算法与数据结构
Lecture 12: 二叉树

郝家胜

hao@uestc.edu.cn

School of Automation Engineering,
University of Electronic Sci. & Tech. of China

- 内容略少，尚可补充约15分钟材料

内容回顾

- 树的递归定义
- 树的基本概念
- 二叉树的概念及其基本形态

树的定义

- 是n (n>0)个结点的有限集合T。
- 在一棵树(n>0)中：

- ①有且仅有一个称为根root的结点；
- ②其余结点元素可分为m (m ≥ 0)个互不相交的有限集T_1，T_2，…，T_m，
- 其中每个集合本身又是一颗树，称为根的子树。

内容提要

- 二叉树的遍历
- 二叉树的遍历算法
- 二叉树的存储结构

二叉树的遍历

- 指用一定的规律访问树的每一个节点，

  且每个节点均恰好被访问一次。

  访问该结点时可对该结点进行操作

- 二叉树包括3个基本单元：根节点、左子树、右子树，

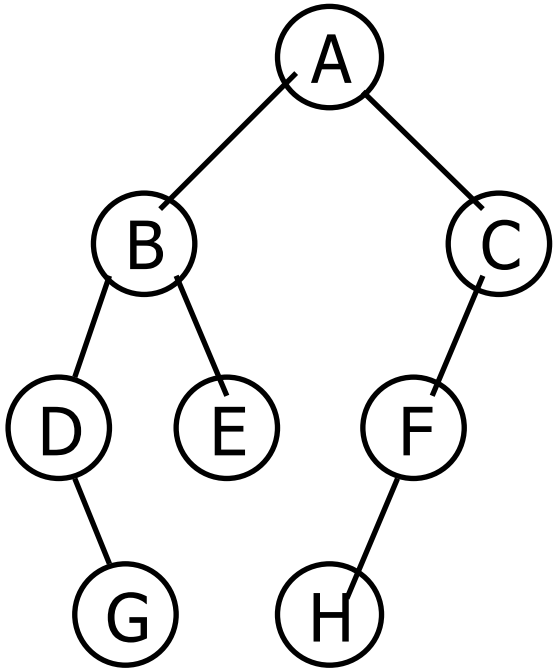
若能依次遍历这3个部分，就遍历了整个二叉树。

- 根据遍历这3个部分的顺序不同，分为三种遍历方式：
  - 先序遍历（先根序DLR）
  - 中序遍历（中根序LDR）
  - 后序遍历（后根序LRD）

限定对子树的访问是先左后右

先序遍历（先根序DLR）

- 先访问根节点
- 访问左子树（按先序遍历）
- 访问右子树（按先序遍历）
- 根 ⇨ 先序左子树 ⇨ 先序右子树



A (B (D ( ) (G)) (E)) (C (F (H) ( )) ( ))

先序遍历序列为：A B D G E C F H

- 中序遍历

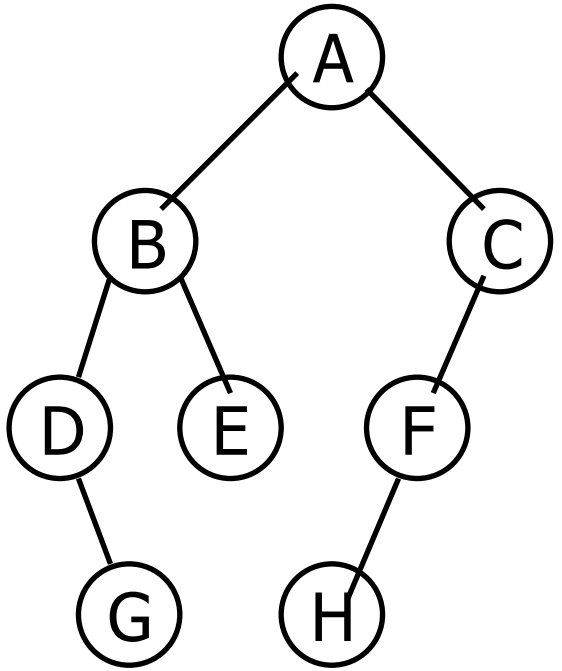
  （中根序LRD）

  - 中序左子树 ⇨ 根 ⇨ 中序右子树
  - D G B E A H F C

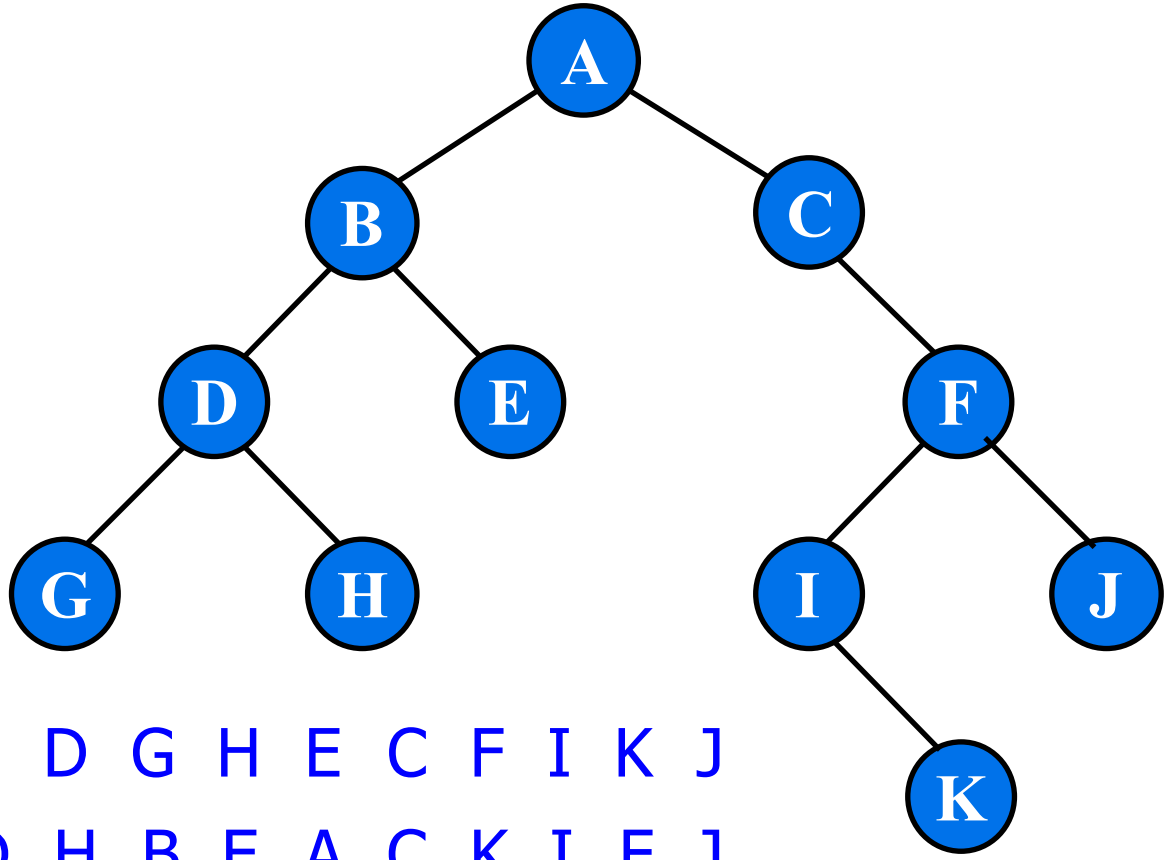
- 后序遍历

  （后根序LRD）

  - 后序左子树 ⇨ 后序右子树 ⇨ 根
  - G D E B H F C A



课堂练习：写出下面这棵树的三种遍历序列



先序：A B D G H E C F I K J

中序：G D H B E A C K I F J

后序：G H D E B K I J F C A

根据遍历序列构造二叉树

- 利用**中序遍历序列**，并结合**先序列遍历序列和后序遍历序列中任何一个**序列能重新构造二叉树。

中序遍历序列形式：

•<左子树所有结点> <根> <右子树所有结点>

先序遍历序列形式：

•<根><左子树所有结点> <右子树所有结点>

后序遍历序列形式：

•<左子树所有结点> <右子树所有结点><根>

- 思考：由先序遍历结果和后序遍历结果能否重新构造二叉树？

二叉树构造步骤

第一步：

- **从先序遍历序列中取出第一个结点，肯定为根；**
- **然后在中序遍历序列中找出根结点；**
- **则中序遍历序列根结点前为左子树的中序遍历序列，根结点后为右子树的中序遍历序列。**

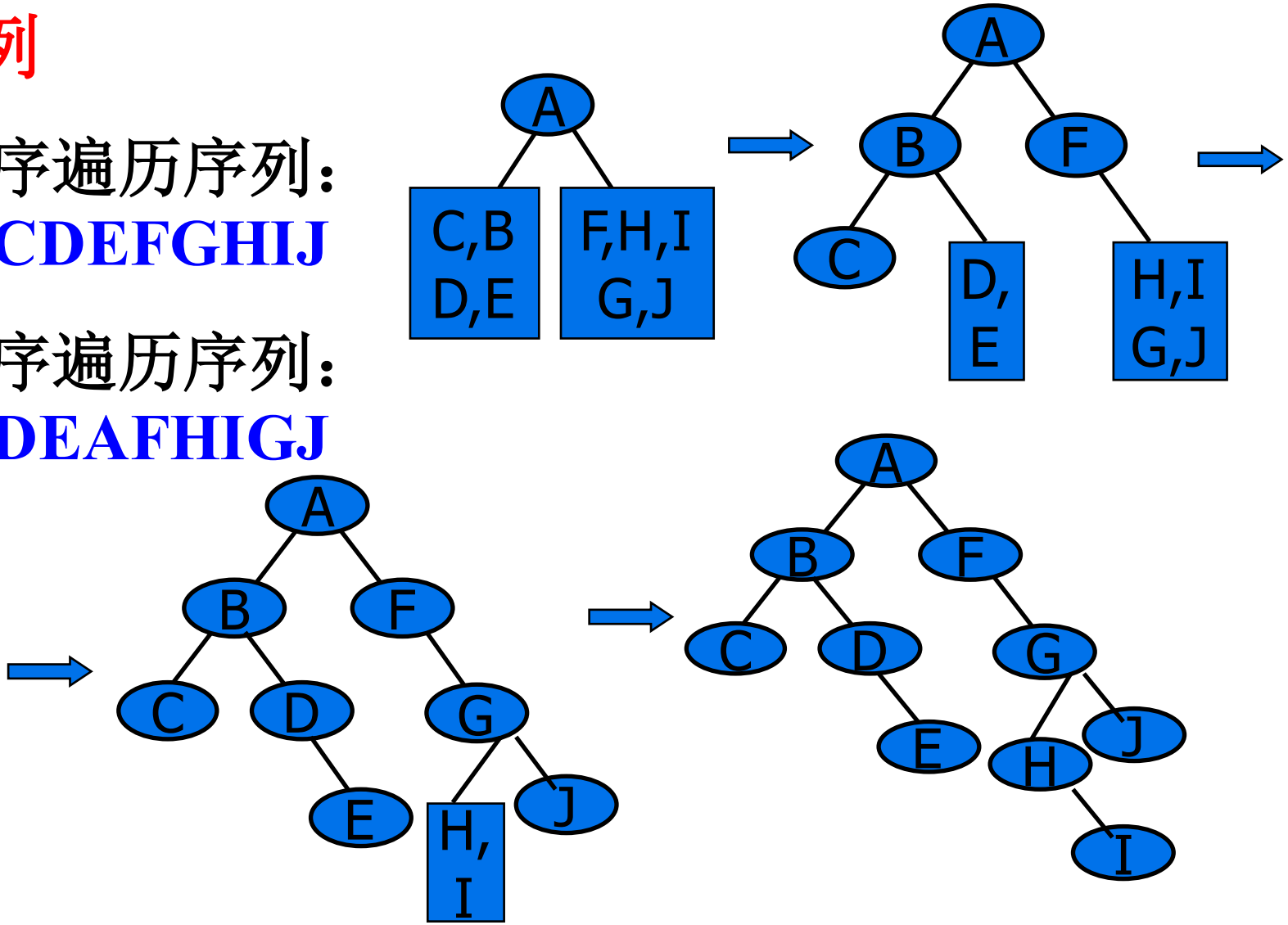
第二步：

- 对根的左子树先序、中序遍历序列
- 及根的右子树先序、中序遍历序列
- 再执行第一步，直至找出所有叶子结点为止。

•例

•先序遍历序列：
ABCDEFGHIJ

•中序遍历序列：
CBDEAFHIGJ



二叉树的抽象数据类型

- 数学模型
- 数据定义

  $T: \{T_1, T_2, \ldots, T_m\}, m \geq 0$

- 接口声明
  - ROOT(T)
  - LEFT(x)
  - RIGHT(x)

先序遍历算法

- 算法分析
  - 访问根节点root
  - 先序遍历左子树
  - 先序遍历右子树

递归方式

步骤：

if 当前节点为空则

　　树空，结束遍历

else

　　处理当前节点

　　递归处理左子树

　　递归处理右子树

递归结束条件：

访问的节点为空

先序遍历算法伪代码

```
PREORDER-TREE-WALK(x)
1  ▷从节点x开始先序遍历
2  if x ≠ NIL then
3        visit(x)
3        PREORDER-TREE-WALK(LEFT(x))
4        PREORDER-TREE-WALK(RIGHT(x))
6  end
```

中序遍历算法

if 指向根节点的指针为空则树空，结束遍历

else

{ 递归处理左子树

处理当前节点

递归处理右子树

}

```
INORDER-TREE-WALK(x)
1  ▷从节点x开始中序遍历
2  if x ≠ NIL then
3        INORDER-TREE-WALK(LEFT(x))
4        visit(x)
5        INORDER-TREE-WALK(RIGHT(x))
6  end
```

后序遍历算法

if 指向根节点的指针为空则树空，结束遍历

else

{ 递归处理左子树

递归处理右子树

处理当前节点

}

```
POSTORDER-TREE-WALK(x)
1  ▷从节点x开始中序遍历
2  if x ≠ NIL then
3        POSTORDER-TREE-WALK(LEFT(x))
4        POSTORDER-TREE-WALK(RIGHT(x))
5        visit(x)
6  end
```

三种遍历算法的比较

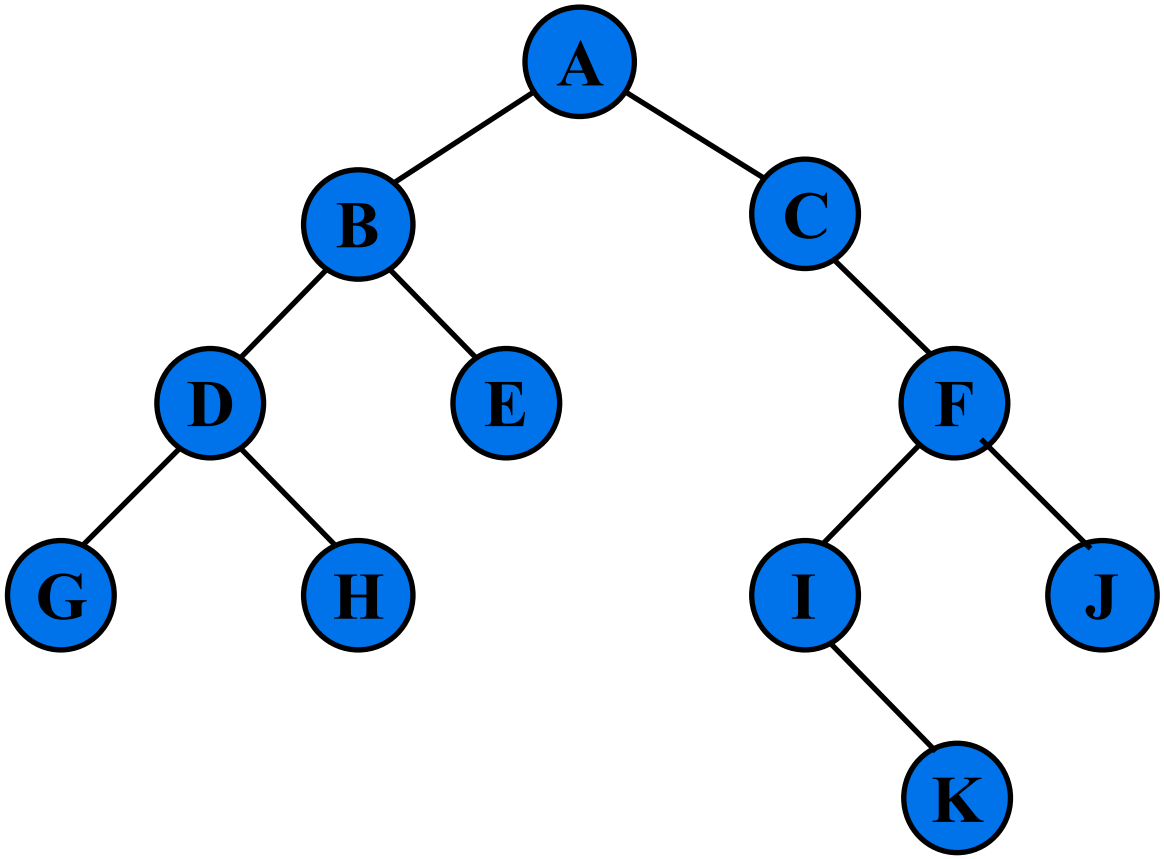
```
2  if x ≠ NIL then
3       visit(x)
3       PREORDER-TREE-WALK(LEFT(x))
4       PREORDER-TREE-WALK(RIGHT(x))
6  end
```

```
2  if x ≠ NIL then
3       POSTORDER-TREE-WALK(LEFT(x))
4       POSTORDER-TREE-WALK(RIGHT(x))
5       visit(x)
6  end
```

```
2  if x ≠ NIL then
3       INORDER-TREE-WALK(LEFT(x))
4       visit(x)
5       INORDER-TREE-WALK(RIGHT(x))
6  end
```

思考

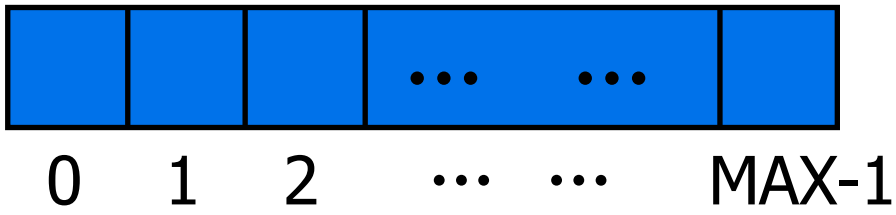
- 逐层遍历如何实现？

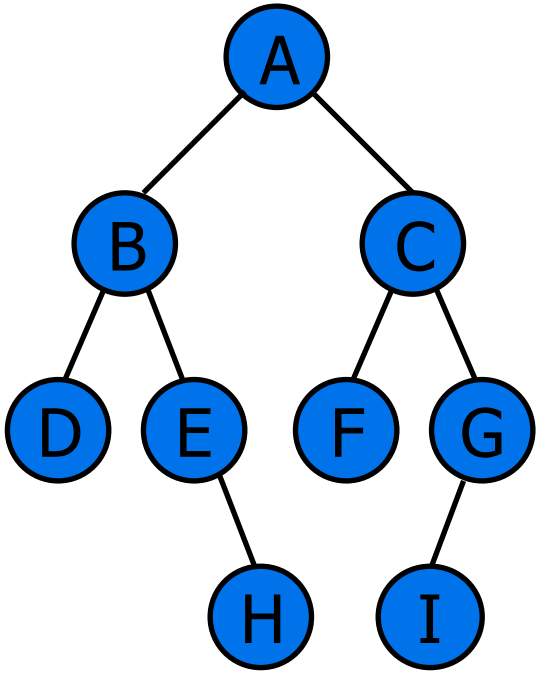


二叉树的存储结构

二叉树的存储结构

- **顺序存储**
  - 所有节点存储到连续单元（数组）





问题的关键：

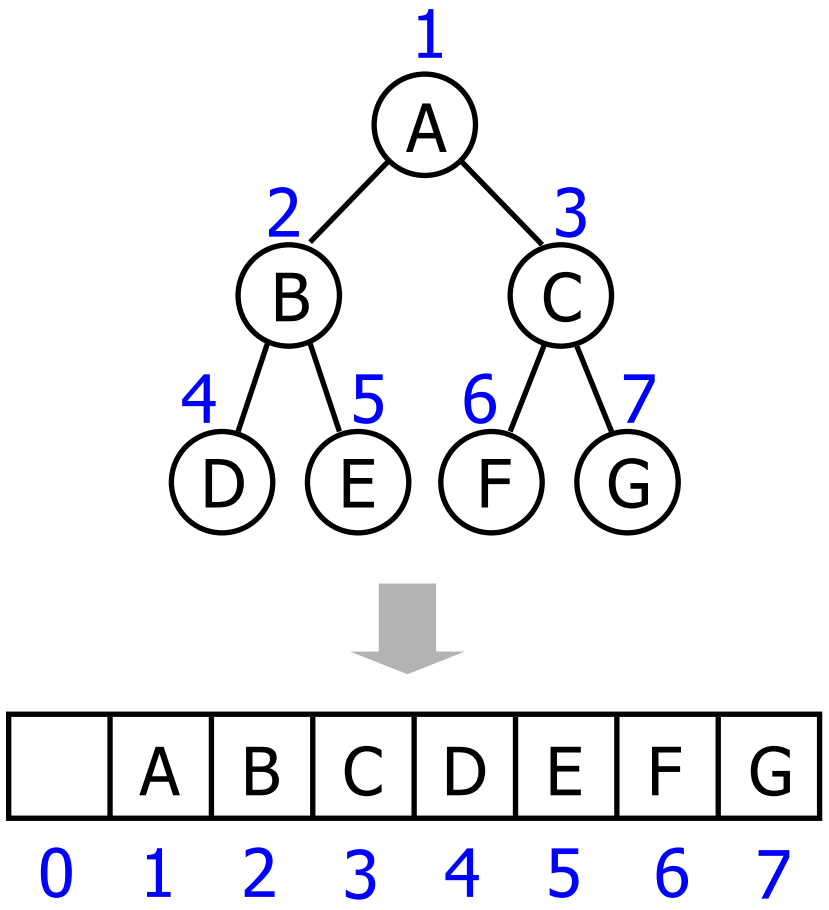
二叉树中的节点按怎样的次序编号，来反映节点之间的逻辑关系

顺序存储二叉树

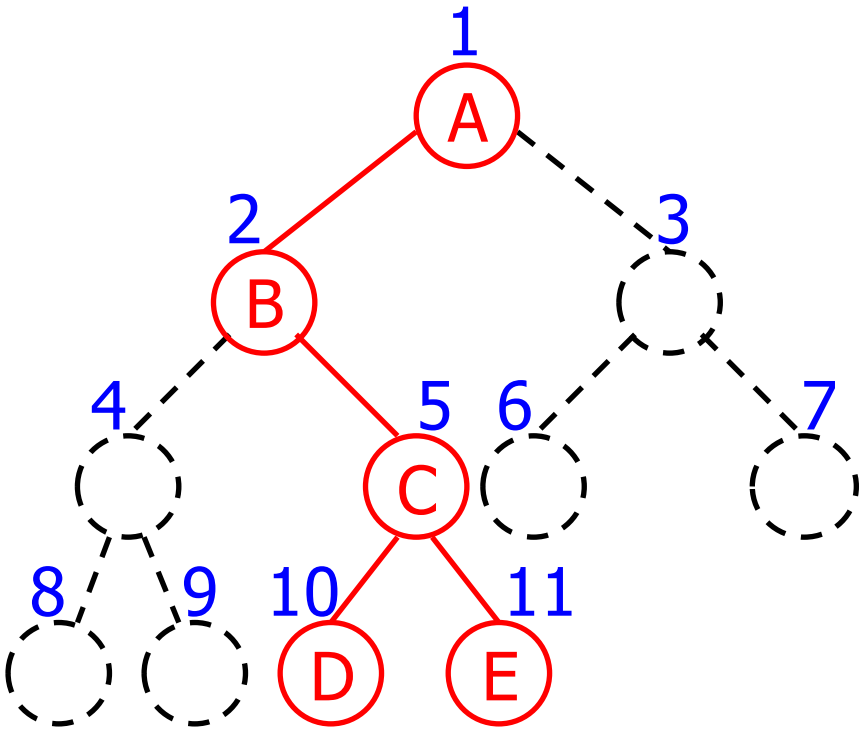
- 在满（完全）二叉树中
- 按从上到下，从左到右的顺序
- 对所有节点由1开始编号
- 再根据编号存入相应的下标中

若已知某节点的编号为n

- 其左子节点编号为2n
- 右子节点编号为2n+1
- 父节点为[n/2]（取整）

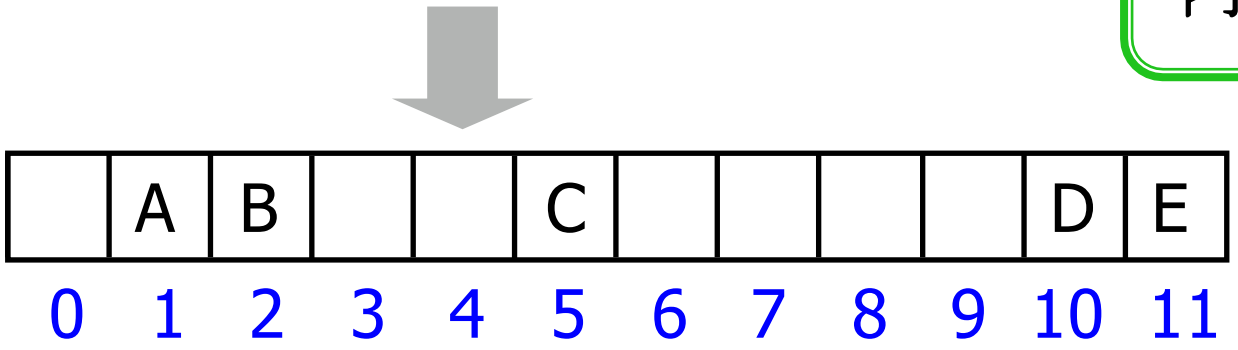


•满二叉树和完全二叉树中结点的序号可以唯一地反应出结点之间的逻辑关系。



对于一般二叉树，将其每个节点编号与满二叉树对应进行存储

这种存储方式，可能会造成大量内存空间的浪费

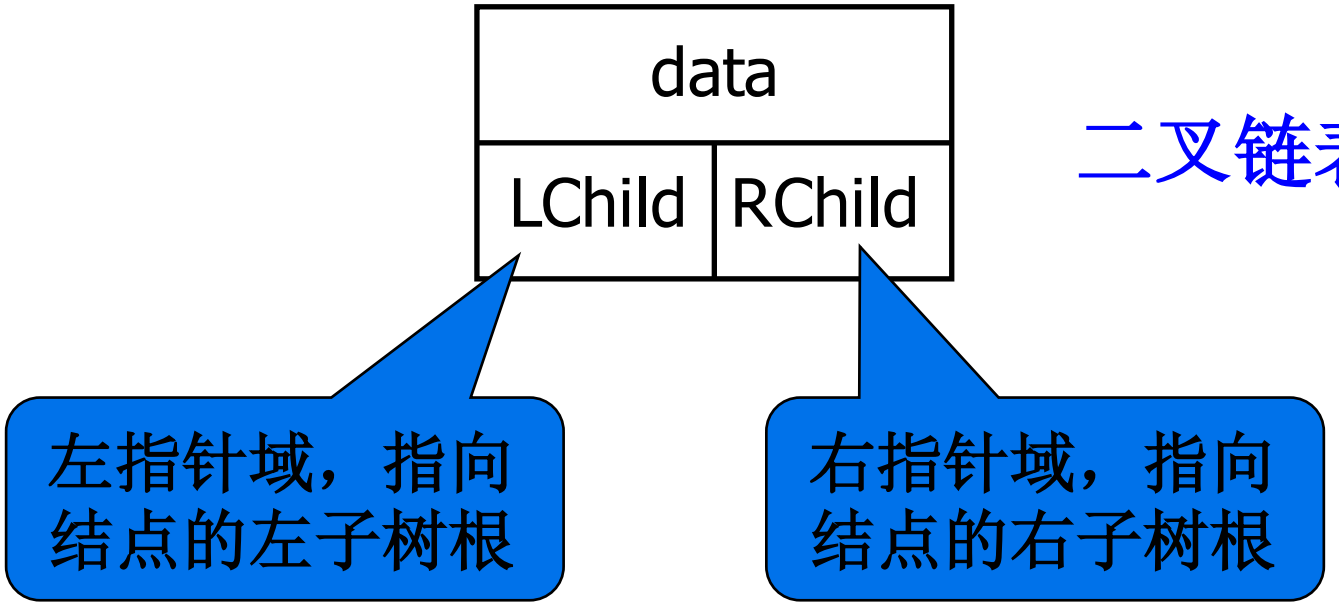


若经常需要插入与删除树中结点时，顺序存储方式不是很好.

链式存储二叉树

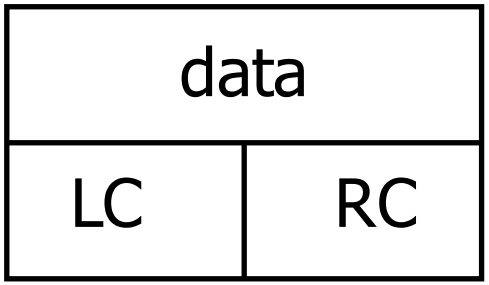
- 通过指针“指向”左右子树

  二叉树的一个节点结构：



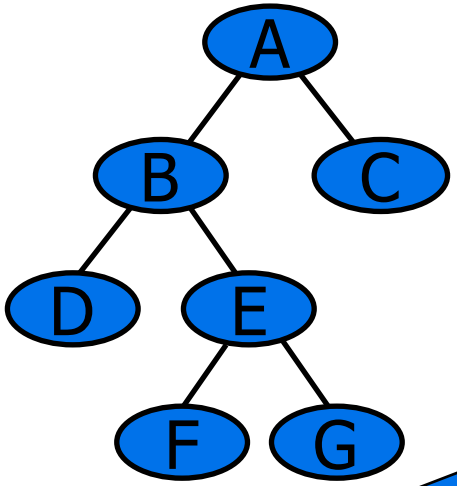
二叉链表

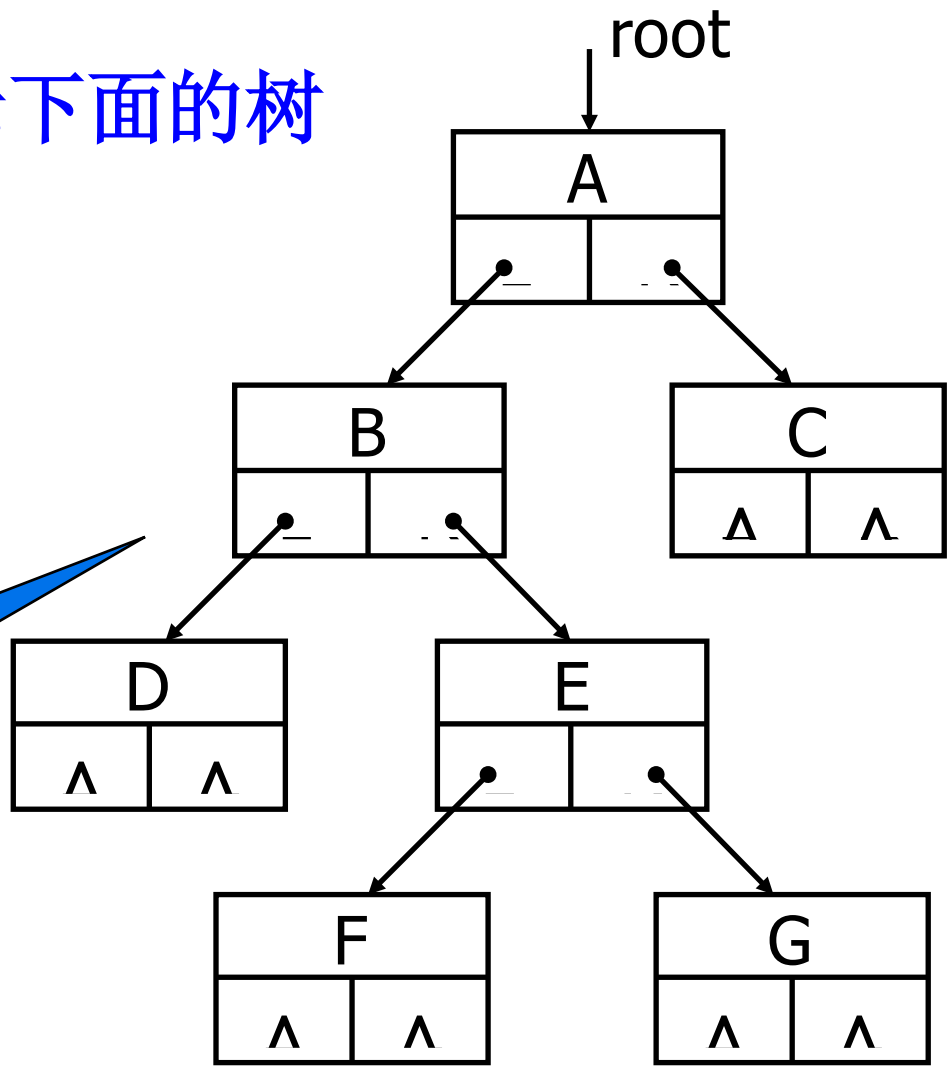
节点结构的C语言描述




```
struct bnode
{    elemtype   data;
     struct  bnode   *LC, *RC;
};
```

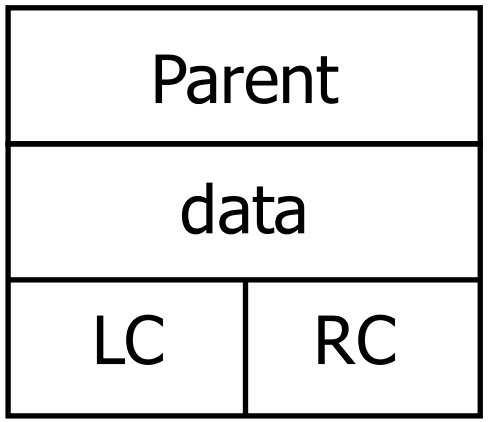
用二叉链表结构表示下面的树





- 二叉链表上可以方便地从某结点出发，找到它的一个子结点。
- 找父结点需从根开始

为了能够从某个节点直接寻找其父节点，采用如下的存储结构：



三叉链表

```
/*节点结构的C语言描述*/
struct   bnode
{  elemtype   data;
    struct  bnode   *Parent;
    struct  bnode   *LC;
    struct  bnode   *RC;
};
```

二叉树的存储结构

- **顺序存储**
  - 所有节点存储到连续单元（数组）
- **链式存储**
  - 通过指针“指向”左右子树

说明：

在不同的存储结构实现二叉树操作的方法也不同

采用什么存储结构，除根据二叉树的形态之外，还应该考虑需要进行何种操作。

小 结

- 二叉树的遍历
- 二叉树的遍历运算
- 二叉树的存储结构