第6章 树

树结构（树形结构）是一种日常生活中应用相当广泛的非线性结构，包括企业内的组织结构、家族的族谱、篮球赛程等。另外，在计算机领域中的操作系统与数据库管理系统都是树结构，如 Windows、Unix 操作系统和文件系统，均是一种树结构的应用。如图 6-1 所示就是 Windows 的文件资源管理器，它是以树结构来存储各种文件的。

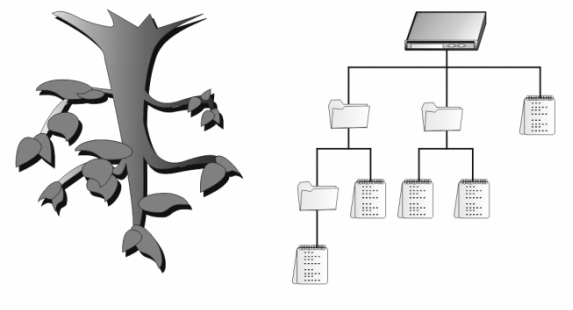


图 6-1

例如，在年轻人喜爱的大型网络游戏中，需要获取某些物体所在的地形信息，如果程序是依次从构成地形的模型三角面寻找，往往就会耗费许多运行时间，非常低效。因此，程序员一般会使用树结构中的二叉空间分割树（BSP tree）、四叉树（Quadtree）、八叉树（Octree）等来代表分割场景的数据，如图 6-2 和图 6-3 所示。

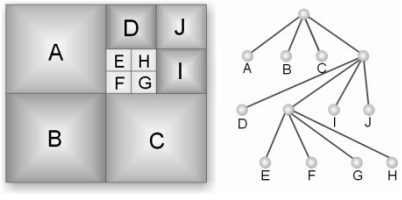


图 6-2

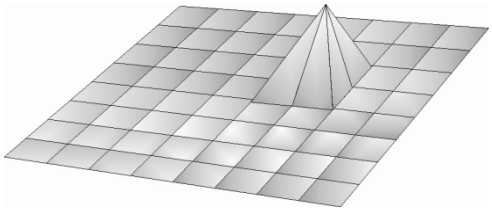


图 6-3

6.1 树的基本概念

“树”（Tree）是由一个或一个以上的节点（Node）组成，存在一个特殊的节点，称为树根（Root）。每个节点是一些数据和指针组合而成的记录。除了树根，其余节点可分为 n≥0 个互斥的集合，即 T_1，T_2，T_3…T_n，其中每一个子集合本身也是一种树形结构，即此根节点的子树。树形结构的示意图如图 6-4 所示，A 为根节点，B、C、D、E 均为 A 的子节点。

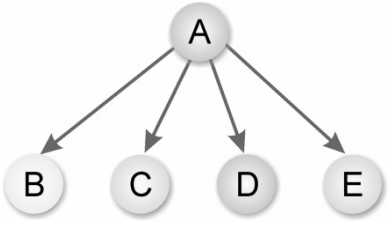


图 6-4

一棵合法的树，节点间虽可以互相连接，但不能形成无出口的回路。如图 6-5 就是一棵不合法的树。

树还可组成森林（Forest），也就是说森林是由 n 个互斥树的集合(n≥0)，移去树根即为森林。如图 6-6 所示就是包含了三棵树的森林。

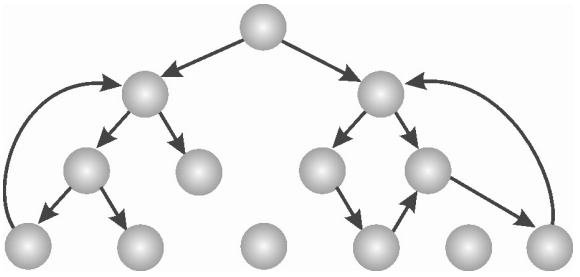


图 6-5

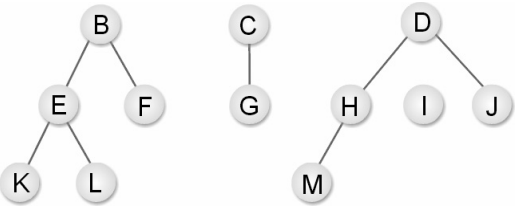


图 6-6

在树结构中，有许多常用的专有名词，本小节将以图 6-7 中这棵合法的树来为大家详细介绍。

- 度数（Degree）：每个节点所有子树的个数。例如图 6-7 中节点 B 的度数为 2，D 的度数为 3，F、K、I、J 等的度数为 0。
- 层数（Level）：树的层数，假设树根 A 为第一层，那么 B、C、D 节点的层数为 2，E、F、G、H、I、J 的层数为 3。
- 高度（Height）：树的最大层数。
- 树叶或称终端节点（Terminal Node）：度数为零的节点就是树叶，如图 6-7 中的 K、L、F、G、M、I、J 就是树叶，图 6-8 则有 4 个树叶节点，如 E，C，H，I。

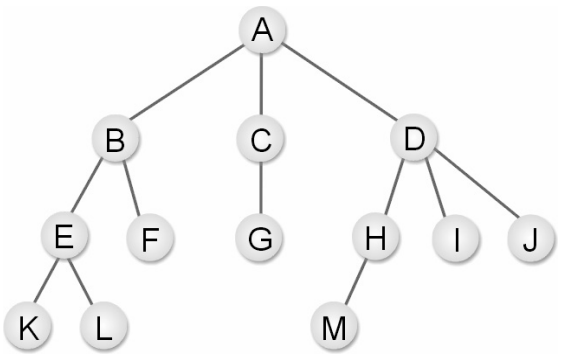


图 6-7

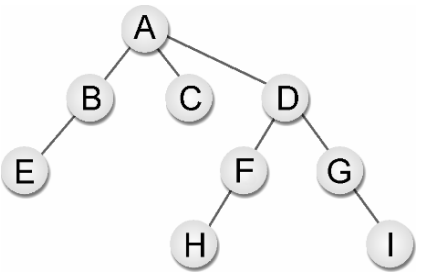


图 6-8

- 父节点（Parent）：每一个节点有连接的上一层节点（即为父节点），如图 6-7 所示，F 的父节点为 B，而 B 的父节点为 A。通常我们在绘制树形图时，会将父节点画在子节点的上方。
- 子节点（Children）：每一个节点有连接的下一层节点为子节点，还是看图 6-7，A 的子节点为 B、C、D，而 B 的子节点为 E 、F。
- 祖先（Ancestor）和子孙（Descendent）：所谓祖先，是指从树根到该节点路径上所包含的节点，而子孙则是在该节点往下追溯子树中的任一节点。在图 6-7 中，K 的祖先为 A、B、E 节点，H的祖先为 A、D 节点，节点 B 的子孙为 E、F、K、L。
- 兄弟节点（Sibling）：有共同父节点的节点为兄弟节点，在图 6-7 中，B、C、D 为兄弟节点，H、I、J 也为兄弟节点。

- 非终端节点（Nonterminal Node）：树叶以外的节点，如图 6-7 中的 A、B、C、D、E、H 等。
- 同代（Generation）：在同一棵树中具有相同层数的节点，如图 6-9 中的 E、F、G、H、I、J，或是 B、C、D。

范例 6.1.1　在图 6-10 中树（tree）有几个树叶节点（leaf node）？
（A）4　（B）5　（C）9　（D）11

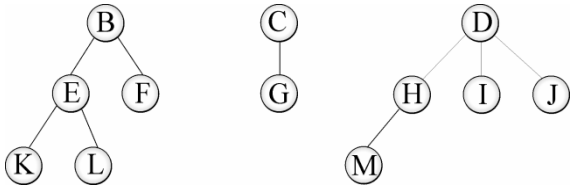


图 6-9

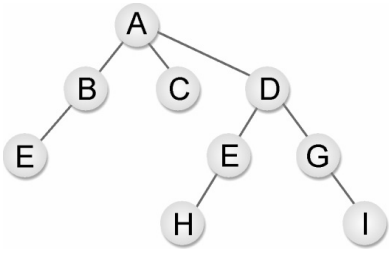


图 6-10

解答 度数为零的节点称为树叶节点，从图 6-10 中可看出答案为（A），即共有 E、C、H、I4 个树叶节点。

6.2 二叉树简介

一般树形结构在计算机内存中的存储方式是以链表（Linked List）为主。对于 n 叉树（n-way 树）来说，因为每个节点的度数都不相同，所以我们必须为每个节点都预留存放 n 个链接字段的最大存储空间。每个节点的数据结构如下：

请大家特别注意，这种 n 叉树十分浪费链接存储空间。假设此 n 叉树有 m 个节点，那么此树共有 n*m 个链接字段。另外，因为除了树根外，每一个非空链接都指向一个节点，所以得知空链接个数为 n*m－(m-1) = m*(n−1) + 1，而 n 叉树的链接浪费率为 $\frac{m*(n-1)+1}{m*n}$。因此，我们可以得出以下结论：

- n=2 时，2 叉树的链接浪费率约为 1/2；
- n=3 时，3 叉树的链接浪费率约为 2/3；
- n=4 时，4 叉树的链接浪费率约为 3/4；

……

因为当 n = 2 时，它的链接浪费率最低，所以为了改进存储空间浪费的缺点，我们经常使用二叉树（Binary Tree）结构来取代其他树形结构。

6.2.1　二叉树的定义

二叉树（又称为 Knuth 树）是一个由有限节点所组成的集合，此集合可以为空集合，或者

由一个树根及其左右两个子树所组成。简单地说，二叉树最多只能有两个子节点，就是度数小于或等于 2。其计算机中的数据结构如下：

二叉树和一般树的不同之处整理如下：

（1）树不可为空集合，但是二叉树可以。
（2）树的度数为 d≥0，但二叉树的节点度数为 0≤d≤2。
（3）树的子树间没有次序关系，二叉树则有。

下面我们来看一棵实际的二叉树，如图 6-11 所示。

图 6-11 是以 A 为根节点的二叉树，且包含了以 B、D 为根节点的两棵互斥的左子树和右子树，如图 6-12 所示。

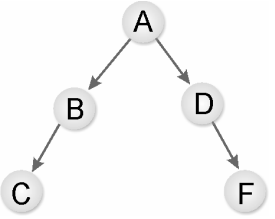


图 6-11

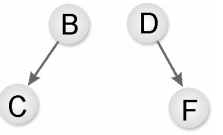


图 6-12

以上这两个左右子树都属于同一种树形结构，不过却是两棵不同的二叉树结构，原因就是二叉树必须考虑前后次序的关系，这点大家要特别注意。

范例 6.2.1　试证明高度为 k 的二叉树的总节点数是 2^k-1。

解答 其节点总数为第 1 层到第 k 层中各层中最大节点的总和。即

$$\sum_{i=1}^{k} 2^{i-1} = 20+21+\ldots\ldots+2k-1 = \frac{2^k-1}{2-1} = 2^k-1$$

范例 6.2.2　对于任何非空二叉树 T，如果 n_0 为树叶节点数，且度数为 2 的节点数是 n_2，试证明 $n_0 = n_2+1$。

解答 可先行假设 n 是节点总数，n_1 是度数等于 1 的节点数，可得 $n = n_0+n_1+n_2$，再进行证明。

范例 6.2.3　在二叉树中，层数（Level）为 i 的节点数最多是 2^{i-1}(i≥0)，试证明之。

解答 我们可利用数学归纳法证明：

（1）当 i = 1 时，因为只有树根一个节点，所以 $2^{i-1} = 2^0 = 1$ 成立。
（2）假设对于 j，且 1 ≤ j ≤ i，层数为 j 的最多节点数为 2^{j-1} 个成立，则在 j = i 层上的节点最多为 2^{i-1} 个。

当 j = i + 1 时，因为二叉树中每一个节点的度数都不大于 2，所以在层数 j = i + 1 时的最多节点数 ≤ $2*2^{i-1} = 2^i$，由此得证。

6.2.2　特殊二叉树简介

由于二叉树的应用相当广泛，因此衍生了许多特殊的二叉树结构。

- **满二叉树（Fully Binary Tree）**

如果二叉树的高度为 h，树的节点数为 2^h-1，h≥0，则我们称此树为“满二叉树”（Full Binary Tree），如图 6-13 所示。

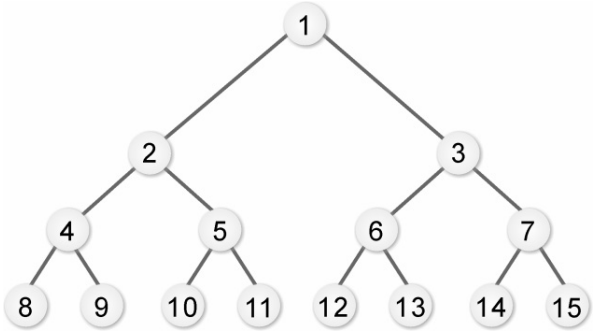


图 6-13

- **完全二叉树（Complete Binary Tree）**

如果二叉树的高度为 h，所含的节点数小于 2^h-1，那么其节点的编号方式如同高度为 h 的满二叉树一样，从左到右，从上到下的顺序一一对应，如图 6-14 所示。

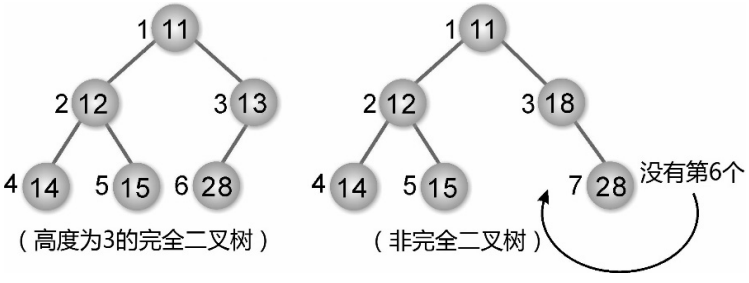


图 6-14

对于完全二叉树而言，假设有 N 个节点，那么此二叉树的层数 h 为 $\lfloor \log_2(N+1) \rfloor$。

- **斜二叉树（Skewed Binary Tree）**

当一个二叉树完全没有右节点或左节点时，我们就把它称为左斜二叉树或右斜二叉树，如图 6-15 所示。

- **严格二叉树（Strictly Binary Tree）**

二叉树中的每一个非终端节点均有非空的左右子树，如图 6-16 所示。

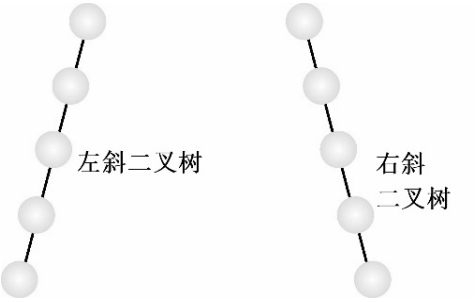


图 6-15

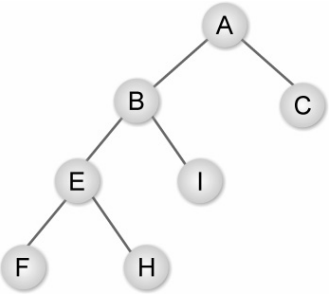


图 6-16

范例▶ 6.2.4　假如有一个非空树，其度为 5，已知度为 i 的节点数有 i 个，其中 1≤i≤5，请问终端节点数总数是多少？

解答▶ 41 个。

6.3 二叉树存储方式

二叉树的存储方式有很多，在数据结构中，我们习惯用链表来表示二叉树，这样在删除或增加节点时，会非常方便且具有弹性。当然，也可以使用一维数组这样的连续存储空间来表示二叉树，不过在对树中的中间节点进行插入与删除操作时，可能要大量移动数组中节点的存储位置来反应树节点的变动。下面我们将分别介绍数组和链表这两种存储方法。

6.3.1 一维数组表示法

使用有序的一维数组来表示二叉树，可将此二叉树假想成一棵满二叉树（Full Binary Tree），而且第 k 层具有 2^{k-1} 个节点，它们按序存放在这个一维数组中。首先来看看使用一维数组建立二叉树的表示方法及数组索引值的设置，如表 6-1 所示。

表 6-1　使用一维数组建立二叉树的表示方法及数组索引值的设置

索引值	1	2	3	4	5	6	7
内容值	A	B			C		D

从图 6-17 中，我们可以看到此一维数组中的索引值有以下关系。

（1）左子树索引值是父节点索引值*2。

（2）右子树索引值是父节点索引值*2+1。

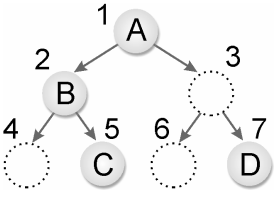


图 6-17

接着来看如何以一维数组建立二叉树，事实上就是建立一个二叉查找树。这是一种很好的排序应用模式，因为在建立二叉树的同时，数据就经过初步的比较与判断，并按照二叉树的建立规则来存放数据。二叉查找树具有以下特点。

（1）可以是空集合，但若不是空集合，则节点上一定要有一个键值。

（2）每一个树根的值须大于左子树的值。
（3）每一个树根的值须小于右子树的值。
（4）左右子树也是二叉查找树。
（5）树的每个节点值都不相同。

现在我们示范用一组数据 32、25、16、35、27，来建立一棵二叉查找树，具体过程如图 6-18 所示。

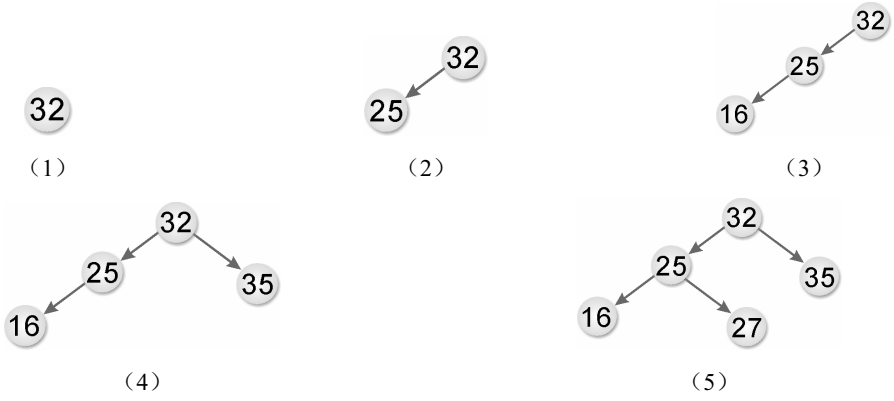


图 6-18

在下面的程序中，我们先建立一个一维数组，并将数组中的值按照上述规则建立一个满二叉树。

范例程序：ch06_01.sln

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using static System.Console;//导入静态类

namespace ch06_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, level;
            int[] data = { 6, 3, 5, 9, 7, 8, 4, 2 }; /*原始数组*/
            int[] btree = new int[16];
            for (i = 0; i < 16; i++) btree[i] = 0;
```

```
            Write("原始数组的内容：\n");
            for (i = 0; i < 8; i++)
                Write("[" + data[i] + "] ");
            WriteLine();
            for (i = 0; i < 8; i++)     /*把原始数组中的值逐一对比*/
            {
                for (level = 1; btree[level] != 0;) /*比较树根和数组内的值*/
                {
                    if (data[i] > btree[level]) /*如果数组内的值大于树根，
                                                  则往右子树比较*/
                        level = level * 2 + 1;
                    else     /*如果数组内的值小于或等于树根，则往左子树比较*/
                        level = level * 2;
                }           /*如果子树节点的值不为 0，则再与数组内的值比较一次*/
                btree[level] = data[i];  /*把数组值放入二叉树*/
            }
            Write("二叉树的内容：\n");
            for (i = 1; i < 16; i++)
                Write("[" + btree[i] + "] ");
            WriteLine();
            ReadKey();
        }
    }
}
```

范例程序的执行结果如图 6-19 所示。

```
原始数组的内容：
[6] [3] [5] [9] [7] [8] [4] [2]
二元树的内容：
[6] [3] [9] [2] [5] [7] [0] [0] [0] [4] [0] [0] [8] [0] [0]
```

图 6-19

通常以数组表示法来存储二叉树，如果越接近满二叉树，则越节省空间；如果是歪斜树，则最浪费空间。另外，要增删数据比较麻烦，必须重新建立二叉树。

图 6-20 是此数组值在二叉树中存放的情形。

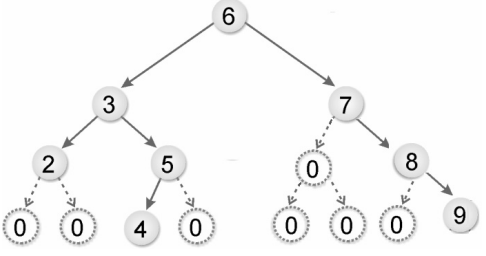


图 6-20

6.3.2　链表表示法

所谓二叉树的链表表示法，就是利用链表来存储二叉树。例如，在 C# 语言中，我们可以定义 TreeNode 类和 BinaryTree 类，其中 TreeNode 就代表二叉树中的一个节点。定义如下：

```
class TreeNode
{
    public int value;
    public TreeNode left_Node;
    public TreeNode right_Node;
    // TreeNode 构造函数
    public TreeNode(int value)
    {
        this.value = value;
        this.left_Node = null;
        this.right_Node = null;
    }
}
```

范例程序：ch06_02.sln

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6  using System.IO;
 7  using static System.Console;//导入静态类
 8
 9  namespace ch06_02
10  {
11      class TreeNode
12      {
13          public int value;
14          public TreeNode left_Node;
15          public TreeNode right_Node;
16          // TreeNode 构造函数
17          public TreeNode(int value)
18          {
19              this.value = value;
20              this.left_Node = null;
21              this.right_Node = null;
22          }
```

```
23      }
24      //二叉树类声明
25      class BinaryTree
26      {
27          public TreeNode rootNode; //二叉树的根节点
28                                    //构造函数:利用传入一个数组的参数来建立二叉树
29          public BinaryTree(int[] data)
30          {
31              for (int i = 0; i < data.Length; i++)
32                  Add_Node_To_Tree(data[i]);
33          }
34          //将指定的值加入到二叉树中适当的节点
35          void Add_Node_To_Tree(int value)
36          {
37              TreeNode currentNode = rootNode;
38              if (rootNode == null)
39              { //建立树根
40                  rootNode = new TreeNode(value);
41                  return;
42              }
43              //建立二叉树
44              while (true)
45              {
46                  if (value < currentNode.value)
47                  { //在左子树
48                      if (currentNode.left_Node == null)
49                      {
50                          currentNode.left_Node = new TreeNode(value);
51                          return;
52                      }
53                      else currentNode = currentNode.left_Node;
54                  }
55                  else
56                  { //在右子树
57                      if (currentNode.right_Node == null)
58                      {
59                          currentNode.right_Node = new TreeNode(value);
60                          return;
61                      }
62                      else currentNode = currentNode.right_Node;
63                  }
64              }
65          }
```

```
66      }
67      class Program
68      {
69          static void Main(string[] args)
70          {
71              int ArraySize = 10;
72              int tempdata;
73              int[] content = new int[ArraySize];
74              WriteLine("请连续输入" + ArraySize + "个数据");
75              for (int i = 0; i < ArraySize; i++)
76              {
77                  Write("请输入第" + (i + 1) + "个数据：");
78                  tempdata = int.Parse(ReadLine());
79                  content[i] = tempdata;
80              }
81              new BinaryTree(content);
82              WriteLine("===用链表方式建立二叉树，成功!!!===");
83              ReadKey();
84          }
85      }
86  }
```

范例程序的执行结果如图 6-21 所示。

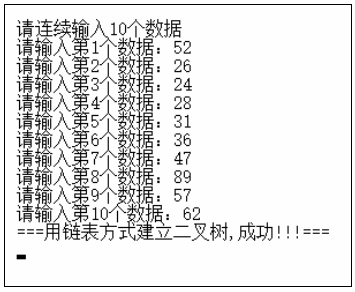


图 6-21

使用链表来表示二叉树的好处是对节点的增加与删除相当容易，缺点是很难找到父节点，除非在每一节点多增加一个父字段。

6.4 二叉树遍历

我们知道线性数组或链表，都只能单向从头至尾遍历或反向遍历。所谓二叉树的遍历（Binary Tree Traversal），最简单的说法就是“访问树中所有的节点各一次”，并且在遍历后，将树中的数据转化为线性关系。以图 6-22 所示的一个简单的二叉树节点来说，每个节点都可分为左右两个分支。

所以可以有 ABC、ACB、BAC、BCA、CAB 和 CBA 等 6 种遍历方法。如果是按照二叉树特性，一律从左向右，就只剩下三种遍历方式，分别是 BAC、ABC、BCA。这三种方式的命名与规则如下：

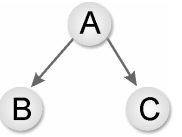


图 6-22

（1）中序遍历（BAC，Inorder）：左子树→树根→右子树。
（2）前序遍历（ABC，Preorder）：树根→左子树→右子树。
（3）后序遍历（BCA，Postorder）：左子树→右子树→树根。

对于这三种遍历方式，大家只需要记住树根的位置，就不会把前序、中序和后序搞混了。例如，中序法即树根在中间，前序法是树根在前面，后序法则是树根在后面。而遍历方式也一定是先左子树，后右子树。下面针对这三种方式，做更加详尽的介绍。

6.4.1 中序遍历

中序遍历（Inorder Traversal）是“左中右”的遍历顺序，也就是从树的左侧逐步向下方移动，直到无法移动，再访问此节点，并向右移动一节点。如果无法再向右移动，则可以返回上层的父节点，并重复左、中、右的步骤进行。

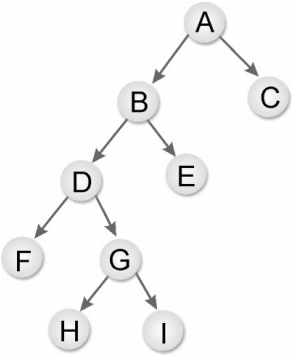


图 6-23

（1）遍历左子树。
（2）遍历（或访问）树根。
（3）遍历右子树。

如图 6-23 所示的遍历为 FDHGIBEAC。
中序遍历的 C# 算法如下：

```
public void InOrder(TreeNode node)
{
    if (node != null)
    {
        InOrder(node.left_Node);
        Write("[" + node.value + "] ");
        InOrder(node.right_Node);
    }
}
```

6.4.2 后序遍历

后序遍历（Postorder Traversal）是“左右中”的遍历顺序，即先遍历左子树，再遍历右子树，最后遍历（或访问）根节点，反复执行此步骤。

（1）遍历左子树。
（2）遍历右子树。
（3）遍历树根。

如图 6-24 所示的后序遍历为 FHIGDEBCA。

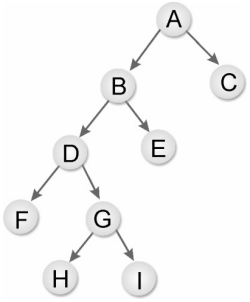


图 6-24

后序遍历的 C# 算法如下：

```
public void PostOrder(TreeNode node)
{
if (node != null)
 {
    PostOrder(node.left_Node);
    PostOrder(node.right_Node);
    Write("[" + node.value + "] ");
 }
}
```

6.4.3　前序遍历

前序遍历（Preorder Traversal）是“中左右”的遍历顺序，也就是先从根节点遍历，再往左方移动，当无法继续时，继续向右方移动，接着重复执行此步骤。

（1）遍历（或访问）树根。
（2）遍历左子树。
（3）遍历右子树。

如图 6-25 所示的前序遍历为 ABDFGHIEC。

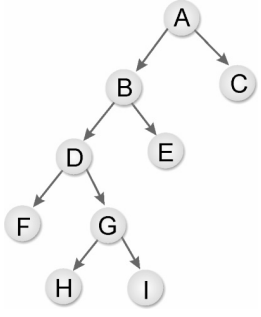


图 6-25

前序遍历的 C#算法如下：

```
public void PreOrder(TreeNode node)
{
    if (node != null)
    {
        Write("[" + node.value + "] ");
        PreOrder(node.left_Node);
        PreOrder(node.right_Node);
    }
}
```

范例 6.4.1　请问如图 6-26 所示的二叉树的中序、前序及后序表示法是什么？

解答 中序遍历为 DBEACF；
前序遍历为 ABDECF；
后序遍历为 DEBFCA。

范例 6.4.2　请问如图 6-27 所示的二叉树的中序、前序及后序遍历的结果是什么？

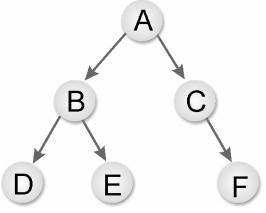


图 6-26

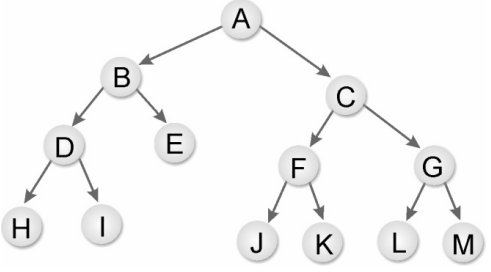


图 6-27

解答
（1）前序为 ABDHIECFJKGLM；
（2）中序为 HDIBEAJFKCLGM；
（3）后序为 HIDEBJKFLMGCA。

6.4.4　二叉树遍历的实现

接着我们开始建立二叉树，并实现中序、前序与后序遍历。在程序中会预先指定二叉树的内容，并在遍历二叉树后把树的前、中、后序打印出来，让读者比较三种遍历方式的不同之处。

【范例程序：ch06_03.sln】

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
```

```
using System.Threading.Tasks;
using System.IO;
using static System.Console;//导入静态类

namespace ch06_03
{
    class TreeNode
    {
        public int value;
        public TreeNode left_Node;
        public TreeNode right_Node;

        public TreeNode(int value)
        {
            this.value = value;
            this.left_Node = null;
            this.right_Node = null;
        }
    }
    class BinaryTree
    {
        public TreeNode rootNode;

        public void Add_Node_To_Tree(int value)
        {
            if (rootNode == null)
            {
                rootNode = new TreeNode(value);
                return;
            }
            TreeNode currentNode = rootNode;
            while (true)
            {
                if (value < currentNode.value)
                {
                    if (currentNode.left_Node == null)
                    {
                        currentNode.left_Node = new TreeNode(value);
                        return;
                    }
                    else
                        currentNode = currentNode.left_Node;
                }
```

```
48                else
49                {
50                    if (currentNode.right_Node == null)
51                    {
52                        currentNode.right_Node = new TreeNode(value);
53                        return;
54                    }
55                    else
56                        currentNode = currentNode.right_Node;
57                }
58            }
59        }
60        public void InOrder(TreeNode node)
61        {
62            if (node != null)
63            {
64                InOrder(node.left_Node);
65                Write("[" + node.value + "] ");
66                InOrder(node.right_Node);
67            }
68        }
69
70        public void PreOrder(TreeNode node)
71        {
72            if (node != null)
73            {
74                Write("[" + node.value + "] ");
75                PreOrder(node.left_Node);
76                PreOrder(node.right_Node);
77            }
78        }
79
80        public void PostOrder(TreeNode node)
81        {
82            if (node != null)
83            {
84                PostOrder(node.left_Node);
85                PostOrder(node.right_Node);
86                Write("[" + node.value + "] ");
87            }
88        }
89    }
90    class Program
```

```
 91      {
 92          static void Main(string[] args)
 93          {
 94              int i;
 95              int[] arr = { 7, 4, 1, 5, 16, 8, 11, 12, 15, 9, 2 };
                                                              /*原始的数组*/
 96              BinaryTree tree = new BinaryTree();
 97              Write("原始数组的内容: \n");
 98              for (i = 0; i < 11; i++)
 99                  Write("[" + arr[i] + "] ");
100              WriteLine();
101              for (i = 0; i < arr.Length; i++) tree.Add_Node_To_Tree(arr[i]);
102              Write("[二叉树的内容]\n");
103              Write("前序遍历的结果: \n");  /*打印前、中、后序遍历的结果*/
104              tree.PreOrder(tree.rootNode);
105              WriteLine();
106              Write("中序遍历的结果: \n");
107              tree.InOrder(tree.rootNode);
108              WriteLine();
109              Write("后序遍历的结果: \n");
110              tree.PostOrder(tree.rootNode);
111              ReadKey();
112          }
113      }
114  }
```

范例程序的执行结果如图 6-28 所示。

此程序所建立的二叉树结构如图 6-29 所示。

```
原始数组的内容:
[7] [4] [1] [5] [16] [8] [11] [12] [15] [9] [2]
[二叉树的内容]
前序遍历的结果:
[7] [4] [1] [2] [5] [16] [8] [11] [9] [12] [15]
中序遍历的结果:
[1] [2] [4] [5] [7] [8] [9] [11] [12] [15] [16]
后序遍历的结果:
[2] [1] [5] [4] [9] [15] [12] [11] [8] [16] [7]
```

图 6-28

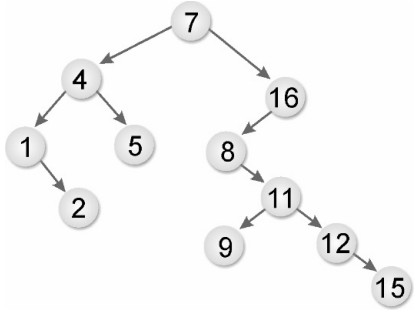


图 6-29

范例 6.4.3　请利用后序遍历将如图 6-30 所示的二叉树的遍历结果按节点中的文字打印出来。

解答　把握左子树→右子树→树根的原则，可得 DBHEGIFCA。

范例 6.4.4　请问如图 6-31 所示的二叉树的中序、前序及后序表示法是什么？

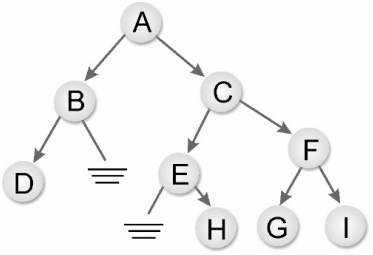


图 6-30

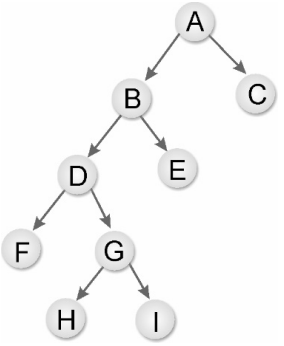


图 6-31

解答

（1）中序为 FDHGIBEAC；

（2）后序为 FHIGDEBCA；

（3）前序为 ABDFGHIEC。

范例 6.4.5　一棵二叉树被表示为 A(B(CD)E(F(G)H(I(JK)L(MNO)))，请画出二叉树的结构及后序与前序遍历的结果（图 6-32）。

解答

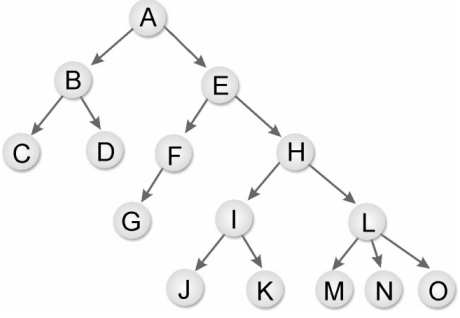


图 6-32

解答　后序遍历为 CDBGFJKIMNOLHEA；

前序遍历为 ABCDEFGHIJKLMNO。

6.4.5　二叉运算树

一般的算术式也可以转换成二叉运算树（Binary Expression Tree）的方式，如图 6-33 所示。建立的方法可遵循以下两种规则：

（1）考虑算术式中运算符的结合性与优先权，再适当地加上括号。

（2）由最内层的括号逐步向外，利用运算符当树根，左边操作数当左子树，右边操作数当右子树，其中优先权最低的运算符作为此二叉运算树的树根。

现在我们尝试将 A-B*(-C+-3.5)表达式转为二叉运算树，并求出此算术式的前序（Prefix）与后序（Postfix）表示法。

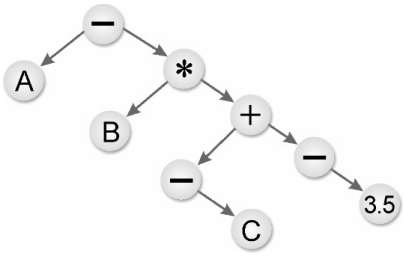


图 6-33

→A-B*(-C+-3.5)
→(A-(B*((-C)+(-3.5))))
→

接着将二叉运算树进行前序与后序遍历，即可得出此算术式的前序法与后序法。

前序表示法为-A*B+-C-3.5。
后序表示法为 ABC-3.5-+*-。

范例 6.4.6　请将 A/B**C+D*E-A*C 转化为二叉运算树。

解答 加括号成为→(((A/B**C))+(D*E))-(A*C))，如图 6-34 所示。

范例 6.4.7　请问如图 6-35 所示的二叉运算树的中序、后序与前序的表示法是什么？

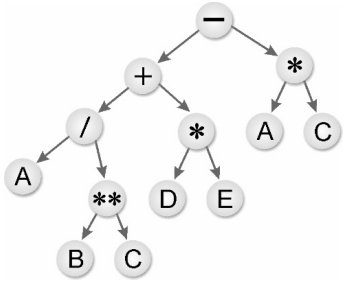


图 6-34

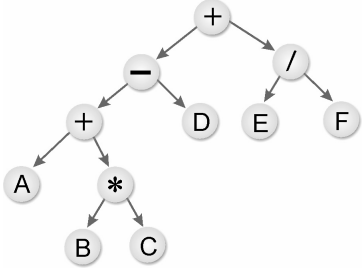


图 6-35

解答
（1）中序为 A+B*C-D+E/F；
（2）前序为+-+A*BCD/EF；
（3）后序为 ABC*+D-EF/+。

范例程序：ch06_04.sln

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.IO;
7   using static System.Console;//导入静态类
8
9   namespace ch06_04
```

```
{
    // 用链表实现二叉运算树

    // 节点类的声明
    class TreeNode
    {
        public int value;
        public TreeNode left_Node;
        public TreeNode right_Node;
        // TreeNode 构造函数
        public TreeNode(int value)
        {
            this.value = value;
            this.left_Node = null;
            this.right_Node = null;
        }
    }
    //二叉查找树类声明
    class Binary_Search_Tree
    {
        public TreeNode rootNode; //二叉树的根节点
        //构造函数：建立空的二叉查找树
        public Binary_Search_Tree() { rootNode = null; }
        //构造函数:利用传入一个数组的参数来建立二叉树
        public Binary_Search_Tree(int[] data)
        {
            for (int i = 0; i < data.Length; i++)
                Add_Node_To_Tree(data[i]);
        }
        //将指定的值加入到二叉树中适当的节点
        void Add_Node_To_Tree(int value)
        {
            TreeNode currentNode = rootNode;
            if (rootNode == null)
            { //建立树根
                rootNode = new TreeNode(value);
                return;
            }
            //建立二叉树
            while (true)
            {
                if (value < currentNode.value)
                { //符合这个判断表示此节点在左子树
```

```
53                  if (currentNode.left_Node == null)
54                  {
55                      currentNode.left_Node = new TreeNode(value);
56                      return;
57                  }
58                  else currentNode = currentNode.left_Node;
59              }
60              else
61              { //符合这个判断表示此节点在右子树
62                  if (currentNode.right_Node == null)
63                  {
64                      currentNode.right_Node = new TreeNode(value);
65                      return;
66                  }
67                  else currentNode = currentNode.right_Node;
68              }
69          }
70      }
71   }
72
73   class Expression_Tree:Binary_Search_Tree
74   {
75  // 构造函数
76  public Expression_Tree(char[] information, int index)
77   {
78      // Create 方法可以将二叉树的数组表示法转换成链表表示法
79      rootNode = Create(information, index);
80   }
81   // Create 方法的程序内容
82   public TreeNode Create(char[] sequence, int index)
83   {
84      TreeNode tempNode;
85      if (index >= sequence.Length)   // 作为递归调用的出口条件
86          return null;
87      else
88      {
89          tempNode = new TreeNode((int)sequence[index]);
90          // 建立左子树
91          tempNode.left_Node = Create(sequence, 2 * index);
92          // 建立右子树
93          tempNode.right_Node = Create(sequence, 2 * index + 1);
94          return tempNode;
95      }
```

```
 96      }
 97      // PreOrder(前序遍历)方法的程序内容
 98      public void PreOrder(TreeNode node)
 99      {
100          if (node != null)
101          {
102             Write((char)node.value);
103              PreOrder(node.left_Node);
104              PreOrder(node.right_Node);
105          }
106      }
107      // InOrder(中序遍历)方法的程序内容
108      public void InOrder(TreeNode node)
109      {
110          if (node != null)
111          {
112              InOrder(node.left_Node);
113                 Write((char)node.value);
114              InOrder(node.right_Node);
115          }
116      }
117      // PostOrder(后序遍历)方法的程序内容
118      public void PostOrder(TreeNode node)
119      {
120          if (node != null)
121          {
122              PostOrder(node.left_Node);
123              PostOrder(node.right_Node);
124                 Write((char)node.value);
125          }
126      }
127      // 判断表达式如何运算的方法
128      public int Condition(char oprator, int num1, int num2)
129      {
130          switch (oprator)
131          {
132              case '*': return (num1 * num2); // 乘法请返回 num1 * num2
133              case '/': return (num1 / num2); // 除法请返回 num1 / num2
134              case '+': return (num1 + num2); // 加法请返回 num1 + num2
135              case '-': return (num1 - num2); // 减法请返回 num1 - num2
136              case '%': return (num1 % num2); // 取余数法请返回 num1 % num2
137          }
138          return -1;
```

```
139      }
140      // 传入根节点，用来计算此二叉运算树的值
141      public int Answer(TreeNode node)
142      {
143          int firstnumber = 0;
144          int secondnumber = 0;
145              // 递归调用的出口条件
146              if (node.left_Node == null && node.right_Node == null)
147                  // 将节点的值转换成数值后返回
148                  return Convert.ToInt32((char)node.value)-48;
149              else
150              {
151                  firstnumber = Answer(node.left_Node);// 计算左子树表达式的值
152                  secondnumber = Answer(node.right_Node);//计算右子树表达式的值
153                  Return Condition((char)node.value, firstnumber,
                                     secondnumber);
154              }
155      }
156  }
157  class Program
158      {
159          static void Main(string[] args)
160          {
161              // 将二叉运算树以数组的方式来声明
162              // 第一个表达式
163              char[] information1 = { ' ', '+', '*', '%', '6', '3', '9', '5' };
164              // 第二个表达式
165              char[] information2 = {' ','+','+','+','*','%','/','*',
166                                       '1','2','3','2','6','3','2','2' };
167              Expression_Tree exp1 = new Expression_Tree(information1, 1);
168              WriteLine("====二叉运算树数值运算范例 1: ====");
169              WriteLine("================================");
170              Write("===转换成中序表达式===: ");
171              exp1.InOrder(exp1.rootNode);
172              Write("\n===转换成前序表达式===: ");
173              exp1.PreOrder(exp1.rootNode);
174              Write("\n===转换成后序表达式===: ");
175              exp1.PostOrder(exp1.rootNode);
176              // 计算二叉树表达式的运算结果
177              Write("\n 此二叉运算树，经过计算后所得到的结果值为: ");
178              WriteLine(exp1.Answer(exp1.rootNode));
179              // 建立第二棵二叉查找树对象
180              Expression_Tree exp2 = new Expression_Tree(information2, 1);
```

```
181             WriteLine();
182             WriteLine("====二叉运算树数值运算范例 2: ====");
183             WriteLine("==================================");
184             Write("===转换成中序表达式===: ");
185             exp2.InOrder(exp2.rootNode);
186             Write("\n===转换成前序表达式===: ");
187             exp2.PreOrder(exp2.rootNode);
188             Write("\n===转换成后序表达式===: ");
189             exp2.PostOrder(exp2.rootNode);
190             // 计算二叉树表达式的运算结果
191             Write("\n 此二叉运算树，经过计算后所得到的结果值为: ");
192             WriteLine(exp2.Answer(exp2.rootNode));
193             ReadKey();
194         }
195     }
196 }
```

范例程序的执行结果如图 6-36 所示。

```
====二叉运算树数值运算范例 1: ====
==================================
===转换成中序表达式===: 6*3+9%5
===转换成前序表达式===: +*63%95
===转换成后序表达式===: 63*95%+
此二叉运算树，经过计算后所得到的结果值为: 22

====二叉运算树数值运算范例 2: ====
==================================
===转换成中序表达式===: 1*2+3%2+6/3+2*2
===转换成前序表达式===: +++12%32+/63*22
===转换成后序表达式===: 12*32%+63/22*++
此二叉运算树，经过计算后所得到的结果值为: 9
```

图 6-36

6.5 二叉树的高级研究

除了之前所介绍的二叉树遍历方式外，二叉树还有许多常见的应用，如二叉排序树、二叉查找树（二叉搜索树）、线索二叉树等。

6.5.1 二叉排序树

事实上，二叉树是一种很好的排序应用模式，因为在建立二叉树的同时，数据已经经过初步的比较，并按照二叉树的建立规则来存放数据。其规则如下：

（1）第一个输入数据作为此二叉树的树根。

（2）之后的数据以递归的方式与树根进行比较，小于树根置于左子树，大于树根置于右子树。

从上面的规则我们可以知道，左子树内的值一定小于树根，而右子树的值一定大于树根。因此，只要利用“中序遍历”方式就可以得到从小到大排序好的数据，如果想从大到小排列，则可将最后结果置于堆栈内再依次弹出（POP）即可。

下面我们示范用一组数据 32、25、16、35、27，建立一棵二叉排序树（二叉查找树），如图 6-37 所示。

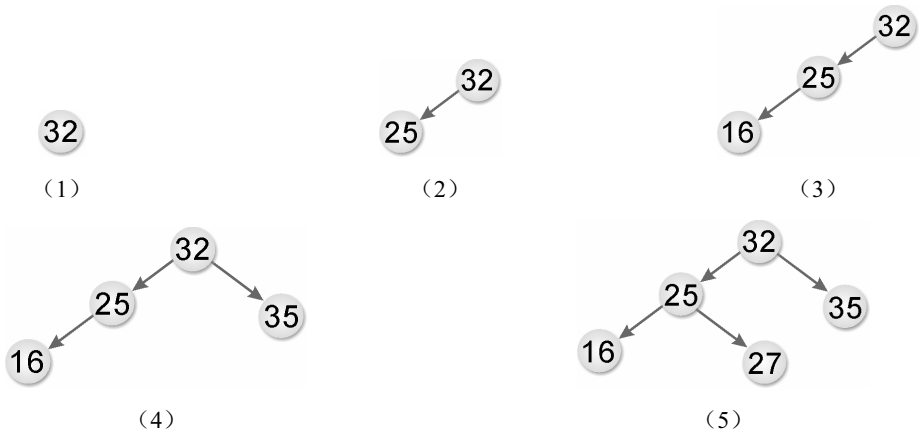


图 6-37

图 6-37 中的最后一个就是建立完成的二叉排序树，通过中序遍历后，可得出 16、25、27、32、35 从小到大的排列。因为在输入数据的同时就开始建立二叉树，所以在完成数据输入并建立二叉排序树后，通过中序遍历就可以轻松得到排序的结果，请看下面的 C# 程序范例。

范例程序：ch06_05.sln

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using static System.Console;//导入静态类

namespace ch06_05
{
    class TreeNode
    {
        public int value;
        public TreeNode left_Node;
        public TreeNode right_Node;
```

```
16
17          public TreeNode(int value)
18          {
19              this.value = value;
20              this.left_Node = null;
21              this.right_Node = null;
22          }
23      }
24      class BinaryTree
25      {
26          public TreeNode rootNode;
27
28          public void Add_Node_To_Tree(int value)
29          {
30              if (rootNode == null)
31              {
32                  rootNode = new TreeNode(value);
33                  return;
34              }
35              TreeNode currentNode = rootNode;
36              while (true)
37              {
38                  if (value < currentNode.value)
39                  {
40                      if (currentNode.left_Node == null)
41                      {
42                          currentNode.left_Node = new TreeNode(value);
43                          return;
44                      }
45                      else
46                          currentNode = currentNode.left_Node;
47                  }
48                  else
49                  {
50                      if (currentNode.right_Node == null)
51                      {
52                          currentNode.right_Node = new TreeNode(value);
53                          return;
54                      }
55                      else
56                          currentNode = currentNode.right_Node;
57                  }
58              }
```

```
 59          }
 60          public void InOrder(TreeNode node)
 61          {
 62              if (node != null)
 63              {
 64                  InOrder(node.left_Node);
 65                  Write("[" + node.value + "] ");
 66                  InOrder(node.right_Node);
 67              }
 68          }
 69
 70          public void PreOrder(TreeNode node)
 71          {
 72              if (node != null)
 73              {
 74                  Write("[" + node.value + "] ");
 75                  PreOrder(node.left_Node);
 76                  PreOrder(node.right_Node);
 77              }
 78          }
 79
 80          public void PostOrder(TreeNode node)
 81          {
 82              if (node != null)
 83              {
 84                  PostOrder(node.left_Node);
 85                  PostOrder(node.right_Node);
 86                  Write("[" + node.value + "] ");
 87              }
 88          }
 89      }
 90      class Program
 91      {
 92          static void Main(string[] args)
 93          {
 94              int value;
 95              BinaryTree tree = new BinaryTree();
 96              Write("请输入数据，要结束请输入-1：\n");
 97              while (true)
 98              {
 99                  value = int.Parse(ReadLine());
100                  if (value == -1)
101                      break;
```

```
102                 tree.Add_Node_To_Tree(value);
103             }
104             Write("====================: \n");
105             Write("排序完成的结果: \n");
106             tree.InOrder(tree.rootNode);
107             WriteLine();
108             ReadKey();
109         }
110     }
111 }
```

范例程序的执行结果如图 6-38 所示。

```
请输入数据，要结束请输入-1:
52
26
41
85
97
10
-1
====================:
排序完成的结果:
[10] [26] [41] [52] [85] [97]
```

图 6-38

范例 6.5.1　我们可利用二叉树按照中序方式进行排序，请大家依次回答空格部分。

（1）二叉树的每一节点（Node）至少包含三个字段，其中一个存数据，另外两个分别为____及____，分为_____及_____之用，设其使用密集表（Dense List）存放，则须另有一根指针（Root）指其开始根部。

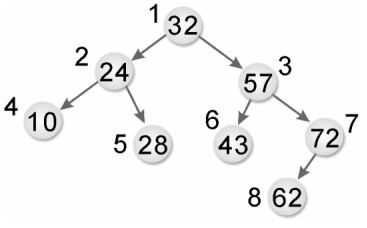
（2）试将 32、24、57、28、10、43、72、62 按照中序方式存入可放 10 个节点（Node）的 list 内，试画出其结果，画出方式为何？

（3）若插入数据为 30，试写出其相关操作与位置变化。

（4）若删除数据为 32，试写出其相关操作与位置变化。

解答

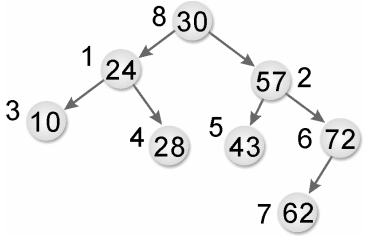
（1）左链接、右链接、指向左节点、指向右节点。



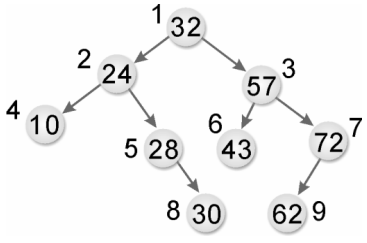
（2）

root=1	left	data	right
1	2	32	3
2	4	24	5
3	6	57	7
4	0	10	0
5	0	28	0
6	0	43	0
7	8	72	0
8	0	62	0
9			
10			

（3）



root=1	left	data	right
1	2	32	3
2	4	24	5
3	6	57	7
4	0	10	0
5	0	28	8
6	0	43	0
7	9	72	0
8	0	30	0
9	0	62	0
10			



（4）

root=1	left	data	right
1	3	24	4
2	5	57	6
3	0	10	0
4	0	28	0
5	0	43	0
6	7	72	0
7	0	62	0
8	1	30	2
9			
10			

6.5.2 二叉查找树

如果一个二叉树符合“每一个节点的数据大于左子节点且小于右子节点”，那么这棵树便称为二分树。因为二分树便于排序和搜索，所以二叉排序树或二叉查找树都是二分树的一种。当建立一棵二叉排序树之后，我们要清楚如何在一个排序树中查找一个数据。事实上，二叉查找树或二叉排序树可以说是一体两面，没有差别。

二叉查找树具有以下特点：

（1）可以是空集合，但若不是空集合，则节点上一定要有一个键值。
（2）每一个树根的值须大于左子树的值。
（3）每一个树根的值须小于右子树的值。
（4）左右子树也是二叉查找树。
（5）树的每个节点值都不相同。

基本上，只要懂二叉树的排序就可以理解二叉树的查找。只需在二叉树中比较树根及要查找的值，再按左子树<树根<右子树的原则遍历二叉树，就可以找到要查找的值。

接着我们来实现一个二叉查找树的查找程序，首先建立一个二叉查找树，并输入要查找的值。如果节点中有相等的值，就会显示出查找的次数；如果找不到这个值，也将会显示信息。

范例程序：ch06_06.sln

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.IO;
7   using static System.Console;  //导入静态类
```

```
 8
 9  namespace ch06_06
10  {
11      class TreeNode
12      {
13          public int value;
14          public TreeNode left_Node;
15          public TreeNode right_Node;
16
17          public TreeNode(int value)
18          {
19              this.value = value;
20              this.left_Node = null;
21              this.right_Node = null;
22          }
23      }
24      class BinarySearch
25      {
26          public TreeNode rootNode;
27          public static int count = 1;
28          public void Add_Node_To_Tree(int value)
29          {
30              if (rootNode == null)
31              {
32                  rootNode = new TreeNode(value);
33                  return;
34              }
35              TreeNode currentNode = rootNode;
36              while (true)
37              {
38                  if (value < currentNode.value)
39                  {
40                      if (currentNode.left_Node == null)
41                      {
42                          currentNode.left_Node = new TreeNode(value);
43                          return;
44                      }
45                      else
46                          currentNode = currentNode.left_Node;
47                  }
48                  else
49                  {
50                      if (currentNode.right_Node == null)
```

```
51                      {
52                          currentNode.right_Node = new TreeNode(value);
53                          return;
54                      }
55                      else
56                          currentNode = currentNode.right_Node;
57                  }
58              }
59          }
60
61          public bool FindTree(TreeNode node, int value)
62          {
63              if (node == null)
64              {
65                  return false;
66              }
67              else if (node.value == value)
68              {
69                 Write("共查找" + count + "次\n");
70                  return true;
71              }
72              else if (value < node.value)
73              {
74                  count += 1;
75                  return FindTree(node.left_Node, value);
76              }
77              else
78              {
79                  count += 1;
80                  return FindTree(node.right_Node, value);
81              }
82          }
83
84      }
85      class Program
86      {
87          static void Main(string[] args)
88          {
89              int i, value;
90              int[] arr = { 7, 4, 1, 5, 13, 8, 11, 12, 15, 9, 2 };
91              Write("原始数组的内容: \n");
92              for (i = 0; i < 11; i++)
93                 Write("[" + arr[i] + "] ");
```

```
 94              WriteLine();
 95              BinarySearch tree = new BinarySearch();
 96              for (i = 0; i < 11; i++) tree.Add_Node_To_Tree(arr[i]);
 97              Write("请输入要查找值： ");
 98              value = int.Parse(ReadLine());
 99              if (tree.FindTree(tree.rootNode, value))
100                 Write("您要查找的值 [" + value + "] 找到了！\n");
101           else
102                 Write("抱歉，没有找到。\n");
103
104              ReadKey();
105           }
106       }
107   }
```

范例程序的执行结果如图 6-39 所示。

```
原始数组的内容：
[7] [4] [1] [5] [13] [8] [11] [12] [15] [9] [2]
请输入要查找的值：12
共查找5次
您要查找的值 [12] 找到了！
```

图 6-39

以上程序的二叉查找树有如图 6-40 所示的结构。

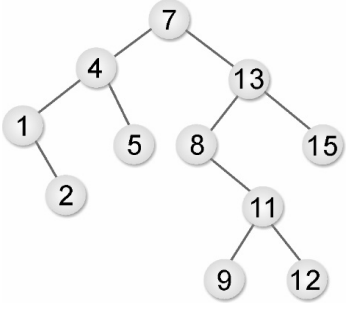


图 6-40

范例 6.5.2　关于二叉查找树（Binary Search Tree）的叙述，哪一个是错误的？

（A）二叉查找树是一棵完整二叉树（Complete Binary Tree）。
（B）可以是歪斜树（Skewed Binary Tree）。
（C）一节点最多只有两个子节点（Child Node）。
（D）一节点的左子节点的键值不会大于右节点的键值。

解答 (A)

6.5.3 线索二叉树

虽然我们把树转换为二叉树可减少空间的浪费——由 2/3 降低到 1/2，但是如果各位读者仔细观察之前我们使用链表建立的 n 节点二叉树，就会发现用来指向左右两节点的指针只有 n-1 个链接，另外的 n+1 个指针都是空链接。

所谓“线索二叉树”（Threaded Binary Tree），就是把这些空的链接加以利用，再指到树的其他节点，而这些链接就称为“线索”（Thread），这棵树就称为线索二叉树（Threaded Binary Tree）。将二叉树转换为线索二叉树的步骤如下：

步骤 01 先将二叉树通过中序遍历方式按序排出，并将所有空链接改成线索。

步骤 02 如果线索链接指向该节点的左链接，则将该线索指到中序遍历顺序下的前一个节点。

步骤 03 如果线索链接指向该节点的右链接，则将该线索指到中序遍历顺序下的后一个节点。

步骤 04 指向一个空节点，并将此空节点的右链接指向自己，而空节点的左子树是此线索二叉树。

线索二叉树的基本结构如下：

- LBIT：左控制位。
- LCHILD：左子树链接。
- DATA：节点数据。
- RCHILD：右子树链接。
- RBIT：右控制位。

与链表所建立的二叉树不同之处在于，为了区别正常指针或线索而加入的两个字段：LBIT 和 RBIT。

- 如果 LCHILD 为正常指针，则 LBIT=1。
- 如果 LCHILD 为线索，则 LBIT=0。
- 如果 RCHILD 为正常指针，则 RBIT=1。
- 如果 RCHILD 为线索，则 RBIT=0。

节点的声明方式如下：

```
class ThreadedNode
{
     int data,lbit,rbit;
     ThreadedNOde lchild;
     ThreadedNode rchild;
    //构造函数
    public ThreadedNode(int data,int lbit,int rbit)
    {
```

```
            初始化程序代码
        }
    }
```

接着我们来练习如何将如图 6-41 所示的二叉树转为线索二叉树。

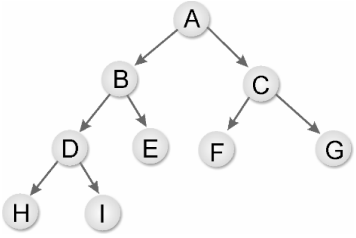


图 6-41

步骤：

步骤 01 以中序遍历二叉树：HDIBEAFCG。

步骤 02 找出相对应的线索二叉树，并按照 HDIBEAFCG 顺序求得如图 6-42 所示的结果。

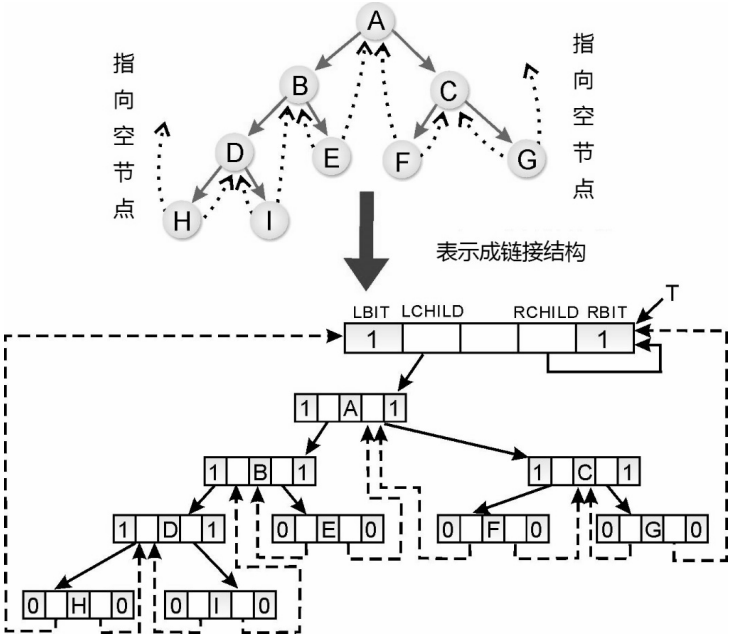


图 6-42

以下是使用线索二叉树的优缺点。

优点：

（1）在二叉树进行中序遍历时，不需要使用堆栈处理，但一般二叉树需要。

（2）由于充分使用空链接，所以避免了链接闲置浪费的情况。另外，中序遍历时的速度也较快，节省了不少时间。

（3）任一个节点都容易找出它的中序先行者与中序后继者，在中序遍历时可以不使用堆栈或递归。

缺点：

（1）在加入或删除节点时的速度比一般二叉树慢。

（2）线索子树间不能共用。

以下 C#程序是利用线索二叉树来遍历某一节点 X 的中序前行者与中序后续者。

范例程序：ch06_07.sln

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using static System.Console;//导入静态类

namespace ch06_07
{
    //线索二叉树中的节点声明
    class ThreadNode
    {
        public int value;
        public int left_Thread;
        public int right_Thread;
        public ThreadNode left_Node;
        public ThreadNode right_Node;
        // TreeNode 构造函数
        public ThreadNode(int value)
        {
            this.value = value;
            this.left_Thread = 0;
            this.right_Thread = 0;
            this.left_Node = null;
            this.right_Node = null;
        }
    }
    //线索二叉树的类声明
    class Threaded_Binary_Tree
    {
        public ThreadNode rootNode; //线索二叉树的根节点

```

```
34          //无传入参数的构造函数
35          public Threaded_Binary_Tree()
36          {
37              rootNode = null;
38          }
39
40          //构造函数:建立线索二叉树，传入参数为一数组
41          //数组中的第一个数据是用来建立线索二叉树的树根节点
42          public Threaded_Binary_Tree(int[] data)
43          {
44              for (int i = 0; i < data.Length; i++)
45                  Add_Node_To_Tree(data[i]);
46          }
47          //将指定的值加入到线索二叉树
48          void Add_Node_To_Tree(int value)
49          {
50              ThreadNode newnode = new ThreadNode(value);
51              ThreadNode current;
52              ThreadNode parent;
53              ThreadNode previous = new ThreadNode(value);
54              int pos;
55              //设置线索二叉树的开头节点
56              if (rootNode == null)
57              {
58                  rootNode = newnode;
59                  rootNode.left_Node = rootNode;
60                  rootNode.right_Node = null;
61                  rootNode.left_Thread = 0;
62                  rootNode.right_Thread = 1;
63                  return;
64              }
65              //设置开头节点所指的节点
66              current = rootNode.right_Node;
67              if (current == null)
68              {
69                  rootNode.right_Node = newnode;
70                  newnode.left_Node = rootNode;
71                  newnode.right_Node = rootNode;
72                  return;
73              }
74              parent = rootNode; //父节点是开头节点
75              pos = 0; //设置二叉树中的行进方向
76              while (current != null)
```

```
77              {
78                  if (current.value > value)
79                  {
80                      if (pos != -1)
81                      {
82                          pos = -1;
83                          previous = parent;
84                      }
85                      parent = current;
86                      if (current.left_Thread == 1)
87                          current = current.left_Node;
88                      else
89                          current = null;
90                  }
91                  else
92                  {
93                      if (pos != 1)
94                      {
95                          pos = 1;
96                          previous = parent;
97                      }
98                      parent = current;
99                      if (current.right_Thread == 1)
100                         current = current.right_Node;
101                     else
102                         current = null;
103                 }
104             }
105             if (parent.value > value)
106             {
107                 parent.left_Thread = 1;
108                 parent.left_Node = newnode;
109                 newnode.left_Node = previous;
110                 newnode.right_Node = parent;
111             }
112             else
113             {
114                 parent.right_Thread = 1;
115                 parent.right_Node = newnode;
116                 newnode.left_Node = parent;
117                 newnode.right_Node = previous;
118             }
119             return;
```

```
120         }
121         //线索二叉树中序遍历
122         public void Print()
123         {
124             ThreadNode tempNode;
125             tempNode = rootNode;
126             do
127             {
128                 if (tempNode.right_Thread == 0)
129                     tempNode = tempNode.right_Node;
130                 else
131                 {
132                     tempNode = tempNode.right_Node;
133                     while (tempNode.left_Thread != 0)
134                         tempNode = tempNode.left_Node;
135                 }
136                 if (tempNode != rootNode)
137                     WriteLine("[" + tempNode.value + "]");
138             } while (tempNode != rootNode);
139         }
140     }
141     class Program
142     {
143         static void Main(string[] args)
144         {
145             WriteLine("线索二叉树经建立后，以中序遍历有排序的效果");
146             WriteLine("除了第一个数字作为线索二叉树的开头节点外");
147             int[] data1 = { 0, 10, 20, 30, 100, 399, 453, 43, 237, 373, 655 };
148             Threaded_Binary_Tree tree1 = new Threaded_Binary_Tree(data1);
149             WriteLine("====================================");
150             WriteLine("范例 1 ");
151             WriteLine("数字从小到大的排序顺序结果为：");
152             tree1.Print();
153             int[] data2 = { 0, 101, 118, 87, 12, 765, 65 };
154             Threaded_Binary_Tree tree2 = new Threaded_Binary_Tree(data2);
155             WriteLine("====================================");
156             WriteLine("范例 2 ");
157             WriteLine("数字从小到大的排序顺序结果为：");
158             tree2.Print();
159             ReadKey();
160         }
161     }
162 }
```

范例程序的执行结果如图 6-43 所示。

```
线索二叉树经建立后，以中序遍历有排序的效果
除了第一个数字作为线索二叉树的开头节点外
====================================
范例 1
数字从小到大的排序顺序结果为：
[10]
[20]
[30]
[43]
[100]
[237]
[373]
[399]
[453]
[655]
====================================
范例 2
数字从小到大的排序顺序结果为：
[12]
[65]
[87]
[101]
[118]
[765]
```

图 6-43

6.6 树的二叉树表示法

在前面的小节中介绍了许多关于二叉树的操作，然而二叉树只是树形结构的特例，广义的树形结构其父节点可拥有多个子节点，我们姑且将这样的树称为多叉树。由于二叉树的链接浪费率最低，因此如果把树转换为二叉树来操作，就会增加许多操作上的便利。

6.6.1 树转化为二叉树

对于将一般树形结构转化为二叉树，使用的方法为“CHILD-SIBLING”（leftmost-child-next-right-sibling）法则。以下是其执行步骤：

步骤 01 将节点的所有兄弟节点用横线连接起来。

步骤 02 删掉所有与子节点间的连接，只保留与最左子节点的连接。

步骤 03 顺时针旋转 45° 。

请按照下面的范例过程实际转换一次，就可以有更清楚的认识，步骤如图 6-44~图 6-47 所示。

步骤 01 将树的各层兄弟用横线连接起来。

步骤 02 删掉所有子节点间的连接，只保留最左边的父子节点的连接。

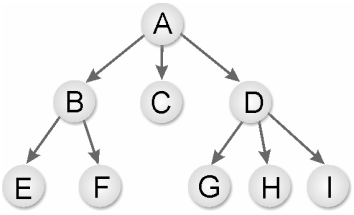


图 6-44

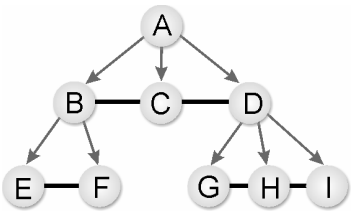


图 6-45

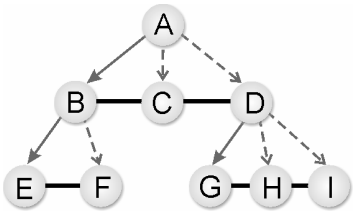


图 6-46

步骤 03　顺时针转 45°。

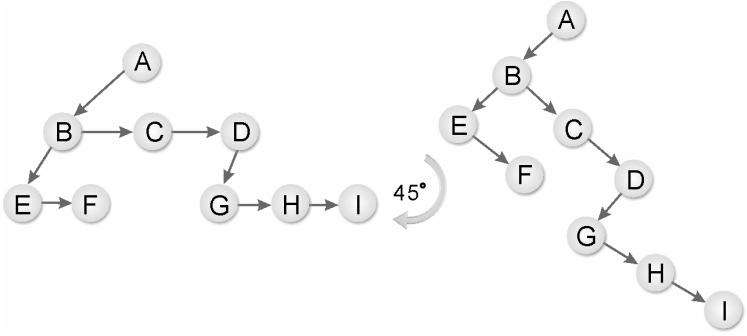


图 6-47

二叉树转换为树

既然树可转化为二叉树，当然也可以将二叉树转化为树（即多叉树），如图 6-48 所示。

这其实就是树转化为二叉树的逆向步骤，方法也很简单。首先是逆时针旋转 45°，如图 6-49 所示。

另外，由于(ABE)(DG)左子树代表父子关系，而(BCD)(EF)(GH)右子树代表兄弟关系，按这种父子关系增加连接，同时删除兄弟节点间的连接，结果如图 6-50 所示。

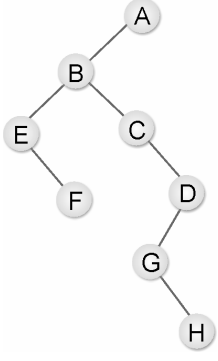


图 6-48

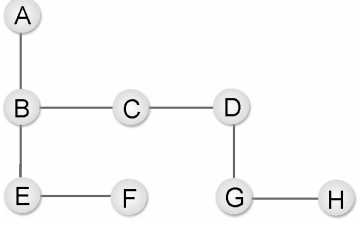


图 6-49

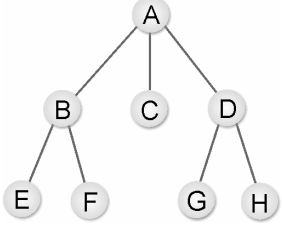


图 6-50

范例 6.6.1　将如图 6-51 所示的树转化为二叉树。

解答

（1）将树的各阶层兄弟用平行线连接起来，如图 6-52 所示。

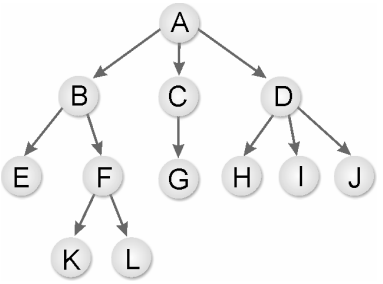


图 6-51

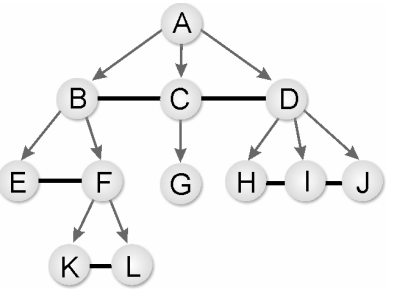


图 6-52

（2）删除所有子节点间的串联，只保留最左边的子节点，如图 6-53 所示。

（3）顺时针旋转 45°，如图 6-54 所示。

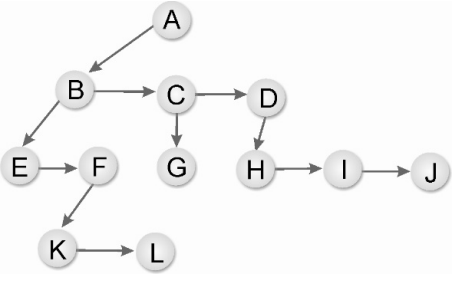


图 6-53

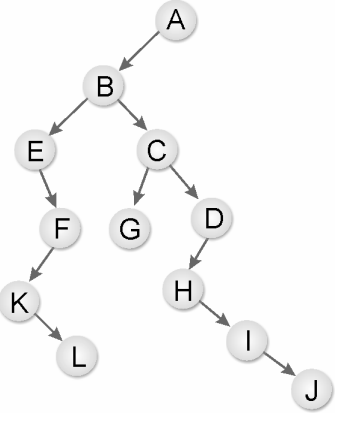


图 6-54

6.6.2 树林转化为二叉树

除了一棵树可以转化为二叉树外，其实好几棵树所形成的树林也可以转化成二叉树。

（1）从左到右将每棵树的树根（Root）连接起来。

（2）仍然利用树转化为二叉树的方法操作。

接着以下面的树林（如图 6-55）为范例进行介绍。

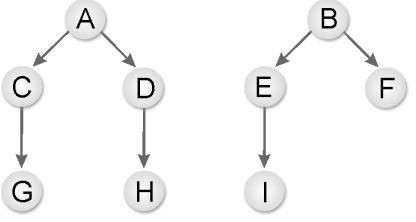


图 6-55

步骤 01 将各树的树根从左到右连接，如图 6-56 所示。

步骤 02 利用树转换为二叉树的原则，如图 6-57 所示。

步骤 03 顺时针旋转 45°，如图 6-58 所示。

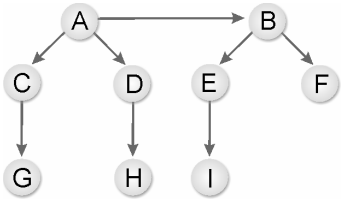


图 6-56

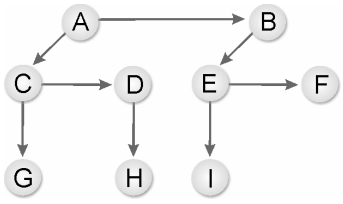


图 6-57

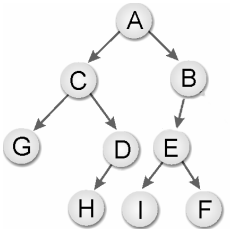


图 6-58

二叉树转换成树林

二叉树转换成森林的方法则是按照森林转化为二叉树的方法倒推回去，如图 6-59 所示的二叉树。

首先，把原图逆时旋转 45°，如图 6-60 所示。

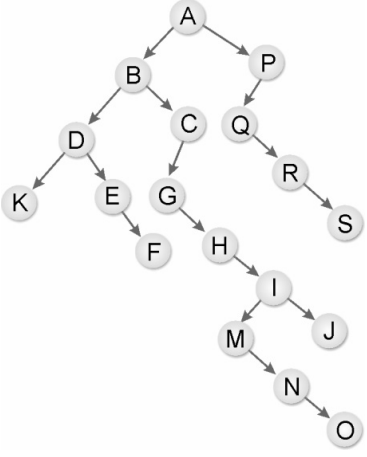


图 6-59

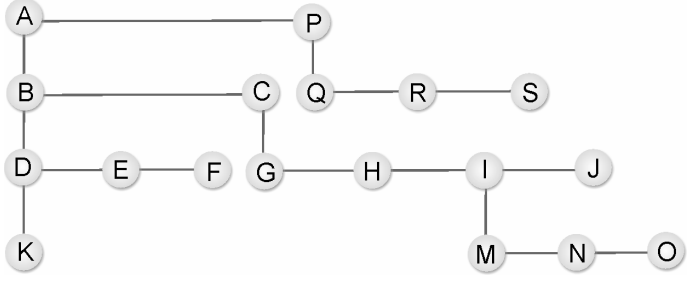


图 6-60

再按照左子树为父子关系，右子树为兄弟关系的原则逐步划分，如图 6-61 所示。

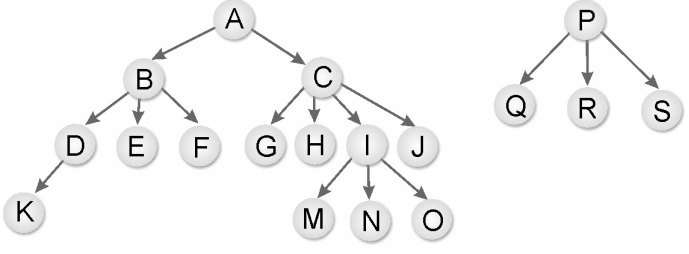


图 6-61

6.6.3　树与森林的遍历

除了二叉树的遍历可以有中序遍历、前序遍历与后序遍历三种方式外，树与森林的遍历也是这三种。但方法略有差异，下面我们以范例来说明。

假设树根为 R，且此树有 n 个节点，并可分成如图 6-62 所示的 m 个子树，分别是 T_1， T_2，T_3…T_m。

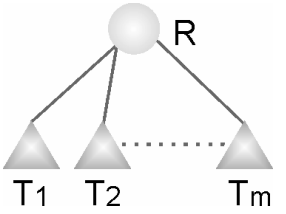


图 6-62

三种遍历方式的步骤如下：

- **中序遍历（Inorder Traversal）**

（1）以中序法遍历 T_1。
（2）访问树根 R。
（3）再以中序法遍历 T_2，T_3，…T_m。

- **前序遍历（Preorder Traversal）**

（1）访问树根 R。
（2）再以前序法依次遍历 T_1，T_2，T_3，…T_m。

- **后序遍历（Postorder Traversal）**

（1）以后序法依次访问 T_1，T_2，T_3，…T_m。
（2）访问树根 R。

至于森林的遍历方式则从树的遍历衍生过来。步骤如下：

- **中序遍历（Inorder Traversal）**

（1）如果森林为空，则直接返回。
（2）以中序遍历第一棵树的子树群。
（3）中序遍历森林中第一棵树的树根。
（4）按中序法遍历森林中其他的树。

- **前序遍历（Preorder Traversal）**

（1）如果森林为空，则直接返回。
（2）遍历森林中第一棵树的树根。
（3）按前序遍历第一棵树的子树群。
（4）按前序法遍历森林中其他的树。

- **后序遍历（Postorder Traversal）**

（1）如果森林为空，则直接返回。
（2）按后序遍历第一棵树的子树。
（3）按后序法遍历森林中其他的树。
（4）遍历森林中第一棵树的树根。

范例▶ 6.6.2　将下列森林（如图 6-63）转化为二叉树，并分别求出转化前森林与转化后二叉树的中序、前序与后序遍历结果。

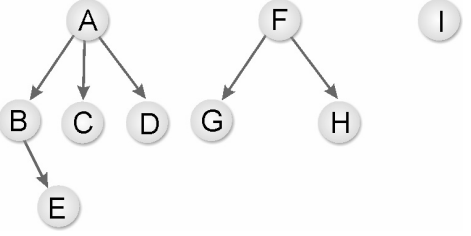


图 6-63

解答▶ 步骤如图 6-64~图 6-66 所示。

（1）

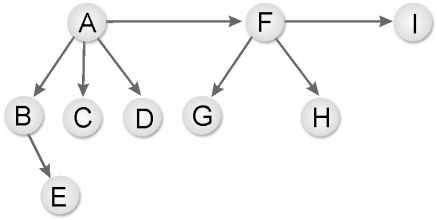


图 6-64

（2）

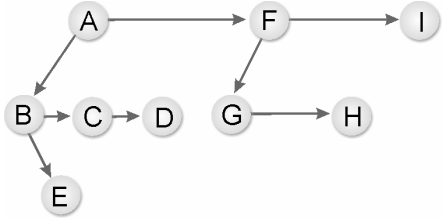


图 6-65

（3）

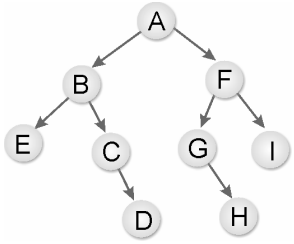


图 6-66

森林遍历：

（1）中序遍历为 EBCDAGHFI；
（2）前序遍历为 ABECDFGHI；
（3）后序遍历为 EBCDGHIFA。

二叉树遍历：

（1）中序遍历为 EBCDAGHFI；
（2）前序遍历为 ABECDFGHI；
（3）后序遍历为 EDCBHGIFA。
（注意，转化前后的后序遍历结果不同）

范例 6.6.3　求图 6-67 所示的森林转化为二叉树前后的中序、前序与后序遍历结果。

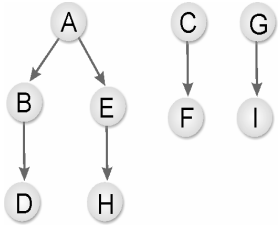


图 6-67

森林的遍历：

（1）中序遍历为 DBHEAFCIG；

（2）前序遍历为 ABDEHCFGI；
（3）后序遍历为 DHEBFIGCA。

转换为二叉树如图 6-68 所示。

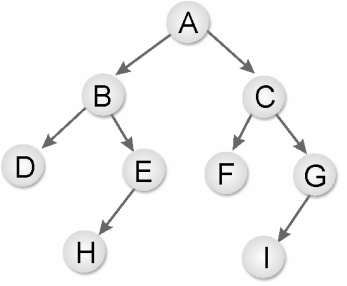


图 6-68

二叉树遍历：
（1）中序遍历为 DBHEAFCIG；
（2）前序遍历为 ABDEHCFGI；
（3）后序遍历为 DHEBFIGCA。

6.6.4 确定唯一二叉树

在二叉树的三种遍历方法中，如果有中序与前序的遍历结果或中序与后序的遍历结果，就可以从这些结果中求得唯一的二叉树。不过，如果只具备前序与后序的遍历结果，则无法确定唯一的二叉树。

现在来看一个范例。例如二叉树的中序遍历为 BAEDGF，前序遍历为 ABDEFG，请画出唯一的二叉树。

解答▶
中序遍历：左子树 树根 右子树
前序遍历：树根 左子树 右子树

如图 6-69~图 6-71 所示。

（1）

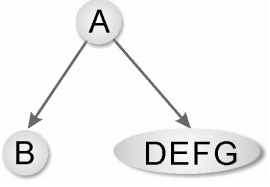


图 6-69

（2）

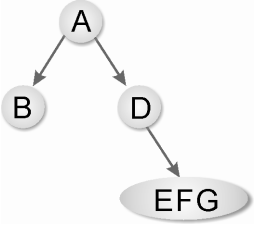


图 6-70

（3）

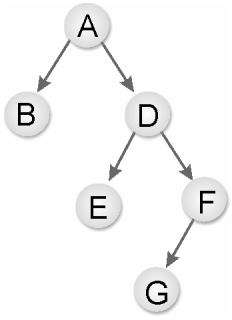


图 6-71

范例 6.6.4　某二叉树的中序遍历为 HBJAFDGCE，后序遍历为 HJBFGDECA，请绘出此二叉树。

解答

中序遍历：左子树　树根　右子树

后序遍历：左子树　右子树　树根

如图 6-72~图 6-75 所示。

（1）

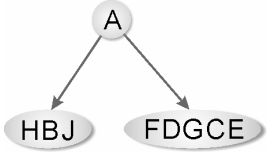


图 6-72

（2）

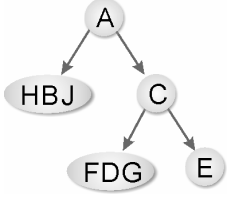


图 6-73

（3）

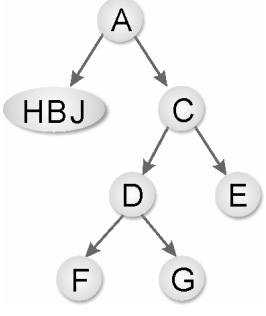


图 6-74

（4）

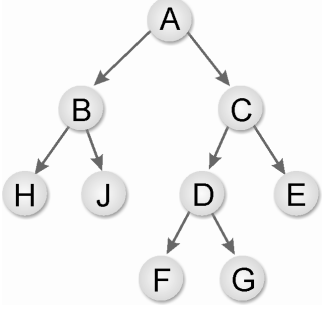


图 6-75

6.7 优化二叉查找树

之前我们讲过，如果一个二叉树符合“每一个节点的数据大于左子节点且小于右子节点”，这棵树便具有二叉查找树的特性。而所谓的优化二叉查找树，简单来说，就是在所有可能的二叉查找树中，有最小查找成本的二叉树。

6.7.1 扩充二叉树

至于什么是最小查找成本呢？就让我们先从扩充二叉树（Extension Binary Tree）谈起。任何一个二叉树中，若具有 n 个节点，则有 n-1 个非空链接和 n+1 个空链接。如果在每一个空链接加上一个特定节点，则称为外节点，其余的节点称为内节点，因而定义此种树为“扩充二叉树”。另外定义：外径长＝所有外节点到树根距离的总和，内径长＝所有内节点到树根距离的总和。我们将以图 6-76 中的图（a）和图（b）来说明它们的扩充二叉树的绘制过程，如图 6-77 和图 6-78 所示。

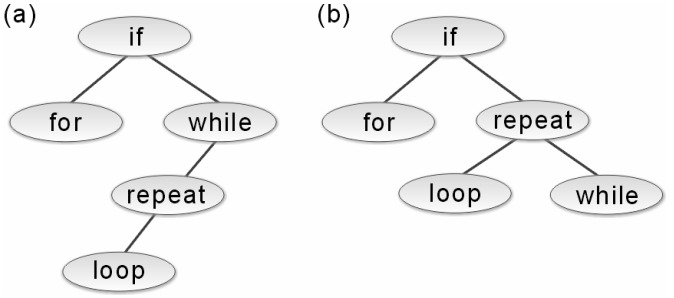


图 6-76

代表外部节点。

（a）

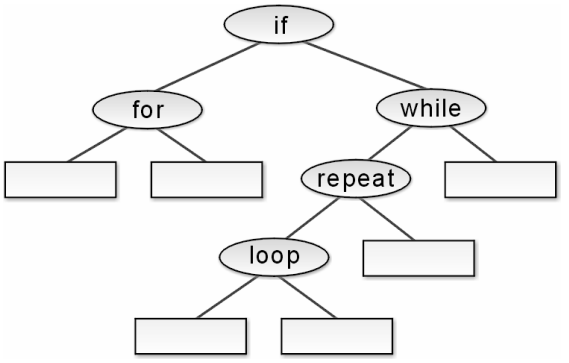


图 6-77

外径长为(2+2+4+4+3+2)=17；内径长为(1+1+2+3)=7。

（b）

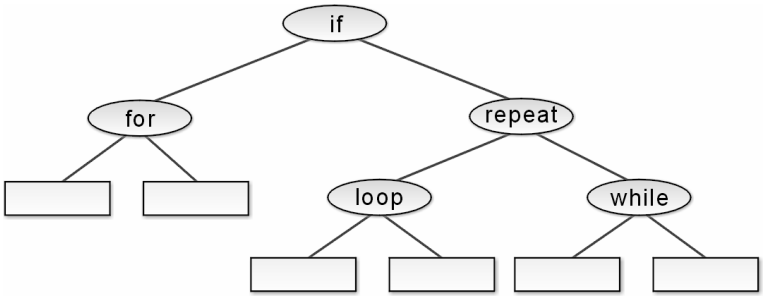


图 6-78

外径长为(2+2+3+3+3+3)=16；内径长为(1+1+2+2)=6。

以图 6-77 和图 6-78 为例，如果每个外部节点都有加权值（如查找概率等），则外径长必须考虑相关加权值，或者称为加权外径长.下面将讨论 6-77 和图 6-78 的加权外径长，如图 6-79 和图 6-80 所示。

对图 6-77 来说，2×3 + 4×3 + 5×2 + 15×1 = 43。

对图 6-78 来说，2×2 + 4×2 + 5×2 + 15×2 = 52。

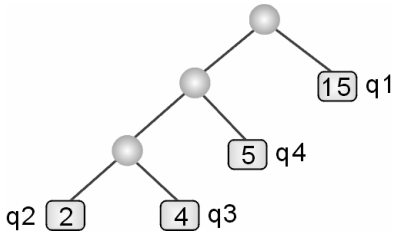


图 6-79

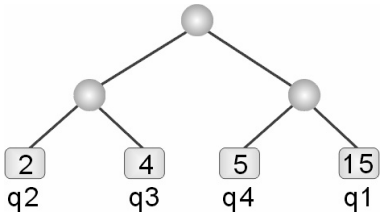


图 6-80

6.7.2　霍夫曼树

霍夫曼树经常应用于处理数据压缩，可以根据数据出现的频率来构建的二叉树。例如数据的存储和传输是数据处理的两个重要领域，两者都与数据量的大小息息相关，而霍夫曼树正好可以用于数据压缩的算法。

简单来说，如果有 n 个权值（$q_1, q_2 \ldots q_n$），且构成一个有 n 个节点的二叉树，每个节点的外部节点的权值为 qi，则加权外径长度最小的就称为“优化二叉树”或“霍夫曼树”（Huffman Tree）。对上一小节中，图图 6-77 和图图 6-78 的二叉树而言，图图 6-77 就是二者的优化二叉树。接下来我们将说明，对一个含权值的链表，该如何求其优化二叉树。步骤如下：

步骤 01　产生两个节点，对数据中出现过的每一元素各自产生一个树叶节点，并赋予树叶节点该元素的出现频率。

步骤 02　令 N 为 T_1 和 T_2 的父节点，T_1 和 T_2 是 T 中出现频率最低的两个节点，令 N 节点的出现频率等于 T_1 和 T_2 出现频率的总和。

步骤03 消去步骤 2 的两个节点，插入 N，再重复步骤 1。

我们将利用以上的步骤来实现求取霍夫曼树的过程，假设现在 5 个字母 BDACE 的出现频率分别为 0.09、0.12、0.19、0.21 和 0.39，请说明霍夫曼树的构建过程。

（1）取出最小的 0.09 和 0.12，合并成另一棵新的二叉树，其根节点的频率为 0.21，如图 6-81 所示。

（2）再取出 0.19 和 0.21 为根的二叉树合并后，得到 0.40 为根的新二叉树，如图 6-82 所示。

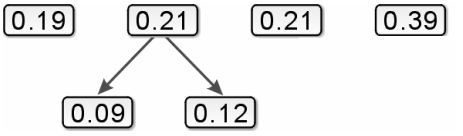


图 6-81

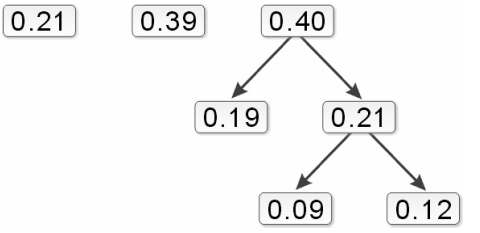


图 6-82

（3）取出 0.21 和 0.39 的节点，产生频率为 0.6 的新节点，得到右边的新二叉树，如图 6-83 所示。

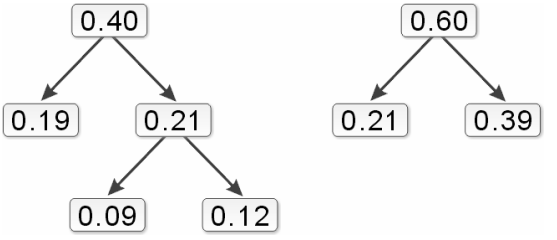


图 6-83

最后取出 0.40 和 0.60 两个二叉树的根节点，将它们合并成频率为 1.0 的节点。至此二叉树就完成了。

6.8 平衡树

二叉查找树的缺点是无法永远保持在最佳状态。在加入的数据部分已排序的情况下，极有可能会产生斜二叉树，因而使树的高度增加，导致查找效率降低。因此，一般的二叉查找树不适用于数据经常变动（加入或删除）的情况。相对地，比较适合不会变动的数据，如程序设计语言中的“保留字”等。

6.8.1 平衡树的定义

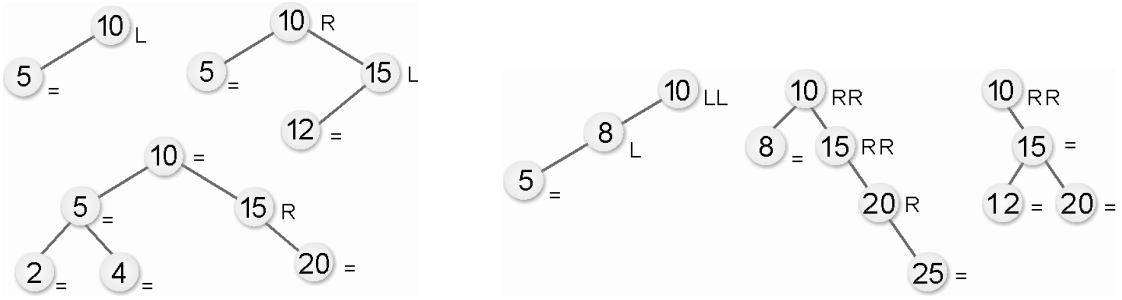
所谓平衡树（Balanced Binary Tree），又称为 AVL 树（是由 Adelson-Velskii 和 Landis 两个人发明的），它本身也是一棵二叉查找树。在 AVL 树中，每次在插入数据或删除数据后，必要的时候会对二叉树做一些高度的调整，而这些调整就是要让二叉查找树的高度随时维持平

衡。T 是一个非空的二叉树，T_l 及 T_r 分别是它的左右子树，若符合以下两个条件，则称 T 是高度平衡树。

（1）T_l 和 T_r 也是高度平衡树。

（2）$|h_l-h_r|\leqslant 1$，h_l 和 h_r 分别为 T_l 和 T_r 的高度，也就是所有内部节点的左右子树高度相差必定小于或等于 1。

如图 6-84 所示的平衡树。



（a）AVL 树　　（b）非 AVL 树

图 6-84

至于如何调整一棵二叉搜索树成为一棵平衡树，最重要的是找出“不平衡点”，再按照以下 4 种不同的旋转形式重新调整其左右子树的长度。首先，令新插入的节点为 N，且其最近的一个具有±2 的平衡因子节点为 A，下一层为 B，再下一层为 C，分述如下。

- 左左型（LL 型，如图 6-85 所示）

图 6-85

- 左右型（LR 型，如图 6-86 所示）

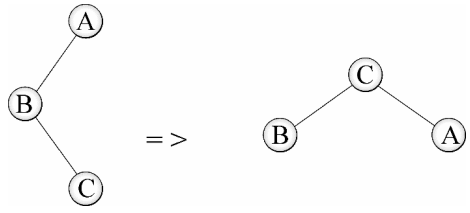


图 6-86

- 右右型（RR 型，如图 6-87 所示）

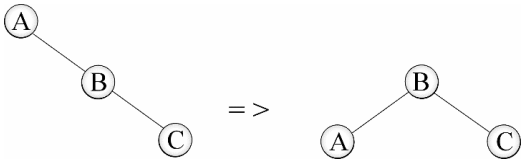


图 6-87

- 右左（RL 型，如图 6-88 所示）

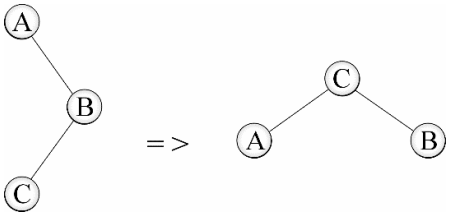


图 6-88

现在我们来实现一个范例，如图 6-89 所示的二叉树原来是平衡的，加入节点 12 后就不平衡了，请重新调整为平衡树，但不可破坏原有的次序结构。

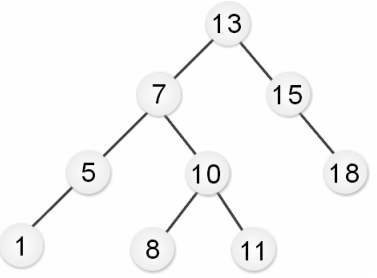


图 6-89

调整结果后如图 6-90 所示。

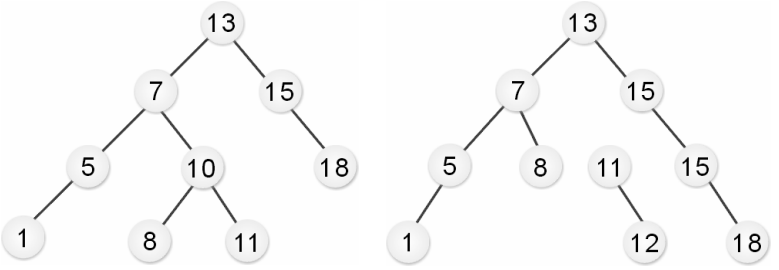


图 6-90

6.8.2 B 树

B 树（B Tree）是一种高度大于等于 1 的 m 阶查找树，也是一种平衡树概念的延伸，由 Bayer 和 Mc Creight 两位专家提出。主要有以下特点：

（1）B 树上每一个节点都是 m 阶节点。

（2）每一个 m 阶节点存放的键值最多为 m-1 个。

（3）每一个 m 阶节点度数均小于等于 m。

（4）除非是空树，否则树根节点至少必须有两个以上的子节点。

（5）除了树根和树叶节点外，每一个节点最多不超过 m 个子节点，但至少包含 $\lceil m/2 \rceil$ 个子节点。

（6）每个树叶节点到树根节点所经过的路径长度都一致，也就是说，所有的树叶节点都必须在同一层（Level）。

（7）当要增加树的高度时，处理方法就是将该树根节点一分为二。

（8）B 树其键值分别为 k_1、k_2、k_3、k_4…k_{m-1}，则 $k_1<k_2<k_3<k_4\ldots<k_{m-1}$。

（9）B 树的节点表示法为 $P_{0,1}$，k_1，$P_{1,2}$，$k_2\ldots P_{m-2,m-1}$，k_{m-1}，$P_{m-1,m}$。

其节点结构如下所示。

$P_{0,1}$	k_1	$P_{1,2}$	k_2	$P_{2,3}$	k_3 ·····	k_{m-1}	$P_{m-1,m}$

其中，$k_1<k_2<k_3\ldots<k_{m-1}$。

（1）$P_{0,1}$ 指针所指向的子树 T_1 中的所有键值均小于 k_1。

（2）$P_{1,2}$ 指针所指向的子树 T_2 中的所有键值均大于等于 k_1 且小于 k_2。

（3）以此类推，$P_{m-1,m}$ 指针所指向的子树 T_m 中所有键值均大于等于 k_{m-1}。

课 后 习 题

1. 一般树形结构在计算机内存中的存储方式是以链表为主，对于 n 叉树（n-way 树）来说，我们必须取 n 为连接个数的最大固定长度，请说明为了改进存储空间浪费的缺点，为何经常使用二叉树（Binary Tree）结构来取代树形结构。

2. 下列哪一种不是树（Tree）？

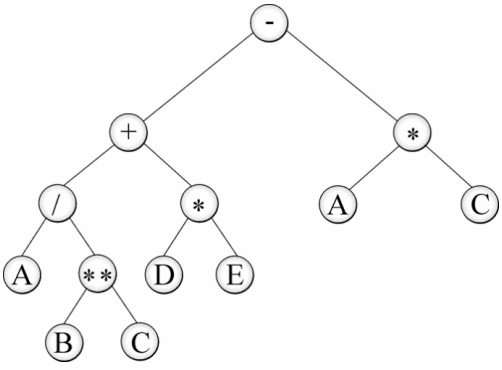
（a）一个节点；

（b）环形链表；

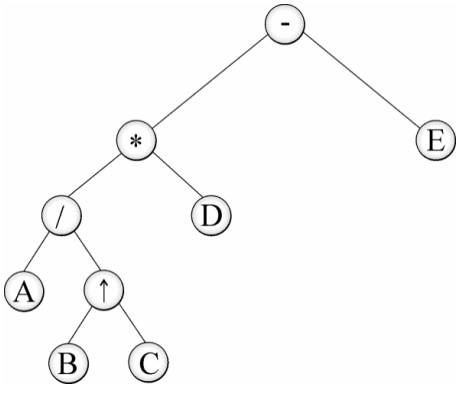
（c）一个没有回路的连通图（Connected Graph）；

（d）一个边数比点数少 1 的连通图。

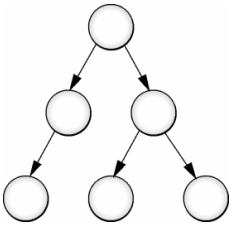
3. 请问以下二叉树的中序法、后序法及前序法表达式分别是什么？



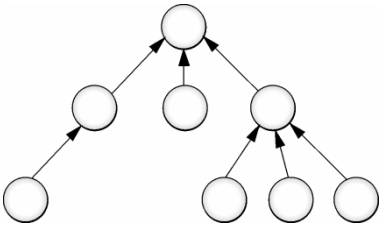
4. 请问以下二叉树的中序法、前序法及后序法表达式分别是什么？



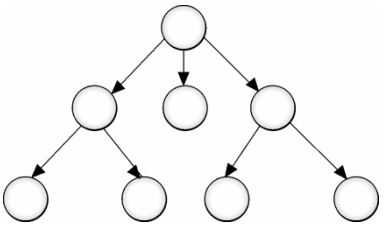
5. 试以链表来描述以下树形结构的数据结构。



（a）



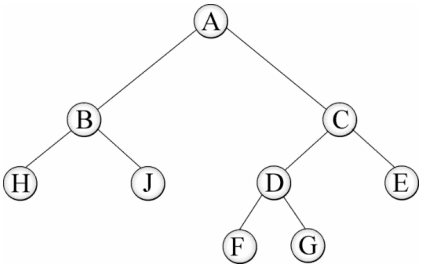
（b）



（c）

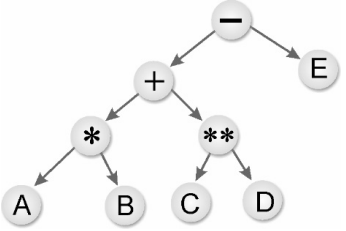
6. 假如有一个非空树，其度数为 5，已知度数为 i 的节点数有 i 个，其中 $1 \le i \le 5$，请问终端节点数总数是多少？

7. 请问以下二叉树的中序、前序及后序遍历结果分别是什么？

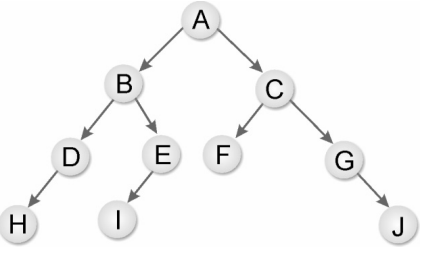


8. 用二叉查找树去表示 n 个元素时，最小高度和最大高度的二叉查找树（Height of Binary Search Tree）其值分别是什么？

9. 请问以下运算二叉树的中序法、后序法及前序法表示法分别是多少？



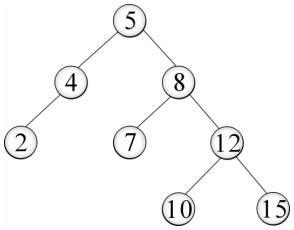
10. 下图为一个二叉树。



（1）请问此二叉树的前序遍历、中序遍历及后序遍历结果。
（2）空的线索二叉树是什么？
（3）以线索二叉树表示其存储情况。

11. 形成 8 层的平衡树最少需要几个节点？

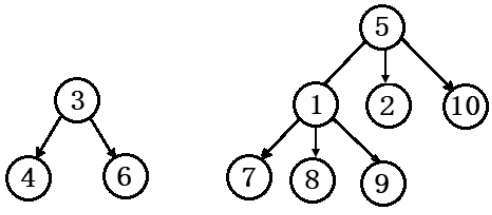
12. 在下图平衡二叉树中加入节点 11，重新调整后的平衡树是什么？



13. 请说明二叉搜索树的特点。

14.试写出一个伪码 SWAPTREE(T) 将二叉树 T 的所有节点的左右子节点对换，并说明之。

15. （1）用一维数组 A[1:10] 来表示下图的两棵树。



（2）利用数据结构设计一算法，该算法将两棵树合并（Union）成为一棵树。

16. 假设一棵二叉树其中序遍历为 BAEDGF，前序遍历为 ABEDFG，求此二叉树。

17. 试述如何对一个二叉树进行中序遍历不用堆栈或递归？

18. 将下图的树转化为二叉树。