哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：数据结构与算法

课程类型：必修

实验项目：树形结构的建立与遍历

实验题目：二叉树存储结构的建立与遍历

实验日期：2017.11.19

设计成绩	报告成绩	指导老师
		张岩

一、实验目的

树型结构的遍历是树型结构算法的基础，本实验要求编写程序演示二叉树

的存储结构的建立方法和遍历过程。

二、实验要求及实验环境

要求：

1．至少采用两种方法，编写建立二叉树的二叉链表存储结构（左右链表示）的程序，并以适当的形式显示和保存二叉树；

2．采用二叉树的二叉链表存储结构，编写程序实现二叉树的先序、中序和后序遍历的递归和非递归算法以及层序遍历算法，并以适当的形式显示和保存二叉树及其相应的遍历序列；

3．在二叉树的二叉链表存储结构基础上，编写程序实现二叉树的中序线索链表存储结构（中序线索二叉树）建立的算法，并以适当的形式显示和保存二叉树的相应的线索链表；

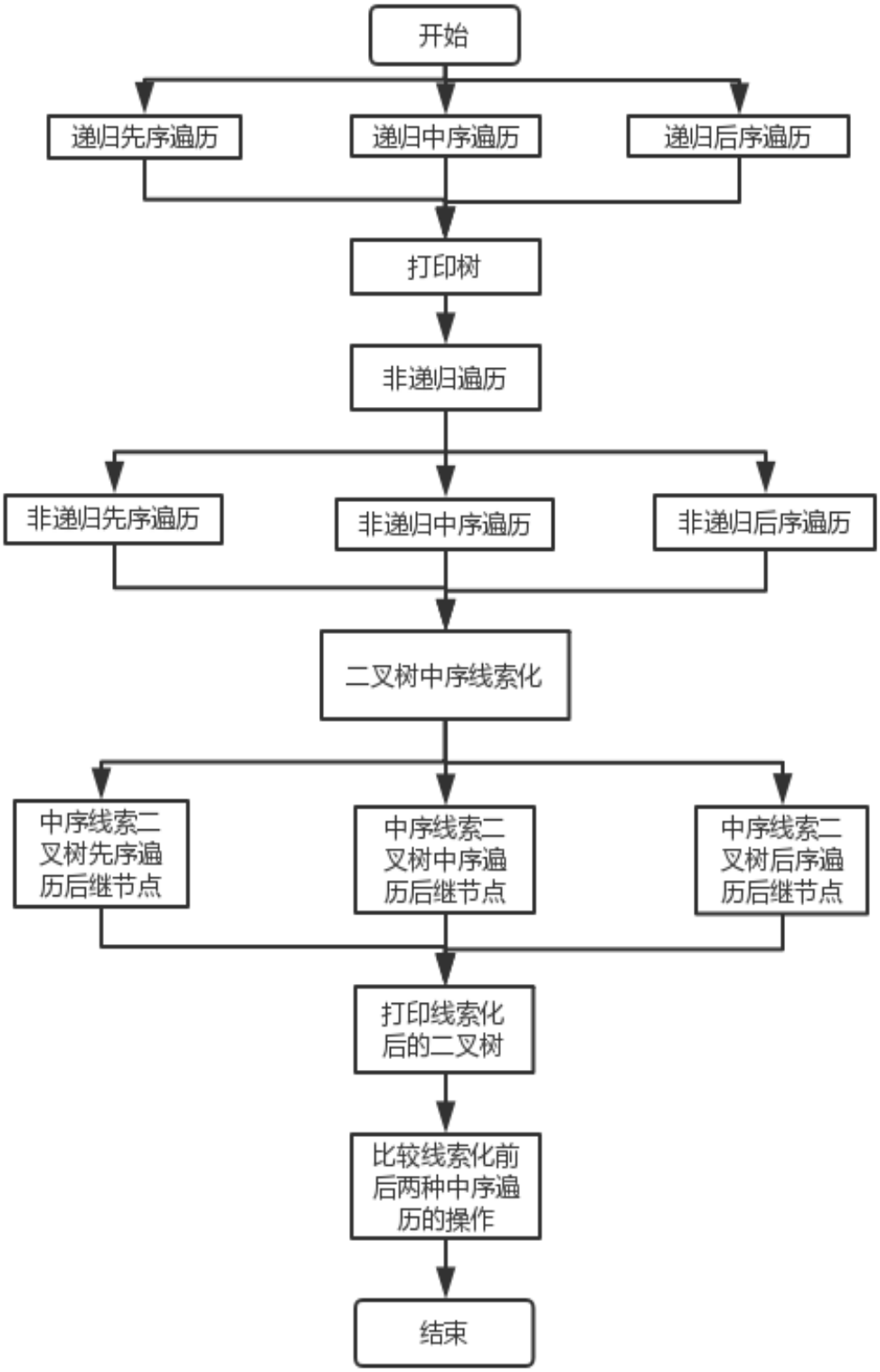
4．在二叉树的线索链表存储结构上，编写程序分别实现，求二叉树任意结点的先序、中序和后序遍历的后继结点算法；

5．以上一条要求为基础，编写程序实现对中序线索二叉树进行先序、中序和后序遍历的非递归算法，并以适当的形式显示和保存二叉树和相应的遍历序列。**环境：**

64 位 Codeblocks16.01

三、设计思想（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系）

1．逻辑设计


开始
递归先序遍历
递归中序遍历
递归后序遍历
打印树
非递归遍历
非递归先序遍历
非递归中序遍历
非递归后序遍历
二叉树中序线索化
中序线索二叉树先序遍历后继节点
中序线索二叉树中序遍历后继节点
中序线索二叉树后序遍历后继节点
打印线索化后的二叉树
比较线索化前后两种中序遍历的操作
结束

2．物理设计

（1）结点结构

```
typedef struct BiTree
{
    char data;
    struct BiTree *Lchild;
    struct BiTree *Rchild;
    bool ltag, rtag;
}BiTree,*Binary_Tree;
```

（2）

两种树的建立方式：

① 按照先序遍历的顺序输入，递归建立

② 给出某个节点在层序遍历中的位置，非递归建立

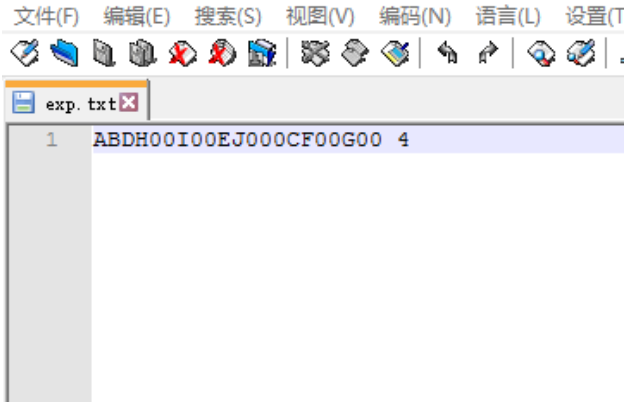
（3）

三种求后继结点的方法已经在三种线索化后的树的非递归遍历中使用，故无需验证

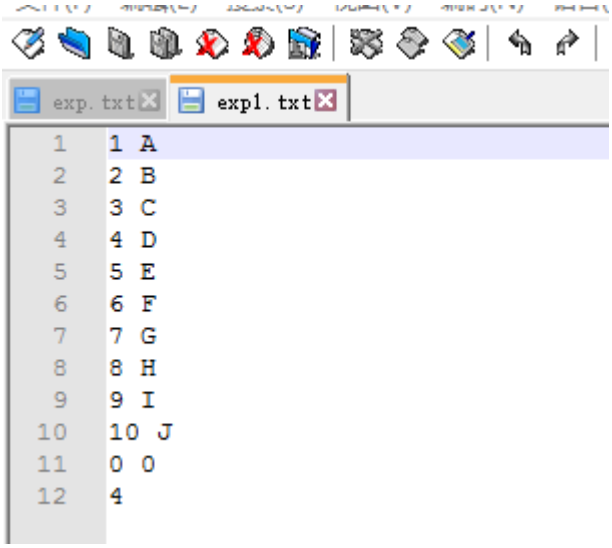
（4）树的先中后序遍历的非递归实现是通过栈和队列实现的，或者通过线索化的二叉树实现的。

四、测试结果

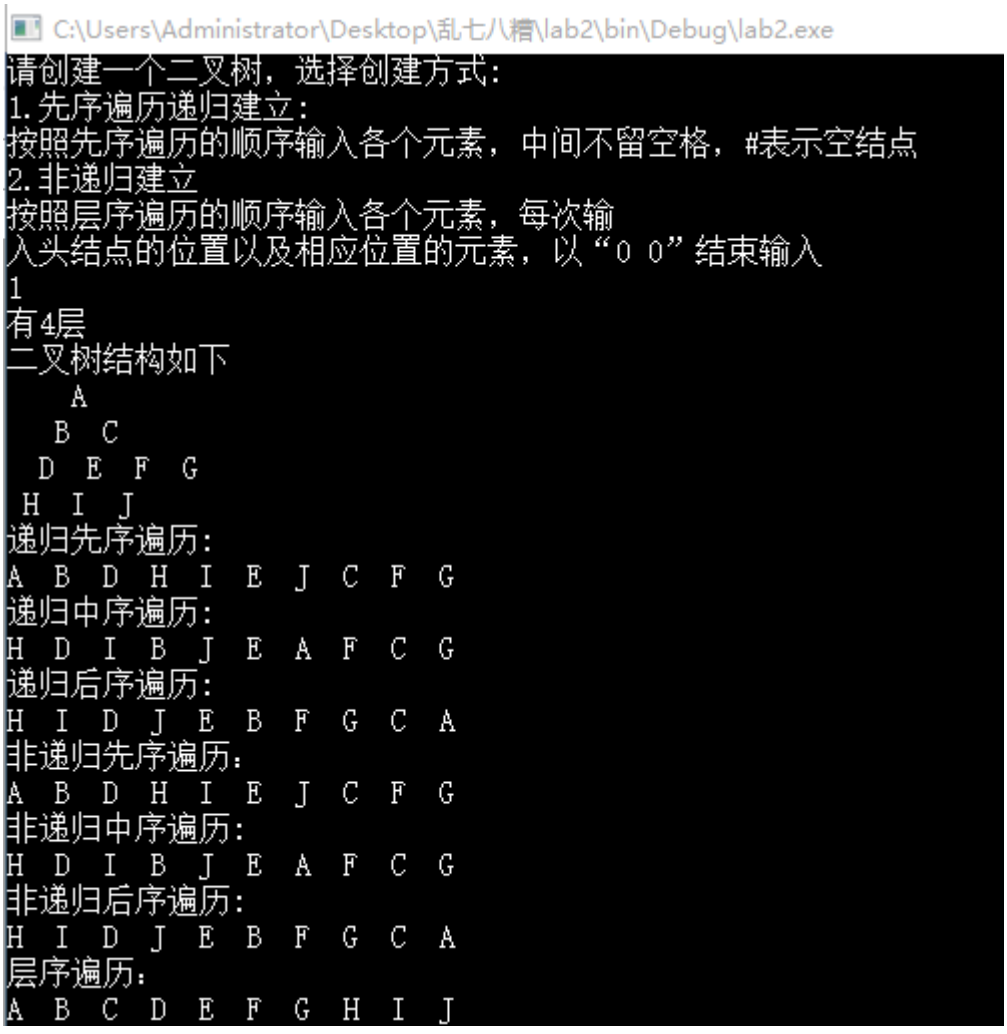
输入 1：

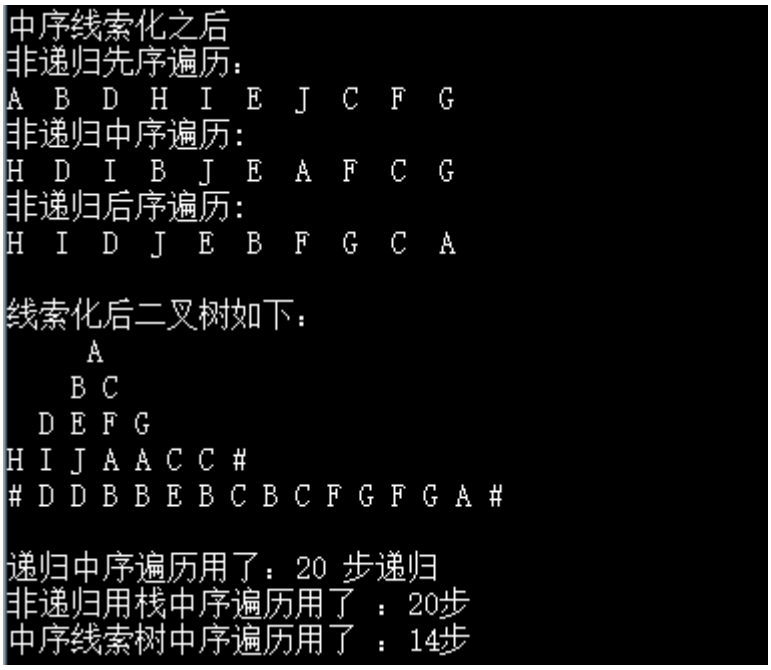


输入 2：


exp.txt
exp1.txt
1 1 A
2 2 B
3 3 C
4 4 D
5 5 E
6 6 F
7 7 G
8 8 H
9 9 I
10 10 J
11 0 0
12 4

输出：


C:\Users\Administrator\Desktop\乱七八糟\lab2\bin\Debug\lab2.exe
请创建一个二叉树，选择创建方式：
1.先序遍历递归建立：
按照先序遍历的顺序输入各个元素，中间不留空格，#表示空结点
2.非递归建立
按照层序遍历的顺序输入各个元素，每次输
入头结点的位置以及相应位置的元素，以“0 0”结束输入
1
有4层
二叉树结构如下
A
B C
D E F G
H I J
递归先序遍历:
A B D H I E J C F G
递归中序遍历:
H D I B J E A F C G
递归后序遍历:
H I D J E B F G C A
非递归先序遍历:
A B D H I E J C F G
非递归中序遍历:
H D I B J E A F C G
非递归后序遍历:
H I D J E B F G C A
层序遍历:
A B C D E F G H I J



五、经验体会与不足

经验体会：

（1） 发现中序线索化后的二叉树中序遍历，相比用栈来非递归遍历，节省的指针操作步骤很多；

（2） 中序线索化后的二叉树也可以得到他们的先序遍历和后序遍历的后继节点

（3） 打印二叉树的时候可以把二叉树当成一个矩阵（部分为空白，部分为有字符的矩阵）

不足：

二叉树的打印还是不够清楚。

六、附录：源代码（带注释）

```
#include<stdio.h>
#include<stdlib.h>
#include<cstdio>
#include<iostream>
#include<stack>
#include<queue>
#include <string.h>
#include<math.h>
```

```
#include<time.h>
#define max 100
using namespace std;

clock_t startTime1,endTime1,startTime2,endTime2,startTime3,endTime3;
int cnt2=0,cnt3=0;
typedef struct BiTree
{
    char data;
    struct BiTree *Lchild;
    struct BiTree *Rchild;
    bool ltag, rtag;
}BiTree,*Binary_Tree;
Binary_Tree Head=NULL;
int x;
Binary_Tree s[max]; /*  辅助指针数组，存放二叉树结点指针 */
Binary_Tree *ptmp;
int cnt=0;
Binary_Tree CreateBT()
{   int i,j;
    char ch;
    Binary_Tree bt,p; /* bt 为根，p 用于建立结点 */
    cin>>i>>ch ;
    while ( i != 0&&ch != '0') {
    p =new struct BiTree;
    p->data=ch;
    p->Lchild=NULL; p->Rchild=NULL;

    s[i]=p;
```

```
    if (i==1) bt=p ;
    else {
    j=i/2; /*父结点的编号*/
    if (i%2==0) s[j]->Lchild=p; /*i 是 j 的左儿子 */
    else s[j]->Rchild=p; /*i 是 j 的右儿子 */
    }
    cin>>i>>ch ;
    }
    return s[1];
}

//创建一个二叉树

char ch;
int CreateBiTree(Binary_Tree *T)
{
   *T=(Binary_Tree)malloc(sizeof(BiTree));
    if(!(*T))
        exit(OVERFLOW);
    scanf("%c",&ch);

    if(ch=='0')
       {*T=NULL;}
    else
    {
        (*T)->data=ch;
        (*T)->ltag=true;
        (*T)->rtag=true;
        CreateBiTree(&((*T)->Lchild));
```

```
        CreateBiTree(&((*T)->Rchild));
    }
    return 1;
}

//先遍历二叉树
int PreShowBiTree(Binary_Tree T)
{
    if(T!=NULL)
    {
        printf("%c  ",T->data);
        PreShowBiTree(T->Lchild);
        PreShowBiTree(T->Rchild);
    }
    return 1;
}


//中遍历二叉树
int MidShowBiTree(Binary_Tree T)
{
    if(T!=NULL)
    {
        MidShowBiTree(T->Lchild);
        printf("%c  ",T->data);
        MidShowBiTree(T->Rchild);
    }
    return 1;
}
```

```
//后遍历二叉树
int BehShowBiTree(Binary_Tree T)
{
    if(T!=NULL)
    {
        BehShowBiTree(T->Lchild);
        BehShowBiTree(T->Rchild);
        printf("%c  ",T->data);
    }
    return 1;
}
//层序遍历二叉树
int LevelOrderTraversal(Binary_Tree T)
{
    queue<struct BiTree>Que;
    if(T==NULL)return -1;
    Que.push(*T);
    while(!Que.empty())
    {
        BiTree temp;
        temp=Que.front();
        Que.pop();
        cout<<temp.data<<"  ";
        cnt++;
        T=&temp;
        if(T->Lchild!=NULL){Que.push(*(T->Lchild));}
        if(T->Rchild!=NULL){Que.push(*(T->Rchild));}
```

```
    }
    return 0;
}
int PreShowBiTree1(Binary_Tree T)
{
    stack<Binary_Tree>S1;
    ptmp=(Binary_Tree *)malloc(sizeof(Binary_Tree)*cnt);
    int i=-1;
    while(T||!S1.empty())
    {
        while(T)
        {
            printf("%c  ",T->data);
            ptmp[++i]=T;
            S1.push(T);
            T=T->Lchild;
        }
        if(!S1.empty())
        {
            T=S1.top();
            S1.pop();
            T=T->Rchild;
        }
    }
    return 1;
}

//中遍历二叉树
int MidShowBiTree1(Binary_Tree T)
```

```
{
   stack<Binary_Tree>Stk;
   Binary_Tree T1=T;
   while(T1||!Stk.empty())
   {
       while(T1)
       {
           Stk.push(T1);
           T1=T1->Lchild;
           cnt2++;
       }
       if(!Stk.empty())
       {
           T1=Stk.top();
           Stk.pop();
           printf("%c  ",T1->data);
           T1=T1->Rchild;
           cnt2++;
       }
   }
   return 1;
}
int BehShowBiTree1(Binary_Tree T)
{
    stack<Binary_Tree> S2;
    Binary_Tree curr = T ;           // 指向当前要检查的节点
    Binary_Tree previsited = NULL;    // 指向前一个被访问的节点
    while(curr != NULL || !S2.empty())  // 栈空时结束
    {
```

```
        while(curr != NULL)             // 一直向左走直到为空
        {
            S2.push(curr);
            curr = curr->Lchild;
        }
        curr = S2.top();
        // 当前节点的右孩子如果为空或者已经被访问，则访问当前节点
        if(curr->Rchild == NULL || curr->Rchild == previsited)
        {
            printf("%c  ",curr->data);
            previsited = curr;
            S2.pop();
            curr = NULL;
        }
        else
            curr = curr->Rchild;       // 否则访问右孩子
    }
   return 1;
}

BiTree *pre=NULL;
void InThread(Binary_Tree T)
{
    Binary_Tree p;
    p=T;
    if(p)
    {
        InThread(p->Lchild);
        p->ltag=(p->Lchild)?true:false;
```

```
        p->rtag=(p->Rchild)?true:false;
        if(pre){
            if(pre->rtag==false)
                pre->Rchild=p;
            if(p->ltag==false)
                p->Lchild=pre;
        }
        pre=p;
        InThread(p->Rchild);
    }
}
Binary_Tree InOrderTh(Binary_Tree T)//线索化
{
    Binary_Tree temp1=T;
    Binary_Tree temp2=T;
    Binary_Tree Head;
    Head=new BiTree;
    Head->Lchild=T;
    Head->Rchild=Head;
    Head->data='#';
    Head->ltag=true;
    Head->rtag=true;
    InThread(T);
    while(temp1->ltag!=false)
    {
        temp1=temp1->Lchild;
    }
    temp1->Lchild=Head;
    while(temp2->rtag!=false)
```

```
    {
        temp2=temp2->Rchild;
    }
    temp2->Rchild=Head;
    return Head;

}
Binary_Tree PreNext(Binary_Tree p)//先序后继
{
  //  if(p==Head)return NULL;
    Binary_Tree q;
    if(p->ltag==true)
        q=p->Lchild;
    else{
        q=p;
        while(q->rtag==false)
            q=q->Rchild;
        q=q->Rchild;
    }
    return q;
}
//如果有左儿子，就是左儿子，如果没有左儿子，就是右子树的最左结点
Binary_Tree InNext(Binary_Tree p)//中序后继
{
    Binary_Tree q;
    q=p->Rchild;
    cnt3++;
    if(p->rtag==true)
    {
```

```
        if(q->ltag==true)
        while(q->ltag==true)
        {q=q->Lchild;cnt3++;}
        else cnt3++;
    }
    return q;
}
Binary_Tree BeNext(Binary_Tree p)//后序后继
{
    Binary_Tree q;
    Binary_Tree tmp;
    Binary_Tree T;
    for(int i=0;i<cnt;i++)
    {
        if(ptmp[i]->Lchild==p&&ptmp[i]->ltag==true)
        {
            tmp=ptmp[i];
            break;
        }
        if(ptmp[i]->Rchild==p&&ptmp[i]->rtag==true)
        {
            tmp=ptmp[i];
            break;
        }
    }

    q=tmp;
    if(q->Lchild==p&&q->rtag==true)
    {
```

```
        q=q->Rchild;
        while(q->ltag==true)q=q->Lchild;
        return q;
    }
    if(q->Rchild==p||q->rtag==false)
    {
        return q;
    }
}
void PreShowBiTree2(Binary_Tree Head)//先序遍历非递归
{
    Binary_Tree p;
    p=Head->Lchild;
    while(p!=Head)
    {
        cout<<p->data<<"  ";
        p=PreNext(p);
    }
}
void InOrderTh2(Binary_Tree Head)//中序遍历非递归
{
    Binary_Tree p=Head->Lchild;

     while(p->ltag==true)
            p=p->Lchild;
    while(p!=Head)
    {

      printf("%c  ",p->data);
```

```
      p=InNext(p);
    }

}
```

//找到其父结点的左儿子如果就是该节点，那么是父结点右子树的最左节点。如果是右儿子，就是父亲节点。

```
void BeOrderTh2(Binary_Tree Head)//后序遍历非递归
{
    Binary_Tree p=Head->Lchild;
     while(p->ltag==true)
            p=p->Lchild;
     while(p!=Head->Lchild)
     {
        cout<<p->data<<"  ";
        p=BeNext(p);
     }
     cout<<p->data<<endl;

     cout<<endl;
}
char *a=(char *)malloc(sizeof(char)*(pow(2,x)-1));
void Print2x(Binary_Tree T,int x)
{
    queue<struct BiTree>Q;
    int m=1;
    if(T==NULL){return;}
    Q.push(*T);
```

```
    int x1=x;

    while(!Q.empty())
    {
        BiTree temp;
        temp=Q.front();
        Q.pop();
        a[m++]=temp.data;
        T=&temp;
        if(T->Lchild==NULL&&m<=pow(2,x)-1)a[m++]='#';
        if(T->Rchild==NULL&&m<=pow(2,x)-1)a[m++]='#';
        if(T->Lchild!=NULL){
                Q.push(*(T->Lchild));
        }
        if(T->Rchild!=NULL){
                Q.push(*(T->Rchild));
        }
        if(m>pow(2,x)-1)return ;
    }

}
void Print2(Binary_Tree T,int x)
{
    Print2x(T,x);
    for(int k=0;k<x;k++)cout<<" ";
    for(int i=1;i<=pow(2,x)-1;i++)
    {
        cout<<a[i]<<" ";
```

```
if(fabs((log(i+1)/log(2))-(int)((log(i+1)/log(2)))==0||i==7||i==15))
            {
                cout<<endl;
             for(int k=0;k<x-i;k++)cout<<" ";
            }
    }
    cout<<endl;
    return;
}
void Print1(Binary_Tree T,int x)
{
    queue<struct BiTree>Q1;
    if(T==NULL){return;}
    Q1.push(*T);
    double t=0;
    int x1=x;
    for(int i=0;i<x;i++)cout<<" ";
    while(!Q1.empty())
    {

        BiTree temp;
        temp=Q1.front();
        Q1.pop();
        cout<<temp.data<<"  ";
        t++;


if(fabs((log(t+1)/log(2))-(int)((log(t+1)/log(2))))==0||t==7||t==15)
        {
```

```
            cout<<endl;
            x1--;
            for(int i=0;i<x1;i++)cout<<" ";
        }
        T=&temp;
        if(T->Lchild!=NULL){Q1.push(*(T->Lchild));}
        if(T->Rchild!=NULL){Q1.push(*(T->Rchild));}
        if((log(t+1)/log(2))==x)break;
    }
    cout<<endl;
    return ;
}

int main()
{
    BiTree * T=NULL;
    printf("请创建一个二叉树，选择创建方式:\n");
    cout<<"1.先序遍历递归建立:"<<endl;
    cout<<"按照先序遍历的顺序输入各个元素，中间不留空格，#表示空结点
"<<endl;
    cout<<"2.非递归建立"<<endl;
    cout<<"按照层序遍历的顺序输入各个元素，每次输"<<endl<<"入头结点的
位置以及相应位置的元素，以“0 0”结束输入"<<endl;
    int option;
    cin>>option;
    if(option==1){ freopen("exp.txt","r",stdin);CreateBiTree(&T);}
    if(option==2){  freopen("exp1.txt","r",stdin);T=CreateBT();}
    scanf("%d",&x);
```

```
cout<<"二叉树结构如下"<<endl;
Print1(T,x);
printf("递归先序遍历: \n");
PreShowBiTree(T);
printf("\n");
printf("递归中序遍历:  \n");
startTime1 = clock();
MidShowBiTree(T);
endTime1 = clock();
printf("\n");
printf("递归后序遍历: \n");
BehShowBiTree(T);
printf("\n");
printf("非递归先序遍历：\n");
PreShowBiTree1(T);
printf("\n");
printf("非递归中序遍历:  \n");
startTime2 = clock();
MidShowBiTree1(T);
endTime2 = clock();
printf("\n");
printf("非递归后序遍历: \n");
BehShowBiTree1(T);
printf("\n");
printf("层序遍历：\n");
LevelOrderTraversal(T);
printf("\n");
printf("中序线索化之后\n");
Binary_Tree t;
```

```
    t=InOrderTh(T);
    cout<<"线索化后二叉树如下："<<endl;
    Print2(T,x+1);
    cout<<endl;
    printf("非递归先序遍历：\n");
    PreShowBiTree2(t);
    printf("\n");
    printf("非递归中序遍历:  \n");
    startTime3 = clock();
    InOrderTh2(t);
    endTime3=clock();
    printf("\n");
    printf("非递归后序遍历: \n");
    BeOrderTh2(t);
    printf("\n");

    fclose(stdin);
   // cout << "Totle Time : " <<(double)(endTime1 - startTime1) /
CLOCKS_PER_SEC << "s" << endl;
   // cout << "Totle Time : " <<(double)(endTime2 - startTime2) /
CLOCKS_PER_SEC << "s" << endl;
   //  cout << "Totle Time : " <<(double)(endTime3 - startTime3) /
CLOCKS_PER_SEC << "s" << endl;

    cout<<"非递归用栈中序遍历用了 ："<<cnt2<<"步"<<endl;
    cout<<"中序线索树中序遍历用了 ："<<cnt3<<"步"<<endl;
    return 0;
}
```