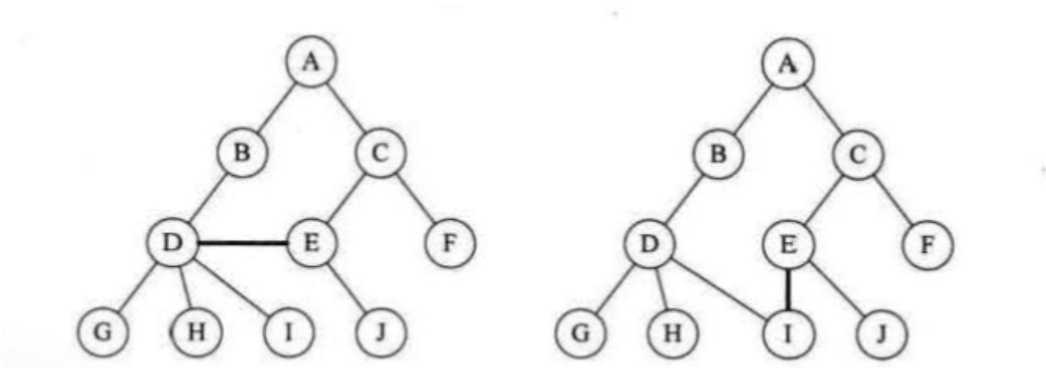
定义

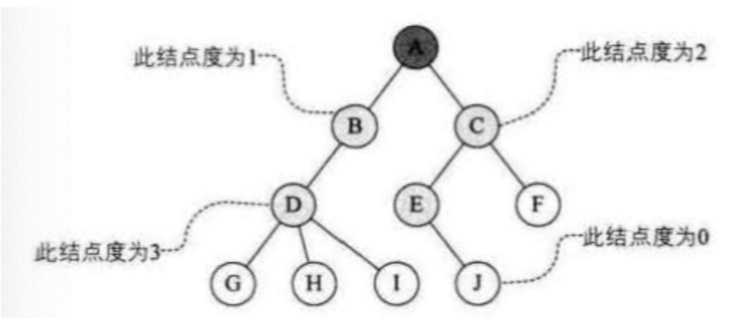
树（tree）是包含n（n>=0）个结点的有穷集

1. n=O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

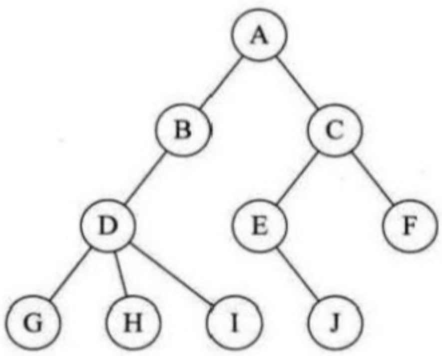
1. 节点的度：一个节点含有的子树的个数称为该节点的度



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>=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

```java
package day5.tree.array;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TreeParent<Item> {

    public static class Node<T> {
        private T data;
```

```java
        private int parent;

        public Node(T data, int parent) {
            this.data = data;
            this.parent = parent;
        }

        public T getData() {
            return data;
        }

        @Override
        public String toString() {
            return "Node{" +
                    "data=" + data +
                    ", parent=" + parent +
                    '}';
        }
    }

    // 树的容量，能容纳的最大结点数
    private int treeCapacity;
    // 树的结点数目
    private int nodesNum;
    // 存放树的所有结点
    private Node<Item>[] nodes;

    // 以指定树大小初始化树
    public TreeParent(int treeCapacity) {
        this.treeCapacity = treeCapacity;
        nodes = new Node[treeCapacity];

    }

    // 以默认的树大小初始化树
    public TreeParent() {
        treeCapacity = 128;
        nodes = new Node[treeCapacity];
    }

    public void setRoot(Item data) {
        // 根结点
        nodes[0] = new Node<>(data, -1);
        nodesNum++;
    }

    public void addChild(Item data, Node<Item> parent) {
        if (nodesNum < treeCapacity) {
            // 新的结点放入数组中第一个空闲位置
            nodes[nodesNum] = new Node<>(data, index(parent));
            nodesNum++;
        } else {
            throw new RuntimeException("树已满，无法再添加结点！");
        }
```

```java
    }

    // 用nodeNum是因为其中无null，用treeCapacity里面很多null值根本无需比较
    private int index(Node<Item> parent) {
        for (int i = 0; i < nodesNum; i++) {
            if (nodes[i].equals(parent)) {
                return i;
            }
        }
        throw new RuntimeException("无此结点");
    }

    public void createTree(List<Item> datas, List<Integer> parents) {
        if (datas.size() > treeCapacity) {
            throw new RuntimeException("数据过多，超出树的容量！");
        }

        setRoot(datas.get(0));
        for (int i = 1; i < datas.size(); i++) {
            addChild(datas.get(i), nodes[parents.get(i - 1)]);
        }
    }

    // 是否为空树
    public boolean isEmpty() {
        return nodesNum == 0;
        // or return nodes[0] == null
    }

    public Node<Item> parentTo(Node<Item> node) {
        return nodes[node.parent];
    }

    // 结点的孩子结点
    public List<Node<Item>> childrenFromNode(Node<Item> parent) {
         List<Node<Item>> children = new ArrayList<>();
         for (int i = 0; i < nodesNum; i++) {
             if (nodes[i].parent == index(parent)) {
                 children.add(nodes[i]);
             }
         }
         return children;
     }

     // 树的度
     public int degreeForTree() {
         int max = 0;
         for (int i = 0; i < nodesNum; i++) {
             if (childrenFromNode(nodes[i]).size() > max) {
                 max = childrenFromNode(nodes[i]).size();
             }
         }
         return max;
     }
```

```java

    public int degreeForNode(Node<Item> node) {
        return childrenFromNode(node).size();
    }

    // 树的深度
    public int depth() {
        int max = 0;
        for (int i = 0; i < nodesNum; i++) {
            int currentDepth = 1;
            int parent = nodes[i].parent;
            while (parent != -1) {
                // 向上继续查找父结点，知道根结点
                parent = nodes[parent].parent;
                currentDepth++;
            }
            if (currentDepth > max) {
                max = currentDepth;
            }
        }
        return max;
    }


    // 树的结点数
    public int nodesNum() {
        return nodesNum;
    }

    // 返回根结点
    public Node<Item> root() {
        return nodes[0];
    }

    // 让树为空
    public void clear() {
        for (int i = 0; i < nodesNum; i++) {
            nodes[i] = null;
            nodesNum = 0;
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Tree{\n");
        for (int i = 0; i < nodesNum - 1; i++) {
            sb.append(nodes[i]).append(", \n");
        }
        sb.append(nodes[nodesNum - 1]).append("}");
        return sb.toString();
    }

    public static void main(String[] args) {
```

```
        // 按照以下定义，生成树
        List<String> datas = new ArrayList<>(Arrays.asList("Bob", "Tom", "Jerry", "Rose", "Jack"));
        List<Integer> parents = new ArrayList<>(Arrays.asList(0, 0, 1, 2));

        TreeParent<String> tree = new TreeParent<>();
        tree.createTree(datas, parents);
        TreeParent.Node<String> root = tree.root();
        // root的第一个孩子
        TreeParent.Node<String> aChild = tree.childrenFromNode(root).get(0);
        System.out.println(aChild.getData() + "的父结点是" + tree.parentTo(aChild).getData());
        System.out.println("根结点的孩子" + tree.childrenFromNode(root));
        System.out.println("该树深度为" + tree.depth());
        System.out.println("该树的度为" + tree.degreeForTree());
        System.out.println("该树的结点数为" + tree.nodesNum());
        System.out.println(tree);
    }
}


/* 输出
Tom的父结点是Bob
根结点的孩子[Node{data=Tom, parent=0}, Node{data=Jerry, parent=0}]
该树深度为3
该树的度为2
该树的结点数为5
Tree{
Node{data=Bob, parent=-1},
Node{data=Tom, parent=0},
Node{data=Jerry, parent=0},
Node{data=Rose, parent=1},
Node{data=Jack, parent=2}}
*/
```

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了容量的表来存，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

```
package day5.tree.array;

import java.util.*;

public class TreeChildren<Item> {

    public static class Node<T> {
        private T data;
        private List<Integer> children;

         public Node(T data) {
```

```java
            this.data = data;
            this.children = new LinkedList<>();
        }

        public Node(T data, int[] children) {
            this.data = data;
            this.children = new LinkedList<>();
            for (int child : children) {
                this.children.add(child);
            }
        }

        public T getData() {
            return data;
        }

        @Override
        public String toString() {
            return "Node{" +
                    "data=" + data +
                    ", children=" + children +
                    '}';
        }
    }

    // 树的容量，能容纳的最大结点数
    private int treeCapacity;
    // 树的结点数目
    private int nodesNum;
    // 存放树的所有结点
    private Node<Item>[] nodes;

    public TreeChildren(int treeCapacity) {
        this.treeCapacity = treeCapacity;
        nodes = new Node[treeCapacity];
    }

    public TreeChildren() {
        treeCapacity = 128;
        nodes = new Node[treeCapacity];
    }

    public void setRoot(Item data) {
        nodes[0].data = data;
        nodesNum++;
    }


    public void addChild(Item data, Node<Item> parent) {
        if (nodesNum < treeCapacity) {
            // 新的结点放入数组中第一个空闲位置
            nodes[nodesNum] = new Node<>(data);
            // 父结点添加其孩子
            parent.children.add(nodesNum);
```

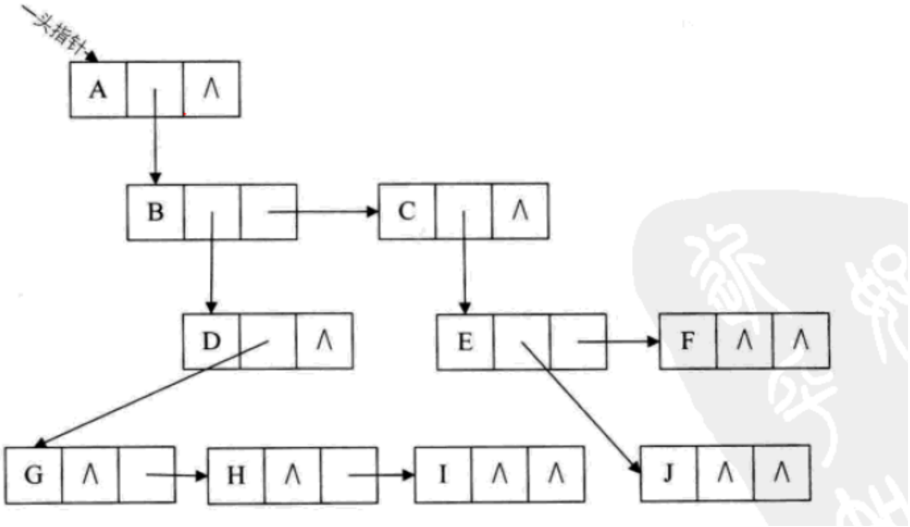
```java
            nodesNum++;
        } else {
            throw new RuntimeException("树已满，无法再添加结点！");
        }
    }

    public void createTree(Item[] datas, int[][] children) {
        if (datas.length > treeCapacity) {
            throw new RuntimeException("数据过多，超出树的容量！");
        }

        for (int i = 0; i < datas.length; i++) {
            nodes[i] = new Node<>(datas[i], children[i]);
        }

        nodesNum = datas.length;
    }

    // 根据给定的结点查找再数组中的位置
    private int index(Node<Item> node) {
        for (int i = 0; i < nodesNum; i++) {
            if (nodes[i].equals(node)) {
                return i;
            }
        }
        throw new RuntimeException("无此结点");
    }

    public List<Node<Item>> childrenFromNode(Node<Item> node) {
        List<Node<Item>> children = new ArrayList<>();
        for (Integer i : node.children) {
            children.add(nodes[i]);
        }
        return children;
    }

    public Node<Item> parentTo(Node<Item> node) {
        for (int i = 0; i < nodesNum; i++) {
            if (nodes[i].children.contains(index(node))) {
                return nodes[i];
            }
        }
        return null;
    }

    // 是否为空树
    public boolean isEmpty() {
        return nodesNum == 0;
        // or return nodes[0] == null
    }

    // 树的深度
    public int depth() {
        return nodeDepth(root());
```

```java
    }

    // 求以node为根结点的子树的深度
    public int nodeDepth(Node<Item> node) {
        if (node == null) {
            return 0;
        }
        // max是某个结点所有孩子中的最大深度
        int max = 0;
        // 即使没有孩子，返回1也是正确的
        if (node.children.size() > 0) {
            for (int i : node.children) {
                int depth = nodeDepth(nodes[i]);
                if (depth > max) {
                    max = depth;
                }
            }
        }
        // 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
        return max + 1;
    }

    public int degree() {
        int max = 0;
        for (int i = 0; i < nodesNum; i++) {
            if (nodes[i].children.size() > max) {
                max = nodes[i].children.size();
            }
        }
        return max;
    }

    public int degreeForNode(Node<Item> node) {
        return childrenFromNode(node).size();
    }

    public Node<Item> root() {
        return nodes[0];
    }

    // 树的结点数
    public int nodesNum() {
        return nodesNum;
    }

    // 让树为空
    public void clear() {
        for (int i = 0; i < nodesNum; i++) {
            nodes[i] = null;
            nodesNum = 0;
        }
    }

    @Override
```

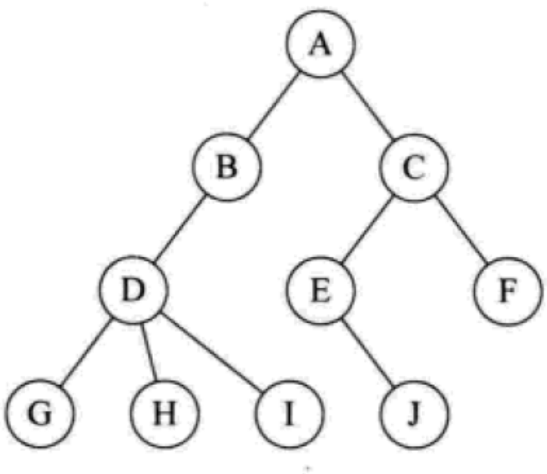
```java
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Tree{\n");
        for (int i = 0; i < nodesNum - 1; i++) {
            sb.append(nodes[i]).append(", \n");
        }
        sb.append(nodes[nodesNum - 1]).append("}");
        return sb.toString();
    }

    public static void main(String[] args) {
        String[] datas = {"Bob", "Tom", "Jerry", "Rose", "Jack"};
        int[][] children = {{1, 2}, {3}, {4}, {}, {}};
        TreeChildren<String> tree = new TreeChildren<>();
        tree.createTree(datas, children);

        TreeChildren.Node<String> root = tree.root();
        TreeChildren.Node<String> rightChild = tree.childrenFromNode(root).get(1);
        System.out.println(rightChild.getData() + "的度为" + tree.degreeForNode(rightChild));
        System.out.println("该树的结点数为" + tree.nodesNum());
        System.out.println("该树根结点" + tree.root());
        System.out.println("该树的深度为" + tree.depth());
        System.out.println("该树的度为" + tree.degree());
        System.out.println(tree.parentTo(rightChild));

        tree.addChild("Joe", root);
        System.out.println("该树的度为" + tree.degree());
        System.out.println(tree);

    }
}


/* 输出
Jerry的度为1
该树的结点数为5
该树根结点Node{data=Bob, children=[1, 2]}
该树的深度为3
该树的度为2
Node{data=Bob, children=[1, 2]}
该树的度为3
Tree{
Node{data=Bob, children=[1, 2, 5]},
Node{data=Tom, children=[3]},
Node{data=Jerry, children=[4]},
Node{data=Rose, children=[]},
Node{data=Jack, children=[]},
Node{data=Joe, children=[]}}
*/
```

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，也有可能有兄弟，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构的，上图其实就是下面这棵树




```
package day5.tree.array;

import java.util.ArrayList;
import java.util.List;

public class TreeChildSib<Item> {

    public static class Node<T> {
        private T data;
         private Node<T> nextChild;
         private Node<T> nextSib;

         public T getData() {
             return data;
```

```java
        }

        public Node(T data) {
            this.data = data;
        }

        public Node<T> getNextChild() {
            return nextChild;
        }

        public Node<T> getNextSib() {
            return nextSib;
        }

        @Override
        public String toString() {
            String child = nextChild == null ? null : nextChild.getData().toString();
            String sib = nextSib == null ? null : nextSib.getData().toString();

            return "Node{" +
                    "data=" + data +
                    ", nextChild=" + child +
                    ", nextSib=" + sib +
                    '}';
        }
    }

    private Node<Item> root;
    // 存放所有结点，每次新增一个结点就add进来
    private List<Node<Item>> nodes = new ArrayList<>();

    // 以指定的根结点初始化树
    public TreeChildSib(Item data) {
        setRoot(data);
    }

    // 空参数构造器
    public TreeChildSib() {

    }

    public void setRoot(Item data) {
        root = new Node<>(data);
        nodes.add(root);
    }

    public void addChild(Item data, Node<Item> parent) {
        Node<Item> node = new Node<>(data);
        // 如果该parent是叶子结点，没有孩子
        if (parent.nextChild == null) {
            parent.nextChild = node;
            // 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
        } else {
            // 从parent的第一个孩子开始，追溯到最后一个兄弟
```

```
            Node<Item> current = parent.nextChild;
            while (current.nextSib != null) {
                current = current.nextSib;
            }
            current.nextSib = node;
        }
        nodes.add(node);
    }

    public List<Node<Item>> childrenFromNode(Node<Item> node) {
        List<Node<Item>> children = new ArrayList<>();
        for (Node<Item> cur = node.nextChild; cur != null; cur = cur.nextSib) {
            {
                children.add(cur);
            }
        }
        return children;
    }

    public Node<Item> parentTo(Node<Item> node) {
        for (Node<Item> eachNode : nodes) {
            if (childrenFromNode(eachNode).contains(node)) {
                return eachNode;
            }
        }
        return null;
    }

    public boolean isEmpty() {
        return nodes.size() == 0;
    }

     public Node<Item> root() {
         return root;
     }

     public int nodesNum() {
         return nodes.size();
     }

     public int depth() {
         return nodeDepth(root);
     }

     public int nodeDepth(Node<Item> node) {
         if (node == null) {
             return 0;
         }

         int max = 0;
         if (childrenFromNode(node).size() > 0) {
             for (Node<Item> child : childrenFromNode(node)) {
                 int depth = nodeDepth(child);
                 if (depth > max) {
```

```
                max = depth;
            }
        }
    }
    return max + 1;
}

public int degree() {

    int max = 0;
    for (Node<Item> node : nodes) {
        if (childrenFromNode(node).size() > max) {
            max = childrenFromNode(node).size();
        }
    }
    return max;
}

public int degreeForNode(Node<Item> node) {
    return childrenFromNode(node).size();
}

public void deleteNode(Node<Item> node) {
    if (node == null) {
        return;
    }
    deleteNode(node.nextChild);
    deleteNode(node.nextSib);
    node.nextChild = null;
    node.nextSib = null;
    node.data = null;
    nodes.remove(node);
}

public void clear() {
    deleteNode(root);
    root = null;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("Tree{\n");
    for (int i = 0; i < nodesNum() - 1; i++) {
        sb.append(nodes.get(i)).append(", \n");
    }
    sb.append(nodes.get(nodesNum() - 1)).append("}");
    return sb.toString();
}

public static void main(String[] args) {
    TreeChildSib<String> tree = new TreeChildSib<>("A");
    TreeChildSib.Node<String> root = tree.root();
    tree.addChild("B", root);
```

```java
        tree.addChild("C", root);
        tree.addChild("D", root);
        TreeChildSib.Node<String> child1 = tree.childrenFromNode(root).get(0);
        TreeChildSib.Node<String> child2 = tree.childrenFromNode(root).get(1);
        TreeChildSib.Node<String> child3 = tree.childrenFromNode(root).get(2);
        tree.addChild("E", child1);
        tree.addChild("F", child2);
        tree.addChild("G", child1);
        tree.addChild("H", child3);

        System.out.println(tree);
        System.out.println("该树结点数为" + tree.nodesNum());
        System.out.println("该树深度为" + tree.depth());
        System.out.println("该树的度为" + tree.degree());
        System.out.println(child1.getData() + "的度为" + tree.degreeForNode(child1));
        System.out.println(child2.getData() + "的父结点为" + tree.parentTo(child2).getData());

        tree.clear();
        System.out.println(child1);
        System.out.println(tree.isEmpty());
    }
}


/*输出
Tree{
Node{data=A, nextChild=B, nextSib=null},
Node{data=B, nextChild=E, nextSib=C},
Node{data=C, nextChild=F, nextSib=D},
Node{data=D, nextChild=H, nextSib=null},
Node{data=E, nextChild=null, nextSib=G},
Node{data=F, nextChild=null, nextSib=null},
Node{data=G, nextChild=null, nextSib=null},
Node{data=H, nextChild=null, nextSib=null}}
该树结点数为8
该树深度为3
该树的度为3
B的度为2
C的父结点为A
Node{data=null, nextChild=null, nextSib=null}
true
*/
```