树和二叉树

本笔记内容来自于李刚《Java程序员的基本修养》

Author: Yi

Date: Dec/30/2015

- 树的概念和存储实现
- 二叉树的概念和实现
- 前序，中序，后序，BFS遍历树

1: 树的概念

树是一种非线性表，元素之间的对应关系是一对多。
树的基本操作：

- 初始化一棵树
- 返回一颗树的深度
- 添加节点到一棵树中
- 返回指定位置的节点数据
- ……

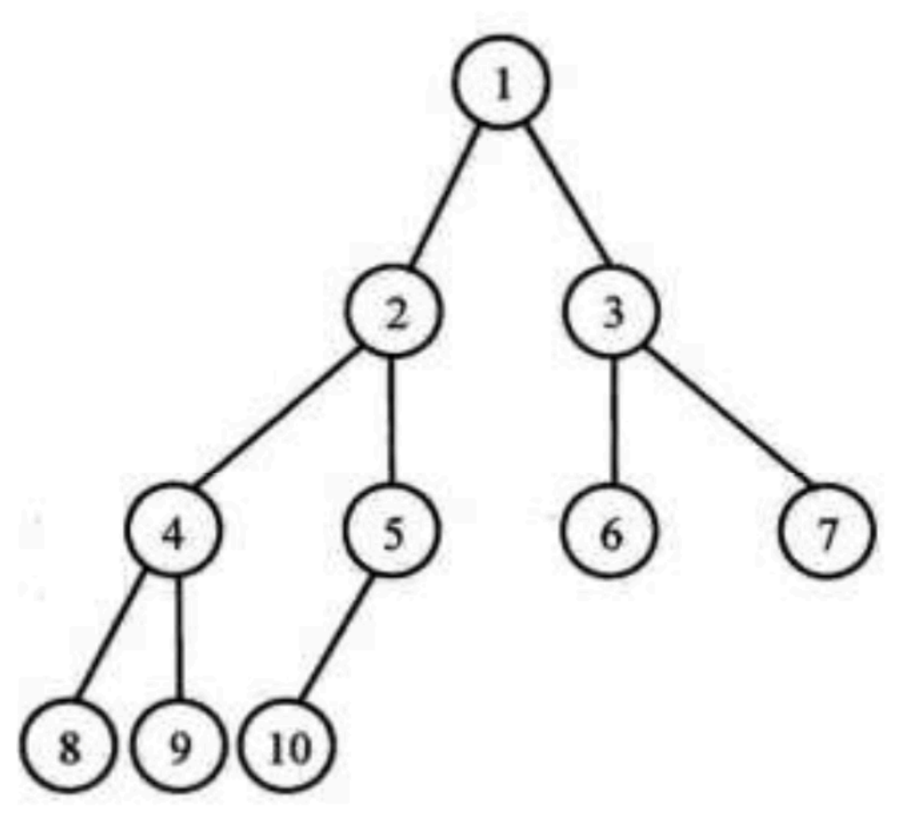
树的存储实现

- 父节点表示法
  每个节点包含两部分信息，一部分是当前节点的数据信息，另一部分是父节点的位置。这种实现方式可以很方便的找到一个节点的父节点，但是如果如果要找到某一个节点的字节点就不是那么方便。
- 子节点链表示法
  字节点链表示法是在每一个节点后面都维护一个该节点的字节点链表，因此对于每个字节点来说很容易找到它的字节点，但是要找到节点的父节点就比较麻烦。

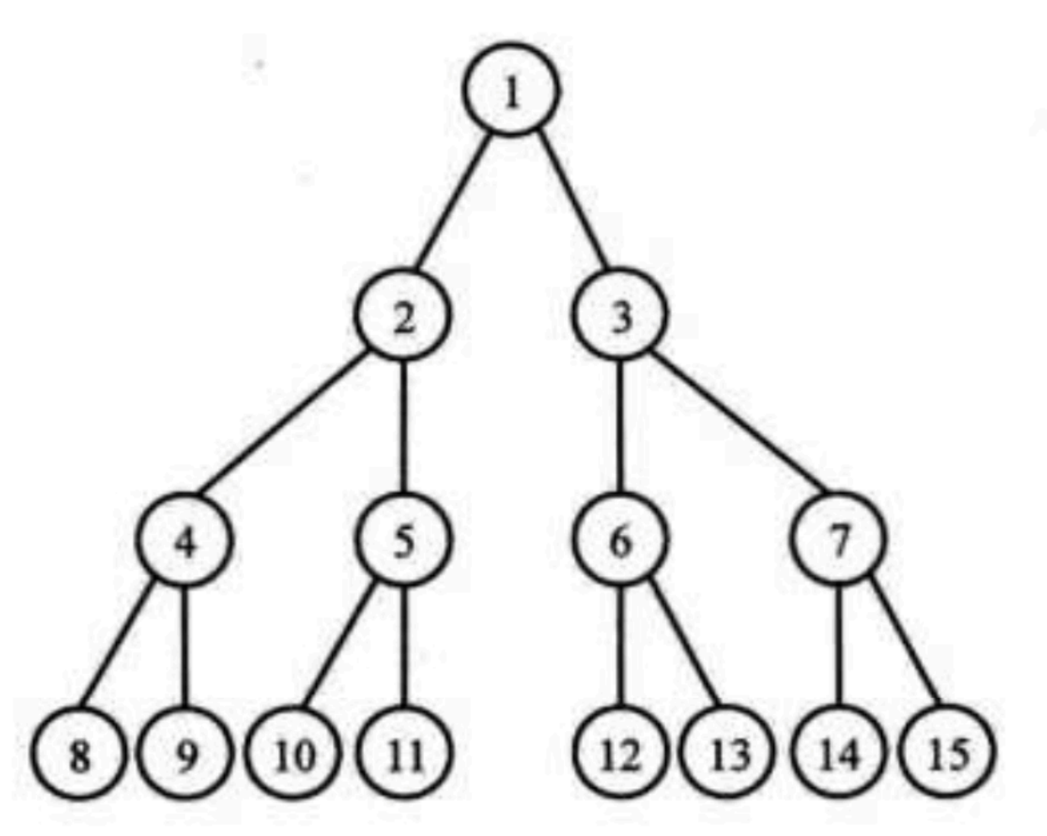
2: 二叉树

普通的树由于字节点数不定，所以在操作上会出现很大的困难。因此我们常用的就是二叉树，也即对于树中的任意一个节点，只有两个字节点，一个称为左节点，一个称为右节点。且二叉树多左右节点不能调换，是有序的。

- 完全二叉树：对于完全二叉树而言，除了最后一层的叶子节点之外，每一层的节点都是满节点；且最后一层的叶子节点或满或右边连续缺节点；



- 满二叉树：满二叉树就是在完全二叉树的基础上保证了最后一层的节点也是满节点；



二叉树常用的操作：

- 初始化一棵二叉树
- 为指定节点添加字节点
- 判断二叉树是否为空
- 返回指定节点的父节点
- 返回二叉树深度
- 返回指定节点位置
- ……

根据二叉树的常用操作，二叉树的实现方式有一下三种：

- 顺序存储：用一个数组维护一棵树；
- 二叉链表存储(常用)：维护一个保留其左右子节点的节点；
- 三叉链表存储：维护一个保留其左右子节点和父节点的节点；
  下面分三部分来讲讲这三种实现方法：

2.1: 顺序存储

在上面的介绍中提到，顺序存储是用一个数组进行维护的，而且根据二叉树的特性我们知道对于每一层的节点数最大值是：

$$n = 2^k (k >= 0)$$

因此对于一棵二叉树而言，其最大数的节点数是：

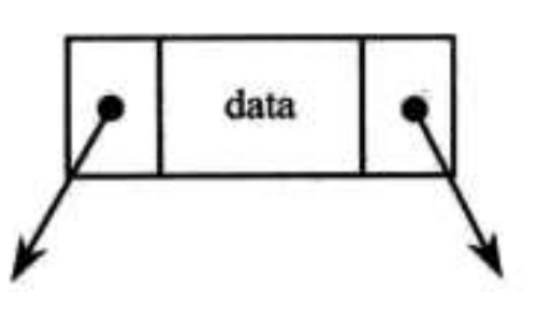
$$n = 2^i - 1(i\text{是树的高度})$$

所以我们可以定义一个长度为$2^i - 1$的数组即可。顺序存储是按照完全二叉树中的顺序来存放，因此如果有节点不存在，在数组对应的位置上设置为null即可。因此对于这种实现方式，数组利用率不高，会存在大量浪费，尤其是当二叉树的为全右二叉树时。但是其优点就是查询访问非常高效，因为是数组实现的。

2.2: 二叉链表存储

用一个Node来表示一个节点，在节点中存放当前节点数据和其左右节点地址。看下面代码：

```java
class Node {
    Object data;
    Node left;
    Node right;
}
```

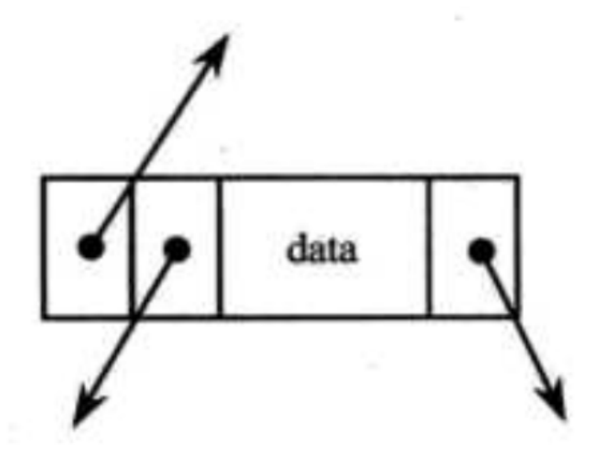


在之前章节我们就看到，一般由链表实现的插入和删除都比较方便。对于这里也是一样，插入新节点只要将原来树中的节点左右中任意一个指向它即可。

2.3: 三叉链表存储

因为在二叉链表存储中，存在一个缺陷就是要得知一个节点的父节点时，要遍历整棵树才能确定，但是在三叉链表存储中，不仅存放了当前节点数据，左右节点地址，还存放了其父节点的地址，这样对于树中的任意一个节点，可以很轻易的找到它的父节点。看下面代码：

```java
class Node {
    Object data;
    Node left;
    Node right;
    Node parent;
}
```



3: 遍历二叉树

遍历二叉树就是将树中的元素按照线性一定的顺序输出来。按照访问算法的不同，可以分为BFS(Breadth First Search)和DFS(Depth First Search).

- 广度优先搜索，又叫做层序遍历
- 深度优先搜索
  - 前序遍历(pre-order)
  - 中序遍历(in-order)
  - 后序遍历(post-order)

前序，中序和后序针对的都是根节点而言的。L(左节点), D(根节点), R(右节点)。因此前序为DLR；中序为LDR；后序为LRD。

先看看深度遍历：

3.1: 前序遍历

前序遍历的顺序是根节点，左节点，右节点。上代码（递归实现）：
规则：

1. 先访问根节点
2. 递归访问左节点
3. 递归访问右节点

```java
public List<Node> preOrder(Node root) {
    return helpler(root);
}

private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    list.add(root);
    if (root.left != null) {
        list.addAll(helper(root.left));
    }
    if (root.right != null) {
        list.addAll(helper(root.right));
    }
}
```

上代码，递归实现：

```java
public List<Node> preOrder(Node root) {
    return helper(root);
}

private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    Stack<Node> stack = new Stack<Node>();
    Node cur = root
    while (cur != null && !stack.isEmpty()) {
        while (cur != null) {
            list.add(cur);
            stack.push(cur);
            cur = cur.left;
        }
        cur = stack.pop();
        cur = cur.right;
    }
    return list;
}
```

3.2: 中序遍历

规则：

- 先访问左节点
- 访问根节点
- 访问右节点

上代码，递归版：

```java
public List<Node> inOrder(Node root) {
    return helper(root);
}
private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    if (root.left != null) {
        list.addAll(helper(root.left));
    }
    list.add(root);
    if (root.right != null) {
        list.addAll(helper(root.right));
    }
}
```

上代码，递归实现：

```java
public List<Node> preOrder(Node root) {
    return helper(root);
}

private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    Stack<Node> stack = new Stack<Node>();
    Node cur = root
    while (cur != null && !stack.isEmpty()) {
        while (cur != null) {
            stack.push(cur);
            cur = cur.left;
        }
        cur = stack.pop();
        list.add(cur);
        cur = cur.right;
    }
    return list;
}
```

3.3: 后序遍历

规则：

- 访问左节点
- 访问右节点
- 访问根节点

上代码，递归版：

```java
public List<Node> inOrder(Node root) {
    return helper(root);
}
private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    if (root.left != null) {
        list.addAll(helper(root.left));
    }
    if (root.right != null) {
        list.addAll(helper(root.right));
    }
    list.add(root);
}
```

3.4: 层序遍历（广度优先搜索）

上代码，用队列实现：

```java
public List<Node> levelTraversal(Node root) {
    return helper(root);
}

private List<Node> helper(Node root) {
    List<Node> list = new ArrayList<Node>();
    Queue<Node> queue = new LinkedList<Node>();
    queue.offer(root);
    while (!queue.isEmpty()) {
            Node cur = queue.poll();
            list.add(cur);
            if (cur.left != null) queue.offer(cur.left);
            if (cur.right != null) queue.offer(cur.right);
        }
    return list;
}
```