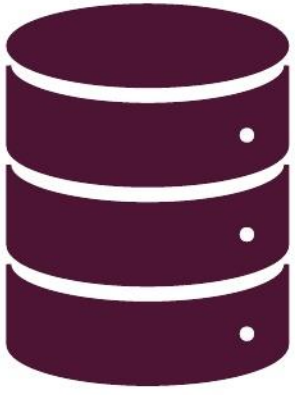


# 数据结构

WENYE 李
CUHK-SZ

# 大纲

树

树的类型　实现

示例

线性数据结构，如数组、链表、栈和队列　所有元素都按顺序排列。

选择数据结构时考虑的因素

需要存储什么类型的数据?

可能某种数据结构最适合某种数据。　操作成本

例如，我们有一个简单的列表，我们必须对其执行搜索操作;然后，我们可以创建一个数组，其中的元素按排序顺序存储，以执行二分查找。二分查找对于简单列表的工作速度非常快，因为它将搜索空间分成了一半。

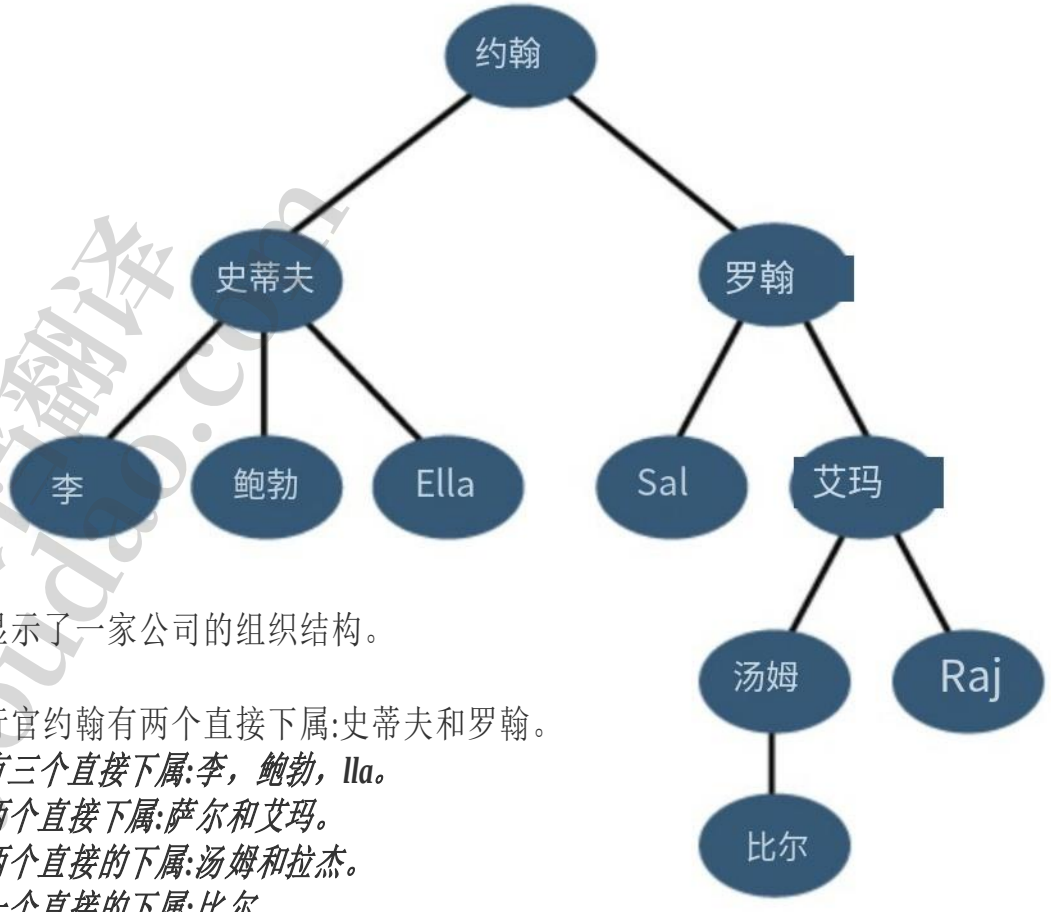
内存使用情况

有时候，我们想要一个使用更少内存的数据结构。

# 树

*树是一种数据结构*
表示层次数据。

假设我们想要显示员工及其职位以等级形式存在可以表示为显示。




这棵树显示了一家公司的组织结构。

首席执行官约翰有两个直接下属:史蒂夫和罗翰。
*史蒂夫有三个直接下属:李，鲍勃，lla。*
*鲍勃有两个直接下属:萨尔和艾玛。*
*艾玛有两个直接的下属:汤姆和拉杰。*
*汤姆有一个直接的下属:比尔。*

在这个结构中，根在上面，分支在上面都是向下移动的。因此，我们可以说树形数据结构是一种有效的方法以分层的方式存储数据。

# 树

树数据结构被定义为称为节点的对象或实体的集合，这些节点连接在一起以表示或模拟层次结构。

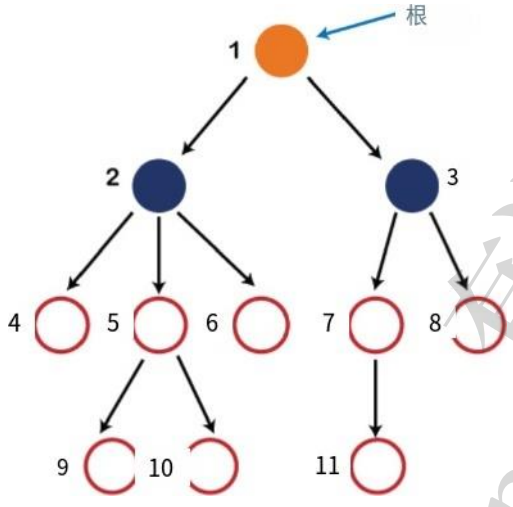
树是一种非线性数据结构，因为它不以顺序方式存储。它

是一个层次结构，因为树中的元素被安排在多个层次上。 在树中，最顶层的节点被称为根节点。

每个节点包含一些数据，这些数据可以是任何类型。

每个节点包含其他可以称为子节点的链接或引用。

# 基本条款




根:根节点是树层次结构中最高的节点。换句话说，根节点就是没有任何父节点的节点。

子节点:如果该节点是任何节点的后代，那么该节点被称为子节点。

父节点:如果该节点包含任何子节点，则称该节点为该子节点的父节点。

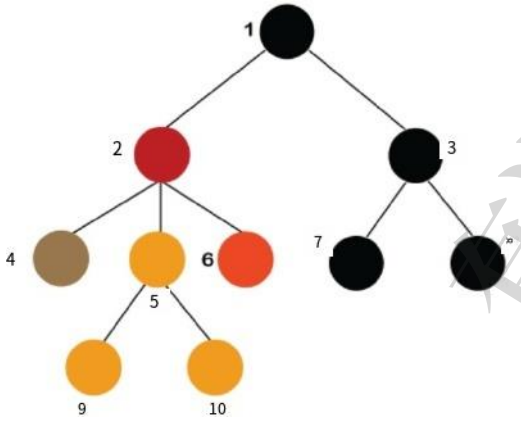
兄弟节点:具有相同父节点的节点称为兄弟节点。

叶节点:树的节点，没有任何子节点，称为叶节点。叶节点是树的最底层节点。叶节点也可以称为外部节点。

内部节点:一个节点至少有一个子节点称为内部节点。祖先节点:一个节点的祖先是从根节点到该节点的路径上的任何前一个节点。根节点没有任何祖先，例如节点 1、2、5 是节点 10 的祖先。

后代:给定节点的直接继承者称为节点的后代，例如，10 是节点 5 的后代。

# 属性




递归数据结构:树可以被定义为递归的。根节点包含一个链接到它所有子树的根。左边的子树显示为黄色，右边的子树显示为红色。左子树可以进一步分裂为三种不同颜色的子树。递归意味着以一种自相似的方式减少某些东西。

边的数量:如果有 n 个节点，那么有 n-1 条边。结构中的每个箭头代表链接或路径。除根节点外，每个节点将至少有一个进入的链接，称为边。父子关系会有一个链接。

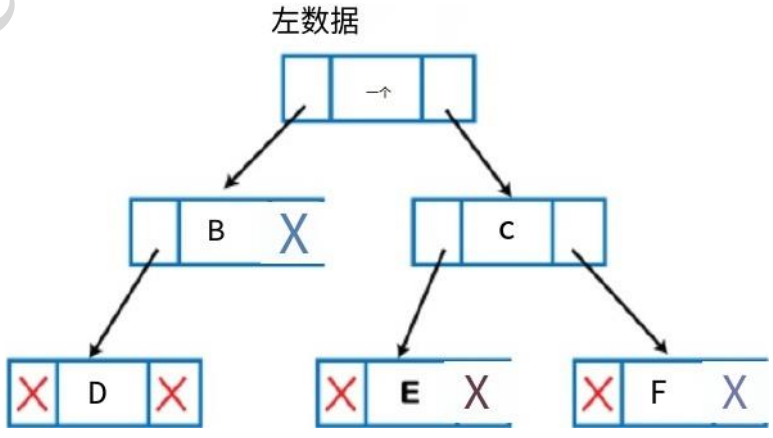
节点 x 的深度:从根节点到节点 x 的路径长度，即。，根节点和节点 x 之间的边数，根节点的深度为 0。

节点 x 的高度:从节点 x 到叶节点的最长路径。

# 实现

树可以通过在指针的帮助下动态创建节点来创建。一个节点包含三个字段。第二个字段存储数据;第一个字段存储左子节点的地址，第三个字段存储右子节点的地址。

注意:这个结构只能定义在二叉树上，因为一棵二叉树最多有两个子节点，而通用树可以有两个以上的子节点。与二叉树相比，通用树的节点结构会有所不同。



# 应用程序

存储自然层次数据:存储在磁盘驱动器上的文件系统，文件和文件夹都是自然层次数据的形式，以树的形式存储。

组织数据:用于组织数据以实现高效的插入、删除和搜索。例如，二叉树搜索一个元素的时间是logN。

Trie:这是一种特殊的树，用于存储字典，快速高效地进行动态拼写检查。

Heap:它是一种使用数组实现的树数据结构。它用于实现优先级队列。

- B-树和 B+树:B-树和 B+树是树型数据结构，用于实现数据库中的索引。

路由表:树数据结构用于在路由器中存储路由表中的数据。

# 大纲

树

树的类型　实现

示例

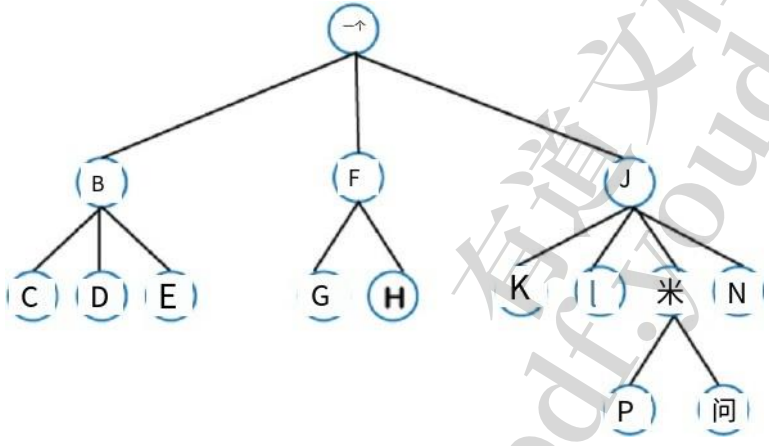
# 树的类型

一般树　二叉树

+ 二叉搜索树+ AVL 树:在单独的幻灯片

中+ 红黑树:在单独的幻灯片中

# 一般的树

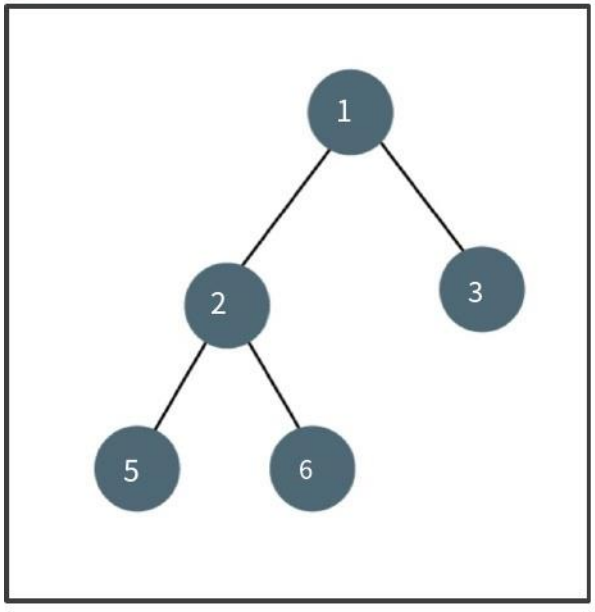



一个节点可以有 0 个或最大 n 个节点。对节点的度(一个节点可以包含的节点数)没有限制。最顶层的节点被称为根节点。父节点的子节点称为子树。

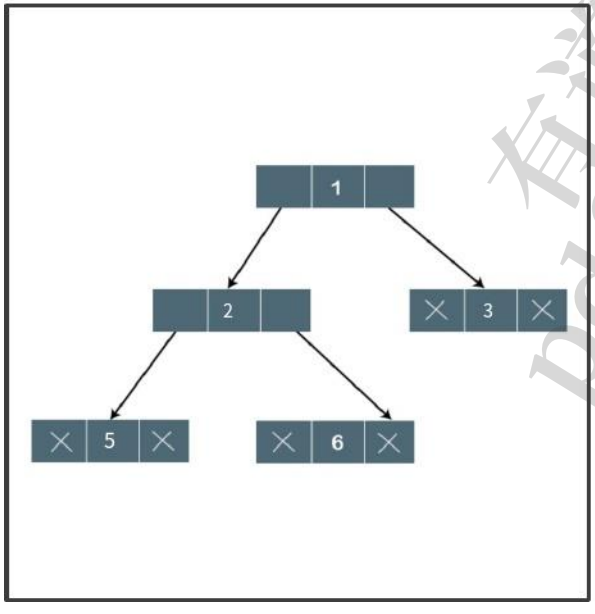
一般树中可以有 n 个子树。在一般树中，子树是无序的，因为子树中的节点不能被排序。

每棵非空树都有一条向下的边，这些边与被称为子节点的节点相连。根节点被标记为 0 级。具有相同父节点的节点被称为兄弟节点。

# 二叉树




二叉树意味着节点最多可以有两个的孩子。每个节点可以有 0 个、1 个或 2 个子节点。



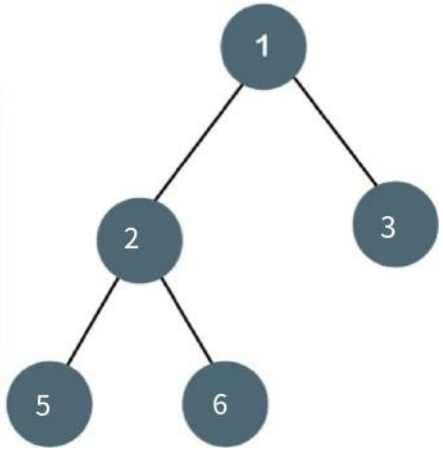

在例子中，

节点 1 包含左指针和右指针，分别指向左和分别为右节点。

节点 2 同时包含左节点和右节点。

节点 3、5、6 为叶节点;所有这些节点都包含左边和右边都是空指针。

# 二叉树的性质




- 在 i 的每一层，节点的最大数量是 $2^i$。
- 树的高度被定义为从根节点到叶节点的最长路径。
  - 高度 3 处的最大节点数为(1+2+4+8)= 15。
  - 一般来说，高度 h 处可能的最大节点数为(20 + 21 + 22+....2h) = 2h+1 -1。
- 高度 h 处可能的最小节点数等于 h+1。

- 如果节点数是最小的，那么树的高度就是最大值。
- 如果节点数是最大值，那么树的高度将是最小值。

# 二叉树的性质

假设二叉树有 n 个节点。最小高度可以计算为:  如我们所知，n = 2h+1 -1 和 n+1 = 2h+1

两边取对数，

日志 $_2(n+1) = log2(2^{h+1})$

日志 $_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

最大高度可以计算为:

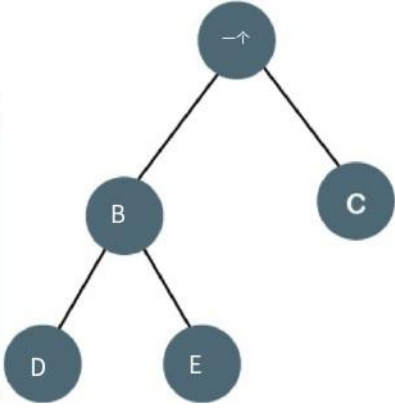
如我们所知，n = h+1

h= n-1

# 二叉树的类型

完全/适当/严格二叉树　完全二叉树　完美二叉树　平衡二叉树

# 全二叉树




每个节点包含 0 个或 2 个子节点。

叶节点的数量等于内部节点的数量加 1。 最大节点数为 2h+1 - 1。

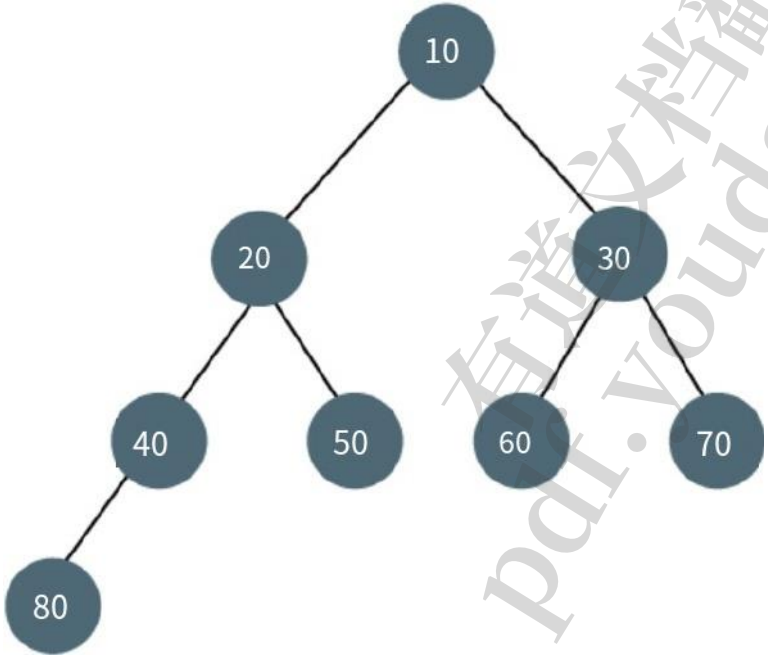
最小节点数为 2*h + 1。

满二叉树的最小高度为 $\log_2(n+1) - 1$。 满二叉树的最大高度可以计算为: n= 2*h + 1

n-1 = 2*h

h = (n-1)/2

# 完全二叉树




除了最后一层，所有节点都被完全填满了。在最后一层，所有节点必须尽可能的留下。

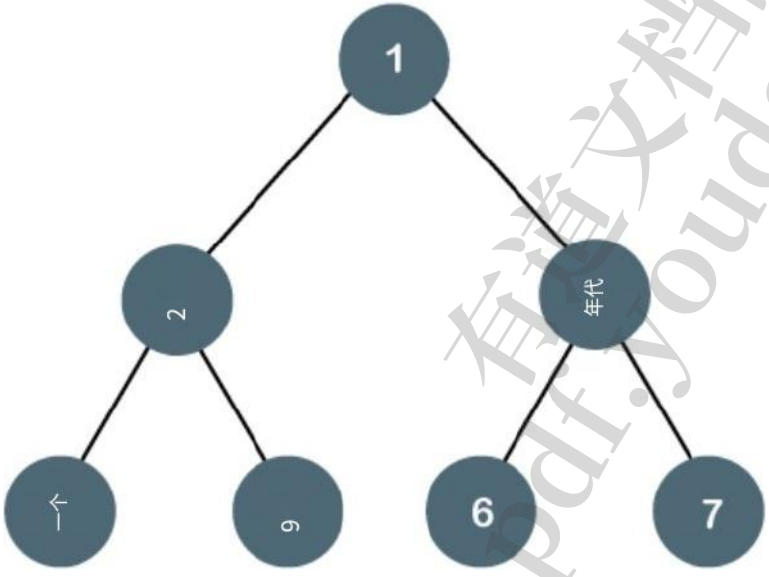
完全二叉树的最大节点数为 $2^{h+1}-1$。

完全二叉树的最小节点数为 2h。

完全二叉树的最小高度为 $\log_2(n+1)-1$。

完全二叉树的最大高度为 $\log_2(n)$。

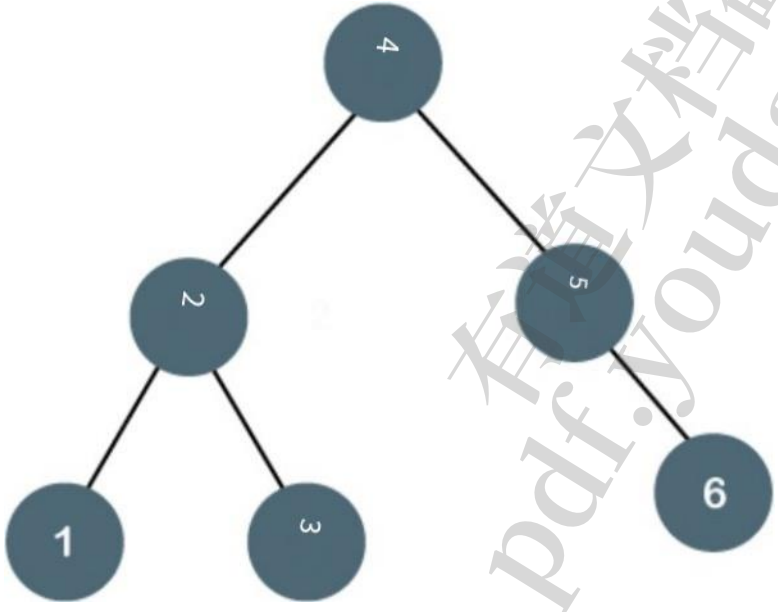
# 完美二叉树




所有内部节点都有 2 个子节点，和所有的叶节点都在同一层。

所有的完美二叉树都是完备的二叉树也有满二叉树。但反之则不成立，即。,所有完全二叉树和完全二叉树树就是完美的二叉树。

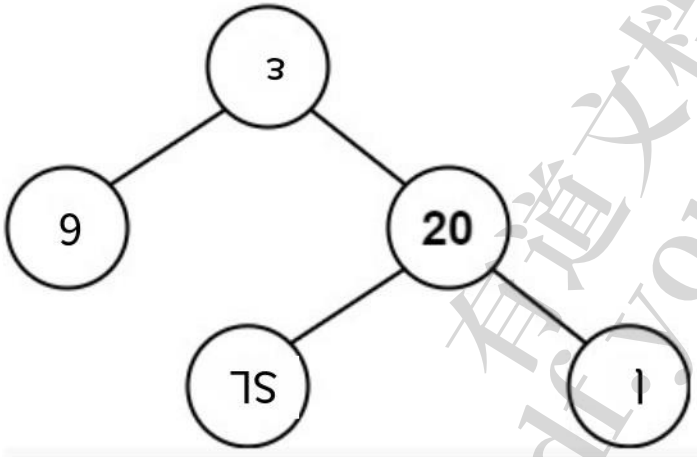
# 平衡二叉树

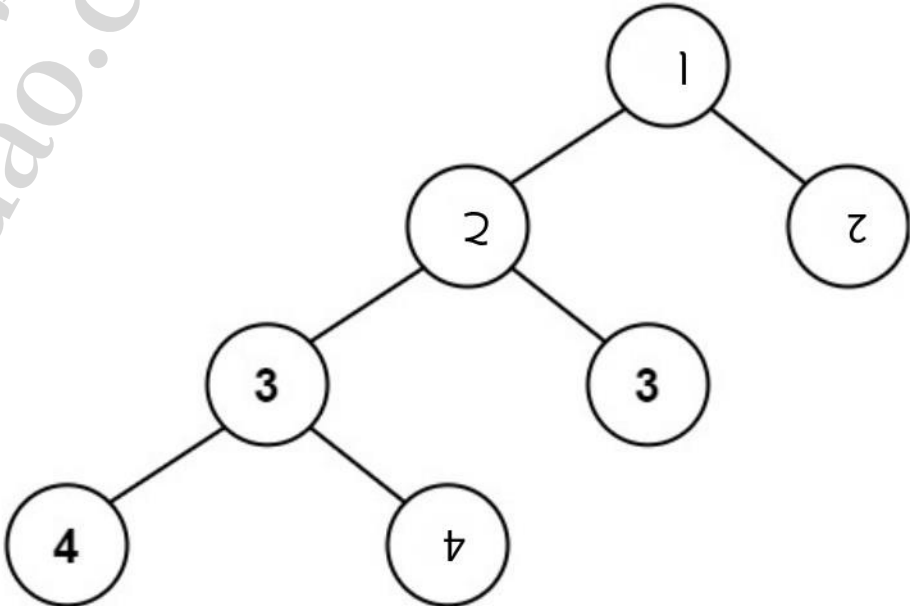



一棵二叉树，其中左右每个节点的子树在高度上相差不超过1。

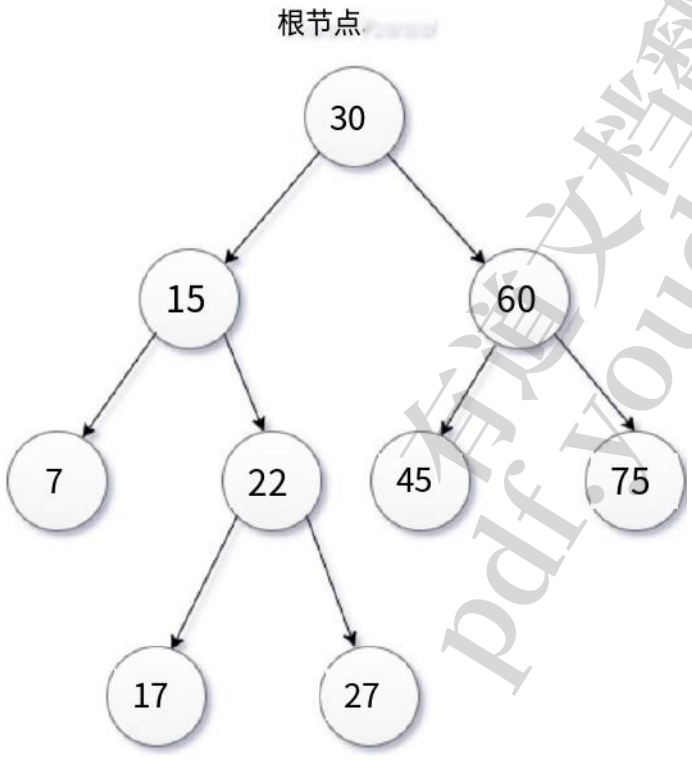
例如，AVL 和红黑树是平衡二叉树。

平衡不平衡?



# 二叉查找树




一类二叉树，其中的节点按特定的顺序排列，也称为有序二叉树。

左子树中所有节点的值都小于根节点的值。

右子树中所有节点的值大于等于根节点的值。

该规则递归应用于根节点的所有左右子树。

# BST 的优点

在 BST 中，搜索是非常高效的，因为我们在每一步都得到一个提示，关于哪一个子树包含想要的元素。

与数组和链表相比，BST 被认为是一种高效的数据结构。在搜索过程中，它每一步都删除半个子树。在二叉搜索树中搜索一个元素的时间为 $o(\log_2 n)$。在最坏情况下，搜索一个元素所需的时间是 0(n)。

与数组和链表相比，它也加快了插入和删除操作的速度。

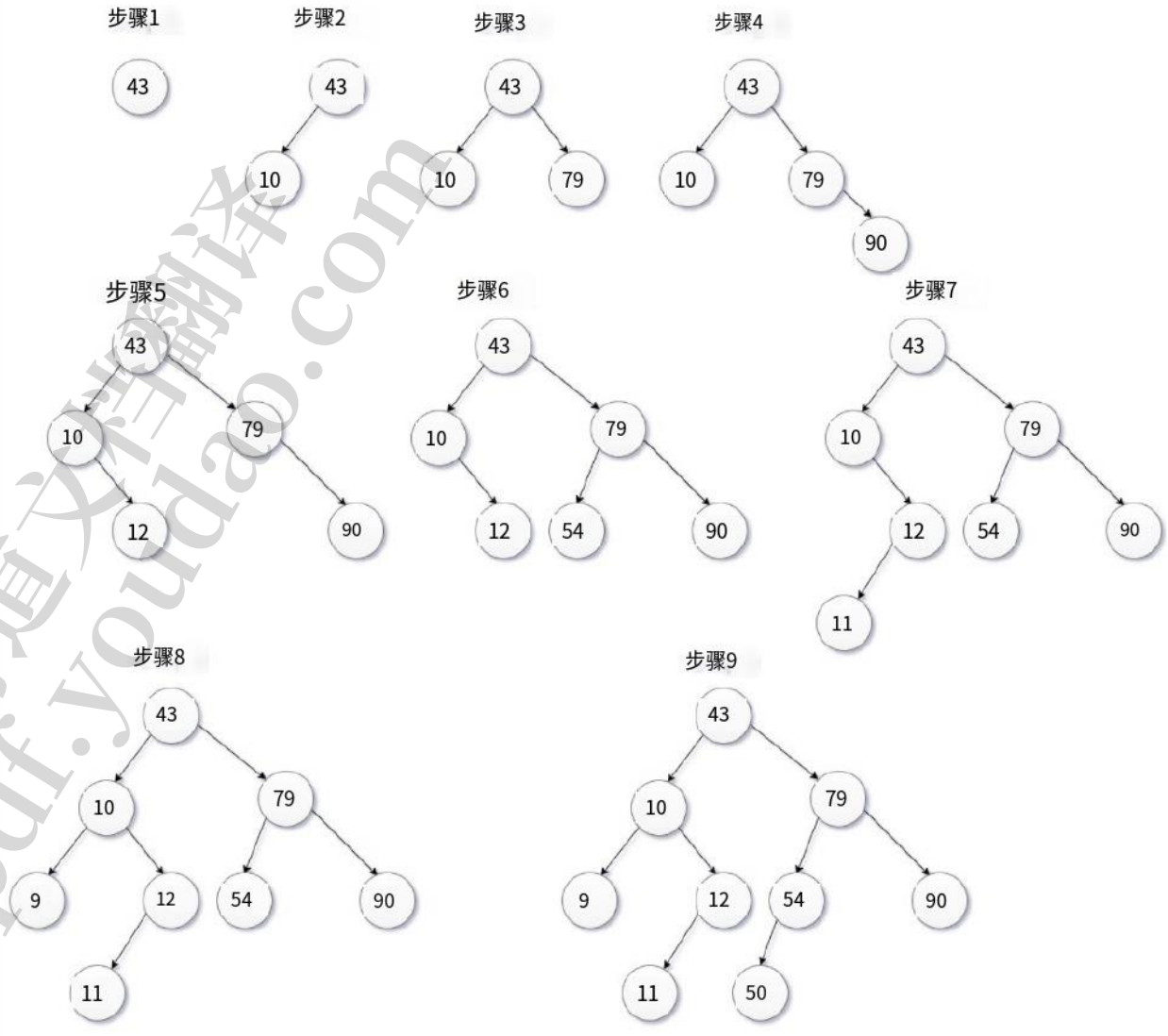
# 创建 BST

43、10、79、90、12、54、11、

在树中插入 43 作为树的根。

读取下一个元素。如果它小于根节点，将其插入为左子-的根

否则，插入为右根子树。



# 二叉 TREE 的遍历

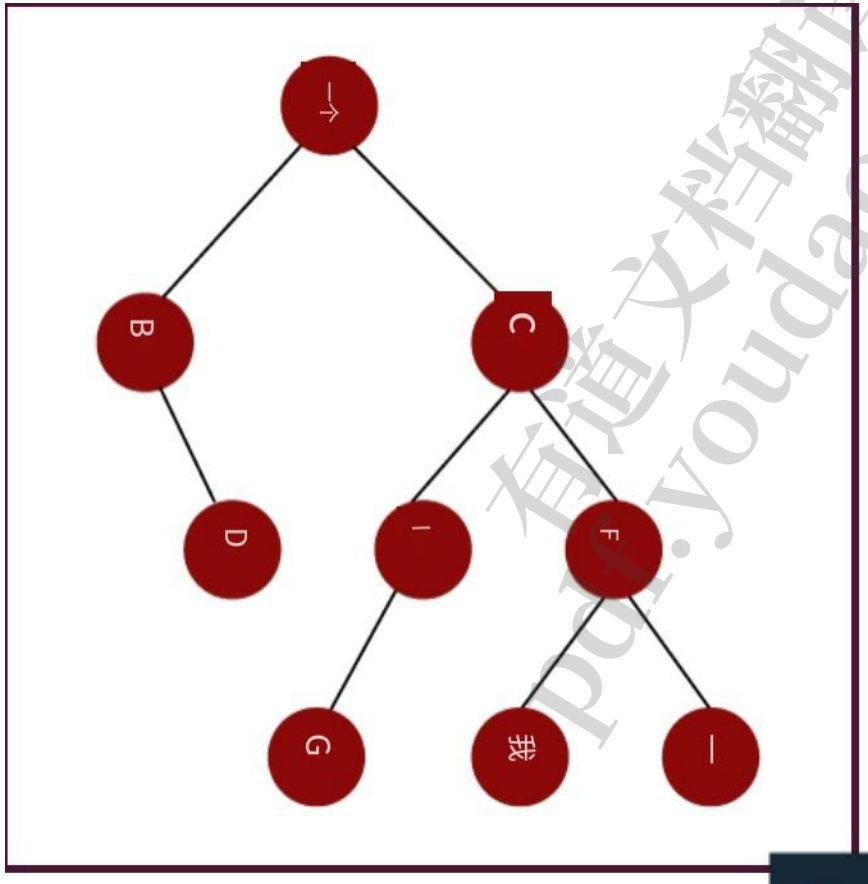
树遍历:遍历或访问树的每个节点。

栈、队列、链表等线性数据结构的遍历方式只有一种。

树有各种方式遍历/访问每个节点。

- Inorder 遍历
- 预序遍历
- 后序遍历

# 有条不紊地进行遍历




左根右

遍历根的左子树;
然后是根节点;
然后是右子树。

顺序(TREE):

第一步:重复步骤 2 到 4, TREE 零

第二步:排序(TREE->左边)

步骤 3:写入 TREE->数据

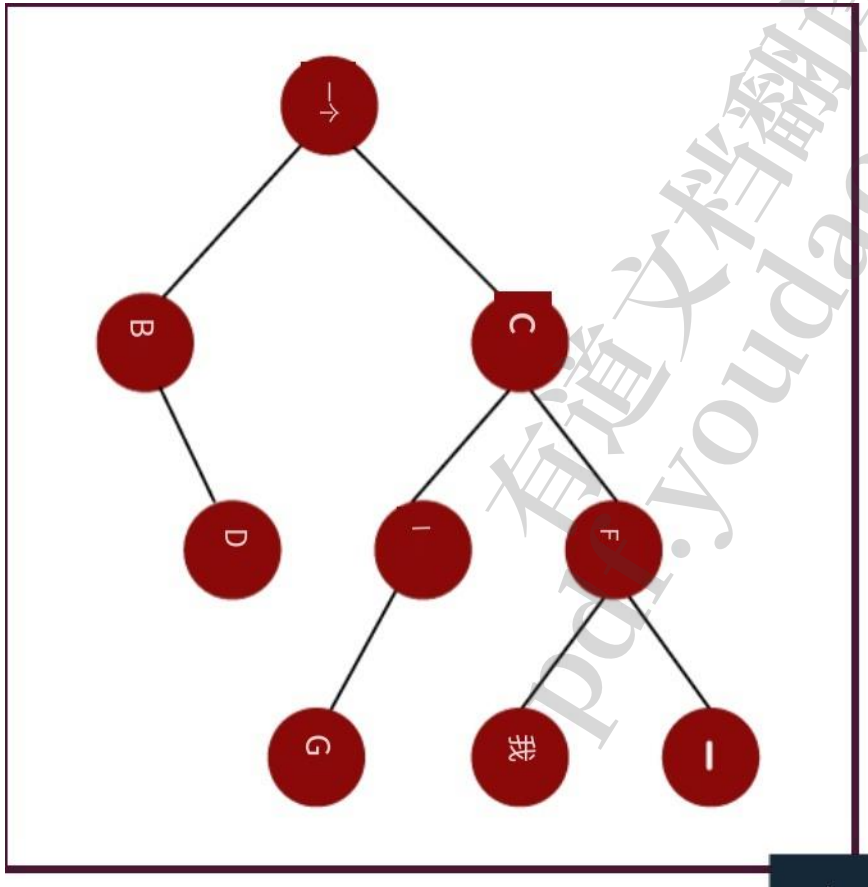
第四步:排序(TREE->右)

第五步:结束



B D G E C H F I

# 前序遍历




根左右

遍 历 树 的 根 节 点 ；

然后是左子树；

然后遍历右子树。

预订(TREE):

第一步:重复步骤 2 到 4，TREE

零

+ 步骤 2:写入 TREE->数据

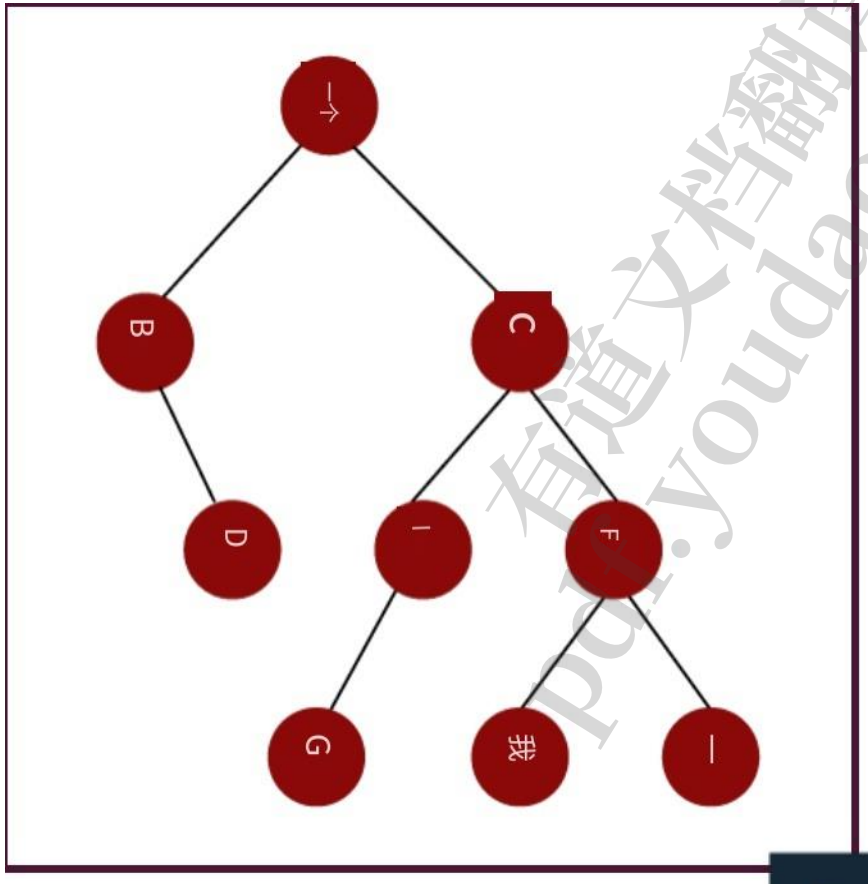
第 3 步:预订(TREE->左)

+ 步骤 4:预购(TREE->右)

第五步:结束



# 后根次序遍历




左右根

遍历根的左子树;

然后是右子树;

然后是根节点。

海报(TREE):

第一步:重复步骤 2 到 4，TREE
零

第二步:poststorder(TREE->左边)

第三步:posterder(TREE->右)

步骤 4:写入 TREE->数据

第五步:结束



B D G E C H F I

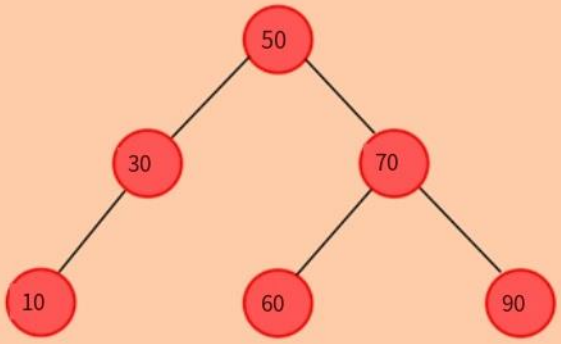
# 大纲

树

树的类型　实现

示例

# BST 实现




用三个属性定义节点类:data、left 和 right。

Left 表示节点的左子节点，right 表示节点的右子节点。 Root 表示树的根节点，并将其初始化为 null。

+ insert():将新值插入到 BST 中

如果新值小于根节点，将插入到左子树;否则，插入到右子树。+ deleteNode():从 BST 中删除一个特定的节点

如果要删除的节点是叶子节点，则该节点的父节点将指向 null。

如果我们删除 90，父节点 70 将指向 null。

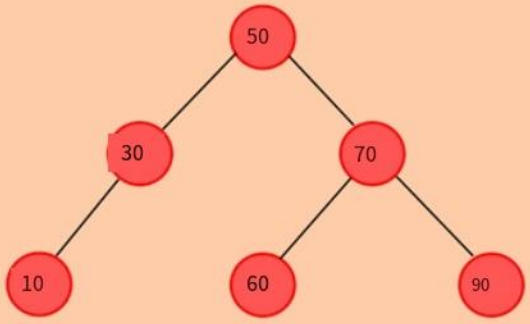
如果要删除的节点有一个子节点，子节点就会成为父节点的子节点。 如果我们删除 30，节点 10 原本是 30 的左子节点，将变成 50 的左子节点。

如果要删除的节点有两个子节点，从当前节点的右子树中查找值最小的 minNode。将当前节点替换为 minNode。

```
公共类BinarySearchTree f //表示二叉树的一个节点

        int数据;
        节点离开;
        节点对的;
        public节点(int data) {

            I/将数据赋给新节点，将左右子结点设为null this。Data = Data;这一点。Left = null;这一点。右= null;

        }
    }

    //表示二叉树的根
    public节点根;
    public BinarySearchTree() {
        根= nulL;
    }

    //insert()将向二叉搜索树中添加新节点public void insert(int data) {

        //创建一个新节点
        Node newNode = new Node(data);
        //检查树是否为空
        if(root == nulL) {

            root = newNode;返回;

        }其他{
            //当前节点指向树的根
            节点current = root, parent = null;
            而(真){
                //parent跟踪当前节点的父节点。
                Parent = current;
                i/如果data小于current的data, node会被插入到树的左边
                if(data < current.data) {
                    Current = Current .left;
                    if(current == null) {

                        的父母。left = newNode;返回;

                    }
                } else f //如果data大于当前的data, node将被插入到树的右边
                    Current = Current .right;
                    if(current == null) {
                        parent.right = newNode;
                        返回;
                    }
                }
            }
        }
    }
```

```
//minNode()将找出最小节点公共节点minNode(节点根){
    如果(根。left != nullL)否则返回
    minNode(root.left);
        返回根;
}
//deleteNode()将从二叉搜索树中删除给定的节点
    if(node == null) {
    返回null;}其他{

        //value小于节点的数据，则搜索左子树的值if(value < node.data)
            节点。左=deleteNode(节点。左值);
        Else if(value > node data) //value大于节点的数据则，搜索右子树中的值
            节点。右=deleteNode(节点)。右,值);
        else f //如果value等于node的数据，即我们已经找到了要删除的节点l //如果要删除的节点没有子节点，则将该节点设置为null If (node. conf)。
        left == nullL && node. confRight == null)
                Node = null;
            如果其他节点。left == null)(//如果要删除的节点只有一个右子节点= node.right;

            } else if(node. right)right == null) f //如果要删除的节点只有一个左子节点= node.left;

            } else{//如果要删除的节点有两个子节点
                //然后从右子树中找到最小的节点
                Node temp = minNode(Node .right);
                //交换node和temp之间的数据
                节点。Data = temp.data;
                //从右子树中删除节点重复节点
                node.right = deleteNode(node.right, temp.data);
            }
        }
        返回节点;
    }
//inorder()将对二叉搜索树执行序遍历
public void inorderTraversal(Node Node) {
    //检查树是否为空
    if(root == null) {
        system . out。println(“树是空的”);返回;
    }其他{
        如果节点。Left != null)
            inorderTraversal(node.left);
        System.out.print(节点。Data + “” );
        如果节点。对!= null)
            inorderTraversal(node.right);
    }
}
```

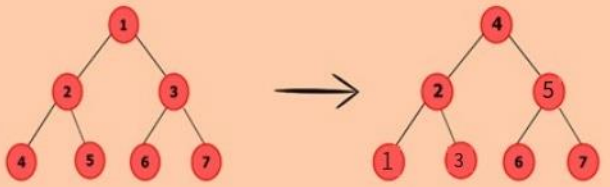
```
104     public static void main(String[] args) {
105
106         BinarySearchTree bt = new BinarySearchTree();
107         //向二叉树中添加节点
108         bt.insert(50);
109         bt.insert(30)              ;
110
111         bt.insert(70bt.insert(60   ;
                                       ;
112                                    ;
113         bt.insert(10)bt.insert(90);
114
115
116         system . out。println(“插入后的二叉搜索树:”);//显示二叉树
117         bt.inorderTraversal(bt.root);
118
119         节点deletedNode = null;
120         //删除无子节点90
121         deletedNode = bt.deleteNode(bt.root, 90);
122         system . out。println(“\ n二叉搜索树删除节点90后:”)                    ;
123         bt.inorderTraversal(bt.root);
124
125         //删除有一个子节点的节点30
126         deletedNode = bt.deleteNode(bt.root, 30);
127         system . out。println(“\ n二叉搜索树删除节点30后:”);
128         bt.inorderTraversal(bt.root);
129
130         //删除有两个子节点的节点50
131         deletedNode = bt.deleteNode(bt.root, 50);
132         system . out。println(“\ n二叉搜索树删除节点50后:”);
133         bt.inorderTraversal(bt.root);
134     }
135 }
```




输出:

```
插入后的二叉搜索树:
10 30 50 60 70 90
删除节点90后的二叉搜索树:
10 30 50 60 70
删除节点30后的二叉搜索树:
10 50 60 70

二叉搜索树删除节点50后:10 60 70
```

# 将二叉 TREE 转换为 BST




convertBTBST
()：

通过调用 convertBTtoArray()将二叉树转换为对应的数组。

对结果数组进行升序排序。

通过调用 createBST()将数组转换为二叉搜索树。

calculateSize()计算树中存在的节点数。

convertBTtoArray()将二叉树转换为数组表示。

createBST()选择排序后的树数组的中间节点作为根节点，创建一棵相应的二叉搜索树。树数组被分成两部分:[0,mid-1]和[mid+1, end]。递归地从每个数组中找到中间节点，分别创建左子树和右子树。

Inorder()以按顺序的方式显示节点。，左子节点，根节点，右子节点。

```
进口java.util.Arrays;

公共类ConvertBTtoBST{

    //表示二叉树公共静态类的一个节点node {

        int数据;
        节点离开;
        节点对的;
        public节点(int data) {

            这一点。数据=数据;
            这一点。Left = null;      //为新节点分配数据，将左右子节点设为null this。Right = null;


        }
    }

    //表示二叉树的根
    public节点根;
     int[]树数组;
     Int index = 0;
    c .ConvertBTtoBST函数
        Root = null;
    }

    //convertBTBST()将二叉树转换为二叉搜索树
    公共节点convertBTBST(节点节点){
        //变量树大小将保存树的大小
        树大小= calculateSize(节点);树数组=新int[树大
        小];

        //转换二叉树为数组
        convertBTtoArray(node);

        //对树数组排序
        数组排序(树数组);

        //转换数组为二叉搜索树Node d = createBST(0,
        treeArray.)Length - 1);返回d;

    }
    //calculateSize()将计算树的大小公共int calculateSize(Node
    Node) {
        Int size = 0;
        If (node == null)

        返回0;else {size =calculateSize(node.left) +calculateSize(node.right) + 1;返回大小;


        }
    }
```

```
53
54      I/convertBTtoArray()将把给定的二叉树转换为相应的数组表示。
55          //检查树是否为空
56          if(root == nulL) {
57
58              system . out。println(“树是空的”);返回;
59          }其他{
60              如果节点。Left != null)
61                  convertBTtoArray (node.left);
62              //将二叉树的节点添加到treeArray中
63              treeArray[index] = node.data;
64              指数+ +;
65              如果节点。对!= null)
66                  convertBTtoArray (node.right);
67          }
68      }
69      //createBST()将数组转换为二叉搜索树
70      公共节点createBST(int start, int end)
71          //它将避免溢出
72          If (start > end) {
73              返回null;
74          }
75
76          //变量将存储数组的中间元素，并使其成为二叉搜索树的根int mid = (start + end) / 2;Node Node = new
77          Node(treeArray[mid]);
78
79          //构造左子树
80          节点。left =createBST(start, mid - 1);
81
82          //构造右子树
83          node.right = createBST(mid + 1, end);
84
85          返回节点;
86      }
87      //inorder()将对二叉搜索树执行序遍历
88      public void inorderTraversal(Node节点)
89          //检查树是否为空
90          if(root == null) {
91
92              system . out。println(“树是空的”);返回;
93          }其他{
94              如果节点。left != nulL)
95                  inorderTraversal(node.left);
96              System.out.print(node.data          + "  ");
97              如果节点。对!= null)
98                  inorderTraversal(node.right);
99          }
100     }
101
102
```

```
103     public static void main(String[] args) {
104
105         ConvertBTtoBST bt =新的ConvertBTtoBST();
106
107         //添加节点到二叉树bt.root = new
108         Node(1);bt.root.left = new
109         Node(2);bt.root.right =新节点
110         (3);bt.root.left.left =新节点(4);bt.root.left.right
111          =新节点(5);bt.root.right.left =新节点
112         (6);bt.root.right.right =新节点(7);
113
114
115         //显示给定二叉树System.out。println("按顺序表示二叉树:");
116
117         bt.inorderTraversal(bt.root);
118
119         //将二叉树转换为相应的二叉搜索树Node bst =
120         bt.convertBTBST(bt.root);
121
122         //显示相应的二叉搜索树System.out。println("\结果二叉搜索树的nInorder表示形式:");
123
124         bt.inorderTraversal(bst);
125     }
126 }
```

输出:

```
二叉树的顺序表示:
4251637
结果二叉搜索树的顺序表示:
1               234567
```

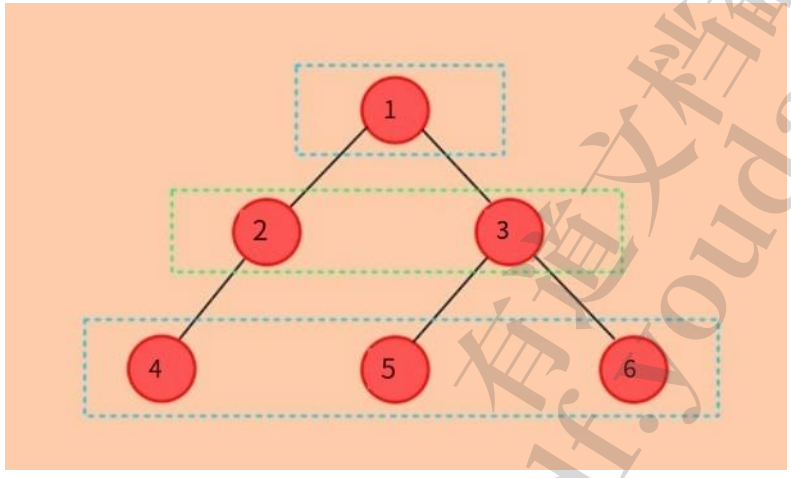
# 大纲

树

树的类型　实现

示例

# 二叉 TREE 奇数层结点和偶数层结点和之差




差值= (L1 + L 3 + L5) - (L2 + L4) OddLevelSum = 1 + 4 + 5 + 6 = 16 EvenLevelSum = 2 + 3 = 5 差值= |16 - 5| = 11

差分():

使用队列逐级遍历二叉树。

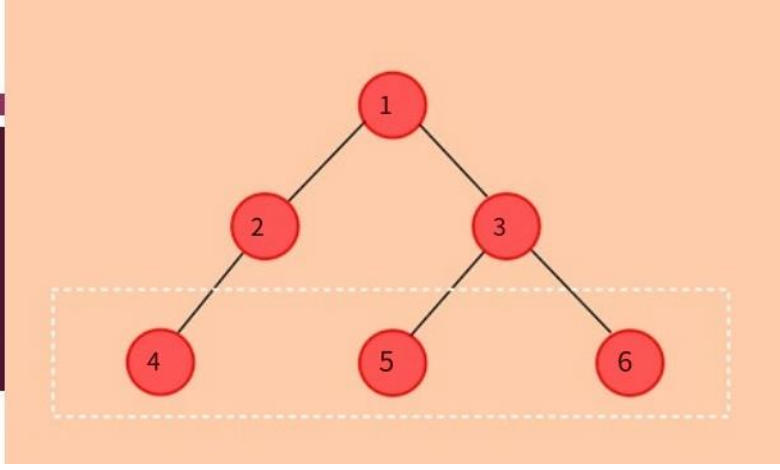
使用变量 currentLevel 跟踪当前级别。

如果 currentLevel 能被 2 整除，则将 currentLevel 中所有节点的值加到变量偶数级中。否则，将所有节点的值添加到变量 oddLevel 中。

通过从 oddLevel 减去偶数级的值来计算差值。

```
//difference()将计算二叉树奇数级和偶数级之和的差值
        int oddLevel = 0, evenLevel = 0, diffoddEven = 0;
        //变量nodesInLevel记录每层的节点数
        int nodesInLevel= 0;
        //变量currentLevel记录二叉树中的level
        int currentLevel = 0;
        //Queue将用于跟踪树的节点级别
        Queue<Node> Queue = new LinkedList<Node>();
        //检查root是否为空
        if(root == null) {
                system . out。println(“树是空的”);返回0;
        } else {
                //将根节点添加到队列中，因为它代表第一级
                queue.add(根);
                currentLevel + +;
                while(queue.size() != 0) {
                        //变量nodesInLevel将保存队列的大小，即队列中元素的数量
                        nodesInLevel= queue.size();
                        while(nodesInLevel > 0) {
                                节点当前= queue.remove();
                                //检查currentLevel是否为偶数。
                                currentLevel% 2 == 0
                                        //如果level为偶数，将节点的值添加到变量evenLevel中
                                        evenLevel += current.data;
                                其他的
                                        //如果level为奇数，则向变量oddLevel添加节点的oddLevel+=
                                        current.data;

                                //添加左子节点到队列中
                                如果电流。left != nulL)
                                        queue.add (current.left);
                                //添加右子队列
                                如果电流。对!= null)
                                queue.add
                                (current.right);nodesInLevel——;
                        }
                        currentLevel + +;
                °

                //计算奇数级和偶数级的差值。abs(奇数级-偶数级);


        return diffoddEven;
}
```

# 所有的树叶都在同一个级别?




isSameLevel():检查给定二叉树的所有叶子是否在同一层次 它检查根是否为 null，这意味着树是空的。

如果树不为空，则遍历树，检查左子节点和右子节点是否为 null。

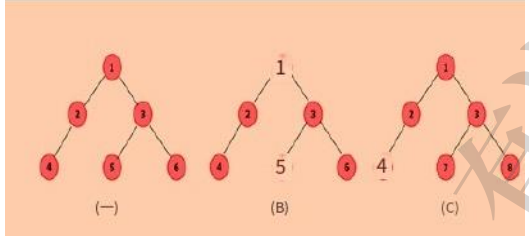
CurrentLevel 将跟踪正在遍历的当前层。

当遇到第一个叶节点时，将 currentLevel 的值存储在变量 level 中。

递归遍历所有关卡，检查后续叶子节点。如果所有叶子的 currentLevel 等于存储在 level 中的值，那么所有叶子都在同一个 level 上。

```
公共类LeafLevel f
    //表示二叉树公共静态类的一个节点node {
        int数据;
        节点离开;
        节点对的;
        public节点(int data) {
            //为新节点分配数据，将左右子节点设为null this。Left = null;这一点。
            右= null;
        }
    }
    //表示二叉树的根
    public节点根;
    //它将存储第一次遇到的叶子public static int level = 0;
    public LeafLevel() {
        Root = null;
    }
    I/isSameLevel()将检查二叉树的所有叶子是否在同一个级别或notl公共布尔isSameLevel(Node temp,
    int currentLevel) if(root == null) //检查树是否为空System.out。println(“树是空的”);返回true;
        }其他{
            if(temp == null) //检查节点是否为空返回true;
            如果(temp。Left == null && temp.right == null) {
                if(level == 0) f //如果遇到第一个叶子，设置level为当前的level level =currentLevel;返回true;
                } else //检查其他叶子是否与第一个叶子的return处于相同的level (level ==currentLevel);
            }
            //递归地检查左子树和右子树的叶子节点。
            返回(isSameLevel);左，currentLevel+ 1)&&isSameLevel(temp。对，currentLevel + 1));
        }
    }
    public static void main (String[] args) {
        叶级bt =新的叶级();
        //向二叉树中添加节点
        bt.root = new Node(1);
        bt.root.left =新节点(2);
        bt.root.right =新节点(3);
        bt.root.left.left = new Node(4);
        bt.root.right.left =新节点(5);bt.root.right.right =
        新节点(6);
        //检查给定二叉树的所有叶子是否都在同一层if(bt. issamelevel (bt. issamelevel);根,1))
            system . out。println(“所有1l叶子都在同一层”);
        其他的
            system . out。println(“所有叶子不在同一层”);
    }
}
```

# 两棵树是相同的吗?




areIdenticalTrees():检查两棵树是否相同? 如果两棵树的根节点为null，则它们是相同的。 如果只有一棵树的根节点为null，则树不相同，返回false。

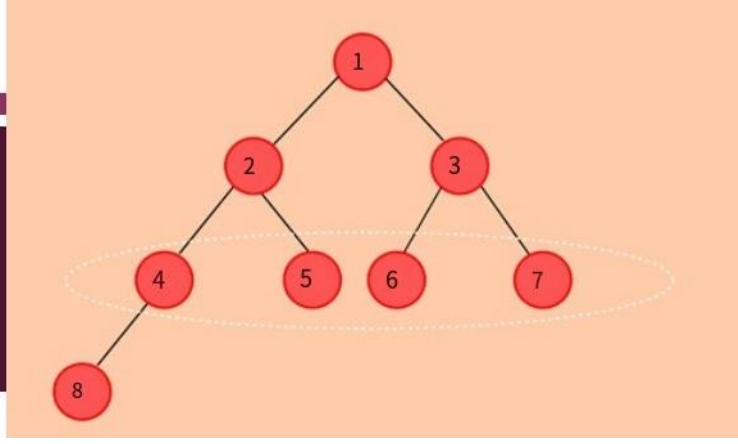
如果所有树的根节点都不为空，

检查两个节点的数据是否相等， 然后递归检查的左子树和右子树

一棵树是否与另一棵树相同。

```
公共类identialtrees f //表示二叉树公共静态类的节点node {

        int数据:
        节点离开;
        节点对的;
        public Node(int data) [this.
        data]Data = Data;这一点。
        Left = null;这一点。右= null;
        }
    }
    public节点根;//表示二叉树的根
    公共IdenticalTrees () {
        Root = null;
    }
    //areIdenticalTrees(节点root1，节点root2) f if(root1 == null && root2 == null)
        //检查两个树是否都是空的

            返回true;

        //如果只有一棵树的根为空，则树不相同，如果(root1 == null && root2 == null)返回true则返回false;

        //如果两棵树都不是空的，检查节点的数据是否相等//重复左子树和右子树的步骤If (root1 != null &&
        root2 != null) f .

            ((root1返回。Data == root2.data) &&
                (areIdenticalTrees(root1.left, root2.left)) &&
                (areIdenticalTrees(root1.right, root2.right)));
        }
        返回错误;
    }
    public static void main(String[] args) {
        同一棵树bt1 =新同一棵树();//向第一个二叉树添加节点
        bt1。root = new Node(1);
        bt1.root。left = new Node(2);
        bt1.root。right = new Node(3);
        bt1.root.left。left = new Node(4);
        bt1.root.right。left = new Node(5);
        bt1.root.right。right = new Node(6);
        同一棵树bt2 =新同一棵树();//向第二个二叉树添加节点
        bt2。root = new Node(1);
        bt2.root。left = new Node(2);
        bt2.root。right = new Node(3);
        bt2.root.left。left = new Node(4);
        bt2.root.right。left = new Node(5);
        bt2.root.right。right = new Node(6);
        //显示两个树是否相同
        if(areIdenticalTrees(bt1.root, bt2.root))
            system . out。println(“两个二叉树都是相同的”);
        其他的
            system . out。println(“两个二叉树都不相同”);
    }
}
```

# 最大宽度
# 二叉树的最大宽度




二叉树的最大宽度为 4，用白色椭圆表示。

findMaximumWidth():找出给定二叉树的最大宽度

变量 MaxWidth 存储任意层中存在的最大节点数。

该队列用于逐层遍历二叉树。

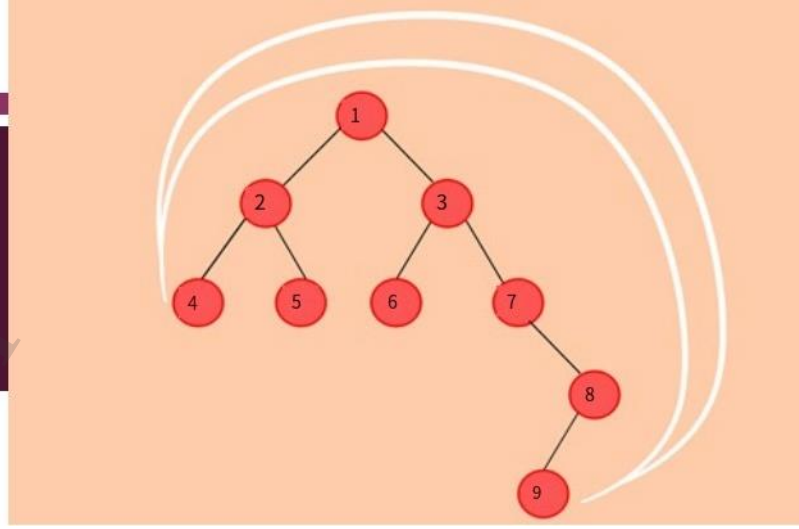
检查根是否为 null，这意味着树是空的。

如果不为空，则将根节点添加到队列中。变量 nodesInLevel 跟踪每一层的节点数量。 如果 nodesInLevel> 0，从队列前面移除该节点，并将其左、右子节点加入队列。对于第一次迭代，节点 1 将被删除，其子节点 2 和 3 将被添加到队列中。在第二次迭代中，节点 2 将被删除，其子节点 4 和 5 将被添加到队列中，以此类推。

存储 max(MaxWidth，nodesInLevel)。在任何给定的时间点，它代表最大的节点数。 这个过程一直持续到树的所有层都被遍历。

```
进口java.util.LinkedList;进口java.util.Queue;公共类
BinaryTree f //表示二叉树公共静态类的节点node {

        int数据;
        节点离开;
        节点对的;
        公共节点(int数据){

            //将数据分配给新节点，设置左66 this。Data = Data;
            这一点。Left = null;这一点。右= null;

        }
    }

    //表示二叉树的根public节点根;
    public BinaryTree() {
        根= null;
    }
    //findMaximumwidth()将找出给定二叉树的最大宽度public int findMaximumwidth() {
        int maxWidth = 0;
        //变量nodesInLevel记录每层的节点数
        int nodesInLevel= 0;
        //queue将用于跟踪树的节点级别queue <Node> queue = new
        LinkedList<Node>();//检查root是否为空，则width将为o
        if(root == null) {
            system . out。println(“树是空的”);返回0;
        } else {
                //添加根节点到队列，因为它代表第一级队列。Add (root);
            while(queue.size() != 0) {
                //变量nodesInLevel将保存队列的大小，即队列中元素的数量
                nodesInLevel= queue.size();
                //maxWidth将保持最大宽度。
                l/如果nodesInLevel大于maxWidth，则maxWidth将保存nodesInLevel的值maxWidth= Math。max(maxWidth，nodesInLevel);

                //如果变量nodesInLevel包含多个节点l/，那么，对于每个节点，我们将添加该节点的左和右子节点到队列中
                while(nodesInLevel > 0) {
                    节点当前= queue.remove();
                    如果电流。Left != null)
                    queue.add (current.left);如果电流。right
                    != nulL)

                    queue.add (current.right);nodesInLevel

                }
            }
        }
        返回maxWidth;
    }

    public static void main(String[] args){二叉树bt =新二叉树();
        //向二叉树中添加节点
        bt.root =新节点(1);
        bt.root.left =新节点(2);
        bt.root.right =新节点(3);
        bt.root.left.left =新节点(4);
        bt.root.left.right =新节点(5);bt.root.right.left =新节点
        (6);bt.root.right.right =新节点(7);

        bt.root.left.left.left = new Node(8);
        //显示给定树的最大宽度
        system . out。println(“二叉树的最大宽度:”+ bt.findMaximumwidth( );
    }
}
```

# 最大距离
# 在二叉树中




nodesAtMaxDistance():找出在最大距离上存在的节点 calculateSize():计算树中存在的节点数。

convertBTtoArray():通过遍历树并将元素添加到 treeArray 中，将二叉树转换为其数组表示。

getDistance():计算给定节点到根节点的距离。

LowestCommonAncestor():找出两个节点的最小共同祖先。 FindDistance():计算两个节点之间的距离。

```
进口java.util.ArrayList;公共类
MaxDistance {
    公共静态类Node {
        int数据;

        节点离开;节点右:公共节点(int
        data) {this.;Data = Data;这一点
        。Left = null;这一点。右= null;

        }
    }

    //表示二叉树的根
     public节点根;
    int [] treeArray;
     Int index = 0;
    public MaxDistance() {
        根= null;
     }


    //calculateSize()将计算树公共int的大小\calculateSize(节点节点){int大小=
    0;If (node == null)返回0;其他{


            size =calculateSize(node.left) +calculateSize(node.right) + 1;返回大小;

        }
     }
    //convertBTtoArray()将把二叉树转换成它的数组形式

        if(root == null) f //检查树是否为空System.out. if(root ==
        null)println(“树是空的”);返回;
        } else {if(节点。左!= null)convertBTtoArray(node.left);//
        将二叉树节点添加到treeArray中
            treeArray[index] = node.data;
            指数+ +;
            如果节点。= null) convertBTtoArray(node.right);
        }
     }
    //getDistance()将查找根节点和指定节点之间的距离
        If (temp != null){
            Int x = 0;if ((temp.data == n1) II (x = getDistance(temp. data == n1)左，n1)) > o .
                    ll (x = getDistance(temp.right, n1)) > 0) {
                //x将存储temp和节点n1之间的边数，返回x + 1;
            }
            返回0;
        }
        返回    0;
    }
```

```
I/lowestCommonAncestor()将找出节点node1和node2l公共节点的最低公共祖先节点lowestCommonAncestor(Node temp, int
node1, int node2) f
        If (temp != null) {

                //如果root等于node1或node2中的任意一个，则返回root If (temp.data == node1 || temp.data
                == node2) {return temp;


                //遍历左、右子树节点Left, node1, node2);Node right
                =lowestCommonAncestor(临时变量)。右，node1, node2);}

                //如果节点temp有一个节点(node1或node2)作为左子节点，一个节点(node1或node2)作为右子节点
                //然后返回节点temp作为最低公共祖先
                If(左!= null &&右!= null){返回temp;
                }

                //如果节点node1和node2在左子树If (left != null){返回left;


                  //如果节点node1和node2在右子树中
                If (right != null) {
                        返回正确的
                1
        }
        返回null;

//findDistance()将查找两个给定节点之间的距离
findDistance(int node1, int node2) {
        //计算第一个节点到根节点的距离
        int d1 =getDistance(root, node1) - 1;
        //计算第二个节点到根节点的距离
        int d2 =getDistance(root, node2) - 1;
        //计算两个节点的最小公共祖先节点
        节点祖先= lowestCommonAncestor(root, node1, node2);
        //如果最低公共祖先不是根那么，减去2 *(根到祖先的距离)返回(d1 + d2) - 2 * d3;
        int d3 = getDistance(root，祖宗.data) - 1;
```

```
    //nodesAtMaxDistance()将显示处于最大距离的节点
        int maxDistance = 0, distance = 0;
        <Integer> arr = new数组列表<>();
        //初始化树数组
        树大小= calculateSize(节点);树数组=新int[树大小];

        //转换二叉树为它的数组表示
        convertBTtoArray(node);
        //计算二叉树中所有节点之间的距离，并将最大距离存储在变量maxdistance中
        for(int i = 0; i < treeArray.length; i++) {
            For (int j = i;j < treeArray.length;j + +) {
                (树数组[i]，树数组[j]);
                //如果距离大于maxDistance，则如果(distance >maxDistance)，则maxDistance将保存距离的值

                    maxDistance=距离;arr.clear ();
                    I/在treeArray中添加I和j位置的节点
                    arr.add (treeArray[我]);
                    arr.add(treeArray[il);
                } else if(distance ==最大距离){
                    I/如果超过一对节点在最大距离，那么将所有对添加到treeArrayl
                    arr.add(treeArray[i]);
                    arr.add(treeArray[j]);
                }
            }
        }
        //显示所有距离最大的节点对。println("距离最大的节点:");For (int I = 0;I <
        arr.size();I = I + 2) {
            System.out.println("( " + arr.get(i)   "," + arr.get(i + 1) + " )");
        }
    }

    public static void main(String[] args) {
        最大距离bt =新的最大距离();

        //向二叉树中添加节点bt.root = new Node(1);

        bt.root.left = new Node(2);
        bt.root.right =新节点(3);
        bt.root.left.left =新节点(4);
        bt.root.left.right =新节点(5);
        bt.root.right.left =新节点(6);
        bt.root.right.right =新节点(7);
        bt.root.right.right =新节点(8);
        bt.root.right.right. left =新节点(9);
        //找出所有距离最大的节点对
        bt.nodesAtMaxDistance(bt.root);
    }
}
```

# 哈夫曼编码

+ 以一种令人惊讶的方式使用二叉树来压缩数据的一个算法。赫夫曼码，取自 1952 年大卫赫夫曼的发现。

数据压缩在很多情况下都很重要。

一个例子是通过互联网发送数据，特别是通过拨号连接，

传输可能需要很长时间。

一个普通的未压缩文本文件中的每个字符都由 1 字节(ASCII 码)或 2 字节(Unicode 码，设计用于所有语言)表示。

每个字符都需要相同数量的比特。

| 字符 | 小数 | 二进制 |
|---|---|---|
| A | 65 | 01000000 |
| B | 66 | 01000001 |
| C | 67 | 01000010 |
| … | … | … |
| X | 88 | 01011000 |
| Y | 89 | 01011001 |
| Z | 90 | 01011010 |

# 霍夫曼密码

频率表

| 字符 | 数 |
| --- | --- |
| 一个 | 2 |
| E | 2 |
| I | 2 |
| 年代 | 2 |
| T | 1 |
| U | 1 |
| Y | 2 |
| 空间 | 4 |
| 换行 | 1 |

霍夫曼编码

| 字符 | 代码 |
| --- | --- |
| 一个 | 010 |
| E | 1111 |
| I | 110 |
| 年代 | 10 |
| T | 0110 |
| U | 01111 |
| Y | 1110 |
| 空间 | 00 |
| 换行 | 01110 |

减少代表最常用的比特数字符。

E 是最常用的字母，所以少用是合理的位尽量对其进行编码。

另一方面，Z 很少使用，所以使用大号位数就没那么差了。

规则:任何代码都不能作为任何其他代码的前缀。

例如，如果 E 是 01,X 是 01011000，那么任何人都可以解码 01011000 就不知道最初的 01 表示的是 E 还是 X 的开头。

# 创建哈夫曼树

为消息中使用的每个字符创建一个节点对象。

每个节点有两个数据项:字符和该字符在消息中的频率。 为这些节点中的每一个创建一个树对象。该节点成为树的根。 将这些树插入优先队列中，

按频率排序，最小的频率具有最高的优先级。

继续重复下面的步骤，直到队列中只剩下一棵树。

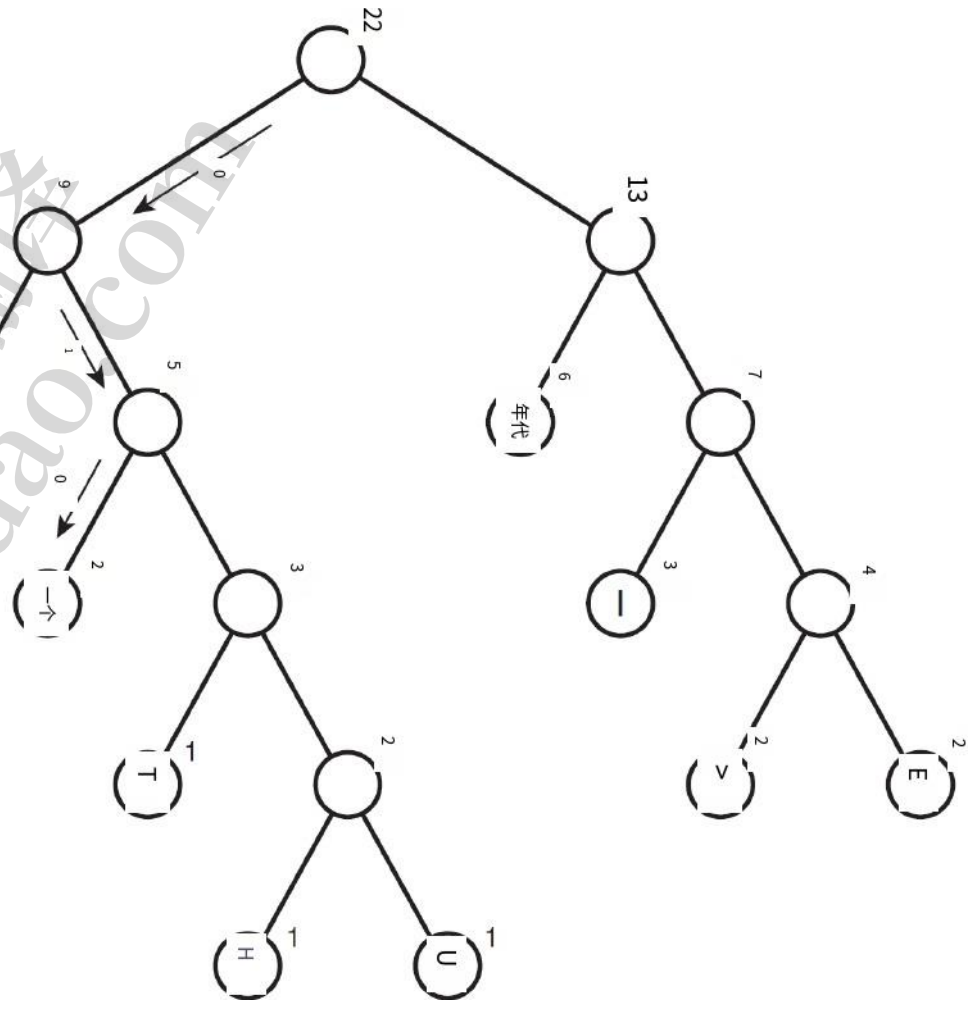
从优先队列中删除两棵树，并使它们成为新节点的子节点。 新节点的频率是子节点频率之和。

将这个新的三节点树插入到优先队列中。

苏西说这很容易

## 频率表

| 字符 | 数 |
| --- | --- |
| 一个 | 2 |
| F | 2 |
| I | 3 |
| 年代 | 6 |
| H | 1 |
| U | 1 |
| V | 2 |
| 空间 | 4 |
| 换行 | 1 |



谢谢