# **数据结构与算法--树的三种存储结构**

　　**数据结构与算法--树的三种存储结构**

　　之前学的链表、队列、栈，都是线性表，因为其中每个数据元素只有一个前驱和一个后继。是一对一的关系。

　　假如是一对多的关系呢？这种数据结构就是今天要学的树。

　**树的定义**

树是由有限个结点（假设为n）构成的集合。n = 0说明这是棵空树。一棵树中，有且只有一个根结点，按照习惯在位于树的顶端。根结点可以理解为始祖一般的存在，他们有若干个孩子，但是他本身没有双亲。如图1中结点A就是根结点。



假设把树从中截断（并没有），可以得到若干个互不相交（没有交集）的集合，每一个集合本身又是一棵树，称为根的子树，以后就直接叫“子树”。比如假想我断开了A-B, A-C的连接，结点B和C没有双亲成为了根结点，产生了两棵互不相交的子树。如下。



什么叫互不相交呢？下面粗线连接部分如左图D和E，他们到底属于哪棵树？可以认为构成子树的集合之间有交集，造成了原来的子树T1和子树T2相交。**这样相交的树，不叫子树，因为这不符合树的定义。**

****

　　**树的结点与深度**

　　上面的图中每一个圆圈代表的就是一个结点，结点之间的连线表示了结点之间的关系。结点拥有的子树数目称为该结点的度，也可以简单理解为该结点拥有的孩子个数。如上面的图中A结点的度为2。度为0的结点称为叶子结点——也就是没孩子。度不为0的结点称为非叶子结点或非终端结点。树的度为树中各个结点度的最大值，如图1中结点D拥有的孩子有3个，最多，所以树的度就是3。

　　树中各个结点之间有什么关系呢？某结点它的子树的根称做该节点的孩子（Child），该结点称为这些孩子的双亲（Parent），或者直接叫父结点。同一个父结点的孩子之间互称为兄弟（Sibling）。举例来说，图1中结点C的孩子结点有E和F，E和F的父结点为C，而E和F之间是兄弟关系。

　　树的深度就是指树的层数，根结点处为第一层，其孩子结点为第二层，以此类推。易知图1树的深度为4。

　**树的存储结构**

**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。每个结点都有一个数据域data和一个数值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。



　　package Chap6;

　　import java.util.ArrayList;

　　import java.util.Arrays;

　　import java.util.List;

　　public class TreeParent<Item> {

　　 public static class Node<T> {

　　 private T data;

　　 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;

　　 // 以指定的根结点和树大小初始化树

　　 public TreeParent(int treeCapacity) {

　　 this.treeCapacity = treeCapacity;

　　 nodes = new Node[treeCapacity];

　　 }

　　 // 以指定的根结点和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！");

　　 }

　　 }

　　 // 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;

　　 }

　　 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);

　　 }

　　}

　　/\* Outputs

　　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}}

　　\*/

　　setRoot方法必须首先被调用，可以看到根结点始终被放置在数组中第一个位置（下标为0），之后才能调用addChild方法。createTree将创建树的过程简化了，我们只需输入一组数据datas，和这组数据对应的parents传给createTree就行，注意datas的第一个数据是根结点信息，在代码中默认使用-1表示其parent，所以它在parents中没有对应的parent值，也就是说datas的第二个值才和parents的第一个值对应，以此类推。树创建完成后，若想再添加结点到树，调用addChild就行。

　　childrenFromNode方法获取某个结点的所有孩子结点，由代码看出它需要遍历所有结点，复杂度为O(n)。parentTo方法获取某结点的父结点，复杂度O(1)。

　　另外求树的度的时候，也是遍历了所有结点，从中选出最大的度作为树的度，复杂度为O(n)。求树的深度也类似，遍历了所有结点，从下往上，一直追溯到根结点，用currentDepth记录了当前结点的深度，从所有结点中选择最大深度值作为树的深度。

　**孩子表示法**

　　换种思路，既然双亲表示法获取某结点的所有孩子有点麻烦，我们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一个不确定的值，虽然最多只有树的度那么多，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。自然我们容易想到用一个可变容量的表来存，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组中的下标就行了。

　　package Chap6;

　　import java.util.\*;

　　public class TreeChildren<Item> {

　　 public static class Node<T> {

　　 private T data;

　　 private List<Integer> children;

　　 public Node(T data) {

　　 this.data = data;

　　 children = new LinkedList<>();

　　 }

　　 public Node(T data, List<Integer> children) {

　　 this.data = data;

　　 this.children = children;

　　 }

　　 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);

　　 nodesNum++;

　　 } else {

　　 throw new RuntimeException("树已满，无法再添加结点！");

　　 }

　　 }

　　 public void createTree(List<Item> datas, List<Integer>[] children) {

　　 if (datas.size() > treeCapacity) {

　　 throw new RuntimeException("数据过多，超出树的容量！");

　　 }

　　 for (int i = 0; i < datas.size(); i++) {

　　 nodes[i] = new Node<>(datas.get(i), children[i]);

　　 }

　　 nodesNum = datas.size();

　　 }

　　 // 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());

　　 }

　　 // 求以node为根结点的子树的深度

　　 public int nodeDepth(Node<Item> node) {

　　 if (isEmpty()) {

　　 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

　　 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) {

　　 List<String> datas = new ArrayList<>(Arrays.asList("Bob", "Tom", "Jerry", "Rose", "Jack"));

　　 List<Integer>[] children = new LinkedList[datas.size()];

　　 children[0] = new LinkedList<>(Arrays.asList(1, 2));

　　 children[1] = new LinkedList<>(Collections.singletonList(3));

　　 children[2] = new LinkedList<>(Collections.singletonList(4));

　　 children[3] = new LinkedList<>();

　　 children[4] = new LinkedList<>();

　　 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);

　　 }

　　}

　　/\* Outputs

　　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=[]}}

　　\*/

　　有些方法的实现和双亲表示法一样，有些方法的实现改变了。

　　addChild参数列表没变，实现变为新添加的结点在数组中的下标（其实就是数组的第一个空闲位置）add进父结点的孩子链表中。createTree可以按照定义一次性生成树，只需传入结点信息的表和对应的孩子链表就行，复杂度也是O(n)。childrenFromNode获得某个结点的所有孩子，这就比双亲表示法好点了，它没有遍历所有结点也无需进行if判断，而仅仅将该结点的孩子链表中的内容（整型值）转换成Node对象返回而已。不过该实现要获取某个结点的父结点就没有双亲法好了，孩子表示法必须遍历所有结点，复杂度为O(n)。获取父结点方法中，遍历所有结点，如果有某个结点的孩子链表中包含了所求结点，则返回该结点。

　　求树的深度方法也变成了递归实现，在双亲法实现中由于存在parent域，所以从下至上查找比较方便；而在孩子表示法中，获取孩子结点比较方便，所以从根结点开始从上至下查找，这里使用到了递归的思想。由于返回的是max + 1（为什么是这个值后面会解释），所以需要对树空的情况进行正确的处理。若是叶子结点，循环不会执行，应该返回max + 1 = 1，正确；其他情况，该结点有孩子，进入循环开始递归，递归直到遇到叶子结点停止，开始返回，叶子结点返回1。回到父结点的nodeDepth函数，max被赋值为1。现在说说这个max到底是什么意思，代码中for (int i: node.children)，遍历当前结点的所有孩子，它们共享同一个max，所以max的意义就是某结点所有孩子结点的深度的最大值。于是max + 1就是当前结点的深度。接着说，函数一直返回，每次返回实际就是往上一层，到所求结点的孩子结点处，其孩子结点中的最大深度赋值给max，那么最后返回的max + 1就是所求结点作为根结点时的子树深度。

　　**孩子表示法的优化**

　　再说获取某结点父结点的方法，从代码看出它遍历了所有结点。如果要改进，可以将双亲表示法融合进去，增加一个parent域就行。也就是说，Node类改成如下就行，这种实现可以称为双亲孩子表示法。

　　public static class Node<T> {

　　 private int parent;

　　 private T data;

　　 private List<Integer> children;

　　}

　　这样获取父结点的复杂度就变成了O(1)，就懒得实现了，稍微改改代码就好了。

　**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，也可能两者都有，或者两者都没。基于这种思想，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。特别注意的是，双亲表示法和孩子结点表示法，都使用了数组存放每一个结点的信息，若稍加分析，使用数组是有必要的。但在这种结构中，我们摒弃了数组，根结点可以作为头指针，以此开始可以遍历到树的全部结点——根结点肯定是没有兄弟的（根结点如果有兄弟这棵树就有两个根结点了），如果它没有孩子，则这棵树只有根结点；若有孩子，就如下图，它的nextChild的指针域就不为空，现在看这个左孩子，有兄弟（实际就是根结点的第二个孩子）还有孩子，则左孩子的两个指针域都不为空，再看这个左孩子的nextSib，他有个孩子...一直这样下去，对吧，能够访问到树的全部结点的。



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。这种结构也能直观反映树的结构的，上图其实就是下面这棵树。



　　说了这么多，反正把它当链表就行了，就是多了一个指针域而已。（和双向链表区别开，双向链表是a.next =b，必然有b.prev = a;但是这里二叉链表却没有这个限制，它指向任意一个结点都可以）。

　　好了现在来实现吧！

　　package Chap6;

　　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;

　　 }

　　 public Node(T data) {

　　 this.data = data;

　　 }

　　 @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;

　　 }

　　 public static void main(String[] args) {

　　 TreeChildSib<String> tree = new TreeChildSib<>("A");

　　 TreeChildSib.Node<String> root = tree.root();

　　 tree.addChild("B", root);

　　 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(root);

　　 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());

　　 }

　　}

　　/\* Outputs

　　Node{data=A, nextChild=B, 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

　　\*/

　　由于有的方法需要遍历树的所有结点，所以自建一个表List<Node<Item>> nodes来存放，具体来说就是每次添加结点的同时将这个结点加入到该表中。

　　addChild方法很重要，如果需要依附的父结点还没有孩子（if分支），那需要添加的结点成为它的第一个孩子；如果父结点有孩子了（else分支），那么就从父结点的第一个孩子开始，一直到它最后一个兄弟之后，新添加的结点位于此处。

　　获取某结点的所有孩子childrenFromNode方法，就是从所求结点的第一个孩子开始，不断找到其兄弟，第一个孩子与其所有兄弟全部就是所求结点的所有孩子。

　　求度，求深度的算法和孩子表示法差不多，就不再赘述。来看clear清空树的方法，定义了一个删除子树的方法，将子树的信息清空，同时nodes里面存放的结点也移除。再传入root就是清空整棵树了，但是只是root的信息被清空了，root本身还不是null，所以手动清空一下。

　　若是想让获取父结点变得方便些，也可以多设置一个parent域，见孩子表示法的优化。

孩子兄弟表示法有一个优点，可以将一棵普通树转化成二叉树，由于二叉树有诸多特征，使得处理起来变得简单。孩子兄弟表示法上面的那个链表，稍微拉伸下改变下结构，就能变成一棵二叉树，如下。

