

# 第一节 树

2016年8月31日 22:50

## 7.1.1 动机

Vector与List静态操作与动态操作的不可兼容性

## 7.1.2 应用

## 7.1.3 有根树

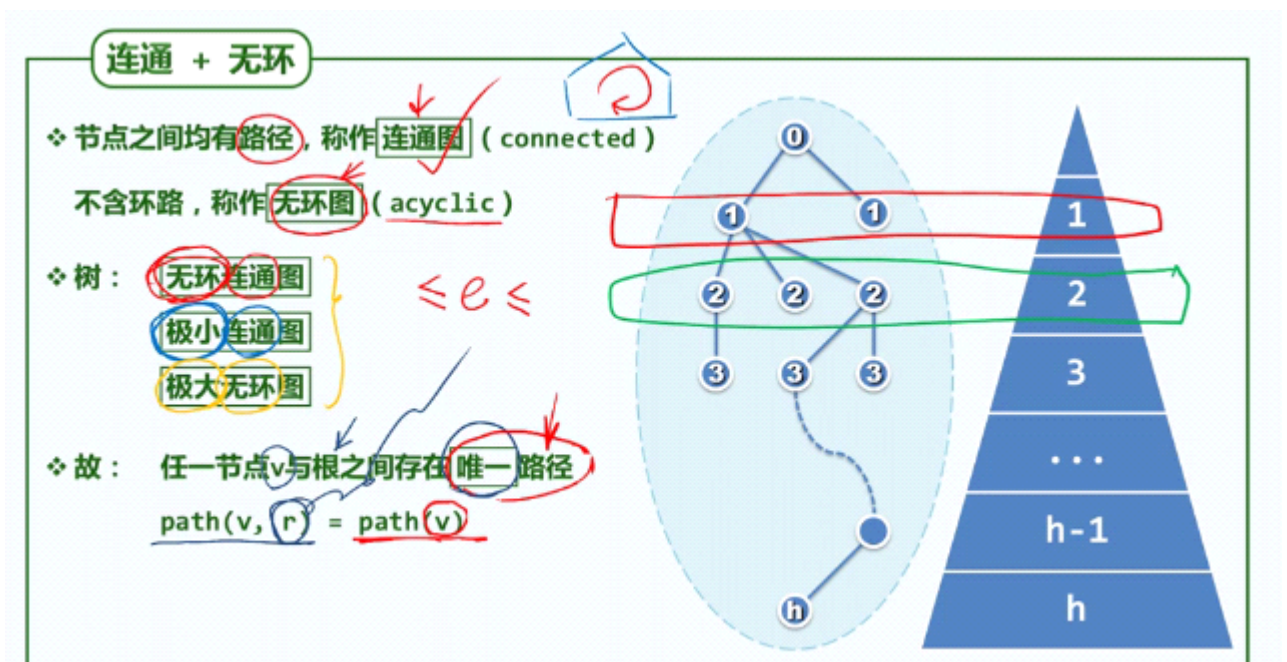
有根树之并仍为有根树，子树。

## 7.1.4 有序树

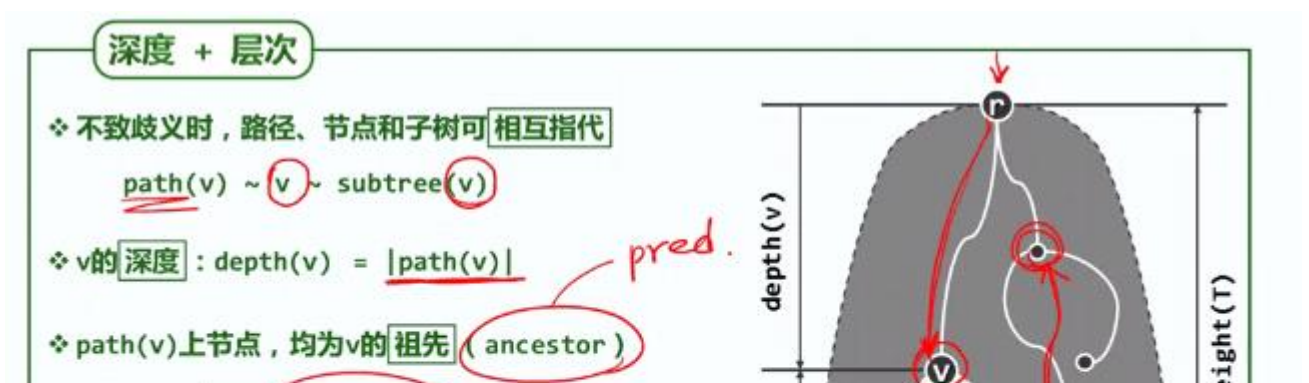
兄弟树之间有序。

## 7.1.5 路径path + 环路loop

## 7.1.6 连通 + 无环



## 7.1.7 深度 + 层次



## 深度 + 层次

❖ 不致歧义时，路径、节点和子树可相互指代

$$\text{path}(v) \sim v \sim \text{subtree}(v)$$

❖  $v$  的深度:  $\text{depth}(v) = |\text{path}(v)|$

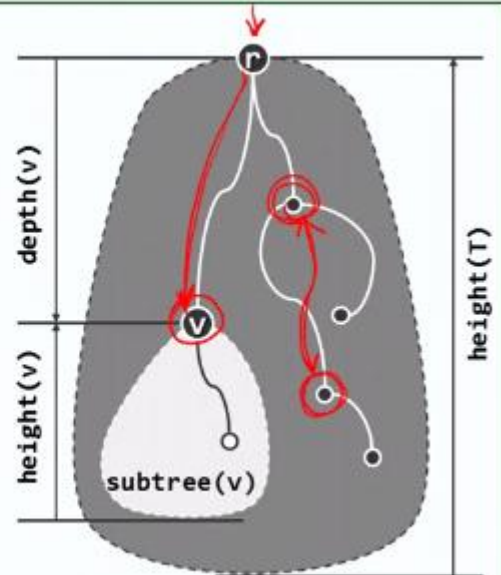
❖  $\text{path}(v)$  上节点，均为  $v$  的祖先 (ancestor)

$v$  是它们的后代 (descendant)

❖ 其中，除自身以外，是真 (proper) 祖先/后代

❖ 半线性: 在任一深度

$v$  的祖先/后代若存在，则必然/未必唯一



全树的高度、子树的高度、任一节点的高度

## 第二节 树的表示

2016年8月31日 23:21

### 7.2.1 表示法

接口操作

接口	
节点	功能
root()	✓ 根节点
parent()	父节点
<u>firstChild()</u>	长子
<u>nextSibling()</u>	兄弟
insert( <u>i</u> , <u>e</u> )	将e作为第i个孩子插入
remove( <u>i</u> )	删除第i个孩子 (及其后代)
→ traverse()	遍历

### 7.2.2 父亲

#### 父节点

❖ 空间性能:  $O(n)$  ✓

❖ 时间性能

- ☺  $parent(): O(1)$  ✓
- ☹  $root(): O(n)$  或  $O(1)$  ✓
- ☹  $firstChild(): O(n)$
- ☹  $nextSibling(): O(n)$

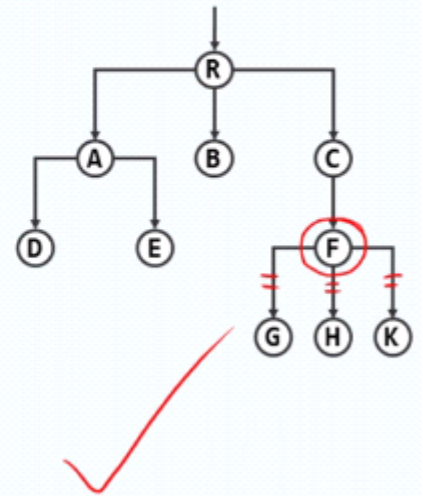
rank	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

查找长子或兄弟节点，效率不够高

### 7.2.3 孩子

## 孩子节点

	data	children
0	A	→ 3 → 5 → ^
1	B	^
2	C	→ 6 → ^
3	D	^
4	R	→ 0 → 1 → 2 → ^
5	E	^
6	F	→ 7 → 8 → 9 → ^
7	G	^
8	H	^
9	K	^

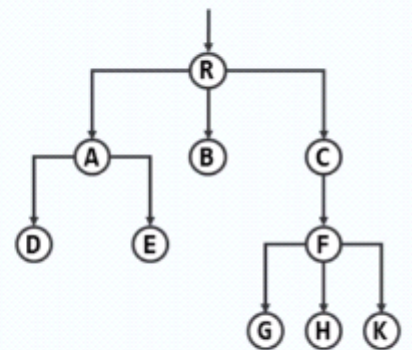


选择将所有孩子保存在父亲中的引用，解决了向下查找问题，带来了向上查找的问题。

### 7.2.4 父亲 + 孩子

## 父节点 + 孩子节点

	data	parent	children
0	A	4	→ 3 →
1	B	4	^
2	C	4	→ 6 → ^
3	D	0	^
4	R	-1	→ 0 → 1 → 2 → ^
5	E	0	^
6	F	2	→ 7 → 8 → 9 → ^
7	G	6	^
8	H	6	^
9	K	6	



我们讲过 每一个小的数据集的长度

解决了向上与向下的访问。

问题：节点的平均孩子数据集为 $O(1)$ ，但是分开而言某个节点数据集规模可能达到 $O(n)$ 。

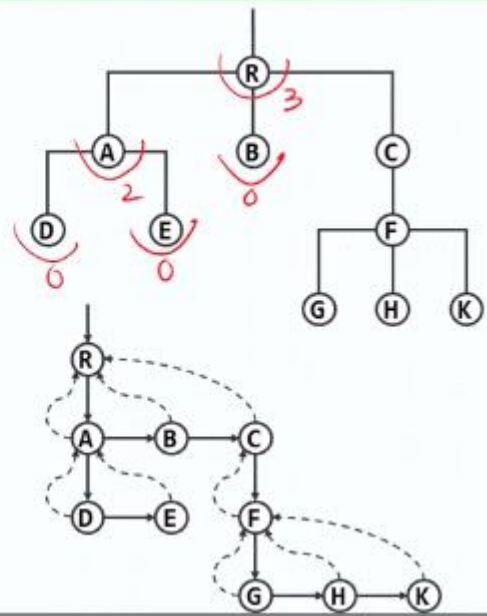
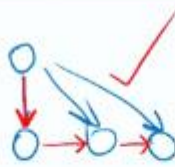
### 7.2.5 长子 + 兄弟

## 长子 + 兄弟

❖ 每个节点均设两个引用

纵: firstChild()

横: nextSibling()



每个节点保留两个数据，即firstchild()，nextsibling()



## 第三节 二叉树

2016年9月1日 10:24

### 7.3.1 二叉树

$$h < n < 2^{h+1}$$

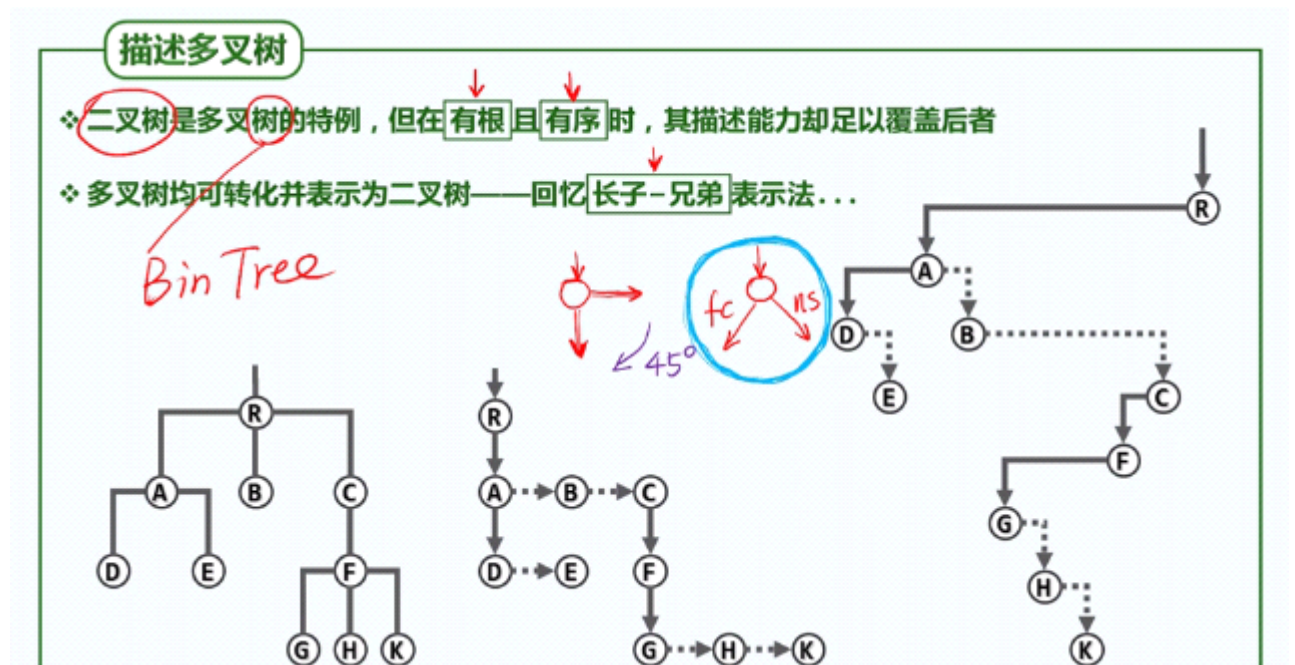
n: 节点数目 h: 树的高度

满二叉树

### 7.3.2 真二叉树

每个节点出度为偶数（2或0）

### 7.3.3 描述多叉树



由于每个节点只保留firstchild（），与nextsibling（）所以可以将nextsibling（）作为右孩子。而firstchild（）作为左孩子

## 第四节 二叉树实现

2016年9月1日 10:41

### 7.4.1 binnode类

**BinNode模板类**

```
❖ #define BinNodePosi(T) BinNode<T>* //节点位置
❖ template <typename T> struct BinNode {
    BinNodePosi(T) parent, lChild, rChild; //父亲、孩子
    T data; int height; int size(); //高度、子树规模
    BinNodePosi(T) insertAsLC( T const & ); //作为左孩子插入新节点
    BinNodePosi(T) insertAsRC( T const & ); //作为右孩子插入新节点
    BinNodePosi(T) succ(); // (中序遍历意义下) 当前节点的直接后继
    template <typename VST> void travLevel( VST & ); //子树层次遍历
    template <typename VST> void travPre( VST & ); //子树先序遍历
    template <typename VST> void travIn( VST & ); //子树中序遍历
    template <typename VST> void travPost( VST & ); //子树后序遍历
};
```

Diagram illustrating the BinNode structure and its insertion:

```
graph TD
    Node((data)) -- parent --> Parent[ ]
    Node -- lChild --> LChild[ ]
    Node -- rChild --> RChild[ ]
```

### 7.4.2 binnode接口

**BinNode接口实现**

```
❖ template <typename T> BinNodePosi(T) BinNode<T>::insertAsLC(T const & e)
{ return lChild = new BinNode( e, this ); } // this.LC == NULL
❖ template <typename T> BinNodePosi(T) BinNode<T>::insertAsRC(T const & e)
{ return rChild = new BinNode( e, this ); }
❖ template <typename T>
int BinNode<T>::size() { //后代总数，亦即以其为根的子树的规模
    int s = 1; //计入本身
    if (lChild) s += lChild->size(); //递归计入左子树规模
    if (rChild) s += rChild->size(); //递归计入右子树规模
    return s;
} //O(n = |size|)
```

Diagram illustrating the size() function logic:

### 7.4.3 bintree类

## BinTree模板类

```

❖ template <typename T> class BinTree {
protected:
    int _size; //规模
    BinNodePosi(T) _root; //根节点
    virtual int updateHeight( BinNodePosi(T) x ); //更新节点x的高度
    void updateHeightAbove( BinNodePosi(T) x ); //更新x及祖先的高度
public:
    int size() const { return _size; } //规模
    bool empty() const { return !_root; } //判空
    BinNodePosi(T) root() const { return _root; } //树根
    /* ... 子树接入、删除和分离接口 ... */
    /* ... 遍历接口 ... */

```

### 7.4.4 高度更新

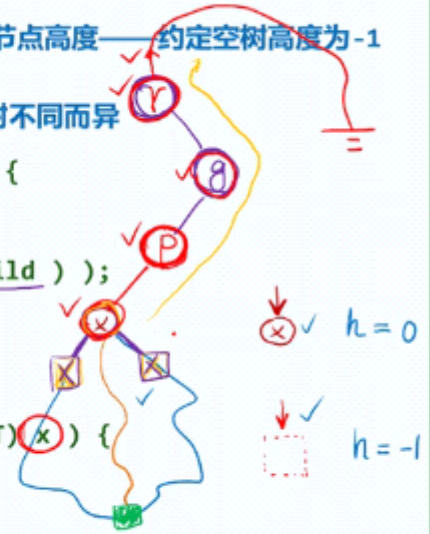
#### 高度更新

```

❖ #define stature(p) ( (p) ? (p)->height : -1 ) //节点高度——约定空树高度为-1
❖ template <typename T> //更新节点x高度，具体规则因树不同而异
int BinTree<T>::updateHeight( BinNodePosi(T) x ) {
    return x->height = 1 +
        max( stature( x->lChild ), stature( x->rChild ) );
} //此处采用常规二叉树规则，O(1)

❖ template <typename T> //更新v及其历代祖先的高度
void BinTree<T>::updateHeightAbove( BinNodePosi(T) x ) {
    while (x) //可优化：一旦高度未变，即可终止
    { updateHeight(x); x = x->parent; }
} //O( n = depth(x) )

```



### 7.4.5 节点插入



## 节点插入

```
❖ template <typename T> BinNodePosi(T)
```

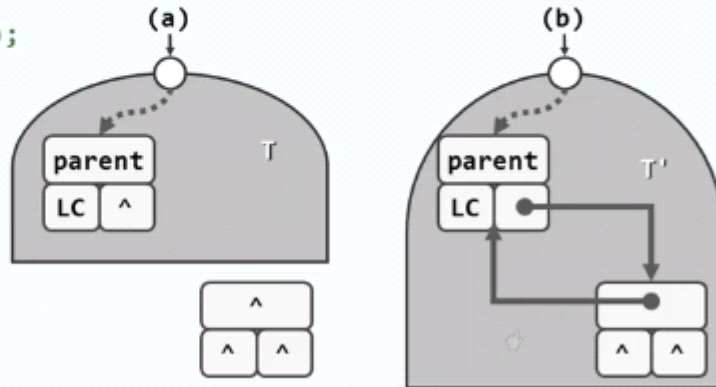
```
    BinTree<T>::insertAsRC( BinNodePosi(T) x, T const & e ) { //insertAsLC()对称
```

```
        _size++; x->insertAsRC(e); //x祖先的高度可能增加，其余节点必然不变
```

```
        updateHeightAbove(x);
```

```
        return x->rChild;
```

```
    }
```



Data Structures & Algorithms (Fall 2012), Tsinghua University

5

节点的插入要涉及相连接的节点，而且要注意整个拓扑结构的变化。

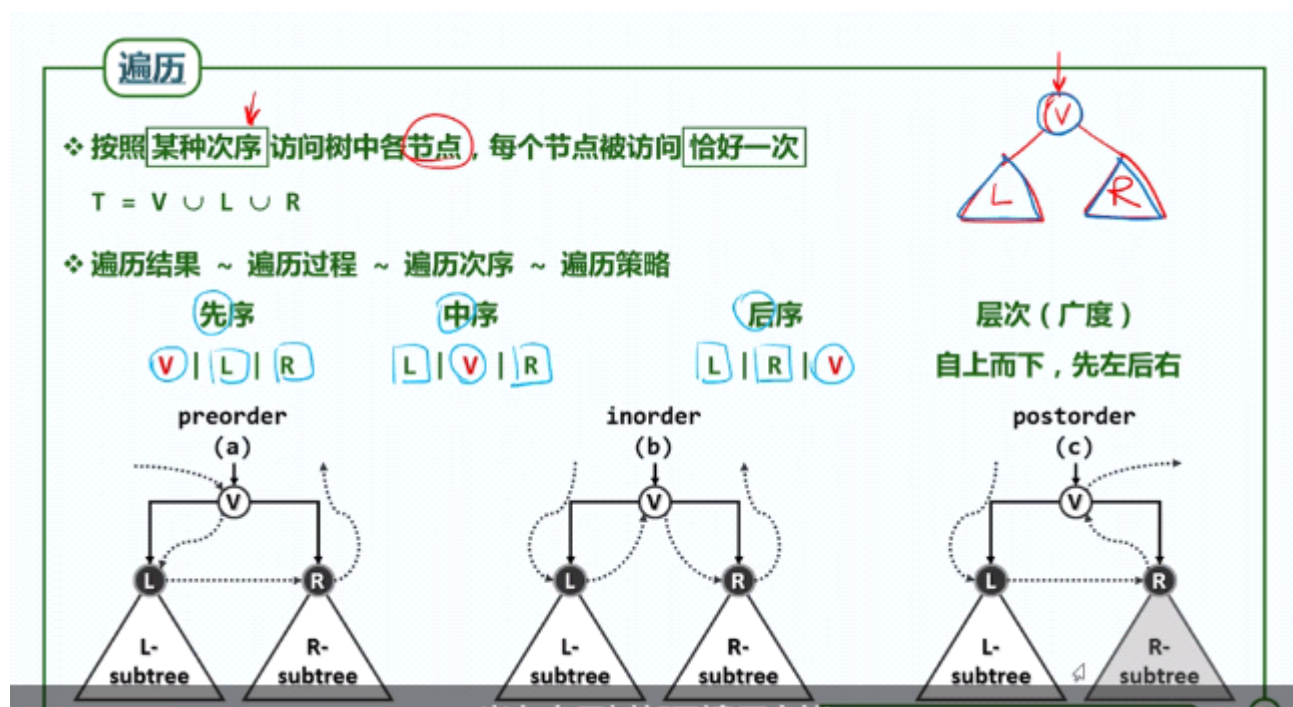
## 第五节 先序遍历

2016年9月1日 11:07

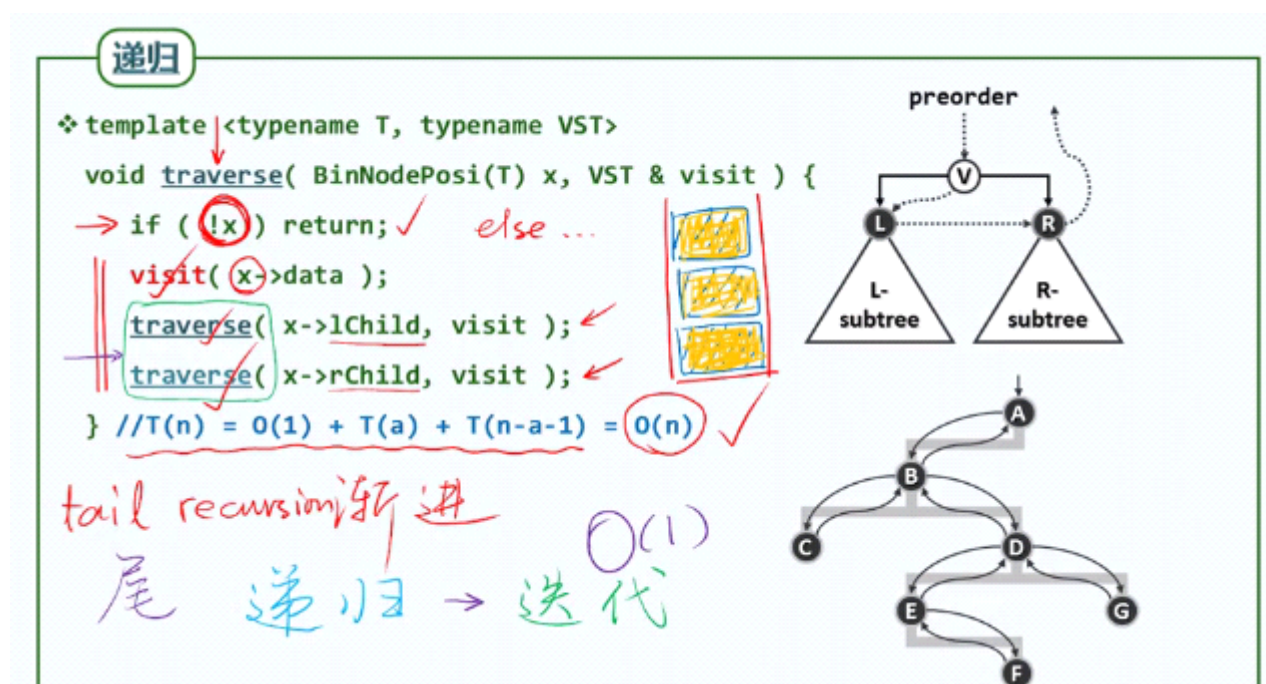
### 7.5.1 转化策略

半线性结构 -> 线性结构

### 7.5.2 遍历规则



### 7.5.3 递归实现



#### 7.5.4 迭代实现 (1)

### 迭代1：实现

```

❖ template <typename T, typename VST>
void travPre_I1( BinNodePosi(T) x, VST & visit ) {
    Stack <BinNodePosi(T)> S; //辅助栈
    if (x) S.push(x); //根节点入栈
    while ( !S.empty() ) { //在栈变空之前反复循环
        x = S.pop(); visit(x->data ); //弹出并访问当前节点
        if ( HasRChild( *x ) ) S.push( x->rChild ); //右孩子先入后出
        if ( HasLChild( *x ) ) S.push( x->lChild ); //左孩子后入先出
    } //体会以上两句的次序
}
                
```

preorder

#### 7.5.5 实例

### 迭代1：实例

preorder

abcdef

a

b  
c

c

d  
f

f

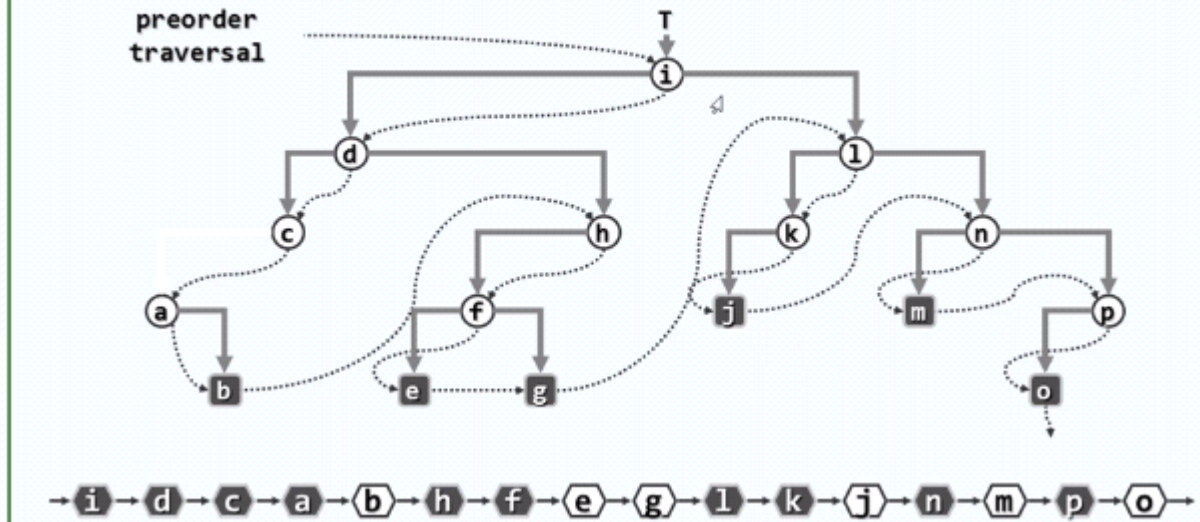
e  
f

f

不能推广至中序、后序遍历

#### 7.5.6 新思路

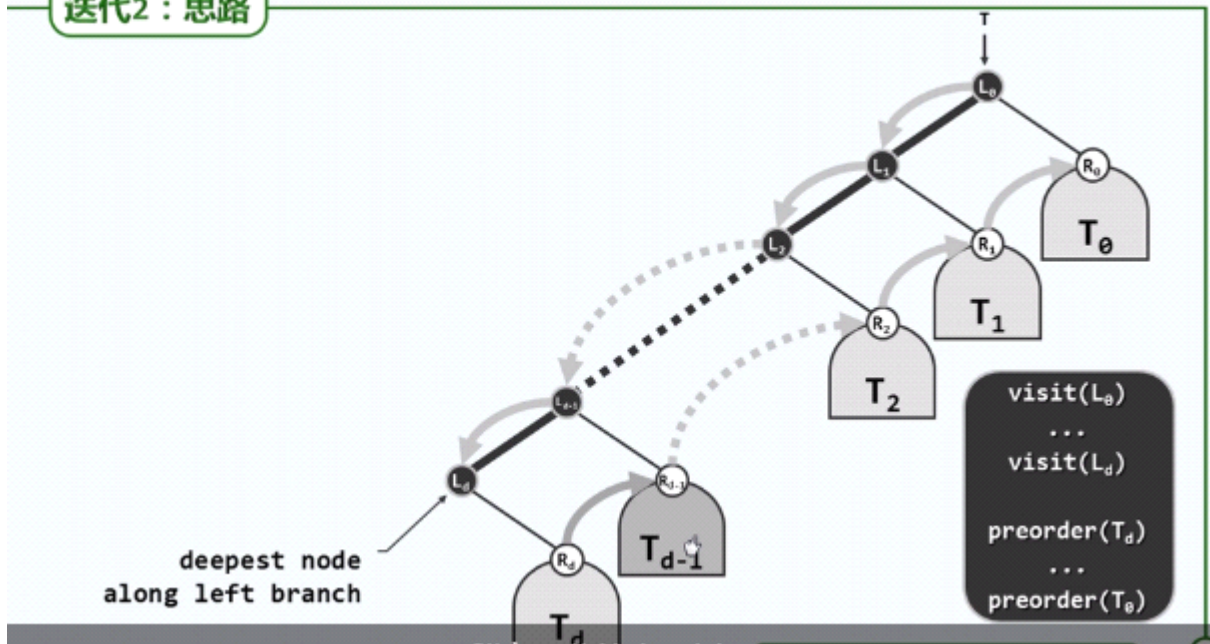
### 迭代2：思路



所谓的先序遍历也就是访问左侧链的左优先原则。  
先自上而下访问左侧链上的节点,再自下而上访问它们的右子树

#### 7.5.7 新构思

### 迭代2：思路



#### 7.5.8 迭代实现（2）



### 迭代2：实现

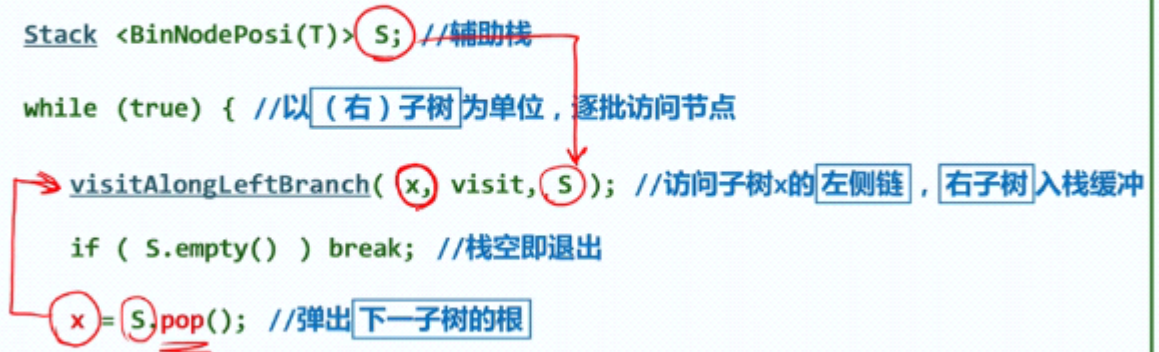
```
❖ template <typename T, typename VST> //分摊O(1)
static void visitAlongLeftBranch(
    BinNodePosi(T) x,
    VST & visit,
    Stack <BinNodePosi(T)> & S )
{
    while (x) { //反复地
        visit( x->data ); //访问当前节点
        S.push( x->rChild ); //右孩子（右子树）入栈（将来逆序出栈）
        x = x->lChild; //沿左侧链下行
    } //只有右孩子、NULL可能入栈——增加判断以剔除后者，是否值得？
}
```



主算法

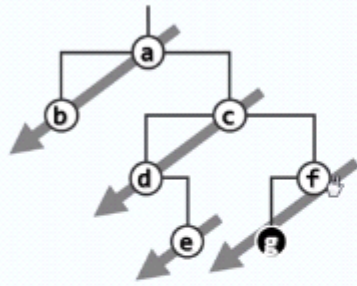
### 迭代2：实现

```
❖ template <typename T, typename VST>
void travPre_I2( BinNodePosi(T) x, VST & visit ) {
    Stack <BinNodePosi(T)> S; //辅助栈
    while (true) { //以（右）子树为单位，逐批访问节点
        visitAlongLeftBranch( x, visit, S ); //访问子树x的左侧链，右子树入栈缓冲
        if ( S.empty() ) break; //栈空即退出
        x = S.pop(); //弹出下一子树的根
    } // #pop = #push = #visit = O(n) = 分摊O(1)
}
```

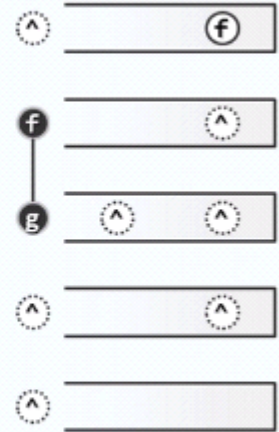
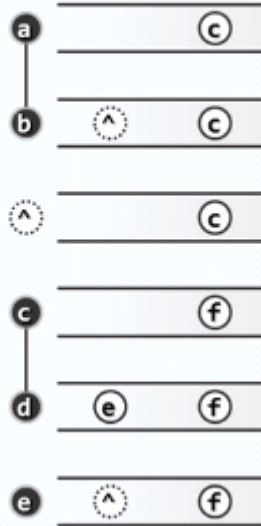
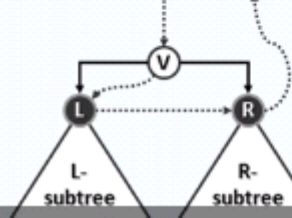


7.5.9 实例

## 迭代2：实例



preorder



## 第六节 中序遍历

2016年9月1日 14:03

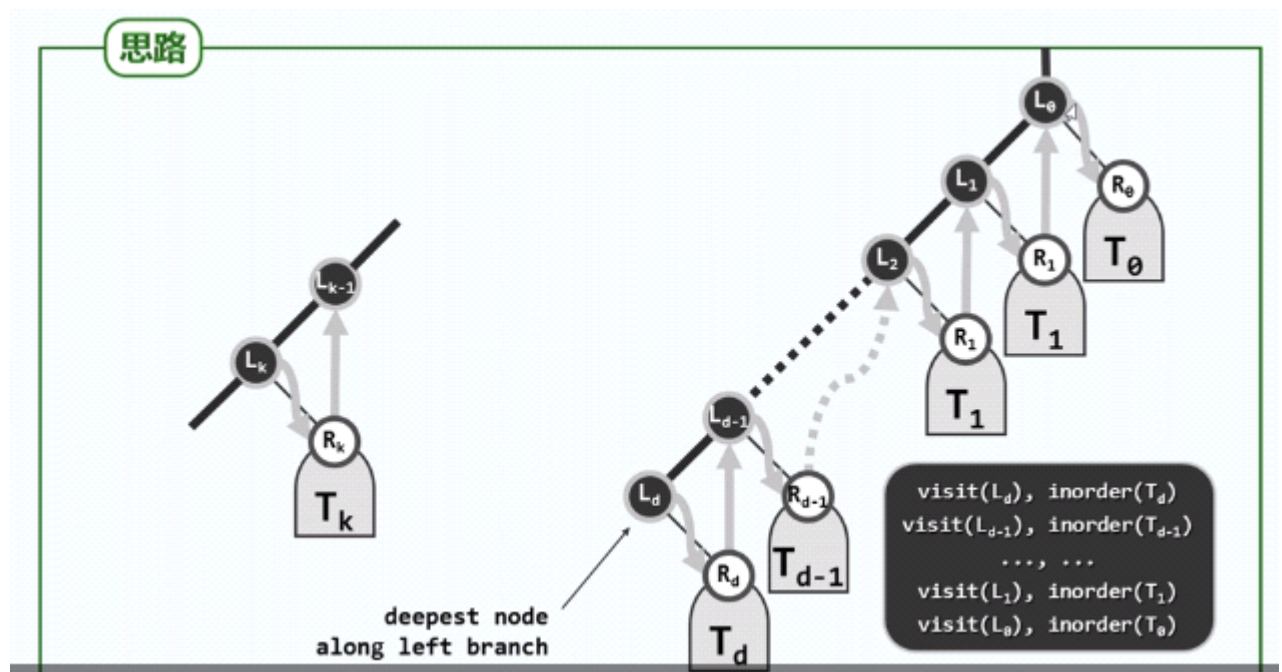
### 7.6.1 递归

```
0 if ( !x ) return; ←  
[ traverse( x->lChild, visit );  
  visit( x->data );  
  traverse( x->rChild, visit ); ] ?
```

### 7.6.2 观察

我们发现了不同层次下的左侧链

### 7.6.3 思路



### 7.6.4 构思

从访问次序上我们了解到需要采用栈。

### 7.6.5 实现

## 实现

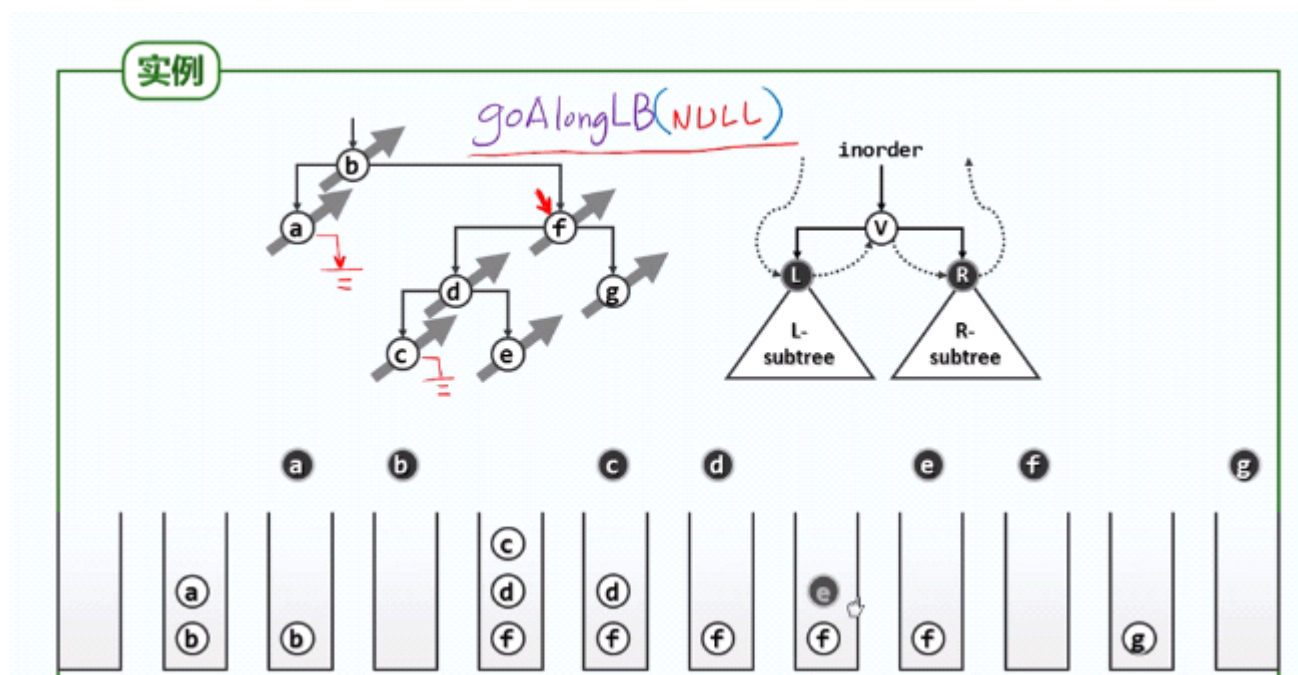
```

❖ template <typename T>
    static void goAlongLeftBranch( BinNodePosi(T) x, Stack <BinNodePosi(T)> & S )
    { while (x) { S.push(x); x = x->lChild; } } //反复地入栈，沿左分支深入

❖ template <typename T, typename V> void travIn_I1( BinNodePosi(T) x, V& visit )
    { Stack <BinNodePosi(T)> S; //辅助栈
      while (true) { //反复地
        ⇒ goAlongLeftBranch( x, S ); //从当前节点出发，逐批入栈
        if ( S.empty() ) break; //直至所有节点处理完毕
        x = S.pop(); //x的左子树或为空，或已遍历（等效于空），故可以
        visit( x->data ); //立即访问之
        x = x->rChild; //再转向其右子树（可能为空，留意处理手法）
      }
    }

```

### 7.6.6 实例



### 7.6.7 分摊分析

复杂度仍为 $O(n)$ ，但是在常系数条件下，优于递归版本。



## 第七节 层次遍历

2016年9月1日 15:12

### 7.7.1 次序

采用队列这种数据结构。

### 7.7.2 实现

实现

```
❖ template <typename T> template <typename VST>
void BinNode<T>::travLevel( VST & visit ) { //二叉树层次遍历

    Queue<BinNodePosi(T)> Q; //引入辅助队列

    Q.enqueue( this ); //根节点入队

    while ( !Q.empty() ) { //在队列再次变空之前，反复迭代

        BinNodePosi(T) x = Q.dequeue(); //取出队首节点，并随即
        visit( x->data ); //访问之

        if ( HasLChild(*x) ) Q.enqueue( x->lChild ); //左孩子入队
        if ( HasRChild(*x) ) Q.enqueue( x->rChild ); //右孩子入队
    }
}
```

### 7.7.3 实例

实例

preorder S: R L >

Q: L R >

A

B

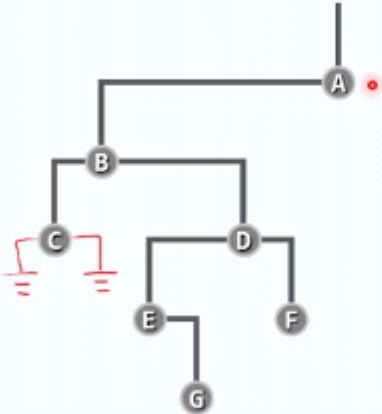
C

D

E

F

G



A		B		C	D
				D	

	E	F	F	G	
	F		G		

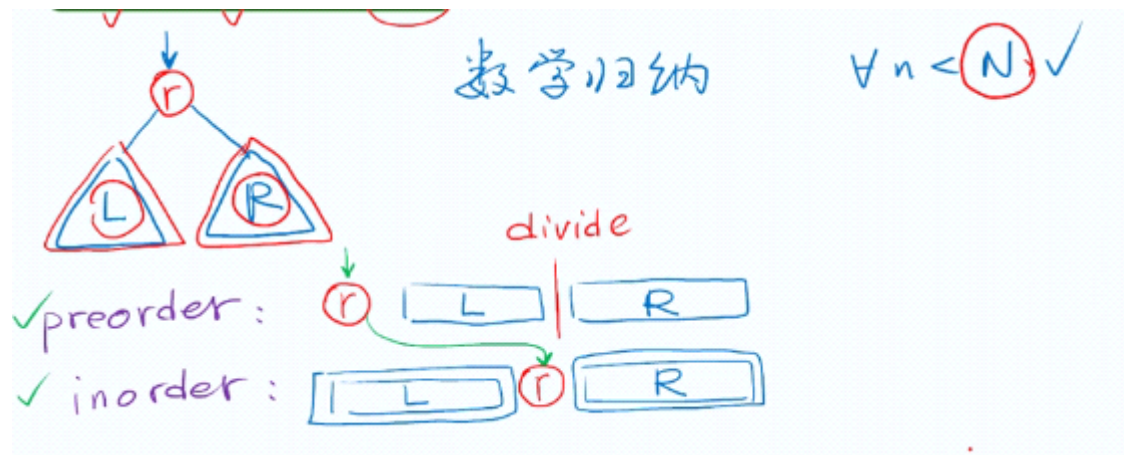
## 第八节 重构

2016年9月1日 15:23

### 7.8.1 遍历序列

由遍历序列，如何还原出树的拓扑结构。

### 7.8.2 (先序|后序) + 中序



先序与右序遍历，无法判断是左子树还是右子树（只存在左子树或右子树）。

### 7.8.3 (先序|后序) \* 真

