Java之美[从菜鸟到高手演变]之数据结构基础之树、二叉树

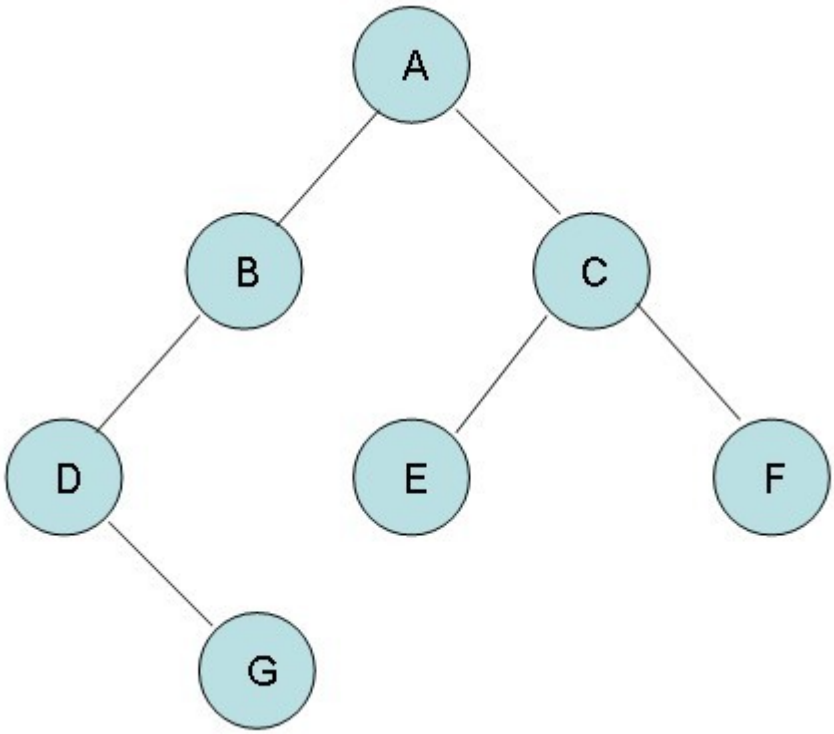
这里提到了广义表，所以说一下。

1、二叉树的建立

首先，我们采用广义表建立二叉树（关于广义表的概念，请查看百科的介绍：http://baike.baidu.com/view/203611.htm ）

我们建立一个字符串类型的广义表作为输入：

String expression = "A(B(D(,G)),C(E,F))";与该广义表对应的二叉树为：



写代码前，我们通过观察二叉树和广义表，先得出一些结论：

- 每当遇到字母，将要创建节点
- 每当遇到“（”，表面要创建左孩子节点
- 每当遇到“，”，表明要创建右孩子节点
- 每当遇到“）”，表明要返回上一层节点
- 广义表中“（”的数量正好是二叉树的层数

根据这些结论，我们基本就可以开始写代码了。首先建议一个节点类（这也属于一种自定义的数据结构）。

```
package com.xtfggef.algo.tree;

public class Node {
    private char data;

    private Node lchild;
```

```
    private Node rchild;

    public Node(){
    }

    public char getData() {
        return data;
    }

    public void setData(char data) {
        this.data = data;
    }

    public Node getRchild() {
        return rchild;
    }

    public void setRchild(Node rchild) {
        this.rchild = rchild;
    }

    public Node getLchild() {
        return lchild;
    }

    public void setLchild(Node lchild) {
        this.lchild = lchild;
    }

    public Node(char ch, Node rchild, Node lchild) {
        this.data = ch;
        this.rchild = rchild;
        this.lchild = lchild;
    }

    public String toString() {
        return "" + getData();
    }
}
```

根据广义表创建二叉树的代码如下：

```
public Node createTree(String exp) {
        Node[] nodes = new Node[3];
        Node b, p = null;
        int top = -1, k = 0, j = 0;
        char[] exps = exp.toCharArray();
        char data = exps[j];
        b = null;
        while (j < exps.length - 1) {
            switch (data) {
            case '(':
                top++;
```

```
                nodes[top] = p;
                k = 1;
                break;
            case ')':
                top--;
                break;
            case ',':
                k = 2;
                break;
            default:
                p = new Node(data, null, null);
                if (b == null) {
                    b = p;
                } else {
                    switch (k) {
                    case 1:
                        nodes[top].setLchild(p);
                        break;
                    case 2:
                        nodes[top].setRchild(p);
                        break;
                    }
                }
            }
            j++;
            data = exps[j];
        }
        return b;
}
```

思路不难，结合上述的理论，自己断点走一遍程序就懂了！

2、二叉树的递归遍历

二叉树的遍历有三种：先序、中序、后序，每种又分递归和非递归。递归程序理解起来有一定的难度，但是实现起来比较简单。对于上述二叉树，其：

a 先序遍历

A B D G C E F

b 中序遍历

D G B A E C F

c 后序遍历

G D B E F C A

先、中、后序递归遍历如下：

```
/**
     * pre order recursive
     *
     * @param node
     */
```

```
    public void PreOrder(Node node) {
        if (node == null) {
            return;
        } else {
            System.out.print(node.getData() + " ");
            PreOrder(node.getLchild());
            PreOrder(node.getRchild());

        }
    }

    /**
     * in order recursive
     *
     * @param node
     */
    public void InOrder(Node node) {
        if (node == null) {
            return;
        } else {
            InOrder(node.getLchild());
            System.out.print(node.getData() + " ");
            InOrder(node.getRchild());
        }
    }

    /**
     * post order recursive
     *
     * @param node
     */
    public void PostOrder(Node node) {
        if (node == null) {
            return;
        } else {
            PostOrder(node.getLchild());
            PostOrder(node.getRchild());
            System.out.print(node.getData() + " ");
        }
    }
```

二叉树的递归遍历实现起来很简单，关键是非递归遍历有些难度，请看下面的代码：

3、二叉树的非递归遍历

先序非递归遍历：

```
public void PreOrderNoRecursive(Node node) {
        Node nodes[] = new Node[CAPACITY];
        Node p = null;
        int top = -1;
        if (node != null) {

            top++;
```

```
            nodes[top] = node;
            while (top > -1) {
                p = nodes[top];
                top--;
                System.out.print(p.getData() + " ");
                if (p.getRchild() != null) {
                    top++;
                    nodes[top] = p.getRchild();
                }
                if (p.getLchild() != null) {
                    top++;
                    nodes[top] = p.getLchild();
                }
            }
        }
    }
```

原理：利用一个栈，先序遍历即为根先遍历，先将根入栈，然后出栈，凡是出栈的元素都打印值，入栈之前top++，出栈之后top--，利用栈后进先出的原理，右节点先于左节点进栈，根出栈后，开始处理左子树，然后是右子树，读者朋友们可以自己走一遍程序看看，也不算难理解！

中序非递归遍历

```
public void InOrderNoRecursive(Node node) {
        Node nodes[] = new Node[CAPACITY];
        Node p = null;
        int top = -1;
        if (node != null)
            p = node;
        while (p != null || top > -1) {
            while (p != null) {
                top++;
                nodes[top] = p;
                p = p.getLchild();
            }
            if (top > -1) {
                p = nodes[top];
                top--;
                System.out.print(p.getData() + " ");
                p = p.getRchild();
            }
        }
    }
```

原理省略。

后续非递归遍历：

```
public void PostOrderNoRecursive(Node node) {
        Node[] nodes = new Node[CAPACITY];
        Node p = null;
        int flag = 0, top = -1;
        if (node != null) {
```

```
            do {
                while (node != null) {
                    top++;
                    nodes[top] = node;
                    node = node.getLchild();
                }
                p = null;
                flag = 1;
                while (top != -1 && flag != 0) {
                    node = nodes[top];
                    if (node.getRchild() == p) {
                        System.out.print(node.getData() + " ");
                        top--;
                        p = node;
                    } else {
                        node = node.getRchild();
                        flag = 0;
                    }
                }
            } while (top != -1);
        }
    }
```