中国科学技术大学

University of Science and Technology of China

数据结构

树与二叉树（10 学时）

August 11, 2022

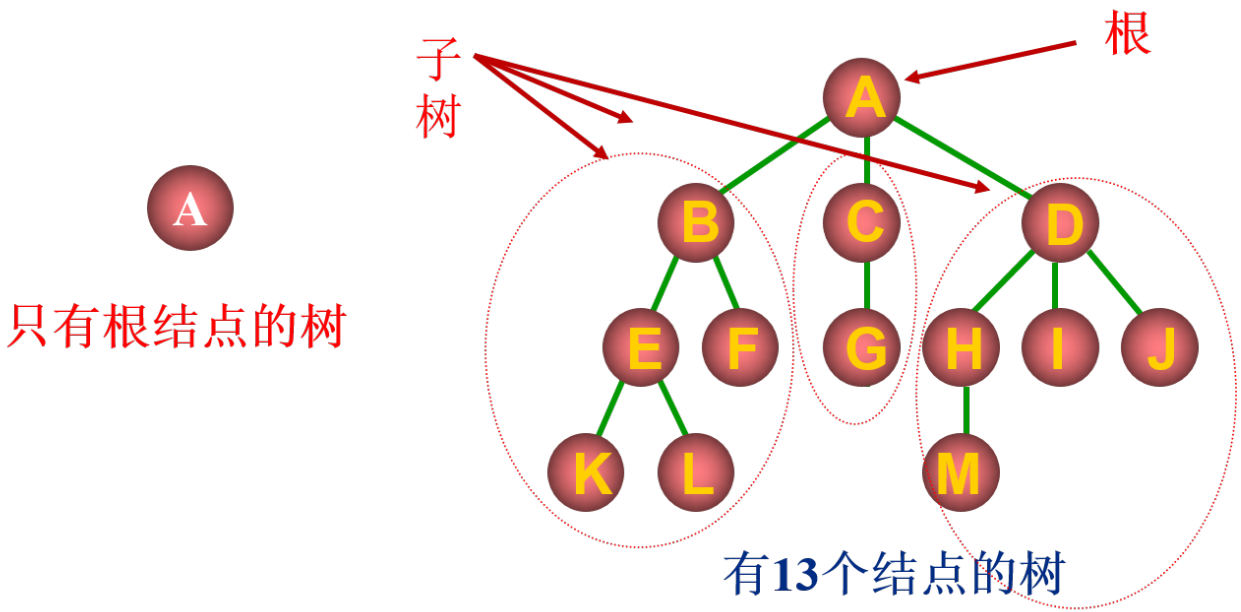
目录

树的定义

树是 $n(\geq 0)$ 个结点的有限集。

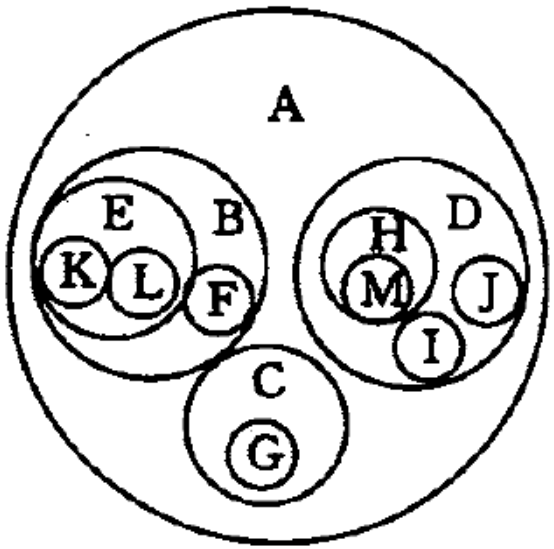
- 如果 n $=$ 0，称为空树；
- 如果 n $>$ 0，则
  - 有且仅有一个特定的称之为根 (Root) 的结点，它只有直接后继，但没有直接前驱；
  - 当 n $>$ 1，除根以外的其它结点划分为 m (m $>$0) 个互不相交的有限集 T1, T2 ,$\cdots$, Tm，其中每个集合本身又是一棵树，并且称为根的子树 (SubTree)。

图解



树的表示方法

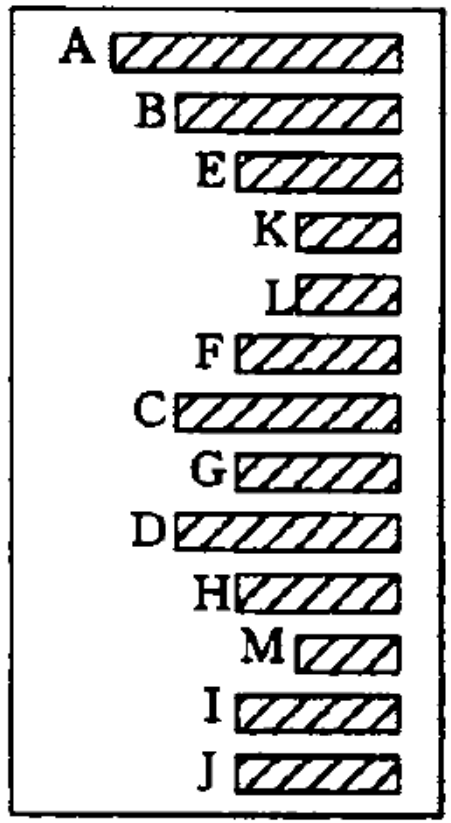
嵌套集合、广义表和凹入表



(a)

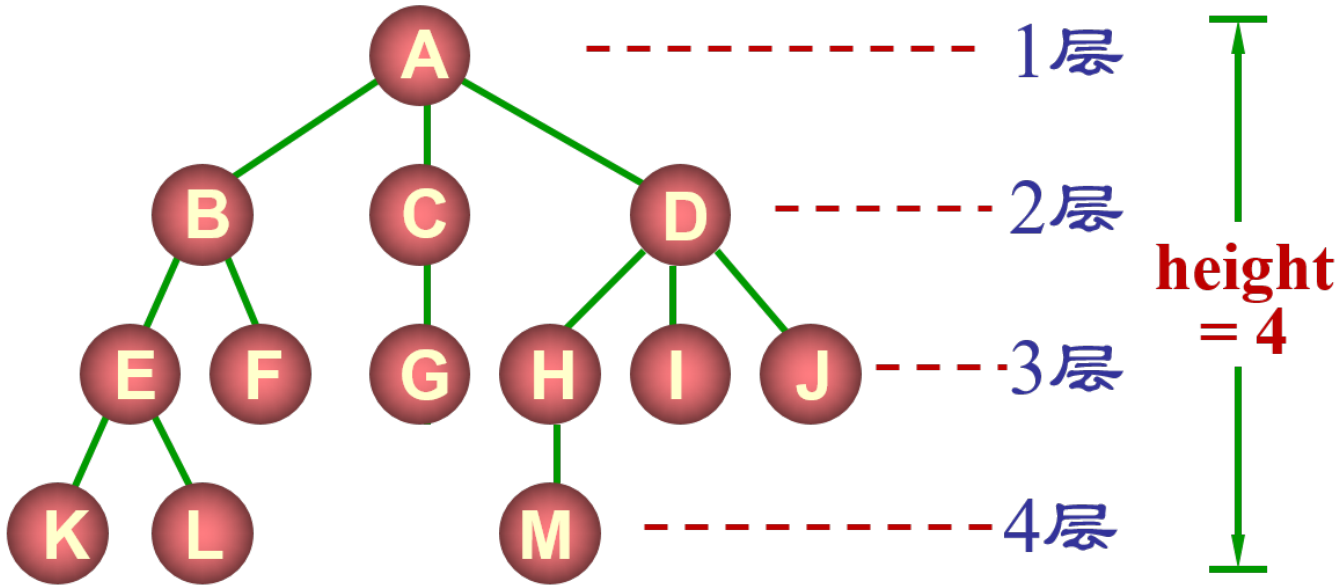
(A(B(E(K,L),F),C(G),D(H(M),I,J)))

(b)



(c)

树的基本术语



- 结点 (node)
- 结点的度 (degree)
- 叶结点 (leaf)
- 分支结点
- 树的度

- 结点层次 (level)
- 树的深度 (高度)
- 有序树/无序树
- 森林 (forest)

- 孩子 (child)
- 双亲 (parent)
- 兄弟 (sibling)
- 祖先 (ancestor)
- 子孙 (descendant)

树的 ADT

```
1  ADT Tree{ //阅读教材6.1节，了解数据对象和数据关系
2     …
3     基本操作:
4        InitTree(&T);
5        …
6        Value(T,cur_e);
7        Assign(T,cur_e,value);
8        Parent(T,cur_e);
9        LeftChild(T,cur_e);
10       …
11       InsertChild(&T,&p,i,c);//插入c为T中p所指结点的第i棵子树
12       DeleteChild(&T,&p,i); //删除T中p所指结点的第i棵子树
13       TraverseTree(T,Visit()); //树的遍历
14 }ADT Tree
```

认识二叉树/Binary Tree

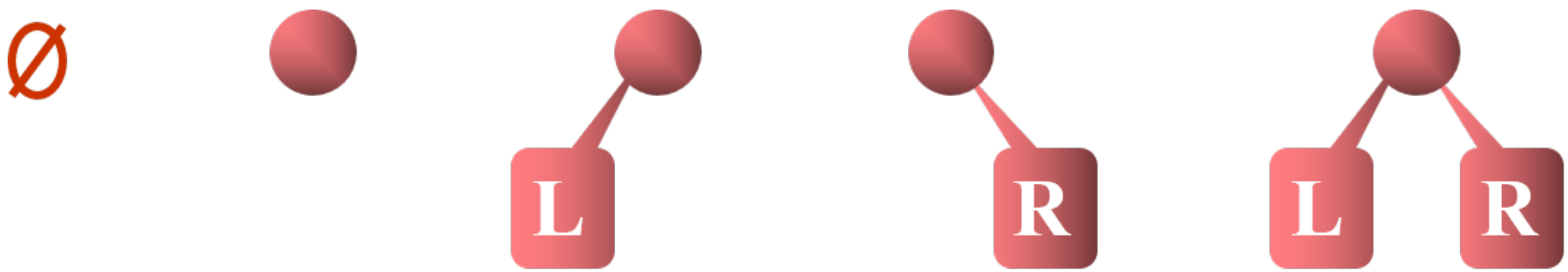
定义

- 一棵二叉树是结点的一个有限集，该集合或为空，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。

特点

- 每个结点至多只有两棵子树（二叉树中不存在度大于 2 的结点）

五种基本形态



二叉树的 ADT

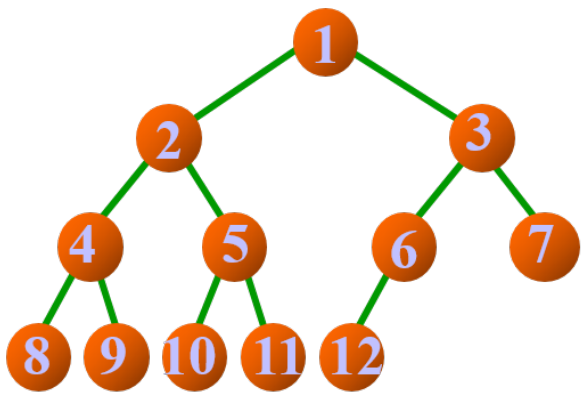
```
ADT BinaryTree{
    …
    基本操作:
        InitBiTree(&T);
        …
        Value(T,e);
        Assign(T,&e,value);
        Parent(T,e);
        LeftChild(T,e);
        …
        InsertChild(T,p,LR,c);
        DeleteChild(T,p,LR);
        PreOrderTraverse(T,Visit()); //先序遍历
        InorderTraverse(T,Visit()); //中序遍历
        PostOrderTraverse(T,Visit()) //后序遍历
        LevelOrderTraverse(T,Visit()) //层序遍历
}ADT BinaryTree
```

特殊的二叉树

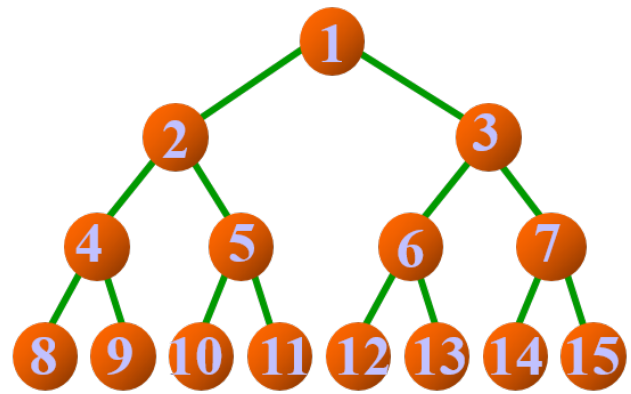
完全二叉树和满二叉树

- 若设二叉树的高度为 h，则共有 h 层。除第 h 层外，其它各层 $(1 \sim h-1)$ 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是完全二叉树。
- 一棵深度为 k 且每层节点数都达到最大个数的二叉树称为满二叉树。

图例



完全二叉树

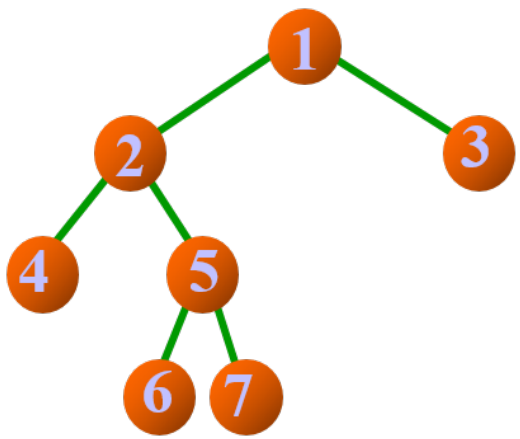


满二叉树

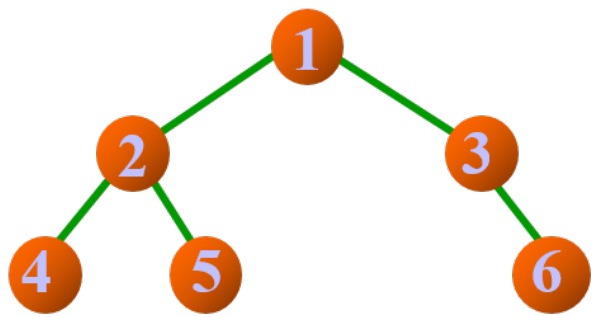
特殊二叉树

完全二叉树和满二叉树的特点

- 若对结点按从上到下，自左至右连续编号，则完全二叉树每个结点和相同高度满二叉树的编号结点一一对应。
- 叶结点只可能在层次最大的两层上出现。
- 任一结点，若其右分支下的子孙的最大层次为 l，则其左分支下的子孙的最大层次必为 l 或 $l+1$。
- 如图所示的非完全二叉树



非完全二叉树



二叉树的性质（1）

性质 1：二叉树的某一层的节点数量

- 结论：在二叉树的第 i 层上至多有 2^{i-1} 个结点。$(i \geq 1)$
- 用数学归纳法证明：
  - 当 $i = 1$ 时，只有根结点，$2^{i-1} = 2^0 = 1$，结论成立。
  - 假设对所有 $j, i > j \geq 1$，命题成立，即第 j 层上至多有 2^{j-1} 个结点。
  - 由归纳假设第 $i-1$ 层上至多有 $2^{i\,2}$ 个结点。由于二叉树的每个结点的度至多为 2，故在第 i 层上的最大结点数为第 $i-1$ 层上的最大结点数的 2 倍，即 $2 * 2^{i-2} = 2^{i-1}$

性质 2：二叉树的所有节点数量

- 结论：深度为 k 的二叉树至多有 $2^k - 1$ 个结点 $(k \geq 1)$
- 证明：
  依据性质 1，对各层节点数求和，得到
  $\Sigma_{i=1}^{k}$(二叉树各层节点数的最大值) $= \Sigma_{i=1}^{k} 2^{i-1} = 2^k - 1$

二叉树的性质（2）

性质 3：不同类型节点的数量关系

- 结论：对任何一棵二叉树 T，如果其叶结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0 = n_2 + 1$
- 证明：
  若度为 1 的结点有 n_1 个，总结点个数为 n，总边数为 e，则根据二叉树的定义，结点总数为 $n = n_0 + n_1 + n_2$，边的总数为 $e = 2n_2 + n_1 = n–1$
  因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1 \Rightarrow n_2 = n_0 - 1 \Rightarrow n_0 = n_2 + 1$

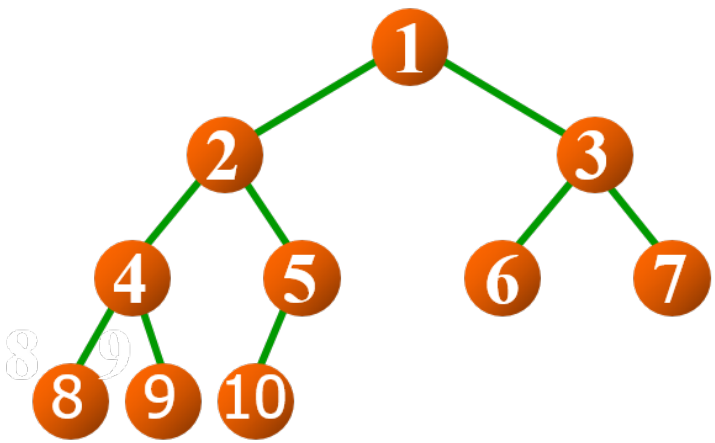
性质 4：完全二叉树的节点数量和深度的关系

- 结论：具有 $n(\geq 0)$ 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$
- 证明：
  设完全二叉树的深度为 h，则根据性质 2 和完全二叉树的定义有 $2^{h-1} - 1 < n \leq 2^h - 1$，即 $2^{h-1} \leq n < 2^h$
  取对数得 $h - 1 \leq \log_2 n < h$，因为 h 是整数，所以 $h - 1 = \lfloor \log_2 n \rfloor$
  故，$h = \lfloor \log_2 n \rfloor + 1$

二叉树的性质（3）

完全二叉树的节点间关系

- 如将一棵有 n 个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号 $1, 2, \ldots, n$，则有以下关系：
  - 若 $i = 1$，则 i 无双亲；
  - 若 $i > 1$，则 i 的双亲为 $\lfloor i/2 \rfloor$
  - 若 $2i > n$，则 i 无左孩子；否则，i 的左孩子为 $2i$；
  - 若 $2i + 1 > n$，则 i 无右孩子；否则，i 的右孩子为 $2i + 1$；



二叉树的性质（4）

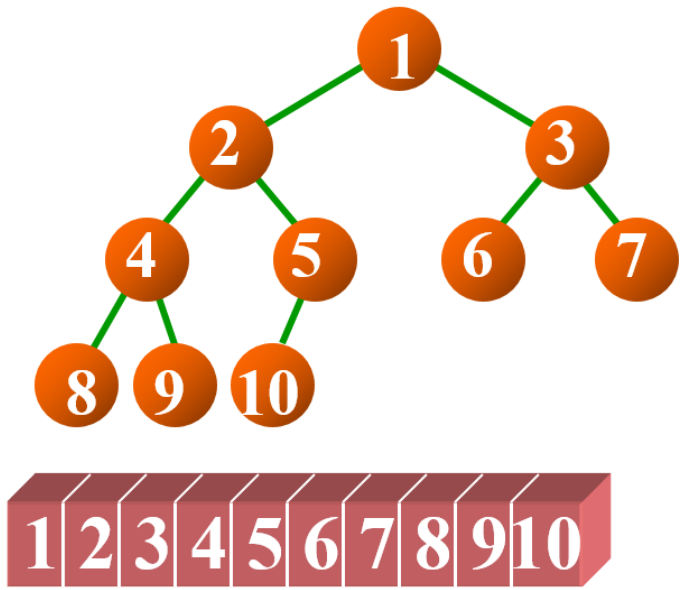
思考题

- 设二叉树，树叶 $l_1, l_2, \ldots, l_M$，各树叶具有的深度分别是 $d_1, d_2, \ldots, d_M$，(树根的深度为 0)
  证明: $\Sigma_{i=1}^{M} 2^{-d_i} \leq 1$
- 等号何时成立？
- (提示：假设老祖宗（树根）拥有 1 份财产，给每个后代 1/2；后代也会把自己继承的财产分自己的后代各 1/2；)

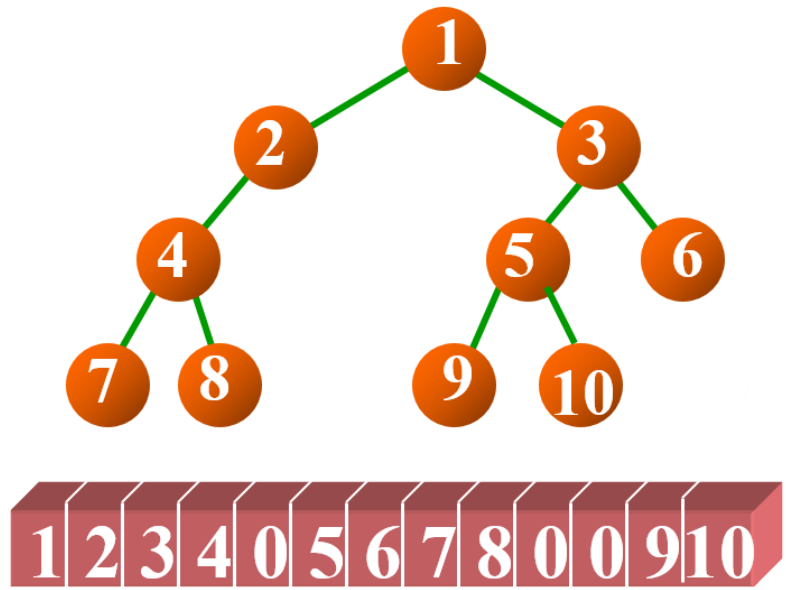
二叉树的存储结构

完全二叉树与顺序存储

- 存储效率低，大量占位用的 0



完全二叉树
的顺序表示

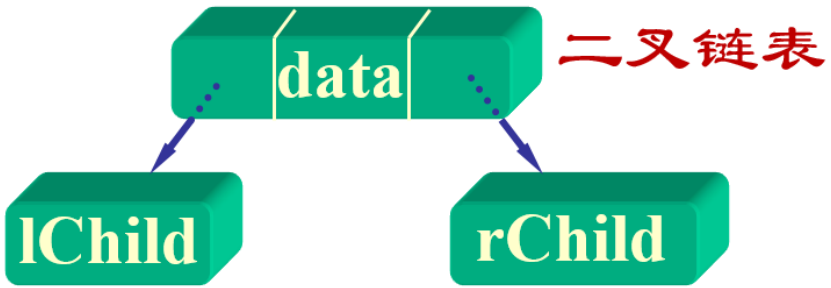


一般二叉树
的顺序表示

二叉树的存储结构

链式存储：二叉链表

- 结点包含数据域和左、右孩子结点的指针域，如图所示
- 含有 n 个结点的二叉链表中有 n+1 个空链域。



二叉链表

含两个指针域的结点结构

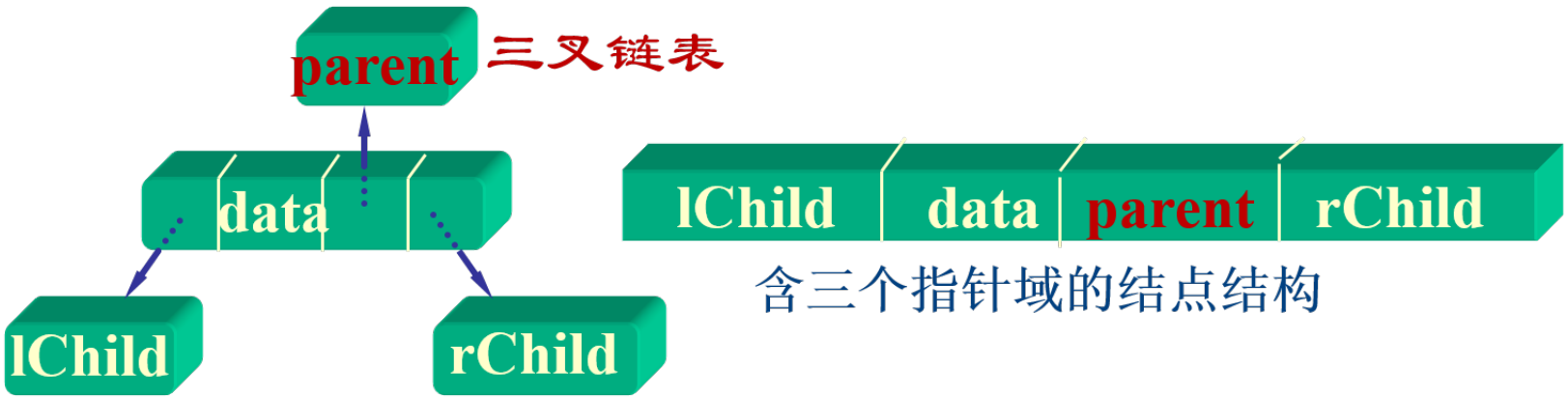
二叉链表在内存中的表示：静态部分

```
1  typedef struct BiTnode {    //结点定义
2      TElemType data;
3      struct BiTnode * lchild, * rchild;
4  } BiTNode,*BiTree;
```

二叉树的存储结构

链式存储：三叉链表

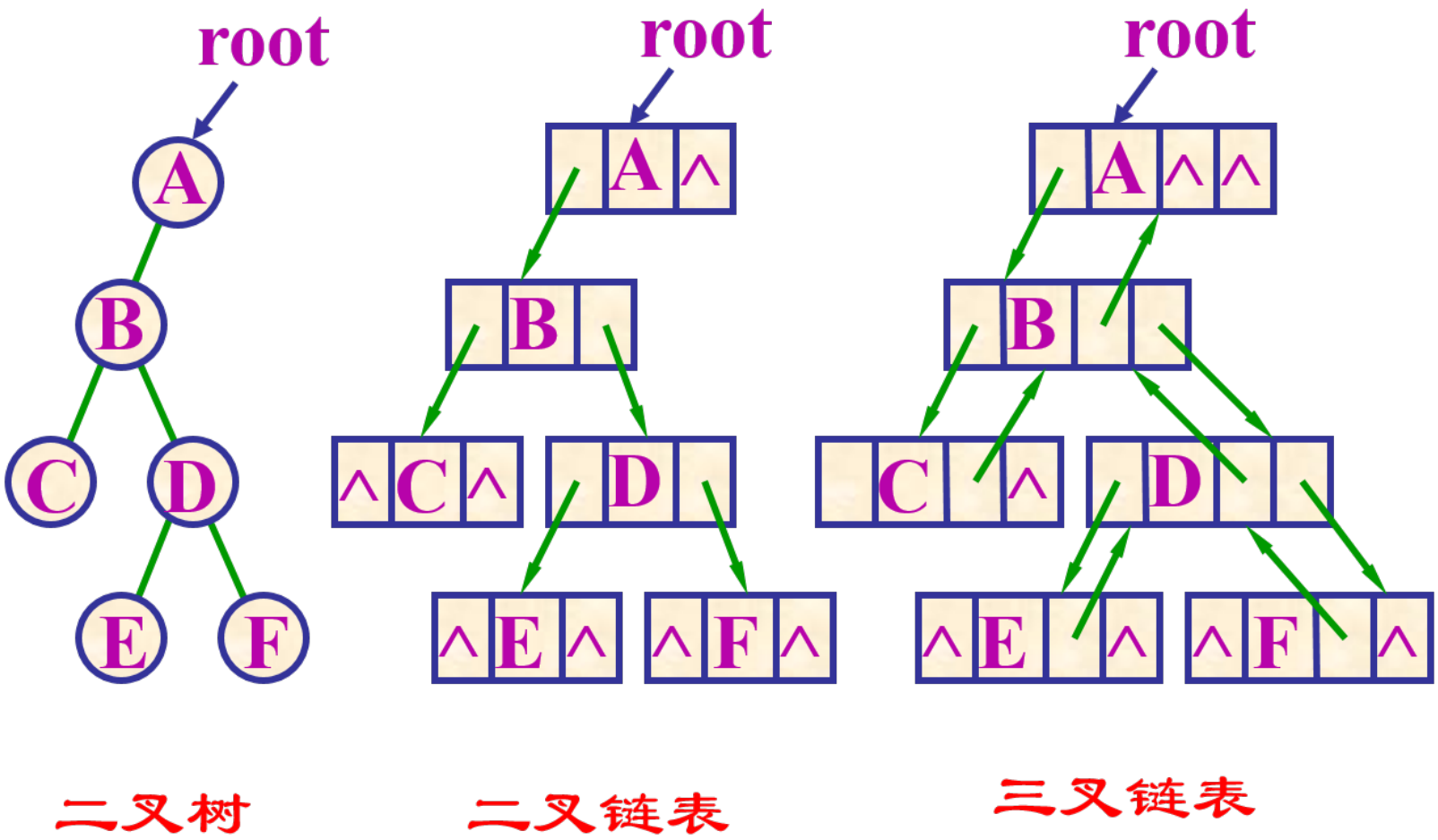
- 为便于找到结点的双亲，可引入 parent 指针域！



三叉链表在内存中的表示：静态部分

```
1  typedef struct BiTnode{    //结点定义
2      TElemType data;
3      struct BiTnode *lchild, *rchild, *parent;
4  } BiTNode,*BiTree;
```

二叉树的链式存储结构例子



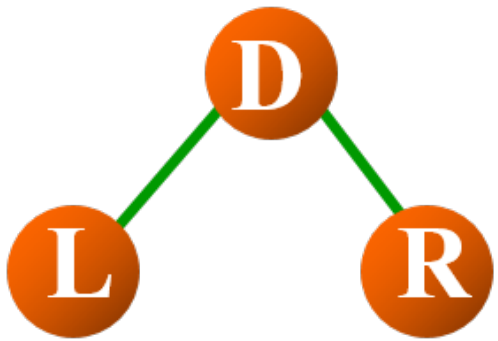
二叉树基本操作的实现

核心操作：遍历二叉树

- 树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次；所谓“访问”可理解为读取或修改节点的信息。
- 简化理解：树的遍历就是将树的节点转换为线性序列（按访问次序）
- 遍历操作是树的所有操作中最基本、最核心的
- 如右下图所示，设访问根结点记作 D，遍历根的左子树记作 L，遍历根的右子树记作 R

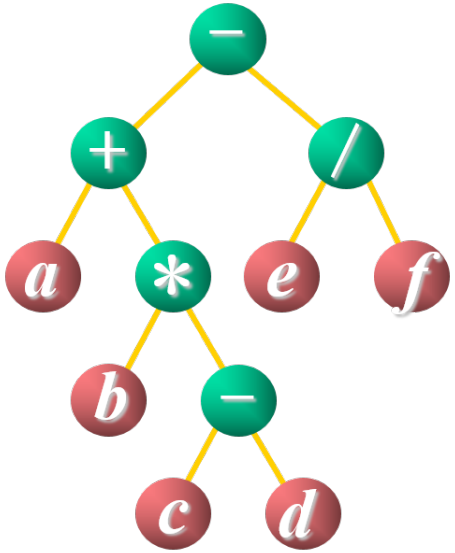
可能的遍历二叉树的次序包括

- LDR $\Rightarrow$ 中序遍历
- DLR $\Rightarrow$ 先序遍历
- LRD $\Rightarrow$ 后序遍历

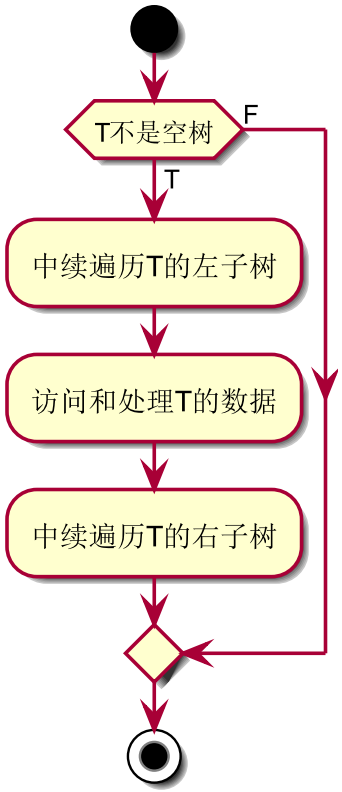


中序遍历

- 如右图所示二叉树，执行中序遍历，算法思想为：
  1. 若二叉树为空，则空操作；否则
  2. 中序遍历左子树 (L)；
  3. 访问根结点 (D)；
  4. 中序遍历右子树 (R)。
- 中序遍历结果为 a+b*c-d-e/f，中缀式



中续遍历二叉树

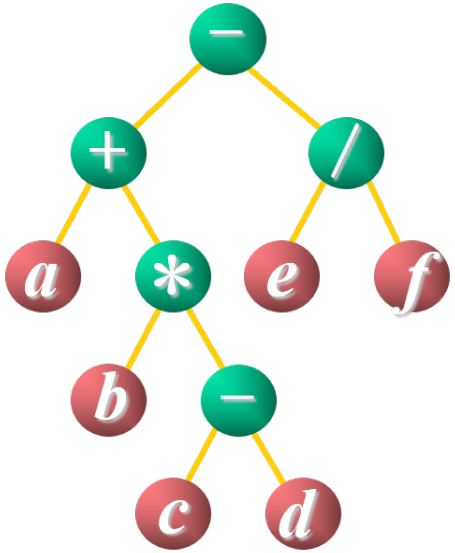


中序遍历伪代码

```
1  Status InOrder(BiTree
       T,Status(*Visit)(TElemType e)){
2      if (T) {
3          InOrder(T->lchild);
4          Visit(T->data);
5          InOrder(T->rchild);
6      } else return OK;
7  }
```

先序遍历

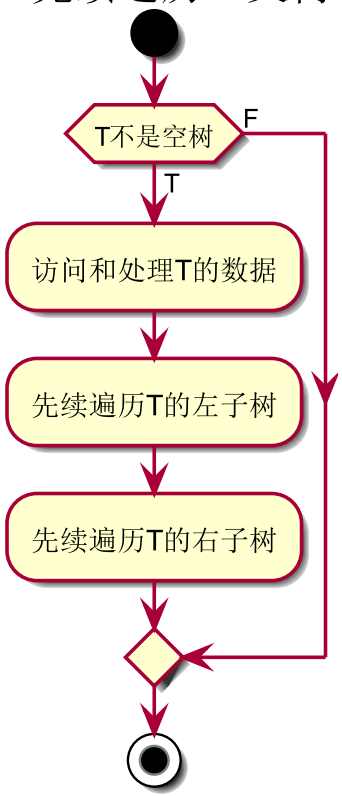
- 如右图所示二叉树，执行先序遍历，算法思想为：
  1. 若二叉树为空，则空操作；否则
  2. 访问根结点 (D)；
  3. 先序遍历左子树 (L)；
  4. 先序遍历右子树 (R)。
- 遍历结果-+a*b-cd/ef，前缀式/波兰式



先序遍历伪代码

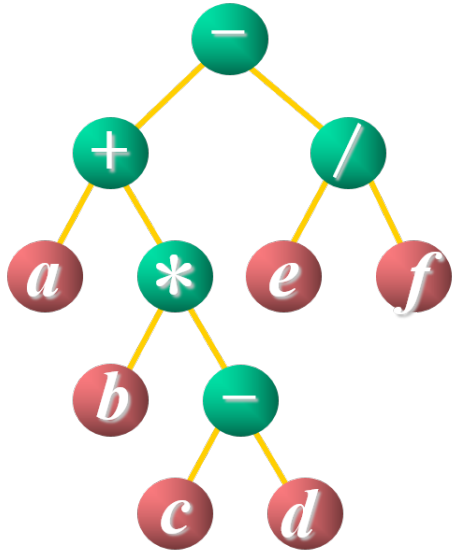
```
1  Status PreOrder(BiTree
       T,Status(*Visit)(TElemType e)){
2      if (T) {
3        Visit(T->data);
4        PreOrder(T->lchild);
5        PreOrder(T->rchild);
6      } else return OK;
7  }
```

先续遍历二叉树

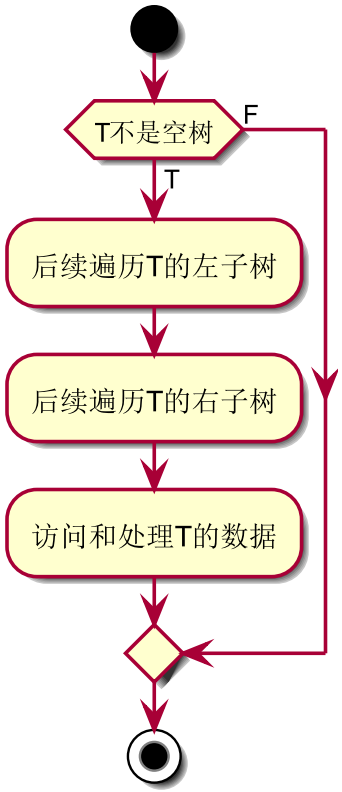


后序遍历

- 如右图所示二叉树，执行后序遍历，算法思想为：
  1. 若二叉树为空，则空操作；否则
  2. 后序遍历左子树 (L)；
  3. 后序遍历右子树 (R)。
  4. 访问根结点 (D)；
- 遍历结果 abcd-*+ef/-，后缀式/逆波兰式



后续遍历二叉树



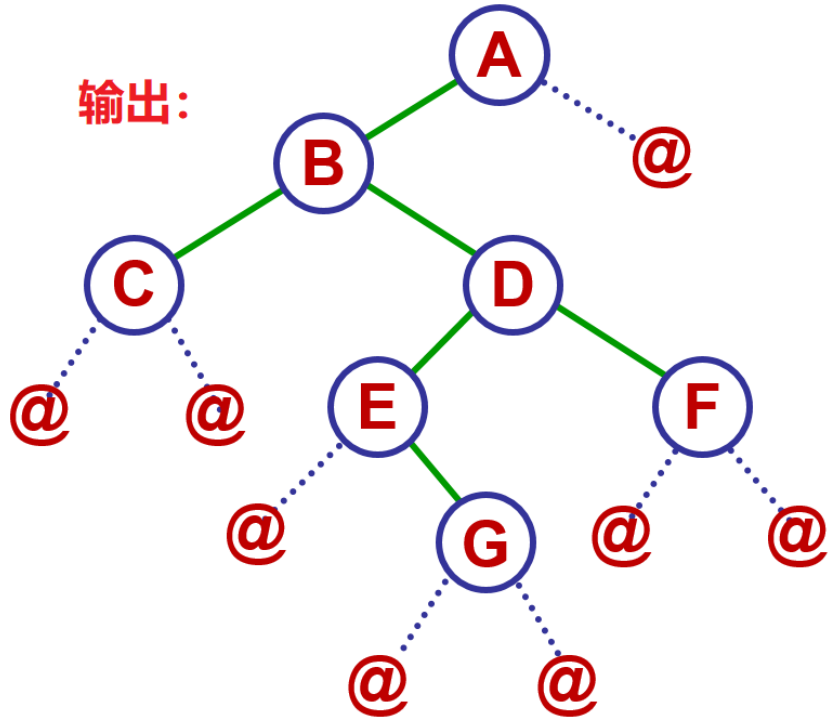
后序遍历伪代码

```
1  Status PostOrder(BiTree
       T,Status(*Visit)(TElemType e)){
2    if (T) {
3      PostOrder(T->lchild);
4      PostOrder(T->rchild);
5      Visit(T->data);
6    } else
7    return OK;
8  }
```

构建二叉树

输入/输出

- 输入：给定一个二叉树的先序遍历序列，如下图所示。以输入序列中不可能出现的值作为空结点的值，例如用“@”表示先序序列的空结点。
- 输出：二叉树（内存中），如右图所示



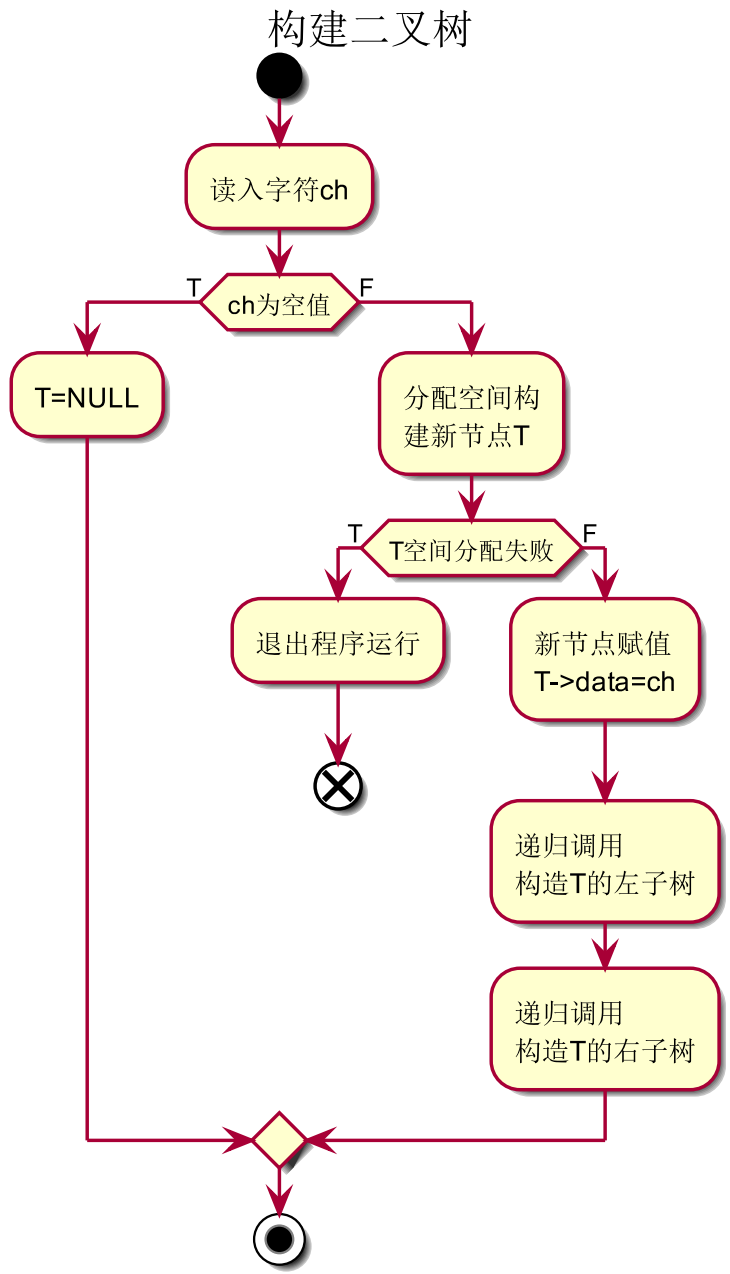
输入： 二叉树的先序遍历顺序为

A B C @ @ D E @ G @ @ F @ @ @

构建二叉树

递归实现的伪代码

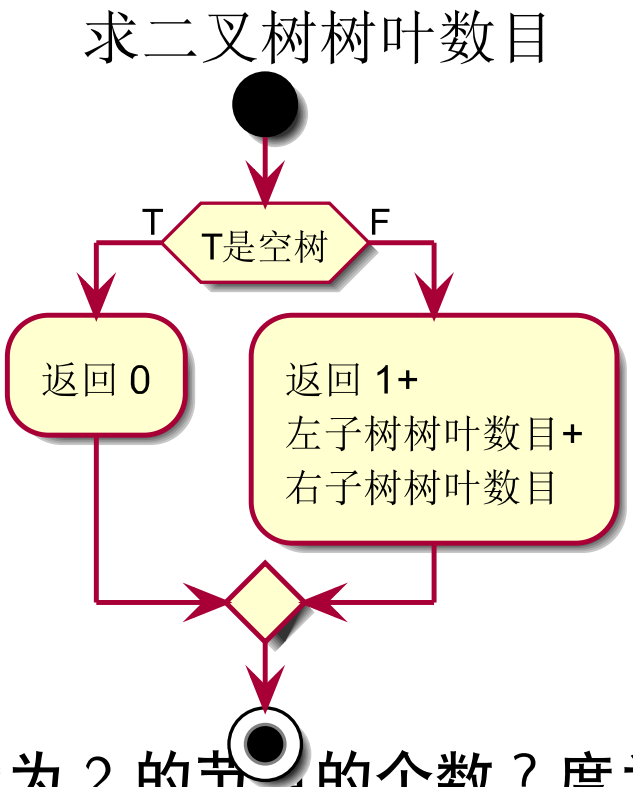
```
1  Status CreateBiTree (BiTree &T){
2    scanf(&ch);
3    if (ch==RefValue)
4      T =NULL; //递归出口
5    else {
6      T=(BiTNode *)malloc(sizeof(BiTNode));
7      if (!T) exit (OVERFLOW);
8      T->data = ch; //生成根结点
9      CreateBiTree(T->lchild ); //构造左子树
10     CreateBiTree(T->rchild ); //构造右子树
11   }
12   return ok;
13 }
14 //写代码时注意：这里用&参数仅指示T会改变
15 //若想实现带回返回值，将&改成*，方式错误
```



二叉树遍历的更多应用

计算二叉树节点的个数

```
1  int Count(BiTree T){
2    if (T==NULL) return 0;
3    else return 1+Count(T->lchild)
4                 +Count(T->rchild);
5  }
```



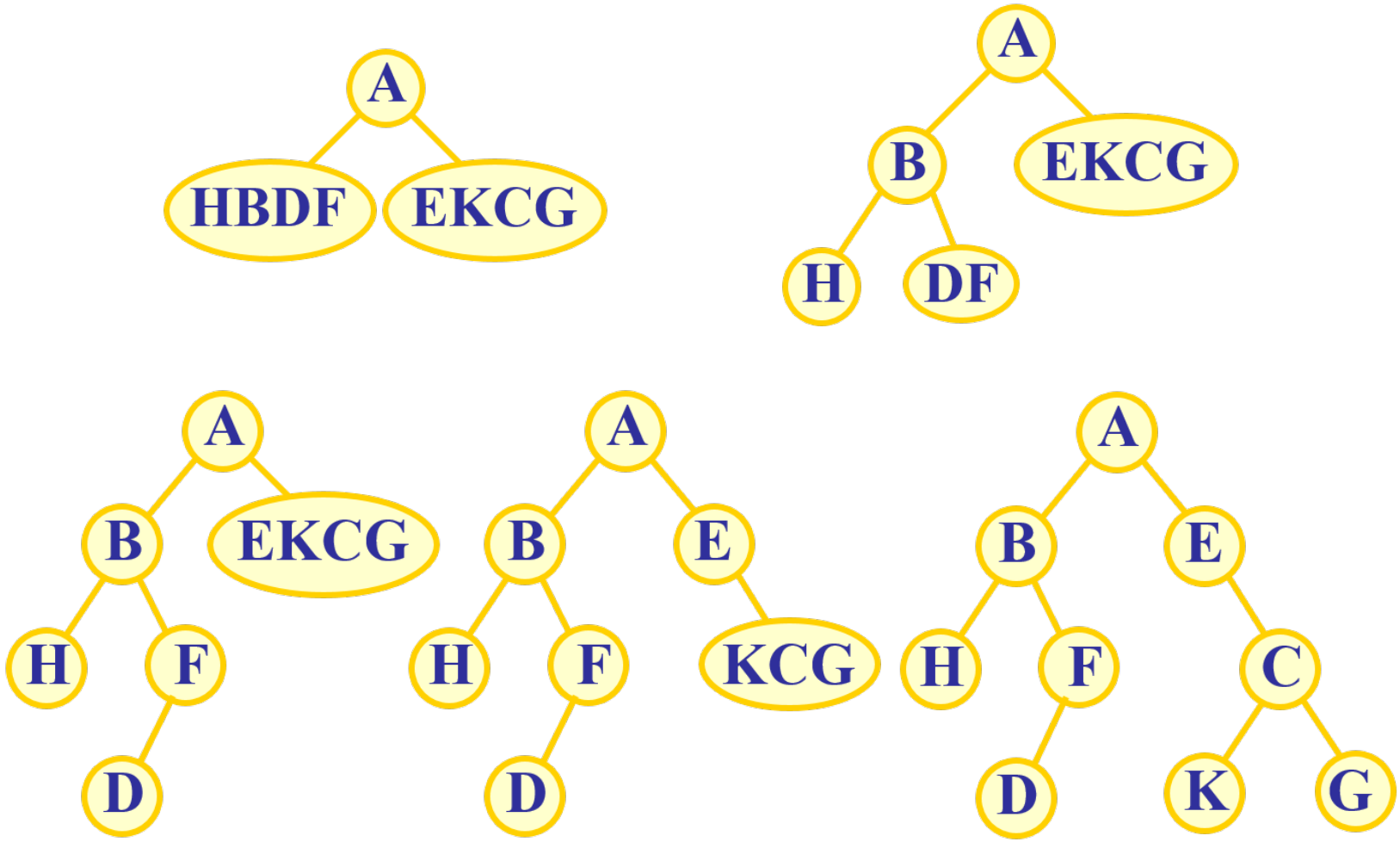
思考题

- 求叶节点的个数？非叶节点的个数？度为 2 的节点的个数？度为 1 的节点的个数？非叶节点上数据的和？
- 求二叉树的高度？求某个节点，其两棵子树的高度差的绝对值最大？
- 两棵二叉树相似，当且仅当二者都为空树，或者同时具有相似的左子树和右子树，编程判断两棵二叉树是否相似。

练习题

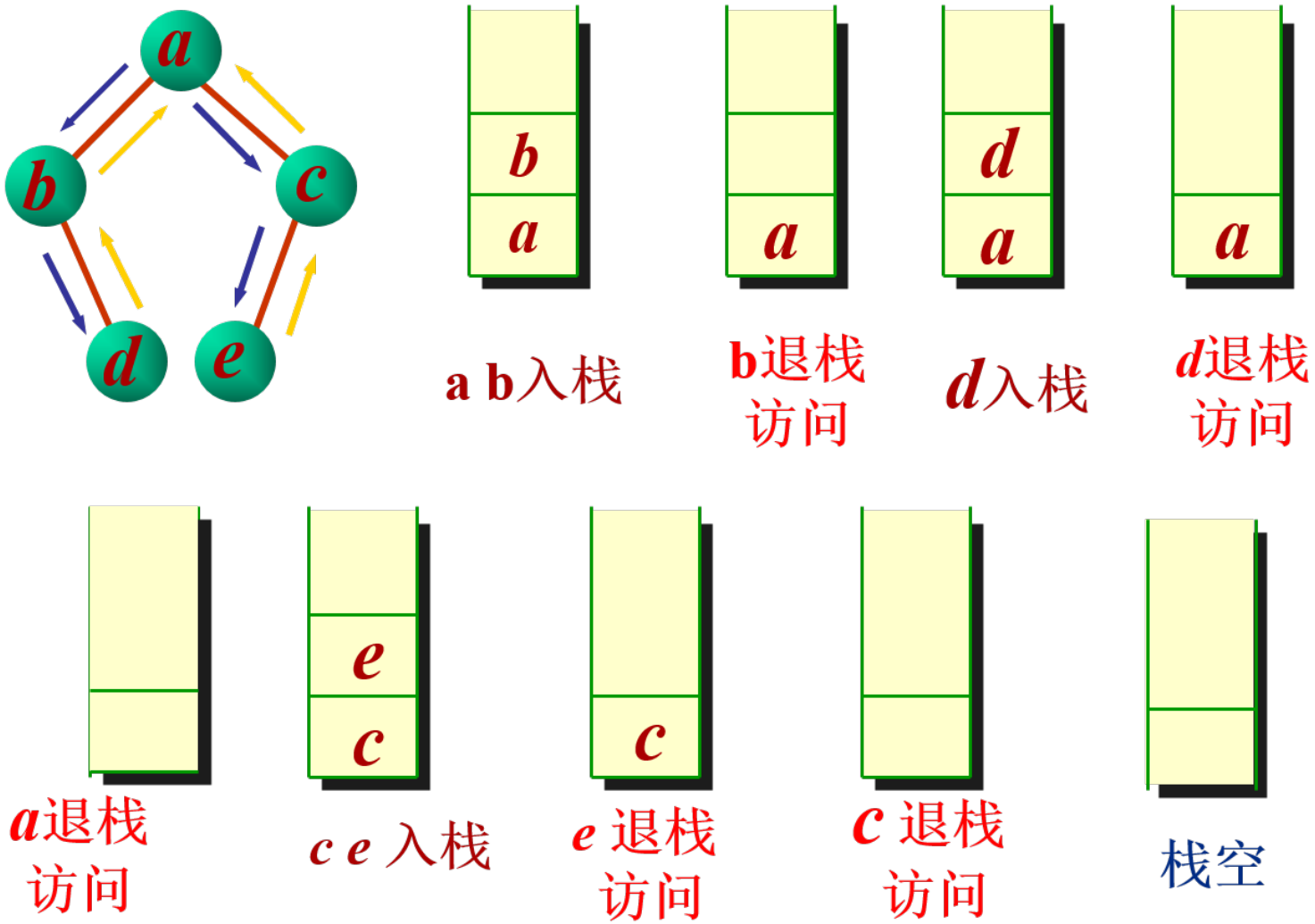
由二叉树的前序序列和中序序列可唯一地确定一棵二叉树

- 给定前序序列 ABHFDECKG 和中序序列 HBDFAEKCG，构造二叉树



中序遍历的非递归算法

例子图解



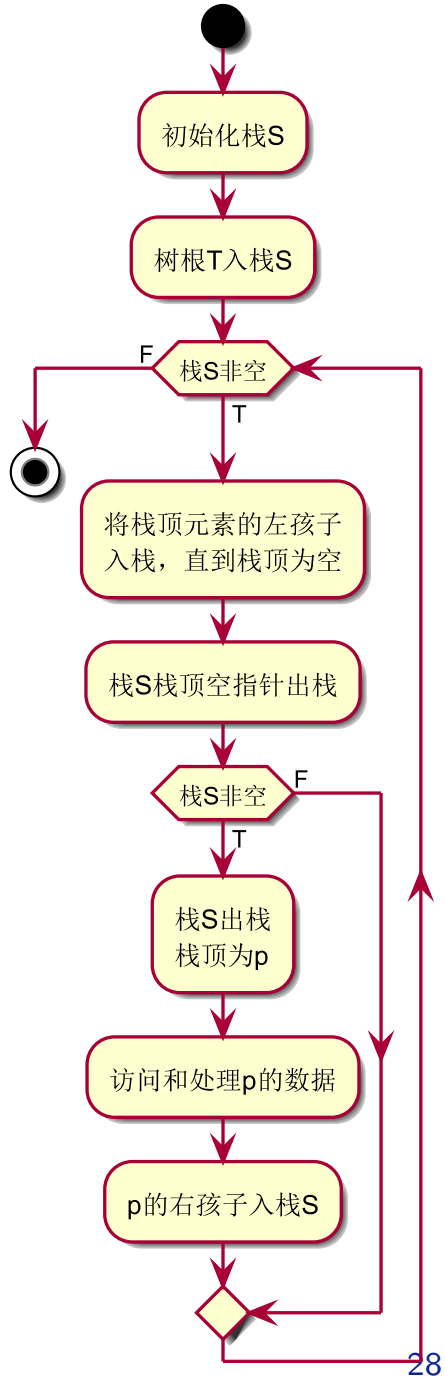
二叉树遍历的非递归伪代码

中序遍历二叉树非递归算法思想

- 非递归算法利用栈保存子树的根，出栈时访问

```
1 //非递归二叉树中序遍历算法
2 Status InOrderTraverse(BiTree
      T,Status(*Visit)(TElemType e)){
3   InitStack(S);
4   Push(S,T) //根指针进栈
5   while(!StackEmpty(S)){
6     while(GetTop(S,p)&&p)
7       Push(S,p->lchild);//向左走到尽头
8     Pop(S,p);              //空指针退栈
9     if (!StackEmpty(S)){//栈非空
10      Pop(S, p);
11      Visit(p->data); //退栈访问
12      Push(S, p->rChild) ; //向右一步
13    }} //endif endwhile
14 return ok;}
```

二叉树中序遍历的非递归算法



先序遍历的非递归算法

算法思想

- 栈的特性是“后入先出”，也就是说，要右子树节点后出（后访问），就需要先入栈，或者是在其它数据“入栈 + 出栈”后再入栈
- 根据先序遍历的特点，一个节点 p，其右子树的节点访问在左子树之后；因此，右子树应该先入栈，或者左子树节点“入栈 + 出栈”之后再入栈（但是此时如何找到节点 p，再找到其右子树，成为一个问题，我们丢失节点 p 这个指针）
- 整体思路：从根节点开始，入栈；节点 p 出栈，若 p 非空时，访问，再将其非空右孩子入栈，然后再将其左孩子入栈，接下来出栈，并形成循环；
- 注意与中序非递归算法另一个区别：左孩子进栈一个，就考虑出栈，而非左孩子一直入栈到底（中序非递归算法 while 循环）。

课后编写代码

后序遍历的非递归算法

算法思想

- 整体思想：有中序非递归遍历的思想，也有先序非递归遍历的思想。先将左孩子尽可能入栈（中序非递归算法），但是在左孩子入栈之前，先把非空的右孩子入栈（先序非递归算法），保证右子树先入栈，后访问。
- 基本过程：从根节点开始，入栈；节点 p 出栈，若 p 非空时，将其非空右孩子入栈，然后再将其左孩子入栈，并循环；
- 注意当一个节点左右子树都访问过了，才访问该节点，如何记录和判断这个条件？逻辑上复杂，实现的难点。

课后编写代码

- 算法的特点：打算访问某个节点 p 时，它的左右子树都已经访问过了，栈内无任何子孙节点，而其“直系祖先”都在栈内；
- 算法的应用：求根到叶的一条路径上的边权值或顶点权值之和；求一个节点的所有“直系祖先”等等

线索二叉树

定义与理解

- 二叉树的遍历输出一个特定的线性序列（节点的访问次序），本质上是线性化树这种非线性的结构
- 能否保存遍历序列的直接前驱和直接后继关系？这些关系在遍历过程中，“打印”出来了，但是在内存中并没有存在（比如构造了一个单链表）。
- 修改数据结构：节点增加前驱和后继指针，在执行二叉树的遍历时，修改这些指针的值。
  - 如何写算法？
  - 存在问题：存储密度低，浪费空间

利用二叉链表的空链域，构造存储二叉树的线索链表

- 二叉链表有 $n+1$ 个空指针，利用它们指向前驱和后继节点
- 要设计方法来区分节点指针是指向了左右孩子还是前驱或后继

线索二叉树的实现

节点结构

- 如图所示
- lTag=0, lChild 指向节点的左孩子；lTag=1, lChild 指向节点的前驱
- rTag=0, rChild 指向节点的右孩子；rTag=1, rChild 指向节点的后继

线索链表的节点结构

相关概念

- 线索链表：以上述节点构成的，存储二叉树的链表
- 线索：指向前驱和后继的指针
- 线索二叉树：包含线索的二叉树
- 线索化：对二叉树进行某种次序的遍历，同时使其成为线索二叉树的过程

线索二叉树例子

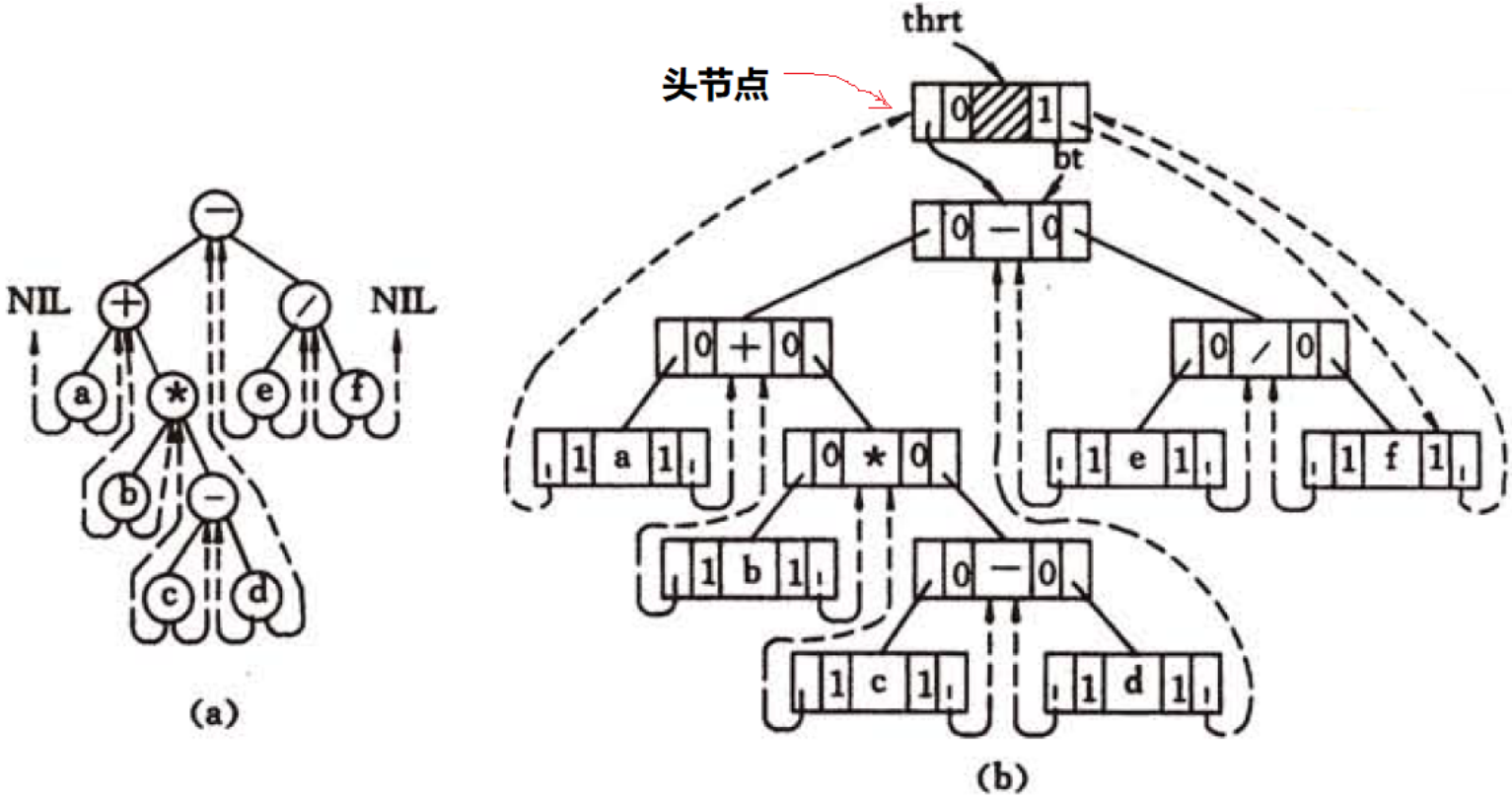


图 6.11　线索二叉树及其存储结构

(a) 中序线索二叉树；(b) 中序线索链表

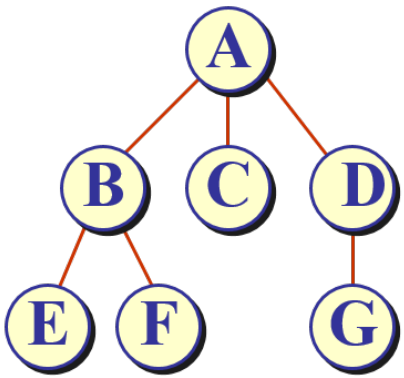
线索链表存在问题

- 浪费存储空间
- 操作复杂，比如找后继或前驱 (感兴趣的同学课后思考不同次序遍历的线索二叉树如何找前驱和后继)

树的存储结构

双亲表示法

- 以一组连续空间存储树的结点，同时在结点中附设一个指针，存放双亲结点在链表中的位置。
- 便于找双亲，找孩子耗时；适用于以找双亲为主的操作或算法。



	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	−1	0	0	0	1	1	3

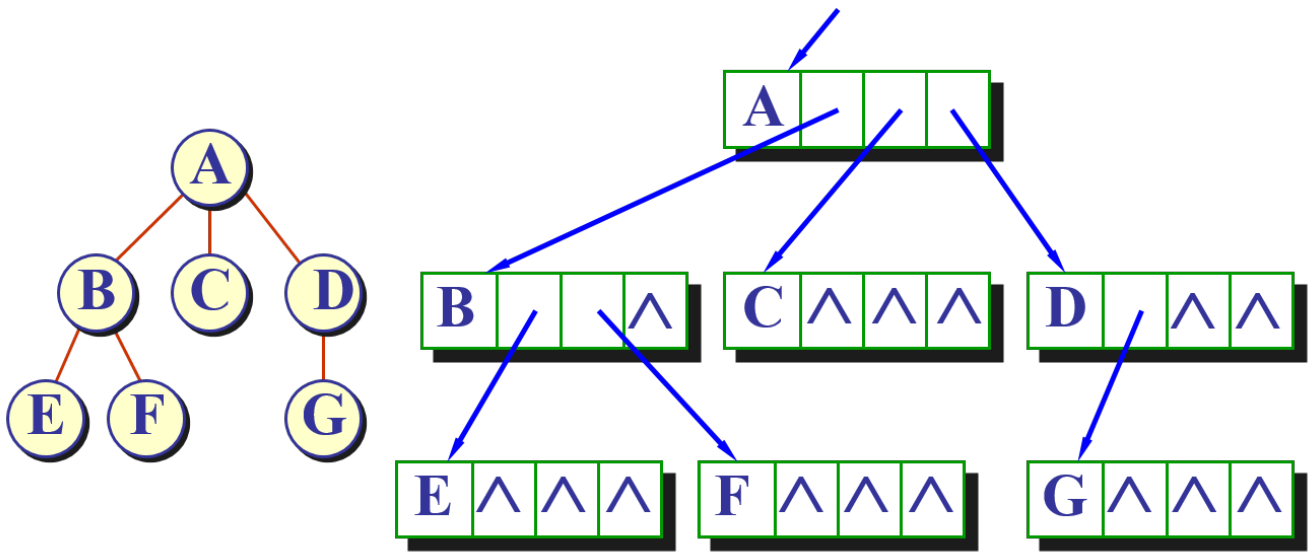
在内存中的表示

```
1  #define MAX_TREE_SIZE  //最大结点个数
2  typedef struct PTNode { //树结点定义
3      TElemType data;
4      int parent;
5  } PTNode;
6  typedef struct{
7        PTNode nodes[MAX_TREE_SIZE];
8        int r,n;          //根的位置和结点数
9  }Ptree;
```

树的存储结构

孩子表示法一：多重链表

- 每个结点有多个指针域分别指向子树的根结点。如下图
- 两种实现：节点的结构相同，节点含的孩子指针数目等于树的度 (存储效率低)；或者节点的孩子指针数目等于实际需求 (操作复杂性大大增加)
- 思考：优点？缺点？(提示：操作复杂性，存储密度等)



同构的结点

树的存储结构

孩子表示法二：孩子链表（图的邻接表）

- 把每个结点的孩子结点组织为一个单链表，n 个单链表的头指针组成一个线性表。如图所示例子
- 找孩子的操作非常简便，但找双亲复杂；适用于操作主要是找孩子的算法
- 改进孩子链表为带双亲的孩子链表，使得找双亲简单化
- 课后思考：如何在内存中表示孩子链表？

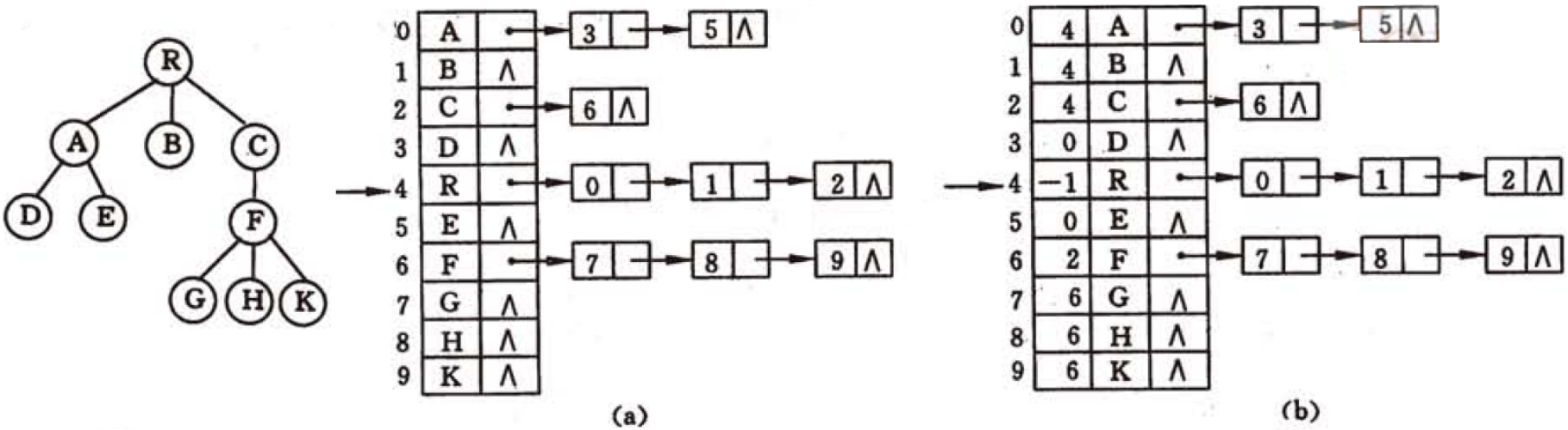


图　树的另外两种表示法

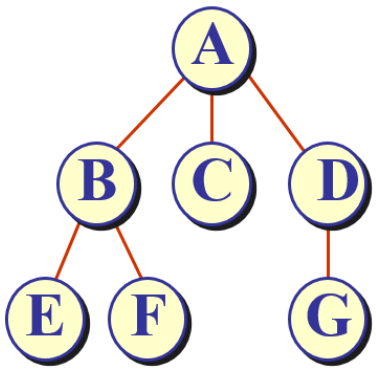
（a）孩子链表；（b）带双亲的孩子链表

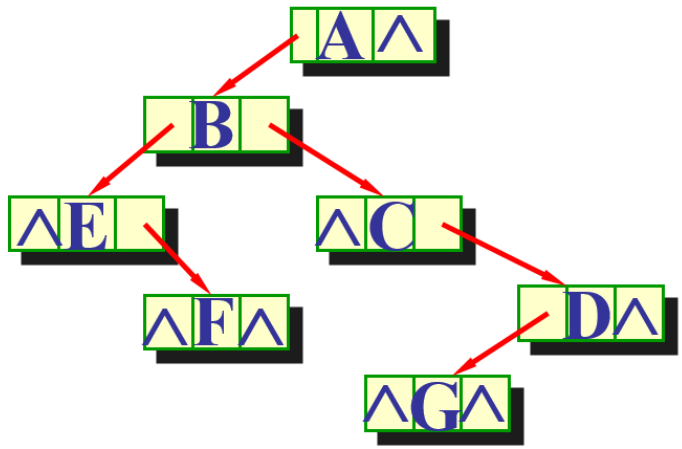
树的存储结构

孩子兄弟表示法

- 二叉链表在存储表示和操作实现上的简单性使我们想将树用二叉链表表示出来，如图所示例子

结点结构





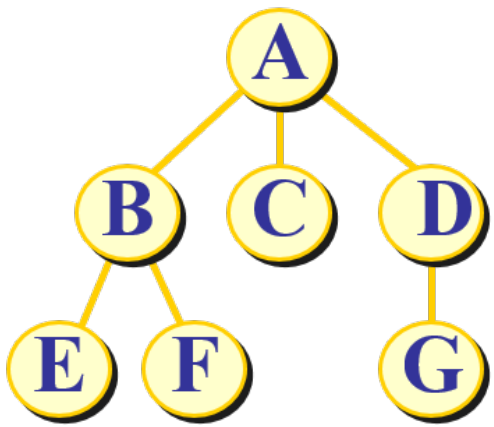
在内存中的表示

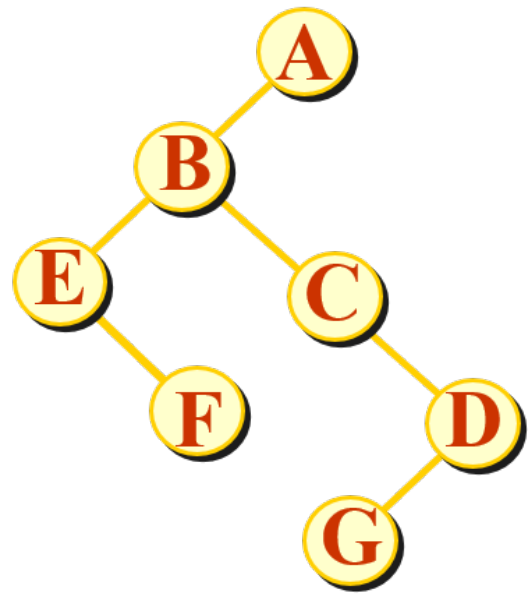
```
1  typedef struct CSNode {
2      ElemType data;
3      struct CSNode *firstChild, *nextSibling;
4  } CSNode ,* CSTree;
```

孩子兄弟表示法的优点

将任意的树转换成二叉树

- 我们已经熟悉二叉树的表示和操作，学会转换，问题就解决了
- 如下图所示的树转换为二叉树的例子

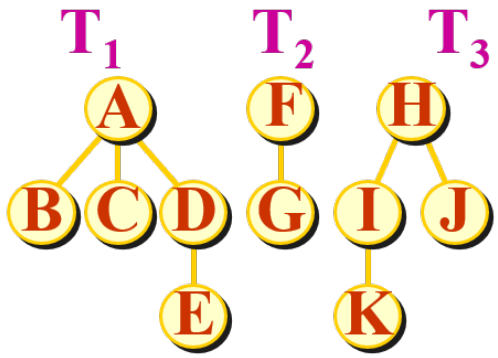




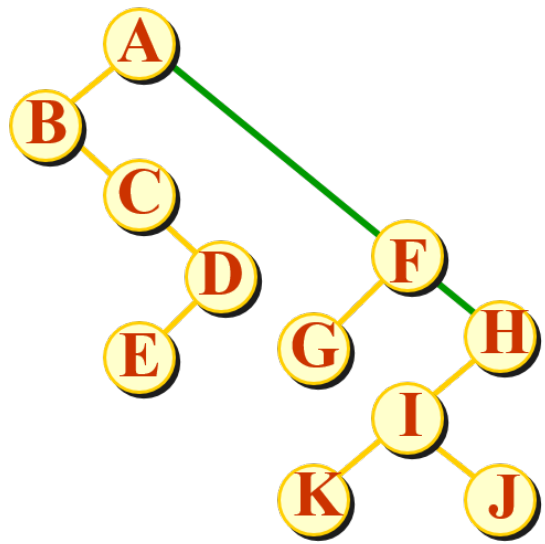
孩子兄弟表示法的优点

将任意的森林转换成二叉树

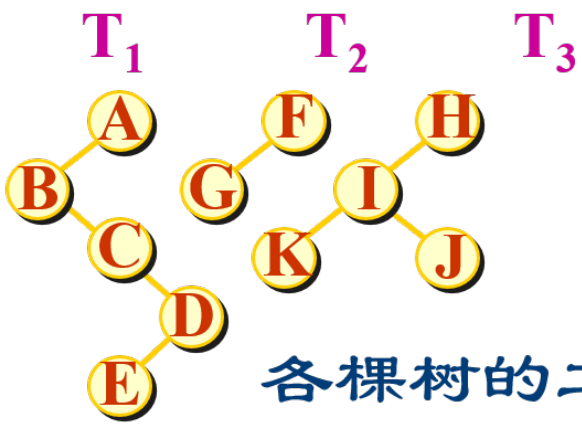
- 增加一个虚拟的“根节点”连接所有的树的根节点，我们得到一棵树；问题转换为“树 $\Rightarrow$ 二叉树”问题
- 如下图所示的森林转换为二叉树的例子



3 棵树的森林



森林的二叉树表示



各棵树的二叉树表示

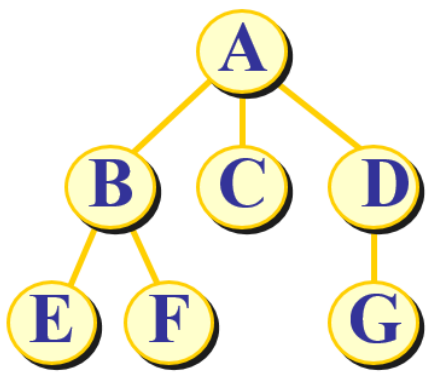
树的遍历

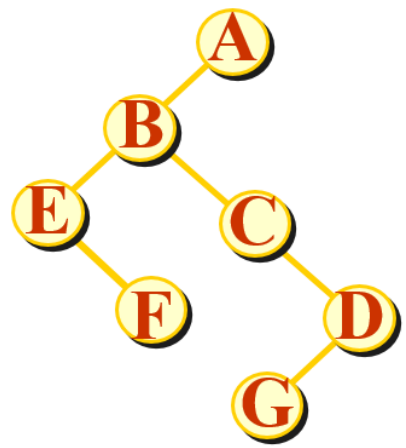
树与二叉树遍历的不一样

- 没有中序，因为一般的树若包含三个及以上的孩子，中序访问根节点时，第几个访问根时称为“中序”？
- 因此，只有先序或后序访问根节点，此时称为树的“先根遍历”和“后根遍历”

树的先根遍历

- 算法思想 (递归)：当树非空时，先访问根结点；然后依次先根遍历根的各棵子树
- 如左下图所示的树，右子图是其孩子兄弟表示的二叉树，树先根遍历 ABEFCDG，对应的二叉树的先序遍历 ABEFCDG
- 故，树的先根遍历可以借用对应二叉树的先序遍历算法实现

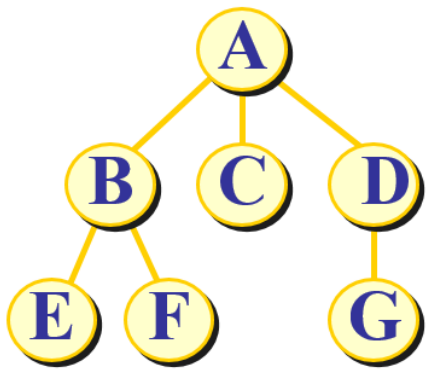


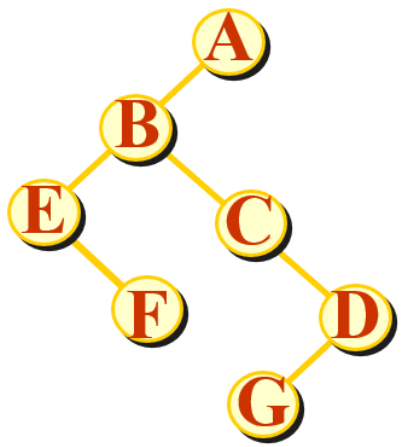


树的遍历

树的后根遍历

- 算法思想 (递归)：当树非空时，先依次后根遍历根的各棵子树，然后访问根结点；
- 如左下图所示的树，右子图是其孩子兄弟表示的二叉树，树后根遍历 EFBCGDA，对应的二叉树的中序遍历 EFBCGDA
- 故，树的后根遍历可以借用对应二叉树的中序遍历算法实现





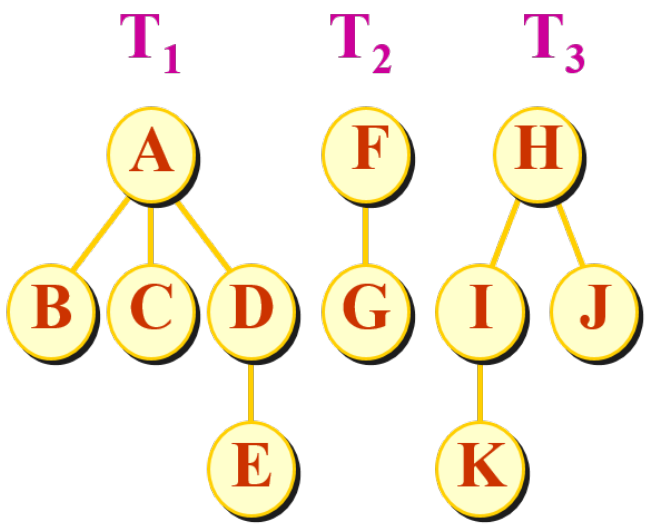
森林的遍历

森林与树、二叉树的不一样

- 先序遍历：若森林非空，则：访问森林第一棵树的根；然后先序遍历第一棵树去掉根之后的子树森林；在先序遍历除去第一棵树之后剩余的森林
- 中序遍历：若森林非空，则：中序遍历森林第一棵树的根的子树森林；访问森林第一棵树的根；中序遍历除去第一棵树之后剩余的森林
- 森林的先序遍历就是依次先根遍历各棵子树；森林的中序遍历就是依次后根遍历各棵子树
- 森林的先序和中序遍历和对应二叉树的先序和中序遍历结果相同。

例子

- 遍历如右图所示森林
- 先序遍历：ABCDEFGHIKJ
- 中序遍历：BCEDAGFKIJH



哈夫曼树的应用背景

数据压缩

- 无损压缩：数据文件被压缩，解压后恢复原始数据，不损失任何数据；如zip,rar,7zip
- 有损压缩：更为常见，如图像、视频、语音等的各种格式，其格式标准基本都是数据压缩标准
- 数据压缩的基本原理：数据（文件）就是一长的 0-1 串，统计发现数据中 0-1 子串出现的规律，将出现频率很高的、较长的 0-1 子串用较短的 0-1 串表示出来；
- 人工智能中的机器学习的本质就是设计算法发现 0-1 串中的规律，并用函数或程序代码（如神经网络）表示该规律，实现数据压缩。

数据压缩存在的问题

- 如何寻找规律：子串的长度是多少？寻找子串的时间代价？子串出现的分布是什么？
- 假设子串的出现规律找到，如何写算法实现压缩过程。（利用哈夫曼树实现哈夫曼编码）

数据压缩的例子

无损压缩：哈夫曼编码

- 输入：设给出一个字符串：CASTCASTSATATATASA
- 输出：每个字母映射成一个 0-1 串
- 要求：上述字符串用 0-1 表示后长度最短；同时能解码回输入字符串

有损压缩：Gif 图像

- 输入：任给一张图像，比如一张 3 MB 的图像，由 1M 个像素点构成，每个像素点 3Bytes（RGB 三原色），可能包含 2^{24} 种不同颜色
- 输出：Gif 图像
- Gif 图像的特点：只有 256 中颜色
- 处理的基本方法：把给定图像中的不同颜色想办法聚类成 256 种颜色，这样原始图像的每个像素点就可以用 1Bytes 来表示

压缩即编码，编码是什么？

- 关键在于找到一张映射表：原始 0-1 串 $\Rightarrow$ 编码后的 0-1 串

哈夫曼树

什么是哈夫曼树

- 带权路径长度最小的二叉树

路径长度

- 两个结点之间的路径长度是连接两结点的路径上的分支数。
- 树的路径长度：从根到非根节点的路径长度之和
  - 树的外部/内部路径长度：从根到所有叶/非叶节点的路径长度之和

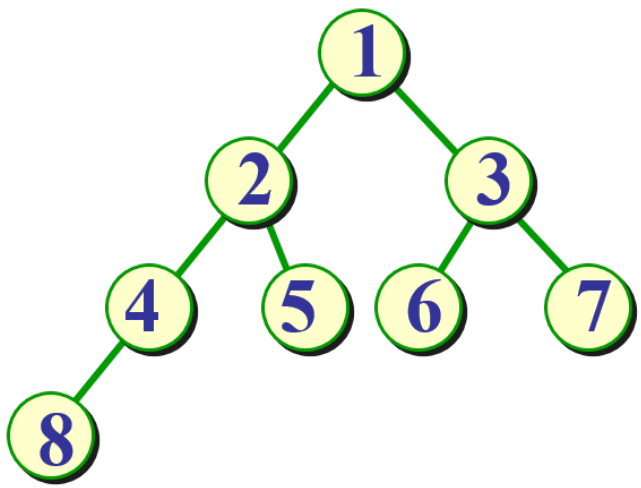
带权路径长度

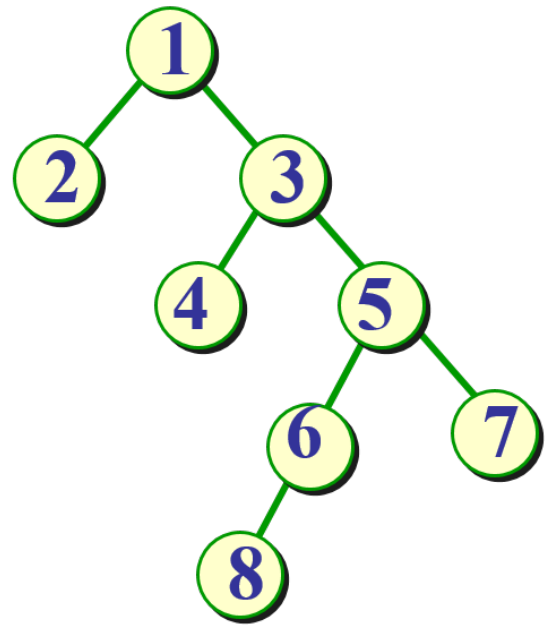
- 每个叶节点 i 都有一个权值 w_i，根节点到叶节点 i 的路径长度 l_i 与权值的乘积即为带权路径长度
- 树的（外部）带权路径长度：从根到所有叶节点的带权路径长度之和 $\Sigma_i w_i l_i$
  - 理解：每片树叶上有 w_i 只蚂蚁，所有蚂蚁都要爬到树根，树的带权路径长度就是所有蚂蚁爬过的距离之和

树的路径长度

如图所示的例子

- 左图：树的外部路径长度 $= 3 * 1 + 2 * 3 = 9$
- 右图：树的外部路径长度 $= 1 * 1 + 2 * 1 + 3 * 1 + 4 * 1 = 10$

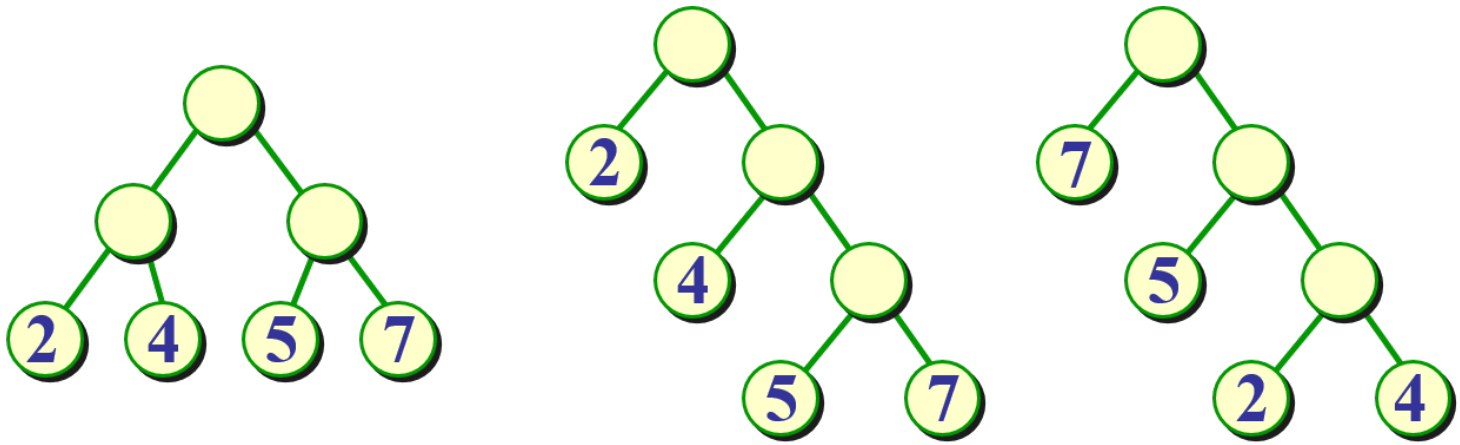




树的带权路径长度

如图所示的例子

- 左图：树的带权外部路径长度 $2*2+4*2+5*2+7*2=36$
- 中图：树的带权外部路径长度 $2*1+4*2+5*3+7*3=46$
- 右图：树的带权外部路径长度 $2*3+4*3+5*2+7*1=35$



右图即为哈夫曼树，权值越大的叶节点离根越近！数据压缩时，出现频率越高的子串权值越大。

构造哈夫曼树

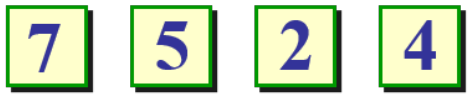
算法思想

- 输入：给定 n 个权值 $\{w_0, w_1, \ldots, w_{n-1}\}$
- 输出：包括 n 个叶节点的二叉树，其带权路径长度最小。
- 步骤：
  1. 为每个权值构造一棵只有根节点的二叉树，获得 n 棵树构成的森林；
  2. 选择森林里根节点权值最小的两棵树，若有多棵树具有相同的最小权值，随机选择两棵；
  3. 用选出的两棵树作为左右孩子，合并出一棵新的树，新树的根的权值为左右孩子根的权值之和；
  4. 用新树替换森林中被合并的两棵树；森林里的树此时减少了一棵；
  5. 重复上述合并过程（步骤 2-4），直到森林里只有一棵树时，算法结束。

哈夫曼树构造的过程

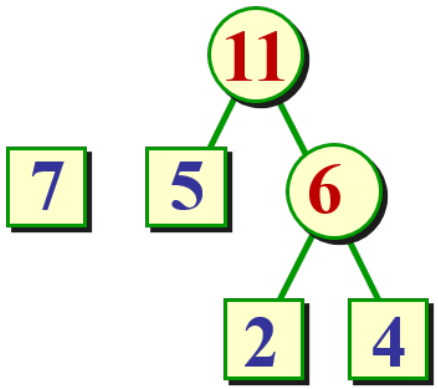
例子如图所示

F : {7} {5} {2} {4}



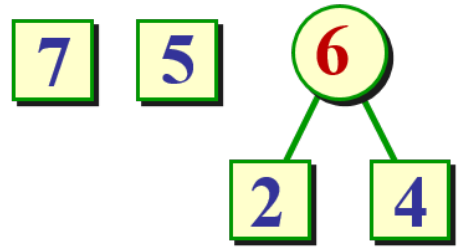
初始

F : {7} {11}



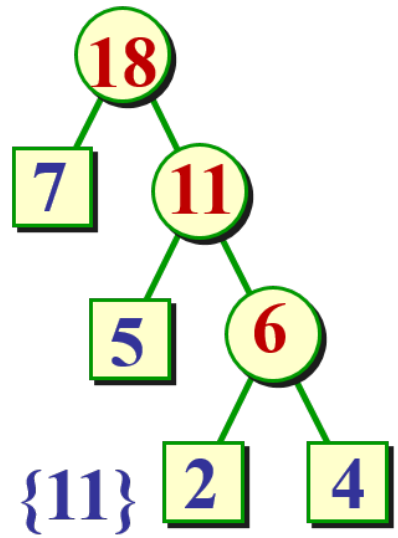
合并{5} {6}

F : {7} {5} {6}



合并{2} {4}

F : {18}



合并{7} {11}

哈夫曼树的构造过程

结合特定的存储结构来实现

- 利用一个结构体数组来实现，如下图所示初态

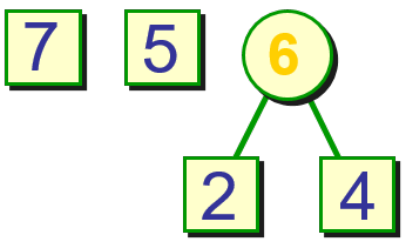
初 态 4 棵树

	Weight	parent	leftChild	rightChild
0	7	−1	−1	−1
1	5	−1	−1	−1
2	2	−1	−1	−1
3	4	−1	−1	−1
4		−1	−1	−1
5		−1	−1	−1
6		−1	−1	−1

哈夫曼树的构造过程

结合特定的存储结构来实现

- 利用一个结构体数组来实现，如下图所示过程 1



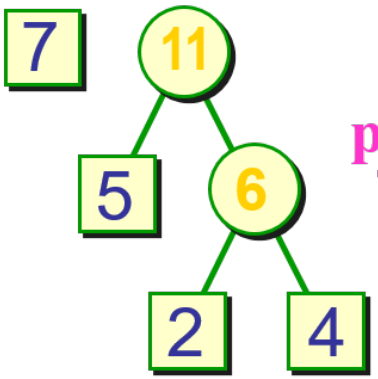
过 程 1

	Weight	parent	leftChild	rightChild
0	7	−1	−1	−1
1	5	−1	−1	−1
p1 → 2	2	4 ~~−1~~	−1	−1
p2 → 3	4	4 ~~−1~~	−1	−1
i → 4	6	−1	2 ~~−1~~	3 ~~−1~~
5		−1	−1	−1
6		−1	−1	−1

哈夫曼树的构造过程

结合特定的存储结构来实现

- 利用一个结构体数组来实现，如下图所示过程 2



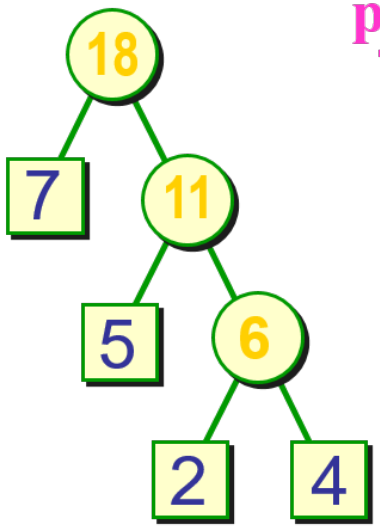
过 程 2

	Weight	parent	leftChild	rightChild
0	7	−1	−1	−1
p1 → 1	5	5 ~~−1~~	−1	−1
2	2	4	−1	−1
3	4	4	−1	−1
p2 → 4	6	5 ~~−1~~	2	3
i → 5	11	−1	1 ~~−1~~	4 ~~−1~~
6		−1	−1	−1

哈夫曼树的构造过程

结合特定的存储结构来实现

- 利用一个结构体数组来实现，如下图所示终态



终 态

	Weight	parent	leftChild	rightChild
p1 → 0	7	6 ~~-1~~	-1	-1
1	5	5	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5	2	3
p2 → 5	11	6 ~~-1~~	1	4
i → 6	18	-1	0 ~~-1~~	5 ~~-1~~

哈夫曼树的实现

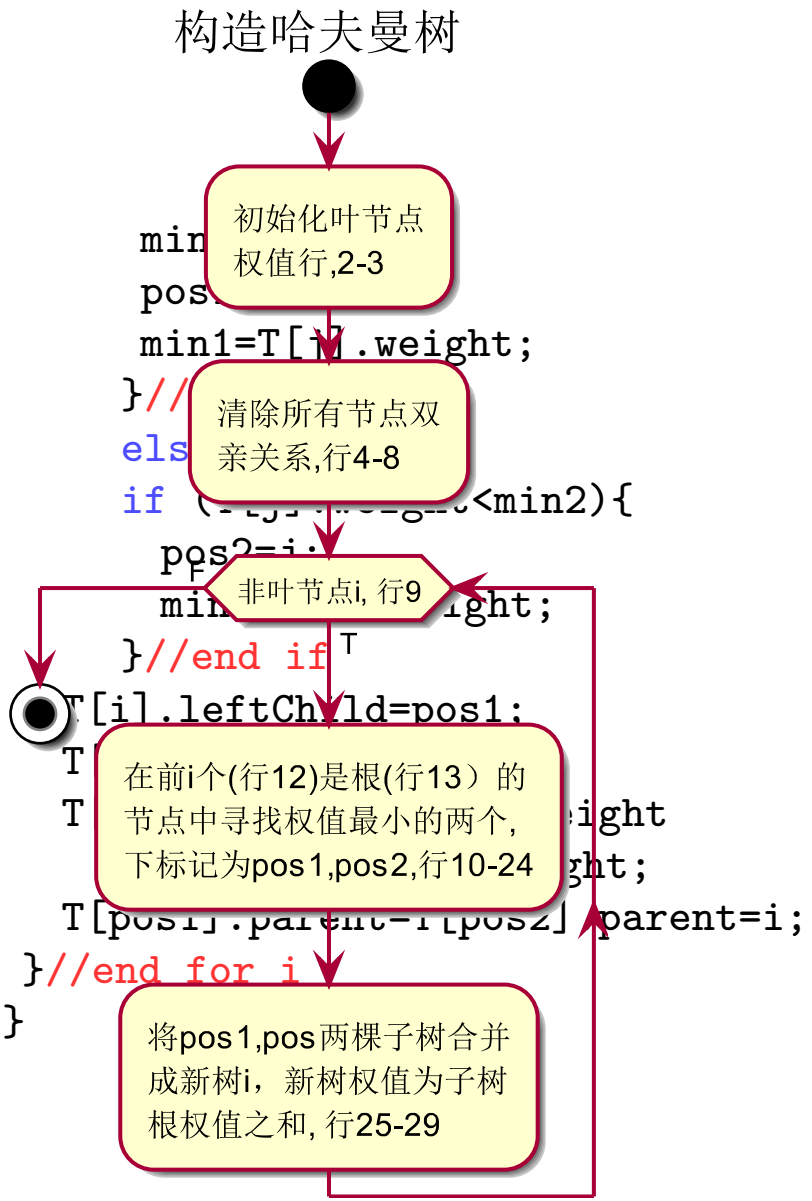
哈夫曼树在内存中的表示

```
1       const int n = 20;//叶结点数
2       const int m = 2*n -1;//结点数
3
4       typedef struct {
5           float weight;
6           int parent, leftChild, rightChild;
7       } HTNode;
8
9       typedef HTNode HuffmanTree[m];
```

哈夫曼树的实现

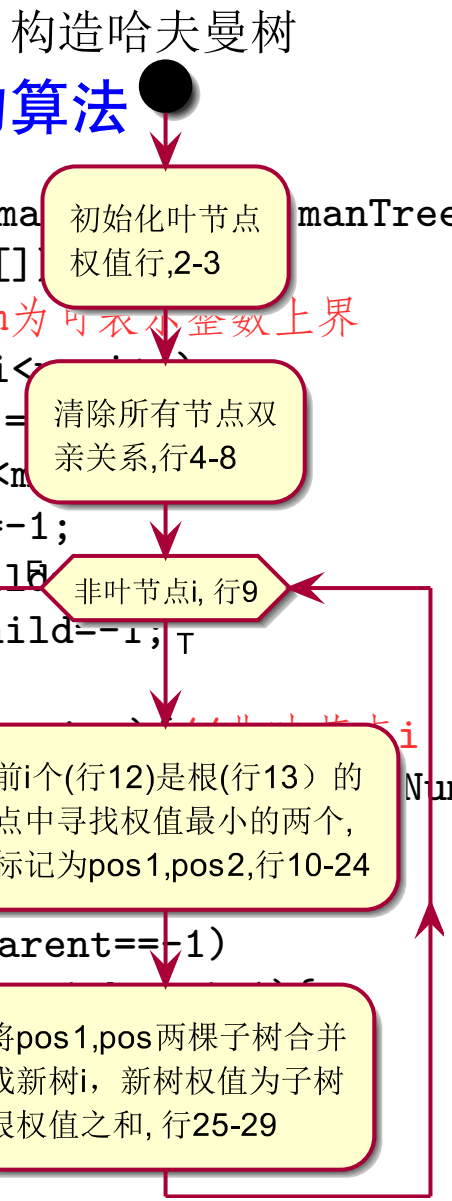
构造哈夫曼树的算法

```
1  void CreateHuffmanTree( HuffmanTree
       T,float fr[])
       {//设MaxNum为可表示整数上界
2    for(int i=0; i<n; i++)
3      T[i].weight = fr[i];
4    for(int i=0;i<m;i++){
5      T[i].parent=-1;
6      T[i].leftChild=-1;
7      T[i].rightChild=-1;
8    }
9    for(int i=n;i<m;i++){//非叶节点i
10     float min1,min2;min1=min2=MaxNum;
11     int pos1=pos2=0;
12     for(int j=0; j<i;j++)
13       if (T[j].parent==-1)
14         if (T[j].weight<min1){
15           pos2=pos1;
16           min
17           pos
18           min1=T[j].weight;
19         }//
20         els
21         if (T[j].weight<min2){
22           pos2=j;
23           min
24         }//end if
25     T[i].leftChild=pos1;
26     T
27     T
28
29     T[pos1].parent=T[pos2].parent=i;
30   }//end for i
31 }
```



哈夫曼树的实现

构造哈夫曼树的算法

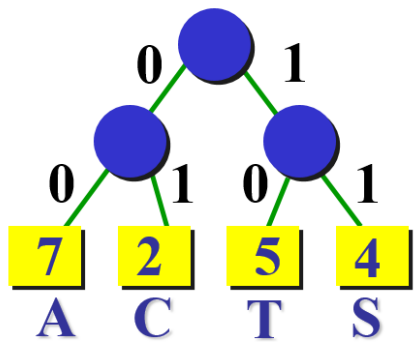



```
1  void CreateHuffma          manTree
       T,float fr[]
       {//设MaxNum为
2    for(int i=0; i<
3      T[i].weight =
4    for(int i=0;i<m
5      T[i].parent=-1;
6      T[i].leftChild
7      T[i].rightChild=-1;
8    }
9    for(int
10     float min                Num;
11     int pos1=
12     for(int j
13       if (T[j].parent==-1)
14         if (T[j
15           pos2=
16           min2=min1;
17           pos1=j;
18           min1=T[j].weight;
19         }//end if
20         else
21         if (T[j].weight<min2){
22           pos2=j;
23           min2=T[j].weight;
24         }//end if
25     T[i].leftChild=pos1;
26     T[i].rightChild=pos2;
27     T[i].weight=T[pos1].weight
28                 +T[pos2].weight;
29     T[pos1].parent=T[pos2].parent=i;
30   }//end for i
31 }
```

哈夫曼树的应用

数据压缩编码问题及分析

- 设给出一个字符串：CASTCASTSATATATASA
- 共有四个字母 $\{C, A, S, T\}$，每个字母出现的频度为：$\{2, 7, 4, 5\}$
- 在计算机内，每个字符用 8bits 表示，共计 18*8bits
- 若采用等长编码来压缩，比如 $A:00, C:01, T:10, S:11$，则总的编码长度为 $(2+7+4+5)*2=36$

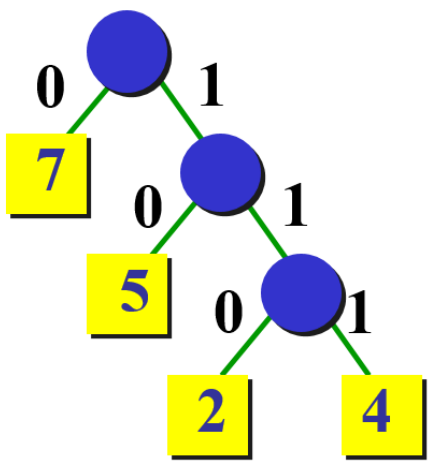


等长编码树

哈夫曼树的应用

哈夫曼编码

- 若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度
- 各字符出现的频度为 $\{2, 7, 4, 5\}$，转换为概率就是 $\{2/18, 7/18, 4/18, 5/18\}$; 以它们为各叶结点上的权值, 建立霍夫曼树
- 左分支赋 0，右分支赋 1，得霍夫曼编码 (变长编码),$A : 0, T : 10, C : 110, S : 111$, 总的编码长度为 $7 * 1 + 5 * 2 + (2 + 4) * 3 = 35$
- 总编码长度正好等于霍夫曼树的带权路径长度
- 霍夫曼编码是一种无前缀编码, 解码时不会混淆



霍夫曼编码树

练习题及课后扩展阅读

练习题

- 已知一段报文共包含 10 个字符，每个字符出现的次数如下：
- $A:29, B:13, C:22, D:33, E:54, F:10, G:13, H:23, I:36, J:38$
- 画出对应哈夫曼树。
- 要用哈夫曼编码对这段报文进行压缩，压缩比是多少？假设每个字符原来用 1 个字节表示。

扩展阅读

- 思考：解码器如何实现，网上查阅相关资料验证自己的想法，能证明解码时不会混淆吗？

哈夫曼树的应用

最佳判定树

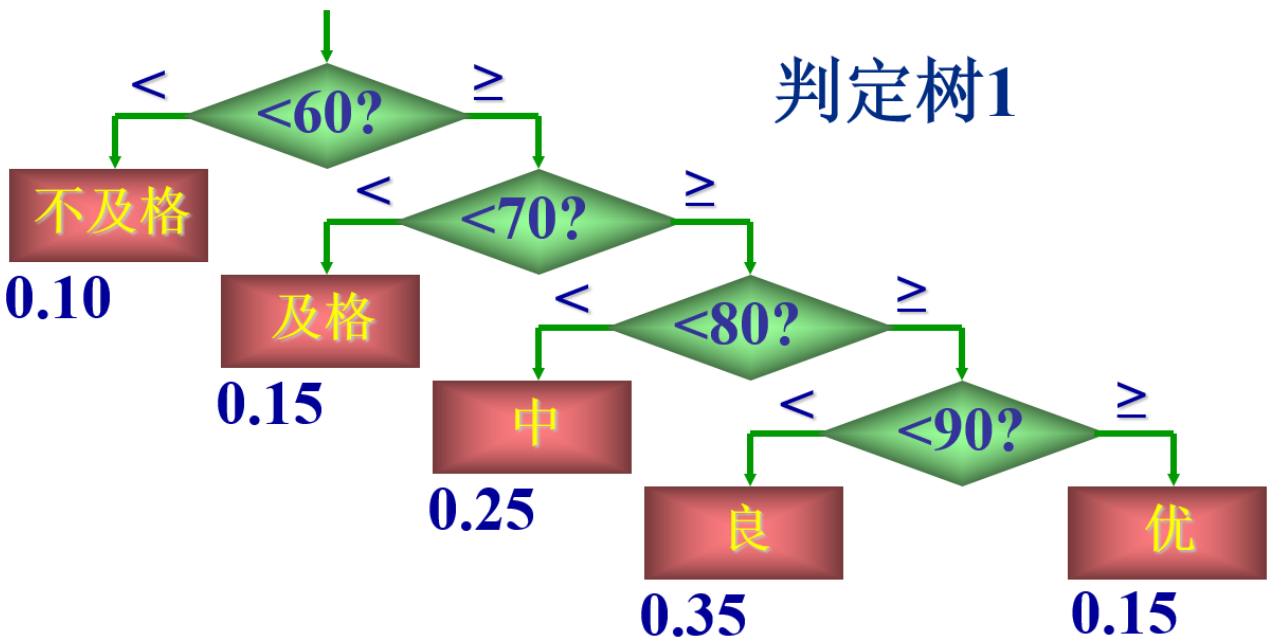
- 大量的判定过程，如何安排先判定哪一个？比如把 10000 万个 100 分制的成绩换算成“优、良、中、及格、不及格”五个等级，如图所示
- 全部用 if 语句来实现判定和转换；
- 如何安排次序？即按照“优、良、中、及格、不及格”的次序来判定，还是“良、中、优、及格、不及格”或其它的次序来判定？
- 最终 if 语句的调用次数影响运行时间
- 图中给出了 10000 万个成绩中每个等级成绩的比例，用于帮助决定采用哪种次序写程序中的 if 语句

[0, 60)	[60, 70)	[70, 80)	[80, 90)	[90, 100)
不及格	及格	中	良	优
0.10	0.15	0.25	0.35	0.15

最佳判定树

如图所示的判定树 1，其过程执行过程的伪代码如下

```
1  if (input<60) print(``不及格'')
2  else if (input<70) print(``及格'')
3       else if (input<80) print(``中'')
4            else if (input<90) print(``良'')
5                 else print(``优'')
6  //判定次数的期望 =0.10 * 1 + 0.15 * 2 + 0.25 * 3 + 0.35 * 4 + 0.15 * 4 = 3.15
```



最佳判定树

如图所示的最佳判定树，其过程执行过程的伪代码如下

```
1  if (input>=80)
2    if (input<90) print(``良'')
3    else print(``优'')
4  else if (input>=70) print(``中'')
5         else if (input<60) print(``不及格'')
6              else print(``及格'')
7  //判定次数的期望 =0.10 * 3 + 0.15 * 3 + 0.25 * 2 + 0.35 * 2 + 0.15 * 2 = 2.25
```