



Data Structures and Algorithms

Trees and Binary Trees



Preview

1

Definition

2

Implementation

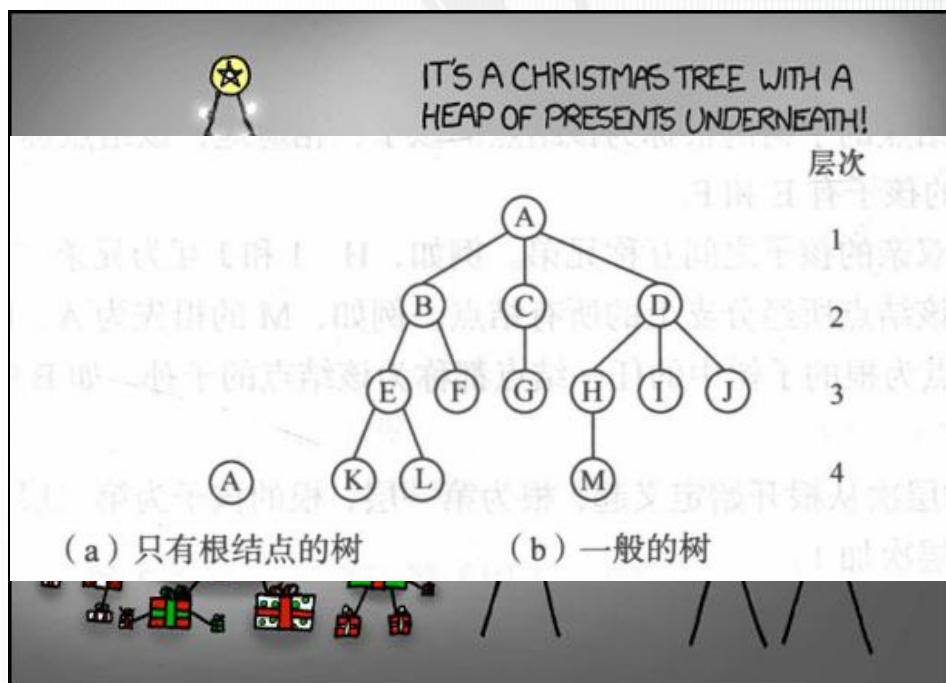
Definition

◆ 树的定义：

树是 n ($n \geq 0$) 个结点的有限集。当 $n = 0$ 时，称为空树。树的结构定义是一个递归的定义。在任意一棵非空树中应满足：

- 1) 有且仅有一个特定的称为根的结点。
- 2) 当 $n > 1$ 时，其余结点可分为 m ($m > 0$) 个互不相交的有限集，其中每个集合本身又是一棵树，并且称为根的子树。

例如，在右图中，(a) 是只有一个根结点的树；(b) 是有13个结点的树，其中A是根，其余结点分成3个互不相交的子集： $T_1 = \{B, E, F, K, L\}$, $T_2 = \{C, G\}$, $T_3 = \{D, H, I, J, M\}$ 。 T_1 、 T_2 、 T_3 都是根A的子树，且本身也是一棵树。例如 T_1 ，其根为B, 其余结点分为两个互不相交的子集： $T_{11} = \{E, K, L\}$, $T_{12} = \{F\}$ 。





Definition

◆ 树的基本术语：

- 结点(Node): 树中的一个独立单元。包含一个数据元素及若干指向其子树的分支。
- 结点的度(Degree of Node): 结点拥有的子树数称为结点的度。
- 树的度(Degree of Tree): 树的度是树内各结点度的最大值。
- 叶子(Leaf): 度为0的结点称为叶子或终端结点。



Definition

- 树的基本术语：
- 非终端结点(Non-Terminal Node): 度不为0的结点称为非终端结点或分支结点。除根结点之外，非终端结点也称为内部结点。
- 双亲和孩子(Parent and Child): 结点的子树的根称为该结点的孩子，相应地，该结点称为孩子的双亲。
- 兄弟(Siblings): 同一个双亲的孩子之间互称兄弟。
- 祖先(Ancestor): 从根到该结点所经分支上的所有结点。



Definition

◆ 树的基本术语：

- 子孙(Descendants): 以某结点为根的子树中的任一结点都称为该结点的子孙。
- 层次 (Level) : 结点的层次从根开始定义起，根为第一层，根的孩子为第二层。
树中任一结点的层次等于其双亲结点的层次加1。
- 堂兄弟(Cousin): 双亲在同一层的结点互为堂兄弟。
- 树的深度(Depth of Tree): 树中结点的最大层次称为树的深度或高度。



Definition

◆ 树的基本术语：

- 有序树和无序树(Ordered/Unordered Tree): 如果将树中结点的各子树看成从左至右是有次序的(即不能互换), 则称该树为有序树, 否则称为无序树。在有序树中最左边的子树的根称为第一个孩子, 最右边的称为最后一个孩子。
- 森林(Forest): 是 n ($n \geq 0$) 棵互不相交的树的集合。对树中每个结点而言, 其子树的集合即为森林。由此, 也可以用森林和树相互递归的定义来描述树。
- 路径和路径长度(Path and Length of Path): 树中两个结点之间的路径是由这两个结点之间所经过的结点序列构成的, 而路径长度是路径上所经过的边的个数。



Definition

◆ 树的基本性质：

- 1) 树中的结点数等于所有结点的度数之和加1。
- 2) 度为m的树中第i层上至多有 $m^{(i-1)}$ 结点。
- 3) 高度为h的m叉树至多有 $(m^h-1)/(m-1)$ 个结点。
- 4) 具有n个结点的m叉树的最小高度为 $\log_m[(n(m-1))+1]$ 向上取整。

◆ 数据对象 D:

D是具有相同特性的数据元素的集合。

◆ 数据关系 R:

若D为空集，则称为空树。

否则：

- (1) 在D中存在唯一的称为根的数据元素root；
- (2) 当n>1时，其余结点可分为m ($m>0$)个互不相交的有限集T1, T2, ..., Tm，其中每一棵子集本身又是一棵符合本定义的树，称为根root的子树。



Definition

ADT Tree

{ 数据对象 **D**: **D** 是具有相同特性的数据元素的集合

数据关系 **R**: 若 **D** 为空集, 则称为空树

否则:

(1) 在 **D** 中存在唯一的称为 **根** 的数据元素 **root**;

(2) 当 $n > 1$ 时, 其余结点可分为 m ($m > 0$) 个互不相交的有限集 T_1, T_2, \dots, T_m , 其中每一棵子集本身又是一棵符合本定义的树, 称为根 **root** 的 **子树**

基本操作:

查找类操作

插入类操作

删除类操作

} ADT Tree



Implementation

查找类：

TreeEmpty(T)

初始条件：树T已存在

操作结果：空树，返回 TRUE;否则 FALSE

TreeDepth(T)

初始条件：树T已存在

操作结果：返回 T 的深度

Root(T)

初始条件：树T已存在

操作结果：返回T的根

Value(T, cur_e)

初始条件：树T已存在,cur_e是T中结点

操作结果：返回cur_e 的值

Parent(T, cur_e)

初始条件：树T已存在,cur_e是T中结点

操作结果：若cur_e是T中非根结点,返回其双亲;否则,返回空



Implementation

查找类：

LeftChild(T, cur_e)

初始条件：树T已存在,cur_e是T中结点

操作结果：若cur_e是T中非叶子结点,返回其最左孩子;否则,返回空

RightChild(T, cur_e)

初始条件：树T已存在,cur_e是T中结点

操作结果：若cur_e是T中非叶子结点,返回其最右孩子;否则,返回空

TraverseTree(T, visit())

初始条件：树T已存在

操作结果：按某种次序对T 的每个元素调用函数visit()



Implementation

插入类：

InitTree(*T)

操作结果：构造空树T

CreateTree(*T, definition)

初始条件： definition给出树的定义

操作结果：按definition构造树T

Assign(T, cur_e, value)

初始条件：树T已存在,cur_e是T中结点

操作结果：结点cur_e赋值为value

InsertChild(*T, *p,i,c)

初始条件：树T存在,p指向T中结点, $1 \leq i \leq p$ 指结点度+1,非空树c与T不相交

操作结果：将以c为根的树插入为T中p指结点的第i棵子树



Implementation

删除类：

DestroyTree(*T)

初始条件：树T 已存在

操作结果：销毁树T

ClearTree(*T)

初始条件：树T 已存在

操作结果：将树T 清为空树

DeleteChild(*T, *p,i)

初始条件：树T存在,p指向T中结点, $1 \leq i \leq p$ 指结点度

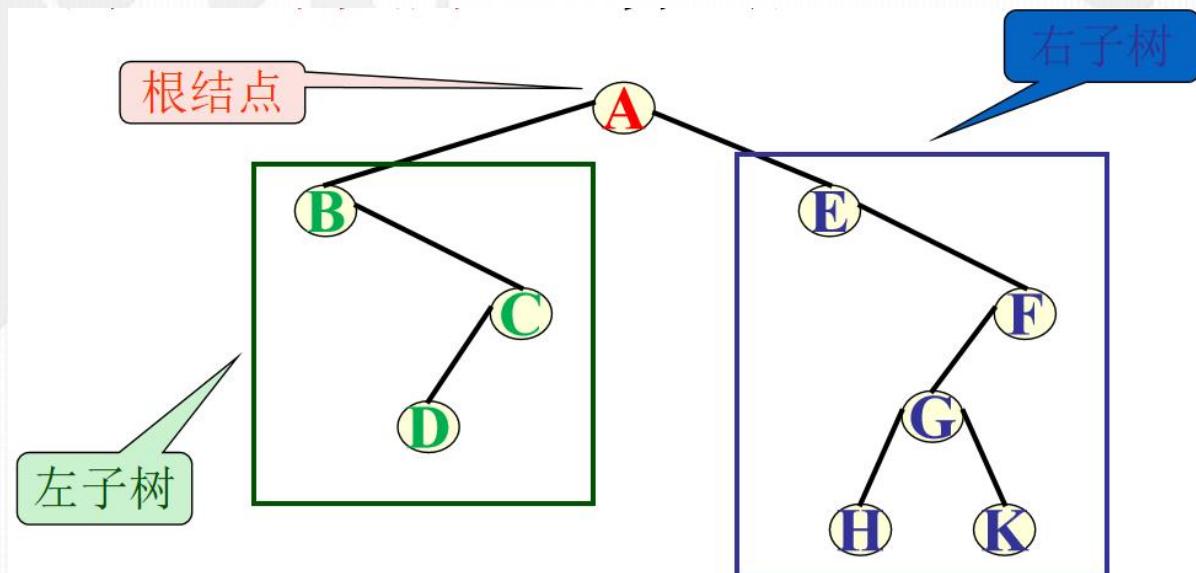
操作结果：删除T中p指结点的第i棵子树

Definition

◆ 二叉树的定义：

二叉树(Binary Tree)是n ($n \geq 0$)个结点所构成的集合，它或为空树 ($n=0$)；或为非空树 ($n>0$)，对于非空树T：

- 1) 有且仅有一个称之为根的结点；
- 2) 除根结点以外的其余结点分为两个互不相交的子集T1和T2 分别称为T的左子树和右子树，且T1和T2本身又都是二叉树。





Definition

◆ 二叉树与树的区别：

二叉树与树一样具有递归性质，二叉树与树的区别主要有以下两点：

- 1) 二叉树每个结点至多只有两棵子树(即二叉树中不存在度大于2的结点);
- 2) 二叉树的子树有左右之分，其次序不能任意颠倒。

二叉树的递归定义表明二叉树或为空，或是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。由于这两棵子树也是二叉树，则由二叉树的定义，它们也可以是空树。由此，二叉树可以有5种基本形态。

注意：区分m叉树和树的度为m。m叉树指一个节点最多有m个分支。树的度为m指树最少有一个节点最多有m个分支。



Definition

◆ 二叉树的性质：

✿ 性质1：在二叉树的第 i 层上至多有 2^{i-1} 个结点 ($i \geq 1$)

用归纳法证明：

归纳基： $i = 1$ 层时，只有一个根结点：

$$2^{i-1} = 2^0 = 1;$$

归纳假设：假设对所有的 j , $1 \leq j < i$, 命题成立，即第 j 层上至多有 2^{j-1} 个结点，则第 $i-1$ 层上至多有 2^{i-2} 个结点；

归纳证明：由于二叉树上每个结点至多有两棵子树，
则第 i 层的结点数 $= 2^{i-2} \times 2 = 2^{i-1}$ 。



Definition

- ◆ 二叉树的性质：

✿ 性质2：深度为 k 的二叉树上至多含 $2^k - 1$ 个结点 ($k \geq 1$)

证明：由性质1，可得深度为 k 的二叉树最大结点数是

$$\sum_{i=1}^k (\text{第 } i \text{ 层的最大结点数}) = \sum_{i=1}^k 2^{i-1} = \frac{1 - 2^{k-1} * 2}{1 - 2} = 2^k - 1$$

等比数列求和： $S_n = \frac{a_1 - a_n q}{1 - q}$



Definition

◆ 二叉树的性质：

* 性质3：对任何一棵二叉树 T ，如果其叶子结点数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0 = n_2 + 1$

证明：设 n_1 为二叉树 T 中度为1的结点数

因为：二叉树中所有结点的度均小于或等于2

所以：其结点总数 $n = n_0 + n_1 + n_2$

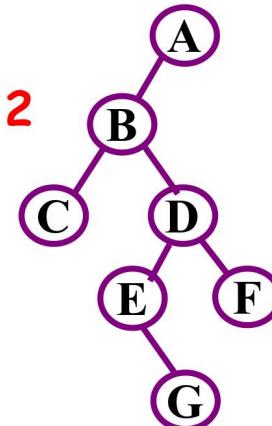
二叉树中，除根结点外，其余结点都只有一个分支进入

设 B 为分支总数，则 $n = B + 1$

又：分支由度为1和度为2的结点射出， $\therefore B = n_1 + 2 * n_2$

于是， $n = B + 1 = n_1 + 2 * n_2 + 1 = n_0 + n_1 + n_2$

$$\therefore n_0 = n_2 + 1$$

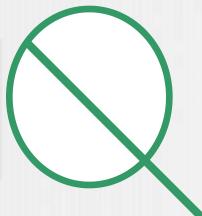


Definition

- ◆ 二叉树的五种基本形态：

只含根结点

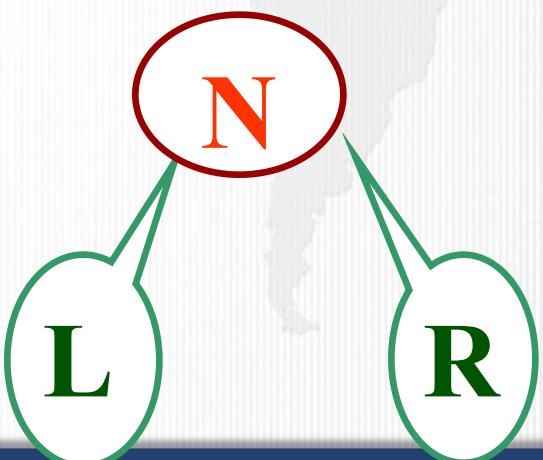
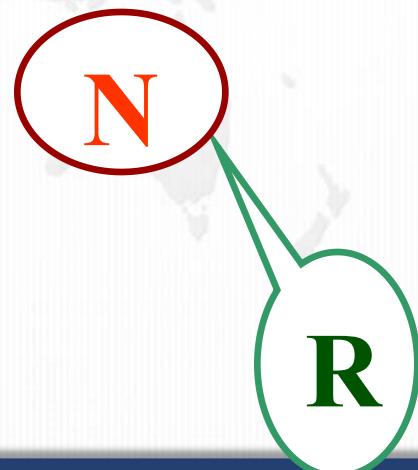
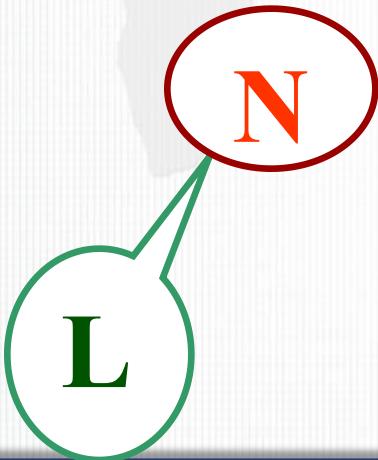
空树



左右子
树均不
为空树

右子树为空树

左子树为空树

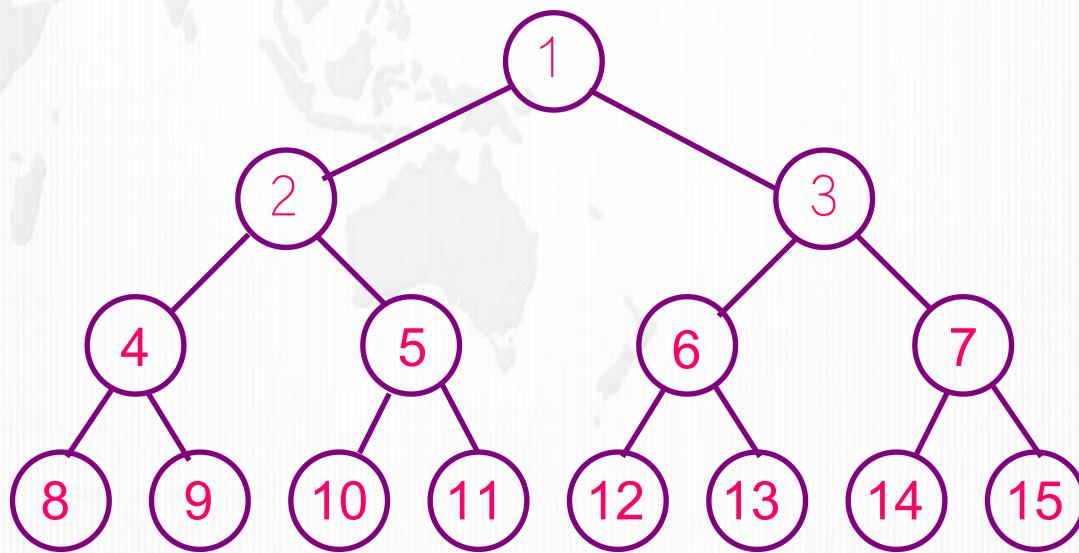


Definition

◆ 几个特殊的二叉树：

满二叉树(Complete Binary Tree)

一棵高度为知且含有 $2^h - 1$ 个结点的二叉树称为满二叉树，即树中的每层都含有最多的结点，满二叉树的叶子结点都集中在二叉树的最下一层，并且除叶子结点之外的每个结点度数均为2。可以对满二叉树按层序编号：约定编号从根结点（根结点编号为1）起，自上而下，自左向右。这样，每个结点对应一个编号，对于编号为i的结点，若有双亲，则其双亲为*l i/2 r*，若有左孩子，则左孩子为 $2i$ ；若有右孩子，则右孩子为 $2i+1$ 。

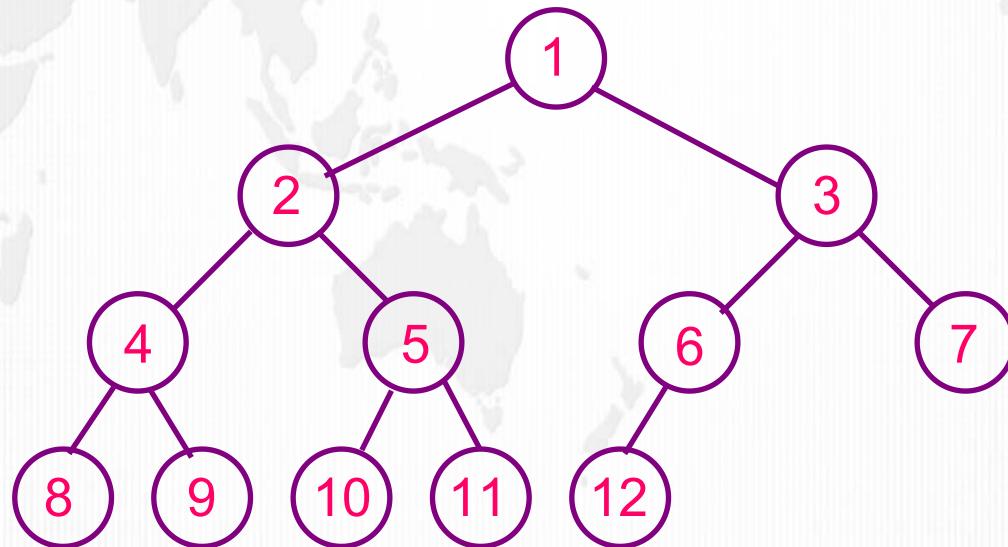


Definition

◆ 几个特殊的二叉树：

完全二叉树

高度为h、有n个结点的二叉树，当且仅当其每个结点都与高度为h的满二叉树中编号为 1-n 的结点一一对应时，称为完全二叉树，（完全二叉树就是对应相同高度的满二叉树缺失最下层最右边的一些连续叶子结点）



Definition

- ◆ 完全二叉树的性质：

***性质4：** 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

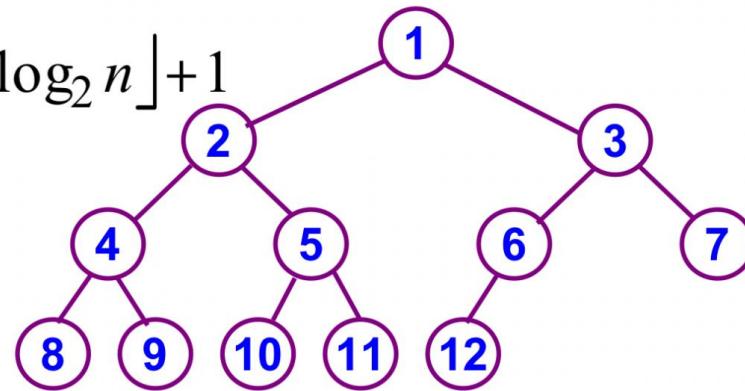
证明：设深度为 k ，根据二叉树性质2知：

$$2^{k-1} - 1 < n \leq 2^k - 1,$$

$$2^{k-1} \leq n < 2^k, \text{ 于是有:}$$

$$k - 1 \leq \log_2 n < k$$

$\because k$ 为整数, \therefore 取 $k = \lfloor \log_2 n \rfloor + 1$

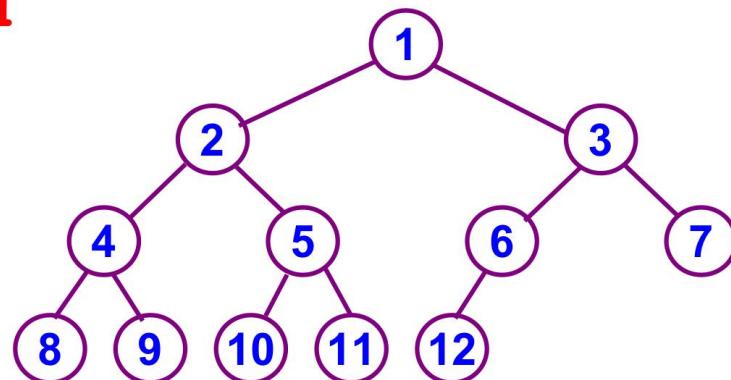


Definition

◆ 完全二叉树的性质：

***性质5：**如果对一棵有n个结点的完全二叉树的结点按层序编号，则对任一结点*i*($1 \leq i \leq n$)，有：

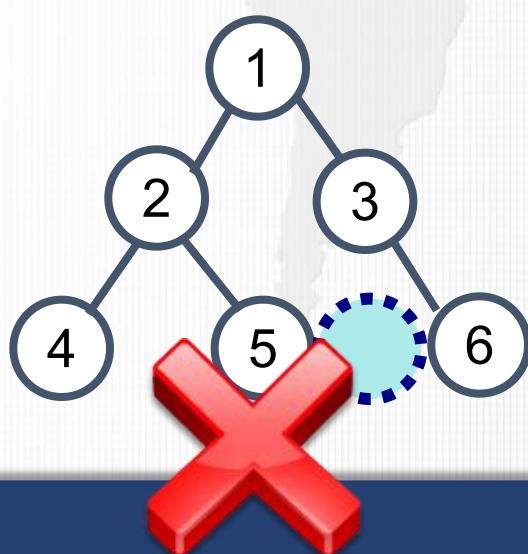
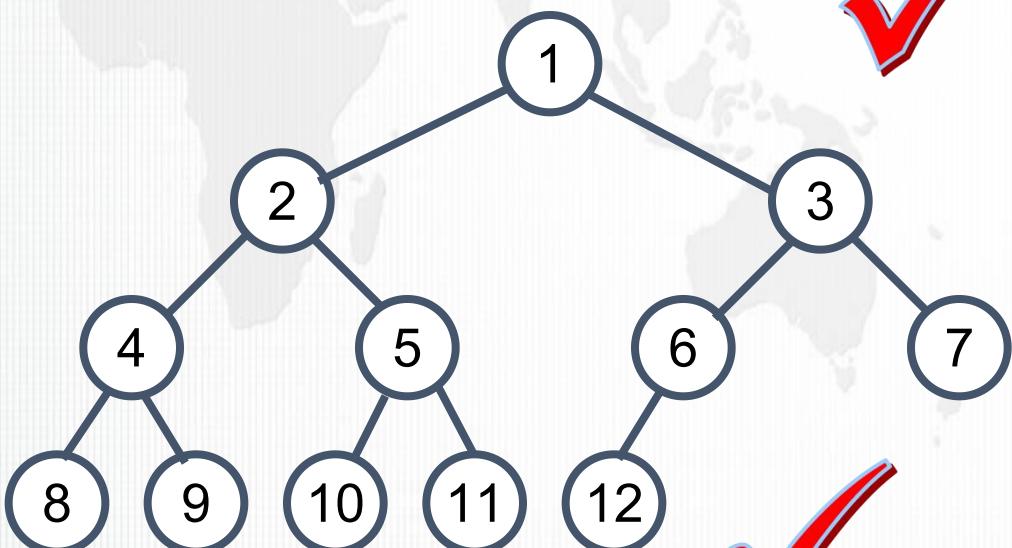
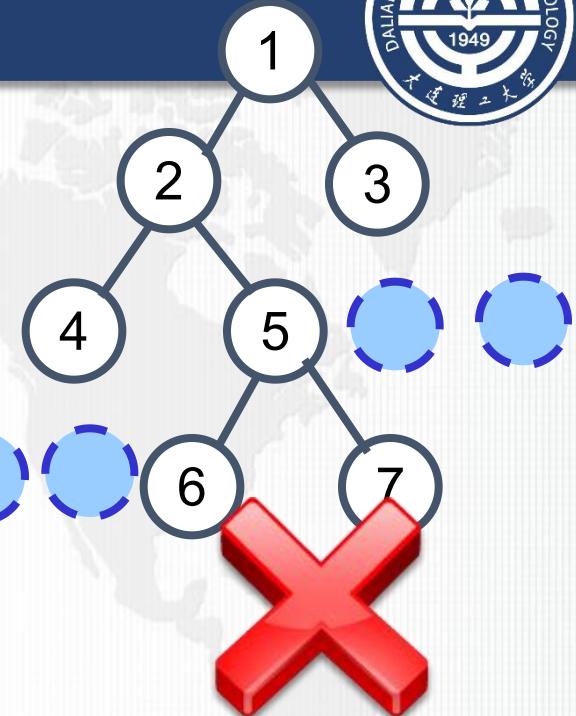
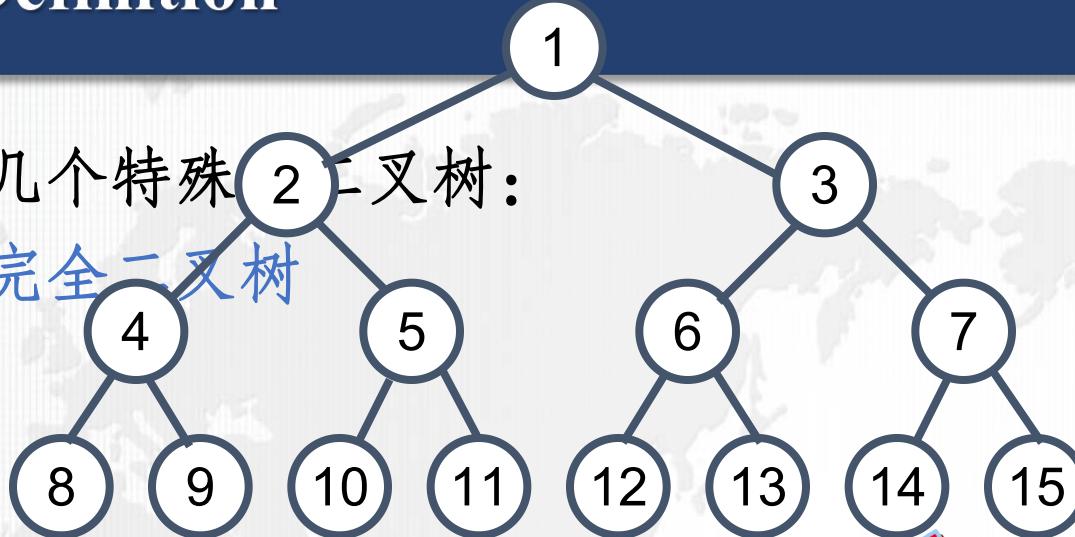
- (1) 如果*i=1*，则结点*i*是二叉树的根，无双亲；如果*i>1*，则其双亲是*[i/2]*
- (2) 如果 $2i > n$ ，则结点*i*无左孩子；如果 $2i \leq n$ ，则其左孩子是 $2i$
- (3) 如果 $2i+1 > n$ ，则结点*i*无右孩子；如果 $2i+1 \leq n$ ，则其右孩子是 $2i+1$



Definition

- ◆ 几个特殊二叉树：

完全二叉树





Definition

◆ 几个特殊的二叉树：

二叉排序树

左子树上所有结点的关键字均小于根结点的关键字；右子树上的所有结点的关键字均大于根结点的关键字；左子树和右子树又各是一棵二叉排序树。

平衡二叉树

树上任一结点的左子树和右子树的深度之差不超过1。



Definition

ADT BinaryTree

{ 数据对象D: D是具有相同特性的数据元素的集合

数据关系R: 若D为空集，则称为空树

否则：

(1) 在D中存在唯一的称为根的数据元素root；

(2) 当n>1时，由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树构成

基本操作：

查找类操作

插入类操作

删除类操作

} ADT BinaryTree



Definition

基本操作：

查找类：

BiTreeEmpty(T)

初始条件：二叉树T已存在

操作结果：空二叉树，返回 TRUE;否则 FALSE

BiTreeDepth(T)

初始条件：二叉树T已存在

操作结果：返回T 的深度

Root(T)

初始条件：二叉树T已存在

操作结果：返回T的根

Value(T, e)

初始条件：二叉树T已存在,e是T中结点

操作结果：返回e 的值



Definition

Parent(T, e)

初始条件：二叉树T已存在, e是T中结点

操作结果：若e是T中非根结点,返回其双亲;否则,返回空

LeftChild(T, e)

初始条件：二叉树T已存在, e是T中结点

操作结果：若e是T中非叶子结点,返回其最左孩子;否则,返回空

RightChild(T, e)

初始条件：二叉树T已存在, e是T中结点

操作结果：若e是T中非叶子结点,返回其最右孩子;否则,返回空



Definition

PreOrderTraverse(T, visit())

初始条件：二叉树T已存在

操作结果：先序遍历二叉树T

InOrderTraverse(T, visit())

初始条件：二叉树T已存在

操作结果：中序遍历二叉树T

PostOrderTraverse(T, visit())

初始条件：二叉树T已存在

操作结果：后序遍历二叉树T

LevelOrderTraverse(T, visit())

初始条件：二叉树T已存在

操作结果：层序遍历二叉树T



Definition

插入类：

InitBiTree(*T)

操作结果：构造空二叉树T

CreateBiTree(*T, definition)

初始条件：definition给出二叉树的定义

操作结果：按definition构造二叉树T

Assign(T, cur_e, value)

初始条件：二叉树T已存在,cur_e是T中结点

操作结果：结点cur_e赋值为value

InsertChild(*T, *p,LR,c)

初始条件：二叉树T存在,p指向T中结点,LR为0或1,非空二叉树c与T不相交

操作结果：插入c为T中p指结点的左或右子树



Definition

删除类：

DestroyBiTree(*T)

初始条件：二叉树T 已存在

操作结果：销毁二叉树T

ClearBiTree(*T)

初始条件：二叉树T 已存在

操作结果：将二叉树T 清为空树

DeleteChild(*T, *p,LR)

初始条件：树T存在,p指向T中结点,LR为0或1

操作结果：删除T中p指结点的左或右子树



Implementation

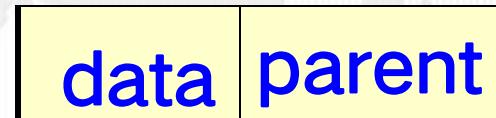
◆ 树的存储结构——双亲表示法：

实现：定义结构数组存放树的结点，每个结点含两个域：

数据域：存放结点本身信息

双亲域：指示本结点的双亲结点在数组中位置

特点：找双亲容易，找孩子难

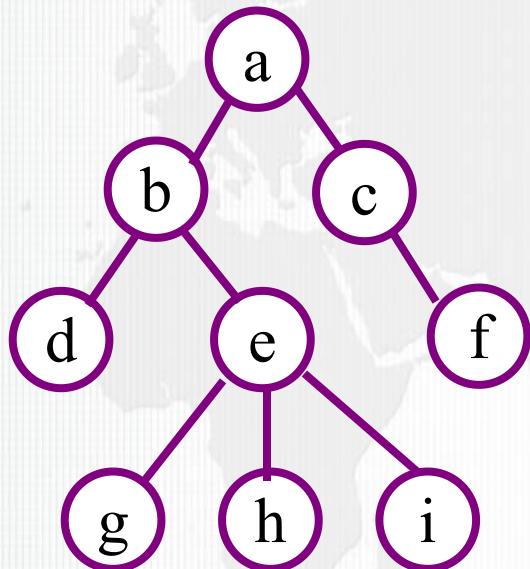


```
● // 双亲表示法
6 # define MAX_TREE_SIZE 100
7
8 typedef int TElemType;
9 // 结点的结构
10 <typedef struct {
11     TElemType data;
12     int parent;
13 } PTNode;
14 // 树的结构
15 <typedef struct {
16     PTNode nodes[MAX_TREE_SIZE];
17     int r, n; // 根的位置和结点数
18 } PTree;
```

Implementation

◆ 树的存储结构——双亲表示法：

实现：定义结构数组存放树的结点，每个结点含两个域：



PTree T;

T.r=0; //根结点位置

T.n=9; //结点个数

	data	parent
0	a	-1
1	b	0
2	c	0
3	d	1
4	e	1
5	f	2
6	g	4
7	h	4
8	i	4

T.nodes

找双亲容易
找孩子难

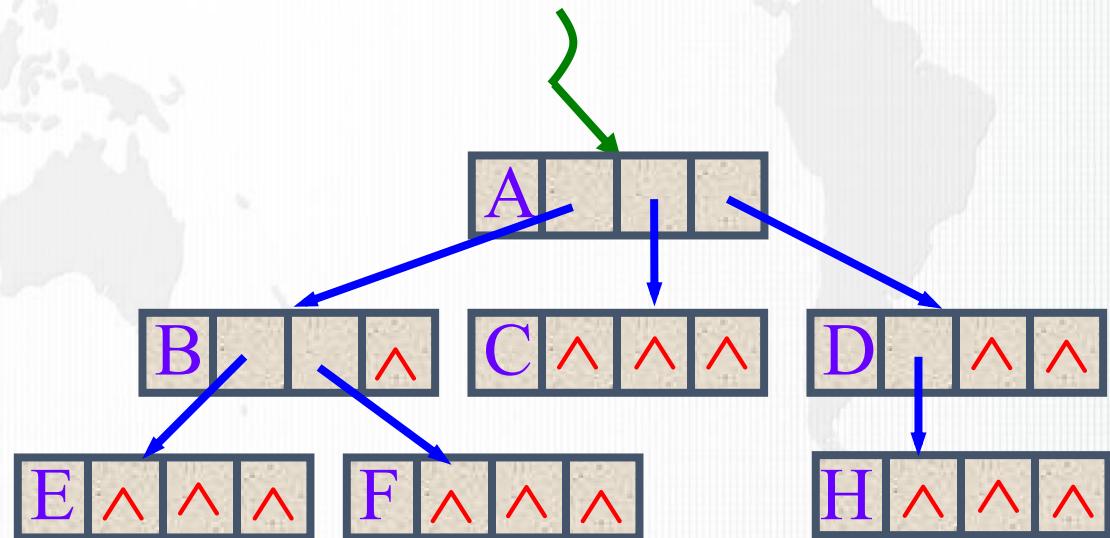
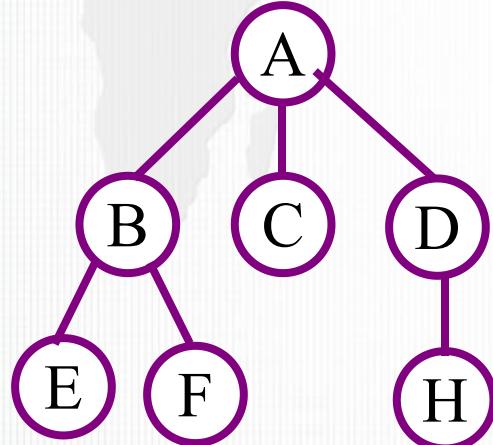
Implementation

◆ 树的存储结构——孩子表示法：多重链表

每个结点有多个指针域，分别指向其子树的根

结点同构：结点的指针个数相等，为树的度D

结点不同构：结点指针个数不等，为该结点的度d

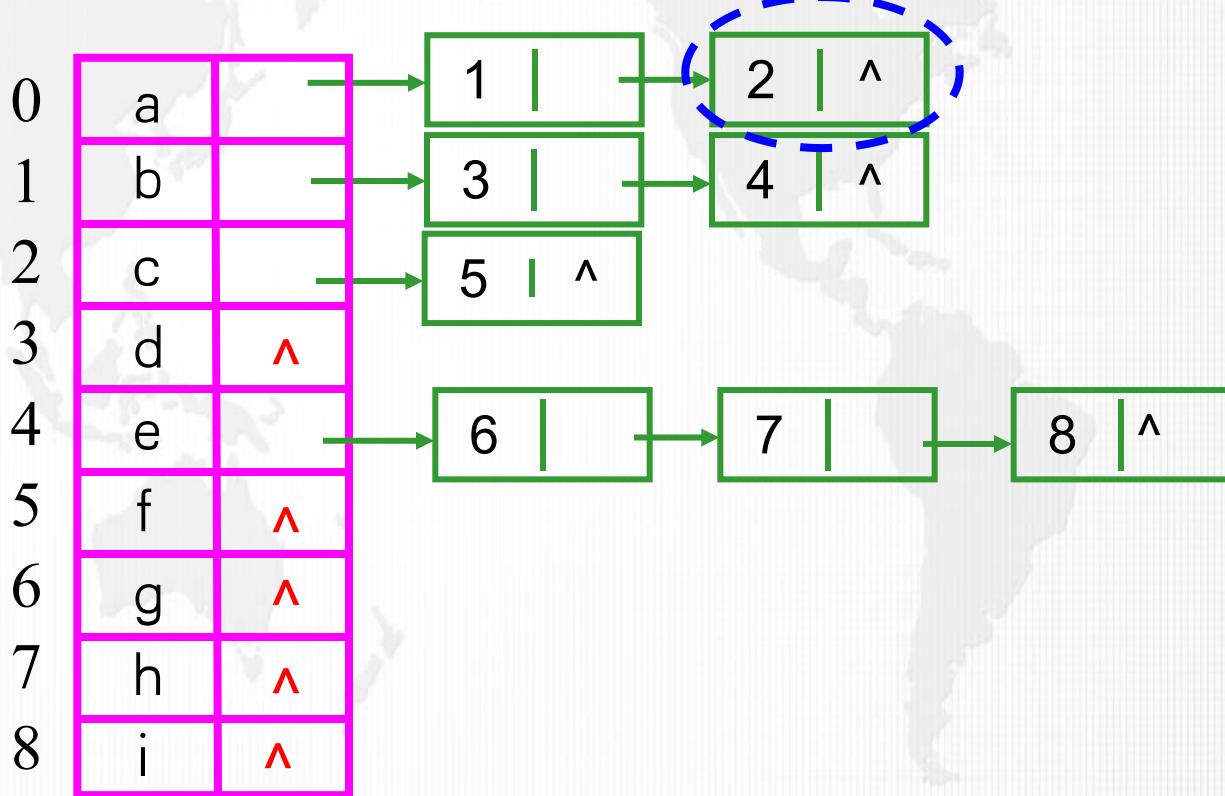
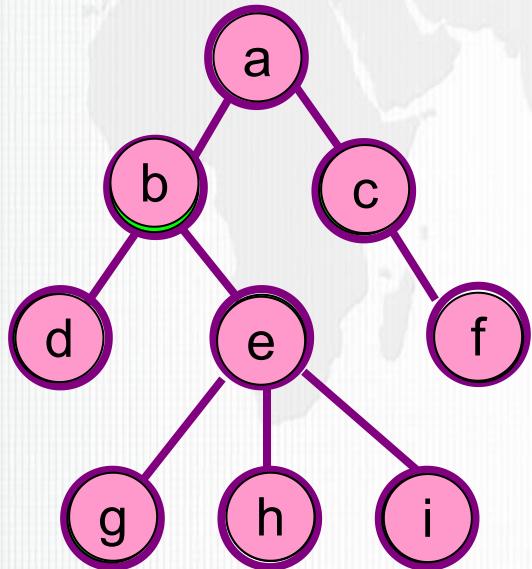


Implementation



◆ 树的存储结构——孩子表示法：孩子链表

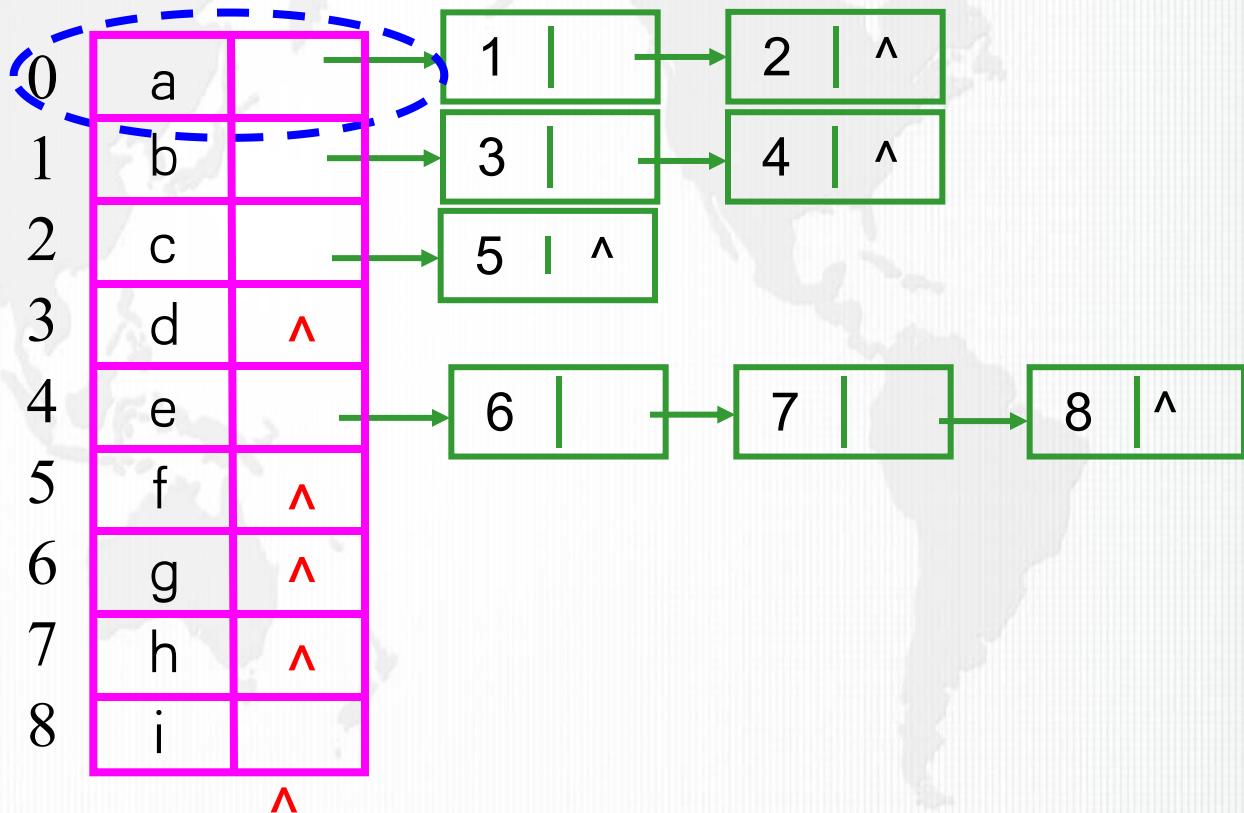
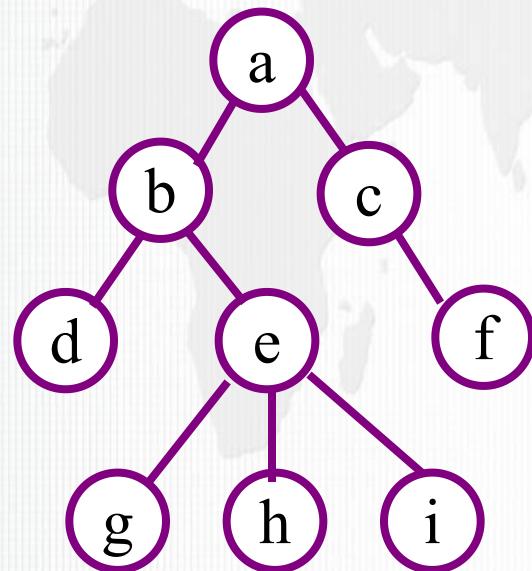
每个结点的孩子结点用单链表存储，再用含n个头指针的线性表指向每个孩子链表



树的存储结构

◆ 树的存储结构——孩子表示法：孩子链表

每个结点的孩子结点用单链表存储，再用含n个头指针的线性表指向每个孩子链表





Implementation

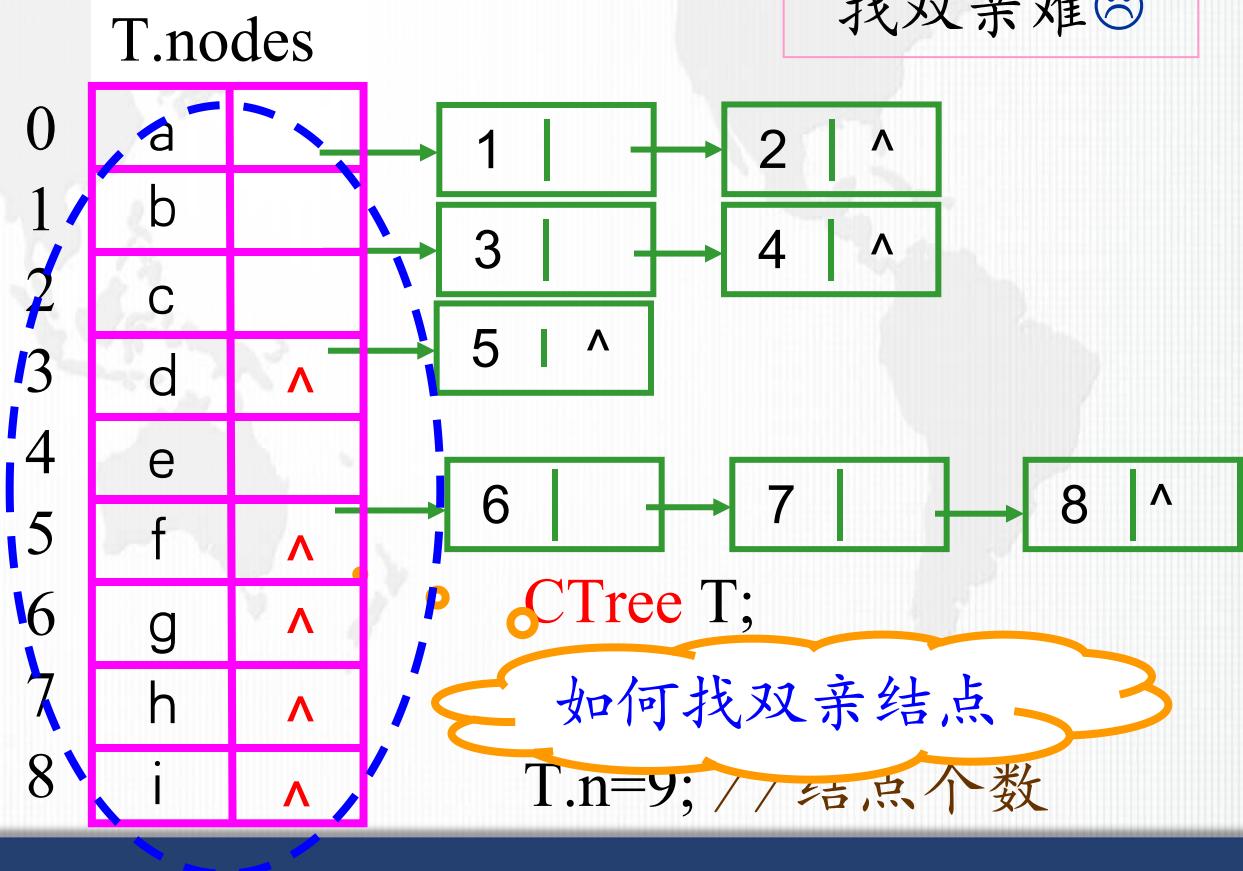
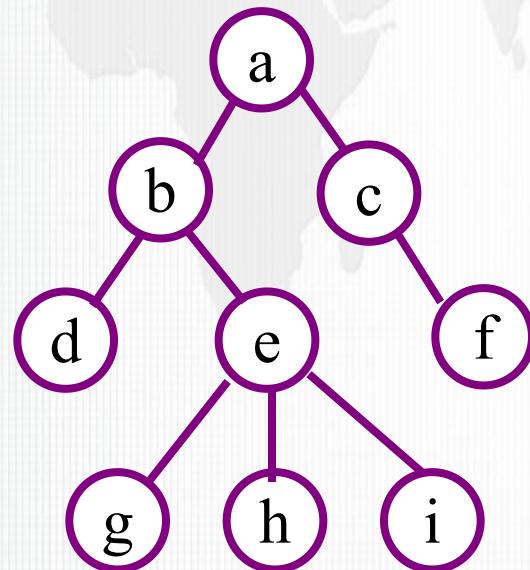
◆ 树的存储结构——孩子表示法：孩子链表

```
18  
19 // 孩子表示法 双重链表  
20 // 定义孩子节点  
21 typedef struct ChildNode {  
22     int childIndex; // 孩子节点在数组中的位置  
23     struct ChildNode* next; // 下一个孩子节点的指针  
24 } ChildNode;  
25 // 定义树的节点。  
26 typedef struct TreeNode {  
27     char data; // 存储的数据，类型可以根据需要变动  
28     struct ChildNode* firstChild; // 第一个孩子节点的指针  
29 } TreeNode;  
30 // 定义树  
31 typedef struct {  
32     TreeNode nodes[MAX_TREE_SIZE];  
33     int r, n; // 根的位置和结点数  
34 } Tree;  
35
```

Implementation

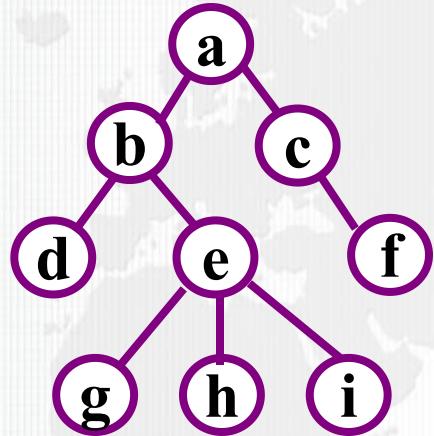
◆ 树的存储结构——孩子表示法：孩子链表

每个结点的孩子结点用单链表存储，再用含n个头指针的线性表指向每个孩子链表



Implementation

◆ 树的存储结构——孩子表示法：带双亲的孩子链表



data parent firstchild

	a	-1		1	2	^
0	a	-1		1	2	^
1	b	0		3	4	^
2	c	0		5	^	
3	d	1	^			
4	e	1		6		
5	f	2	^			
6	g	4	^			
7	h	4	^			
8	i	4	^			

找孩子容易
找双亲容易 😊



Implementation

◆ 树的存储结构——孩子兄弟表示法

用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点

结点结构

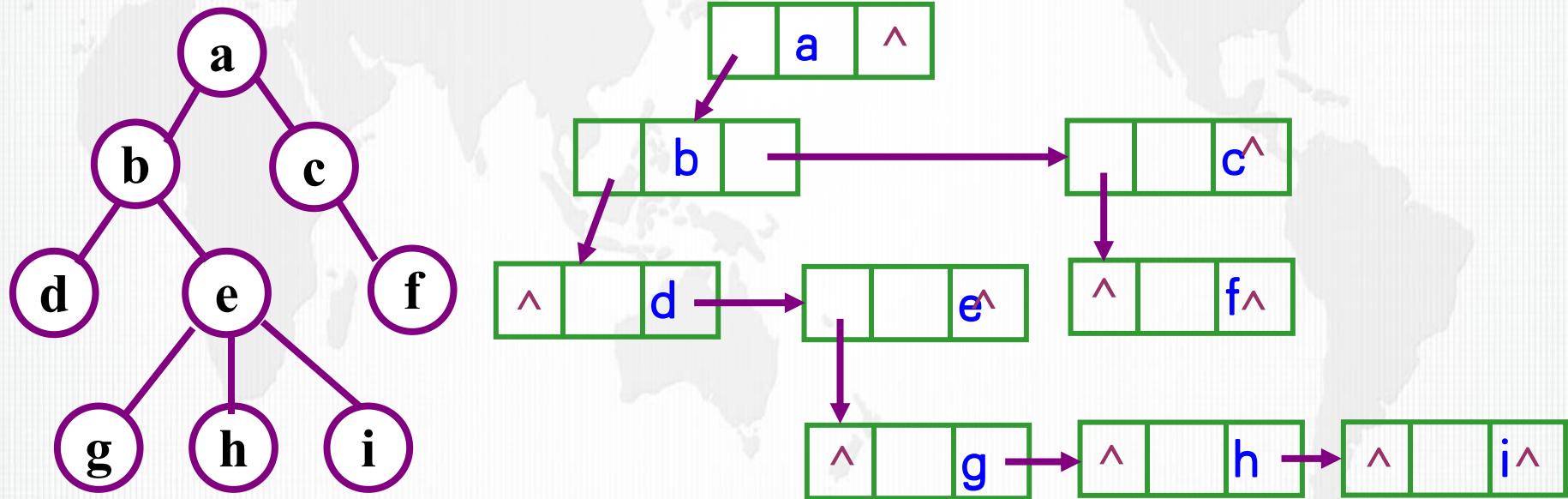
firstchild | **data** | **nextsibling**

```
36 // 孩子兄弟表示法
37 typedef struct CSNode {
38     char data; // 节点数据，可根据需要修改数据类型
39     struct CSNode* firstChild; // 第一个孩子节点的指针
40     struct CSNode* nextSibling; // 右侧兄弟节点的指针
41 } CSNode, * CSTree;
42
```

Implementation

◆ 树的存储结构——孩子兄弟表示法

用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点



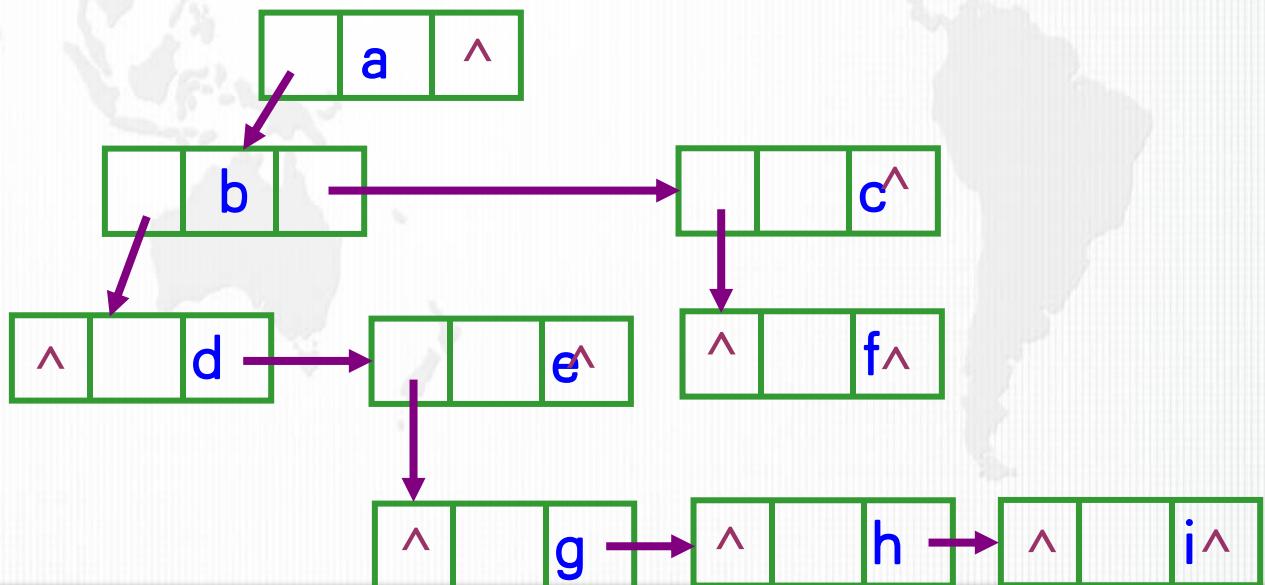
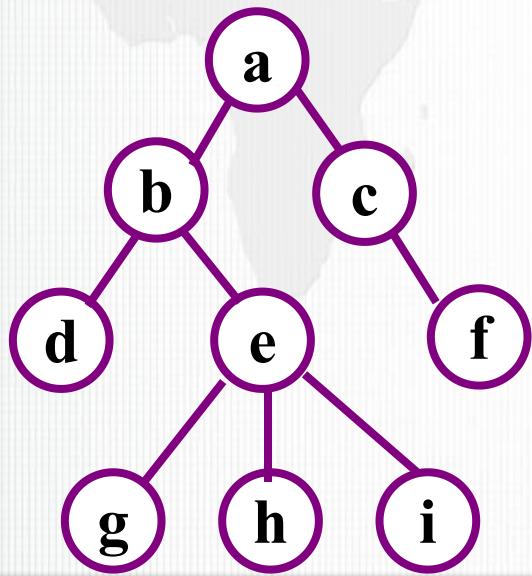
Implementation

◆ 树的存储结构——孩子兄弟表示法

用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点

特点：

1. 操作容易
2. 破坏了树的层次





Implementation

◆ 顺序存储实现：

按完全二叉树的结点层次编号，依次存放二叉树中的数据元素

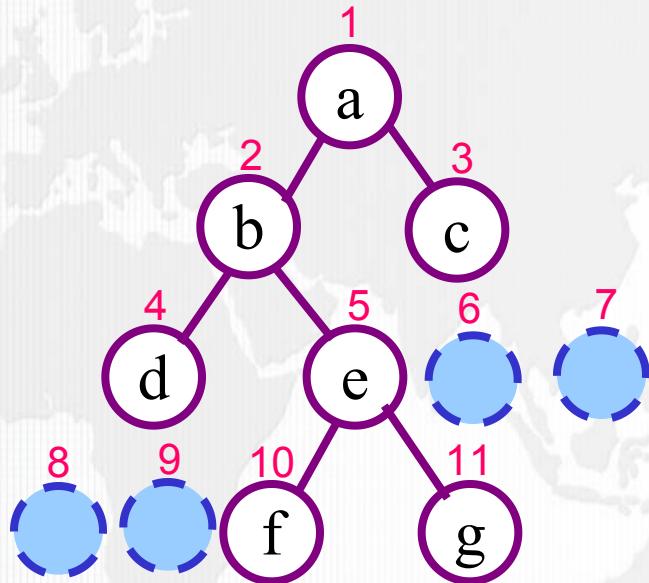
二叉树的顺序存储表示

```
#define Max_Tree_Size 100      //树中结点最大数目  
typedef TElemType SqBiTree[Max_Tree_Size];  
                                // 0号单元存储根结点
```

Implementation

◆ 顺序存储实现：

按完全二叉树的结点层次编号，依次存放二叉树中的数据元素



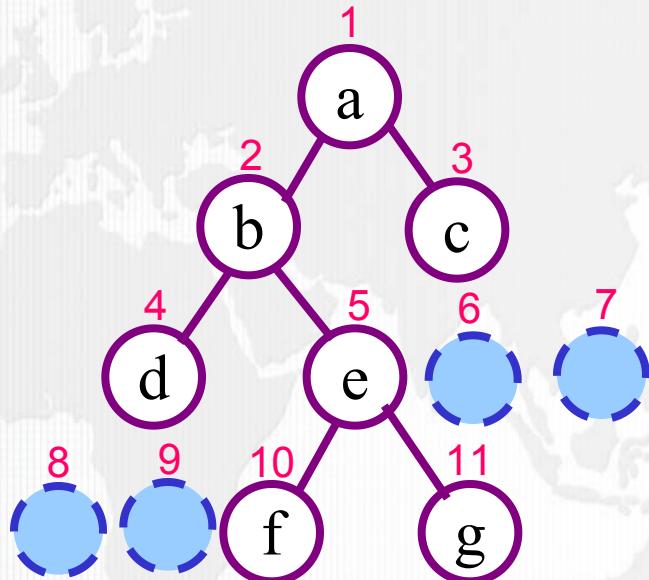
SqBiTree T;

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g

Implementation

◆ 顺序存储实现：

按完全二叉树的结点层次编号，依次存放二叉树中的数据元素



SqBiTree T;

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g



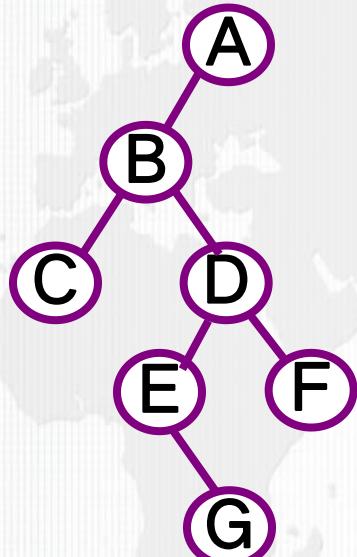
该存储结构能否反映
结点之间的关系？

特点：

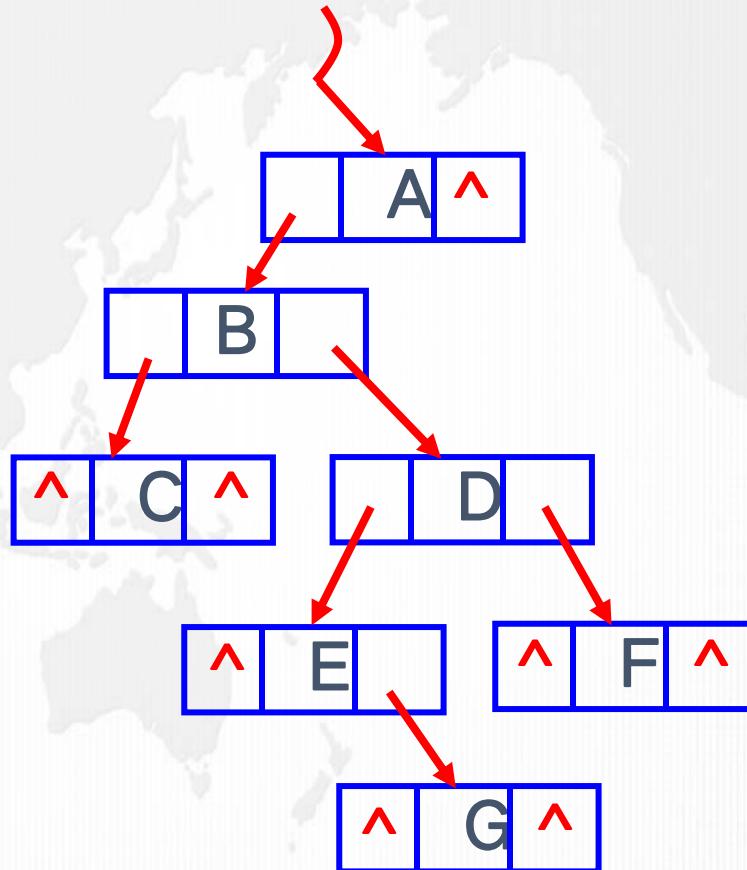
- 结点间的父子关系蕴含在其存储位置中
- 浪费空间，适于存满二叉树和完全二叉树

Implementation

◆ 链式存储实现——二叉链表：



找孩子容易
找双亲难



在n个结点的二叉链表中，有 **n+1** 个空指针域



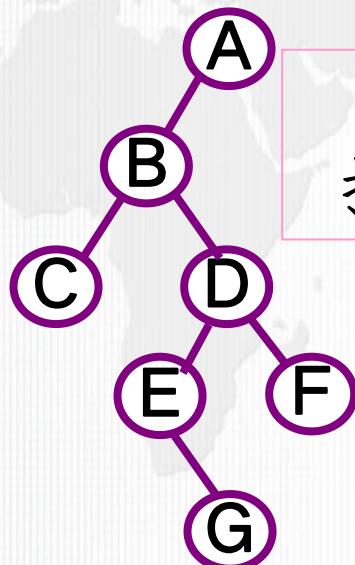
Implementation

◆ 链式存储实现——二叉链表：

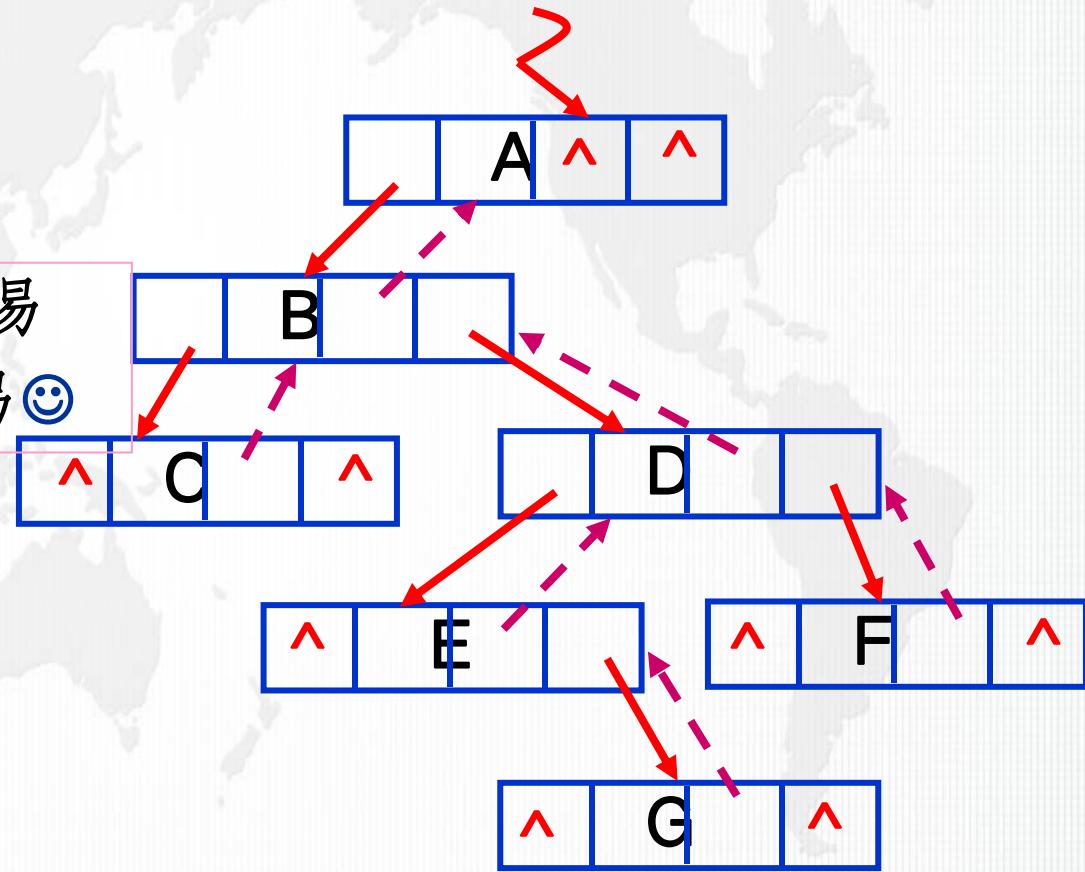
```
// 定义二叉树的节点结构体
typedef struct Node {
    char data; // 节点的数据部分，可以根据需求修改数据类型
    struct Node* left; // 指向左孩子的指针
    struct Node* right; // 指向右孩子的指针
} Node;
```

Implementation

- ◆ 链式存储实现——三叉链表：



找孩子容易
找双亲容易





Implementation

◆ 链式存储实现——三叉链表：

```
12  
13     // 定义三叉链表的节点结构体  
14     typedef struct TriNode {  
15         char data;    // 节点的数据部分，可以根据需求修改数据类型  
16         struct TriNode* left;    // 指向左孩子的指针  
17         struct TriNode* right;   // 指向右孩子的指针  
18         struct TriNode* parent; // 指向父节点的指针  
19     } TriNode;  
20
```

Thanks!



See you in the next session!