|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 实验成绩 |  |
| 专业班级 |  | 实验时间 | 2025-05-31 | 实验地点 | 计算中心 |

实验四：二叉树与树实验

1实验目的

1. 掌握二叉树与树（森林）的存储表示。
2. 掌握二叉树与树（森林）的遍历算法（递归和非递归）。
3. 运用二叉树与树（森林）遍历算法求解有关问题。
4. 掌握树（森林）和二叉树的相互转换。

2 实验要求

1. 对用到的二叉树与树（森林）进行合理封装；
2. 在相应的存储结构上，实现基本运算；
3. 设计算法实现相应的实验任务；
4. 二叉树的测试数据用文本文件方式给出，例如测试数据名为bt151.btr的二叉树，测试数据名为tree10.tre的树或森林，可参考发来的树和森林形状和参考存储文件，可参考发来的二叉树与树（森林）形状和参考存储文件；
5. 二叉树创建方法可自行选择；
6. 程序运行、测试正确；
7. 实验程序有较好可读性，各运算和变量的命名直观易懂，符合软件工程要求；
8. 程序有适当的注释，程序的书写要采用缩进格式。

3 实验任务

**实验1~4所使用的数据结构：**

很抱歉没有使用压缩包自带的头文件，以前写过，为了较为便利的复用，不采用附录中的代码，通过根据附录PPT图片来手动构造；

从字符串构造二叉树：

代码：

typedef int Status;

typedef char ElemTypes;

typedef struct TreeNode {

ElemTypes data;

struct TreeNode \*leftChild,\*rightChild;

} TreeNode,\*Tree;

void CreateTree(Tree \*T) {

ElemTypes ch;

//从1开始，每次递归递增，移动到下一个元素

ch=str[treeIndex++];

if(ch=='#') {

\*T=NULL;

} else {

\*T= (Tree)malloc(sizeof(TreeNode));

if(!\*T) exit(0);

//采用前序遍历的方法构造

(\*T)->data=ch; /\* 生成根结点 \*/

//构造子树

CreateTree(&(\*T)->leftChild);

CreateTree(&(\*T)->rightChild);

}

}

1. **将二叉链表存储的二叉树转换为顺序存储形式。（提示：数组中要扩展为完全二叉树）。实验测试数据基本要求：**

**第一组数据： bt8.btr**

**第二组数据： bt14.btr**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：

1.先计算出顺序存储需要的容量，因为空结点也要存储，用#代替，所以此高度的满二叉树结点数即为顺序数组容量，如下代码：

int count = 0;

for(int i = 0;i < Height(T);i++) {

count += pow(2,i);

}

2.初始化数组所有元素为#

3.进行转换，递归左右子树

4.打印结果

代码：

// 计算顺序存储需要的容量 根据二叉树性质和高度

int count = 0;

for(int i = 0;i < Height(T);i++) {

count += pow(2,i);

}

// 初始化顺序存储数组

char A[count+1];

for(int i = 0;i <= count;i++) {

A[i] = '#';

}

// 转换

Transfer(T,0,A);

// 打印

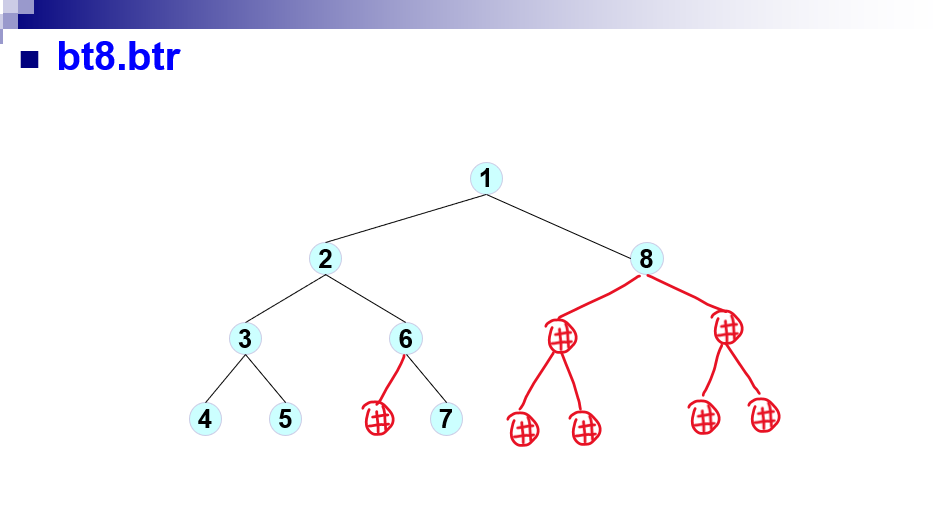
for(int i = 0;i <= count;i++) {

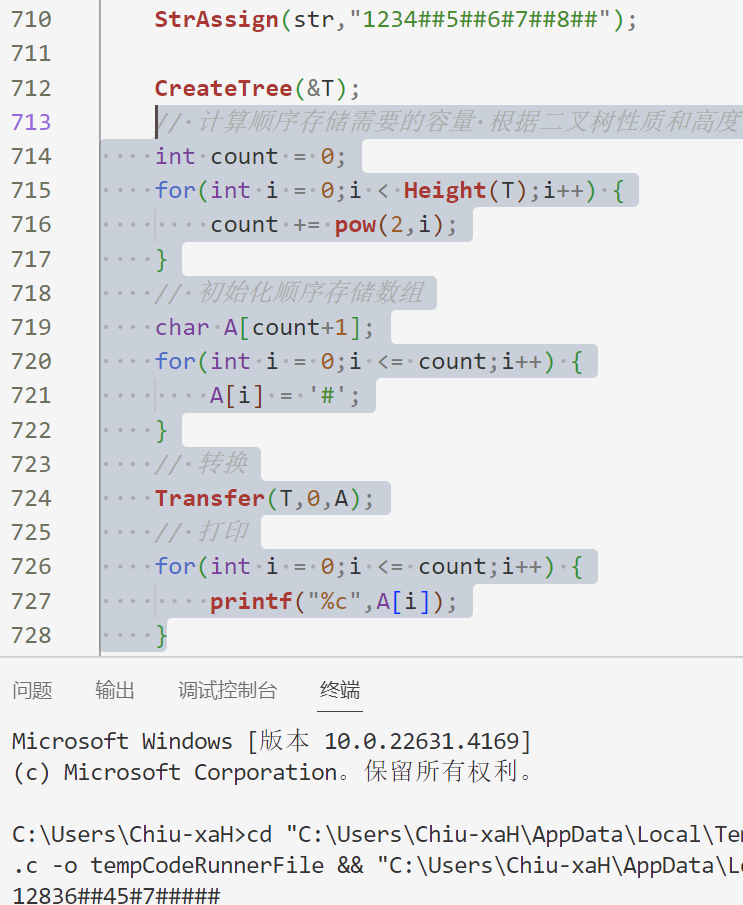
printf("%c",A[i]);

}

**运行结果截图及说明**

第一组使用bt8数据：首先对树进行补全，如下，然后推理出顺序数组应该以层序进行存储，即12836##45#7####，运行代码查看结果





符合预期

同理对bt14数据进行同操作：推理结果为：

ab#ci##dfj######eg#kl###############h####m#########################################n############################################

运行代码：



符合预期

1. **键盘输入一个元素x，求其父节点、兄弟结点、子结点的值，不存在时给出相应提示信息。对兄弟结点和孩子结点，存在时要明确指出是左兄弟、左孩子、右兄弟或右孩子。实验测试数据基本要求：**

**第一组数据： bt31.btr**

**第二组数据： bt21.btr**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：

扩展层序遍历函数（层序遍历需要借助队列），在遍历过程中记录左右孩子与兄弟值即可

层序遍历函数：

void GetLayer(Tree T) {

if (T == NULL) {

return;

}

Queue Q;

InitQueue(&Q);

Tree tree = T;

Add(&Q,tree);

while(!isEmpty(Q)) {

Tree E;

DeAdd(&Q,&E);

printf("%c ",E->data);

if(E->leftChild) {

Add(&Q,E->leftChild);

}

if(E->rightChild) {

Add(&Q,E->rightChild);

}

}

}

代码：

void PrintNodeRelation(Tree T,ElemTypes X) {

if (T == NULL) {

return;

}

Tree target = NULL;

Tree parent = NULL;

Tree leftBrother = NULL, rightBrother = NULL;

Tree leftChild = NULL, rightChild = NULL;

Queue Q;

InitQueue(&Q);

Tree tree = T;

Add(&Q,tree);

while(!isEmpty(Q)) {

Tree E;

DeAdd(&Q,&E);

// 找到目标节点

if (E->data == X) {

target = E;

if (E->leftChild != NULL)

leftChild = E->leftChild;

if (E->rightChild != NULL)

rightChild = E->rightChild;

}

if(E->leftChild) {

Add(&Q,E->leftChild);

if (E->leftChild->data == X) {

parent = E;

if (E->rightChild) rightBrother = E->rightChild;

}

}

if(E->rightChild) {

Add(&Q,E->rightChild);

if (E->rightChild->data == X) {

parent = E;

if (E->leftChild) leftBrother = E->leftChild;

}

}

}

if (!target) {

printf("Not Found Node Which Value Is %c\n", X);

return;

}

if (parent) {

printf("Parent Value Is %c\n", parent->data);

} else {

printf("No Parent\n");

}

if (leftBrother) {

printf("Left Brother Value Is %c\n", leftBrother->data);

}

if (rightBrother) {

printf("Right Brother Value Is %c\n", rightBrother->data);

}

if (!leftBrother && !rightBrother) {

printf("No Brothers\n");

}

if (leftChild) {

printf("Left Child Value Is %c\n", leftChild->data);

}

if (rightChild) {

printf("Right Child Value Is %c\n", rightChild->data);

}

if (!leftChild && !rightChild) {

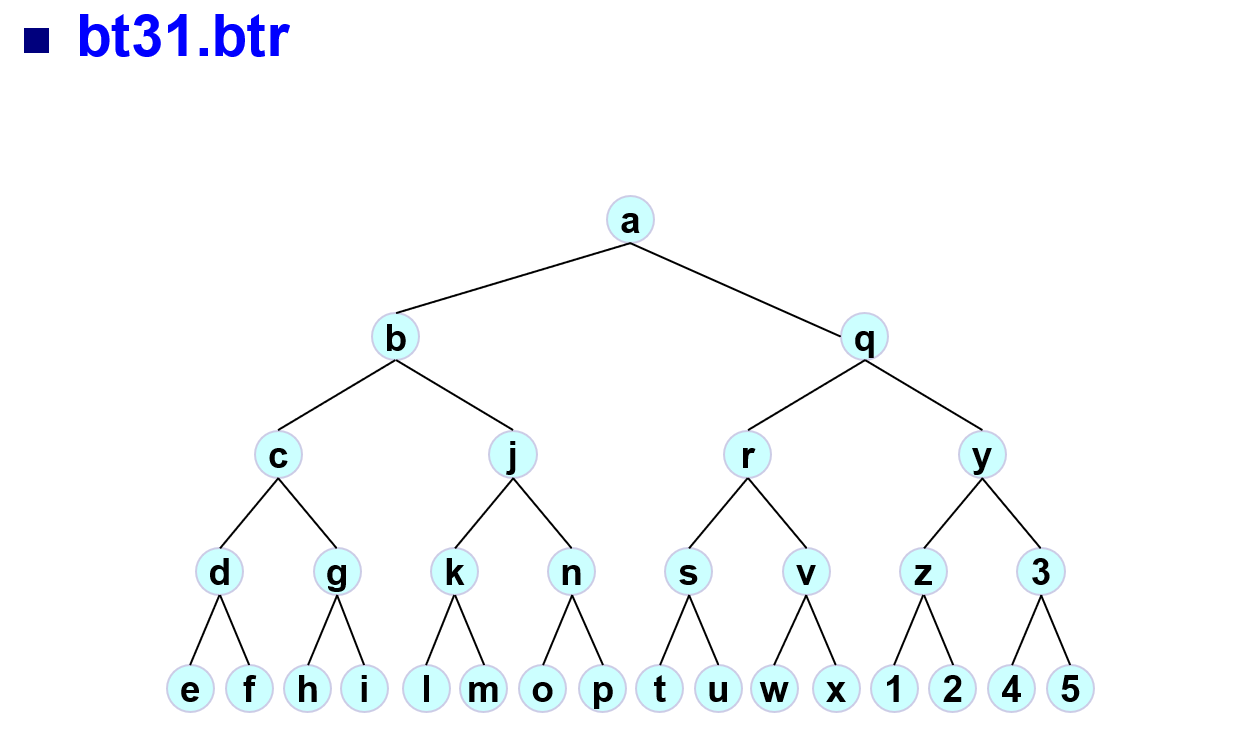
printf("No Children\n");

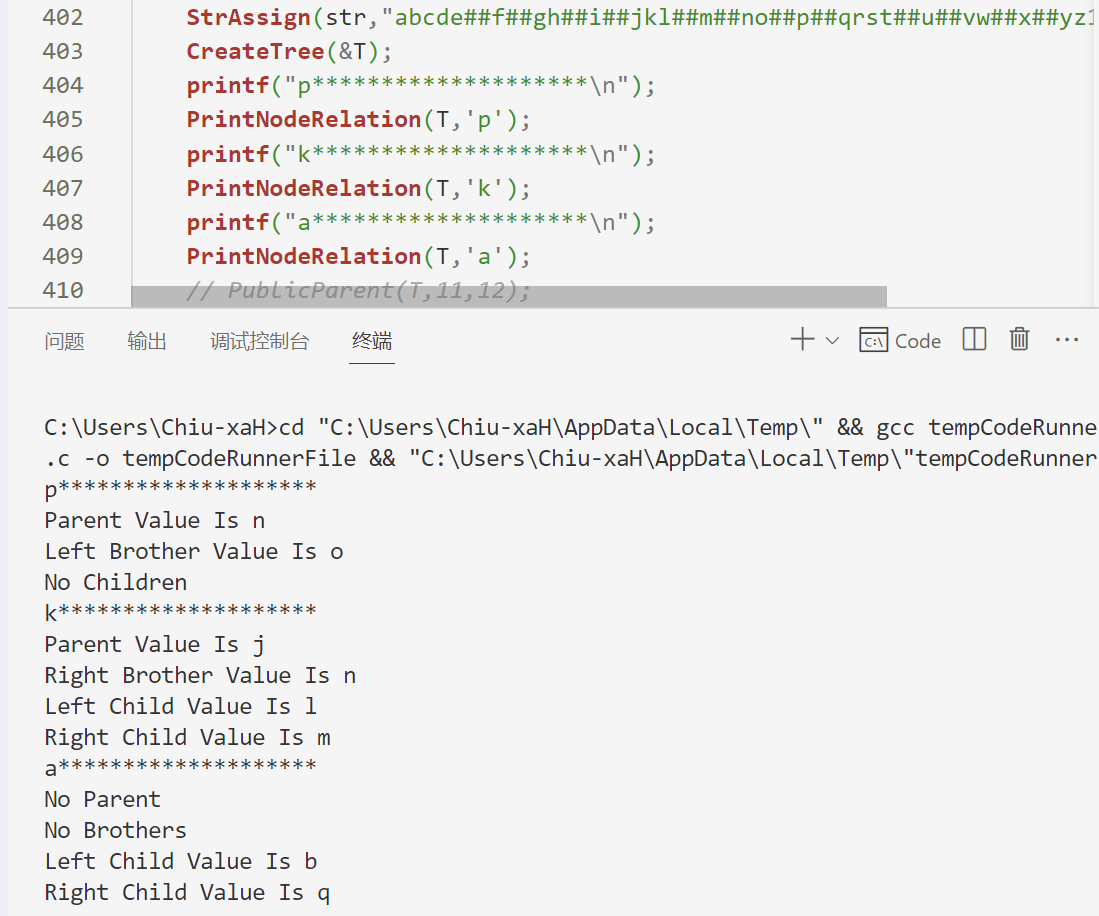
}

}

**运行结果截图及说明**

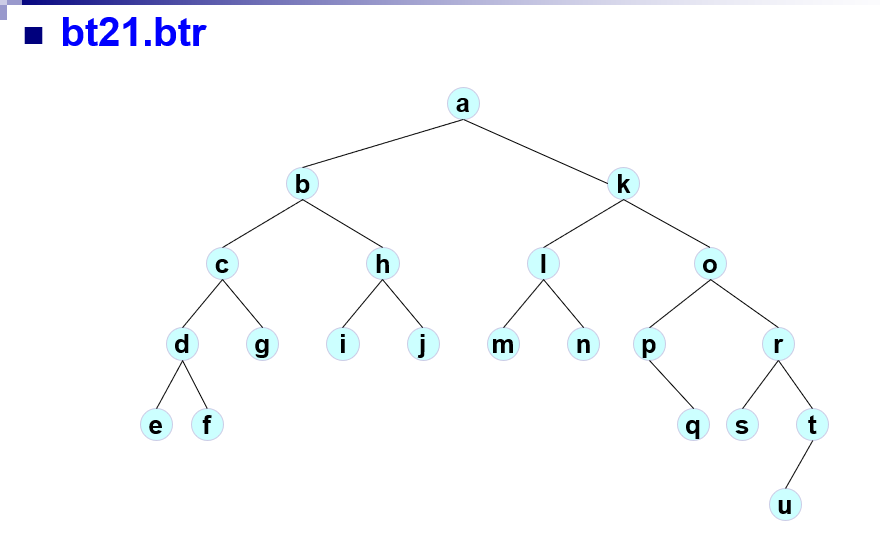
第一组数据为bt31，如图，测试p、k、a三点，打印结果如下

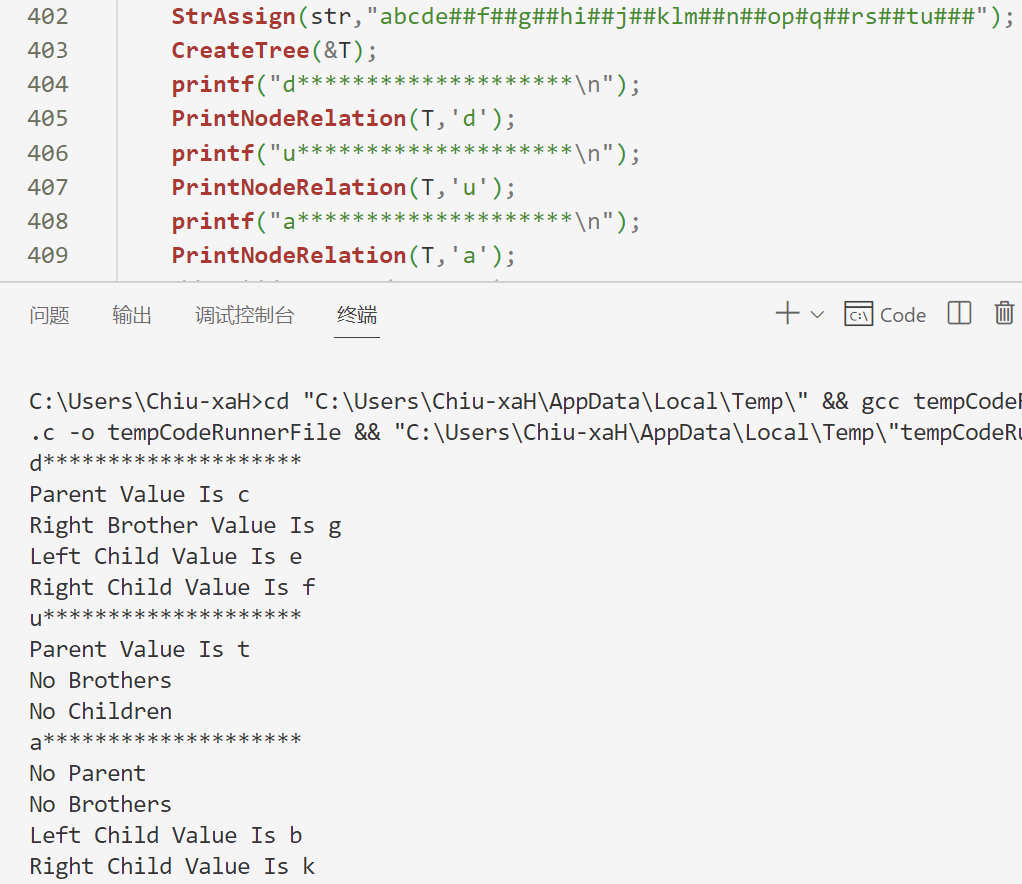




符合预期

同理，对于第二组数据bt21，如图，测试d、u、a三点，打印结果如下





符合预期

1. **对二叉链表表示的二叉树，求2个结点最近的共同祖先。实验测试数据基本要求：**

**第一组数据： bt261.btr**

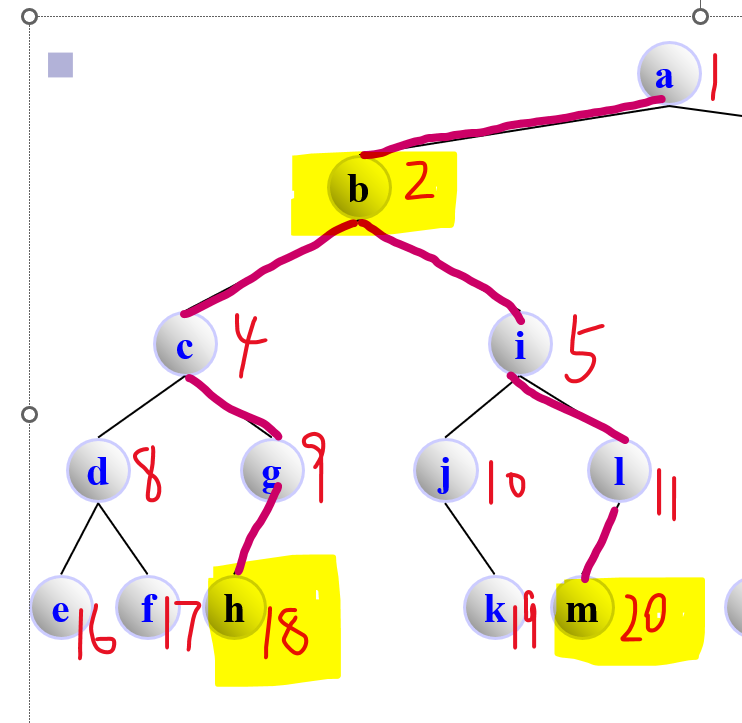
**第二组数据： bt21.btr**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：首先需要一个辅助函数Find，传入层序编号（1、2、3…），借助队列进行层序遍历，直到到达i结束，找到结点指针；然后在SearchPublicParent寻找I与J两个结点的最近公共祖先，可以从根结点出发，通过递归寻找左右子树，寻找第一次发现目标在左右子树的结点，则为公共祖先，返回，否则则说明目标在同一侧子树，用三元表达式返回非空那侧子树

例如：寻找h与m结点的公共祖先，有b和a，但是显然最近的是b



代码：

//根据层序编号找到结点

Status Find(Tree T,int i,Tree \*node) {

Queue Q;

InitQueue(&Q);

if (T == NULL) {

return ERROR;

}

Tree tree = T;

int count = 0;

Add(&Q,tree);

while(count != i && !isEmpty(Q)) {

Tree E;

DeAdd(&Q,&E);

count++;

if(count == i) {

\*node = E;

return OK;

}

if(E->leftChild) {

Add(&Q,E->leftChild);

}

if(E->rightChild) {

Add(&Q,E->rightChild);

}

}

return ERROR;

}

Tree SearchPublicParent(Tree T, Tree I, Tree J) {

if (T == NULL || T == I || T == J) {

return T;

}

Tree left = SearchPublicParent(T->leftChild, I, J);

Tree right = SearchPublicParent(T->rightChild, I, J);

// 目标在左子树和右子树，第一次发现此点即为最近公共祖先

if (left && right) {

return T;

}

// 否则返回非空的子树，两个目标都在那边

return left ? left : right;

}

void PublicParent(Tree T,int i,int j) {

Tree node1, node2;

if (Find(T, i, &node1) == ERROR || Find(T, j, &node2) == ERROR) {

return;

}

Tree ancestor = SearchPublicParent(T, node1, node2);

if (ancestor) {

printf("Nearest is %c\n", ancestor->data);

} else {

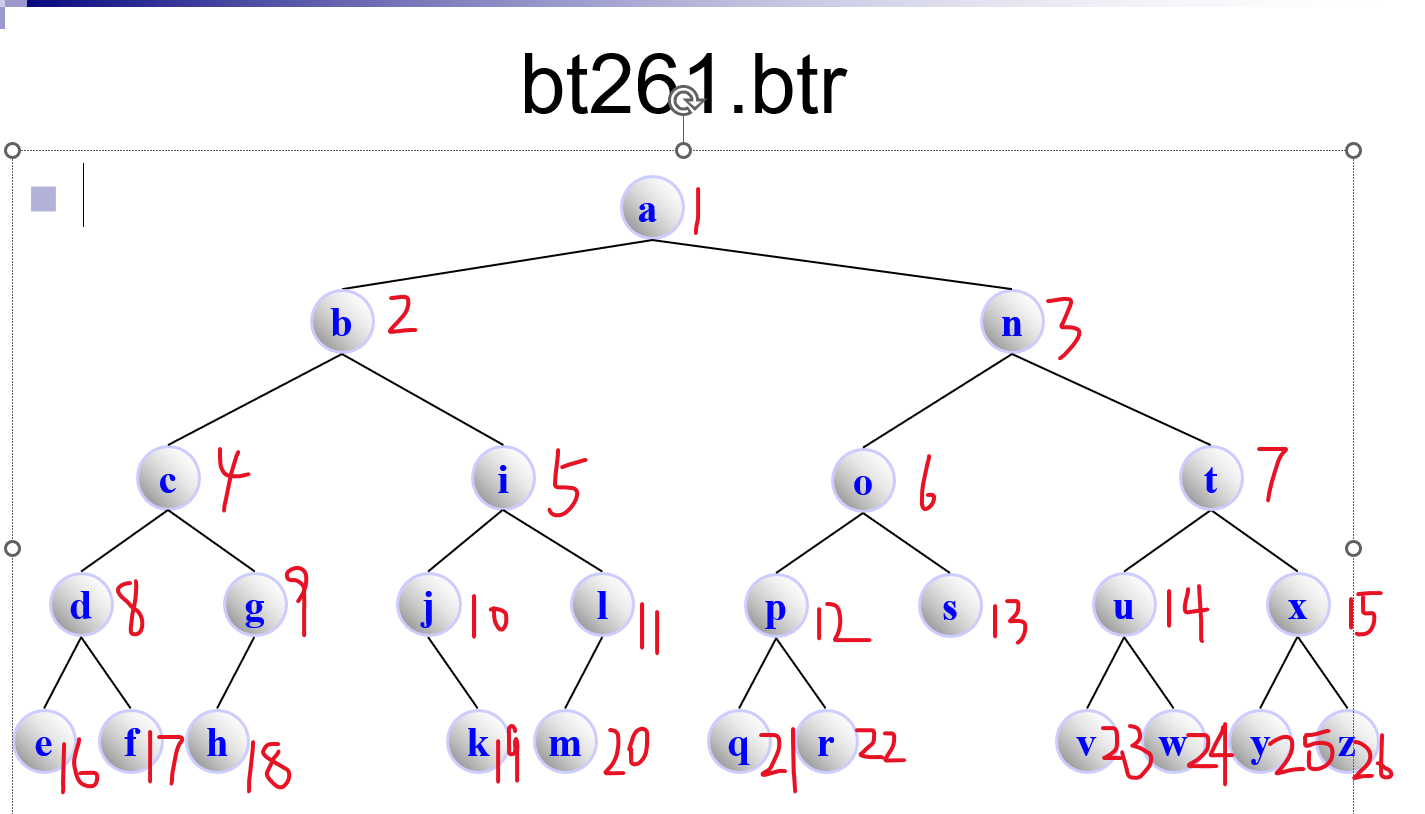
printf("Nearest is NULL");

}

}

**运行结果截图及说明**

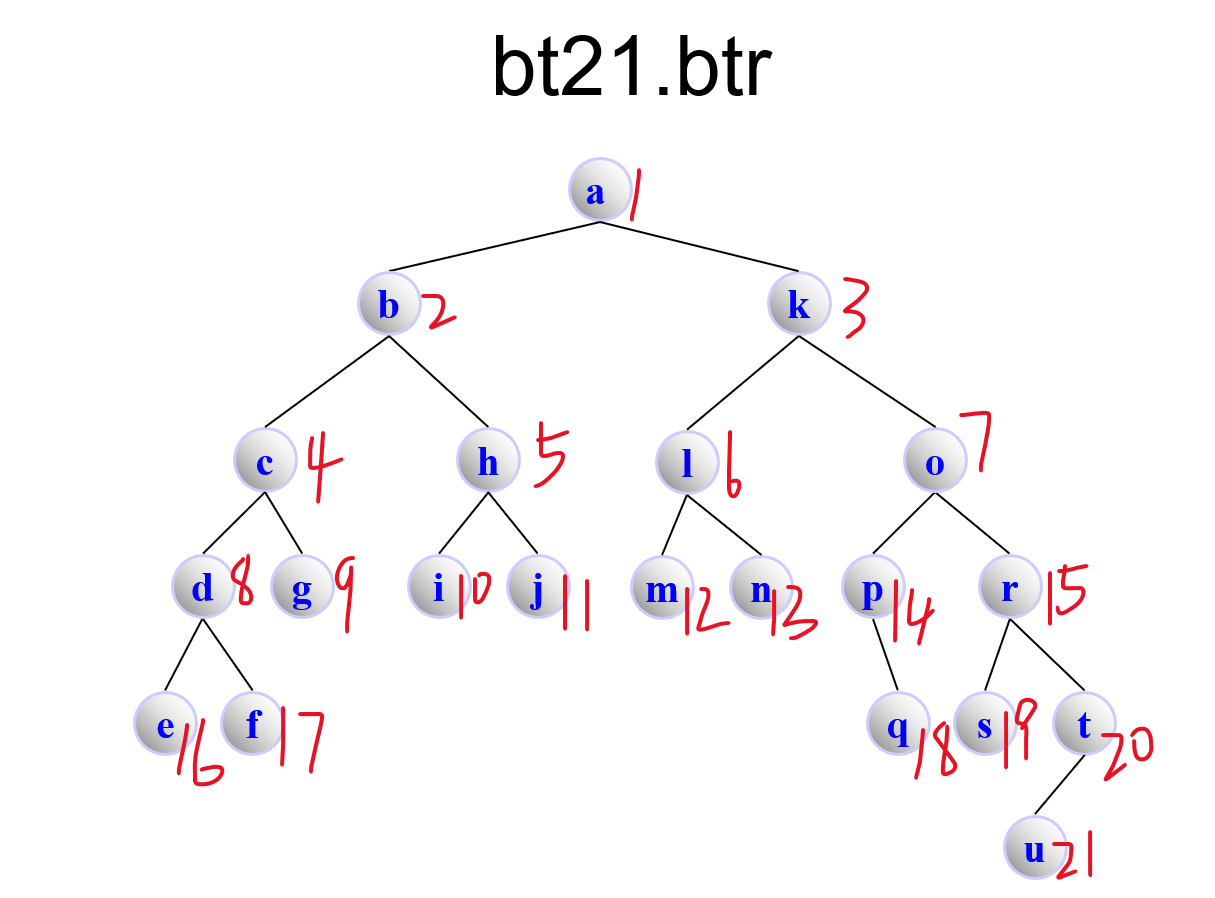
第一组数据为bt261，如图，选择结点序号11与12（最近公共祖先为a）、结点序号18与20（最近公共祖先为b）、结点序号16与17（最近公共祖先为d），预期打印结果依次为a、b、d，运行代码，查看





符合预期

第二组数据为bt21，如图，选择结点序号11与12（最近公共祖先为a）、结点序号18与21（最近公共祖先为o）、结点序号16与17（最近公共祖先为d），预期打印结果依次为a、o、d，运行代码，查看





符合预期

1. **输出二叉树从每个叶子结点到根结点的路径（经历的结点）。实验测试数据基本要求：**

**第一组数据： bt261.btr**

**第二组数据： bt21.btr**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：需要借助栈辅助，将根结点压栈，然后继续递归压左右子树（与先序遍历一致），直到遇到叶结点，打印栈中所有内容，即为其路径，打印完成后退栈进行回溯操作。

代码：

// 输出二叉树从每个叶子结点到根结点的路径（经历的结点）

Status GetLeafRoad(Tree T,Stack \*S) {

if (T == NULL) return ERROR;

Push(S, T->data); // 先压栈

if (T->leftChild == NULL && T->rightChild == NULL) {

GetStack(S); // 到叶子节点时输出路径

} else {

GetLeafRoad(T->leftChild, S);

GetLeafRoad(T->rightChild, S);

}

ElemTypes E;

Pop(S,&E); // 回溯时弹栈，撤销当前路径

return OK;

}

**运行结果截图及说明**

第一个测试数据bt261，如图，推理应该有如下的路径：

g->f->e->d->c->b->a

i->h->e->d->c->b->a

j->h->e->d->c->b->a

l->k->c->b->a

m-> k->c->b->a

o->n->b->a

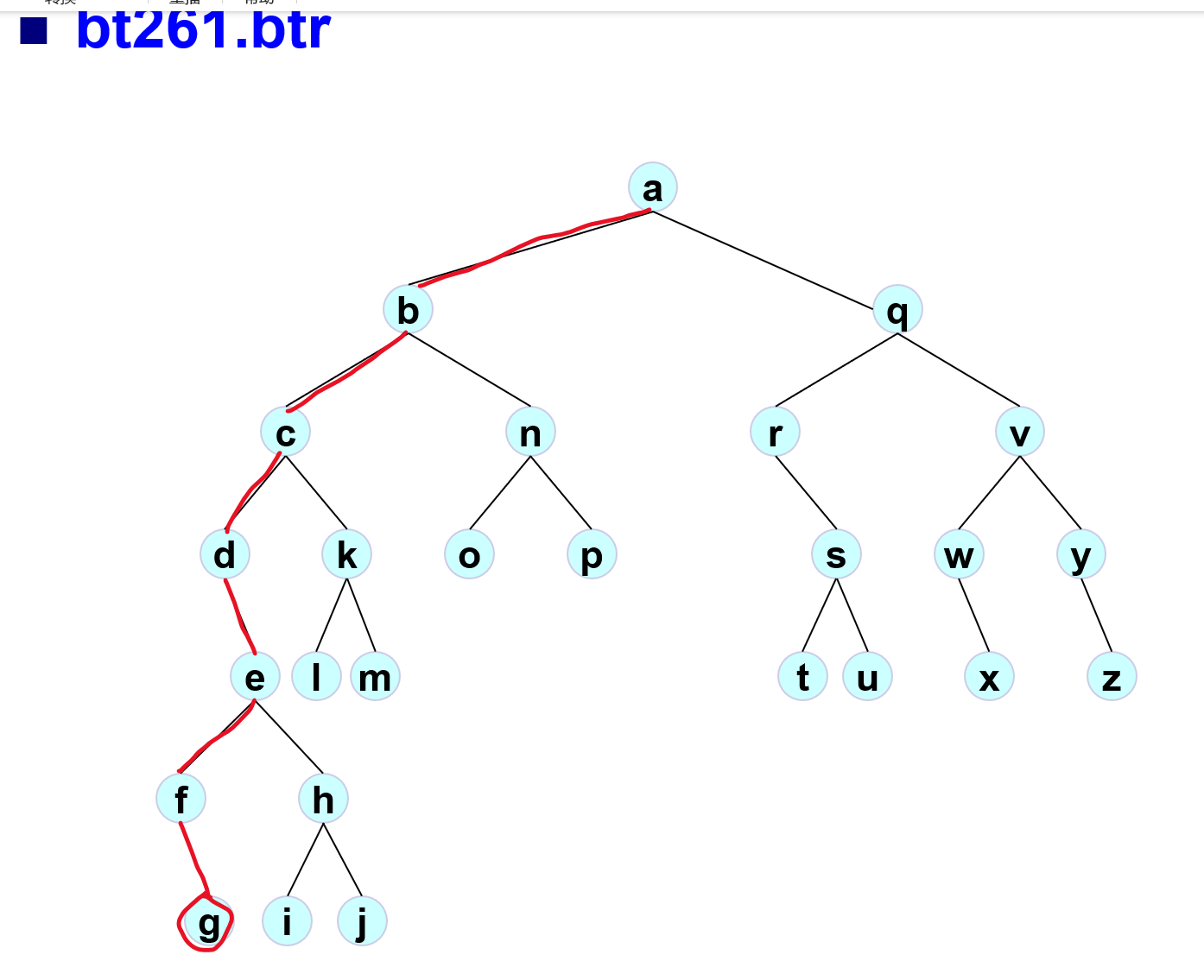
p-> n->b->a

t->s->r->q->a

u-> s->r->q->a

x->w->v->q->a

z->y->v->q->a



运行代码，打印如下：



符合预期

以第二组数据bt21测试：

预期结果：

e->d->c->b->a

f-> d->c->b->a

g->c->b->a

i->h->b->a

j->h->b->a

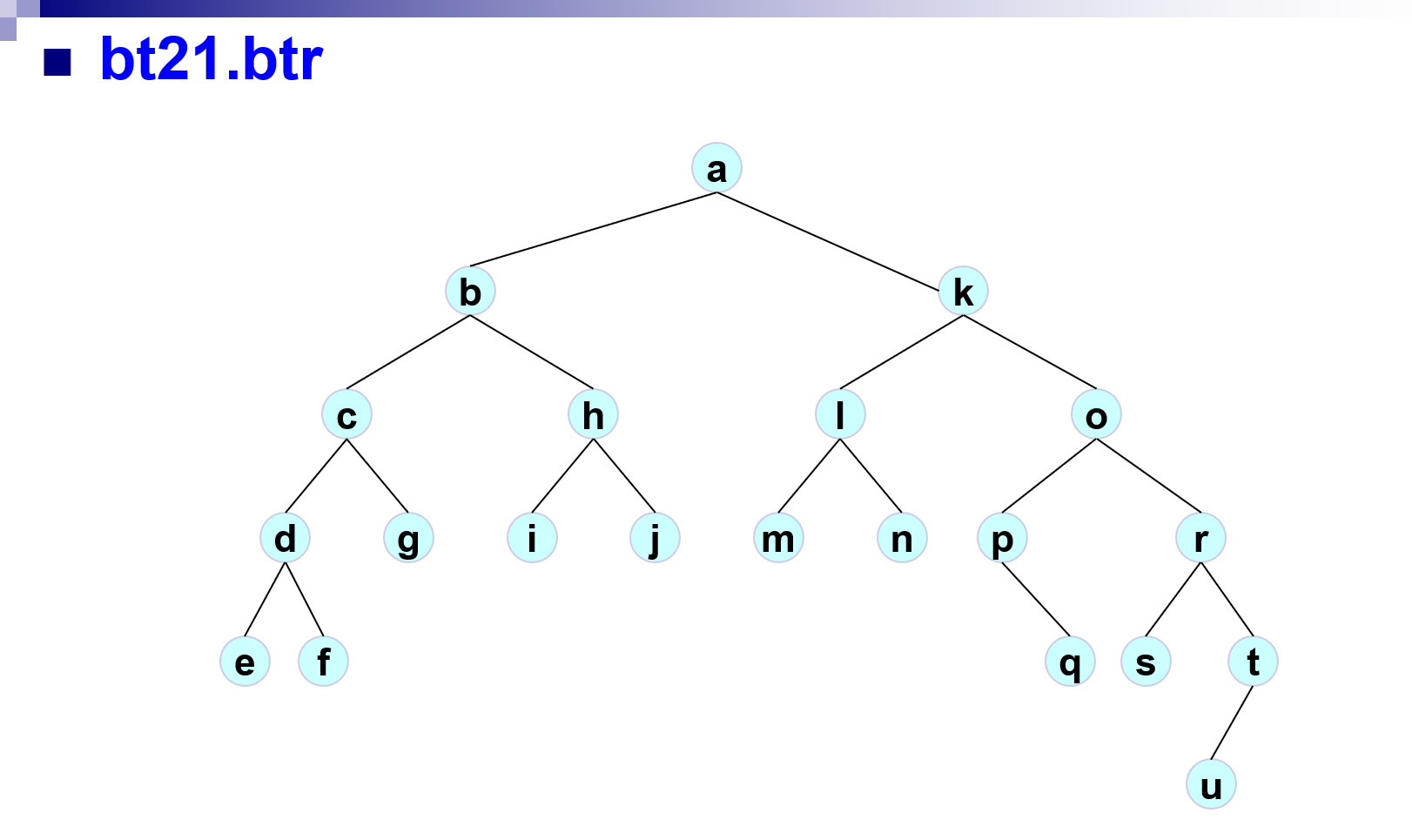
m->l->k->a

n->l->k->a

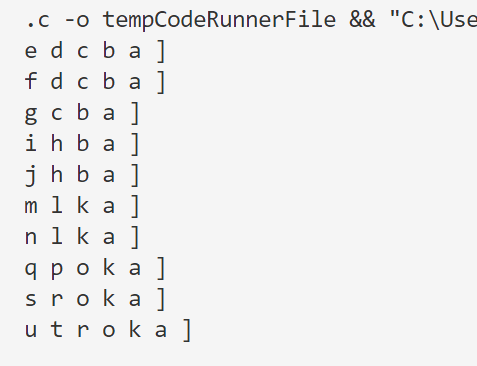
q->p->o->k->a

s->r->o->k->a

u->t->r->o->k->a



运行结果：



符合预期

**实验5与6使用的数据结构：**

树和森林数据结构和构建函数：

森林即使用一个数组存储树的根结点指针即可，普通树和二叉树根据字符串构建的方法（CreateTree）类似，使用递归先序遍历；构造森林（CreateForest）将构建好的树根结点指针放在森林数组即可，同时记录树的数量

typedef char ElemTypes;

typedef struct TreeNode {

ElemTypes data;

struct TreeNode \*firstSon,\*nextBrother;

} TreeNode,\*Tree;

//定义一个数组存储每棵树根节点，表示森林

typedef struct {

Tree root[MAXSIZES];

int treeCount;

} Forest;

//构造树，按顺序读取串，遇到#为空，按前序遍历的顺序放置树

void CreateTree(Tree \*T) {

ElemTypes ch;

//从1开始，每次递归递增，移动到下一个元素

ch=str[treeIndex++];

if(ch=='#') {

\*T=NULL;

} else {

\*T= (Tree)malloc(sizeof(TreeNode));

if(!\*T) exit(0);

//采用前序遍历的方法构造

(\*T)->data=ch; /\* 生成根结点 \*/

//构造子树

CreateTree(&(\*T)->firstSon);

CreateTree(&(\*T)->nextBrother);

}

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void InitTree(Tree \*T) {

(\*T) = NULL;

}

Status InitForest(Forest \*F) {

F->treeCount = 0;

for(int i = 0;i < MAXSIZES;i++) {

F->root[i] = NULL;

}

return OK;

}

Status CreateForest(Forest \*F) {

if(F->treeCount >= MAXSIZES) {

return ERROR;

}

Tree newTree;

InitTree(&newTree);

treeIndex = 1;

CreateTree(&newTree);

if(newTree) {

F->root[F->treeCount] = newTree;

(F->treeCount)++;

}

return OK;

}

1. **树（森林）以孩子兄弟链表存储，求森林树（森林）的度。实验测试数据基本要求：**

**第一组数据： tree11.tre**

**第二组数据： f20.tre**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：先定义一个子函数CountChildren，求当前结点的孩子数，然后TreeDegree用类似选择排序的方式，向下遍历树不断更新结点的度值，最后返回最大值；对于森林，求每棵树的度，再用相同方法选择出最大值即可；

代码：

int CountChildren(Tree T) {

int count = 0;

Tree p = T->firstSon;

while (p != NULL) {

count++;

p = p->nextBrother;

}

return count;

}

int max(int a,int b) {

return (a > b) ? a : b;

}

// 求树的度（最多有几个叉,单个结点拥有的最多孩子数）

int TreeDegree(Tree T) {

if (T == NULL) return 0;

int maxDegree = CountChildren(T);

for (Tree child = T->firstSon; child != NULL; child = child->nextBrother) {

int childDegree = TreeDegree(child);

if (childDegree > maxDegree) {

maxDegree = childDegree;

}

}

return maxDegree;

}

int ForestDegree(Forest \*F) {

int maxDegree = 0;

for (int i = 0; i < F->treeCount; i++) {

int treeDeg = TreeDegree(F->root[i]);

if (treeDeg > maxDegree) {

maxDegree = treeDeg;

}

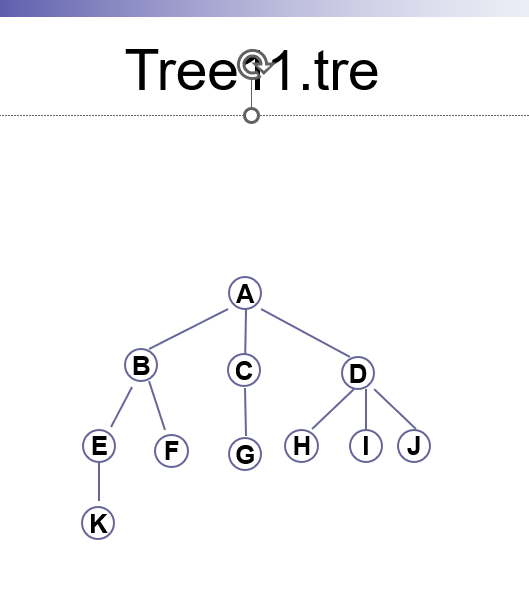
}

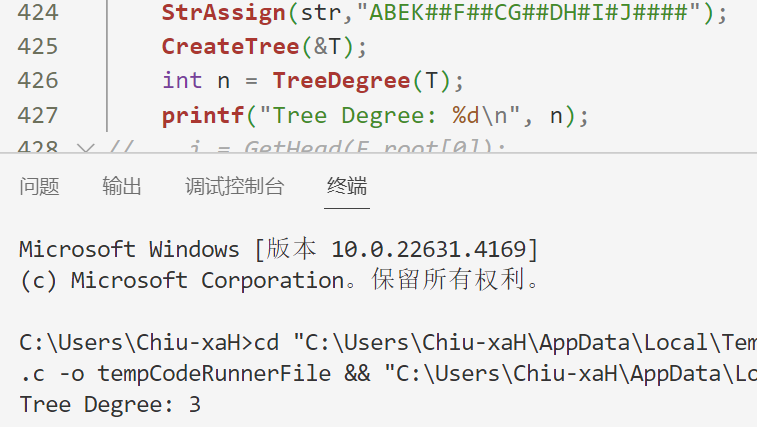
return maxDegree;

}

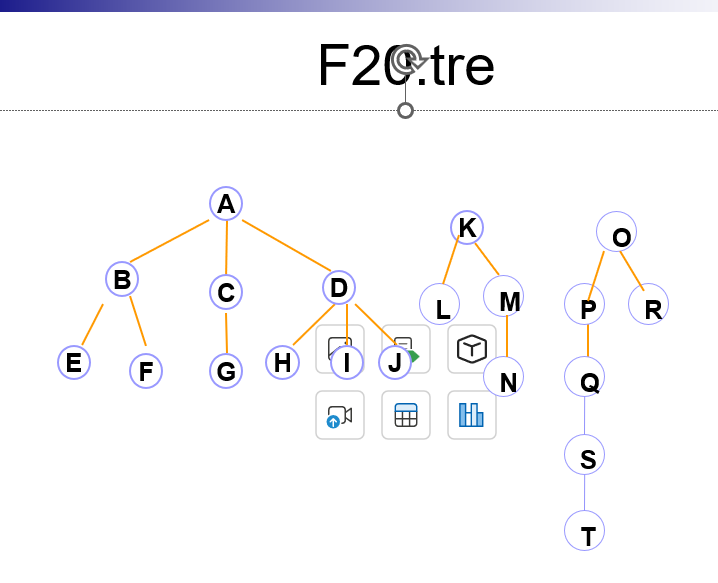
**运行结果截图及说明**

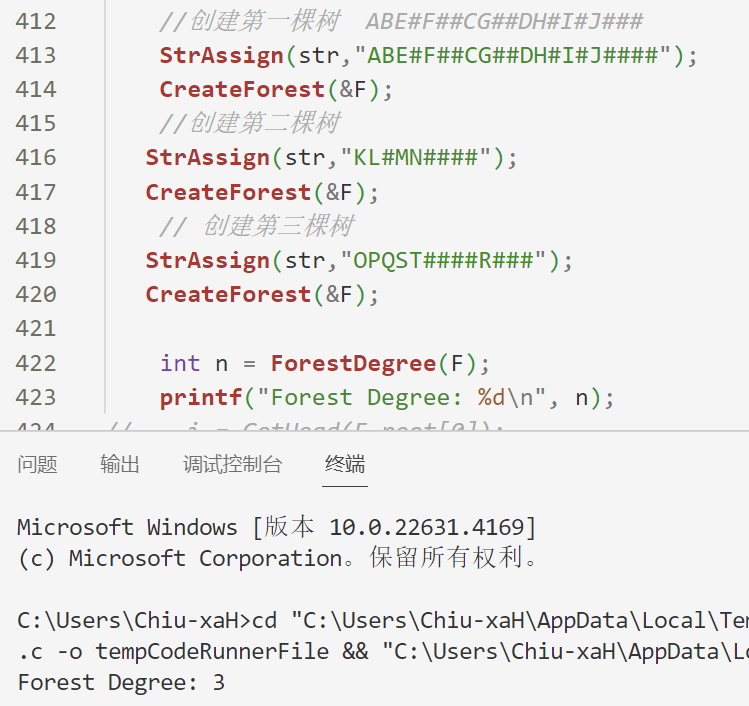
第一组数据为Tree11，由图易得度为3，运行代码，符合预期





第二组数据为F20，如图，为三棵树组成，易得度为3，运行代码，符合预期





1. **树（森林）以孩子兄弟链表存储，输出广义表表示的树（森林）。**

**例对图1所示森林，输出为：A( B(E(K),F,G),C(H,I),D(J)), L(M,N), O(P) )**

**实验测试数据基本要求：**

**第一组数据： tree11.tre**

**第二组数据： f20.tre**

**算法设计与实现描述**

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

思路：PrintTreeWithGList通过递归的方式打印一棵树，先打印根结点，然后如果存在孩子。准备打印括号，括号内为括号外结点的所有孩子，孩子的打印通过递归此函数打印firstSon，然后若存在此孩子的兄弟（即上一结点的除firstSon其他孩子）打印逗号，再继续遍历nextBrother，遍历结束后，结点所有孩子打印完成，打印回括号。在递归的过程中，所有结点都被打印到。

PrintForestWithGList通过遍历森林数组调用PrintTreeWithGList实现对森林的打印。

代码：

void PrintTreeWithGList(Tree T) {

if (!T) return;

printf("%c", T->data); // 打印根结点数据

if (T->firstSon) {

printf("(");

Tree child = T->firstSon;

while (child) {

PrintTreeWithGList(child); // 递归打印子树

if (child->nextBrother) {

printf(",");

}

child = child->nextBrother;

}

printf(")");

}

}

void PrintForestWithGList(Forest F) {

for (int i = 0; i < F.treeCount; i++) {

PrintTreeWithGList(F.root[i]);

if (i != F.treeCount - 1) {

printf(",");

}

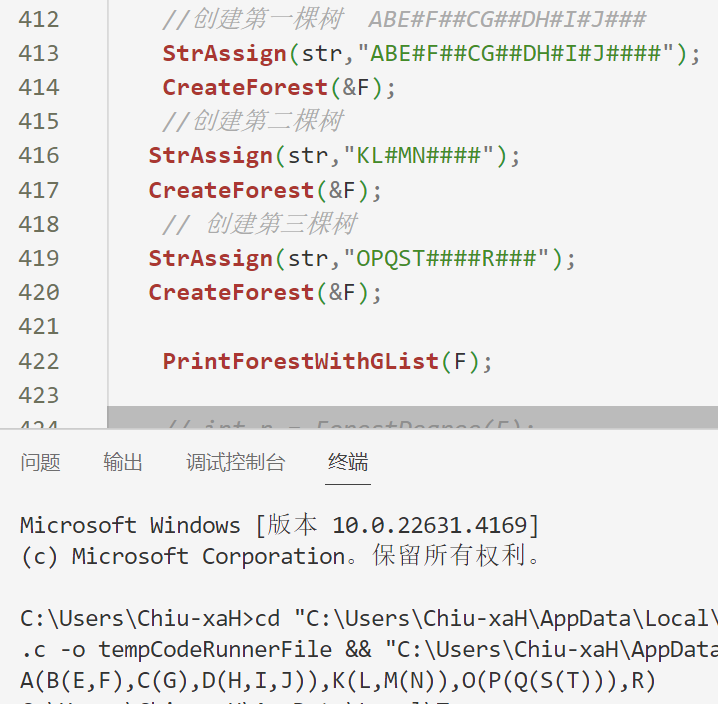
}

}

**运行结果截图及说明**

第一组数据和第二组数据分别结果如下，与树图（图片位于上一个实验）对比，符合预期





6总结、心得和建议

树是一种“有趣”的数据结构，不同于其他线性结构，它更复杂，同时也带来的更多的可能，树的应用更偏向底层，比如二分查找法使用了二叉树、解压缩算法使用哈夫曼树、数据库和文件系统为了更高效的访问查找使用B+树、一些编程语言的Map、Set等也有树的参与。