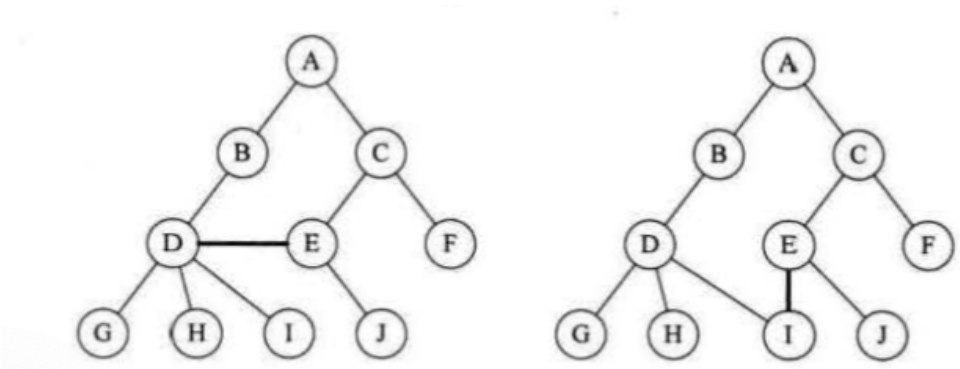


定义
相关术语
树的三种存储结构
父结点（双亲）表示法
孩子表示法
孩子兄弟表示法

定义

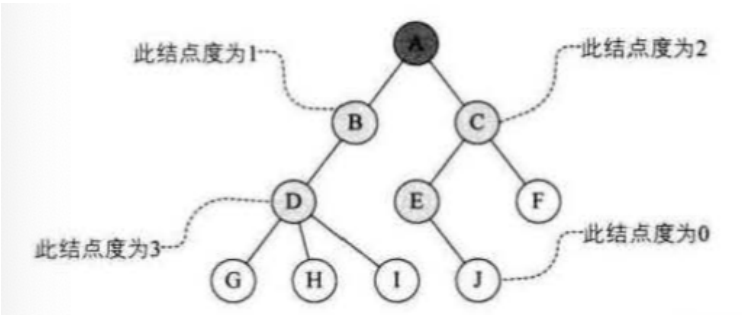
树 (tree) 是包含 n ($n \geq 0$) 个结点的有穷集

1. $n=0$ 时称为空树。
2. 在任意一棵非空树中 有且仅有一个特定的称为根 (Root)的结点，不可能存在多个根结点
3. 子树的个数没有限制，但它们一定是互不相交的，**如下图就不符合树的定义**



相关术语

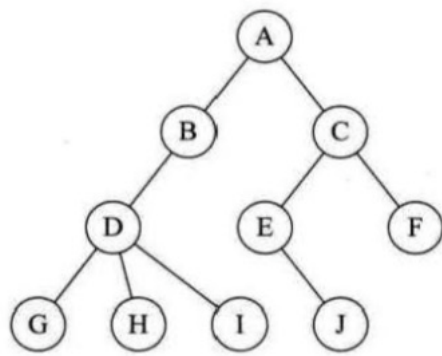
1. 节点的度：一个节点含有的子树的个数称为该节点的度



2. 叶节点或终端节点：度为0的节点称为叶节点
3. 非终端节点或分支节点：度不为0的节点
4. 双亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点
5. 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的孩子节点

6. 兄弟节点：具有相同父节点的节点互称为兄弟节点
7. 树的度：一棵树中，最大的节点的度称为树的度
8. 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推
9. 树的高度或深度：树中节点的最大层次
10. 堂兄弟节点：双亲在同一层的节点互为堂兄弟
11. 节点的祖先：从根到该节点所经分支上的所有节点
12. 子孙：以某节点为根的子树中任一节点都称为该节点的子孙
13. 森林：由m ($m \geq 0$) 棵互不相交的树的集合称为森林

树的三种存储结构



下标	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	2
5	F	2
6	G	3
7	H	3
8	I	3
9	J	4

父结点（双亲）表示法

这种结构的思想比较简单：除了根结点没有父结点外，其余每个结点都有一个唯一的父结点。将所有结点存到一个数组中。一个数值parent指示其双亲在数组中存放的位置。根结点由于没有父结点，parent用-1表示

```
1 package day5.tree.array;  
2  
3 import java.util.ArrayList;  
4 import java.util.Arrays;  
5 import java.util.List;  
6  
7 public class TreeParent<Item> {  
8  
9     public static class Node<T> {  
10         private T data;
```

```

11     private int parent;
12
13     public Node(T data, int parent) {
14         this.data = data;
15         this.parent = parent;
16     }
17
18     public T getData() {
19         return data;
20     }
21
22     @Override
23     public String toString() {
24         return "Node{" +
25             "data=" + data +
26             ", parent=" + parent +
27             '}';
28     }
29 }
30
31 // 树的容量，能容纳的最大结点数
32 private int treeCapacity;
33 // 树的结点数目
34 private int nodesNum;
35 // 存放树的所有结点
36 private Node<Item>[] nodes;
37
38 // 以指定树大小初始化树
39 public TreeParent(int treeCapacity) {
40     this.treeCapacity = treeCapacity;
41     nodes = new Node[treeCapacity];
42 }
43
44 // 以默认树大小初始化树
45 public TreeParent() {
46     treeCapacity = 128;
47     nodes = new Node[treeCapacity];
48 }
49
50
51 public void setRoot(Item data) {
52     // 根结点
53     nodes[0] = new Node<>(data, -1);
54     nodesNum++;
55 }
56
57 public void addChild(Item data, Node<Item> parent) {
58     if (nodesNum < treeCapacity) {
59         // 新的结点放入数组中第一个空闲位置
60         nodes[nodesNum] = new Node<>(data, index(parent));
61         nodesNum++;
62     } else {
63         throw new RuntimeException("树已满，无法再添加结点！");
64     }

```

```

65     }
66
67     // 用nodeNum是因为其中无null, 用treeCapacity里面很多null值根本无需比较
68     private int index(Node<Item> parent) {
69         for (int i = 0; i < nodesNum; i++) {
70             if (nodes[i].equals(parent)) {
71                 return i;
72             }
73         }
74         throw new RuntimeException("无此结点");
75     }
76
77     public void createTree(List<Item> datas, List<Integer> parents) {
78         if (datas.size() > treeCapacity) {
79             throw new RuntimeException("数据过多, 超出树的容量! ");
80         }
81
82         setRoot(datas.get(0));
83         for (int i = 1; i < datas.size(); i++) {
84             addChild(datas.get(i), nodes[parents.get(i - 1)]);
85         }
86     }
87
88     // 是否为空树
89     public boolean isEmpty() {
90         return nodesNum == 0;
91         // or return nodes[0] == null
92     }
93
94     public Node<Item> parentTo(Node<Item> node) {
95         return nodes[node.parent];
96     }
97
98     // 结点的孩子结点
99     public List<Node<Item>> childrenFromNode(Node<Item> parent) {
100         List<Node<Item>> children = new ArrayList<>();
101         for (int i = 0; i < nodesNum; i++) {
102             if (nodes[i].parent == index(parent)) {
103                 children.add(nodes[i]);
104             }
105         }
106         return children;
107     }
108
109     // 树的度
110     public int degreeForTree() {
111         int max = 0;
112         for (int i = 0; i < nodesNum; i++) {
113             if (childrenFromNode(nodes[i]).size() > max) {
114                 max = childrenFromNode(nodes[i]).size();
115             }
116         }
117         return max;
118     }

```

```

119
120     public int degreeForNode(Node<Item> node) {
121         return childrenFromNode(node).size();
122     }
123
124     // 树的深度
125     public int depth() {
126         int max = 0;
127         for (int i = 0; i < nodesNum; i++) {
128             int currentDepth = 1;
129             int parent = nodes[i].parent;
130             while (parent != -1) {
131                 // 向上继续查找父结点，知道根结点
132                 parent = nodes[parent].parent;
133                 currentDepth++;
134             }
135             if (currentDepth > max) {
136                 max = currentDepth;
137             }
138         }
139         return max;
140     }
141
142
143     // 树的结点数
144     public int nodesNum() {
145         return nodesNum;
146     }
147
148     // 返回根结点
149     public Node<Item> root() {
150         return nodes[0];
151     }
152
153     // 让树为空
154     public void clear() {
155         for (int i = 0; i < nodesNum; i++) {
156             nodes[i] = null;
157             nodesNum = 0;
158         }
159     }
160
161     @Override
162     public String toString() {
163         StringBuilder sb = new StringBuilder();
164         sb.append("Tree{\n");
165         for (int i = 0; i < nodesNum - 1; i++) {
166             sb.append(nodes[i]).append(", \n");
167         }
168         sb.append(nodes[nodesNum - 1]).append("}");
169         return sb.toString();
170     }
171
172     public static void main(String[] args) {

```

```

173         // 按照以下定义, 生成树
174         List<String> datas = new ArrayList<>(Arrays.asList("Bob", "Tom", "Jerry", "Rose", "Jack"));
175         List<Integer> parents = new ArrayList<>(Arrays.asList(0, 0, 1, 2));
176
177         TreeParent<String> tree = new TreeParent<>();
178         tree.createTree(datas, parents);
179         TreeParent.Node<String> root = tree.root();
180         // root的第一个孩子
181         TreeParent.Node<String> aChild = tree.childrenFromNode(root).get(0);
182         System.out.println(aChild.getData() + "的父结点是" + tree.parentTo(aChild).getData());
183         System.out.println("根结点的孩子" + tree.childrenFromNode(root));
184         System.out.println("该树深度为" + tree.depth());
185         System.out.println("该树的度为" + tree.degreeForTree());
186         System.out.println("该树的结点数为" + tree.nodesNum());
187         System.out.println(tree);
188     }
189 }
190
191
192 /* 输出
193 Tom的父结点是Bob
194 根结点的孩子[Node{data=Tom, parent=0}, Node{data=Jerry, parent=0}]
195 该树深度为3
196 该树的度为2
197 该树的结点数为5
198 Tree{
199 Node{data=Bob, parent=-1},
200 Node{data=Tom, parent=0},
201 Node{data=Jerry, parent=0},
202 Node{data=Rose, parent=1},
203 Node{data=Jack, parent=2}}
204 */

```

缺点：获取子节点比较麻烦，需要遍历所有的节点来判断父节点是否为当前节点

孩子表示法

双亲表示法获取某结点的所有孩子有点麻烦，索性让每个结点记住他所有的孩子。但是由于一个结点拥有的孩子个数是树的度那么多，但是大多数结点的孩子个数并没有那么多，如果用数组来存放所有孩子，对于大多数结点来说太浪费空间。容量的表来存，选用Java内置的LinkedList是个不错的选择。先用一个数组存放所有的结点信息，该链表只需存储结点在数

```

1 package day5.tree.array;
2
3 import java.util.*;
4
5 public class TreeChildren<Item> {
6
7     public static class Node<T> {
8         private T data;
9         private List<Integer> children;
10
11         public Node(T data) {

```

```

12         this.data = data;
13         this.children = new LinkedList<>();
14     }
15
16     public Node(T data, int[] children) {
17         this.data = data;
18         this.children = new LinkedList<>();
19         for (int child : children) {
20             this.children.add(child);
21         }
22     }
23
24     public T getData() {
25         return data;
26     }
27
28     @Override
29     public String toString() {
30         return "Node{" +
31             "data=" + data +
32             ", children=" + children +
33             '}';
34     }
35 }
36
37 // 树的容量，能容纳的最大结点数
38 private int treeCapacity;
39 // 树的结点数目
40 private int nodesNum;
41 // 存放树的所有结点
42 private Node<Item>[] nodes;
43
44 public TreeChildren(int treeCapacity) {
45     this.treeCapacity = treeCapacity;
46     nodes = new Node[treeCapacity];
47 }
48
49 public TreeChildren() {
50     treeCapacity = 128;
51     nodes = new Node[treeCapacity];
52 }
53
54 public void setRoot(Item data) {
55     nodes[0].data = data;
56     nodesNum++;
57 }
58
59
60 public void addChild(Item data, Node<Item> parent) {
61     if (nodesNum < treeCapacity) {
62         // 新的结点放入数组中第一个空闲位置
63         nodes[nodesNum] = new Node<>(data);
64         // 父结点添加其孩子
65         parent.children.add(nodesNum);

```

```

66         nodesNum++;
67     } else {
68         throw new RuntimeException("树已满, 无法再添加结点! ");
69     }
70 }
71
72 public void createTree(Item[] datas, int[] children) {
73     if (datas.length > treeCapacity) {
74         throw new RuntimeException("数据过多, 超出树的容量! ");
75     }
76
77     for (int i = 0; i < datas.length; i++) {
78         nodes[i] = new Node<>(datas[i], children[i]);
79     }
80
81     nodesNum = datas.length;
82 }
83
84 // 根据给定的结点查找再数组中的位置
85 private int index(Node<Item> node) {
86     for (int i = 0; i < nodesNum; i++) {
87         if (nodes[i].equals(node)) {
88             return i;
89         }
90     }
91     throw new RuntimeException("无此结点");
92 }
93
94 public List<Node<Item>> childrenFromNode(Node<Item> node) {
95     List<Node<Item>> children = new ArrayList<>();
96     for (Integer i : node.children) {
97         children.add(nodes[i]);
98     }
99     return children;
100 }
101
102 public Node<Item> parentTo(Node<Item> node) {
103     for (int i = 0; i < nodesNum; i++) {
104         if (nodes[i].children.contains(index(node))) {
105             return nodes[i];
106         }
107     }
108     return null;
109 }
110
111 // 是否为空树
112 public boolean isEmpty() {
113     return nodesNum == 0;
114     // or return nodes[0] == null
115 }
116
117 // 树的深度
118 public int depth() {
119     return nodeDepth(root());

```



```

120     }
121
122     // 求以node为根结点的子树的深度
123     public int nodeDepth(Node<Item> node) {
124         if (node == null) {
125             return 0;
126         }
127         // max是某个结点所有孩子中的最大深度
128         int max = 0;
129         // 即使没有孩子，返回1也是正确的
130         if (node.children.size() > 0) {
131             for (int i : node.children) {
132                 int depth = nodeDepth(nodes[i]);
133                 if (depth > max) {
134                     max = depth;
135                 }
136             }
137         }
138         // 这里需要+1因为depth -> max是当前结点的孩子的深度，+1才是当前结点的深度
139         return max + 1;
140     }
141
142     public int degree() {
143         int max = 0;
144         for (int i = 0; i < nodesNum; i++) {
145             if (nodes[i].children.size() > max) {
146                 max = nodes[i].children.size();
147             }
148         }
149         return max;
150     }
151
152     public int degreeForNode(Node<Item> node) {
153         return childrenFromNode(node).size();
154     }
155
156     public Node<Item> root() {
157         return nodes[0];
158     }
159
160     // 树的结点数
161     public int nodesNum() {
162         return nodesNum;
163     }
164
165     // 让树为空
166     public void clear() {
167         for (int i = 0; i < nodesNum; i++) {
168             nodes[i] = null;
169             nodesNum = 0;
170         }
171     }
172
173     @Override

```

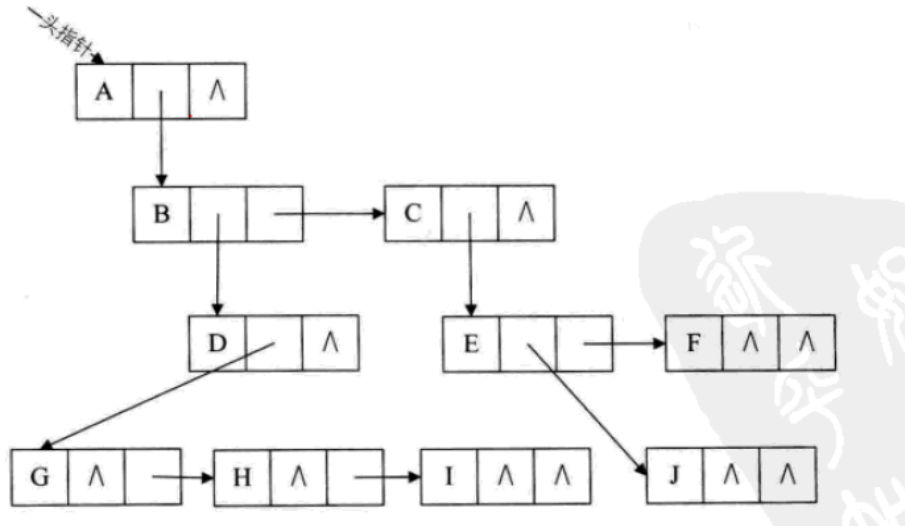
```

174 public String toString() {
175     StringBuilder sb = new StringBuilder();
176     sb.append("Tree{\n");
177     for (int i = 0; i < nodesNum - 1; i++) {
178         sb.append(nodes[i]).append(", \n");
179     }
180     sb.append(nodes[nodesNum - 1]).append("}");
181     return sb.toString();
182 }
183
184 public static void main(String[] args) {
185     String[] datas = {"Bob", "Tom", "Jerry", "Rose", "Jack"};
186     int[][] children = {{1, 2}, {3}, {4}, {}, {}};
187     TreeChildren<String> tree = new TreeChildren<>();
188     tree.createTree(datas, children);
189
190     TreeChildren.Node<String> root = tree.root();
191     TreeChildren.Node<String> rightChild = tree.childrenFromNode(root).get(1);
192     System.out.println(rightChild.getData() + "的度为" + tree.degreeForNode(rightChild));
193     System.out.println("该树的结点数为" + tree.nodesNum());
194     System.out.println("该树根结点" + tree.root());
195     System.out.println("该树的深度为" + tree.depth());
196     System.out.println("该树的度为" + tree.degree());
197     System.out.println(tree.parentTo(rightChild));
198
199     tree.addChild("Joe", root);
200     System.out.println("该树的度为" + tree.degree());
201     System.out.println(tree);
202
203 }
204 }
205
206
207 /* 输出
208 Jerry的度为1
209 该树的结点数为5
210 该树根结点Node{data=Bob, children=[1, 2]}
211 该树的深度为3
212 该树的度为2
213 Node{data=Bob, children=[1, 2]}
214 该树的度为3
215 Tree{
216 Node{data=Bob, children=[1, 2, 5]},
217 Node{data=Tom, children=[3]},
218 Node{data=Jerry, children=[4]},
219 Node{data=Rose, children=[]},
220 Node{data=Jack, children=[]},
221 Node{data=Joe, children=[]}}
222 */

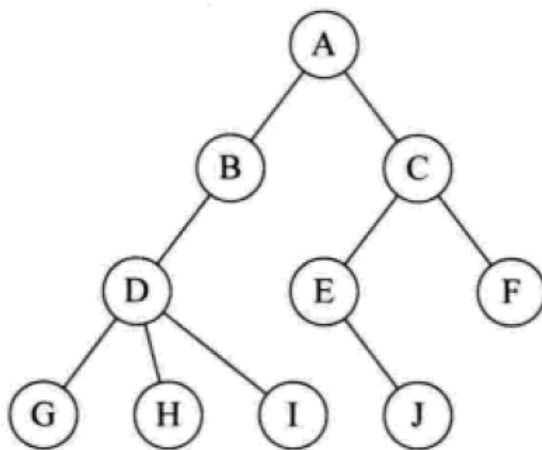
```

孩子兄弟表示法

还有一种表示法，关注某结点的孩子结点之间的关系，他们互为兄弟。一个结点可能有孩子，也有可能兄弟，也可能这种思想，可以用具有两个指针域（一个指向当前结点的孩子，一个指向其兄弟）的链表实现，这种链表又称为二叉链表



整个结构就是一条有两个走向的错综复杂的链表，垂直走向是深入到结点的子子孙孙；水平走向就是查找它的兄弟姐妹的，上图其实就是下面这棵树



```

1 package day5.tree.array;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class TreeChildSib<Item> {
7
8     public static class Node<T> {
9         private T data;
10        private Node<T> nextChild;
11        private Node<T> nextSib;
12
13        public T getData() {
14            return data;
15        }
16    }
17 }

```

```

15     }
16
17     public Node<T> data() {
18         this.data = data;
19     }
20
21     public Node<T> getNextChild() {
22         return nextChild;
23     }
24
25     public Node<T> getNextSib() {
26         return nextSib;
27     }
28
29     @Override
30     public String toString() {
31         String child = nextChild == null ? null : nextChild.getData().toString();
32         String sib = nextSib == null ? null : nextSib.getData().toString();
33
34         return "Node{" +
35             "data=" + data +
36             ", nextChild=" + child +
37             ", nextSib=" + sib +
38             '}';
39     }
40 }
41
42 private Node<Item> root;
43 // 存放所有结点，每次新增一个结点就add进来
44 private List<Node<Item>> nodes = new ArrayList<>();
45
46 // 以指定的根结点初始化树
47 public TreeChildSib(Item data) {
48     setRoot(data);
49 }
50
51 // 空参数构造器
52 public TreeChildSib() {
53
54 }
55
56 public void setRoot(Item data) {
57     root = new Node<>(data);
58     nodes.add(root);
59 }
60
61 public void addChild(Item data, Node<Item> parent) {
62     Node<Item> node = new Node<>(data);
63     // 如果该parent是叶子结点，没有孩子
64     if (parent.nextChild == null) {
65         parent.nextChild = node;
66         // parent有孩子了，只能放在n其第一个孩子的最后一个兄弟之后
67     } else {
68         // 从parent的第一个孩子开始，追溯到最后一个兄弟

```

```

69         Node<Item> current = parent.nextChild;
70         while (current.nextSib != null) {
71             current = current.nextSib;
72         }
73         current.nextSib = node;
74     }
75     nodes.add(node);
76 }
77
78 public List<Node<Item>> childrenFromNode(Node<Item> node) {
79     List<Node<Item>> children = new ArrayList<>();
80     for (Node<Item> cur = node.nextChild; cur != null; cur = cur.nextSib) {
81         {
82             children.add(cur);
83         }
84     }
85     return children;
86 }
87
88 public Node<Item> parentTo(Node<Item> node) {
89     for (Node<Item> eachNode : nodes) {
90         if (childrenFromNode(eachNode).contains(node)) {
91             return eachNode;
92         }
93     }
94     return null;
95 }
96
97 public boolean isEmpty() {
98     return nodes.size() == 0;
99 }
100
101 public Node<Item> root() {
102     return root;
103 }
104
105 public int nodesNum() {
106     return nodes.size();
107 }
108
109 public int depth() {
110     return nodeDepth(root);
111 }
112
113 public int nodeDepth(Node<Item> node) {
114     if (node == null) {
115         return 0;
116     }
117
118     int max = 0;
119     if (childrenFromNode(node).size() > 0) {
120         for (Node<Item> child : childrenFromNode(node)) {
121             int depth = nodeDepth(child);
122             if (depth > max) {

```

```

123         max = depth;
124     }
125 }
126 }
127     return max + 1;
128 }
129
130 public int degree() {
131
132     int max = 0;
133     for (Node<Item> node : nodes) {
134         if (childrenFromNode(node).size() > max) {
135             max = childrenFromNode(node).size();
136         }
137     }
138     return max;
139 }
140
141 public int degreeForNode(Node<Item> node) {
142     return childrenFromNode(node).size();
143 }
144
145 public void deleteNode(Node<Item> node) {
146     if (node == null) {
147         return;
148     }
149     deleteNode(node.nextChild);
150     deleteNode(node.nextSib);
151     node.nextChild = null;
152     node.nextSib = null;
153     node.data = null;
154     nodes.remove(node);
155 }
156
157 public void clear() {
158     deleteNode(root);
159     root = null;
160 }
161
162 @Override
163 public String toString() {
164     StringBuilder sb = new StringBuilder();
165     sb.append("Tree{\n");
166     for (int i = 0; i < nodesNum() - 1; i++) {
167         sb.append(nodes.get(i)).append(", \n");
168     }
169     sb.append(nodes.get(nodesNum() - 1)).append("}");
170     return sb.toString();
171 }
172
173 public static void main(String[] args) {
174     TreeChildSib<String> tree = new TreeChildSib<>("A");
175     TreeChildSib.Node<String> root = tree.root();
176     tree.addChild("B", root);

```

```

177     tree.addChild("C", root);
178     tree.addChild("D", root);
179     TreeChildSib.Node<String> child1 = tree.childrenFromNode(root).get(0);
180     TreeChildSib.Node<String> child2 = tree.childrenFromNode(root).get(1);
181     TreeChildSib.Node<String> child3 = tree.childrenFromNode(root).get(2);
182     tree.addChild("E", child1);
183     tree.addChild("F", child2);
184     tree.addChild("G", child1);
185     tree.addChild("H", child3);
186
187     System.out.println(tree);
188     System.out.println("该树结点数为" + tree.nodesNum());
189     System.out.println("该树深度为" + tree.depth());
190     System.out.println("该树的度为" + tree.degree());
191     System.out.println(child1.getData() + "的度为" + tree.degreeForNode(child1));
192     System.out.println(child2.getData() + "的父结点为" + tree.parentTo(child2).getData());
193
194     tree.clear();
195     System.out.println(child1);
196     System.out.println(tree.isEmpty());
197 }
198 }
199
200
201 /*输出
202 Tree{
203 Node{data=A, nextChild=B, nextSib=null},
204 Node{data=B, nextChild=E, nextSib=C},
205 Node{data=C, nextChild=F, nextSib=D},
206 Node{data=D, nextChild=H, nextSib=null},
207 Node{data=E, nextChild=null, nextSib=G},
208 Node{data=F, nextChild=null, nextSib=null},
209 Node{data=G, nextChild=null, nextSib=null},
210 Node{data=H, nextChild=null, nextSib=null}}
211 该树结点数为8
212 该树深度为3
213 该树的度为3
214 B的度为2
215 C的父结点为A
216 Node{data=null, nextChild=null, nextSib=null}
217 true
218 */

```