

13 | 线索二叉树：如何线索化二叉树以提升访问速度？

2023-03-13 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

今天我要和你分享的主题是“线索二叉树”。

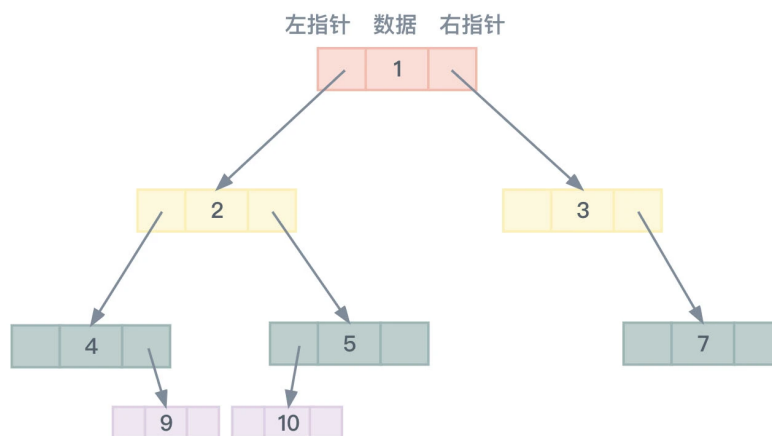
和传统二叉树相比，线索二叉树可以进一步提高访问二叉树节点的速度，从而提高访问二叉树的效率，当然，“线索”这个概念的引入也意味着要对原来的二叉树实现代码做出相应的修改。

shikey.com转载分享

那么，什么是线索？要怎么把传统二叉树转变为线索二叉树？带着这些问题，我们先从线索二叉树的概念开始说起，进而探讨如何对二叉树进行线索化，以及在线索二叉树上的操作问题。

线索二叉树的基本概念

我们先来看一个简单的二叉树链式存储。



极客时间

图 1 二叉树链式存储方式的结构示意图

在图 1 所示二叉树的链式存储中，每个二叉树节点（BinaryTreeNode）都包含两个指针域，分别是左子节点指针（leftChild）和右子节点指针（rightChild）。但一个二叉树中往往某些节点并没有左孩子或者右孩子，尤其是叶子节点，根本就不存在左孩子和右孩子。

这张图里一共有 8 个节点，除了根节点外，**都会有一根来自其他节点的指针指向该节点自身**。仔细数下来，对于 8 个节点的二叉树，虽然一共有 16 个指针域，但实际使用了的只有 7 个。

总结一下，就是如果一个二叉树有 n 个节点，因为一个节点结构包含两个指针域，所以一共有 $2n$ 个指针域，但是，其中只使用了 $n-1$ 个指针域指向二叉树的各个子节点。

我们可以算一下，一共有 $2n$ 个指针域，只使用了 $n-1$ 个指针域，那么有多少个指针域没有使用呢？

$$2n - (n-1) = n+1$$

shikekey.com 转载分享

这 $n+1$ 个指针域就这样白白浪费了。所以，为了物尽其用，我们可以用这 $n+1$ 个指针域记录一些有用的信息。

那要记录什么，又要怎么记录呢？

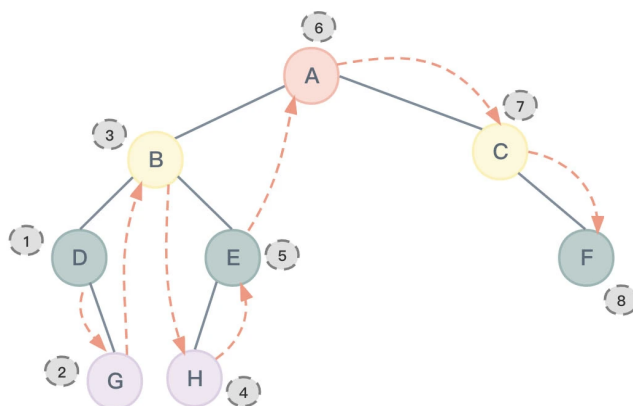


图 2 中序遍历访问的二叉树节点顺序

图 2 所示的这棵二叉树中序遍历（前序、后序遍历也同样道理）的顺序为 DGBHEACF。

在这个序列中，除 D 和 F 节点外，其他节点都有且只有一个直接前趋和一个直接后继。比如 B 的前趋是 G，B 的后继是 H，而 D 节点没有前趋，F 节点没有后继。

问题来了，在存储起一棵二叉树后，我们可以**通过节点 E 的左孩子指针迅速找到节点 H，但并没有办法通过节点 E 迅速找到其在中序遍历序列中的后继节点 A。**

也就是说，只能通过节点的左右孩子指针找到该节点的左右孩子，但没有办法直接得到某个节点在某个遍历序列中的前趋或者后继节点，这种前趋或者后继节点的信息只能在遍历的过程中**动态得到**。

不过，既然节点 E 没有右孩子，那么节点 E 的右孩子指针可以用来记录后继节点 A，也就是指向 A。这样就可以通过节点 E 迅速找到节点 A 了。

shikey.com转载分享

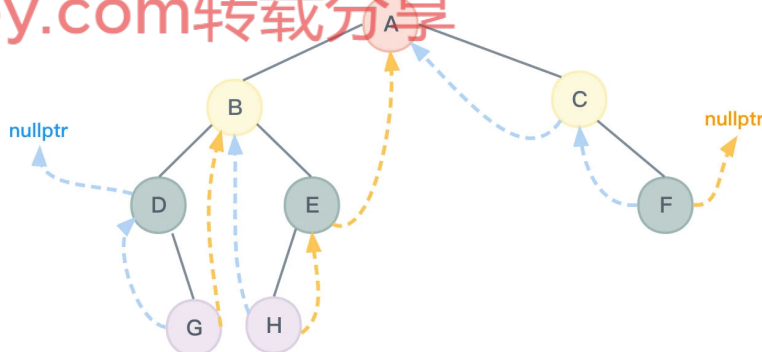


图 3 按照中序遍历序列对一棵二叉树进行线索化，其中虚线表示的是线索

举个例子。在图 3 中：

G 节点的左右子节点指针都没有被使用，因此可以用左节点指针指向 G 节点在中序遍历序列中的前趋节点 D，右节点指针指向 G 节点在中序遍历中的后继节点 B。

D 节点的左子节点指针没有被使用，因此可以用于指向 D 节点在中序遍历序列中的前趋节点，但因为在中序遍历序列中 D 没有前趋节点，因此该指针只能指向 nullptr。

H 节点的左节点指针指向前趋节点 B，右节点指针指向后继节点 E。

E 节点的右指针指向后继节点 A。

F 节点的左节点指针指向前趋节点 C，右节点指针指向后继节点，但因为 F 节点没有后继节点，所以该指针只能指向 nullptr。

C 节点的左节点指针指向前趋节点 A。

总结一下，这些**指向前趋节点和后继节点的指针就叫做线索**。由节点的左孩子指针充当指向前趋节点的线索，由节点的右孩子指针充当指向后继节点的线索。


我们把加上了线索的二叉树称为**线索二叉树**。对二叉树进行某种序列的遍历使其成为一棵线索二叉树的过程称为二叉树的**线索化**。

线索二叉树对于节点的插入、删除、查找前趋后继等都会变得更加方便，**解决了存储空间的浪费问题，也解决了前趋和后继节点的记录问题**，这就是建立这些线索的意义。

当然，线索二叉树也分为**前序、中序、后序**线索二叉树，而且，因为二叉树还可以层序遍历，因此也存在层序线索二叉树。话说回来，前序线索二叉树是以前序遍历序列为依据对二叉树进行线索化，而中序、后序线索二叉树当然是分别以中序、后序遍历序列为依据对二叉树进行线索化。

如何对二叉树进行线索化

要编写线索二叉树的实现代码，得先看一看线索二叉树中每个节点的结构是什么样的。我们先回顾一下以往二叉树链式存储时树中的每个节点的定义。

 复制代码

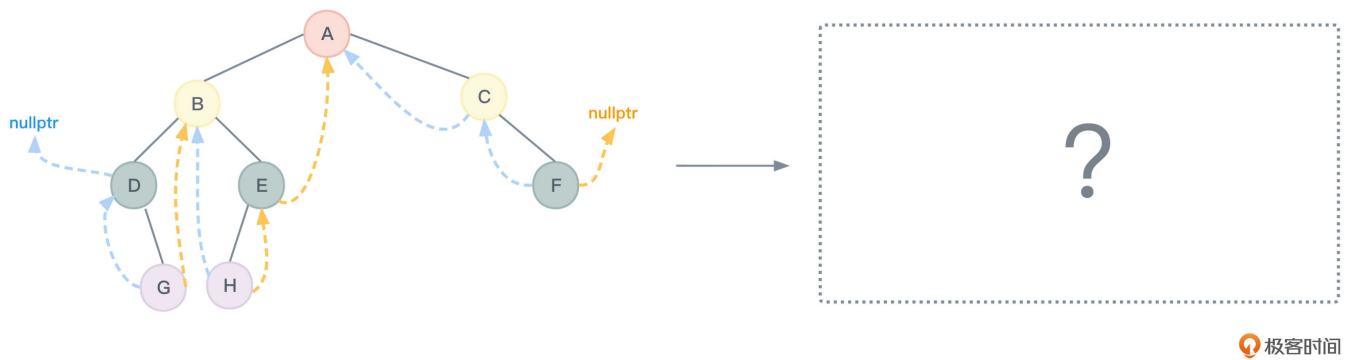
```
1 //树中每个节点的定义
2 template <typename T> //T 代表数据元素的类型
3 struct BinaryTreeNode
4 {
5     T data; //数据域，存放数据元素
6     BinaryTreeNode* leftChild, //左子节点指针
7     * rightChild; //右子节点指针
8 };
```

而在线索二叉树中，左子节点指针和右子节点指针指向的可能是前趋和后继节点，这就需要为 BinaryTreeNode 结构增加两个标志变量，用于标记指针保存的是左、右子节点还是前趋、后继节点，下面是修改后的 BinaryTreeNode 结构。

 复制代码

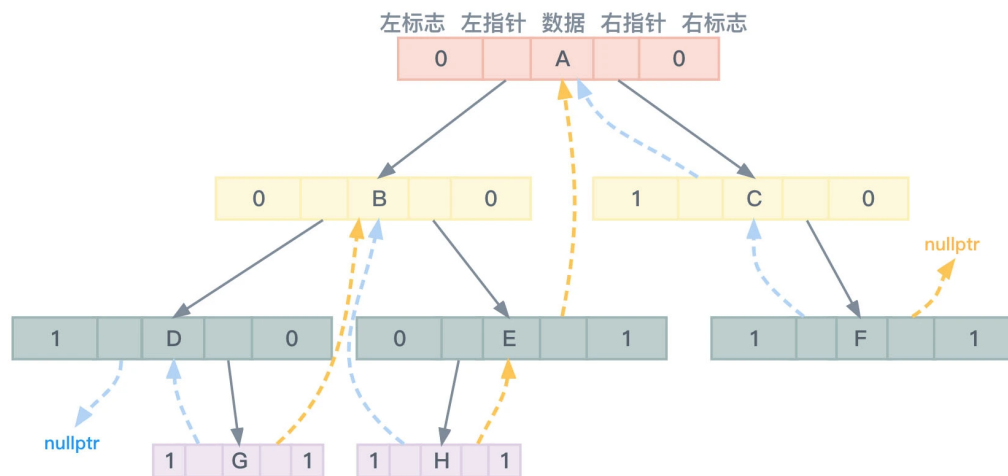
```
1 //树中每个节点的定义
2 template <typename T> //T 代表数据元素的类型
3 struct BinaryTreeNode
4 {
5     T data; //数据域，存放数据元素
6     BinaryTreeNode* leftChild, //左子节点指针
7     * rightChild; //右子节点指针
8     int8_t leftTag, //左标志 = 0 表示 leftChild 指向的是左子节点, =1 表示
9     rightTag; //右标志 = 0 表示 rightChild 指向的是右子节点, =1 表示
10 };
```

我们通常将二叉树存储结构称为**二叉链表**，而在这里，把这种修改后的、适用于线索二叉树的存储结构称为**线索链表**。想一想，对于图 3 所示的线索二叉树对应的链式存储方式结构示意图应该怎么画呢？



极客时间

结果如图 4 所示：



极客时间

图 4 中序遍历的线索二叉树对应的链式存储方式结构示意图

图中，0 表示指向的是左 / 右子节点，1 表示指向的是前趋 / 后继节点。


二叉树的线索化过程针对的是二叉树节点的空指针，也就是我们之前说过的“n+1”个没有左孩子或者右孩子的指针。

换句话说，在线索化一棵二叉树的过程中，我们并没有把所有节点的前趋和后继节点都记录下来，只是利用了 n+1 个空指针域进行前趋和后继节点的记录。

那么，对于没有记录其前趋和后继的节点，就要通过一些其他方法寻找其前趋和后继了。

用中序遍历序列线索化二叉树

我们先静下心来理解一下线索二叉树的定义和实现代码。

 复制代码

```
1 //线索二叉树的定义
2 template <typename T>
3 class ThreadBinaryTree
4 {
5 public:
6     ThreadBinaryTree(); //构造函数
7     ~ThreadBinaryTree(); //析构函数
8 public:
9     //利用扩展二叉树的前序遍历序列来创建一棵二叉树
10    void CreateBTreeAccordPT(char* pstr);
11 private:
12    //利用扩展二叉树的前序遍历序列创建二叉树的递归函数
13    void CreateBTreeAccordPTRecu(BinaryTreeNode<T>*& tnode, char*& pstr); //参数为引用
14 public:
15    //在二叉树中根据中序遍历序列创建线索
16    void CreateThreadInBTreeAccordIO();
17 private:
18    void CreateThreadInBTreeAccordIO(BinaryTreeNode<T>*& tnode, BinaryTreeNode<T>*&
19 private:
20    void ReleaseNode(BinaryTreeNode<T>* pnode);
21 private:
22    BinaryTreeNode<T>* root; //树根指针
23 };
24
25 //构造函数
26 template<class T>
27 ThreadBinaryTree<T>::ThreadBinaryTree()
28 {
29     root = nullptr;
30 }
31
32 //析构函数
33 template<class T>
34 ThreadBinaryTree<T>::~~ThreadBinaryTree()
35 {
36     ReleaseNode(root);
37 };
38
39 //释放二叉树节点
40 template<class T>
41 void ThreadBinaryTree<T>::ReleaseNode(BinaryTreeNode<T>* pnode)
42 {
```

```

43     if (pnode != nullptr)
44     {
45         if(pnode->leftTag == 0)
46             ReleaseNode(pnode->leftChild); //只有真的需要delete的节点, 才会递归调用ReleaseNo
47         if (pnode->rightTag == 0)
48             ReleaseNode(pnode->rightChild); //只有真的需要delete的节点, 才会递归调用ReleaseN
49     }
50     delete pnode;
51 }
52
53 //利用扩展二叉树的前序遍历序列来创建一棵二叉树
54 template<class T>
55 void ThreadBinaryTree<T>::CreateBTreeAccordPT(char* pstr)
56 {
57     CreateBTreeAccordPTRecu(root, pstr);
58 }
59
60 //利用扩展二叉树的前序遍历序列创建二叉树的递归函数
61 template<class T>
62 void ThreadBinaryTree<T>::CreateBTreeAccordPTRecu(BinaryTreeNode<T>*& tnode, char
63 {
64     if (*pstr == '#')
65     {
66         tnode = nullptr;
67     }
68     else
69     {
70         tnode = new BinaryTreeNode<T>; //创建根节点
71         tnode->leftTag = tnode->rightTag = 0; //标志先给0
72         tnode->data = *pstr;
73         CreateBTreeAccordPTRecu(tnode->leftChild, ++pstr); //创建左子树
74         CreateBTreeAccordPTRecu(tnode->rightChild, ++pstr); //创建右子树
75     }
76 }
77
78 //在二叉树中根据中序遍历序列创建线索
79 template<class T>
80 void ThreadBinaryTree<T>::CreateThreadInBTreeAccordIO()
81 {
82     BinaryTreeNode<T>* pre = nullptr; //记录当前所指向的节点的前趋节点 (刚开始的节点没有前
83
84     CreateThreadInBTreeAccordIO(root, pre);
85
86     //注意处理最后一个节点的右孩子, 因为这个右孩子还没处理
87     pre->rightChild = nullptr; //这里之所以直接给nullptr, 是因为中序遍历访问顺序是左根右, 所
88     pre->rightTag = 1; //线索化
89 }
90
91 template<class T>

```



```

92 void ThreadBinaryTree<T>::CreateThreadInBTreeAccordIO(BinaryTreeNode<T>*& tnode,
93 {
94     if (tnode == nullptr)
95         return;
96
97     //中序遍历序列（左根右），递归顺序非常类似于中序遍历
98     CreateThreadInBTreeAccordIO(tnode->leftChild, pre);
99
100    if (tnode->leftChild == nullptr) //找空闲的指针域进行线索化
101    {
102        tnode->leftTag = 1; //线索
103        tnode->leftChild = pre; //如果leftChild ==nullptr, 说明该节点没有前趋节点
104    }
105
106    //这个前趋节点的后继节点肯定是当前这个节点tnode
107    if (pre != nullptr && pre->rightChild == nullptr)
108    {
109        pre->rightTag = 1; //线索
110        pre->rightChild = tnode;
111    }
112    pre = tnode; //前趋节点指针指向当前节点
113
114    CreateThreadInBTreeAccordIO(tnode->rightChild, pre);
115 }

```

依据这些代码，我们就可以创建线索二叉树了。创建线索二叉树可以分成两个步骤。

1. **用任何方法创建二叉树。** 这里将采用前面使用过的利用扩展二叉树的前序遍历序列创建图 3 所示的二叉树。

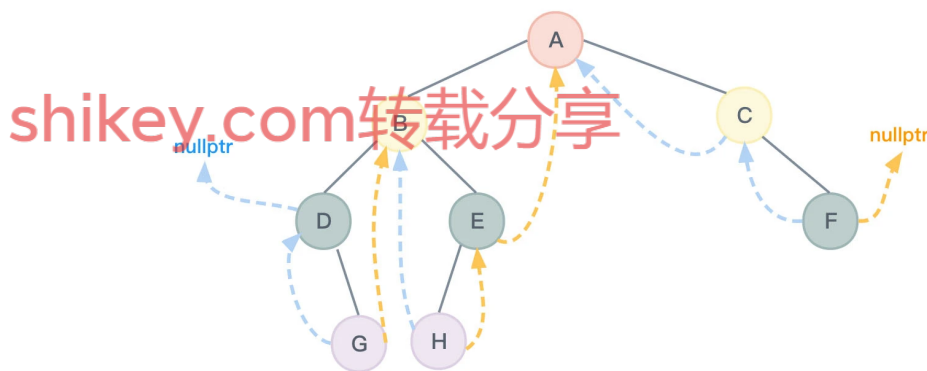



图3回顾 按照中序遍历序列对一棵二叉树进行线索化，其中虚线表示的是线索

2. 将该二叉树按照中序（也可以是前序、后序）遍历序列**创建线索**，这会用到 `CreateThreadInBTreeAccordIO` 成员函数。

在 `main` 主函数，加入下面的代码来创建并按中序遍历线索化一棵二叉树。

 复制代码


```
1 ThreadBinaryTree<int> mythreadtree;
2 mythreadtree.CreateBTreeAccordPT((char*)"ABD#G##EH###C#F##"); //利用扩展二叉树的前序
3 //对二叉树进行线索化(根据中序遍历序列创建线索)
4 mythreadtree.CreateThreadInBTreeAccordIO();
```

从上面的代码不难看到，根据中序遍历序列线索化二叉树的过程其实与中序遍历的过程非常类似，只不过是在遍历的过程中进行线索化而已。

用前序遍历序列线索化二叉树

前面的代码演示了在二叉树中根据中序遍历序列线索化二叉树。那么如果根据前序遍历序列线索化二叉树代码应该是什么样的呢？

可以参照上述的 `CreateThreadInBTreeAccordIO` 成员函数来书写名字叫做 `CreateThreadInBTreeAccordPO` 的新成员函数来实现，参考下面的代码。

 复制代码

```
1 //在二叉树中根据前序遍历序列创建线索
2 template<class T>
3 void ThreadBinaryTree<T>::CreateThreadInBTreeAccordPO()
4 {
5     BinaryTreeNode<T>* pre = nullptr;
6     CreateThreadInBTreeAccordPO(root, pre);
7     pre->rightChild = nullptr;
8     pre->rightTag = 1;
9 }
10 template<class T>
11 void ThreadBinaryTree<T>::CreateThreadInBTreeAccordPO(BinaryTreeNode<T>*& tnode,
12 {
13     if (tnode == nullptr)
14         return;
15 }
```

```

16 //前遍历序列（根左右），递归顺序非常类似于前序遍历
17 if (tnode->leftChild == nullptr) //找空闲的指针域进行线索化
18 {
19     tnode->leftTag = 1; //线索
20     tnode->leftChild = pre; //如果leftChild ==nullptr, 说明该节点没有前趋节点
21 }
22
23 //这个前趋节点的后继节点肯定是当前这个节点tnode
24 if (pre != nullptr && pre->rightChild == nullptr)
25 {
26     pre->rightTag = 1; //线索
27     pre->rightChild = tnode;
28 }
29 pre = tnode; //前趋节点指针指向当前节点
30
31 if(tnode->leftTag == 0) //当leftChild是真正的子节点而不是线索化后的前趋节点时
32     CreateThreadInBTreeAccordPO(tnode->leftChild, pre);
33
34 if (tnode->rightTag == 0) //当rightChild是真正的子节点而不是线索化后的后继趋节点时
35     CreateThreadInBTreeAccordPO(tnode->rightChild, pre);
36 }

```


前序遍历序列化线索二叉树代码与中序遍历序列线索化二叉树的代码有一些非常不同的地方。

因为前序遍历访问节点的顺序是“根、左、右”，所以在处理根的时候，很可能已经线索化了根节点的 leftChild 和 rightChild 指针。

那么我们在对左右子节点进行递归调用 CreateThreadInBTreeAccordPO 时，就一定要判断这些子节点指针指向的是否为真正的子节点，而不是线索化后的前趋或者后继节点。否则就可能写出错误的实现代码导致程序执行产生异常。

main 主函数中，可以把下面的代码行：

shikey.com转载分享

 复制代码

```
1 mythreadtree.CreateThreadInBTreeAccordIO();
```

修改为：

```
1 mythreadtree.CreateThreadInBTreeAccordPO();
```

以根据前序遍历序列线索化二叉树，测试完成后记得修改回如下代码行以免对后序测试产生影响。

```
1 mythreadtree.CreateThreadInBTreeAccordIO();
```

在线索二叉树上进行的操作

二叉树线索化之后，因为左右子节点指针保存的数据发生了改变，所以，许多针对原有二叉树进行操作的代码也必须做出相应的改变。

我们将通常会进行的操作分为 4 类：找第一个和最后一个节点，找某个节点的前趋和后继节点，对线索二叉树进行遍历，以及查找线索二叉树的节点。

操作 1：找线索二叉树的第一个和最后一个节点

由于线索二叉树也分前序、中序和后序，因此找的第一个和最后一个节点肯定指的是相应顺序的节点。比如，如果线索二叉树是用中序遍历序列来线索化的，那么找的第一个和最后一个节点肯定指的是该二叉树进行中序遍历时的第一个和最后一个节点。

以中序遍历序列线索化的二叉树为例看看代码的书写，在 ThreadBinaryTree 类模板的定义中，增加 GetFirst_IO() 和 GetLast_IO() 成员函数 ([参见课件](#)) 即可。

shikey.com 转载分享

操作 2：找线索二叉树中某个节点的前趋和后继节点

以**中序遍历**序列线索化的二叉树为例看代码的书写，在 ThreadBinaryTree 类模板的定义中，可以增加 GetNextPoint_IO() 和 GetPriorPoint_IO() 成员函数 ([参见课件](#))。

至于**前序遍历**序列线索化的二叉树找前趋和后继节点，我们增加 GetNextPoint_PO() 和 GetPriorPoint_PO() 成员函数 ([参见课件](#)) 即可。

操作 3：对线索二叉树进行遍历

要注意的是，传统二叉树遍历的相关代码（递归遍历）并不适用于线索二叉树，因为递归遍历时只遍历真正的孩子节点。所以，我们要对这些遍历代码做出一定的修改。


先来看一下传统递归遍历的代码。

 复制代码

```
1 //传统中序递归遍历来遍历线索二叉树：
2 void inOrder_Org()
3 {
4     inOrder_Org(root);
5 }
6 void inOrder_Org(BinaryTreeNode<T>* tNode) //中序遍历二叉树
7 {
8     if (tNode != nullptr) //若二叉树非空
9     {
10         //左根右顺序
11         if(tNode->leftTag == 0) //是真正的左孩子
12             inOrder_Org(tNode->leftChild); //递归方式中序遍历左子树
13
14         cout << (char)tNode->data << " "; //输出节点的数据域的值
15
16         if (tNode->rightTag == 0) //是真正的右孩子
17             inOrder_Org(tNode->rightChild); //递归方式中序遍历右子树
18     }
19 }
```

我们以中序遍历序列线索化的二叉树为例，所进行的遍历也要求是中序遍历。线索化后的二叉树的遍历方式与以往二叉树的遍历方式不同，必须要充分利用这些线索来增加遍历的效率。

具体的操作方式可以参考如下代码。

 复制代码

```
1 //中序遍历按照“中序遍历序列线索化的二叉树”线索二叉树
2 void inOrder_IO()
```

```

3 {
4     inOrder_IO(root);
5 }
6 void inOrder_IO(BinaryTreeNode<T>* tNode)
7 {
8     BinaryTreeNode<T>* tmpNode = GetFirst_IO(tNode);
9     while (tmpNode != nullptr) //从第一个节点开始一直找后继即可
10    {
11        cout << (char)tmpNode->data << " "; //输出节点的数据域的值
12        tmpNode = GetNextPoint_IO(tmpNode);
13    }
14 }

```

思考一下，如果进行中序逆向遍历是不是也可以做到了呢？这就需要用到 GetLast_IO 和 GetPriorPoint_IO。参考如下代码。


 复制代码

```

1 //逆向中序遍历按照“中序遍历序列线索化的二叉树”线索二叉树
2 void revInOrder_IO()
3 {
4     revInOrder_IO(root);
5 }
6 void revInOrder_IO(BinaryTreeNode<T>* tNode)
7 {
8     BinaryTreeNode<T>* tmpNode = GetLast_IO(tNode);
9     while (tmpNode != nullptr) //从第一个节点开始一直找后继即可
10    {
11        cout << (char)tmpNode->data << " "; //输出节点的数据域的值
12        tmpNode = GetPriorPoint_IO(tmpNode);
13    }
14 }

```

通过代码可以看到，只要能成功找到前趋和后继节点，进行遍历是一件很容易的事。你也可以在 main 主函数中加入如下代码进行测试。

 复制代码

```


1 mythreadtree.inOrder_Org(); //传统中序递归遍历
2 cout << endl;
3 mythreadtree.inOrder_IO();
4 cout << endl;
5 mythreadtree.revInOrder_IO();

```

```
6 cout << endl;
```

操作 4：对线索二叉树的节点进行查找


这里用中序遍历序列线索化的二叉树为例来讲解。参考如下代码。

 复制代码

```
1 //中序遍历序列线索化的二叉树查找某个节点(假设二叉树的节点各不相同)
2 BinaryTreeNode<T>* SearchElem_IO(const T& e)
3 {
4     return SearchElem_IO(root, e);
5 }
6 BinaryTreeNode<T>* SearchElem_IO(BinaryTreeNode<T>* tNode, const T& e)
7 {
8     if (tNode == nullptr)
9         return nullptr;
10    if (tNode->data == e) //从根开始找
11        return tNode;
12
13    //这里的代码取自于 中序遍历按照“中序遍历序列线索化的二叉树”线索二叉树inOrder_IO()的代码
14    BinaryTreeNode<T>* tmpNode = GetFirst_IO(tNode);
15    while (tmpNode != nullptr) //从第一个节点开始一直找后继即可
16    {
17        if (tmpNode->data == e)
18            return tmpNode;
19        tmpNode = GetNextPoint_IO(tmpNode);
20    }
21    return nullptr;
22 }
```

可以在 main 主函数中加入下面的代码进行测试。

shikey.com转载分享

 复制代码

```
1 int val = 'B';
2 BinaryTreeNode<int>* p = mythreadtree.SearchElem_IO(val);
3 if (p != nullptr)
4 {
5     cout << "找到了值为" << (char)val << "的节点" << endl;
6
7     //顺便找下后继和前趋节点
8     BinaryTreeNode<int>* nx = mythreadtree.GetNextPoint_IO(p);
9     if(nx != nullptr)
```



```

10     cout << "后继节点值为" << (char)nx->data << "。" << endl;
11
12     BinaryTreeNode<int>* pr = mythreadtree.GetPriorPoint_IO(p);
13     if(pr != nullptr)
14         cout << "前趋节点值为" << (char)pr->data << "。" << endl;
15
16 }
17 else
18     cout << "没找到值为" << (char)val << "的节点" << endl;

```

执行结果为：


找到了值为 B 的节点

后继节点值为 H。

前趋节点值为 G。

思考一下，对于“前序遍历序列线索化的二叉树”和“后序遍历序列线索化的二叉树”，能通过上述代码进行一定的改造来查找节点吗？

前序遍历序列线索化的二叉树是可以的。因为前序遍历序列线索化的二叉树的第一个节点就是根节点，而且可以通过 GetNextPoint_PO 成员函数不断找到后继节点。参考如下代码。

 复制代码


```

1  //前序遍历序列线索化的二叉树查找某个节点(假设二叉树的节点各不相同)
2  BinaryTreeNode<T>* SearchElem_PO(const T& e)
3  {
4      return SearchElem_PO(root, e);
5  }
6  BinaryTreeNode<T>* SearchElem_PO(BinaryTreeNode<T>* tNode, const T& e)
7  {
8      if (tNode == nullptr)
9          return nullptr;
10
11     BinaryTreeNode<T>* tmpNode = root; //根就是第一个节点
12     while (tmpNode != nullptr) //从第一个节点开始一直找后继即可
13     {
14         if (tmpNode->data == e)
15             return tmpNode;
16         tmpNode = GetNextPoint_PO(tmpNode);
17     }

```

```
18     return nullptr;
19 }
```

后序遍历序列线索化的二叉树也是可以的，虽然这种二叉树找后继节点不行，但是找前趋节点是可以的，通过 `GetPriorPoint_POSTO` 成员函数，而且，最后一个节点就是根节点。从最后一个节点向前找即可。参考下面的代码。

 复制代码

```
1 //后序遍历序列线索化的二叉树查找某个节点(假设二叉树的节点各不相同)
2 BinaryTreeNode<T>* SearchElem_POSTO(const T& e)
3 {
4     return SearchElem_POSTO(root, e);
5 }
6 BinaryTreeNode<T>* SearchElem_POSTO(BinaryTreeNode<T>* tNode, const T& e)
7 {
8     if (tNode == nullptr)
9         return nullptr;
10
11     BinaryTreeNode<T>* tmpNode = root; //根就是最后一个节点
12     while (tmpNode != nullptr) //从最后一个节点开始一直找前趋即可
13     {
14         if (tmpNode->data == e)
15             return tmpNode;
16         tmpNode = GetPriorPoint_POSTO(tmpNode);
17     }
18     return nullptr;
19 }
```

小结

这节课，我讲解了线索二叉树的相关内容，包括线索二叉树的基本概念、二叉树的线索化、线索二叉树的操作 3 个子话题。

线索二叉树存在的意义，就是充分利用线索，尽量简化二叉树的操作，更方便地找一个节点的前趋和后继节点，从而为更快捷地遍历整个二叉树创造条件。

线索二叉树，其实就是加上了线索的二叉树，而线索，也就是指向前趋或后继节点的指针。

我们可以通过给没有被使用到的二叉树节点的指针域，保存这个节点的前趋或后继节点的指针信息，实现二叉树的线索化。

在线索化二叉树的过程中，首先要注意的是必须是空闲的指针域才能被线索化，在线索化的过程中，不但要标记该指针域是被线索化过的，还要确保该指针域正确地指向其前趋或后继节点。

最后提醒一下，这节课我们所有实现的代码都以能够理解为主，无需死记硬背。

归纳思考

1. 前面通过代码已经实现了用**中序**遍历序列线索化二叉树和用**前序**遍历序列线索化二叉树。你是否可以仿照前面的代码自己试着写一下如何用**后序**遍历序列线索化二叉树呢？
2. 参考前面的代码，尝试写出代码完成对**后序**遍历序列线索化的二叉树找前趋和后继节点。
3. 参考前面的代码，对“前序遍历序列线索化的二叉树”进行前序遍历，对“后序遍历序列线索化的二叉树”进行后序遍历，看是否能做到。

这里给予一些提示：

对“前序遍历序列线索化的二叉树”，因为找当前节点的后继节点是没问题的，所以对这种二叉树进行前序遍历是可以的。但因为找当前节点的前趋节点是无法做到的，因此进行前序逆向遍历是不行的。

对“后序遍历序列线索化的二叉树”，因为找当前节点的前趋节点是没问题的，所以对这种二叉树进行逆向后序遍历是可以的。但因为找当前节点的后继节点是无法做到的，因此进行后序遍历是不行的。

shikey.com转载分享

欢迎留言与我分享你的问题，我会尽我所能尽快反馈。如果你觉得有所收获，也欢迎你分享给同事或者朋友，我们一起进步。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

shikey.com转载分享