

Lecture 18: 树、森林

18.1 树的存储结构

18.1.1 双亲表示法

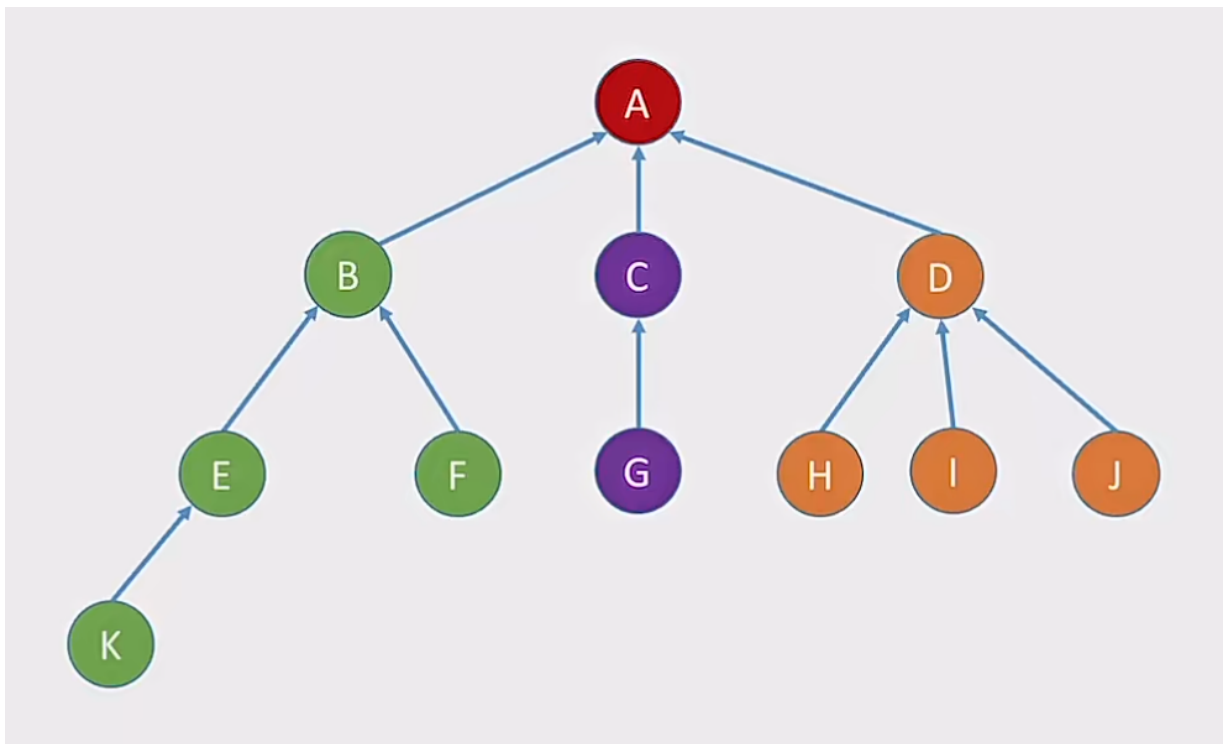
双亲表示法 (顺序存储)

- 概念：
 - 每个结点中保存指向双亲的**指针**；
 - 非根节点的双亲指针 = 父节点在**数组中的下标**；
- 特点：
 - 优点：
 - 查指定的双亲很方便；
 - 还可以存储森林（把多个树的根其 **index 设为 -1** 即可）；
 - 缺点：
 - 查指定结点的孩子只能从头遍历；
 - 找孩子节点不方便；
 - 适合经常找父亲、不经常找孩子；
- 图示：

	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	2
7	H	3
8	I	3
9	J	3
10	K	4
11		
12		
13		

获取笔记

新可加微信: WandD2277



代码实现

```

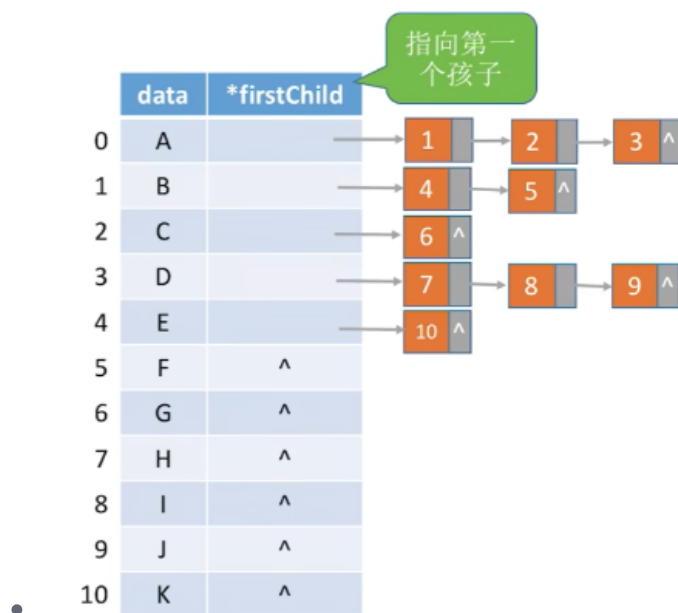
#define MAX_TREE_SIZE 100 //树中最多结点树
typedef struct{           //树的结点定义
    ElemType data;        //数据元素
    int parent;           //双亲位置域，指明双亲在数组中的下标位置
}PTNode;

typedef struct{           //树的类型定义
    PTNode nodes[MAX_TREE_SIZE]; //双亲表示
    int n;                  //结点数
}PTree;
  
```

18.1.2 孩子表示法

孩子表示法 (顺序+链式存储)

- 概念：
 - 用数组顺序存储各个结点，每个结点中保存：
 - 1. 数据元素；
 - 2. 孩子链表头指针；
- 注意：
 - 用孩子表示法存储森林，需要记录多个根的位置；
- 特点：
 - 找孩子节点方便，找父亲节点麻烦；
- 图示：
 - A 节点的孩子是 BCD；



代码实现

```

struct CTNode{
    int child;    //孩子结点在数组中的位置
    struct CTNode *next; //下一个孩子
}

typedef struct{
    ElemType data;
    struct CTNode *firstChild; //第一个孩子
}CTBox;

typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int n,r;    //结点数和根位置
}CTree;

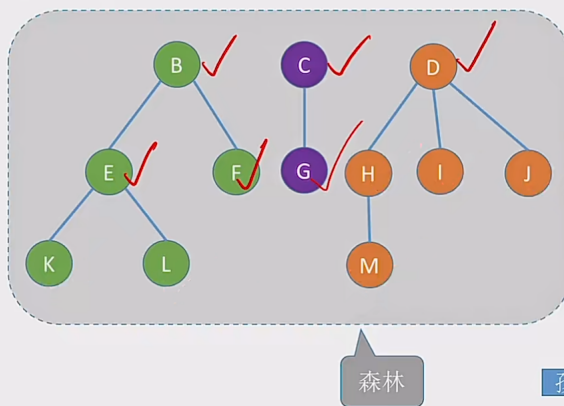
```

18.1.3 孩子兄弟表示法

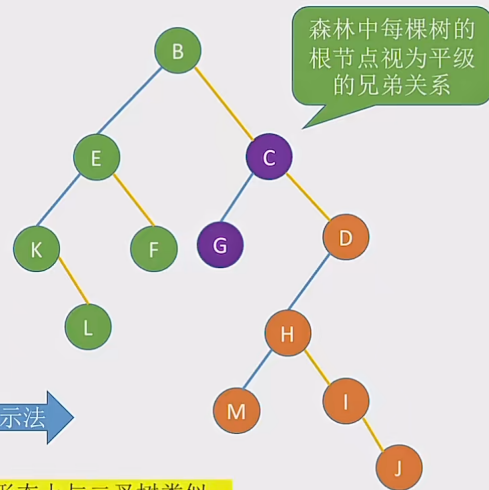
孩子兄弟表示法 (链式存储)

- 概念：
 - 用两个指针，分别指**孩子和右兄弟**；
 - 孩子兄弟表示法和二叉树类似：采用二叉链表实现每个结点内保存数据元素和两个指针，但两个指针的含义与二叉树结点不同；
 - 每个节点**存储数据和两个指针**；
 - 指针一：存储第一个孩子节点；
 - 指针二：存储**右边的第一个兄弟**节点；
- 特点：
 - 优点：可以用我们熟悉的二叉树操作来处理树
- 森林：

森林。森林是 m ($m \geq 0$) 棵互不相交的树的集合



孩子兄弟表示法



- 当使用“孩子兄弟表示法”存储树或森林时，从存储视角来看形态上与二叉树类似。

代码实现

```
//二叉树的结点(链式存储)
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

//数的存储——孩子兄弟表示法
typedef struct CSNode{
    ElemType data;    //数据
    struct CSNode *firstchild,*nextsibling; //第一个孩子和右兄弟指针
    //其中firstchild看做左指针，nextsibling看做右指针
}CSNode,*CSTree;
```

18.2 树和森林的遍历

18.2.1 树的先根遍历

先根遍历

- 概念：
 - 若树非空，先访问根结点，再依次对每颗子树进行先根遍历；
- 注意：
 - 树的先根遍历序列与这棵树相应二叉树的先序序列相同；

代码实现

```
//树的先根遍历
void PreOrder(TreeNode *R){
    if(R!=NULL){
        visit(R); //访问根结点
        while(R还有下一个子树T)
            PreOrder(T); //先根遍历下一棵子树
    }
}
```

18.2.2 树的后根遍历

后根遍历

- 概念：
 - 若树非空，先依次对每棵子树进行后根遍历，最后再访问根结点；
- 注意：
 - 树的后根遍历序列与这棵树相应二叉树的中序序列相同。(因为根节点没有兄弟)

代码实现

```
//树的后根遍历
void PostOrder(TreeNode *r){
    if(R!=NULL){
        while(R还有下一个子树T)
            PostOrder(T);    //后根遍历下一棵子树
        visit(R);    //访问根节点
    }
}
```

18.2.3 森林的先序遍历

先序遍历森林

- 若森林为非空，则按如下规则进行遍历：
 - 1. 访问森林中**第一棵树的根结点**。
 - 2. 先序遍历第一棵树中根结点的子树森林。
 - 3. 先序遍历去除第一棵树之后剩余的树构成的森林。
- 注意：
 - 对森林的先序遍历序列和对二叉树的先序遍历序列相同，可以先转换，后求；

18.2.4 森林的中序遍历

中序遍历森林

- 若森林为非空，则按如下规则进行遍历：
 - 1. 中序遍历森林中第一棵树的根结点的子树森林；
 - 2. 访问第一棵树的根结点；
 - 3. 中序遍历去除第一棵树之后剩余的树构成的森林；

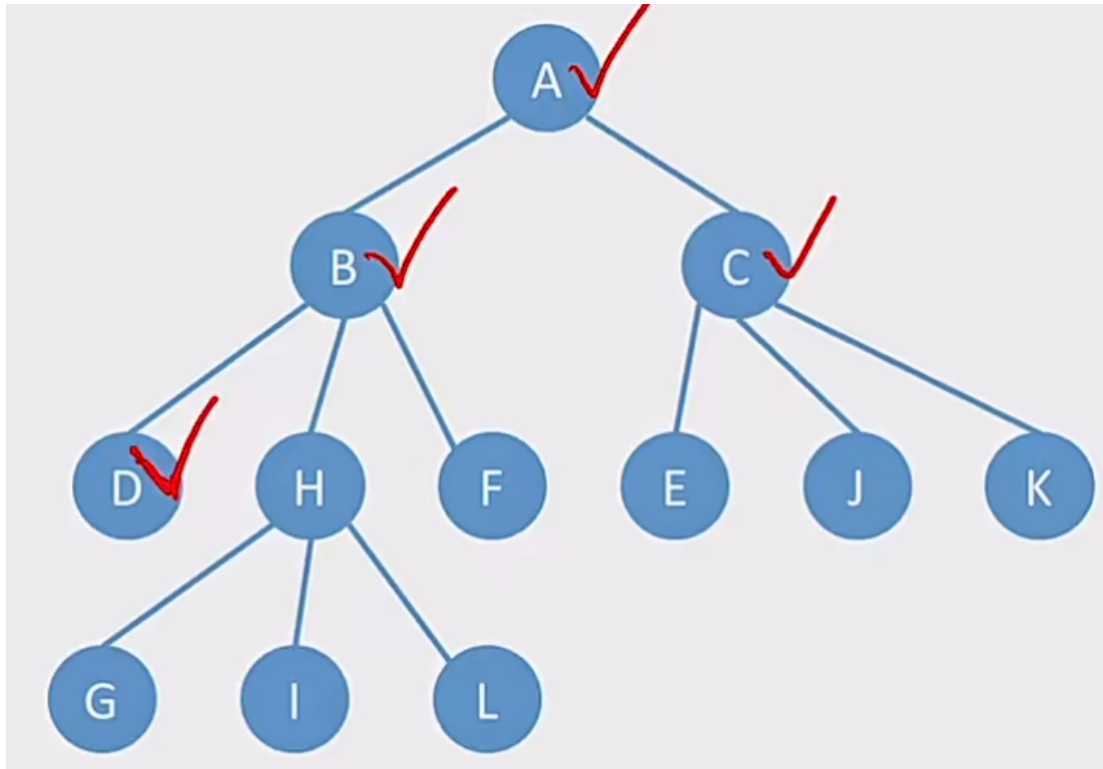
18.3 树、森林与二叉树的转换

18.3.1 树 -> 二叉树

转换方法

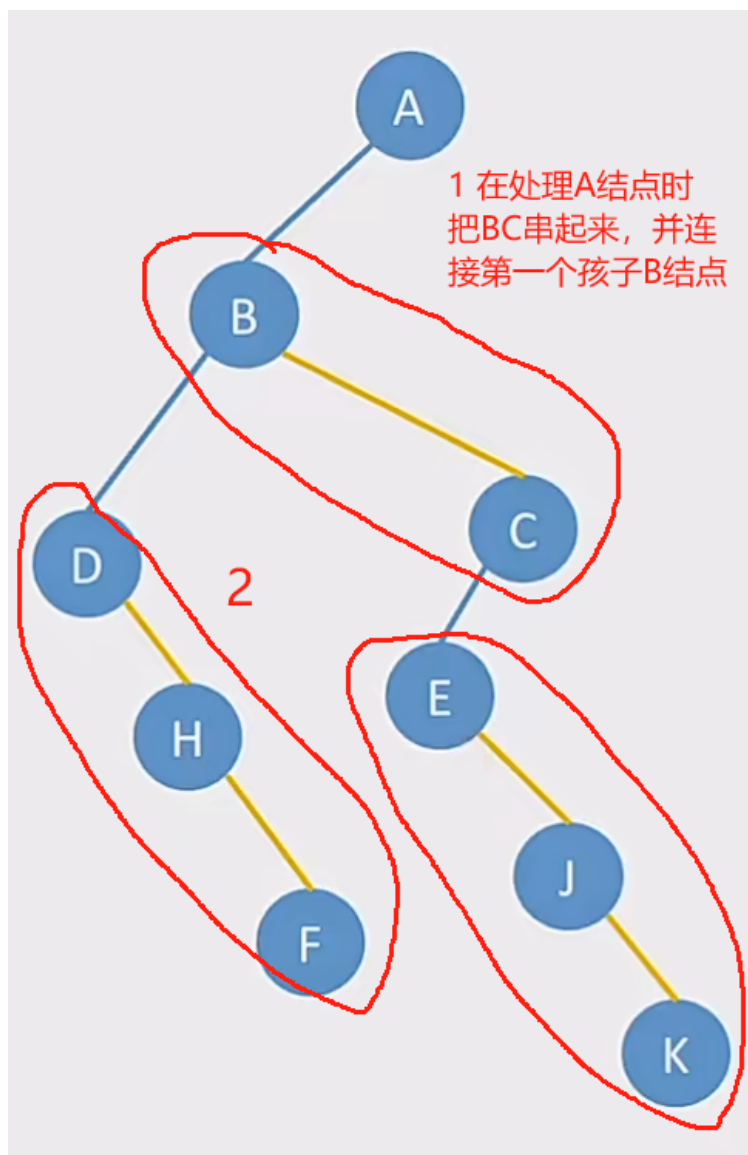
- 核心：
 - 用孩子兄弟表示法存储森林时，将森林中每棵树的根节点视为平级的兄弟关系；

- 概念：
 - 1. 先在二叉树中，画一个根节点；
 - 2. 按“树的层序”，一层层的顺序的依次处理每个结点；
- 处理方法：
 - 如果当前处理的节点在树中有孩子，就把所有孩子结点“用右指针串成糖葫芦”，并在二叉树中把第一个孩子挂在当前结点的左指针下方；
- 图示：
 - 处理 ABC 前：

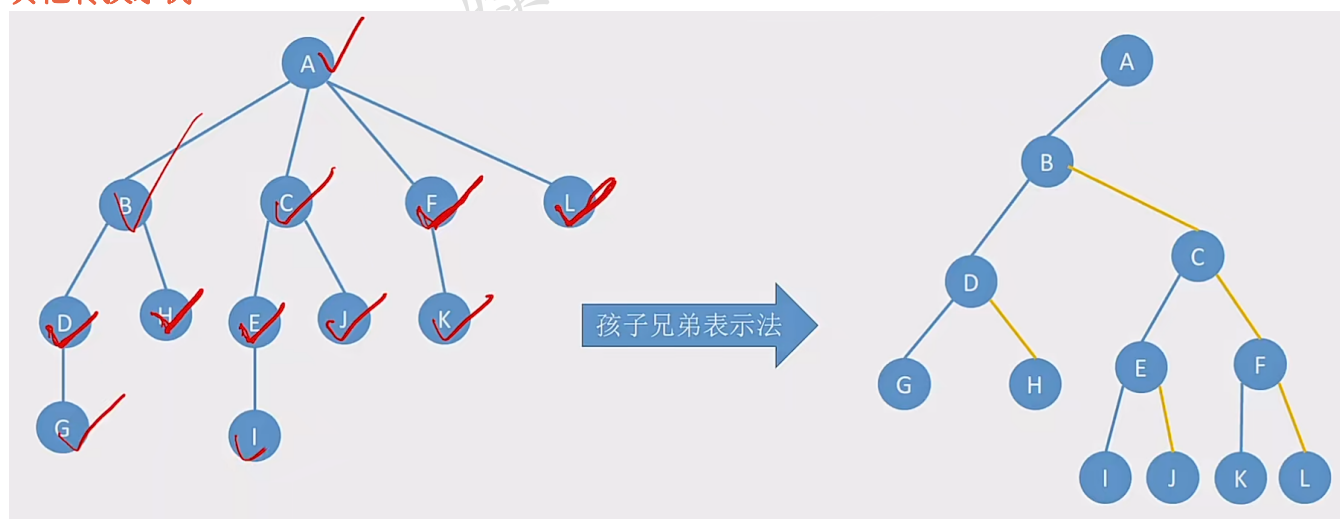


- 处理 ABC 后：

获取笔记持续更新



其他转换示例



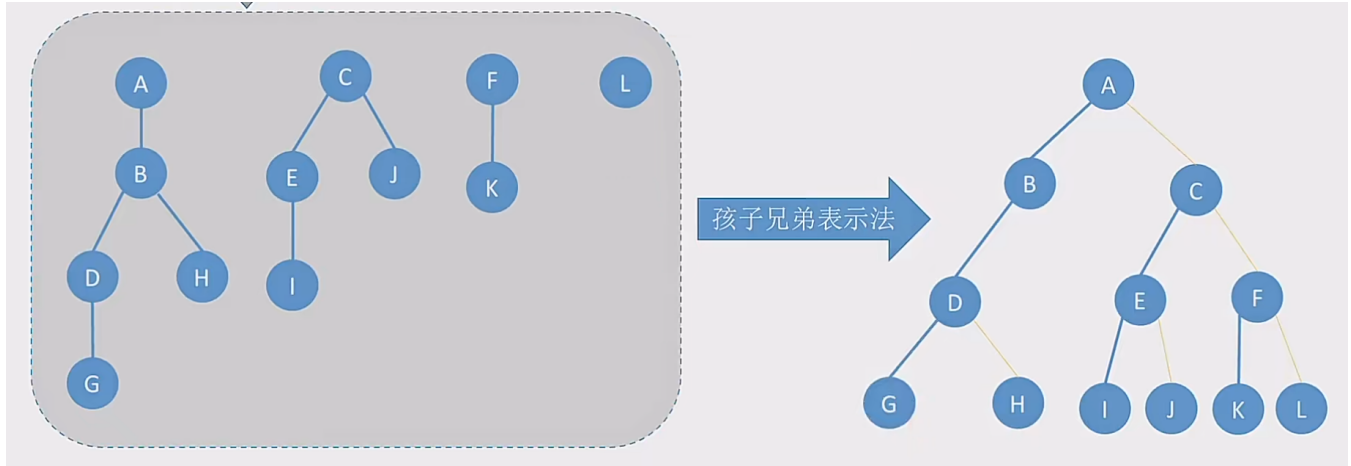
18.3.2 森林 -> 二叉树

转换方法

- 概念：
 - 1. 先把所有树的根结点画出来，在二叉树中用右指针串成糖葫芦；

- 2. 按“森林的层序”依次处理每个结点；
- 处理方法：
 - 处理一个结点的方法是：如果当前处理的结点在树中有孩子，就把所有孩子结点“用右指针串成糖葫芦”，并在二叉树中把第一个孩子挂在当前结点的左指针下方；
- 注意：
 - 森林中各棵树的根节点视为平级的兄弟关系；
 - 在二叉树中的父子结点，在森林中可能是兄弟结点，也可能是父子结点；

转化示例

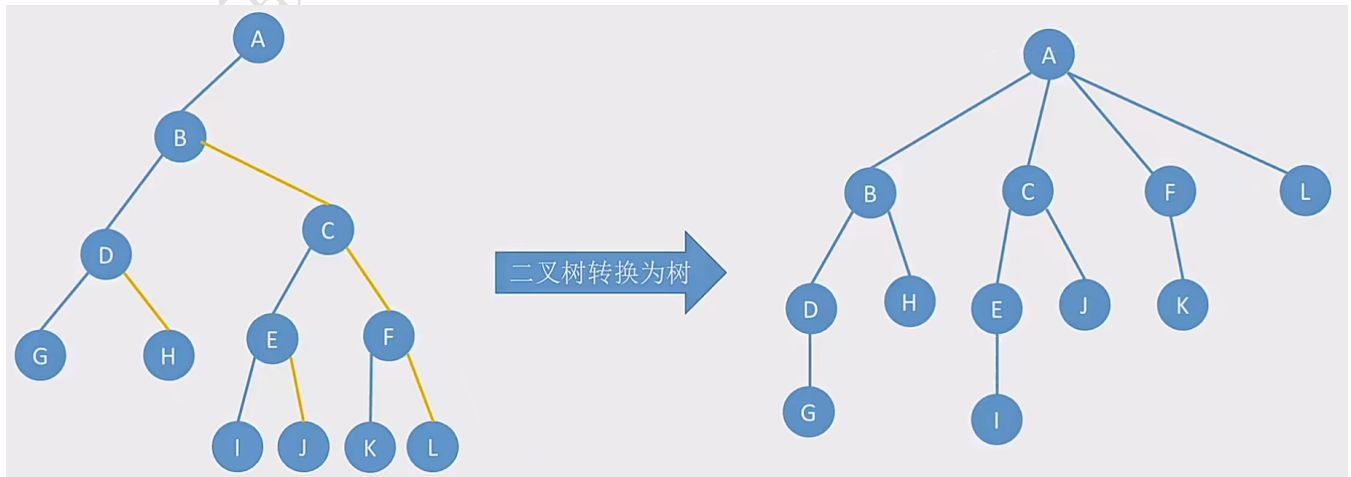


18.3.3 二叉树 -> 树

转换方法

- 概念：
 - 1. 先画出树的根节点；
 - 2. 从树的根节点开始，按“树的层序顺序”一层一层的恢复每个结点的孩子；
- 处理方法：
 - 在二叉树中，如果当前处理的结点有左孩子，就把左孩子和“一整串”右指针糖葫芦拆下来，按顺序挂在当前结点的下方；

转换举例



18.3.4 二叉树 -> 森林

转换方法

- 概念：
 - 1. 先把二叉树的根节点和“一整串右指针糖葫芦”拆下来，作为多棵树的根节点；
 - 2. 按“森林的层序”恢复每个结点的孩子；
- 核心：
 - 糖葫芦 $\rightarrow$ 右指针 $\rightarrow$ 平级关系；
- 处理方法：
 - 在二叉树中，如果当前处理的结点有左孩子，就把左孩子和“一整串右指针糖葫芦”拆下来，并按顺序挂在当前结点的下方；

转换示例

