char c;                     //字母
    int lchild,rchild,parent;   //左、右孩子（默认无左右孩子记为-1），双亲。
};

element huffTree[100];
int number[100][100],l[100];
//l[0] 表示 A 的哈夫曼码的长度，长度为 0 说明文档里面没有哈夫曼码
//number[0] 表示 A 的哈夫曼码
//选出最小的两棵树，编号为 i1，i2

void Select(element huffTree[], int &i1, int &i2, int n)
{
    for(int i = 0; i < 2*n-1; i++)
    {
        if(huffTree[i].parent == -1)
        {
            i1 = i;
            break;
        }
    }
    for(int i = 0; i < 2*n-1; i++)
    {
        if(huffTree[i].parent != -1)  continue;
        if(huffTree[i].weight < huffTree[i1].weight)  i1 = i;
    }
    for(int i = 0; i < 2*n-1; i++)
    {
        if(huffTree[i].parent == -1 && i != i1)
        {
            i2 = i;
            break;
        }
    }
    for(int i = 0; i < 2*n-1; i++)
```

```
    {
        if(huffTree[i].parent != -1 || i == i1)  continue;
        if(huffTree[i].weight < huffTree[i2].weight)  i2=i;
    }
}

//输出
void show(element huffTree[], int n, char a[])
{
    for(int k = 0; k < 2*n-1; k++)
    {
        cout<<k<<":"<<huffTree[k].c<<' '<<huffTree[k].weight<<"
"<<huffTree[k].parent
        <<" "<<huffTree[k].lchild<<" "<<huffTree[k].rchild<<endl;
    }
    cout<<endl;
}


void HuffmanTree(element huffTree[], int w[], int n, char a[])//建立哈弗曼树
{
    for(int i = 0; i < 2*n-1; i++)//初始化
    {
        huffTree[i].parent = -1;
        huffTree[i].rchild = -1;
        huffTree[i].lchild = -1;
        huffTree[i].weight = MAXINT;
        huffTree[i].c = ' ';
    }
    for(int i = 0; i < n; i++)
    {
        huffTree[i].weight = w[i];
        huffTree[i].c = a[i];
    }
    for(int k = n; k < 2*n-1; k++)
    {
        int i1,i2;
        Select(huffTree, i1, i2, n);
        huffTree[i1].parent = k;
        huffTree[i2].parent = k;
        huffTree[k].weight = huffTree[i1].weight + huffTree[i2].weight;
        huffTree[k].lchild = i1;
        huffTree[k].rchild = i2;
    }
```

```
}

void getw(string &s, char a[], int w[], int &n)
{
    cout<<"请输入文档"<<endl;
    int tw[100];
    for(int i = 0; i < 26; i++)  tw[i] = 0;
    cin>>s;
    for(int i = 0; i < s.length(); i++)
    {
        char c = s[i];
        if(c == '#')  break;
        tw[c-'A']++;
    }
    n = 0;
    for(int i = 0; i < 26; i++)
    {
        if(tw[i] != 0)
        {
            w[n] = tw[i];
            a[n] = i+'A';
            n++;
        }
    }
}

void HuffmanCode(int k, int d, int num[])//递归得到每个字母的哈夫曼码
{
    if(huffTree[k].c == ' ')
    {
        num[d] = 0;
        num[d] = 0;
        HuffmanCode(huffTree[k].lchild, d+1, num);
        num[d] = 1;
        HuffmanCode(huffTree[k].rchild, d+1, num);
        return;
    }
    else
    {
        char tc = huffTree[k].c;
        cout<<huffTree[k].c<<"的编码为";
        //储存每个字母的哈夫曼码
        for(int i = 0; i < d; i++)
        {
```

```
            cout<<num[i];
            number[tc-'A'][i] = num[i];
        }
        l[tc-'A'] = d;
        cout<<endl;
    }
}

void  DeHuffmanCode(int sn,int n,int scode[])
{
    int i = 0;
    int now = 2*n-1-1;
    while(i < sn)
    {
        int next = huffTree[now].rchild;
        if(scode[i] == 0)  next = huffTree[now].lchild;
        int nl = huffTree[next].lchild;
        int nr = huffTree[next].rchild;
        if(nl == -1 && nr == -1)
        {
            cout<<huffTree[next].c;
            now = 2*n-1-1;
        }
        else  now=next;
        i++;
    }
}

int main()
{
    for(int i=0;i<100;i++)  l[i]=0;
    int w[100], n;
    char a[100];
    for(int i = 0; i < 100; i++)  a[i]=' ';
    string s;
    getw(s,a,w,n);
    HuffmanTree(huffTree, w, n, a);
    int num[100];
    HuffmanCode(2*n-1-1, 0, num);
    cout<<endl<<"对文档进行哈夫曼压缩编码："<<endl;
    int scode[10000];
    int sn = 0;
    for(int i = 0; i < s.length(); i++)
    {
```

```
        int k = s[i]-'A';
        for(int i = 0; i < l[k]; i++)
        {
            cout<<number[k][i];
            scode[sn] = number[k][i];
            sn++;
        }
    }
    cout<<endl;
    cout<<endl<<"对二进制流解码："<<endl;
    DeHuffmanCode(sn, n, scode);
    return 0;
}
```

五、运行与测试

```
请输入文档
THEPURPOSEOFARTISWASHINGTHEDUSTOFDAILYLIFEOFOURSOULS
G的编码为00000
N的编码为00001
W的编码为00010
Y的编码为00011
O的编码为001
S的编码为010
A的编码为0110
H的编码为0111
L的编码为1000
R的编码为1001
E的编码为1010
F的编码为1011
I的编码为1100
T的编码为1101
U的编码为1110
D的编码为11110
P的编码为11111

对文档进行哈夫曼压缩编码：
11010111101011111111010011111100101010100011011011010011101110001000010011001001111100000010000011010111
01011110111001011010011011111100110110010000001110001100101110100011011001111010010100011110100010

对二进制流解码：
THEPURPOSEOFARTISWASHINGTHEDUSTOFDAILYLIFEOFOURSOULS
--------------------------------
Process exited after 42.85 seconds with return value 0
请按任意键继续. . .
```

```
请输入文档
AAAABBBBCCCCDDDD
A的编码为00
B的编码为01
C的编码为10
D的编码为11

对文档进行哈夫曼压缩编码：
00000000010101011010101011111111

对二进制流解码：
AAAABBBBCCCCDDDD
--------------------------------
Process exited after 5.72 seconds with return value 0
请按任意键继续. . .
```

六、总结与心得

这次的实验中，书上除了哈夫曼树的构建给了 C++描述的伪代码，其他算法都需要自己理解算法本质，自己编程实现，还是有一定的难度。

树这一节，需要要画很多图，自己手动做很多次模拟才能真正理解算法。编程的训练也是必不可少的。“纸上得来终觉浅，绝知此事要躬行。”也确实是这个道理。