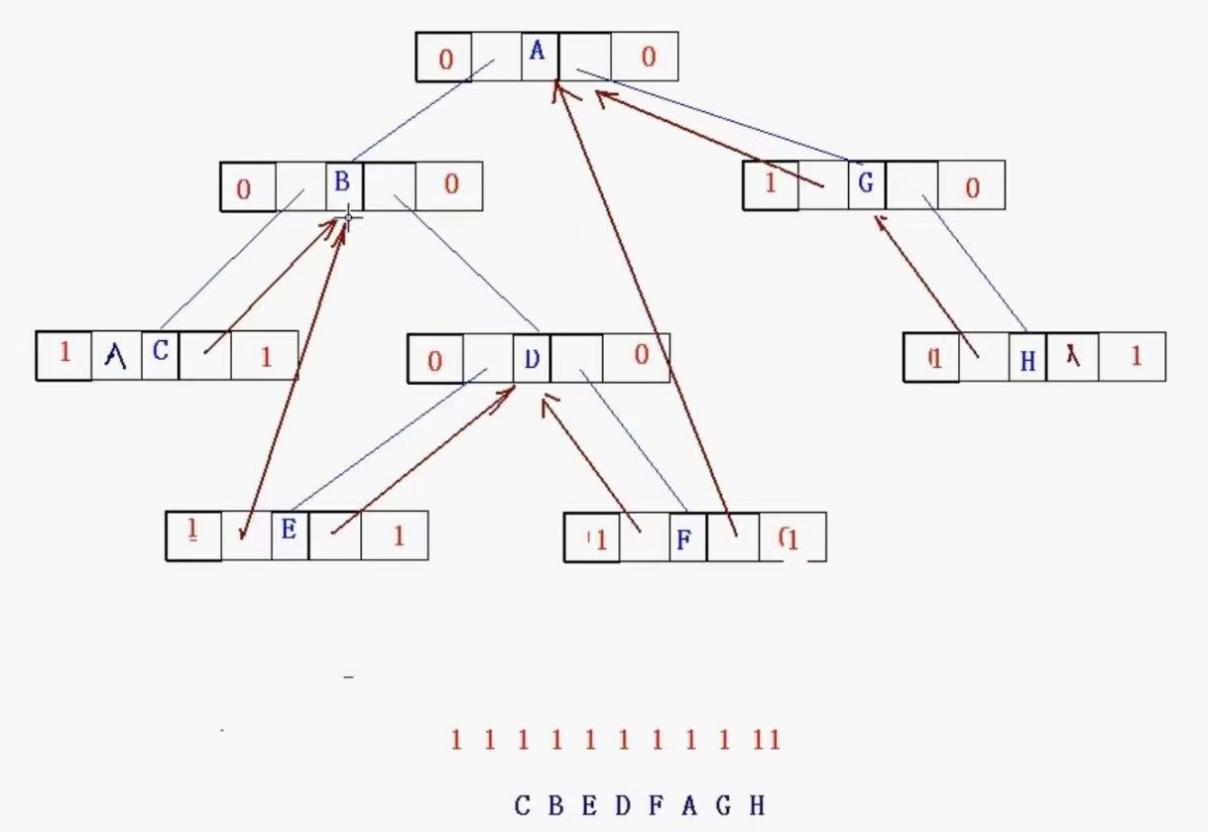
**线索二叉树的创建和方法实现**

以中序为例：C B E D F A G H

并不是每一个结点都可以找到前趋和后继，这里只是运用了“有n个结点的二叉链表存在n+1个空链域”，利用这n+1个空链域来表示前趋和后继。



#include <iostream>

#include<malloc.h>

#include<assert.h>

using namespace std;

#define ElemType char

typedef enum {LINK,THREAD}Tag\_Type; //如果是0就表示指示真实孩子结点（链），如果是1就表示指向前趋后趋

typedef struct BinTreeNode {

ElemType data;

struct BinTreeNode\* leftChild;

struct BinTreeNode\* rightChild;

Tag\_Type ltag;

Tag\_Type rtag;

};

typedef struct BinTree {

BinTreeNode\* root;

ElemType refvalue;

};

void InitBinTree(BinTree&bt,ElemType ref) {

bt.root = NULL;

bt.refvalue = ref;

}

BinTreeNode\* \_Buynode(ElemType x) {

BinTreeNode\* t = (BinTreeNode\*)malloc(sizeof(BinTreeNode));

assert(t != nullptr);

t->data = x;

t->leftChild = t->rightChild = nullptr;

t->ltag = t->rtag = LINK; //创建新的结点的时候先不分链和线，都先初始化为0（即LINK）

return t;

}

**一、线索二叉树的创建**

void CreateBinTree(BinTree& bt, BinTreeNode\*& t, const char\* &str) { //此时的\*str 其实是第一个字符 //注意这里一定要用引用？

if (\*str== bt.refvalue) {

t = NULL;

}

else {

t = \_Buynode(\*str);

CreateBinTree(bt, t->leftChild, ++str);

CreateBinTree(bt, t->rightChild, ++str);

}

}

void CreateBinTree(BinTree& bt, const char\* str) {

CreateBinTree(bt, bt.root, str);

}

//进行中序线索化(在二叉树存在的前提下)

void CreatInThread(BinTreeNode\*& t, BinTreeNode\*& pre) {

if (t ==nullptr) {

return;

}

else {

CreatInThread(t->leftChild, pre);//一直向左，当此步停止后就找到了中序前提下的第一个结点

//下面就要开始对结点进行前趋化以及对前趋进行后继化

if (t->leftChild == nullptr) {

t->ltag = THREAD;

t->leftChild = pre;

}

if (pre != NULL && pre->rightChild == nullptr) { //对右树进行后继化

pre->rtag = THREAD;

pre->rightChild = t;

}

pre = t; //此时所在的结点必为下一个访问的结点的前趋，对pre进行赋值

CreatInThread(t->rightChild, pre);

}

}

void CreatInThread(BinTree& bt) {

BinTreeNode\* pre = nullptr;

CreatInThread(bt.root, pre);

//对最后一个结点的后继信息进行处理

pre->rightChild = nullptr;

pre->rtag = THREAD;

}

**二、线索二叉树的方法实现**

//线索化二叉树求第一个结点

//如果标记是LINK，则一直向左走，如果走到某一个结点其标记为THREAD，说明走到头了

BinTreeNode\* First(BinTreeNode\* t) {

if (t == nullptr) {

return nullptr;

}

BinTreeNode\* p = t;

while (p->ltag == LINK) {

p = p->leftChild;

}

return p;

}

BinTreeNode\* First(BinTree& bt) {

return First(bt.root);

}

//线索化二叉树求最后一个结点

//如果标记是LINK，则一直向右走，如果走到某一个结点其标记为THREAD，说明走到头了

BinTreeNode\* Last(BinTreeNode\* t) {

if (t == nullptr) {

return nullptr;

}

BinTreeNode\* p = t;

while (p->rtag == LINK) {

p = p->rightChild;

}

return p;

}

BinTreeNode\* Last(BinTree& bt) {

return Last(bt.root);

}

//线索化二叉树求当前结点的后继结点

BinTreeNode\* next(BinTreeNode\* t, BinTreeNode\*cur) {

if (t == nullptr || cur == nullptr) {

return nullptr;

}

if (cur->rtag == THREAD) {

return cur->rightChild;

}

//如果标记都是LINK，按照中序序列的访问原则，能访问当前结点，说明已经访问过其左子树。

//那么当前结点的下一个结点就是其右树的第一个结点

return First(cur->rightChild);

}

BinTreeNode\* next(BinTree& bt, BinTreeNode\*cur) {

return next(bt.root,cur);

}

//线索化二叉树求当前结点的前趋结点

BinTreeNode\* Prio(BinTreeNode\* t, BinTreeNode\* cur) {

if (t == nullptr || cur == nullptr) {

return nullptr;

}

if (cur->ltag == THREAD) {

return cur->leftChild;

}

//如果标记都是LINK，按照中序序列的访问原则，能访问当前结点，说明已经访问过其左子树。

//那么当前结点的前趋结点就是其左树的最后一个结点

return Last(cur->leftChild);

}

BinTreeNode\* Prio(BinTree& bt, BinTreeNode\* cur) {

return Prio(bt.root, cur);

}

//线索二叉树的中序遍历（由于是按照中序进行线索化的）

//线索化之后不能用递归！

void InOrder(BinTreeNode\* t) {

BinTreeNode\* p;

for (p = First(t); p != nullptr; p = next(t,p)) { //借助之前创造的函数来遍历

cout << p->data;

}

cout << endl;

}

void InOrder(BinTree& bt) {

InOrder(bt.root);

}

//线索二叉树的查找

BinTreeNode\* Search(BinTreeNode\*t, ElemType key) {

if (t == NULL) {

return nullptr;

}

//这句可能是效率的提升也可能是效率的降低

//如果头结点刚好是要查找的，则直接退出了

//如果头结点不是，那么下面的循环会再验证一次头结点，降低了效率

if (t->data == key) {

return t;

}

//借助之前创造的遍历函数

BinTreeNode\* p;

for (p = First(t); p != nullptr; p = next(t, p)) {

if (p->data == key) {

return p;

}

}

return NULL;

}

BinTreeNode\* Search(BinTree& bt,ElemType key) {

return Search(bt.root, key);

}

//求父结点(复杂)

BinTreeNode\* Parent(BinTreeNode\* t, BinTreeNode\* cur) {

if (t == nullptr || cur == nullptr) {

return NULL;

}

if (t == cur) {

return nullptr;

}

BinTreeNode\* p;

if (cur->ltag == THREAD) {

p = cur->leftChild;

if (p->rightChild == cur) {

//这里为什么是右树呢？

//因为p是当前结点的前趋，当前结点在p的后面，如果p是当前结点的根结点，那么根据“左根右”的顺序

//当前结点应该为根结点的右子树

return p;

}

}

if (cur->rtag == THREAD) {

p = cur->rightChild;

if (p->leftChild == cur) {

return p;

}

}

//对于普通的结点来说，先顺着左树一直往下，知道左树标记=LINK，则顺着线索找到前趋，看前趋结点的右树是否等于我的当前的普通结点

//因为按照左根右的顺序，根结点遍历过后，就轮到右树的最左边的结点

//所以右树最左边的结点的前趋就是该最左边结点所在的右树的父结点，当前结点又是小右树的根结点，因此当前结点的父结点就是小右树的父结点

p = First(cur->leftChild);

p = p->leftChild;

if (p != nullptr &&p->rightChild == cur) { //判空条件是针对本树中的C而言的

return p;

}

//如果之前的方法搜索失败

p = Last(cur->rightChild);

return p = p->rightChild;

}

BinTreeNode\* Parent(BinTree& bt, BinTreeNode\* cur) {

return Parent(bt.root, cur);

}

**测试函数**

int main() {

const char\* str = "ABC##DE##F##G##H##";

BinTree mytree;

InitBinTree(mytree,'#');

CreateBinTree(mytree, str);

CreatInThread(mytree);

//BinTreeNode\* p = First(mytree);

//BinTreeNode\* p = Last(mytree);

BinTreeNode\* p = Search(mytree,'D');

BinTreeNode\* q = next(mytree,p);

return 0;

}