

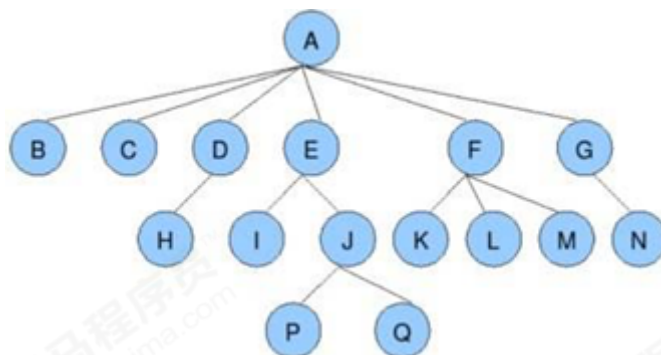
# 一、二叉树入门

之前我们实现的符号表中，不难看出，符号表的增删查操作，随着元素个数 $N$ 的增多，其耗时也是线性增多的，时间复杂度都是 $O(n)$ ，为了提高运算效率，接下来我们学习树这种数据结构。

## 1.1 树的基本定义

树是我们计算机中非常重要的一种数据结构，同时使用树这种数据结构，可以描述现实生活中的很多事物，例如家谱、单位的组织架构、等等。

树是由 $n$  ( $n \geq 1$ ) 个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。



树具有以下特点：

1. 每个结点有零个或多个子结点；
2. 没有父结点的结点为根结点；
3. 每一个非根结点只有一个父结点；
4. 每个结点及其后代结点整体上可以看做是一棵树，称为当前结点的父结点的一个子树；

## 1.2 树的相关术语

**结点的度：**

一个结点含有的子树的个数称为该结点的度；

**叶结点：**

度为0的结点称为叶结点，也可以叫做终端结点

**分支结点：**

度不为0的结点称为分支结点，也可以叫做非终端结点

**结点的层次：**

从根结点开始，根结点的层次为1，根的直接后继层次为2，以此类推

**结点的层序编号：**

将树中的结点，按照从上层到下层，同层从左到右的次序排成一个线性序列，把他们编成连续的自然数。

### 树的度：

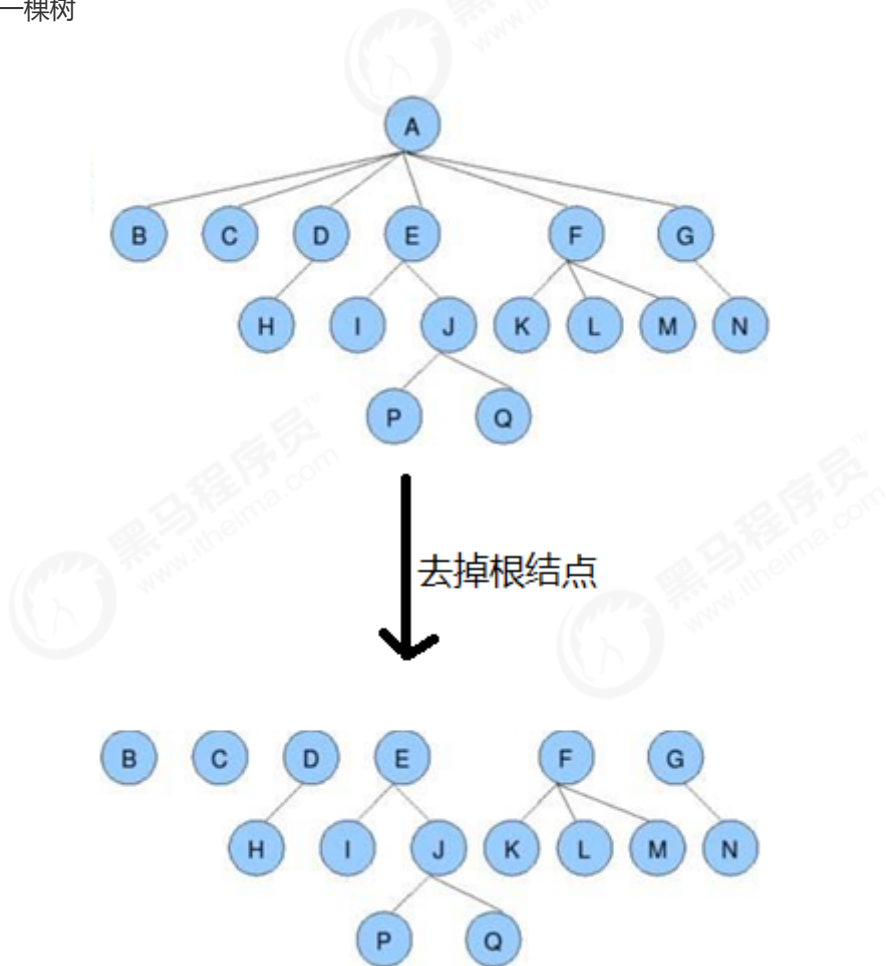
树中所有结点的度的最大值

### 树的高度(深度)：

树中结点的最大层次

### 森林：

$m$  ( $m \geq 0$ ) 个互不相交的树的集合，将一颗非空树的根结点删去，树就变成一个森林；给森林增加一个统一的根结点，森林就变成一棵树



### 孩子结点：

一个结点的直接后继结点称为该结点的孩子结点

### 双亲结点(父结点)：

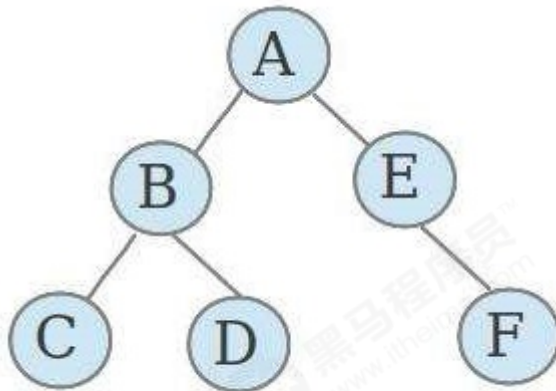
一个结点的直接前驱称为该结点的双亲结点

### 兄弟结点：

同一双亲结点的孩子结点间互称兄弟结点

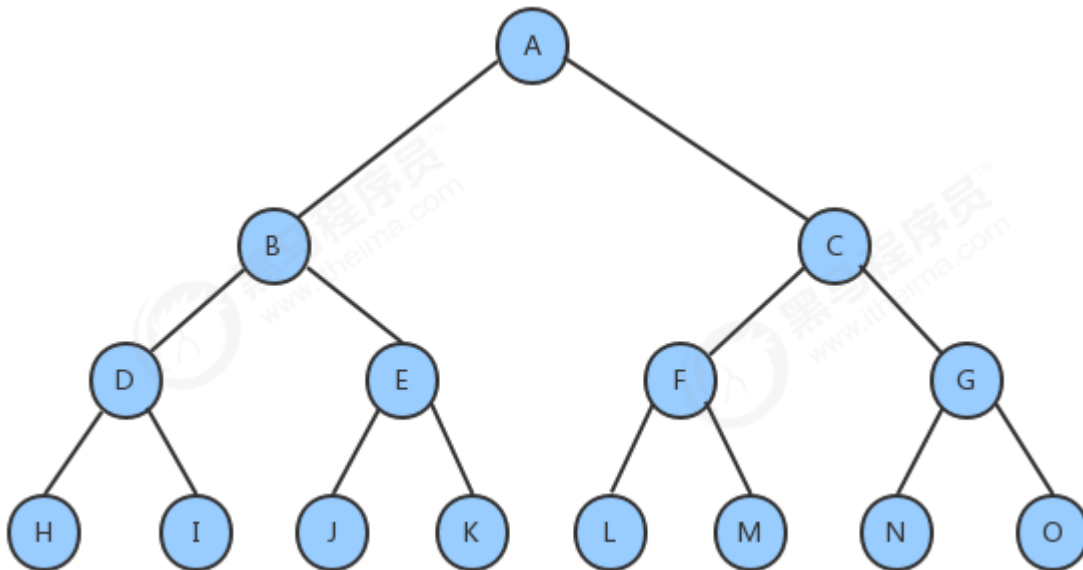
## 1.3 二叉树的基本定义

二叉树就是度不超过2的树(每个结点最多有两个子结点)



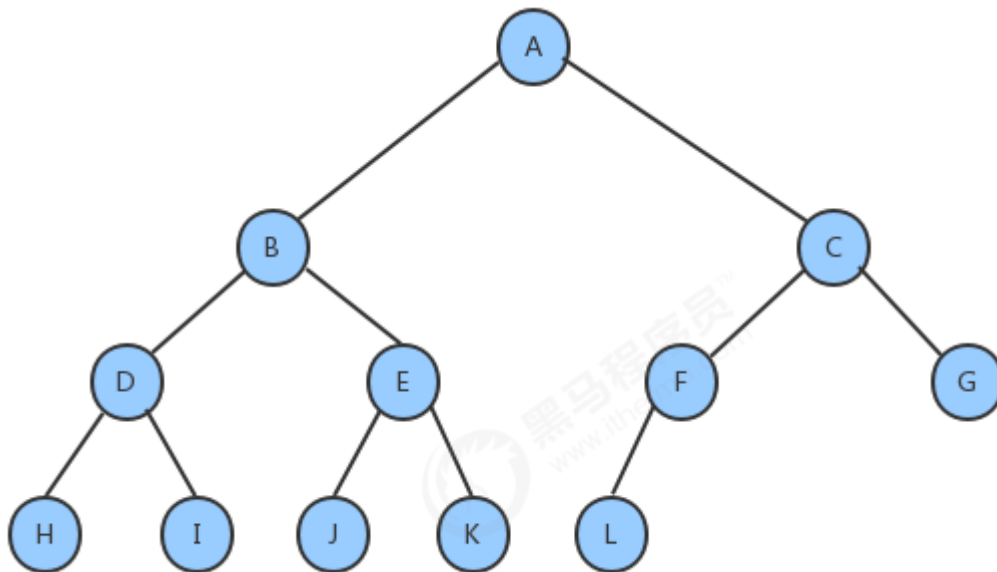
**满二叉树：**

一个二叉树，如果每一个层的结点树都达到最大值，则这个二叉树就是满二叉树。



**完全二叉树：**

叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树



## 1.4 二叉查找树的创建

### 1.4.1 二叉树的结点类

根据对图的观察，我们发现二叉树其实就是由一个一个的结点及其之间的关系组成的，按照面向对象的思想，我们设计一个结点类来描述结点这个事物。

结点类API设计：

类名	Node<Key,Value>
构造方法	Node(Key key, Value value, Node left, Node right)：创建Node对象
成员变量	1.public Node left:记录左子结点 2.public Node right:记录右子结点 3.public Key key:存储键 4.public Value value:存储值

代码实现：

```
1 private class Node<Key,Value>{
2     //存储键
3     public Key key;
4     //存储值
5     private Value value;
6     //记录左子结点
7     public Node left;
8     //记录右子结点
9     public Node right;
10
11     public Node(Key key, Value value, Node left, Node right) {
12         this.key = key;
13         this.value = value;
```

```

14         this.left = left;
15         this.right = right;
16     }
17 }
    
```

## 1.4.2 二叉查找树API设计

类名	BinaryTree, Value value>
构造方法	BinaryTree() : 创建BinaryTree对象
成员变量	1.private Node root:记录根结点 2.private int N:记录树中元素的个数
成员方法	1. public void put(Key key, Value value):向树中插入一个键值对 2.private Node put(Node x, Key key, Value val) : 给指定树x上, 添加键一个键值对, 并返回添加后的新树 3.public Value get(Key key):根据key, 从树中找出对应的值 4.private Value get(Node x, Key key):从指定的树x中, 找出key对应的值 5.public void delete(Key key):根据key, 删除树中对应的键值对 6.private Node delete(Node x, Key key):删除指定树x上的键为key的键值对, 并返回删除后的新树 7.public int size():获取树中元素的个数

## 1.4.3 二叉查找树实现

**插入方法put实现思想：**

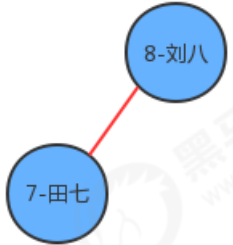
- 1.如果当前树中没有任何一个结点，则直接把新结点当做根结点使用
- 2.如果当前树不为空，则从根结点开始：
  - 2.1如果新结点的key小于当前结点的key，则继续找当前结点的左子结点；
  - 2.2如果新结点的key大于当前结点的key，则继续找当前结点的右子结点；
  - 2.3如果新结点的key等于当前结点的key，则树中已经存在这样的结点，替换该结点的value值即可。



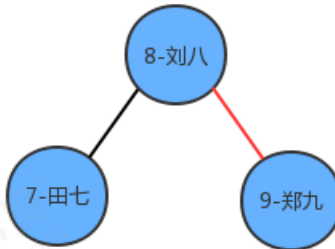
存储第一个元素：8-刘八



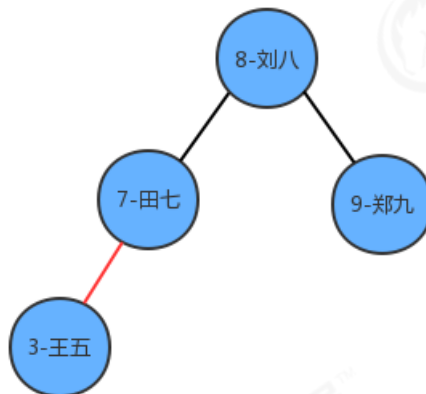
存储第二个元素：7-田七



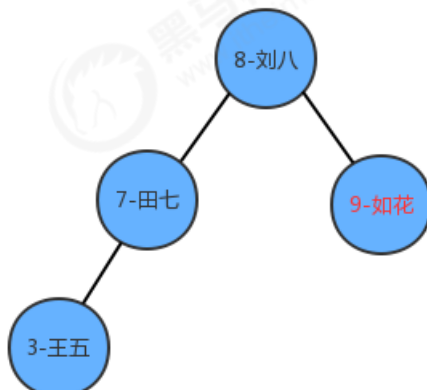
存储第三个元素：9-郑九



存储第四个元素：3-王五



存储第五个元素：9-如花



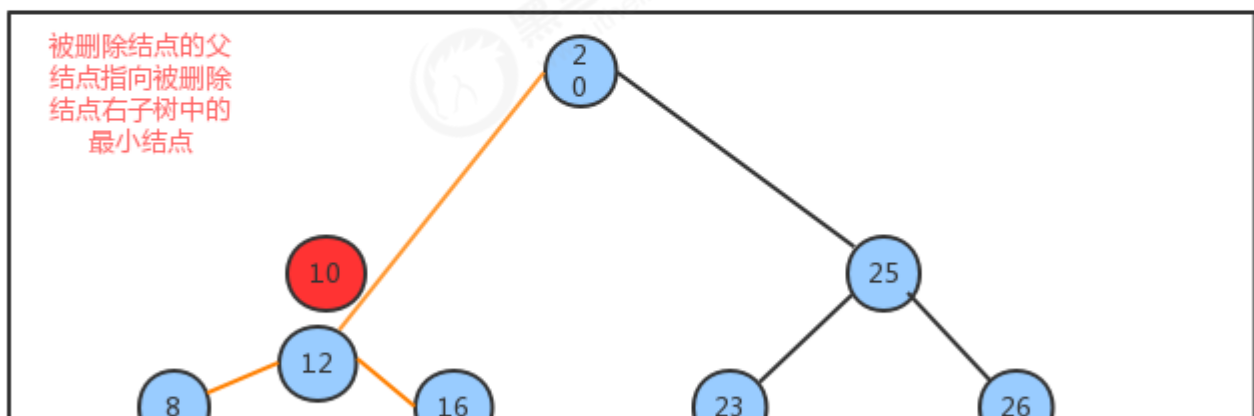
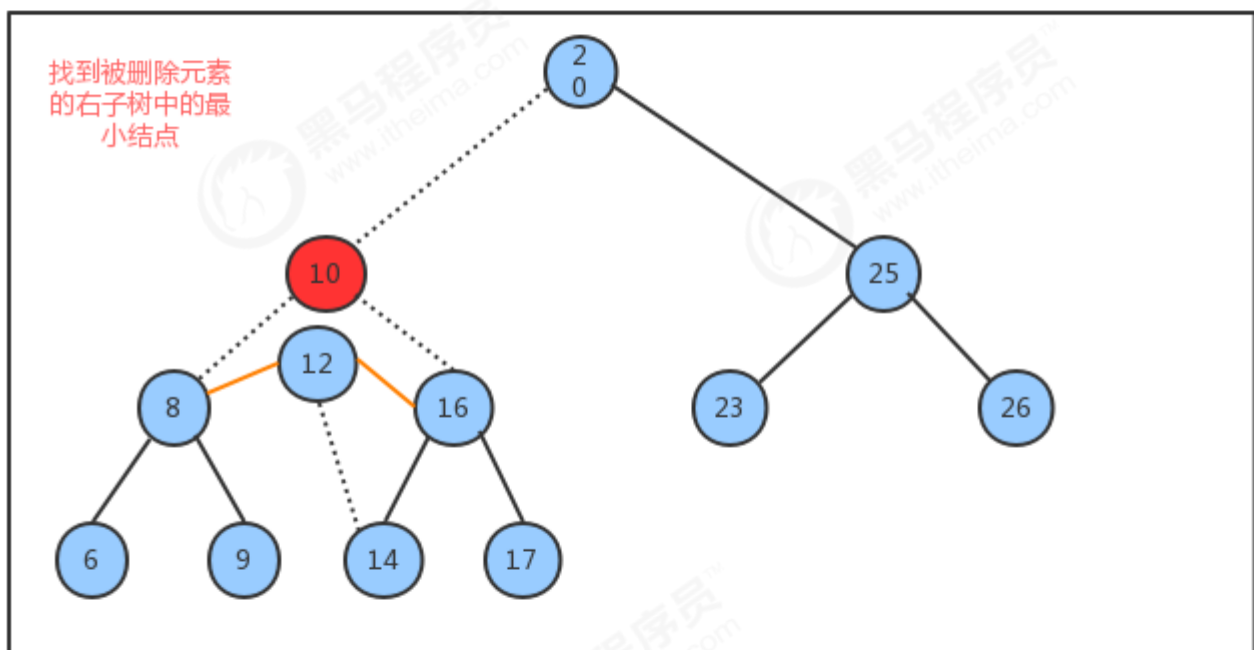
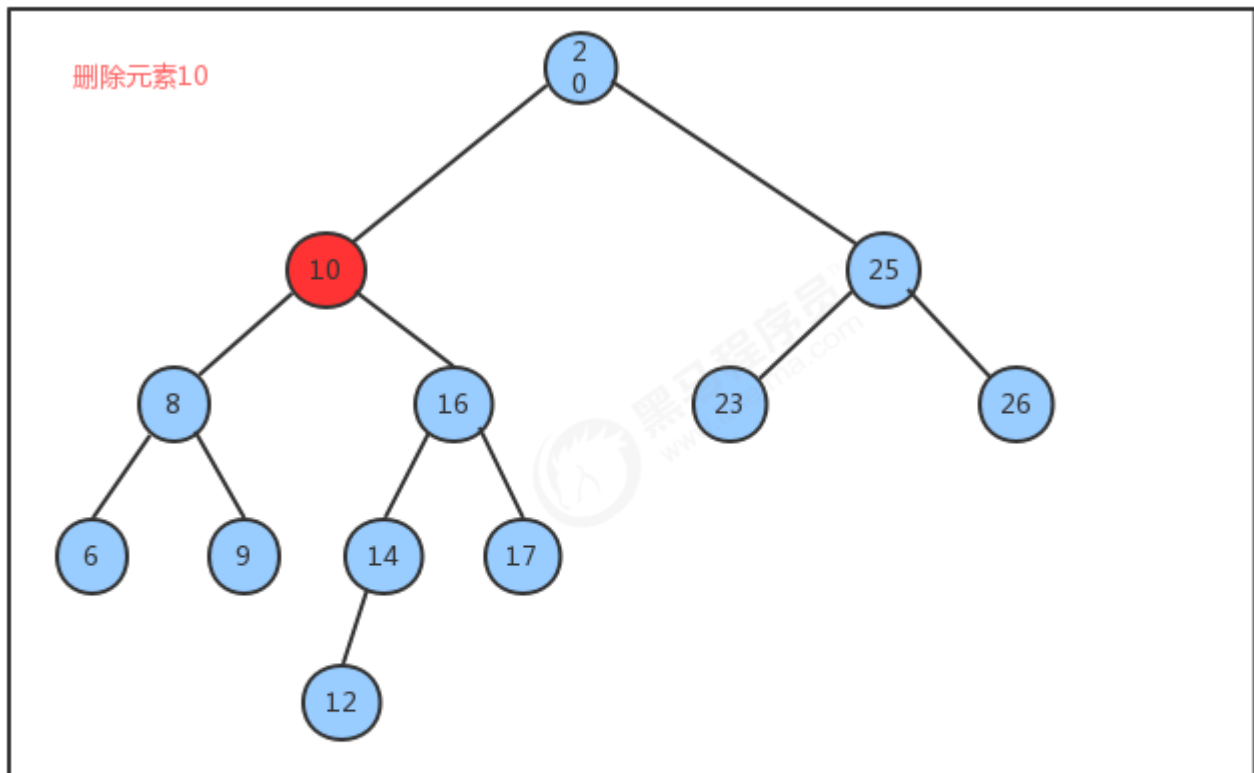
### 查询方法get实现思想：

从根节点开始：

- 1.如果要查询的key小于当前结点的key，则继续找当前结点的左子结点；
- 2.如果要查询的key大于当前结点的key，则继续找当前结点的右子结点；
- 3.如果要查询的key等于当前结点的key，则树中返回当前结点的value。

### 删除方法delete实现思想：

- 1.找到被删除结点；
- 2.找到被删除结点右子树中的最小结点minNode
- 3.删除右子树中的最小结点
- 4.让被删除结点的左子树称为最小结点minNode的左子树，让被删除结点的右子树称为最小结点minNode的右子树
- 5.让被删除结点的父节点指向最小结点minNode







代码：

```
1 //二叉树代码
2 public class BinaryTree<Key extends Comparable<Key>, Value> {
3     //记录根结点
4     private Node root;
5     //记录树中元素的个数
6     private int N;
7
8     //获取树中元素的个数
9     public int size() {
10         return N;
11     }
12
13     //向树中添加元素key-value
14     public void put(Key key, Value value) {
15         root = put(root, key, value);
16     }
17
18     //向指定的树x中添加key-value,并返回添加元素后新的树
19     private Node put(Node x, Key key, Value value) {
20         if (x == null) {
21             //个数+1
22             N++;
23             return new Node(key, value, null, null);
24         }
25         int cmp = key.compareTo(x.key);
26         if (cmp > 0) {
27             //新结点的key大于当前结点的key, 继续找当前结点的右子结点
28             x.right = put(x.right, key, value);
29         } else if (cmp < 0) {
30             //新结点的key小于当前结点的key, 继续找当前结点的左子结点
31             x.left = put(x.left, key, value);
32         } else {
33             //新结点的key等于当前结点的key, 把当前结点的value进行替换
34             x.value = value;
35         }
36         return x;
37     }
38
39     //查询树中指定key对应的value
40     public Value get(Key key) {
41         return get(root, key);
42     }
43
44     //从指定的树x中, 查找key对应的值
```



```
45     public Value get(Node x, Key key) {
46         if (x == null) {
47             return null;
48         }
49         int cmp = key.compareTo(x.key);
50         if (cmp > 0) {
51             //如果要查询的key大于当前结点的key，则继续找当前结点的右子结点；
52             return get(x.right, key);
53         } else if (cmp < 0) {
54             //如果要查询的key小于当前结点的key，则继续找当前结点的左子结点；
55             return get(x.left, key);
56         } else {
57             //如果要查询的key等于当前结点的key，则树中返回当前结点的value。
58             return x.value;
59         }
60     }
61
62     //删除树中key对应的value
63     public void delete(Key key) {
64         root = delete(root, key);
65     }
66
67     //删除指定树x中的key对应的value，并返回删除后的新树
68     public Node delete(Node x, Key key) {
69         if (x == null) {
70             return null;
71         }
72
73         int cmp = key.compareTo(x.key);
74         if (cmp > 0) {
75             //新结点的key大于当前结点的key，继续找当前结点的右子结点
76             x.right = delete(x.right, key);
77         } else if (cmp < 0) {
78             //新结点的key小于当前结点的key，继续找当前结点的左子结点
79             x.left = delete(x.left, key);
80         } else {
81             //新结点的key等于当前结点的key，当前x就是要删除的结点
82             //1.如果当前结点的右子树不存在，则直接返回当前结点的左子结点
83             if (x.right == null) {
84                 return x.left;
85             }
86             //2.如果当前结点的左子树不存在，则直接返回当前结点的右子结点
87             if (x.left == null) {
88                 return x.right;
89             }
90             //3.当前结点的左右子树都存在
91             //3.1找到右子树中最小的结点
92             Node minNode = x.right;
93             while (minNode.left != null) {
94                 minNode = minNode.left;
95             }
96             //3.2删除右子树中最小的结点
97
98             Node n = x.right;
```



```
98         while (n.left != null) {
99             if (n.left.left == null) {
100                 n.left = null;
101             } else {
102                 n = n.left;
103             }
104         }
105
106         //3.3让被删除结点的左子树称为最小结点minNode的左子树，让被删除结点的右子树称为最小结点
minNode的右子树
107         minNode.left = x.left;
108         minNode.right = x.right;
109         //3.4让被删除结点的父节点指向最小结点minNode
110         x = minNode;
111         //个数-1
112         N--;
113     }
114     return x;
115 }
116
117 private class Node {
118     //存储键
119     public Key key;
120     //存储值
121     private Value value;
122     //记录左子结点
123     public Node left;
124     //记录右子结点
125     public Node right;
126
127     public Node(Key key, Value value, Node left, Node right) {
128         this.key = key;
129         this.value = value;
130         this.left = left;
131         this.right = right;
132     }
133 }
134 }
135
136 //测试代码
137 public class Test {
138     public static void main(String[] args) throws Exception {
139         BinaryTree<Integer, String> bt = new BinaryTree<>();
140         bt.put(4, "二哈");
141         bt.put(1, "张三");
142         bt.put(3, "李四");
143         bt.put(5, "王五");
144         System.out.println(bt.size());
145         bt.put(1, "老三");
146         System.out.println(bt.get(1));
147         System.out.println(bt.size());
148         bt.delete(1);
149
150         System.out.println(bt.size());
```

```
150     }  
151 }
```

## 1.4.4 二叉查找树其他便捷方法

### 1.4.4.1 查找二叉树中最小的键

在某些情况下，我们需要查找出树中存储所有元素的键的最小值，比如我们的树中存储的是学生的排名和姓名数据，那么需要查找出排名最低是多少名？这里我们设计如下两个方法来完成：

<b>public Key min()</b>	<b>找出树中最小的键</b>
private Node min(Node x)	找出指定树x中，最小键所在的结点

```
1  //找出整个树中最小的键  
2  public Key min(){  
3      return min(root).key;  
4  }  
5  
6  //找出指定树x中最小的键所在的结点  
7  private Node min(Node x){  
8      if (x.left!=null){  
9          return min(x.left);  
10     }else{  
11         return x;  
12     }  
13 }
```

### 1.4.4.2 查找二叉树中最大的键

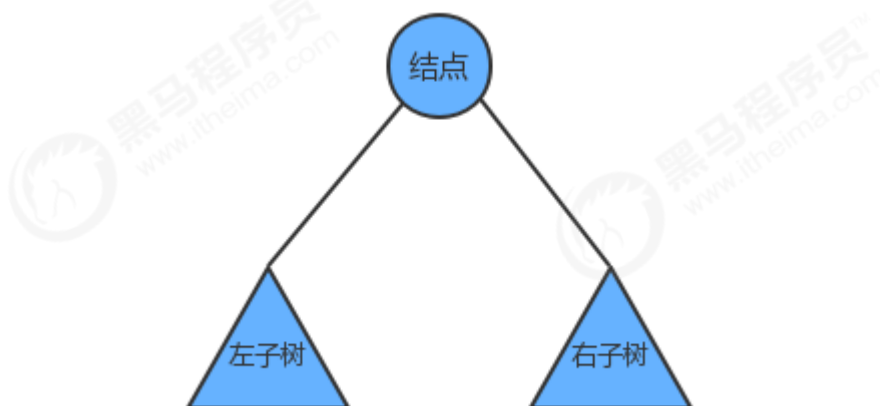
在某些情况下，我们需要查找出树中存储所有元素的键的最大值，比如比如我们的树中存储的是学生的成绩和学生的姓名，那么需要查找出最高的分数是多少？这里我们同样设计两个方法来完成：

<b>public Key max()</b>	<b>找出树中最大的键</b>
public Node max(Node x)	找出指定树x中，最大键所在的结点

```
1 //找出整个树中最大的键
2 public Key max(){
3     return max(root).key;
4 }
5
6 //找出指定树x中最大键所在的结点
7 public Node max(Node x){
8     if (x.right!=null){
9         return max(x.right);
10    }else{
11        return x;
12    }
13 }
```

## 1.5 二叉树的基础遍历

很多情况下，我们可能需要像遍历数组数组一样，遍历树，从而拿出树中存储的每一个元素，由于树状结构和线性结构不一样，它没有办法从头开始依次向后遍历，所以存在如何遍历，也就是按照什么样的搜索路径进行遍历的问题。



我们把树简单的画作上图中的样子，由一个根节点、一个左子树、一个右子树组成，那么按照根节点什么时候被访问，我们可以把二叉树的遍历分为以下三种方式：

1.前序遍历；

先访问根结点，然后再访问左子树，最后访问右子树

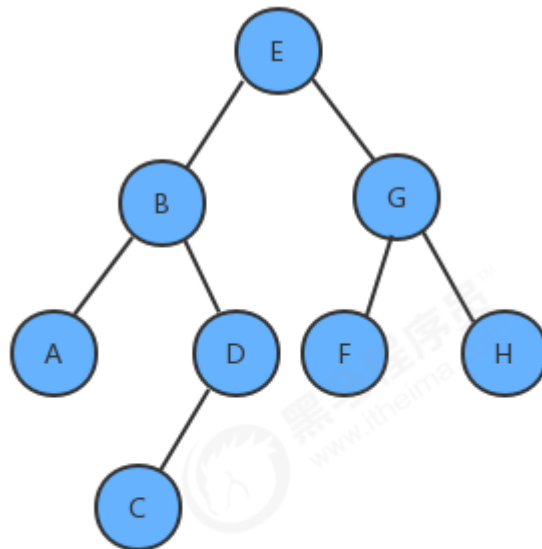
2.中序遍历；

先访问左子树，中间访问根节点，最后访问右子树

3.后序遍历；

先访问左子树，再访问右子树，最后访问根节点

如果我们分别对下面的树使用三种遍历方式进行遍历，得到的结果如下：



前序遍历结果：EBADCGFH

中序遍历结果：ABCDEFGH

后序遍历结果：ACDBFHGE

### 1.5.1 前序遍历

我们在4.4中创建的树上，添加前序遍历的API：

```
public Queue<Key> preErgodic()：使用前序遍历，获取整个树中的所有键
```

```
private void preErgodic(Node x,Queue<Key> keys)：使用前序遍历，把指定树x中的所有键放入到keys队列中
```

实现过程中，我们通过前序遍历，把每个结点的键取出，放入到队列中返回即可。

**实现步骤：**

- 1.把当前结点的key放入到队列中;
- 2.找到当前结点的左子树，如果不为空，递归遍历左子树
- 3.找到当前结点的右子树，如果不为空，递归遍历右子树

**代码：**

```
1 //使用前序遍历，获取整个树中的所有键
2 public Queue<Key> preErgodic(){
3     Queue<Key> keys = new Queue<>();
4     preErgodic(root,keys);
5     return keys;
6 }
7
8 //使用前序遍历，把指定树x中的所有键放入到keys队列中
9 private void preErgodic(Node x,Queue<Key> keys){
10     if (x==null){
11         return;
12     }
```



```
13 //1.把当前结点的key放入到队列中;
14 keys.enqueue(x.key);
15 //2.找到当前结点的左子树，如果不为空，递归遍历左子树
16 if (x.left!=null){
17     preErgodic(x.left,keys);
18 }
19 //3.找到当前结点的右子树，如果不为空，递归遍历右子树
20 if (x.right!=null){
21     preErgodic(x.right,keys);
22 }
23 }
24
25 //测试代码
26 public class Test {
27     public static void main(String[] args) throws Exception {
28         BinaryTree<String, String> bt = new BinaryTree<>();
29         bt.put("E", "5");
30         bt.put("B", "2");
31         bt.put("G", "7");
32         bt.put("A", "1");
33         bt.put("D", "4");
34         bt.put("F", "6");
35         bt.put("H", "8");
36         bt.put("C", "3");
37
38         Queue<String> queue = bt.preErgodic();
39         for (String key : queue) {
40             System.out.println(key+"="+bt.get(key));
41         }
42     }
43 }
44 }
```

## 1.5.2 中序遍历

我们在4.4中创建的树上，添加前序遍历的API：

`public Queue<Key> midErgodic()`：使用中序遍历，获取整个树中的所有键

`private void midErgodic(Node x,Queue<Key> keys)`：使用中序遍历，把指定树x中的所有键放入到keys队列中

**实现步骤：**

- 1.找到当前结点的左子树，如果不为空，递归遍历左子树
- 2.把当前结点的key放入到队列中;
- 3.找到当前结点的右子树，如果不为空，递归遍历右子树

**代码：**

```
1 //使用中序遍历，获取整个树中的所有键
2 public Queue<Key> midErgodic(){
3     Queue<Key> keys = new Queue<>();
4     midErgodic(root,keys);
5 }
```



```
5     return keys;
6 }
7
8 //使用中序遍历，把指定树x中的所有键放入到keys队列中
9 private void midErgodic(Node x, Queue<Key> keys){
10     if (x==null){
11         return;
12     }
13     //1.找到当前结点的左子树，如果不为空，递归遍历左子树
14     if (x.left!=null){
15         midErgodic(x.left,keys);
16     }
17     //2.把当前结点的key放入到队列中;
18     keys.enqueue(x.key);
19     //3.找到当前结点的右子树，如果不为空，递归遍历右子树
20     if (x.right!=null){
21         midErgodic(x.right,keys);
22     }
23 }
24 //测试代码
25 public class Test {
26     public static void main(String[] args) throws Exception {
27         BinaryTree<String, String> bt = new BinaryTree<>();
28         bt.put("E", "5");
29         bt.put("B", "2");
30         bt.put("G", "7");
31         bt.put("A", "1");
32         bt.put("D", "4");
33         bt.put("F", "6");
34         bt.put("H", "8");
35         bt.put("C", "3");
36
37         Queue<String> queue = bt.midErgodic();
38         for (String key : queue) {
39             System.out.println(key+"="+bt.get(key));
40         }
41     }
42 }
43 }
```

### 1.5.3 后序遍历

我们在4.4中创建的树上，添加前序遍历的API：

```
public Queue<Key> afterErgodic() : 使用后序遍历，获取整个树中的所有键
```

```
private void afterErgodic(Node x, Queue<Key> keys) : 使用后序遍历，把指定树x中的所有键放入到keys队列中
```

实现步骤：

- 1.找到当前结点的左子树，如果不为空，递归遍历左子树
- 2.找到当前结点的右子树，如果不为空，递归遍历右子树
- 3.把当前结点的key放入到队列中;



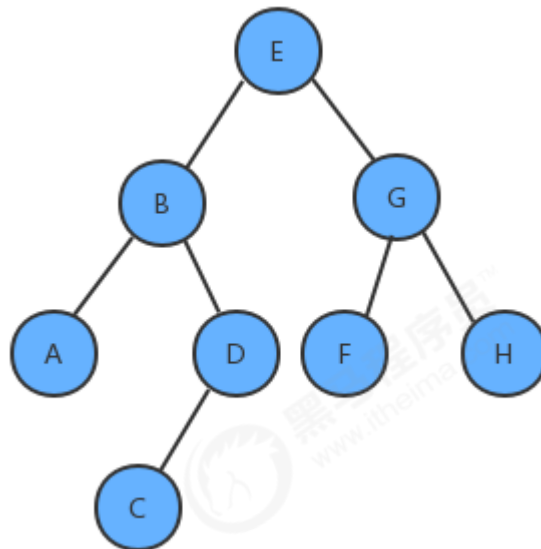


代码：

```
1 //使用后序遍历，获取整个树中的所有键
2 public Queue<Key> afterErgodic(){
3     Queue<Key> keys = new Queue<>();
4     afterErgodic(root,keys);
5     return keys;
6 }
7
8 //使用后序遍历，把指定树x中的所有键放入到keys队列中
9 private void afterErgodic(Node x,Queue<Key> keys){
10     if (x==null){
11         return;
12     }
13     //1.找到当前结点的左子树，如果不为空，递归遍历左子树
14     if (x.left!=null){
15         afterErgodic(x.left,keys);
16     }
17     //2.找到当前结点的右子树，如果不为空，递归遍历右子树
18     if (x.right!=null){
19         afterErgodic(x.right,keys);
20     }
21     //3.把当前结点的key放入到队列中；
22     keys.enqueue(x.key);
23 }
24
25 //测试代码
26 public class Test {
27     public static void main(String[] args) throws Exception {
28         BinaryTree<String, String> bt = new BinaryTree<>();
29         bt.put("E", "5");
30         bt.put("B", "2");
31         bt.put("G", "7");
32         bt.put("A", "1");
33         bt.put("D", "4");
34         bt.put("F", "6");
35         bt.put("H", "8");
36         bt.put("C", "3");
37
38         Queue<String> queue = bt.afterErgodic();
39         for (String key : queue) {
40             System.out.println(key+"="+bt.get(key));
41         }
42     }
43 }
44 }
```

## 1.6 二叉树的层序遍历

所谓的层序遍历，就是从根节点（第一层）开始，依次向下，获取每一层所有结点的值，有二叉树如下：



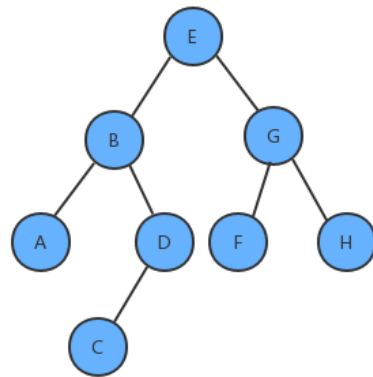
那么层序遍历的结果是：EBGADFHC

我们在4.4中创建的树上，添加层序遍历的API：

```
public Queue<Key> layerErgodic()：使用层序遍历，获取整个树中的所有键
```

**实现步骤：**

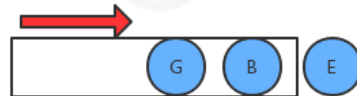
- 1.创建队列，存储每一层的结点；
- 2.使用循环从队列中弹出一个结点：
  - 2.1获取当前结点的key；
  - 2.2如果当前结点的左子结点不为空，则把左子结点放入到队列中
  - 2.3如果当前结点的右子结点不为空，则把右子结点放入到队列中



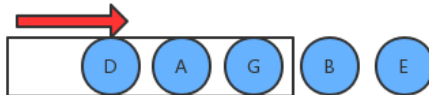
使用队列存储结点，初始化存储根结点E



第一次循环，从队列中弹出结点E，并将左子结点和右子结点放入到队列中



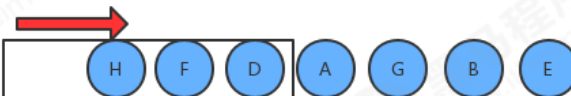
第二次循环，从队列中弹出结点B，并将左子结点和右子结点放入到队列中



第三次循环，从队列中弹出结点G，并将左子结点和右子结点放入到队列中



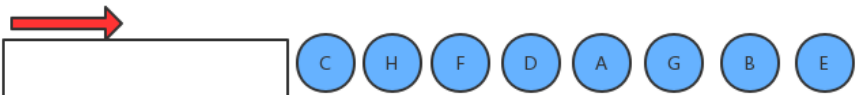
第四次循环，从队列中弹出结点A



第五次循环，从队列中弹出结点D，并把左子结点放入到队列中



第六、七、八次循环，依次从队列中弹出结点F、H、C



代码：

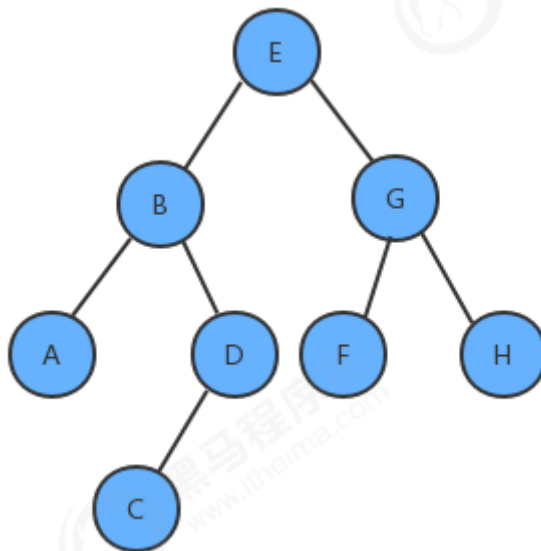
```
1 //使用层序遍历得到树中所有的键
2 public Queue<Key> layerErgodic(){
3     Queue<Key> keys = new Queue<>();
4     Queue<Node> nodes = new Queue<>();
5     nodes.enqueue(root);
6     while(!nodes.isEmpty()){
7         Node x = nodes.dequeue();
8         keys.enqueue(x.key);
9         if (x.left!=null){
10             nodes.enqueue(x.left);
11         }
12         if (x.right!=null){
13             nodes.enqueue(x.right);
14         }
15     }
```

```
16     return keys;
17 }
18 //测试代码
19 public class Test {
20     public static void main(String[] args) throws Exception {
21         BinaryTree<String, String> bt = new BinaryTree<>();
22         bt.put("E", "5");
23         bt.put("B", "2");
24         bt.put("G", "7");
25         bt.put("A", "1");
26         bt.put("D", "4");
27         bt.put("F", "6");
28         bt.put("H", "8");
29         bt.put("C", "3");
30
31         Queue<String> queue = bt.layerErgodic();
32         for (String key : queue) {
33             System.out.println(key+"="+bt.get(key));
34         }
35     }
36 }
37 }
```

## 1.7 二叉树的最大深度问题

需求：

给定一棵树，请计算树的最大深度（树的根节点到最远叶子节点的最长路径上的结点数）；



上面这棵树的最大深度为4。

实现：

我们在1.4中创建的树上，添加如下的API求最大深度：

```
public int maxDepth(): 计算整个树的最大深度
```

```
private int maxDepth(Node x): 计算指定树x的最大深度
```

**实现步骤：**

- 1.如果根结点为空，则最大深度为0；
- 2.计算左子树的最大深度；
- 3.计算右子树的最大深度；
- 4.当前树的最大深度=左子树的最大深度和右子树的最大深度中的较大者+1

**代码：**

```
1  //计算整个树的最大深度
2  public int maxDepth() {
3      return maxDepth(root);
4  }
5
6  //计算指定树x的最大深度
7  private int maxDepth(Node x) {
8      //1.如果根结点为空，则最大深度为0；
9      if (x == null) {
10         return 0;
11     }
12     int max = 0;
13     int maxL = 0;
14     int maxR = 0;
15     //2.计算左子树的最大深度；
16     if (x.left != null) {
17         maxL = maxDepth(x.left);
18     }
19     //3.计算右子树的最大深度；
20     if (x.right != null) {
21         maxR = maxDepth(x.right);
22     }
23     //4.当前树的最大深度=左子树的最大深度和右子树的最大深度中的较大者+1
24     max = maxL > maxR ? maxL + 1 : maxR + 1;
25     return max;
26 }
27
28 //测试代码
29 public class Test {
30     public static void main(String[] args) throws Exception {
31         BinaryTree<String, String> bt = new BinaryTree<>();
32         bt.put("E", "5");
33         bt.put("B", "2");
34         bt.put("G", "7");
35         bt.put("A", "1");
36         bt.put("D", "4");
37         bt.put("F", "6");
38         bt.put("H", "8");
39         bt.put("C", "3");
40
41         int i = bt.maxDepth();
42         System.out.println(i);
```

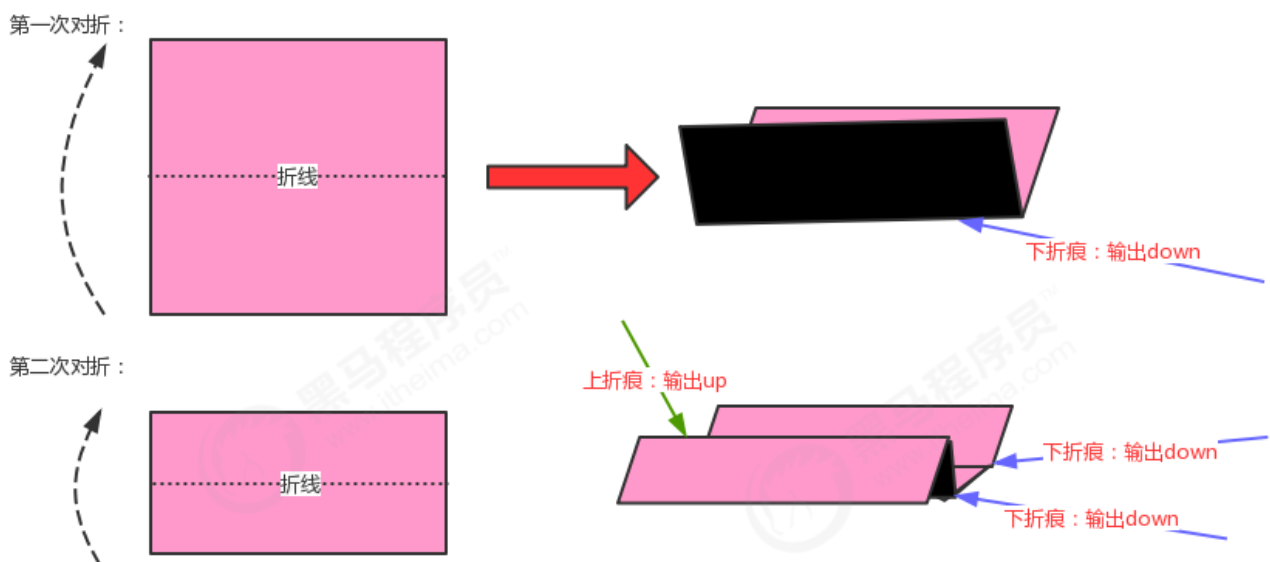
```
43  
44     }  
45 }
```

## 1.8 折纸问题

### 需求：

请把一段纸条竖着放在桌子上，然后从纸条的下边向上方对折1次，压出折痕后展开。此时折痕是凹下去的，即折痕突起的方向指向纸条的背面。如果从纸条的下边向上方连续对折2次，压出折痕后展开，此时有三条折痕，从上到下依次是下折痕、下折痕和上折痕。

给定一个输入参数N，代表纸条都从下边向上方连续对折N次，请从上到下打印所有折痕的方向 例如：N=1时，打印：down；N=2时，打印：down down up

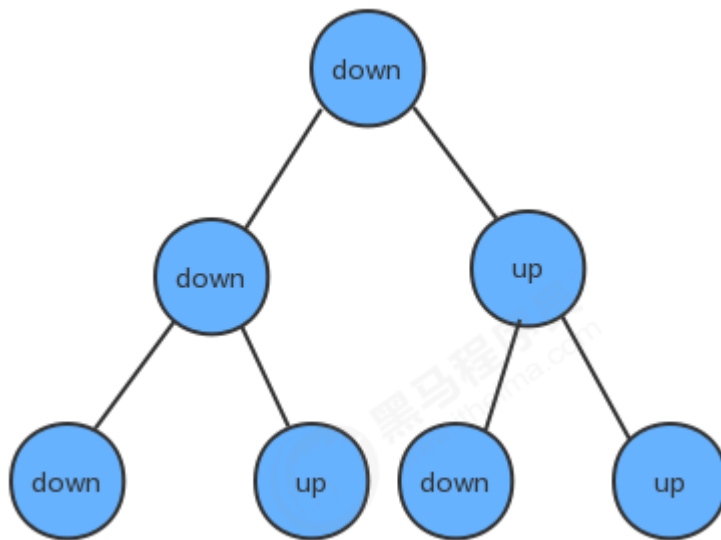


### 分析：

我们把对折后的纸张翻过来，让粉色朝下，这时把第一次对折产生的折痕看做是根结点，那第二次对折产生的下折痕就是该结点的左子结点，而第二次对折产生的上折痕就是该结点的右子结点，这样我们就可以使用树型数据结构来描述对折后产生的折痕。

这棵树有这样的特点：

- 1.根结点为下折痕；
- 2.每一个结点的左子结点为下折痕；
- 3.每一个结点的右子结点为上折痕；



#### 实现步骤：

- 1.定义结点类
- 2.构建深度为N的折痕树；
- 3.使用中序遍历，打印出树中所有结点的内容；

#### 构建深度为N的折痕树：

- 1.第一次对折，只有一条折痕，创建根结点；
- 2.如果不是第一次对折，则使用队列保存根结点；
- 3.循环遍历队列：
  - 3.1从队列中拿出一个结点；
  - 3.2如果这个结点的左子结点不为空，则把这个左子结点添加到队列中；
  - 3.3如果这个结点的右子结点不为空，则把这个右子结点添加到队列中；
  - 3.4判断当前结点的左子结点和右子结点都不为空，如果是，则需要为当前结点创建一个值为down的左子结点，一个值为up的右子结点。

#### 代码：

```
1 public class PaperFolding {
2
3     public static void main(String[] args) {
4         //构建折痕树
5         Node tree = createTree(3);
6
7         //遍历折痕树，并打印
8         printTree(tree);
9     }
10
11     //3.使用中序遍历，打印出树中所有结点的内容；
12     private static void printTree(Node tree) {
13
```



```
14         if (tree==null){
15             return;
16         }
17
18         printTree(tree.left);
19         System.out.print(tree.item+",");
20         printTree(tree.right);
21     }
22
23     //2.构建深度为N的折痕树；
24     private static Node createTree(int N) {
25         Node root = null;
26         for (int i = 0; i < N ; i++) {
27             if (i==0){
28                 //1.第一次对折，只有一条折痕，创建根结点；
29                 root = new Node("down",null,null);
30             }else{
31                 //2.如果不是第一次对折，则使用队列保存根结点；
32                 Queue<Node> queue = new Queue<>();
33                 queue.enqueue(root);
34                 //3.循环遍历队列：
35                 while(!queue.isEmpty()){
36                     //3.1从队列中拿出一个结点；
37                     Node tmp = queue.dequeue();
38                     //3.2如果这个结点的左子结点不为空，则把这个左子结点添加到队列中；
39                     if (tmp.left!=null){
40                         queue.enqueue(tmp.left);
41                     }
42                     //3.3如果这个结点的右子结点不为空，则把这个右子结点添加到队列中；
43                     if (tmp.right!=null){
44                         queue.enqueue(tmp.right);
45                     }
46                     //3.4判断当前结点的左子结点和右子结点都不为空，如果是，则需要为当前结点创建一个
47                     值为down的左子结点，一个值为up的右子结点。
48                     if (tmp.left==null && tmp.right==null){
49                         tmp.left = new Node("down",null,null);
50                         tmp.right = new Node("up",null,null);
51                     }
52                 }
53             }
54
55             return root;
56         }
57
58         //1.定义结点类
59         private static class Node{
60             //存储结点元素
61             String item;
62             //左子结点
63             Node left;
64             //右子结点
65             Node right;
```





```
66  
67     public Node(String item, Node left, Node right) {  
68         this.item = item;  
69         this.left = left;  
70         this.right = right;  
71     }  
72 }  
73 }
```