二叉树

二叉树（Binary Tree）由n个节点组成的有限集合，由一个根节点和两颗互不相交的左子树和右子树的子二叉树构成。

性质

基本

1.若根节点层次为1，则第i层最多有2^（i-1）个结点。

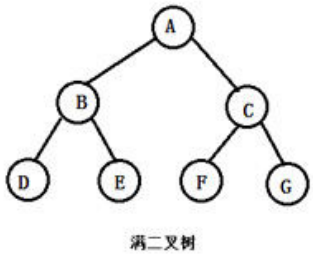
2.高度为h的树中，最多有2^(h-1)

> Q：度为5的二叉树结点最多和最少有多少个?
>
> A：min：5，max：31

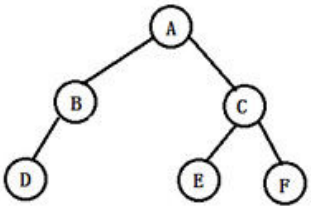
3.二叉树中的叶子结点数为n0，度为2的结点数为n2，则n0=n2+1.

> why？：
>
> **n结 = n0+n1+n2**
>
> **b（分支数） = n（结点总数）-1** `//除了根节点外的结点必向上有一个分支。`
>
> **b（分支数）= n1 + 2n2**
>
> **n（结点总数）-1 = n1 + 2n2** = **n0+n1+n2-1**
>
> **n0 = n2 + 1**

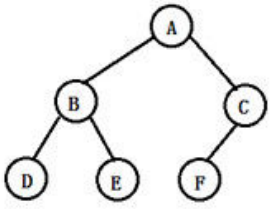
满二叉树和完全二叉树



满二叉树



非完全二叉树



完全二叉树

定义

1. 满二叉树，每一层结点数目都达到最大值，拥有2^h-1个结点的二叉树。
2. 完全二叉树，有n个结点高度为h的二叉树，每一个节点都和高度为h的满二叉树中的序号对应。

性质

1. 完全二叉树的前k-1层是满的，**h = |_log2 k_|+1**，（这个公式接相当于向上取整。）
2. 若i=0，则i为根节点，无父母节点；若i>0，则i的父母节点序号为**|_(i-1)/2_|**
3. i的左孩子：**2i+1**
4. i的右孩子：**2i+2**

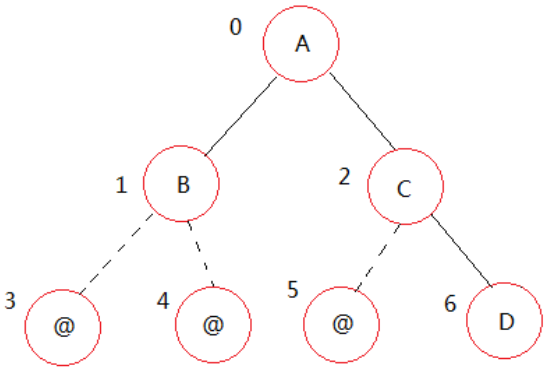
遍历

1. 先根次序：根 左 右
2. 中根次序：左 根 右
3. 后根次序：左 右 根

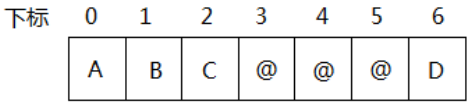
二叉树基本抽象数据类型

线性结构

通过上面的的规律我们可以实现使用线性表操作二叉树，但是存在严重的空间浪费，每一个不符合满二叉树序号的结点都需要补一个null。

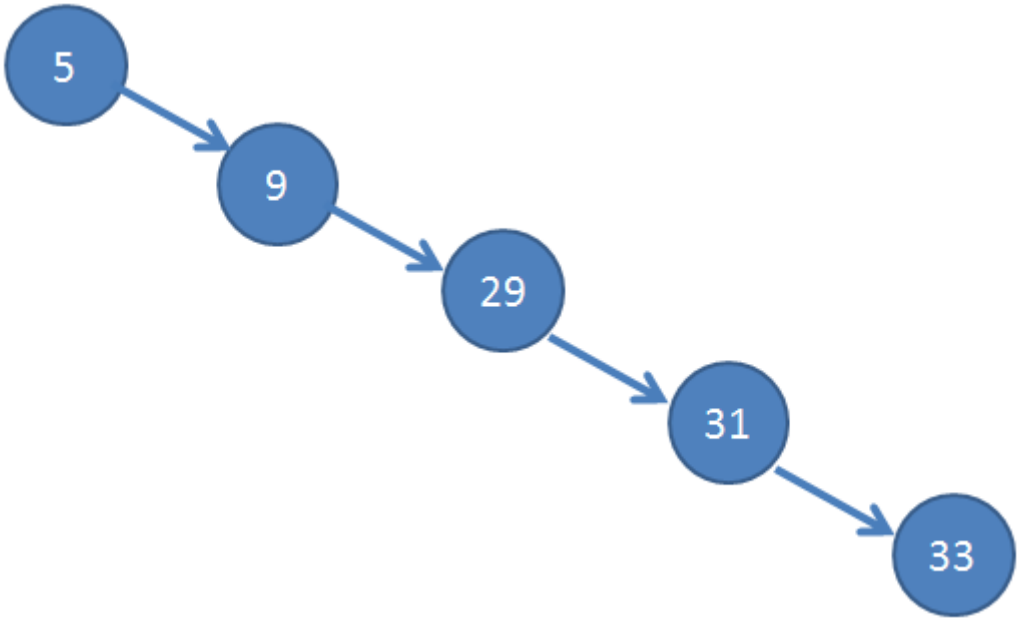


添加实际不存在的虚结点“@”



顺序存储结构

其中右子树空间最为浪费。



链式结构

结点类

```
public class BinaryNode<T> {
    public T data;
    public BinaryNode<T> left,right;
    public BinaryNode(T data,BinaryNode<T> left,BinaryNode<T> right){
        this.data = data;
        this.left = left;
```

```
        this.right = right;
    }
    public BinaryNode(T data){
        this(data,null,null);
    }
    public String toString(){
        return this.data.toString();
    }
    public boolean isLeaf(){
        return this.left == null && this.right == null;
    }
}
```

构造

```
public class BinaryTree<T> {
    public BinaryNode<T> root; //根节点;
    public BinaryTree(){
        this.root = null;
    }
    public boolean isEmpty(){
        return this.root == null;
    }
}

```

插入

```
   public BinaryNode<T> insert(T x){
        //在上面插，原根节点默认作为左子树。
        return this.root = new BinaryNode<T>(x,this.root,null);
    }

    public BinaryNode<T> insert(T x,BinaryNode<T> parent,boolean LeftChild)
{
        //在下面插则可能在做也可能在右。
        if(x == null){
            return null;
        }
        if(LeftChild == true){
            return parent.left = new BinaryNode<T>(x,parent.left,null);
        }else{
            return parent.right = new BinaryNode<T>(x,parent.right,null);
        }
    }
```

删除

```java
    public void remove(BinaryNode<T> parent,boolean leftChild){
        if(leftChild){
            parent.left = null;
        }else{
            parent.right = null;
        }
    }

    public void clear(){
        this.root = null;
    }
```

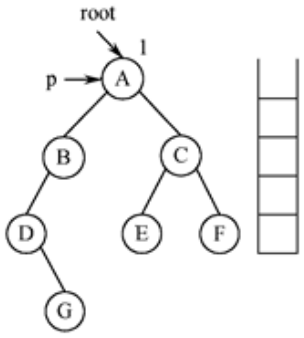
遍历

> 1. 先根次序：根 左 右
> 2. 中根次序：左 根 右
> 3. 后根次序：左 右 根

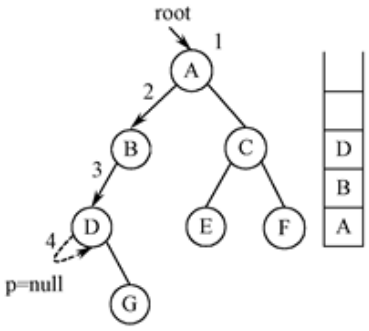
递归

```java
    public void preOrder(BinaryNode<T> p){
        if( p!=null ){
            System.out.println(p.data + " ");
            preOrder(p.left);
            preOrder(p.right);
        }
    }

    public void inOrder(BinaryNode<T> p){
        if( p!=null ){
            inOrder(p.left);
            System.out.println(p.data + "");
            inOrder(p.right);
        }
    }

    public void postOrder(BinaryNode<T> p){
        if( p!=null ){
            postOrder(p.left);
            postOrder(p.right);
            System.out.println(p.data + "");
        }
```

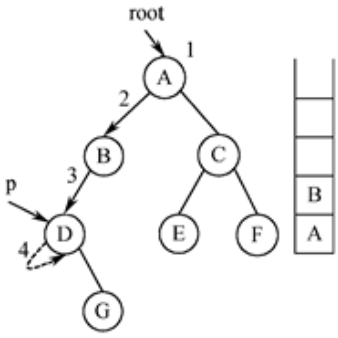
孩子优先



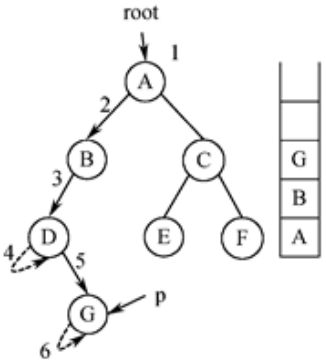
（a）p从根结点开始，空栈



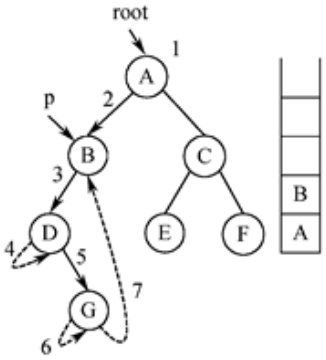
（b）p沿着left链进入左子树，结点入栈，当p==null时，p返回栈顶结点D



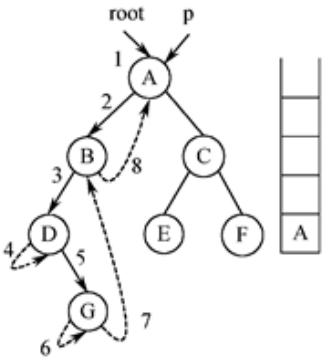
（c）p返回D结点，访问D结点



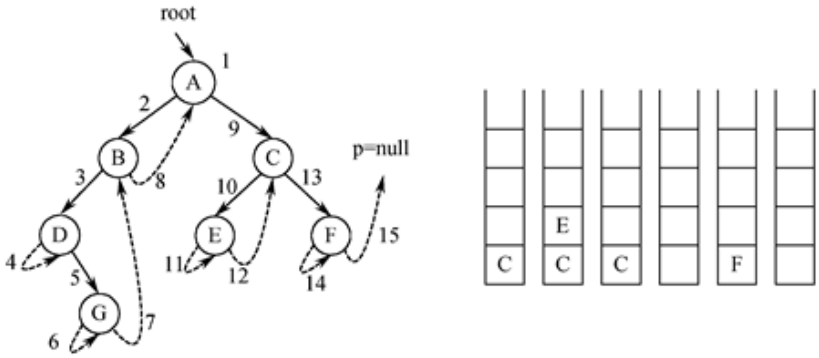
（d）进入D的右子树，G结点入栈，p进入G的左子树，p==null，p再返回栈顶结点G



（e）访问G，p返回栈顶结点B，访问B结点



（f）p返回顶栈结点A，访问A

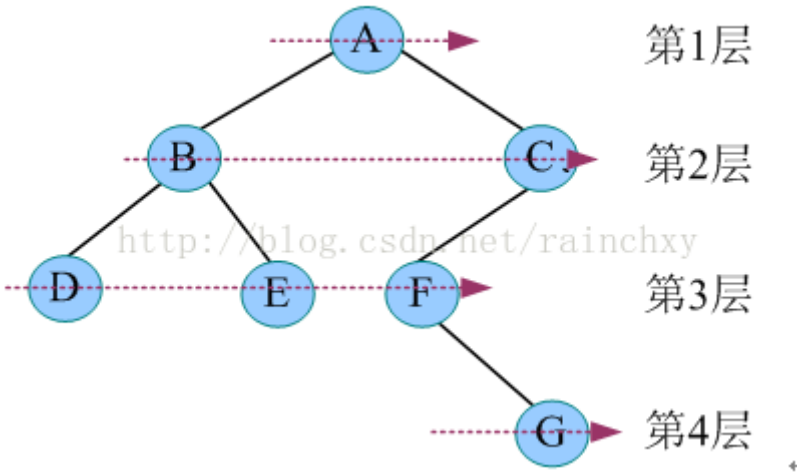


（g）继续按中根次序遍历A的右子树，栈随之变化；访问完所有结点之后，p==null，栈为空

```java
    public void preOrderTraverse2(TreeNode root) {
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode pNode = root;
        while (pNode != null || !stack.isEmpty()) {
            if (pNode != null) {
                System.out.print(pNode.val+"  ");
                stack.push(pNode);
                pNode = pNode.left;
            } else { //pNode == null && !stack.isEmpty()
                TreeNode node = stack.pop();
                pNode = node.right;
            }
        }
    }
```

层级遍历

层层遍历，先进先出，但是出去之前要吧自己的孩子怼到队尾。




```
    public void levelLoader(){
        System.out.print("层次遍历的顺序为：");
        LinkedQueue<BinaryNode<T>> queue = new LinkedQueue<BinaryNode<T>>
();
        BinaryNode<T> p = this.root;
        while(p!=null){
            System.out.println(p.data+" ");
            if(p.left != null) queue.add(p.left);
            if(p.right != null) queue.add(p.right);
            p = queue.poll();
        }
    }
```

完整代码

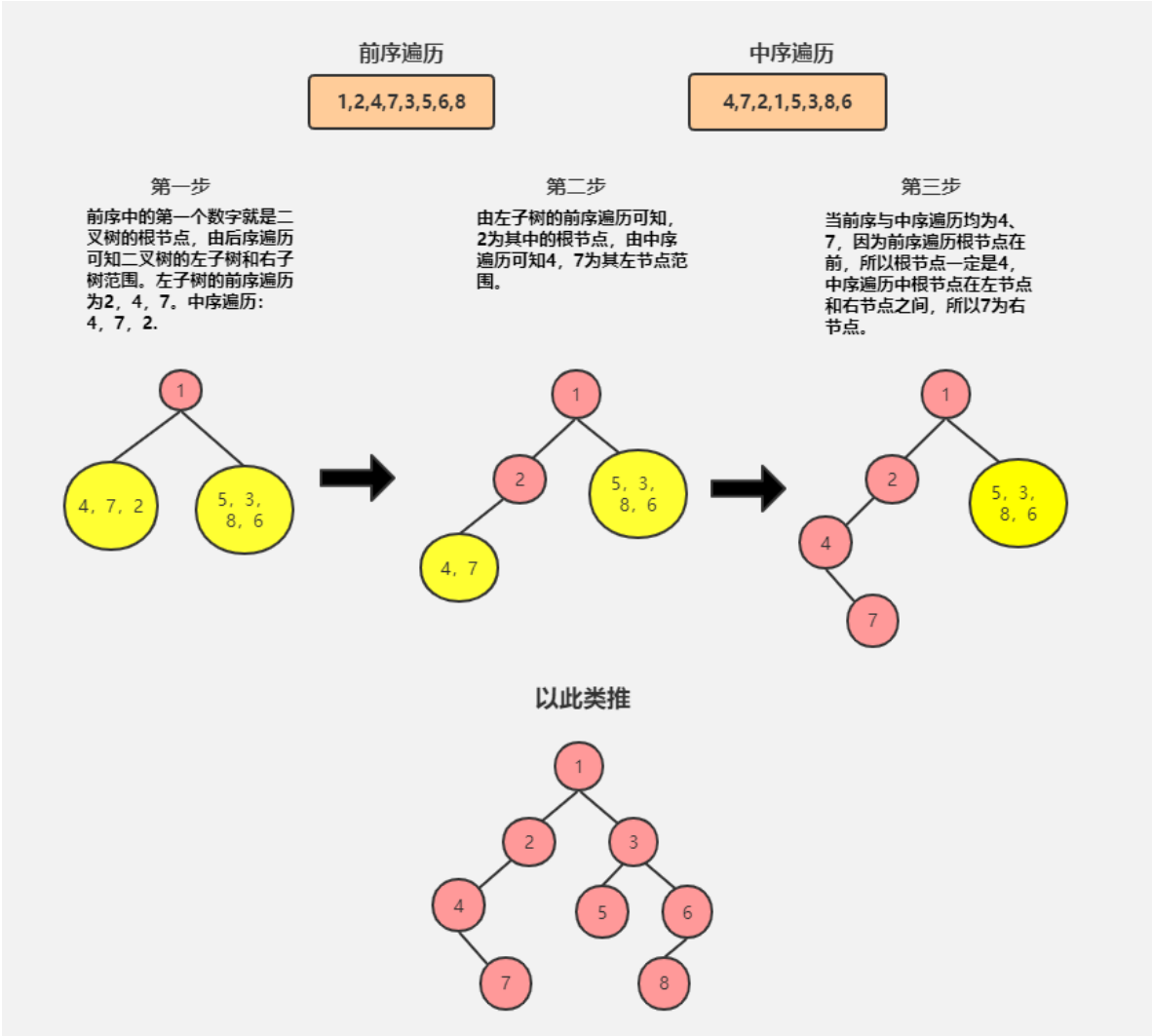
```
public class BinaryTree<T> {
    public BinaryNode<T> root; //根节点;
    public BinaryTree(){
        this.root = null;
    }
    public boolean isEmpty(){
        return this.root == null;
    }

    public BinaryNode<T> insert(T x){
        //在上面插，原根节点默认作为左子树。
        return this.root = new BinaryNode<T>(x,this.root,null);
    }

    public BinaryNode<T> insert(T x,BinaryNode<T> parent,boolean LeftChild)
{
        //在下面插则可能在做也可能在右。
        if(x == null){
            return null;
        }
        if(LeftChild == true){
            return parent.left = new BinaryNode<T>(x,parent.left,null);
        }else{
            return parent.right = new BinaryNode<T>(x,parent.right,null);
        }
    }
```

```
26
27      public void remove(BinaryNode<T> parent,boolean leftChild){
28          if(leftChild){
29              parent.left = null;
30          }else{
31              parent.right = null;
32          }
33      }
34
35      public void clear(){
36          this.root = null;
37      }
38
39      public void preOrder(BinaryNode<T> p){
40          if( p!=null ){
41              System.out.println(p.data + " ");
42              preOrder(p.left);
43              preOrder(p.right);
44          }
45      }
46
47      public void inOrder(BinaryNode<T> p){
48          if( p!=null ){
49              inOrder(p.left);
50              System.out.println(p.data + "");
51              inOrder(p.right);
52          }
53      }
54
55      public void postOrder(BinaryNode<T> p){
56          if( p!=null ){
57              postOrder(p.left);
58              postOrder(p.right);
59              System.out.println(p.data + "");
60          }
61      }
62
63  }
64
```

构造二叉树

推断树的结构

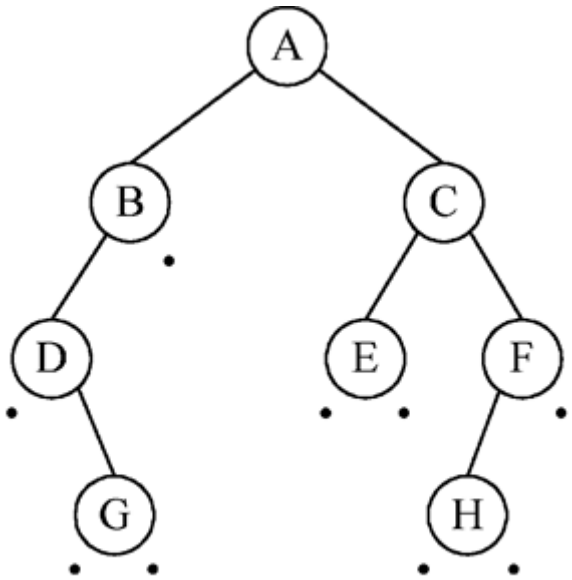
如何通过先序和中序遍历/（中根顺序和后根顺序）推出二叉树结构?



> 1 .首先根据前序遍历我们可以推出来根节点和其左子树的根节点，并且能结合中序遍历推出左右子树元素。
>
> 2 .左子树剩下4，7的时候我们发现，（4，7）中序遍历和先序遍历的顺序是一样的，所以我们这样分析
>
> > 先序：根 左　右
> >
> > 中序：左 根　右
>
> 3 .没有左的时候根和右在先序和中序中排列的顺序是一样的。所以4，7为中，右。
>
> 4.在之后我们可以推断出右子树的根节点 是 3。
>
> 5.而中序遍历中5在3的前面，所以5是3的左孩子。
>
> 6.而6，8在中序和先序中的顺序相反，那就是中和左。于是就推完了。

其实这些应该是代码实现的，总之我们现在推出了树的结构。

根据树的结构构造数据结构



（d）ABD.G...CE..FH...

二叉树表明空子树的先跟序列 prelist

> A B D ^ G ^ ^ ^ C E ^ ^ F H ^ ^ ^

用先根遍历的顺序结构表示完我们推算出的树的结构之后，我们就可以开始造树了。

造树

```
    public BinaryTree(T[] prelist){
        this.root = create(prelist);
    }

    private int i = 0;
    //外部变量，保证不会变量值不会因为递归而重置。
    private BinaryNode<T> create(T[] prelist){
        BinaryNode<T> p = null;
        if(i < prelist.length){
            T ele = prelist[i];
            i++;
            if(ele != null){        //是null就不会创造分支。
                p = new BinaryNode<T>(ele);
                p.left = create(prelist);
                p.right = create(prelist);
            }
        }
        return p;
    }
```

首先我们的树的结构是占空位的先根顺序，那么我们完全可以使用之前的先根遍历的递归方法来造树，

> 值得注意的是：
>
> - prelist 在实际编码的时候，“^”要写成"null",因为T不一定String。
> - 嵌套的p不会对外层p的指向造成影响。

线索二叉树

抽象数据类型ADT

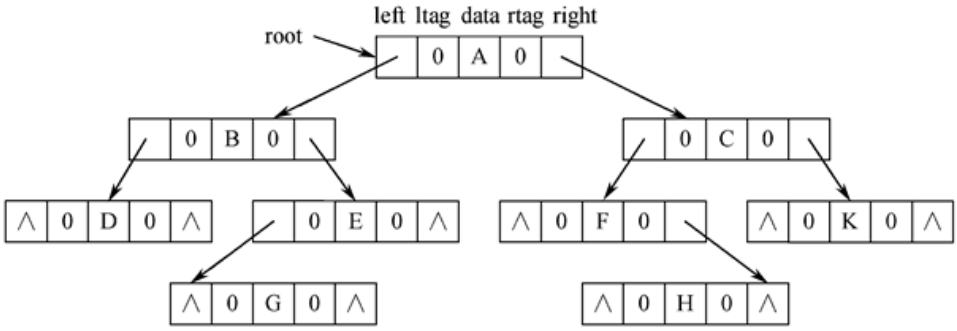
线索节点类

线索节点的线索成员变量用来负责让二叉树判断左右节点地址。

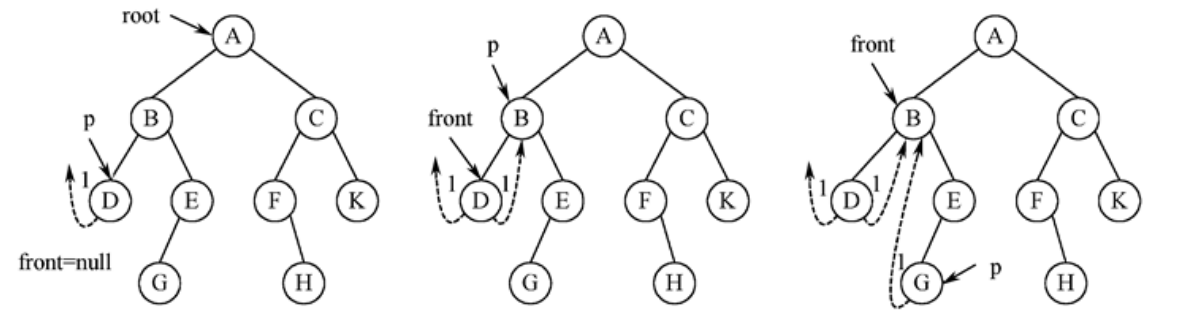
```
package DS2;

public class ThreadBinaryNode<T> {
        public T data;
        public ThreadBinaryNode<T> left,right;
        public boolean ltag,rtag;
        public ThreadBinaryNode(T data,ThreadBinaryNode<T>
left,ThreadBinaryNode<T> right,boolean ltag,boolean rtag){
            this.data = data;
            this.left = left;
            this.right = right;
            this.ltag = ltag;
            this.rtag = rtag;
        }
        public ThreadBinaryNode(T data){
            this(data,null,null,false,false);
        }
        public boolean isLeaft(){
            return this.left == null && this.right == null;
        }
        public String toString(){
            return this.data.toString();
        }
}

```

线索二叉树类

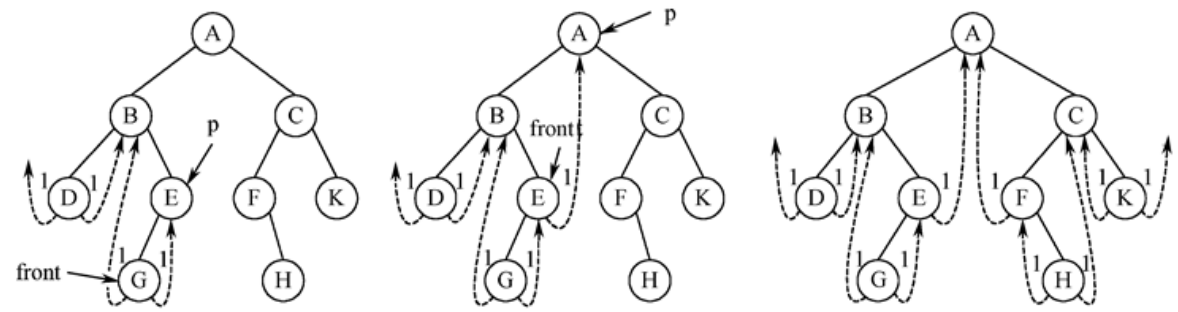
这破树真的好难理解啊



（a）已创建一棵二叉链表的二叉树



（b）D为第1个访问的结点，前驱为空　（c）访问B，设置前驱D的右线索　（d）访问G，设置G的左线索指向前驱B



（e）访问E，设置前驱G的右线索　（f）访问A，设置前驱E的右线索　（g）继续中序线索化A的右子树

构造函数（树的创建）

> 这里我们没有在外边创建全局变量，来保证变量的值稳定改变，而是根据完全二叉树的节点下标的规律来构建这颗树（线索二叉树不一定是完全二叉树，所以数组的残缺的地方一定要补null）。
>
> 左孩子：2*i+1，右孩子：2*i-1

```java
package DS2;

public class ThreadBinaryTree<T> {
    private ThreadBinaryNode<T> root;

    public ThreadBinaryTree(T[] array,int index){
        this.root = this.creatTree(array, index);
    }

    public ThreadBinaryNode<T> creatTree(T[] array,int index){
        ThreadBinaryNode<T> p = null;
        if(index<array.length){
            p = new ThreadBinaryNode<T>(array[index]);
            p.left = creatTree(array,2*index+1);
            p.right = creatTree(array,2*index+2);
            return p;
        }else{
```

```
            return null;
        }
    }
}

```

中序线索化二叉树

> 这里我们使用的是中序遍历，原因是为了让树快速索值到最左边的子叶节点。

```
    public ThreadBinaryNode<T> front = null;

    public void inorderThread(ThreadBinaryNode<T> p){
        if(p!=null){
            inorderThread(p.left);
            //要知道这是中序线索化树，所以说p一开始会索值到最左边的节点，而他的左线索树一定是false；
            if(p.left == null){
                p.ltag = true;
                p.left = front;
            }
            if(p.right == null){
                p.rtag = false;
            }
            if(front != null && front.rtag == true){
                front.right = p;
            }
            front = p;
            inorderThread(p.right);
        }
    }
```