课前签到

数据结构与算法2023秋季

微信扫一扫，使用小程序

1. 微信扫码+实名
2. 点击今日签到

签到时间：

9:45~10:15

签到地方：

珠海校区-教学大楼-C407

人脸识别；智能定位

上课纪律

上课期间手机静音：

1. 关闭手机
2. 飞行模式

数据结构与算法
树

余建兴
中山大学人工智能学院

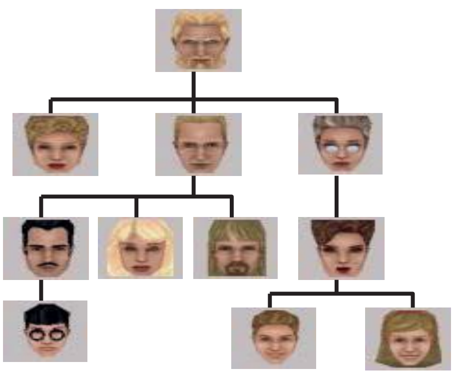
提纲

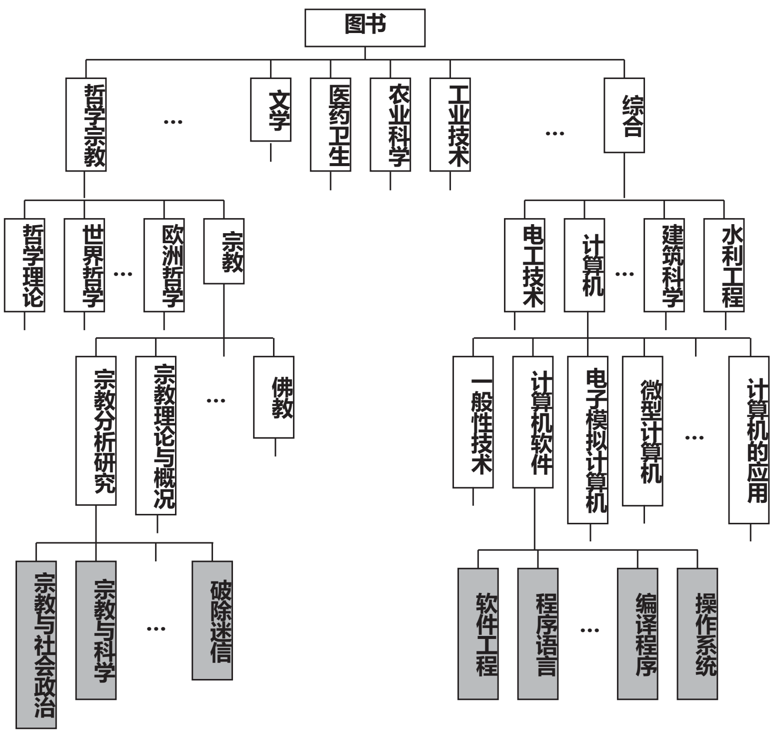
1. 树与树的表示
2. 二叉树及存储结构
3. 二叉树的遍历
4. 应用实例

3.1 树与树的表示

什么是树

- 客观世界中许多事物存在层次关系
  - 人类社会家谱
  - 社会组织结构
  - 图书信息管理





什么是树

分层次组织在管理上具有更高的效率!

数据管理的基本操作之一：查找

如何实现有效率的查找？

查找

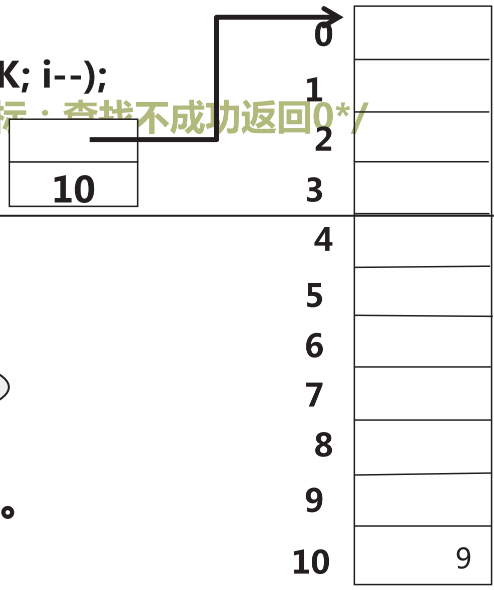
查找：根据某个给定关键字K，从集合R中找出关键字与K相同的记录

- 静态查找：集合中记录是固定的
    - 没有插入和删除操作，只有查找
- 动态查找：集合中记录是动态变化的
    - 除查找，还可能发生插入和删除

静态查找

- 方法1：顺序查找

```
int SequentialSearch (StaticTable *Tbl, ElementType K)
{   /*在表Tbl中查找关键字为K的数据元素，数据元素：Tbl[1]~Tbl[n]*/
  int i;
   Tbl->Element[0] = k; /*建立哨兵*/
   for( i = Tbl->Length; Tbl->Element[i] != K; i--);
   return i; /*查找成功返回数据元素所在单元下标；查找不成功返回0*/
}
```



下标为0的存储空间是临时工作空间

是否需要 i>=0 ?

顺序查找算法的时间复杂度为$O(n)$。

静态查找

- 方法2：二分查找（Binary Search)
  - 假设n个数据元素的关键字满足有序(比如：小到大)

$$k_1 < k_2 < \cdots < k_n$$

并且是连续存放(数组)，那么可以进行二分查找

静态查找

[例] 假设有13个数据元素，按关键字由小到大顺序存放．二分查找关键字为444的数据元素过程如下：

5	16	39	45	51	98	100	202	226	321	368	444	501
1	2	3	4	5	6	7	8	9	10	11	12	13

left mid right

1、left = 1, right = 13; mid = (1+13)/2 = 7: 100 < 444;

2、left = mid+1=8, right = 13; mid = (8+13)/2 = 10: 321 < 444;

3、left = mid+1=11, right = 13; mid = (11+13)/2 = 12: 444 = 444，查找结束

静态查找

[例] 仍然以上面13个数据元素构成的有序线性表为例二分查找关键字为 43 的数据元素如下：

5	16	39	45	51	98	100	202	226	321	368	444	501
1	2	3	4	5	6	7	8	9	10	11	12	13

left mid right

1、left = 1, right = 13; mid = (1+13)/2 = 7: 100 > 43;

2、 left = 1, right = mid-1= 6; mid = (1+6)/2 = 3: 39 < 43;

3、left = mid+1=4, right = 6; mid = (4+6)/2 = 5: 51 > 43;

4、left = 4, right = mid-1= 4; mid = (4+4)/2 = 4: 45 > 43;

5、left = 4, right = mid-1= 3; left > right ? 查找失败，结束。

二分查找算法演示

二分查找算法

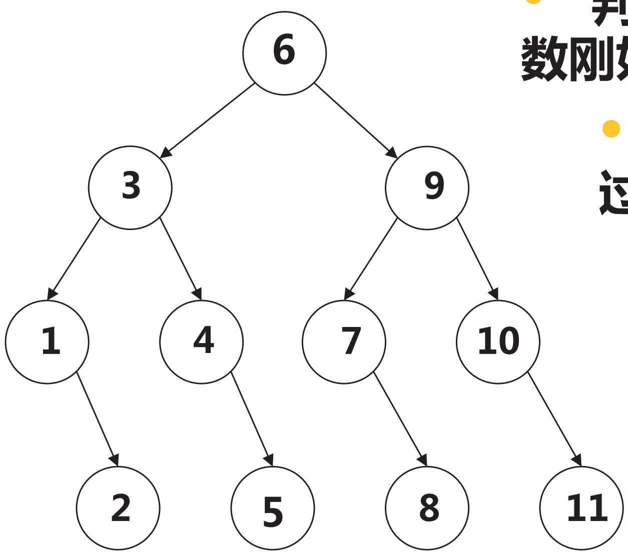
```
int  BinarySearch ( StaticTable * Tbl, ElementType K)
{   /*在表Tbl中查找关键字为K的数据元素*/
    int left, right, mid, NoFound=-1;

    left = 1;                        /*初始左边界*/
    right = Tbl->Length;   /*初始右边界*/
    while ( left  <= right )
    {
        mid = (left+right)/2;  /*计算中间元素坐标*/
        if( K < Tbl->Element[mid])        right = mid-1;    /*调整右边界*/
        else if( K > Tbl->Element[mid])  left = mid+1;      /*调整左边界*/
        else  return  mid;   /*查找成功，返回数据元素的下标*/
    }
    return  NotFound;      /*查找不成功，返回-1*/
}
```

- **二分查找算法的时间复杂度为O(logN)**

二分查找算法

- 11个元素的二分查找判定树



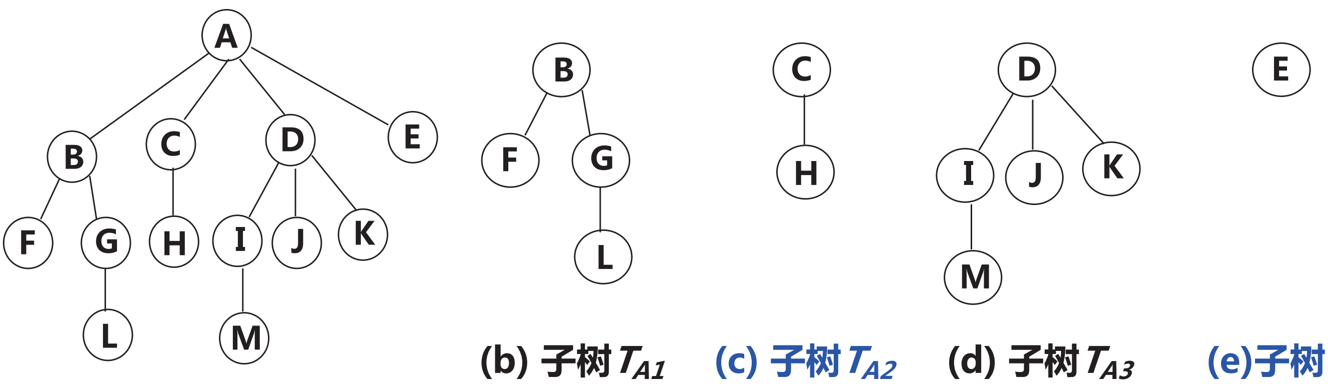
11个元素的判定树

- 判定树上每个结点需要的查找次数刚好为该结点所在的层数
- 查找成功时查找次数不会超过判定树的深度
- ASL = (4*4+4*3+2*2+1)/11 = 3
- n个结点的判定树的深度为$[\log_2 n]+1$
- 折半查找的算法复杂度为$O(\log_2 n)$

二分查找的启示？

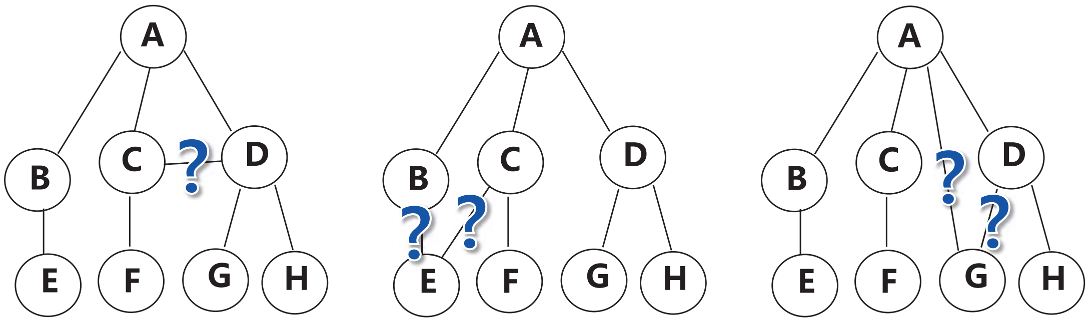
树的定义

- 树(Tree)：n (n≥0) 个结点构成的有限集合。当n=0时，称为空树
- 对于任一棵非空树（n> 0），它具备以下性质：
  - 树中有一个称为“根(Root)”的特殊结点，用 r 表示；
  - 其余结点可分为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棵树，称为原来树的“子树(SubTree)”

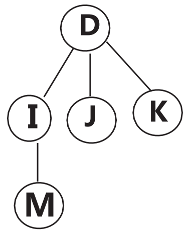


(a) 树 *T*　(b) 子树 T_{A1}　(c) 子树 T_{A2}　(d) 子树 T_{A3}　(e)子树 T_{A4}

树与非树

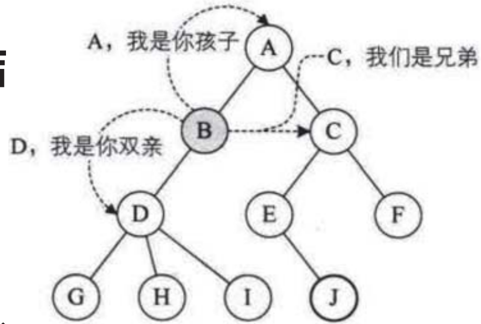


- 子树是**不相交**的
- 除了根结点外，**每个结点有且仅有一个父结点**
- 一棵N个结点的树有**N-1条边**

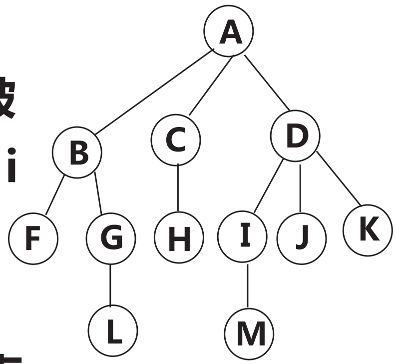


树的基本术语

- **1. 结点的度(Degree)**：一个结点的度是其子树的个数。
- **2. 树的度**：树的所有结点中最大的度数。
- **3. 叶结点(Leaf)**：是**度为0**的结点；叶结点也可称为端结点。
- **4. 父结点(Parent)**：有子树的结点是其子树的根结点的父结点。
- **5. 子结点(Child)**：若A结点是B结点的父结点，则称B结点是A结点的子结点；子结点也称**孩子结点**。
- **6. 兄弟结点(Sibling)**：具有同一父结点的各结点彼此是兄弟结点。

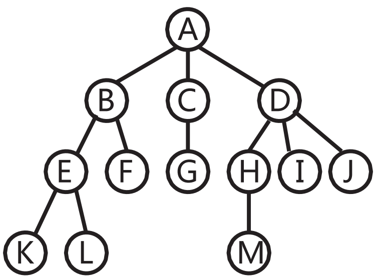


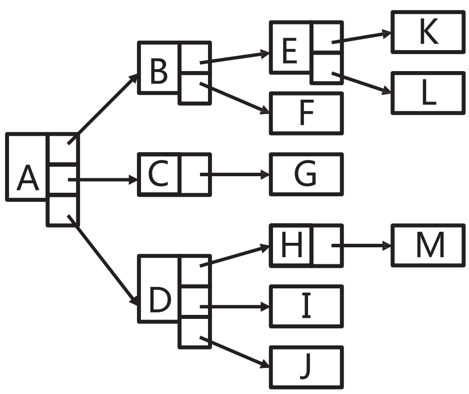
树的基本术语



- **7. 分支**：树中两个相邻结点的连边称为一个分支。
- **8. 路径和路径长度**：从结点n_1到n_k的路径被定义为一个结点序列n_1，n_2，…，n_k，对于$1 \le i \le k$，n_i是n_{i+1}的父结点。一条路径的长度为这条路径所包含的边(分支)的个数。
- **9. 祖先结点(Ancestor)**：沿树根到某一结点路径上的所有结点都是这个结点的祖先结点。
- **10. 子孙结点(Descendant)**：某一结点的子树中的所有结点。
- **11. 结点的层次(Level)**：规定根结点在1层，其它任一结点的层数是其父结点的层数加1。
- **12. 树的高度(Height)**：树中所有结点中的最大层次。

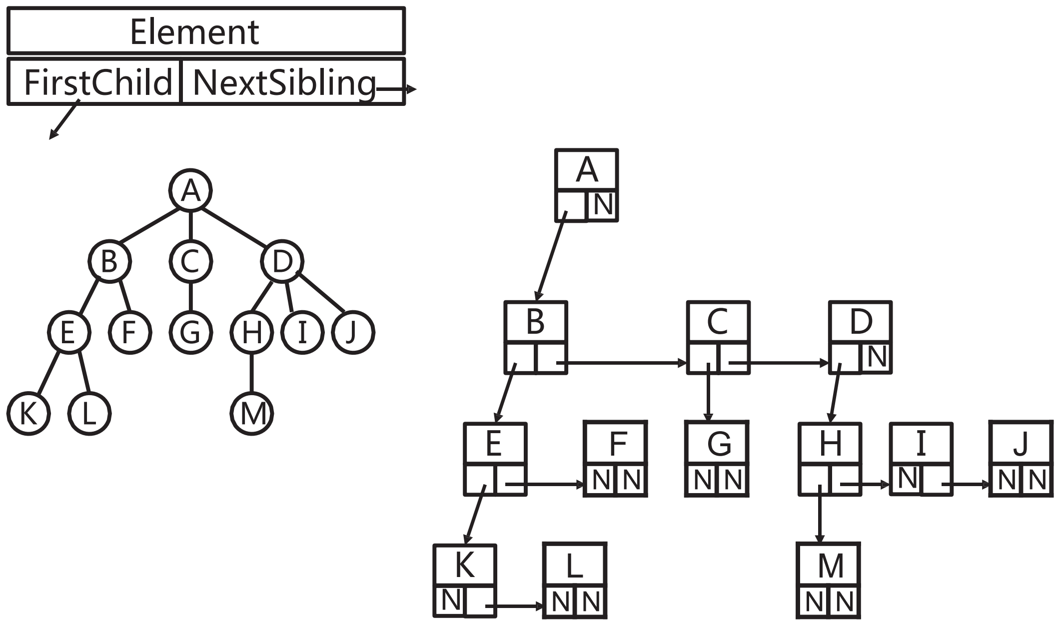
树的表示



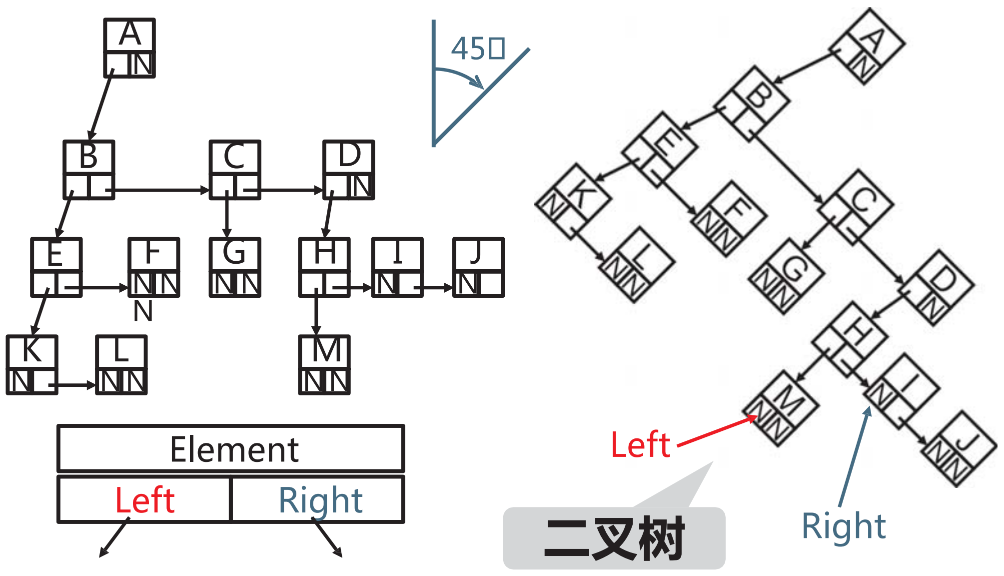


树的表示

- **儿子-兄弟表示法**



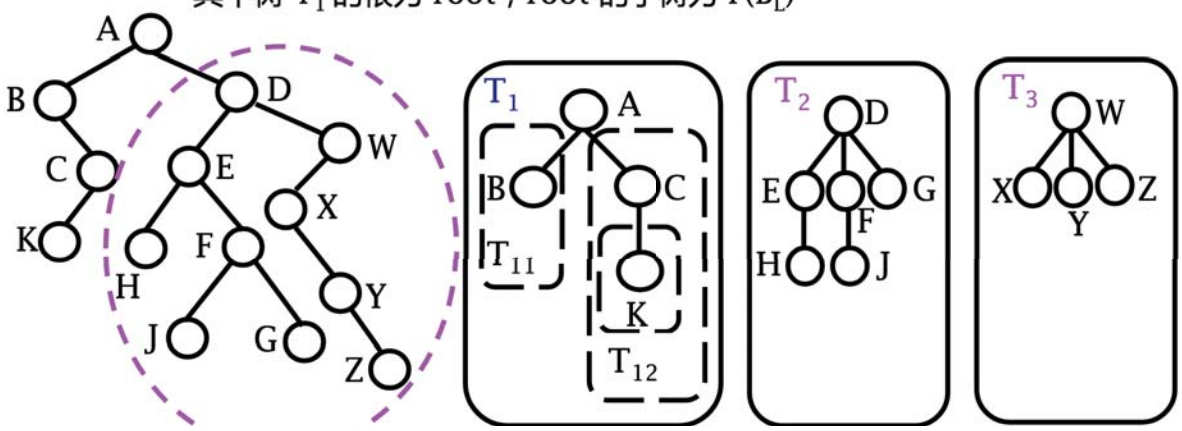
树的表示



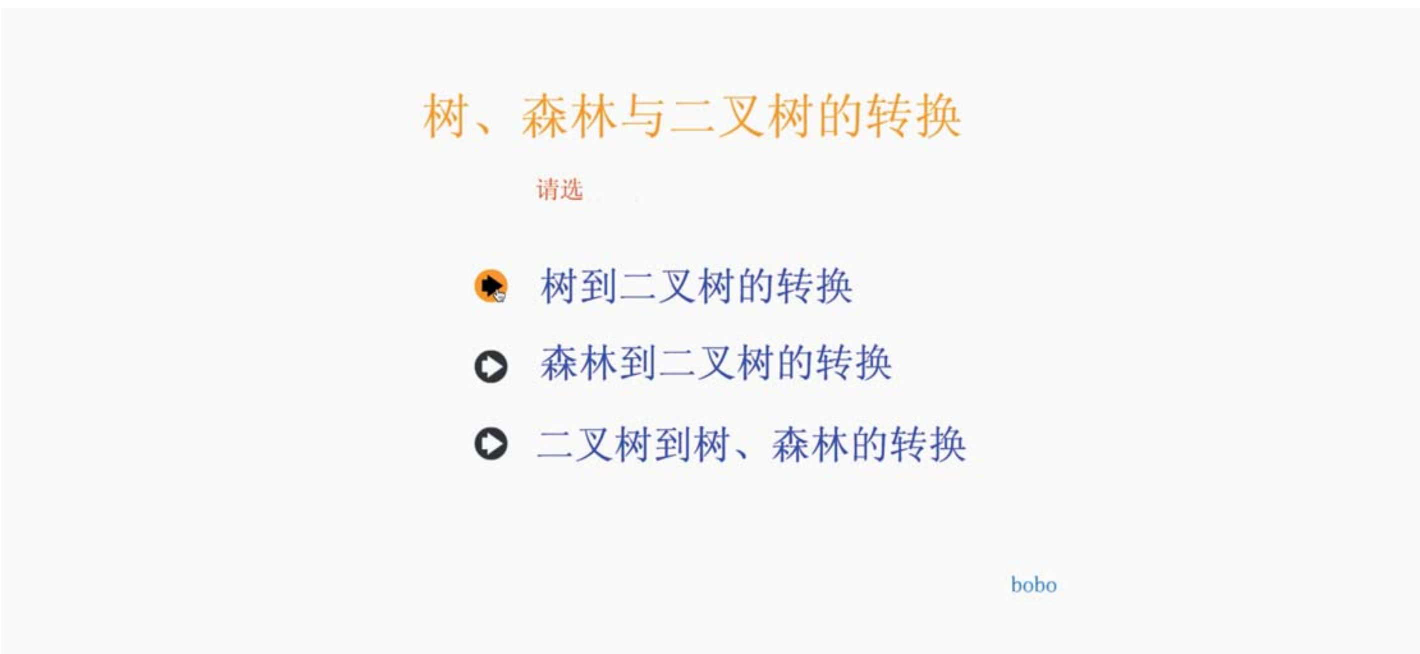
树的表示

二叉树转化成森林或树的形式定义

- 设B是一棵二叉树，root是 B 的根，B_L 是 root 的左子树，B_R 是 root 的右子树，则对应于二叉树B的森林或树 F(B) 的形式定义是：
  - 若 B 为空，则 F(B) 是空的森林
  - 若 B 不为空，则 F(B)是一棵树T_1加上森林 $F(B_R)$，其中树 T_1 的根为 root，root 的子树为 $F(B_L)$



树、森林和二叉树的转换



树的表示

- **遍历森林vs遍历二叉树**
  - 先根次序遍历森林
    - 前序法遍历二叉树
  - 后根次序遍历森林
    - 按中序法遍历对应的二叉树
  - 中根遍历？
    - 无法明确规定根在哪两个子结点之间

树的特点

线性结构	树结构
•第一个数据元素：无前驱	•根结点：无双亲，唯一
•最后一个数据元素：无后继	•叶结点：无孩子，可以多个
•中间元素：一个前驱一个后继	•中间结点：一个双亲多个孩子

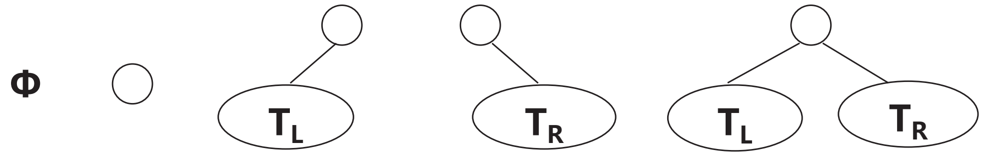
休息时间

第6节休息

3.2 二叉树及存储结构

二叉树的定义

- **二叉树T**：一个有穷的结点集合。这个集合**可以为空**
- 若不为空，则它是由**根结点**和称为其**左子树T_L**和**右子树T_R**的两个不相交的二叉树组成
  - 二叉树具体五种基本形态

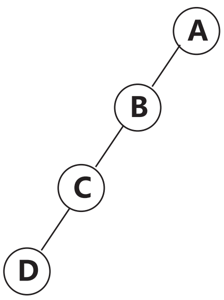


(a) (b) (c) (d) (e)

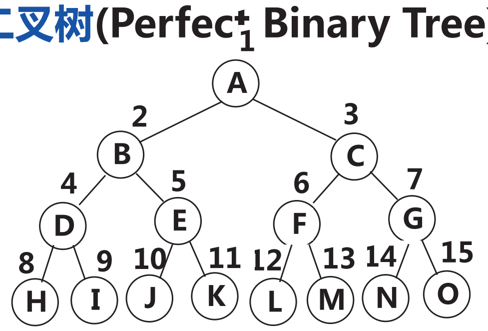
  - 二叉树的**子树有左右顺序之分**

特殊二叉树

- **斜二叉树**(Skewed Binary Tree)

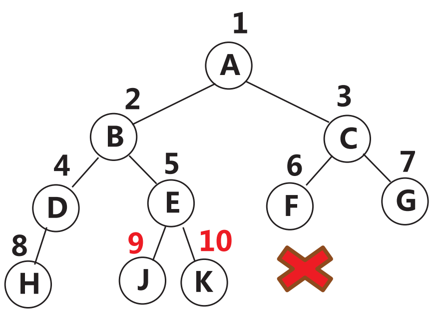


- **完美二叉树**(Perfect Binary Tree)
- **满二叉树**(Perfect Binary Tree)

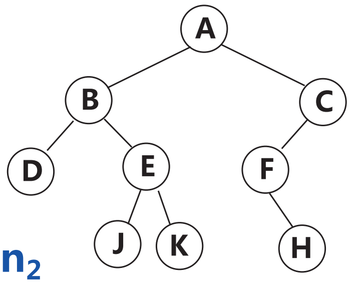


- **完全二叉树**(Complete Binary Tree)

有n个结点的二叉树，对树中结点按从上至下、从左到右顺序进行编号，编号为i（1 ≤ i ≤ n）结点与满二叉树 中编号为 i 结点在二叉树中位置相同



二叉树的重要性质



- 一个二叉树第 i 层的最大结点数为 :2^{i-1}，$i \geq 1$。
- 深度为k的二叉树有最大结点总数为 :$2^{k}-1$，$k \geq 1$。
- 对任何非空的二叉树 T，若n_0表示叶结点的个数、n_2是度为2的非叶结点个数，那么两者满足关系$n_0 = n_2 + 1$。

$n_0 = 4$，$n_1 = 2$
$n_2 = 3$
$n_0 = n_2 + 1$

- n个结点的完全二叉树的深度为k 为 ：$\lfloor \log_2 n \rfloor + 1$

证明: 设 n_1 是度为1结点数, n 是总的结点数. 那么

$$n = n_0 + n_1 + n_2 \quad ①$$

设 B 是全部分枝数. 则 $n \sim B$? $n = B + 1$. ②

因为所有分枝都来自度为1或2的结点, 所以 $B \sim n_1$ & n_2 ?

$$B = n_1 + 2n_2 \quad ③$$

$$\Rightarrow \quad n_0 = n_2 + 1$$

二叉树的抽象数据类型定义

- **类型名称**：二叉树
- **数据对象集**：一个有穷的结点集合。若不为空，则由根结点和其左、右二叉子树组成。
- **操作集**： BT BinTree, Item ElementType，重要操作有：
  - Boolean IsEmpty(BinTree BT)： 判别BT是否为空；
  - void Traversal(BinTree BT)：遍历，按某顺序访问每个结点；
  - BinTree CreatBinTree( )：创建一个二叉树。

- **常用的遍历方法有：**
  - void PreOrderTraversal(BinTree BT)：先序----根、左子树、右子树；
  - void InOrderTraversal(BinTree BT)：中序---左子树、根、右子树；
  - void PostOrderTraversal(BinTree BT)：后序---左子树、右子树、根
  - void LevelOrderTraversal(BinTree BT)：层次遍历，从上到下、从左到右

二叉树的存储结构

- **1. 顺序存储结构**
  - 完全二叉树最适合这种存储结构。
  - n个结点的完全二叉树的结点父子关系，简单地由序列号决定：

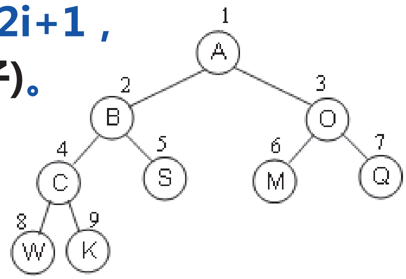
    1、非根结点（序号 i > 1）的父结点的序号是 $\lfloor i / 2 \rfloor$；

    2、结点(序号为 i)的左孩子结点的序号是 2i，
    (若2 i <= n，否则没有左孩子);

    3、结点(序号为 i)的右孩子结点的序号是 2i+1，
    (若2 i +1<= n，否则没有右孩子)。

数据	A	B	O	C	S	M	Q	W	K
编号	1	2	3	4	5	6	7	8	9

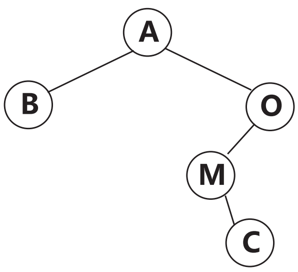
(a)相应的顺序存储结构



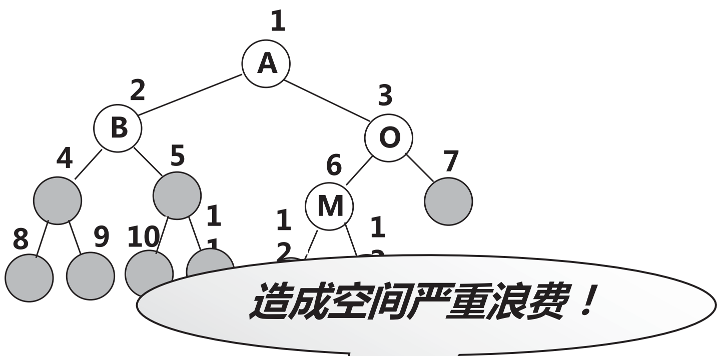
(b) 完全二叉树

二叉树的存储结构

- 一般二叉树采用这种结构将造成空间浪费



(a)一般二叉树



(b) 对应(a)的完[illegible]树

数据	A	B	O	∧	∧	M	∧	∧	∧	∧	∧	∧	C
编号	1	2	3	4	5	6	7	8	9	10	11	12	13

二叉树的存储结构

2. 二叉树的链表存储

二叉树的各结点随机地存储在内存空间中，结点之间的逻辑关系用指针来链接。

- 二叉链表
  - 指针 left 和 right，分别指向结点的左孩子和右孩子

left	info	right

- 三叉链表
  - 指针 left 和 right，分别指向结点的左孩子和右孩子
  - 增加一个父指针

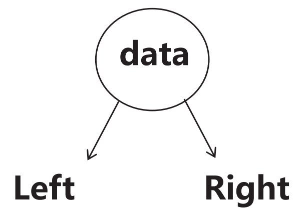
left	info	parent	right

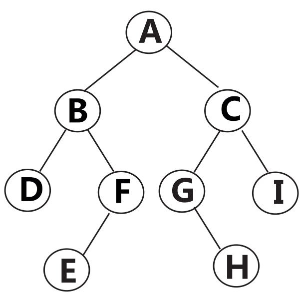
二叉树的存储结构

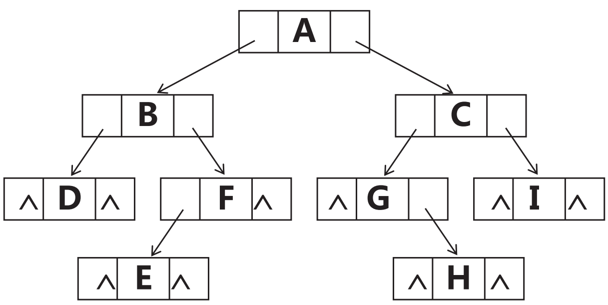
二叉链表

Left	Data	Right

```
typedef  struct  TreeNode *BinTree;
typedef  BinTree Position;
struct  TreeNode{
          ElementType  Data;
          BinTree  Left;
          BinTree  Right;
} ;
```



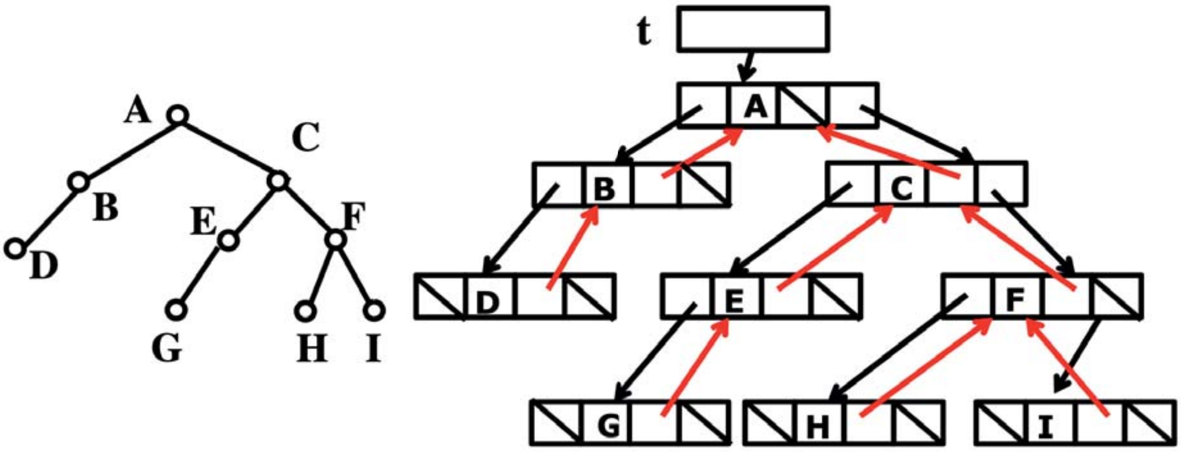




二叉树的存储结构

- **三叉链表**

  - 指向父母的指针parent，“向上”能力



二叉树的建立

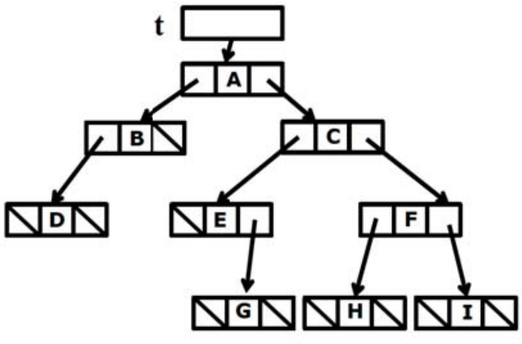
空间开销分析

- 存储密度 α (≤1) 表示数据结构存储的效率

$$\alpha\text{（存储密度）}=\frac{\text{数据本身存储量}}{\text{整个结构占用的存储总量}}$$

- 结构性开销 $\gamma = 1- \alpha$

根据满二叉树定理：一半的指针是空的



(b)

- 每个结点存两个指针、一个数据域
  - 总空间 $(2p + d)n$
  - 结构性开销：$2pn$
  - 如果 $p = d$, 则结构性开销 $2p/(2p + d) = 2/3$

思考

- 用三叉链的存储形式修改二叉树的相应算法。特别注意插入和删除结点，维护父指针信息。
- 完全三叉树的下标公式？

3.3 二叉树的遍历

二叉树的遍历

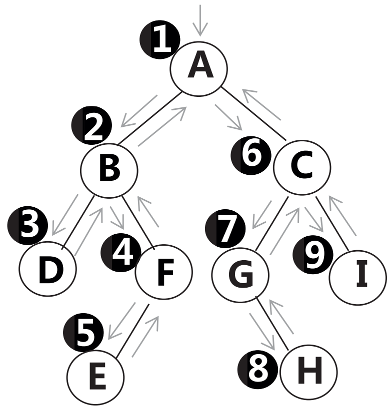
- **1. 先序遍历**

其遍历过程为：
①访问根结点；
②先序遍历其左子树；
③先序遍历其右子树。

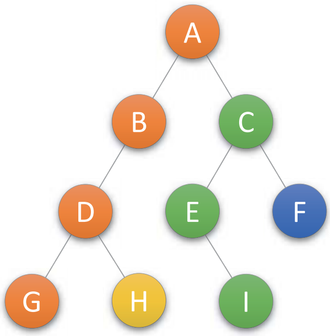
A (B D F E) (C G H I)

先序遍历=> A B D F E C G H I

```
void PreOrderTraversal( BinTree BT )
{
   if( BT ) {
       printf( "%d" , BT->Data);
       PreOrderTraversal( BT->Left );
       PreOrderTraversal( BT->Right );
   }
}
```



二叉树前序遍历



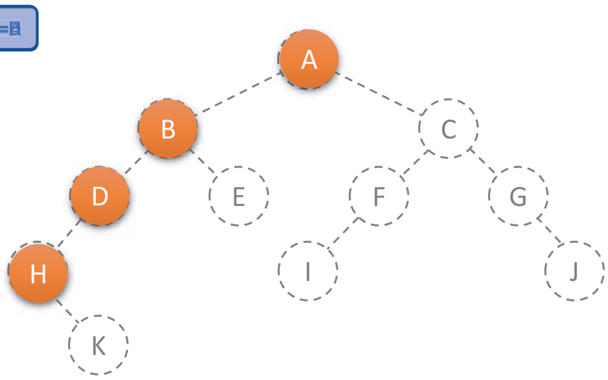
二叉树前序遍历

关键代码

```
void PreOrderTraverse(BiTree T)
{
        if(T==NULL)
                return;
        printf("%c",T->data);
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
}
```

T->data=

数据变化

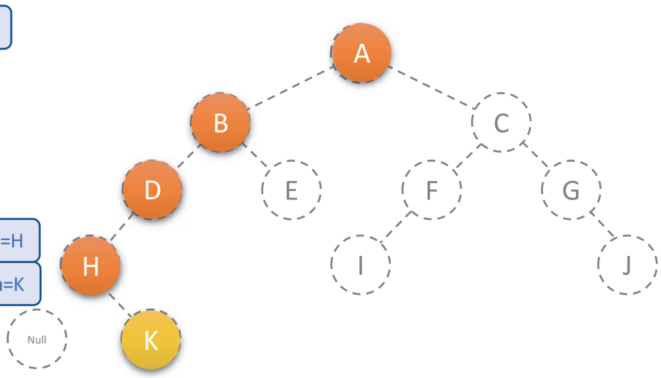


二叉树前序遍历

关键代码

```
void PreOrderTraverse(BiTree T)        T=NULL
{
        if(T==NULL)
                return;
        printf("%c",T->data);
        PreOrderTraverse(T->lchild);   T->data=H
        PreOrderTraverse(T->rchild);   T->data=K
}
```

数据变化



二叉树的遍历

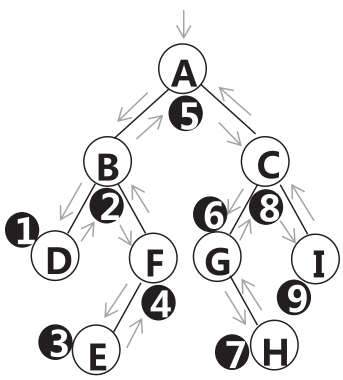
- **2. 中序遍历**

其遍历过程为：
①中序遍历其左子树；
②访问根结点；
③中序遍历其右子树。

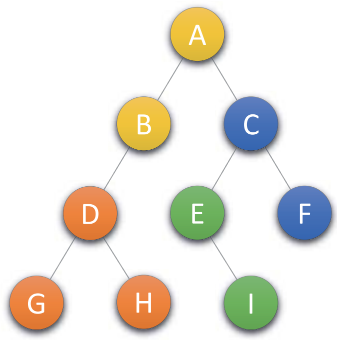
```
void InOrderTraversal( BinTree BT )
{
    if( BT ) {
        InOrderTraversal( BT->Left );
        printf( "%d" , BT->Data);
        InOrderTraversal( BT->Right );
    }
}
```

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

中序遍历=> D B E F A G H C I



二叉树中序遍历



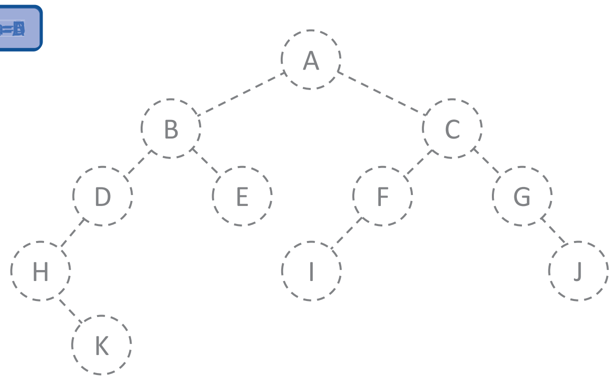
二叉树中序遍历

关键代码

```
void PreOrderTraverse(BiTree T)
{
        if(T==NULL)
                return;
        PreOrderTraverse(T->lchild);
        printf("%c",T->data);
        PreOrderTraverse(T->rchild);
}
```

T->data=[illegible]

数据变化



二叉树中序遍历

关键代码

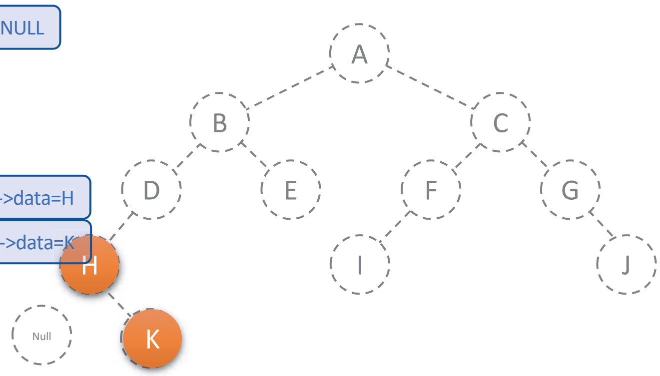
```
void PreOrderTraverse(BiTree T)
{
        if(T==NULL)
                return;
        PreOrderTraverse(T->lchild);
        printf("%c",T->data);
        PreOrderTraverse(T->rchild);
}
```

T=NULL

T->data=H

T->data=K

数据变化



二叉树中序遍历

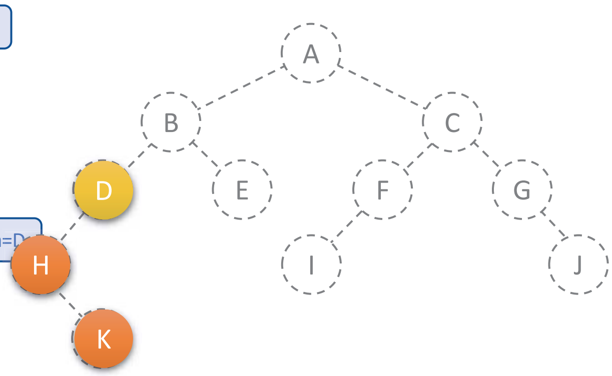
关键代码

```
void PreOrderTraverse(BiTree T)
{
        if(T==NULL)
                return;
        PreOrderTraverse(T->lchild);
        printf("%c",T->data);
        PreOrderTraverse(T->rchild);
}
```

T=NULL

T->data=D

数据变化



二叉树的遍历

- **3. 后序遍历**

其遍历过程为：

①后序遍历其左子树；

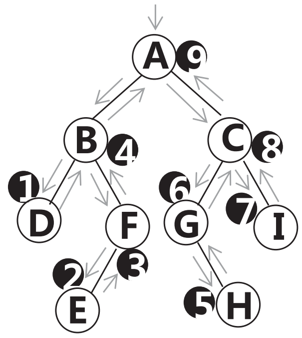
②后序遍历其右子树；

③访问根结点。

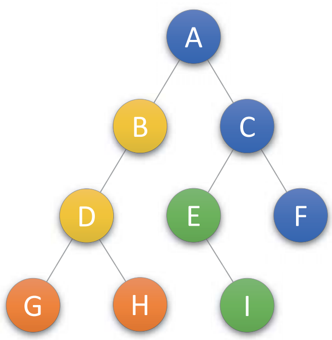
(D E F B) (H G I C) A

后序遍历 => D E F B H G I C A

```
void PostOrderTraversal( BinTree BT )
{
    if( BT ) {
        PostOrderTraversal( BT->Left );
        PostOrderTraversal( BT->Right);
        printf( "%d", BT->Data);
    }
}
```

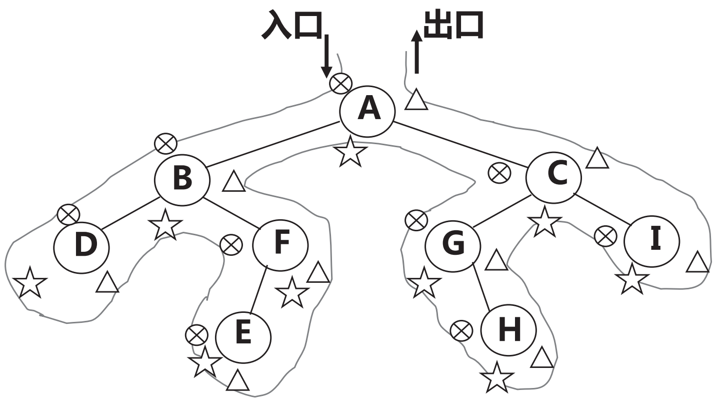


二叉树后续遍历



二叉树的非递归遍历

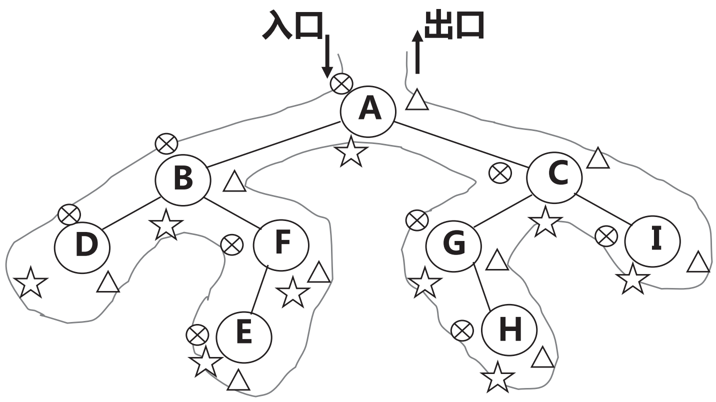
- 先序、中序和后序遍历过程：遍历过程中经过结点的**路线一样**，只是**访问各结点的时机不同**。
- **中序**遍历**非递归**遍历算法
- 非递归算法实现的基本思路：**使用堆栈**
- 图中在从从入口到出口的曲线上用⊗、☆□和△三种符号分别标记出了**先序**、**中序**和**后序**遍历各结点的时刻



二叉树的非递归遍历

- **中序**遍历**非递归**遍历算法

非递归算法实现的基本思路：**使用堆栈**



二叉树的非递归遍历

后序遍历非递归遍历算法比较复杂。作为思考题。

- 中序遍历非递归遍历算法
  - 遇到一个结点，就把它压栈，并去遍历它的左子树；
  - 当左子树遍历结束后，从栈顶弹出这个结点并访问它；
  - 然后按其右指针再去中序遍历该结点的右子树。

```
void InOrderTraversal( BinTree BT )
{   BinTree T BT;
        Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
        while( T || !IsEmpty(S) ){
            while(T){   /*一直向左并将沿途结点压入堆栈*/
                Push(S,T);
                T = T->Left;
            }
            if(!IsEmpty(S)){
                T = Pop(S); /*结点弹出堆栈*/
                printf( "%5d" , T->Data); /*（访问）打印结点*/
                T = T->Right; /*转向右子树*/
            }
        }
}
```

先序遍历非递归遍历算法只需把访问时机改在进栈时

二叉树的非递归遍历

- 先序遍历的非递归遍历算法？

```
void InOrderTraversal( BinTree BT )
{       BinTree T BT;
        Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
        while( T || !IsEmpty(S) ){
            while(T){       /*一直向左并将沿途结点压入堆栈*/
                Push(S,T);
                T = T->Left;
            }
            if(!IsEmpty(S)){
                T = Pop(S); /*结点弹出堆栈*/
                printf( "%5d" , T->Data); /*（访问）打印结点*/
                T = T->Right; /*转向右子树*/
            }
        }
}
```

二叉树的非递归遍历

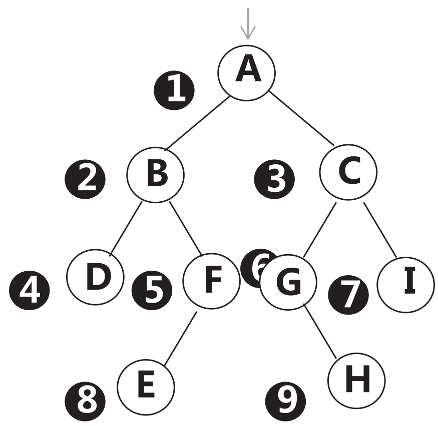
- **后序**遍历**非递归**遍历算法？
  - 遇到一个结点，就把它**(附带标志0以后)压栈**，并去遍历它的**左子树**；
  - 当**左子树遍历结束**后，检查栈顶元素的附带标志是否为0；
  - 若标志为0，则把**标志改成1**，并按其右指针再去遍历该结点的**右子树**；
  - 若标志为1，则从栈顶弹出这个结点并**访问它**。

层序遍历

- 二叉树遍历的核心问题：**二维结构的线性化**
  - 从结点访问其左、右儿子结点
  - 访问左儿子后，右儿子结点怎么办？
    - 需要一个存储结构保存暂时不访问的结点
    - 存储结构：堆栈、队列

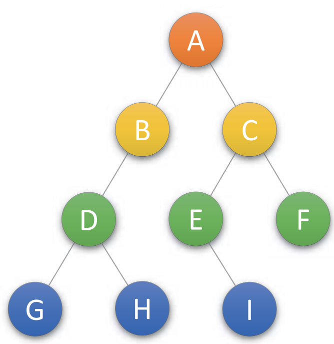
层序遍历

- **队列实现**：遍历从根结点开始，首先将**根结点入队**，然后开始执行循环：结点出队、访问该结点、其左右儿子入队



层序遍历 => A B C D F G I E H

二叉树层序遍历



层序遍历

- **层序基本过程**：先根结点入队，然后：

① 从队列中**取出一个**元素；
② **访问**该元素所指结点；
③ 若该元素所指结点的左、右孩子结点非空，则将其**左、右孩子的指针顺序入队**。

```
void LevelOrderTraversal ( BinTree BT )
{   Queue Q;  BinTree T;
      if ( !BT ) return; /* 若是空树则直接返回 */
      Q = CreatQueue( MaxSize ); /*创建并初始化队列Q*/
      AddQ( Q, BT );
      while ( !IsEmptyQ( Q ) ) {
         T = DeleteQ( Q );
           printf( “%d\n” , T->Data); /*访问取出队列的结点*/
           if ( T->Left )  AddQ( Q, T->Left );
           if ( T->Right )  AddQ( Q, T->Right );
      }
}
```

层序遍历应用

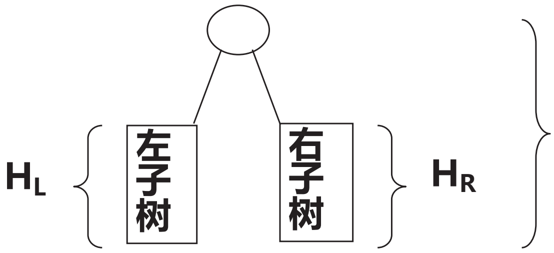
[例] 输出二叉树中的叶子结点。

- 在二叉树的遍历算法中增加检测结点的“**左右子树是否都为空**”。

```
void PreOrderPrintLeaves( BinTree BT )
{
   if( BT ) {
      if ( !BT-Left && !BT->Right )
         printf( “%d” , BT->Data );
      PreOrderPrintLeaves ( BT->Left );
      PreOrderPrintLeaves ( BT->Right );
   }
}
```

层序遍历应用

[例] 求二叉树的高度。

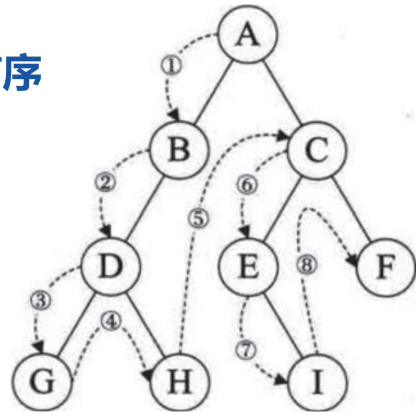


Height=max(H_L, H_R)+1

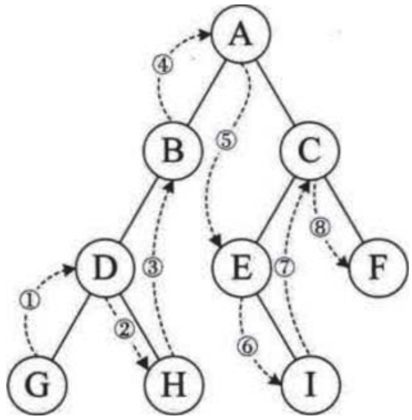
```
int PostOrderGetHeight( BinTree BT )
{   int HL, HR, MaxH;
    if( BT ) {
        HL = PostOrderGetHeight(BT->Left);  /*求左子树的深度*/
        HR = PostOrderGetHeight(BT->Right); /*求右子树的深度*/
        MaxH = HL > HR? HL : HR; /*取左右子树较大的深度*/
        return ( MaxH + 1 ); /*返回树的深度*/
    }
    else  return 0; /* 空树深度为0 */
}
```

小结

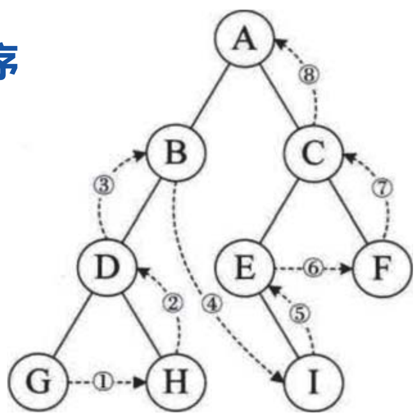
前序



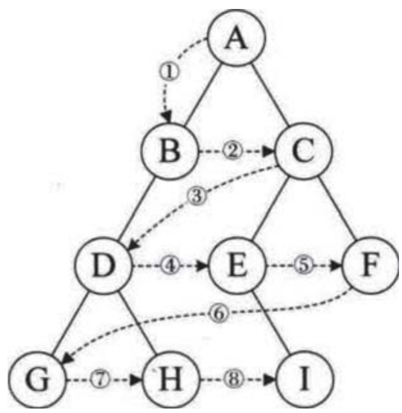
中序



后序



层序



寻找父结点

递归框架

```
template<class T>
BinaryTreeNode<T>* BinaryTree<T>::
Parent(BinaryTreeNode<T> *rt, BinaryTreeNode<T> *current) {
    BinaryTreeNode<T> *tmp,
    if (rt == NULL) return(NULL);
    if (current == rt ->leftchild() || current == rt->rightchild())
        return rt; // 如果孩子是current则返回parent
    if ((tmp =Parent(rt- >leftchild(), current) != NULL)
        return tmp;
    if ((tmp =Parent(rt- > rightchild(), current) != NULL)
        return tmp;
    return NULL;
}
```

寻找父结点

非递归框架

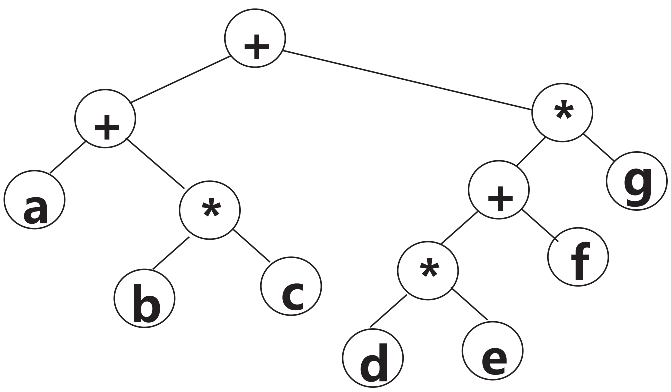
```
BinaryTreeNode<T>* BinaryTree<T>::Parent(BinaryTreeNode<T> *current) {
   using std::stack;                             // 使用STL中的栈
   stack<BinaryTreeNode<T>* > aStack;
   BinaryTreeNode<T> *pointer = root;
   aStack.push(NULL);                            // 栈底监视哨
   while (pointer) {                             // 或者!aStack.empty()
      if (current == pointer->leftchild() || current == pointer->rightchild())
         return pointer;                // 如果pointer的孩子是current则返回parent
      if (pointer->rightchild() != NULL)         // 非空右孩子入栈
         aStack.push(pointer->rightchild());
      if (pointer->leftchild() != NULL)
         pointer = pointer->leftchild();         // 左路下降
      else {                                     // 左子树访问完毕，转向访问右子树
         pointer=aStack.top(); aStack.pop();     // 获得栈顶元素，并退栈
}  } }
```

休息时间

第7节休息

遍历二叉树的应用

[例] 二元运算表达式树及其遍历。



中缀表达式会受到运算符优先级的影响

- 三种遍历可以得到三种不同的访问结果：
  - 中序遍历得到中缀表达式：*a + b * c + d * e + f * g*
  - 先序遍历得到前缀表达式：*+ + a * b c * + * d e f g*
  - 后序遍历得到后缀表达式：*a b c * + d e * f + g * +*

遍历二叉树的应用

[例] 由两种遍历序列确定二叉树。

已知三种遍历中的任意两种遍历序列，
能否唯一确定一棵二叉树呢？

答案是：
必须要有中序遍历才行！

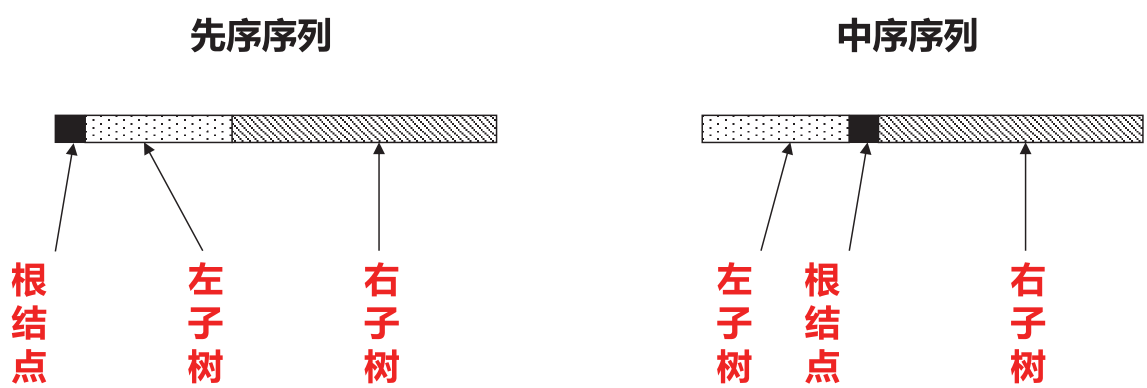
- 没有中序的困扰：
  - 先序遍历序列：A B
  - 后序遍历序列：B A

遍历二叉树的应用

- **先序和中序遍历序列来确定一棵二叉树**

[分析]

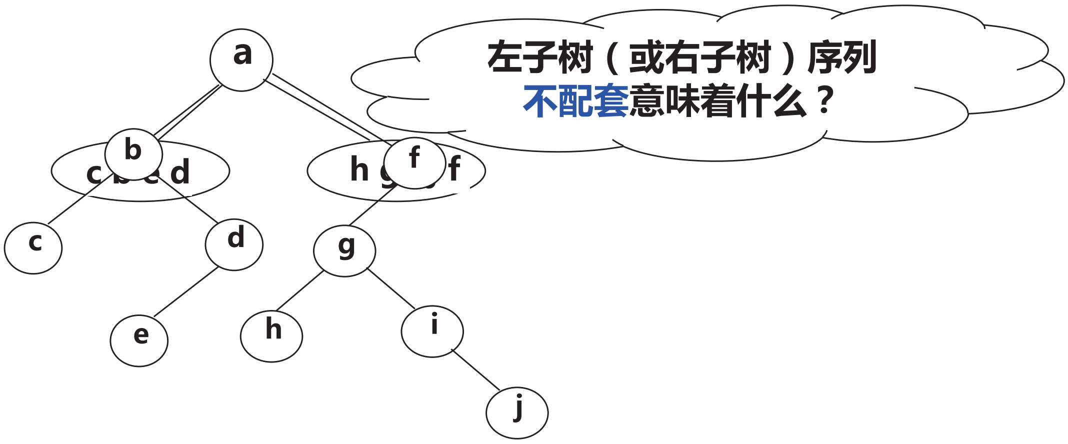
- 根据先序遍历序列第一个结点确定根结点；
- 根据根结点在中序遍历序列中分割出左右两个子序列
- 对左子树和右子树分别递归使用相同的方法继续分解



遍历二叉树的应用

[例] 先序序列： a b c d e f g h i j

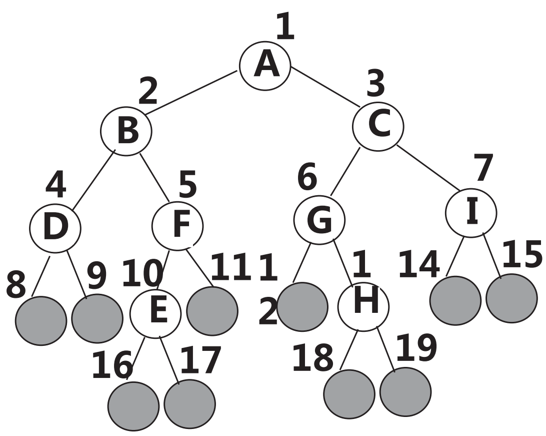
中序序列： c b e d a h g i j f



- 类似地，配套的**后序**和**中序**遍历序列也可以确定一棵二叉树。

二叉树的创建

- 常用的方法是**先序创建**和**层序创建**两种。



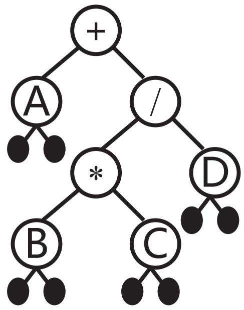
先序创建的输入序列： A, B, D, 0, 0, F, E, 0, 0, 0, C, G, 0, H, 0, 0, I, 0, 0

层序创建的输入序列： A, B, C, D, F, G, I, 0, 0, E, 0, 0, H, 0, 0, 0, 0, 0, 0

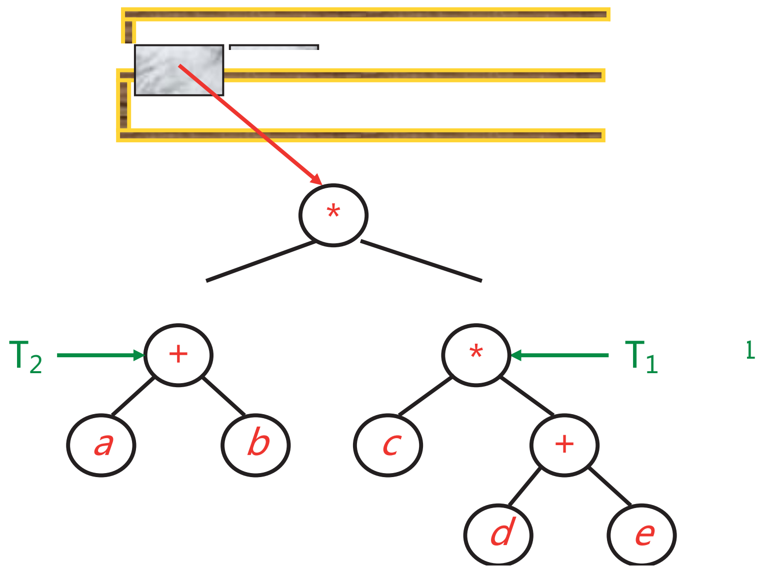
表达式树的构造

- 现在从对应的后缀表达式构建语义树

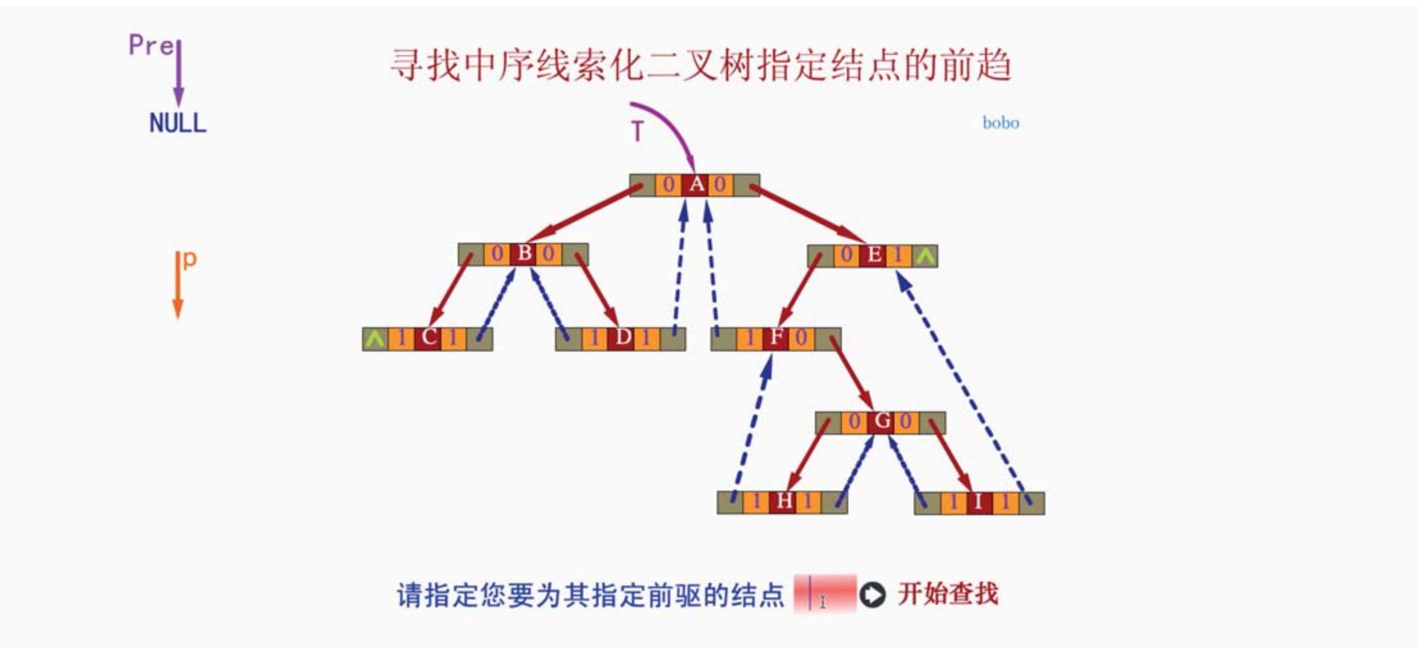
[例] 给定中缀表达式: A + B ∗ C / D



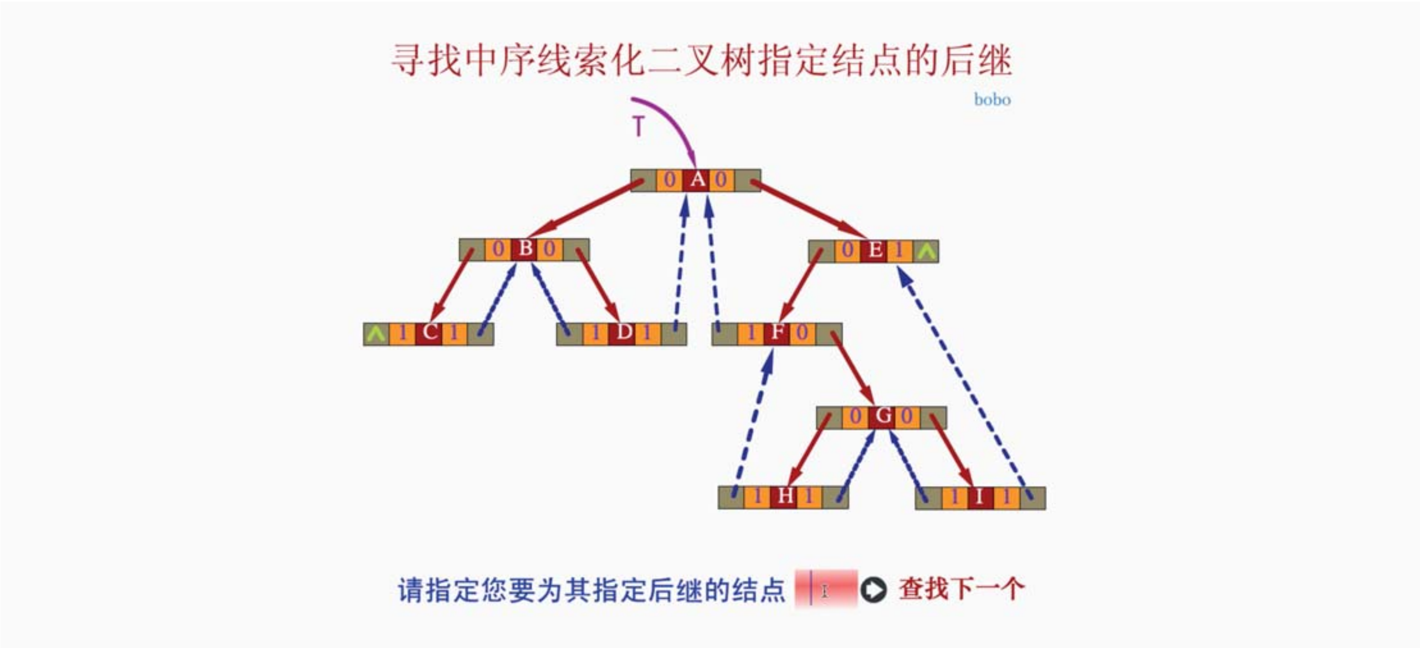
[Example] (*a* + *b*) * (*c* * (*d* + *e*)) = *a b* + *c d e* + * *



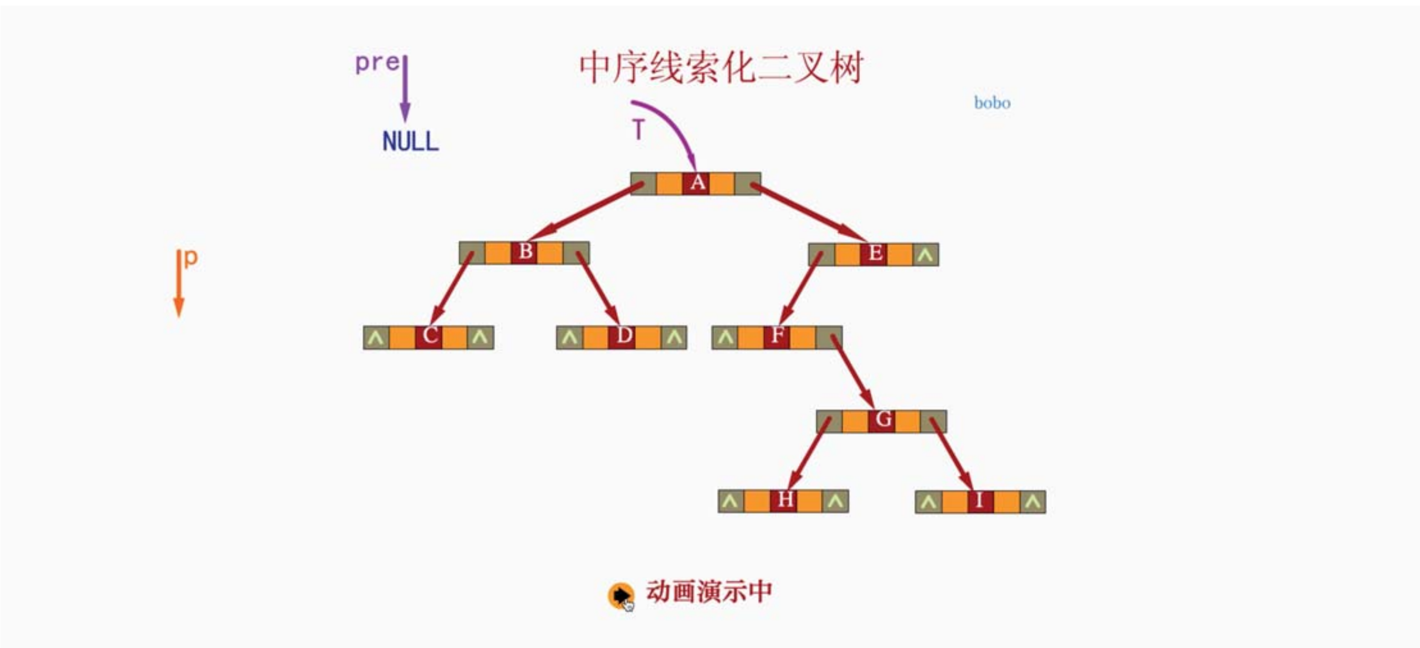
寻找中序线索化二叉树指定结点的前驱



寻找中序线索化二叉树指定结点的后继



中序线索化二叉树

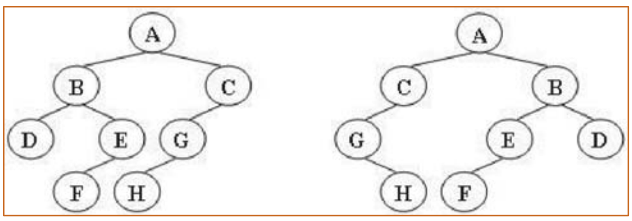


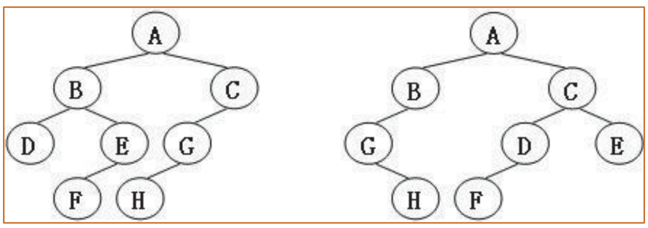
3.4 应用实例

应用1：树的同构判定

- **给定两棵树T1和T2。如果T1可以通过若干次左右孩子互换就变成T2，则我们称两棵树是“同构”的。**

现给定两棵树，请你判断它们是否是同构的。



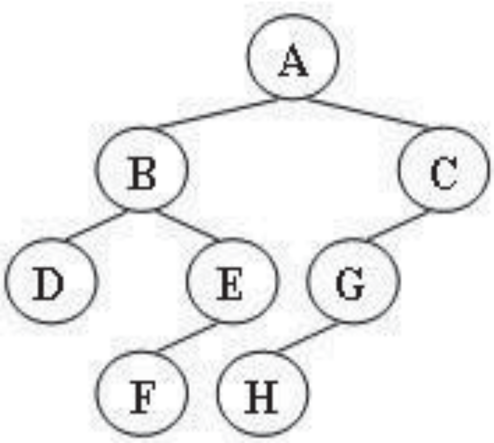


题意理解

- **输入格式: 输入给出2棵二叉树的信息：**
  - 先在一行中给出该树的结点数，随后N行
  - 第i行对应编号第i个结点，给出该结点中存储的字母、其左孩子结点的编号、右孩子结点的编号
  - 如果孩子结点为空，则在相应位置上给出“-”

输入样例: 8

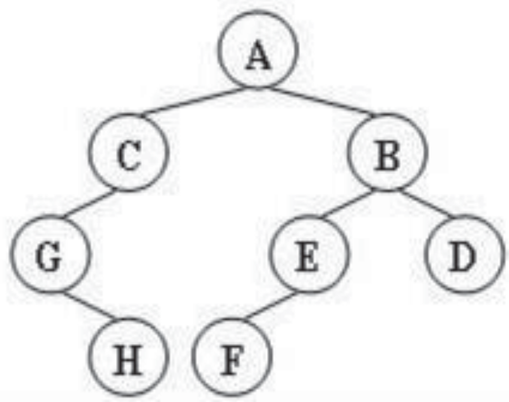
0 A 1 2
1 B 3 4
2 C 5 -
3 D - -
4 E 6 -
5 G 7 -
6 F - -
7 H - -
8
G - 4
B 7 6
F - -
A 5 1
H - -
C 0 -
D - -
E 2 -



题意理解

- **输入格式: 输入给出2棵二叉树的信息：**
  - 先在一行中给出该树的结点数，随后N行
  - 第i行对应编号第i个结点，给出该结点中存储的字母、其左孩子结点的编号、右孩子结点的编号
  - 如果孩子结点为空，则在相应位置上给出“-”

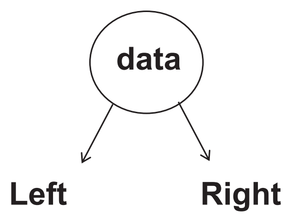
输入样例
: 8
A 1 2
B 3 4
C 5 -
D - -
E 6 -
G 7 -
F - -
H - -
8
0 G - 4
1 B 7 6
2 F - -
3 A 5 1
4 H - -
5 C 0 -
6 D - -
7 E 2 -

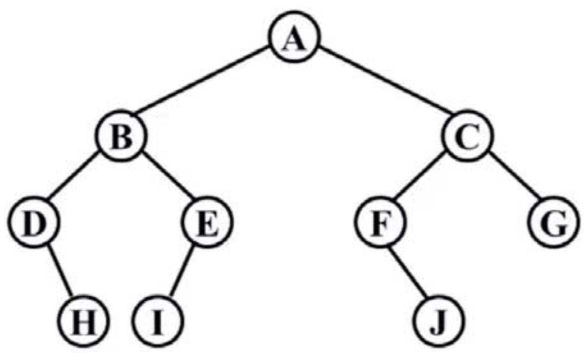


求解思路

- 1. 二叉树表示
- 2. 建二叉树
- 3. 同构判别

二叉树表示





BT

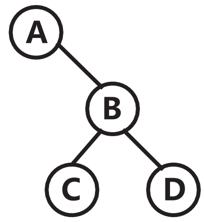
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F	G	--	H	I	--	--	J

二叉树表示

- **结构数组表示二叉树：静态链表**

```
#define MaxTree 10
#define ElementType
char  #define Tree int
#define Null -1

struct TreeNode
{
            ElementType Element;
            Tree  Left;
            Tree  Right;
} T1[MaxTree], T2[MaxTree];
```



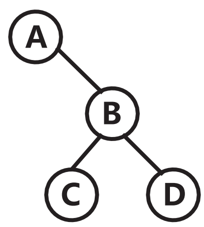
	0	1	2	3	4
	A	B	C	D	
Left	-1	2	-1	-1	
Right	1	3	-1	-1	

二叉树表示

- **结构数组表示二叉树：静态链表**

```
#define MaxTree 10
#define ElementType
char  #define Tree int
#define Null -1

struct TreeNode
{
            ElementType Element;
            Tree  Left;
            Tree  Right;
} T1[MaxTree], T2[MaxTree];
```

	0	1	2	3	4
	B	A		D	C
Left	4	-1		-1	-1
Right	3	0		-1	-1

程序框架搭建

```
int main()
{
     建二叉树1
     建二叉树2
     判别是否同构并输出
     return 0;
}
```

```
int main()
{
    Tree R1, R2;
    R1 = BuildTree(T1);
    R2 = BuildTree(T2);
    if (Isomorphic(R1, R2))
           printf("Yes\n");  else
           printf("No\n");

    return 0;
}
```

需要设计的函数：

- 读数据建二叉树
- 二叉树同构判别

如何建二叉树

```
Tree BuildTree( struct TreeNode T[] )
{         .....
          scanf("%d\n", &N);  if (N)
          {
                    ........
                    for (i=0; i<N; i++) {
                         scanf("%c %c %c\n", &T[i].Element, &cl, &cr);
                         ......
                    }
                    ......

                    Root = ???
          }
          return Root;
}
```

T[i]中没有任何结点的 left(cl)和right(cr)指向它。 只有一个

	8
0	A 1 2
1	B 3 4
2	C 5
3	- D
4	- -
5	E 6 -
6	G 7 -
7	F - -
	H - -

如何建二叉树

```
Tree BuildTree( struct TreeNode T[] )
{       .....
        scanf("%d\n", &N);
        if (N) {
                for (i=0; i<N; i++) check[i] = 0;
                for (i=0; i<N; i++) {
                        scanf("%c %c %c\n", &T[i].Element, &cl, &cr);
                        if (cl != '-') {
                                T[i].Left = cl-'0';
                                check[T[i].Left] = 1;
                        }
                        else T[i].Left = Null;
                        ........
                        /*对cr的对应处理 */
                }
                for (i=0; i<N; i++)
                        if (!check[i]) break;
                Root = i;
        }
        return Root;
}
```

如何判断两二叉树同构

```
int Isomorphic ( Tree R1, Tree R2 )
{
    if ( (R1==Null )&& (R2==Null) )      /* both empty */
        return 1;
    if ( ((R1==Null)&&(R2!=Null)) || ((R1!=Null)&&(R2==Null)) )
        return 0; /* one of them is empty */
    if ( T1[R1].Element != T2[R2].Element )
        return    0; /* roots are different */
    if ( ( T1[R1].Left == Null )&&( T2[R2].Left == Null ) )
        /* both have no left subtree */
        return    Isomorphic( T1[R1].Right, T2[R2].Right );
    ......
}
```

如何判断两二叉树同构

```
int Isomorphic ( Tree R1, Tree R2 )
{       ......
        if ( ((T1[R1].Left!=Null)&&(T2[R2].Left!=Null))&&
            ((T1[T1[R1].Left].Element)==(T2[T2[R2].Left].Element)) )
          /* no need to swap the left and the right */
          return( Isomorphic( T1[R1].Left, T2[R2].Left ) &&
                  Isomorphic( T1[R1].Right, T2[R2].Right ) );

        else
         /* need to swap the left and the right  */
            return ( Isomorphic( T1[R1].Left, T2[R2].Right) &&
                    Isomorphic( T1[R1].Right, T2[R2].Left ) );

}
```

应用2：逆转链表

Given a constant *K* and a singly linked list *L*, you are supposed to reverse the links of every *K* elements on *L*. For example, given *L* being 1→2→3→4→5→6, if *K*=3, then you must output 3→2→1→6→5→4; if *K*=4, you must output 4→3→2→1→5→6.

Sample Input:

```
00100 6 4
00000 4 99999
00100 1 12309
68237 6 -1
33218 3 00000
99999 5 68237
12309 2 33218
```

Sample Output:

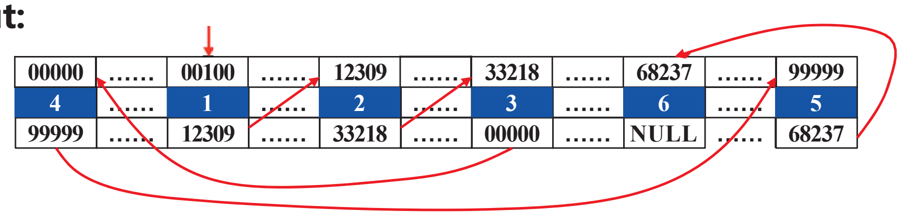
```
00000 4 33218
33218 3 12309
12309 2 00100
00100 1 99999
99999 5 68237
68237 6 -1
```

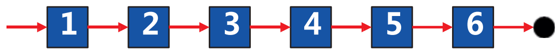
什么是抽象的链表

- 有块地方存数据
- 有块地方存指针 —— 下一个结点的地址

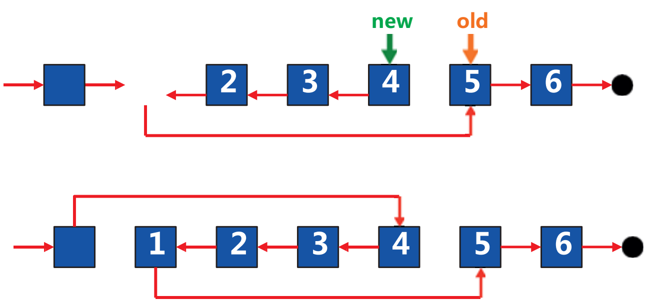
Sample Input:

```
00100 6 4
00000 4 99999
00100 1 12309
68237 6 -1
33218 3 00000
99999 5 68237
12309 2 33218
```





单链表的逆转




```
Ptr Reverse( Ptr head, int K )
{   cnt = 1;
    new = head->next;
    old = new->next;
    while ( cnt < K ) {
        tmp = old->next;
        old->next = new;
        new = old; old = tmp;
        cnt++;
    }
    head->next->next = old;
    return new;
}
```

取巧：用顺序表存储，先排序，再直接逆序输出。

在“内存”里多加几个没用的结点，让你偷懒！

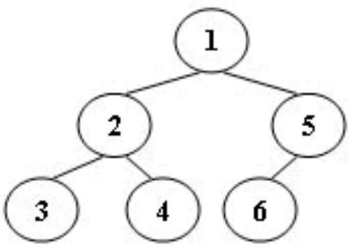
测试数据

- 有尾巴不反转
- 地址取到上下界
- 正好全反转
- K=N全反转
- K=1不用反转
- 最大（最后剩K-1不反转）、最小N
- 有多余结点

边界测试

应用3：非递归中序遍历

- Push的顺序为先序遍历
- Pop的顺序给出中序遍历

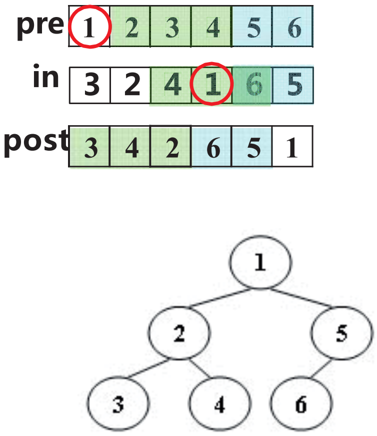


Sample Input:

```
6
Push 1
Push 2
Push 3
Pop
Pop
Push 4
Pop
Pop
Push 5
Push 6
Pop
Pop
```

pre	1	2	3	4	5	6
in	3	2	4	1	6	5

核心算法




```
void solve( int preL, int inL, int postL, int n )
{  if (n==0) return;
   if (n==1) {post[postL] = pre[preL];
              return;}
   root = pre[preL];
   post[postL+n-1] = root;
   for (i=0; i<n; i++)
      if (in[inL+i] == root)
         break;
   L = i;
   R = n-L-1;
   solve(preL+1, inL, postL, L);
   solve(preL+L+1, inL+L+1, postL+L, R);
}
```

休息时间

第8节休息