练习

- 1栈和链表是两种不同的数据结构。

  错，栈是逻辑结构的概念，是特殊殊线性表，而链表是存储结构概念，二者不是同类项。

- 具有n个结点的完全二叉树有?个度为2的结点

- 两个栈共享一片连续内存空间时，为提高内存利用率，减少溢出机会，应把两个栈的栈底分别设在这片内存空间的两端。

  对

练习

试将下列递推过程改写为递归过程。

```
void ditui(int n)
{       i=n;
        while(i>1) printf(i--);
}
```

数据结构

第五章
树与二叉树

第五章 树与二叉树

- 树和森林的概念
- 二叉树
- 二叉树遍历
- 二叉树的计数
- 树与森林
- **Huffman树**

5.1 树的概念

5.1.1 树的定义与术语

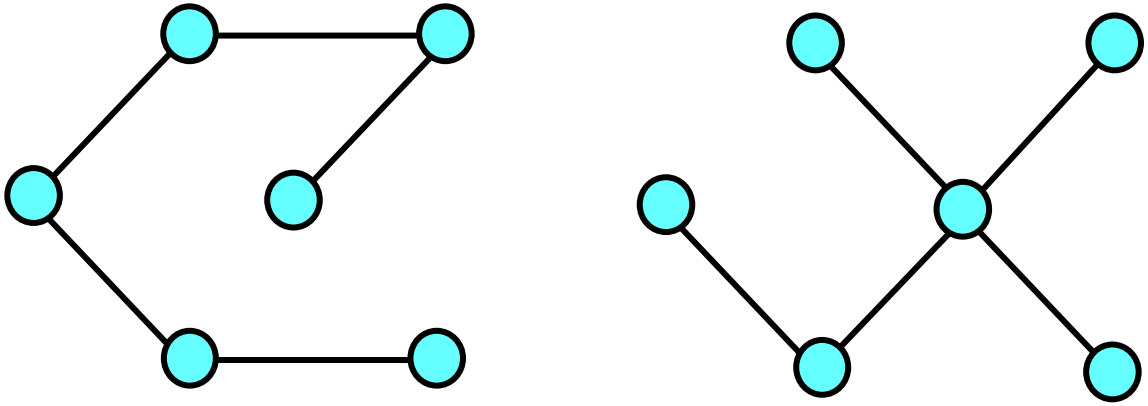
- 两种树：自由树与有根树。
- 自由树：

一棵自由树 T_f 可定义为一个二元组

$$T_f = (V, E)$$

$V = \{v_1, ..., v_n\}$ 是由 n $(n>0)$ 个元素组成的有限非空集合，称为顶点集合。

$E = \{(v_i, v_j) \mid v_i, v_j \in V, 1 \le i, j \le n\}$ 是$n-1$个序对的集合，称为边集合，E 中的元素 (v_i, v_j) 称为边或分支。



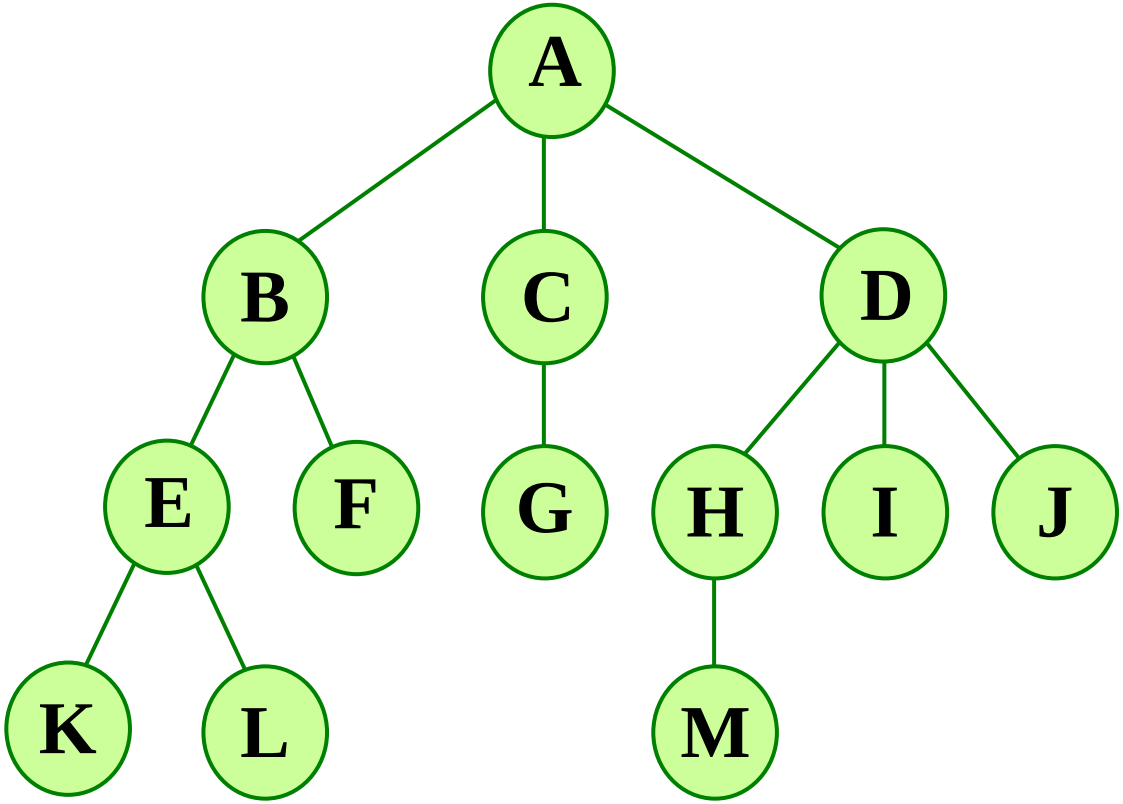
自由树

自由树与有根树

有根树：

一棵有根树 T，简称为树，它是n ($n \geq 0$) 个结点的有限集合。当$n = 0$时，T 称为空树；否则，T 是非空树，记作

$$T = \begin{cases} \Phi, & n = 0 \\ \{r, T_1, T_2, \ldots, T_m\}, & n > 0 \end{cases}$$


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

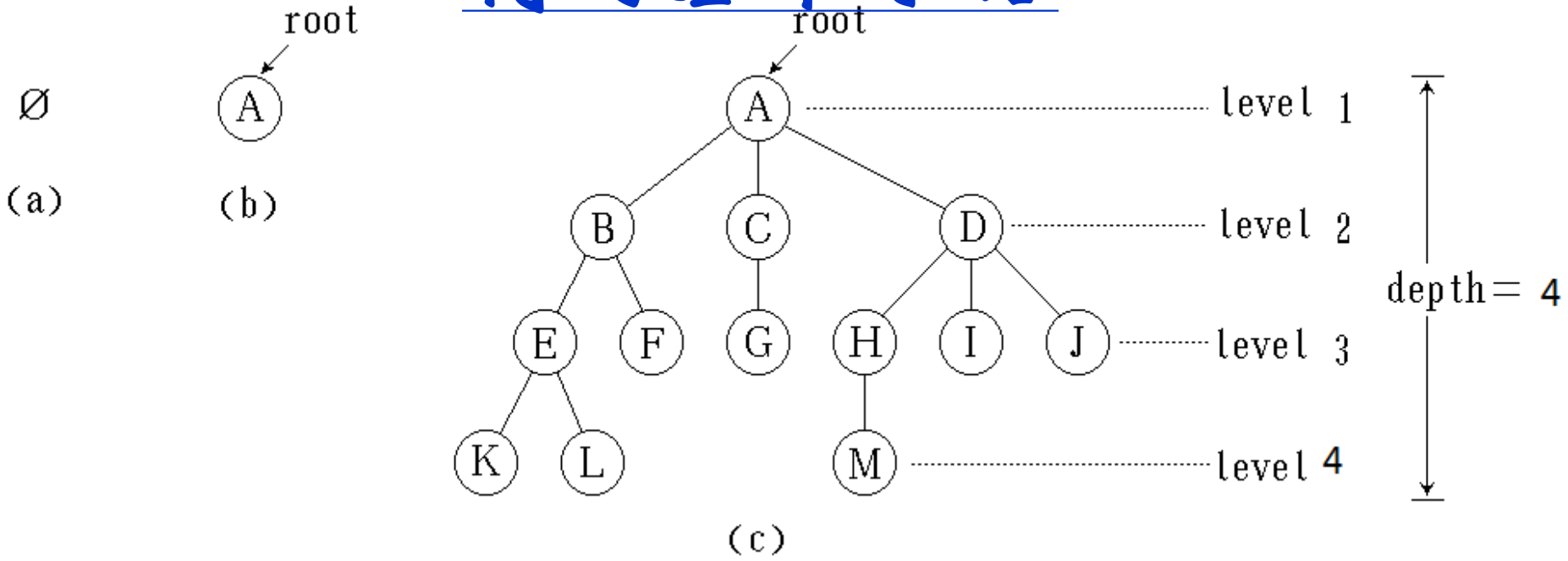
有根树的特征

- r 是一个特定的称为根(root)的结点，它只有直接后继，但没有直接前驱；
- 根以外的其他结点划分为 m ($m \geq 0$) 个互不相交的有限集合$T_1, T_2, \ldots, T_m$，每个集合又是一棵树，并且称之为根的子树。

注：树的定义具有递归性，即树中还有树。

- 每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继。

树的基本术语

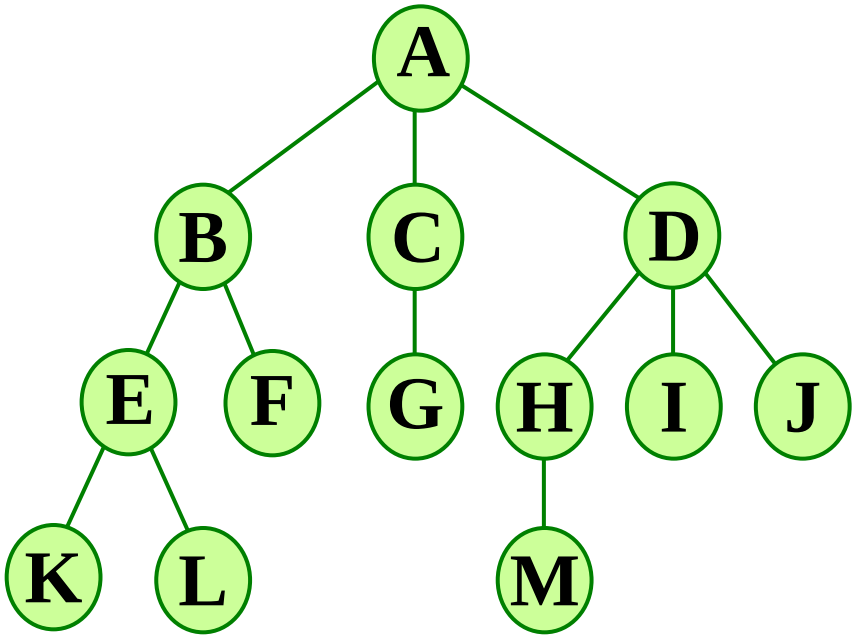


- 结点(node)
- 结点的度(degree)
- 分支(branch)结点
- 叶(leaf)结点
- 子女(child)结点
- 双亲(parent)结点
- 兄弟(sibling)结点
- 祖先(ancestor)结点
- 子孙(descendant)结点
- 结点所处层次(level)
- 树的高度(depth)
- 树的度(degree)

- 有序树
- 无序树
- 森林

树的基本术语

- *子女*：若结点的子树非空，结点子树的根即为该结点的子女。
- *双亲*：若结点有子女，该结点是子女双亲。



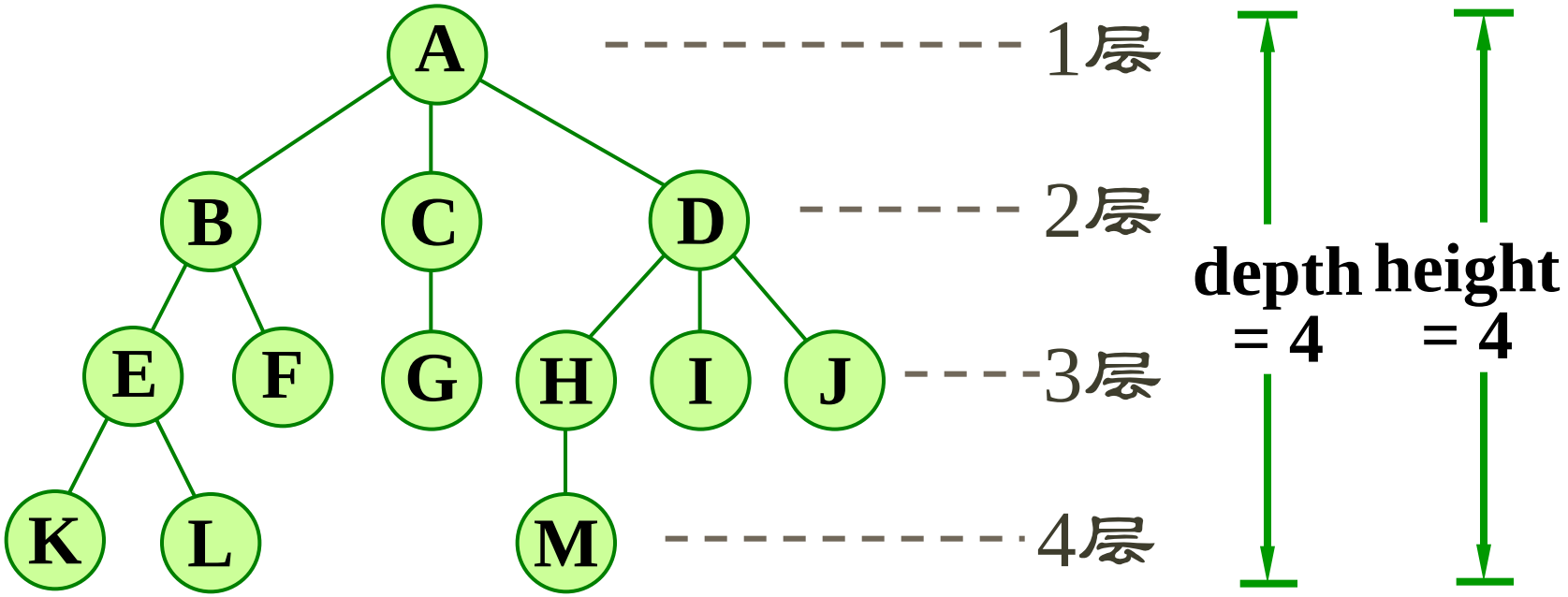
度：结点的子女个数即为该结点的度；树中各个结点的度的最大值称为树的度。

树的基本术语

- *兄弟*：同一结点的子女互称为兄弟。
- *分支结点*：度不为0的结点即为分支结点，亦称为非终端结点。
- *叶结点*：度为0的结点即为叶结点，亦称为终端结点。
- *祖先*：某结点到根结点的路径上的各个结点都是该结点的祖先。
- *子孙*：某结点的所有下属结点，都是该结点的子孙。

树的基本术语

- *结点的层次*：规定根结点在第一层，其子女结点的层次等于它的层次加一。以下类推。
- *深度：*结点的深度即为结点的层次；离根最远结点的层次即为树的深度。



树的基本术语

- *高度*：规定叶结点的高度为1，其双亲结点的高度等于它的高度加一。
- *树的高度*：等于根结点的高度，即根结点所有子女高度的最大值加一。
- *有序树*：树中结点的各棵子树 $T_0, T_1, \ldots$是有次序的，即为有序树。
- *无序树*：树中结点的各棵子树之间的次序是不重要的，可以互相交换位置。
- *森林*：森林是m（$m \geq 0$）棵树的集合。

5.1.2 树的抽象数据类型

```
template <class T>
class Tree {
//对象: 树是由n (≥0) 个结点组成的有限集合。在
//类界面中的 position 是树中结点的地址。在顺序
//存储方式下是下标型, 在链表存储方式下是指针
//型。T 是树结点中存放数据的类型, 要求所有结
//点的数据类型都是一致的。
public:
    Tree ();
    ~Tree ();
```

5.1.2 树的抽象数据类型

BuildRoot (**const** T& value);

//建立树的根结点

position FirstChild(position p);

//返回 p 第一个子女地址, 无子女返回 0

position NextSibling(position p);

//返回 p 下一兄弟地址, 若无下一兄弟返回 0

position Parent(position p);

//返回 p 双亲结点地址, 若 p 为根返回 0

T getData(position p);

//返回结点 p 中存放的值

bool InsertChild(position p, T& value);

//在结点 p 下插入值为 value 的新子女, 若插

//入失败, 函数返回false, 否则返回true

5.1.2 树的抽象数据类型

```
    bool DeleteChild (position p, int i);
      //删除结点 p 的第 i 个子女及其全部子孙结
      //点, 若删除失败, 则返回false, 否则返回true
    void DeleteSubTree (position t);
                                  //删除以 t 为根结点的子树
    bool IsEmpty ();
          //判树空否, 若空则返回true, 否则返回false
    void Traversal (void (*visit)(position p));
                                  //遍历以 p 为根的子树
};
```

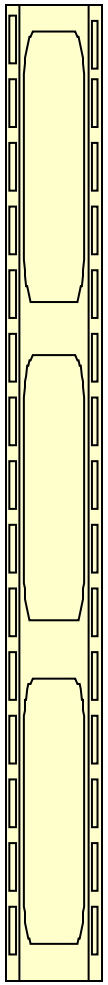
树的逻辑结构

(特点)： 一对多(1:n)，有多个直接后继(如家谱树、目录树等等)，但只有一个根结点，且子树之间互不相交。

讨论1：树和线性结构的比较

线性结构

- 第一个数据元素
  （无前驱）
- 最后一个数据元素
  （无后继）
- 其它数据元素
  （一个前驱，
  一个后继）



树结构

- 根结点
  （无前驱）
- 多个叶子结点
  （无后继）
- 树中其它结点
  （一个前驱，
  多个后继）

树的存储结构

1. 顺序存储方案
2. 链式存储方案

讨论2：树的顺序存储方案应该怎样制定？

可规定为：从上至下、从左至右将树的结点依次存入内存。

重大缺陷：复原困难（不能唯一复原就没有实用价值）。

讨论3：树的链式存储方案应该怎样制定？

可用多重链表：一个前趋指针，n个后继指针。

细节问题：树中结点的结构类型样式该如何设计？

即应该设计成“等长”还是“不等长”？

缺点：等长结构太浪费（每个结点的度不一定相同）；

不等长结构太复杂（要定义好多种结构类型）。

解决思路：先研究最简单、最有规律的树，然后设法把一般的树转化为简单树。

二叉树

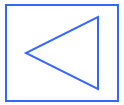
5.2 二叉树

为何要重点研究每结点最多只有两个“叉”的树?

- ✓ 二叉树的结构最简单，规律性最强；
- ✓ 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。

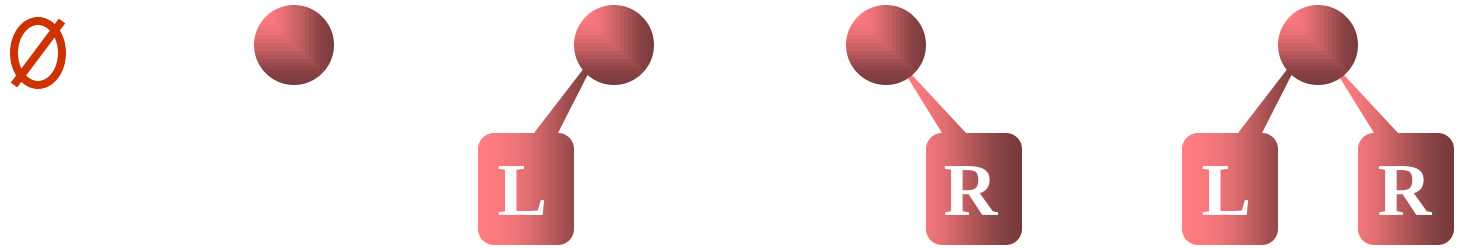
1. 二叉树的定义
2. 二叉树的性质
3. 二叉树的存储结构

（二叉树的运算）



5.2.1 二叉树的定义

一棵二叉树是结点的一个有限集合，该集合或者①为空，或者②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。



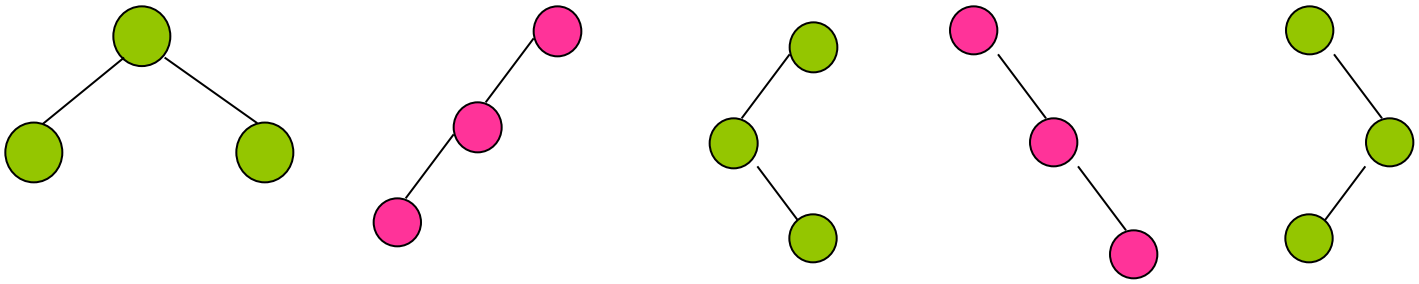
二叉树的五种不同形态

5.2.1 二叉树的定义

- 逻辑结构： 一对二（1：2）
- 基本特征：
- ① 每个结点最多只有两棵子树（不存在度大于2的结点）；
- ② 左子树和右子树次序不能颠倒（有序树）。

问：具有3个结点的二叉树可能有几种不同形态？

5种



5.2.2 二叉树的性质

- 性质1　若二叉树结点的层次从 1 开始, 则在二叉树的第 i 层最多有 2^{i-1} 个结点。($i \geqslant 1$)

  [证明用数学归纳法]

- 性质2　深度为 k 的二叉树最少有 k 个结点，最多有 2^k-1个结点。($k \geqslant 1$)

  因为每一层最少要有1个结点，因此，最少结点数为 k。最多结点个数借助性质1：用求等比级数前k项和的公式

  $$2^0+2^1+2^2+\ldots+2^{k-1}=2^k-1$$

5.2.2 二叉树的性质

- 性质3　对任何一棵二叉树，如果其叶结点有 n_0 个，度为 2 的非叶结点有 n_2 个，则有

$$n_0 = n_2 + 1$$

若设度为 1 的结点有 n_1 个，总结点数为n，总边数为e，则根据二叉树的定义，

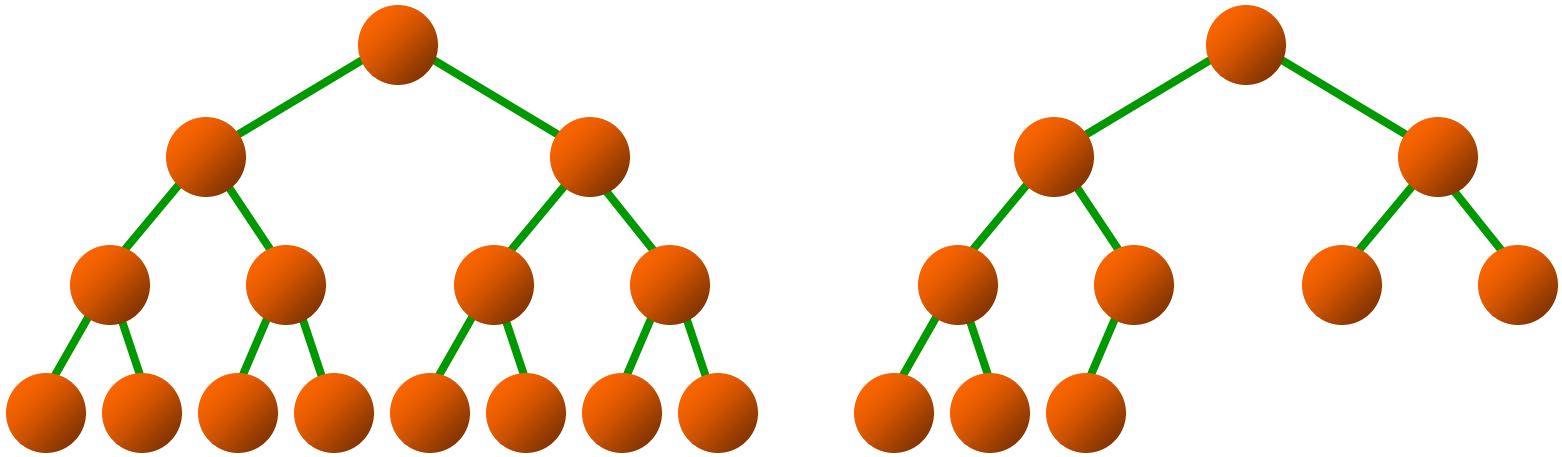
$$n = n_0 + n_1 + n_2 \qquad e = 2n_2 + n_1 = n - 1$$

因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

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

特殊二叉树的定义

- 定义1 满二叉树 (Full Binary Tree)：每一层结点都达到了最大个数的二叉树。深度为k的满二叉树有2^k-1个。
- 定义2 完全二叉树 (Complete Binary Tree)
  - 若设二叉树的深度为 k，则共有 k 层。除第 k 层外，其它各层 (1—k-1) 的结点数都达到最大个数，第k层从右向左连续缺若干结点，这就是完全二叉树。



5.2.2 二叉树的性质

- 性质4　具有 n ($n \geqslant 0$) 个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$

证明：　设完全二叉树的深度为k，则有

$$2^{k-1}-1 < n \leqslant 2^{k}-1$$

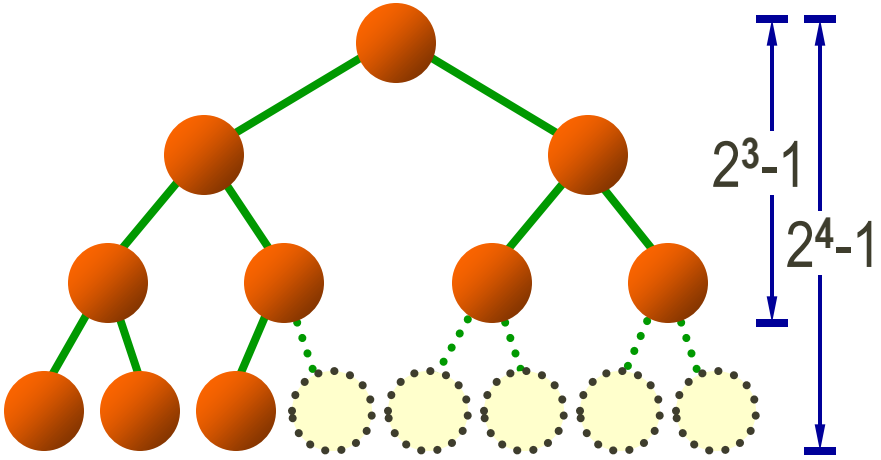
上面$k-1$层结点数　包括第k层的最大结点数

变形 $2^{k-1} < n+1 \leqslant 2^{k}$

取对数

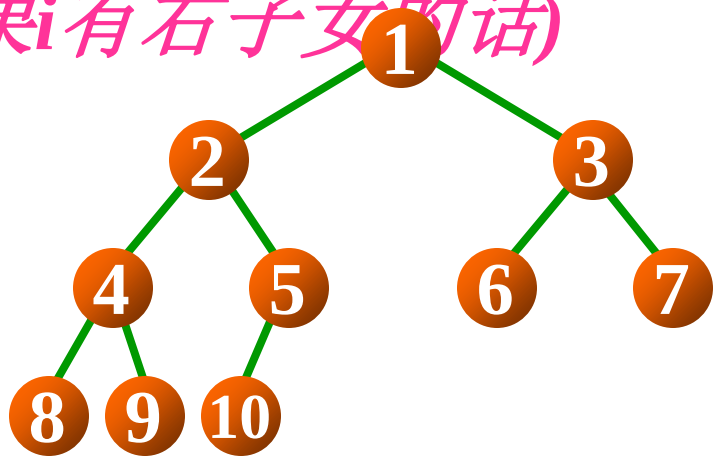
$$k-1 < \log_2(n+1) \leqslant k$$

有　$\lceil \log_2(n+1) \rceil = k$



5.2.2 二叉树的性质

- 性质5 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, …, n，则有以下关系：
  - ✓ 若$i = 1$, 则 i 无双亲
  - ✓ 若$i > 1$, 则 i 的双亲为$\lfloor i / 2 \rfloor$
  - ✓ i 的左子女为 $2*i$(*如果i有左子女的话*)，
    i 的右子女为$2*i+1$ (*如果i有右子女的话*)
  - ✓ 若 i 为奇数, 且i != 1,
    则其左兄弟为$i-1$,
  - ✓ 若 i 为偶数, 且i != n,
    则其右兄弟为$i+1$



课堂讨论：

①：满二叉树和完全二叉树有什么区别？

答：满二叉树是叶子一个也不少的树，而完全二叉树虽然前n-1层是满的，但最底层却允许在右边缺少连续若干个结点。满二叉树是完全二叉树的一个特例。

课堂练习：

1. 树T中各结点的度的最大值称为树T的_____。

A）高度　　B）层次　C）深度　　D）度 √

2. 深度为k的二叉树的结点总数，最多为（）个。

A）2^{k-1}　B）$\log_2 k$　C）2^k-1 √　D）2^k

3. 深度为9的二叉树中至少有_____个结点。

A）2^9　　B）2^8　　C）9 √　　D）2^9-1

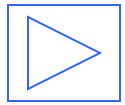
练习432

- 若二叉树用二叉链表作存贮结构，则在n个结点的二叉树链表中只有____个非空指针域
- N-1
- 具有n个结点的完全二叉树有?个度为2的结点
- ⌊n-1 / 2⌋或（n-1/2 n/2-1）
- 一棵含有n个结点的k叉树，可能达到的最大深度为__，最小深度为_。
- N,2
- 一棵度为2的树是一棵二叉树?
- 先序序列和中序序列相同是什么样的二叉树?

5.2.3 二叉树的抽象数据类型

基本操作

- 第一类，查询：寻找满足某种特定关系的结点；查询树的属性：如高度深度，节点个数，状态，特定节点，极其父子节点。
- 第二类，插入、修改或删除：如在树的当前结点上插入一个新结点或删除当前结点的孩子结点等。
- 第三类，上述操作必须建立在**对树结点能够“遍历”**的基础上，遍历树中每个结点，这里着重介绍。
  - **遍历**：指每个结点都被访问且仅访问一次，不遗漏不重复



5.2.3 二叉树的抽象数据类型p3-1

```
template <class T>
class BinaryTree {
//对象: 结点的有限集合, 二叉树是有序树
public:
    BinaryTree ();                          //构造函数
    BinaryTree (BinTreeNode<T> *lch,
                BinTreeNode<T> *rch, T item);
      //构造函数, 以item为根, lch和rch为左、右子
       //树构造一棵二叉树
    int Height ();                          //求树高度
    int Size ();                            //求树中结点个数
```

5.2.3 二叉树的抽象数据类型p3-2

```
bool IsEmpty ();                          //判二叉树空否？
BinTreeNode<T> *Parent (BinTreeNode<T> *t);
                                          //求结点 t 的双亲
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t);
                                          //求结点 t 的左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t);
                                          //求结点 t 的右子女
bool Find (T& item);                      //判断item是否在树中
bool getData (T& item);                   //取得结点数据
bool Insert (T item);                     //在树中插入新元素
bool Remove (T item);//在树中删除元素
```

5.2.3 二叉树的抽象数据类型p3-3

```
BinTreeNode<T> *getRoot ();          //取根
void preOrder (void (*visit) (BinTreeNode<T> *t));
        //前序遍历, visit是访问函数
void inOrder (void (*visit) (BinTreeNode<T> *t));
                                    //中序遍历, visit是访问函数
  void postOrder (void (*visit) (BinTreeNode<T> *t));
                                    //后序遍历, (*visit)是访问函数
  void levelOrder (void (*visit)(BinTreeNode<T> *t));
                                    //层次序遍历, visit是访问函数
};
```

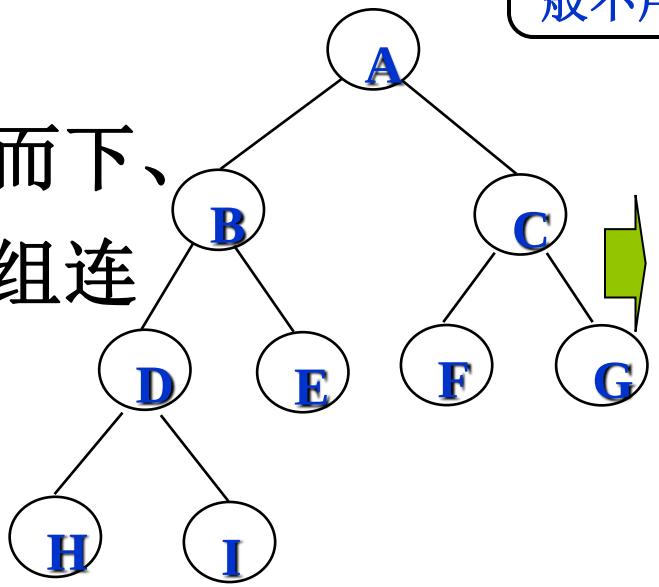
5.3 二叉树的存储表示

5.3.1 二叉树的顺序表示

- **顺序存储结构**：用一组地址连续的存储单元存储二叉树中的数据元素。

一、顺序存储结构

按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。



T[0]一般不用

[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

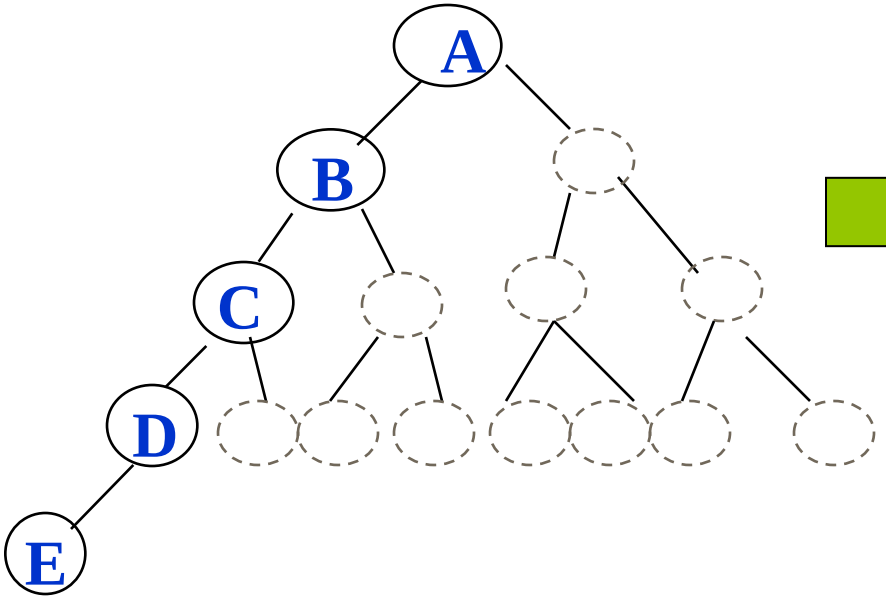
问：顺序存储后能否复原成唯一对应的二叉树形状？

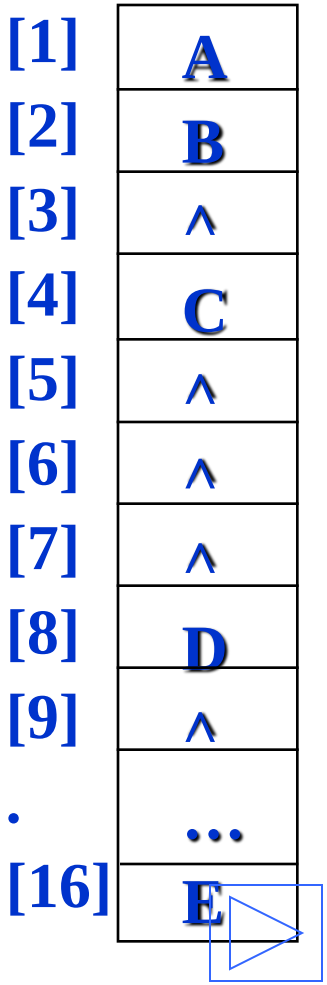
答：若是完全/满二叉树则可以做到唯一复原。

因为根据性质5：可知结点i，左孩子的下标值必为2i，其右孩子的下标值必为2i＋1（即）

讨论：非完全二叉树怎么办？

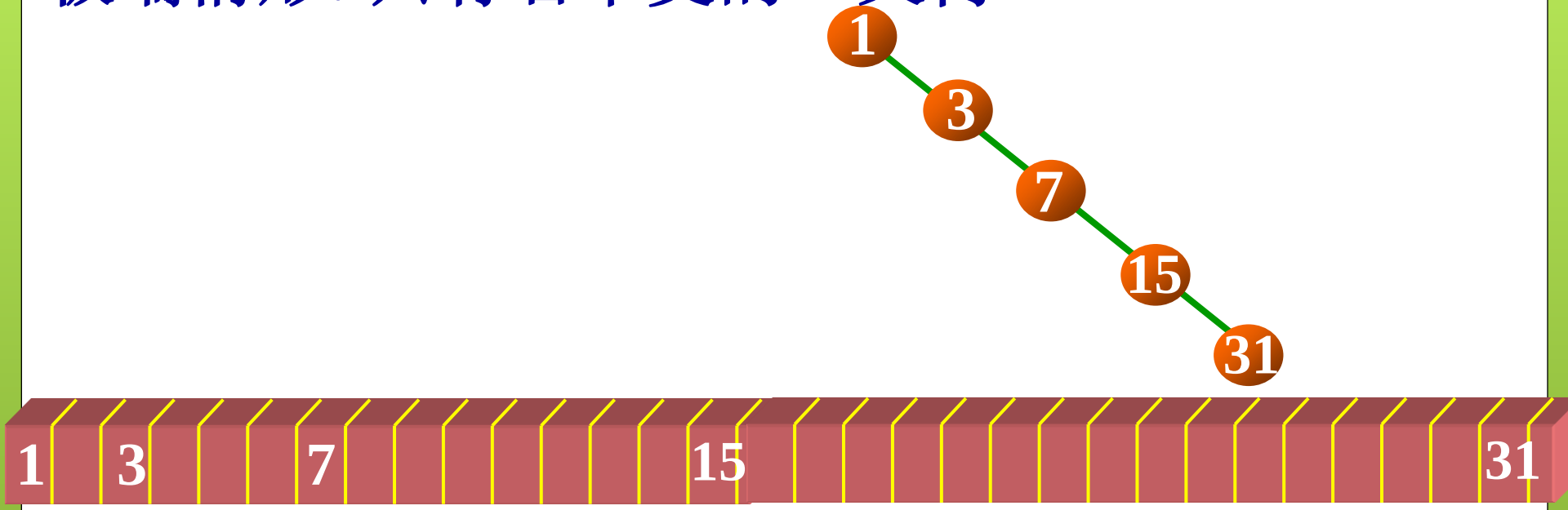
答：将各层空缺处统统补上“虚结点”，其内容为空。





5.3.1 二叉树的顺序表示

极端情形: 只有右单支的二叉树



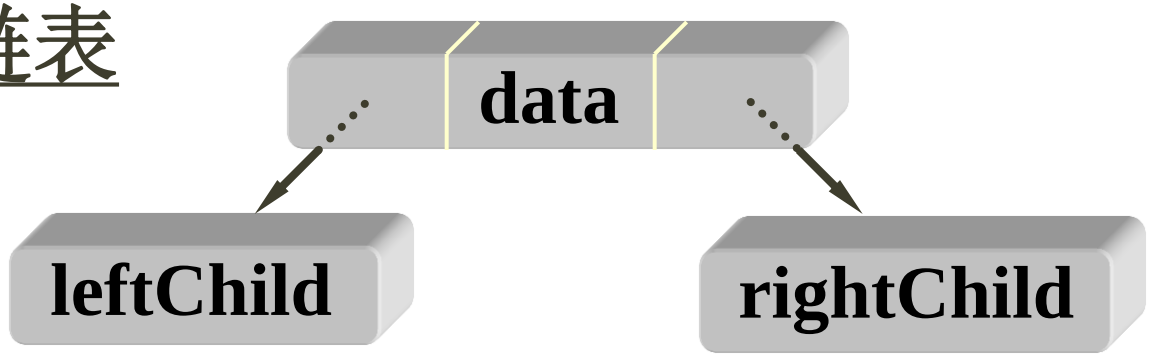
缺点：①浪费空间；②插入、删除不便

结论： 非完全二叉树不适合进行顺序存储

5.3.2 二叉树的链表表示（二叉链表）

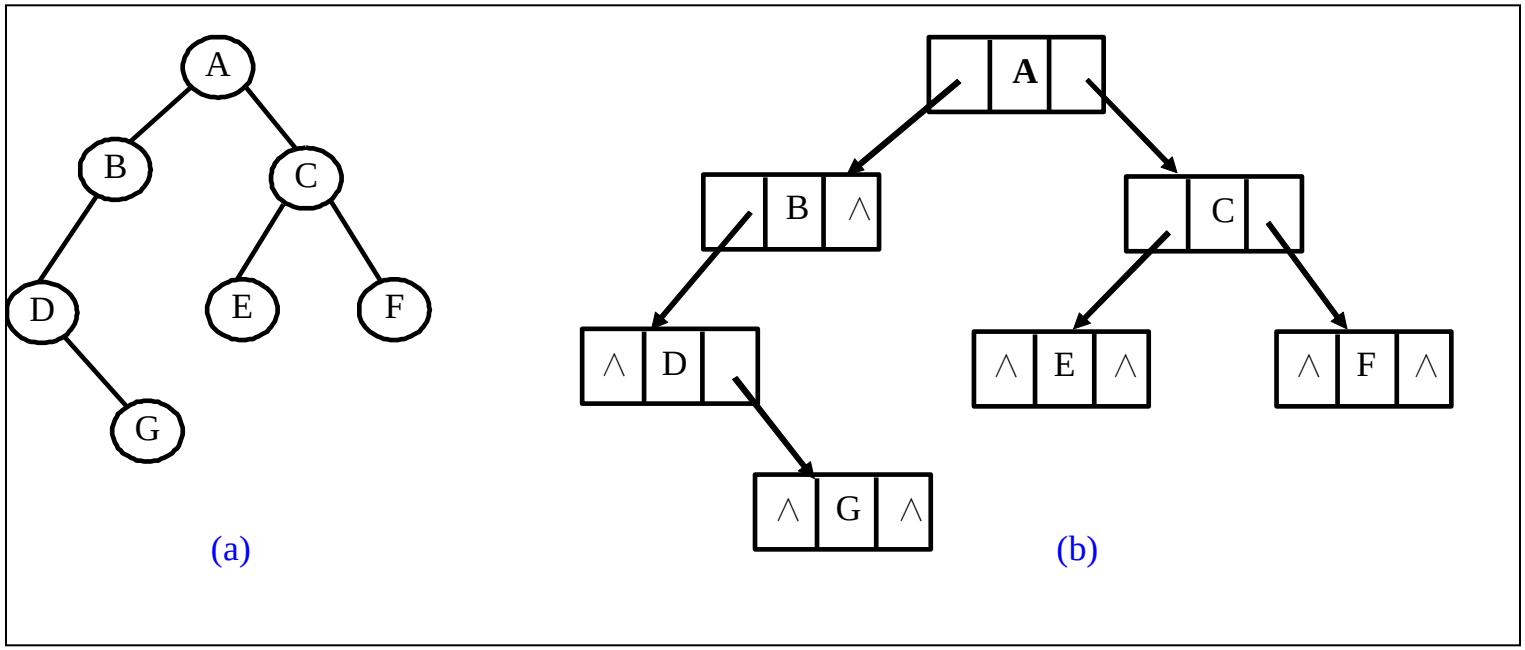
- 二叉树结点定义：每个结点有3个数据成员，data域存储结点数据，leftChild和rightChild分别存放指向左子女和右子女的指针。

二叉链表



二叉链表

一般从根结点开始存储。（相应地，访问树中结点时也只能从根开始）

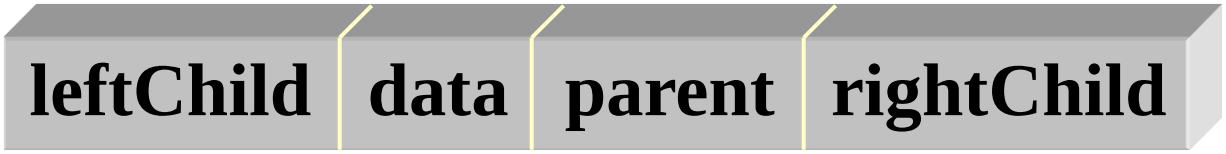


含n个结点的二叉链表其中空链域为n+1个　　即

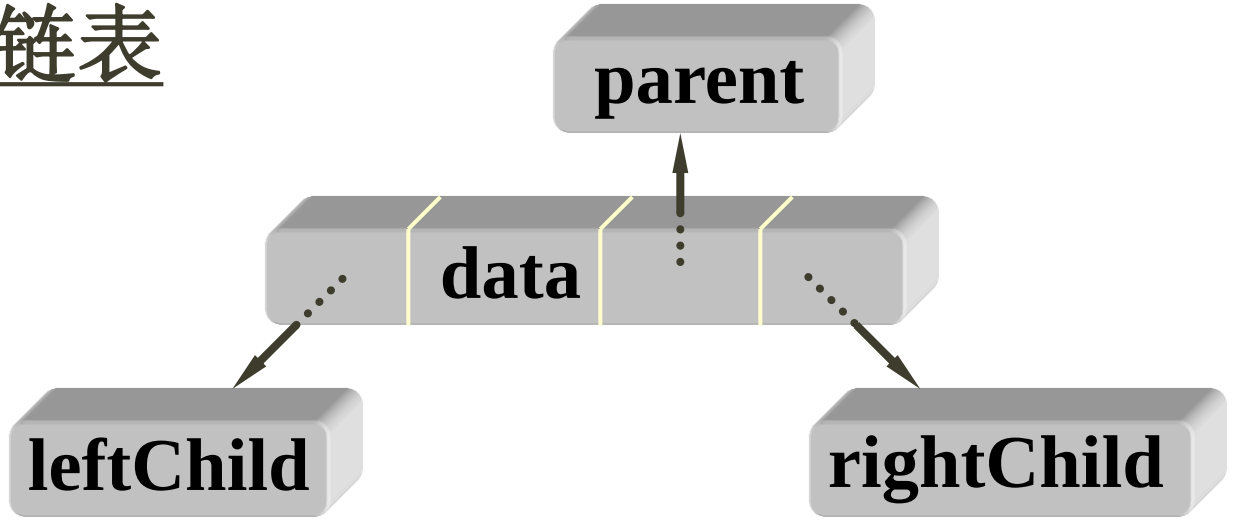
$2n_0+n_1=n_0+n_2+1+n_1=n+1$

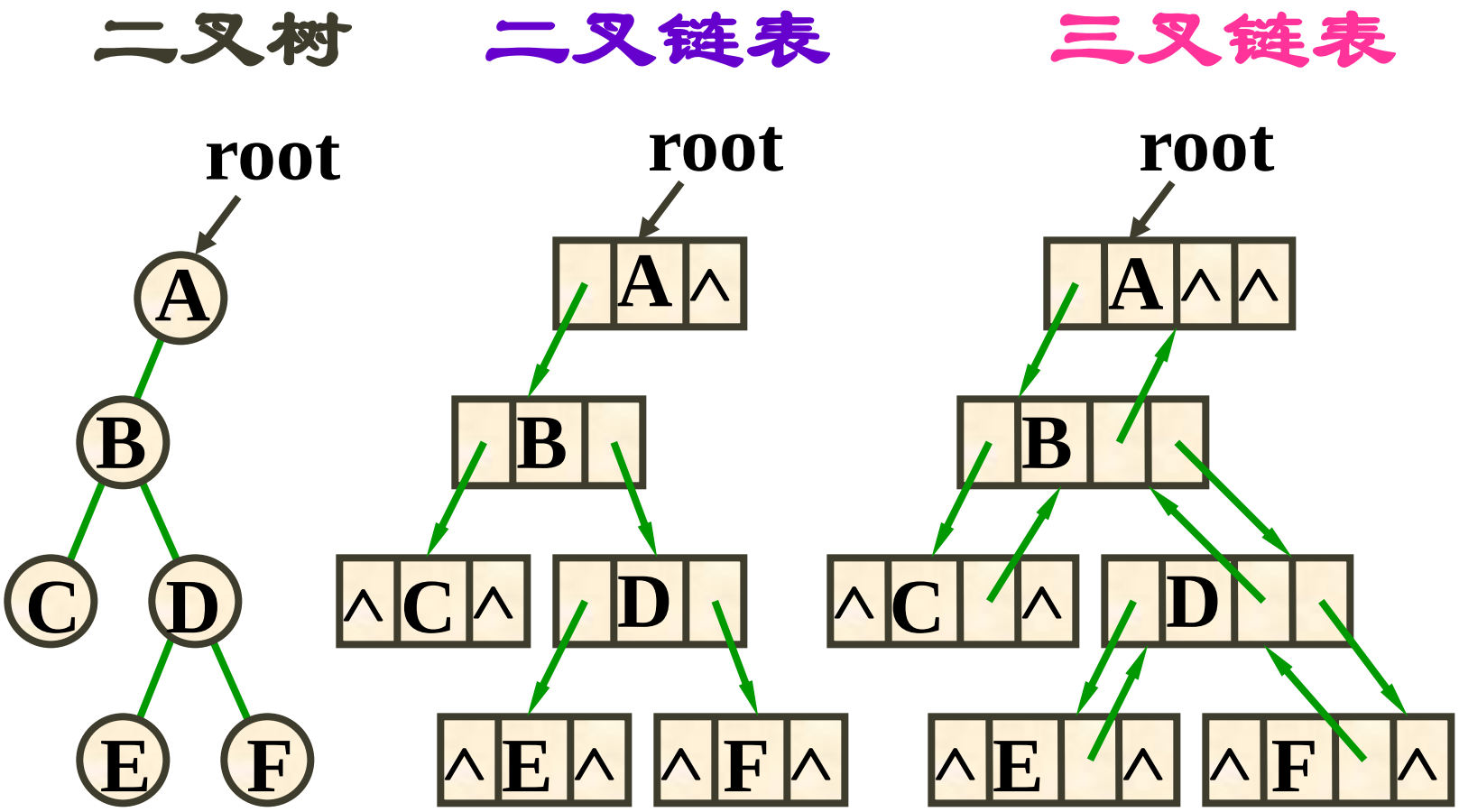
5.3.2 二叉树的链表表示（三叉链表）

- 每个结点增加一个指向双亲的指针parent，使得查找双亲也很方便。

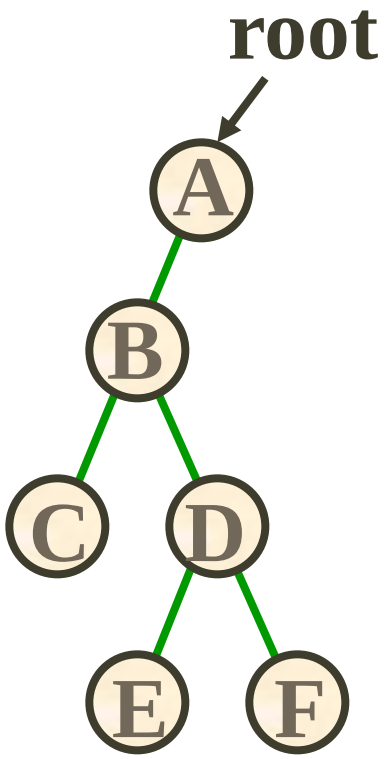


三叉链表





二叉树链表表示的示例



	data	parent	leftChild	rightChild
0	A	−1	1	−1
1	B	0	2	3
2	C	1	−1	−1
3	D	1	4	5
4	E	3	−1	−1
5	F	3	−1	−1

链表的静态结构

二叉树的C语言实现

二叉树结点的数据类型定义：

```
typedef  struct BitNode{
   int                    data;
   struct BitNode  *lchild,*rchild;
} * BitTree;
```

二叉树的实现

```
BitTree  root;
```

二叉树c++的类定义p8-1

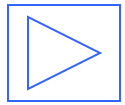
```
template <class T>
struct BinTreeNode {          //二叉树结点类定义
   T data;                    //数据域
    BinTreeNode<T> *leftChild, *rightChild;
                              //左子女、右子女链域
    BinTreeNode ()            //构造函数
      { leftChild = NULL;  rightChild = NULL; }
   BinTreeNode (T& x, BinTreeNode<T> *l = NULL,
                BinTreeNode<T> *r = NULL)
       { data = x;  leftChild = l;  rightChild = r; }
};
```

5.2.3 二叉树的抽象数据类型

基本操作

- 第一类，查询：寻找满足某种特定关系的结点；查询树的属性：如高度深度，节点个数，状态，特定节点，极其父子节点。

- 第二类，插入、修改或删除：如在树的当前结点上插入一个新结点或删除当前结点的孩子结点等。
- 第三类，上述操作必须建立在**对树结点能够“遍历”**的基础上，遍历树中每个结点，这里着重介绍。
  - **遍历**：指每个结点都被访问且仅访问一次，不遗漏不重复



二叉树的类定义p8-2

```
template <class T>
class BinaryTree {          //二叉树类定义
protected:
    BinTreeNode<T> *root;       //二叉树的根指针
    T RefValue;                 //数据输入停止标志
public:
    BinaryTree () : root (NULL) { }        //构造函数
    BinaryTree (T value) : RefValue(value), root(NULL) { }
   BinaryTree (BinaryTree<T>& s);     //复制构造函数
    ~ BinaryTree () { destroy(root); }     //析构函数
    bool IsEmpty () { return root == NULL;}//判二叉树空否
    int Height () { return Height(root); }  //求树高度
```

二叉树的类定义p8-3

```
BinTreeNode<T> *Parent (BinTreeNode <T> *t)
  { return (root == NULL || root ==  t) ?
                              NULL : Parent (root, t); }  //返回双亲结点
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t)
  { return (t != NULL) ? t->leftChild : NULL; }
                                                          //返回左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t)
  { return (t != NULL) ? t->rightChild : NULL; }
                                                          //返回右子女
 BinTreeNode<T> *getRoot () const { return root; }

                                                          //取根
```

二叉树的类定义p8-4

```
void preOrder (void (*visit) (BinTreeNode<T> *t))
  { preOrder (root, visit); }          //前序遍历
void inOrder (void (*visit) (BinTreeNode<T> *t))
  { inOrder (root, visit); }           //中序遍历
void postOrder (void (*visit) (BinTreeNode<T> *t))
  { postOrder (root, visit); }         //后序遍历
void levelOrder (void (*visit)(BinTreeNode<T> *t));
                                       //层次序遍历
int Insert (const T item);             //插入新元素
BinTreeNode<T> *Find (T item) const;   //搜索
```

二叉树的类定义p8-5

```
protected:

    void CreateBinTree (istream& in,
                        BinTreeNode<T> *& subTree);
                                        //从文件读入建树
    bool Insert (BinTreeNode<T> *& subTree,  T& x);
                                                    //插入
    void destroy (BinTreeNode<T> *& subTree);     //删除
    bool Find (BinTreeNode<T> *subTree, T& x);   // 查找
```

二叉树的类定义p8-6

```
BinTreeNode<T> *Copy (BinTreeNode<T> *r);
                                    //复制
int Height (BinTreeNode<T> *subTree);
                                    //返回树高度
int Size (BinTreeNode<T> *subTree);
                                    //返回结点数
BinTreeNode<T> *Parent (BinTreeNode<T> *
    subTree, BinTreeNode<T> *t);
                                    //返回父结点
BinTreeNode<T> *Find (BinTreeNode<T> *
    subTree, T& x) const;       //搜寻x
```

二叉树的类定义p8-7

```
void Traverse (BinTreeNode<T> *subTree, ostream& out);
//前序遍历输出
void preOrder (BinTreeNode<T>& subTree,
                                    void (*visit) (BinTreeNode<T> *t));
                                                    //前序遍历
void inOrder (BinTreeNode<T>& subTree,
                                    void (*visit) (BinTreeNode<T> *t));
                                                    //中序遍历
void postOrder (BinTreeNode<T>& subTree,
                                    void (*visit) (BinTreeNode<T> *t));
                                                    //后序遍历
```

二叉树的类定义p8-8

```
    friend istream& operator >> (istream& in,
            BinaryTree<T>& Tree);    //重载操作：输入
    friend ostream& operator << (ostream& out,
            BinaryTree<T>& Tree);    //重载操作：输出
};
```

问题的提出

- **遍历**——指按某条搜索路线遍访每个结点，使得每个结点均被**访问**一次，而且仅被访问一次（又称周游）。
- **遍历用途**——它是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。
- **遍历方法**——牢记一种约定，对每个结点的查看都是“**先左后右**”。
- **访问的**含义可以很广，**如输出结点的信息等**

- “遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径（因为每个结点均只有一个后继），故不需要另加讨论。
- 而二叉树是非线性结构，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题

- 二叉树深度遍历算法
- 基于递归遍历的基本操作
- 基于递归遍历的应用
- 二叉树的非递归遍历算法
- 层次序遍历二叉树的算法
- 基于层次遍历的应用

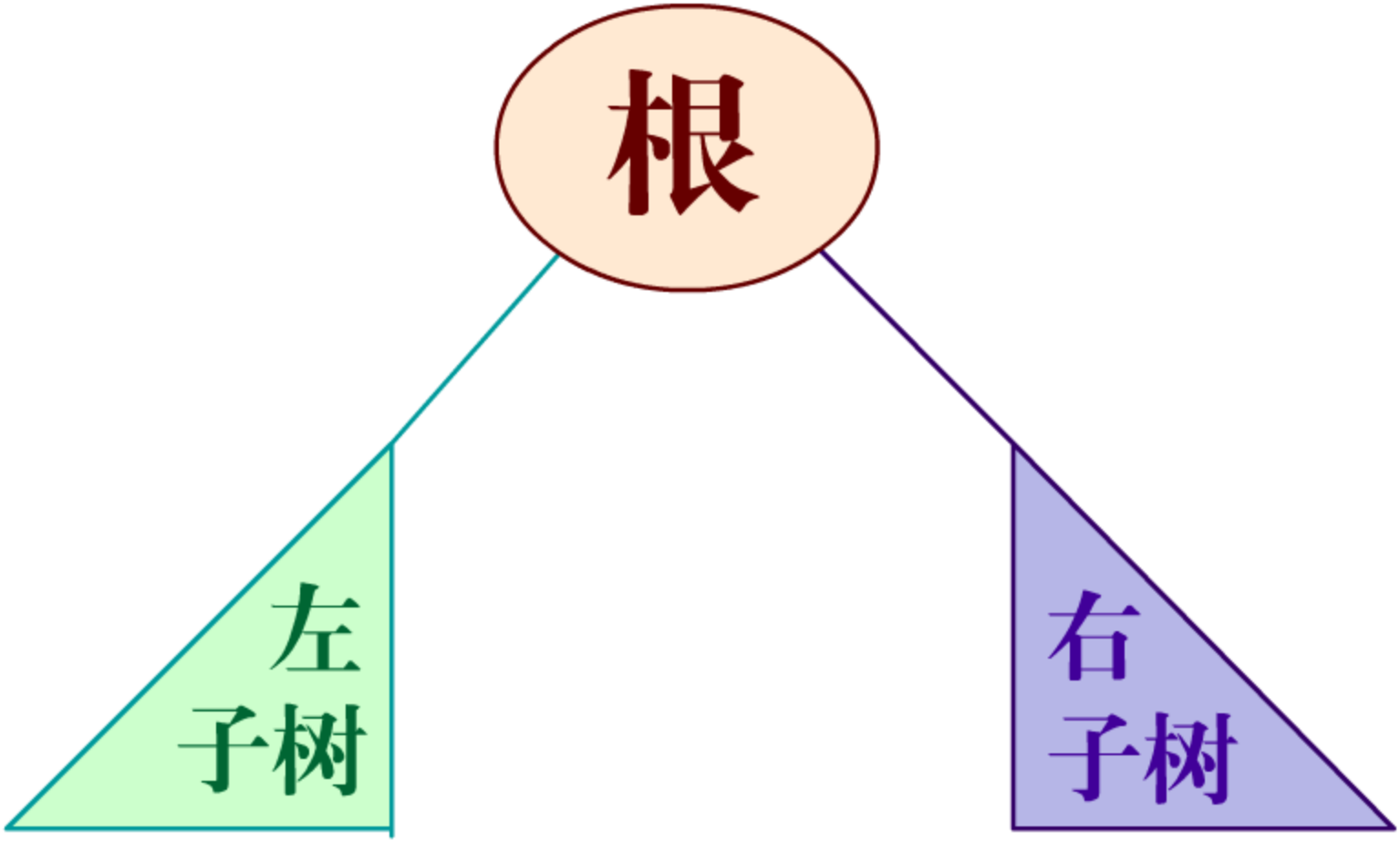
5.4 二叉树遍历

二叉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。遍历可认为是将一棵树进行线性化的处理。

对“二叉树”而言，可以有两类搜索策略：

1. 先上后下的按层次遍历
2. 先左（子树）后右（子树）的深度遍历

深度遍历



- 若限定子树先左后右访问，则对根的访问时机不同，有三种实现方案：

  DLR　先(根)序遍历

  LDR　中(根)序遍历

  LRD　后(根)序遍历

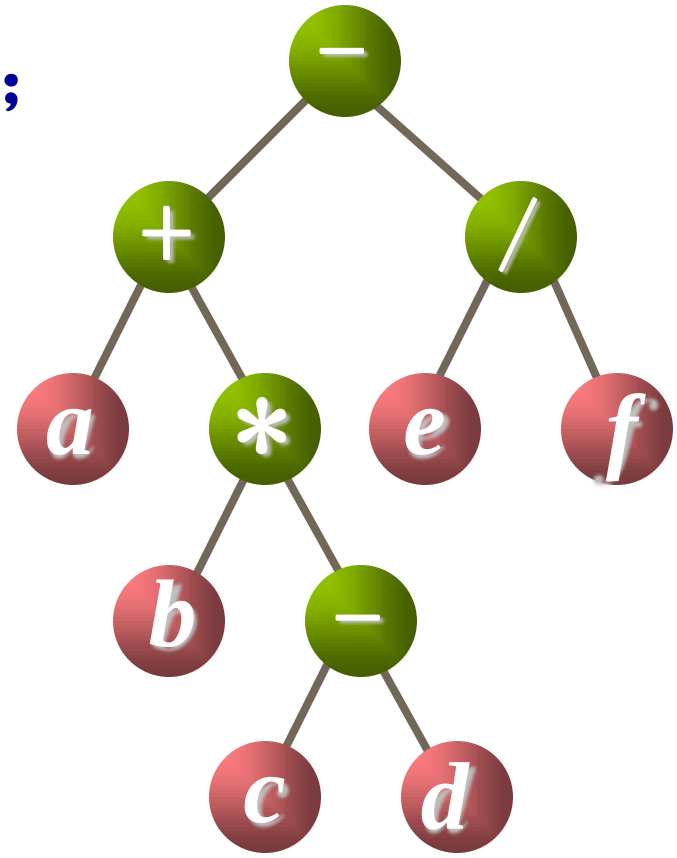
注：“先、中、后”的意思是指访问的结点D是先于子树出现还是后于子树出现。

先序遍历二叉树	中序遍历二叉树	后序遍历二叉树
若二叉树为空，则空操作；否则 (1) 访问根结点； (2) 先序遍历 左子树； (3) 先序遍历 右子树。	■若二叉树为空，则空操作；否则 (1) 中序遍历 左子树； (2) 访问根结点； (3) 中序遍历 右子树。	若二叉树为空，则空操作；否则 (1) 后序遍历 左子树； (2) 后序遍历 右子树； (3) 访问根结点。

中序遍历 (Inorder Traversal)

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历右子树 (R)。



遍历结果

$a + b * c - d - e / f$

中序遍历c版算法

若二叉树为空， 则空操作， 否则

(1)按中序遍历左子树；
(2)访问根结点；
(3)按中序遍历右子树。

```
void InTraverse( BiTree T )
{
  if(T){      //非空二叉树
     InTraverse(T->lchild);        //递归遍历左子树
     printf("%d",T->data);       //访问D
     InTraverse(T->rchild);       //递归遍历右子树
        }
 }
```

- “访问”一个节点的操作可以作为可变函数传入。

二叉树C++版递归中序遍历算法

```
template <class T>
void BinaryTree<T>::InOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t)) {
     if (subTree != NULL) {
          InOrder (subTree->leftChild, visit);
                                        //遍历左子树
          visit (subTree);              //访问根结点
          InOrder (subTree->rightChild, visit);
                                        //遍历右子树
     }
};
```

二叉树递归的中序遍历算法

```
void inOrder (void (*visit) (BinTreeNode<T> *t))
    { inOrder (root, visit); }          //中序遍历
```

复习：函数指针

函数是一段程序，运行时与变量一样占用内存，也就有一个起始地址，即指向此函数的指针。

函数指针定义格式:

类型名 （*函数名）（形参表）[=函数名];

如：若有函数：**int fun(int,char);**

则：**int (*p)(int, char);　p=fun;**

或　**int (*p)(int,char)=fun;　//p为指向函数fun的指针**

注：函数名为指针常量，不可改变；函数指针一般用作函数参数，用来简化有规律的函数调用。

例：使用函数名作函数参数

复习：函数指针

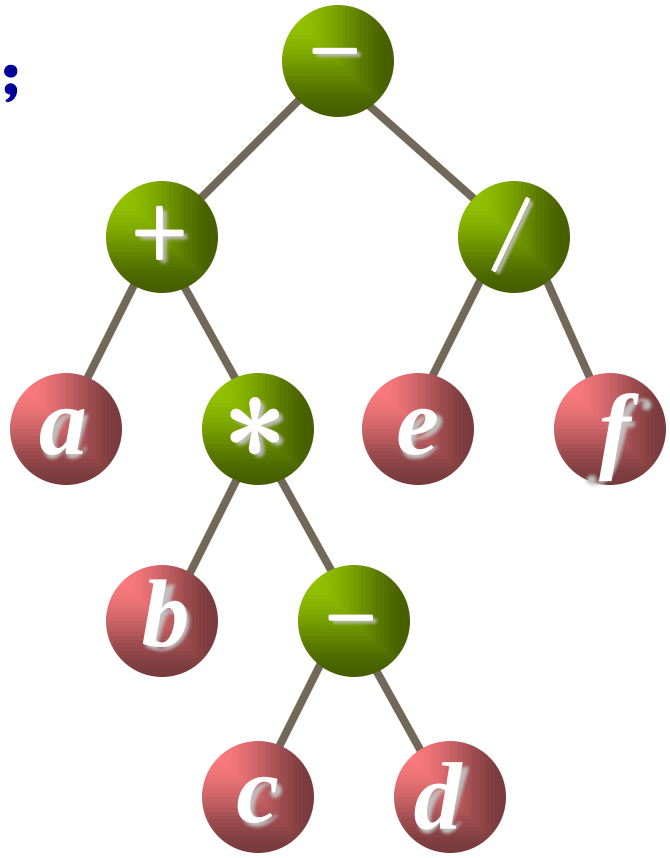
```
#include <iostream.h>
int minus(int a,int b) {return a-b;}
int plus(int a,int b)  {return a+b;}
int multiply(int a,int b)  {return a*b;}
int divide(int a,int b)  {return a/b;}

int op (int a, int b, int (*p)(int,int) ) {    return p(a,b);    }
void main( )
{    cout<<"5+3="<<op(5,3,plus)<<endl;
     cout<<"5-3="<<op(5,3,minus)<<endl;
     cout<<"5*3="<<op(5,3,multiply)<<endl;
     cout<<"5/3="<<op(5,3,&divide)<<endl;
}
```

前序遍历 (Preorder Traversal)

前序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 访问根结点 (V)；
  - 前序遍历左子树 (L)；
  - 前序遍历右子树 (R)。



遍历结果

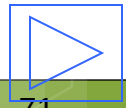
$- + a * b - c\ d / e\ f$

若二叉树为空， 则空操作， 否则

C版先序遍历算法

(1) 访问根结点；
(2) 按先序遍历左子树；
(3) 按先序遍历右子树。

```
void preTraverse( BiTree T )
{
 if(T){      //非空二叉树
    printf("%d",T->data); //访问D
    preTraverse(T->lchild );     //递归遍历左子树
    preTraverse(T->rchild );     //递归遍历右子树
        }
 }
```



二叉树递归的C++版前序遍历算法

```
template <class T>
void BinaryTree<T>::PreOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t)) {

if (subTree != NULL) {
        visit (subTree);                    //访问根结点
        PreOrder (subTree->leftChild, visit);
                                            //遍历左子树
        PreOrder (subTree->rightChild, visit);
                                            //遍历右子树
    }
};
```

二叉树递归的先序遍历算法

```
void PreOrder (void (*visit) (BinTreeNode<T> *t))
    { PreOrder (root, visit); }          //先序遍历
```

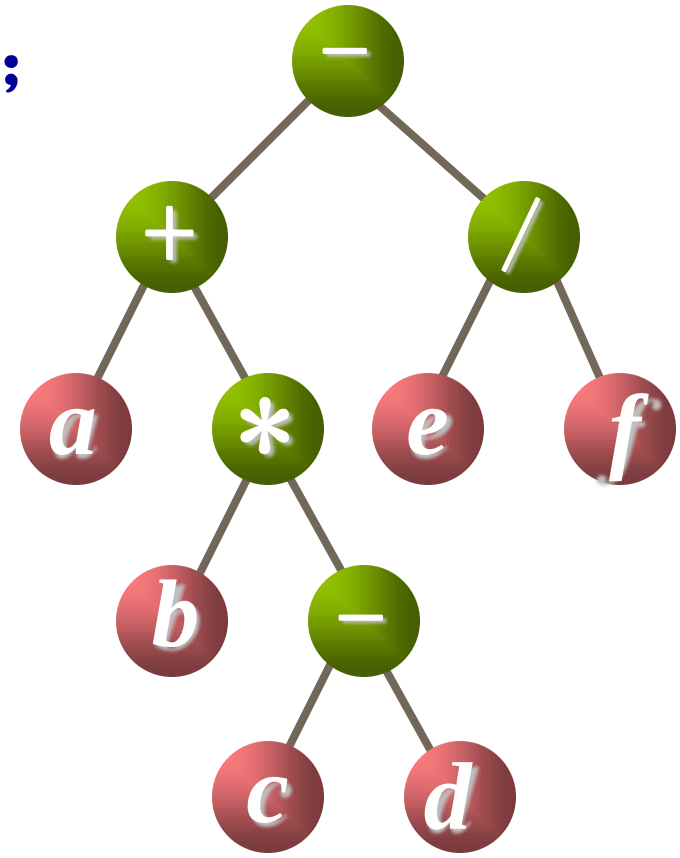
后序遍历 (Postorder Traversal)

后序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 后序遍历左子树 (L)；
  - 后序遍历右子树 (R)；
  - 访问根结点 (V)。

遍历结果

*a b c d – * + e f / –*



C版后序遍历算法

若二叉树为空， 则空操作， 否则

(1) 按后序遍历右子树。
(2) 按后序遍历左子树；
(3) 访问根结点；

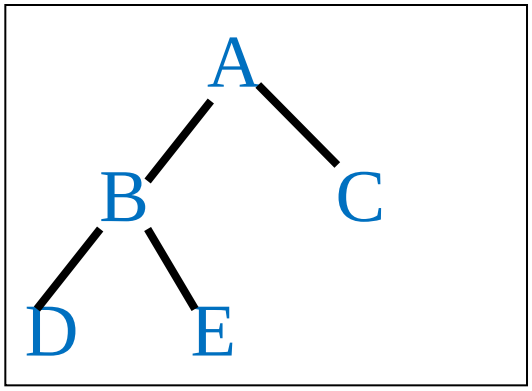
```
void PostTraverse( BiTree T )
{
  if( T ){      //非空二叉树
     PostTraverse(T->lchild);  //递归遍历左子树
     PostTraverse(T->rchild);  //递归遍历右子树
     printf( “%d”,T->data);  //访问D
          }
 }
```

二叉树递归的C++版后序遍历算法

```
template <class T>
void BinaryTree<T>::PostOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t ) ) {

    if (subTree != NULL ) {
        PostOrder (subTree->leftChild, visit);
                                          //遍历左子树
        PostOrder (subTree->rightChild, visit);
  //遍历右子树
        visit (subTree);              //访问根结点
    }
};
```

例 1：

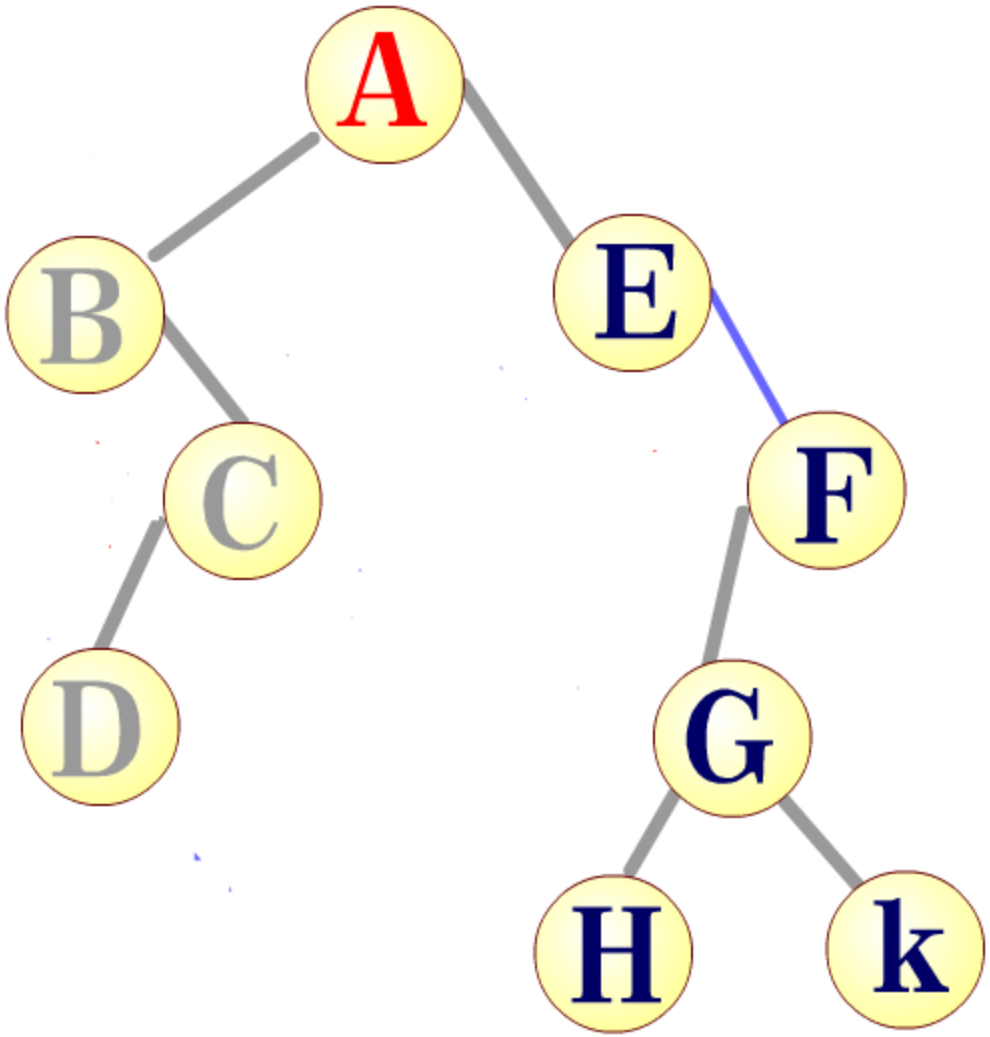


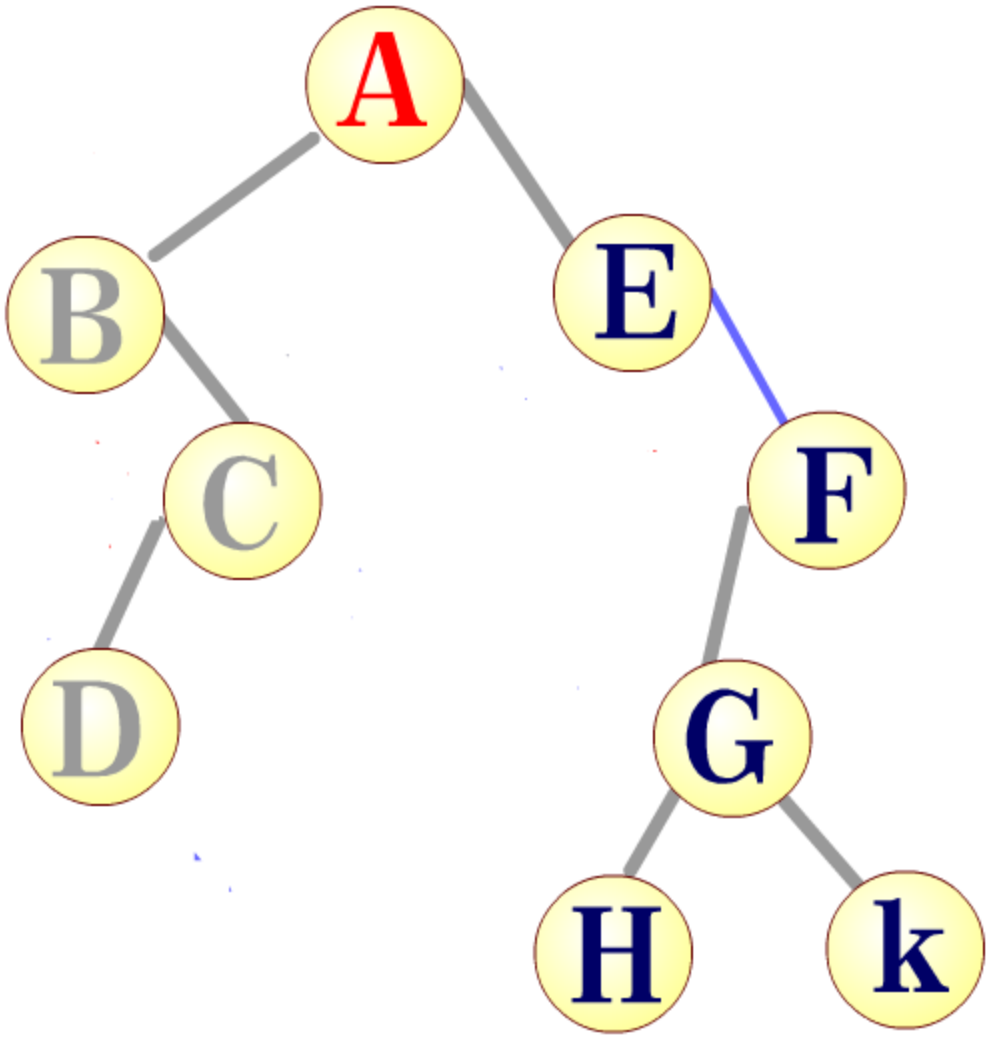
先序遍历的结果是：　A B D E C

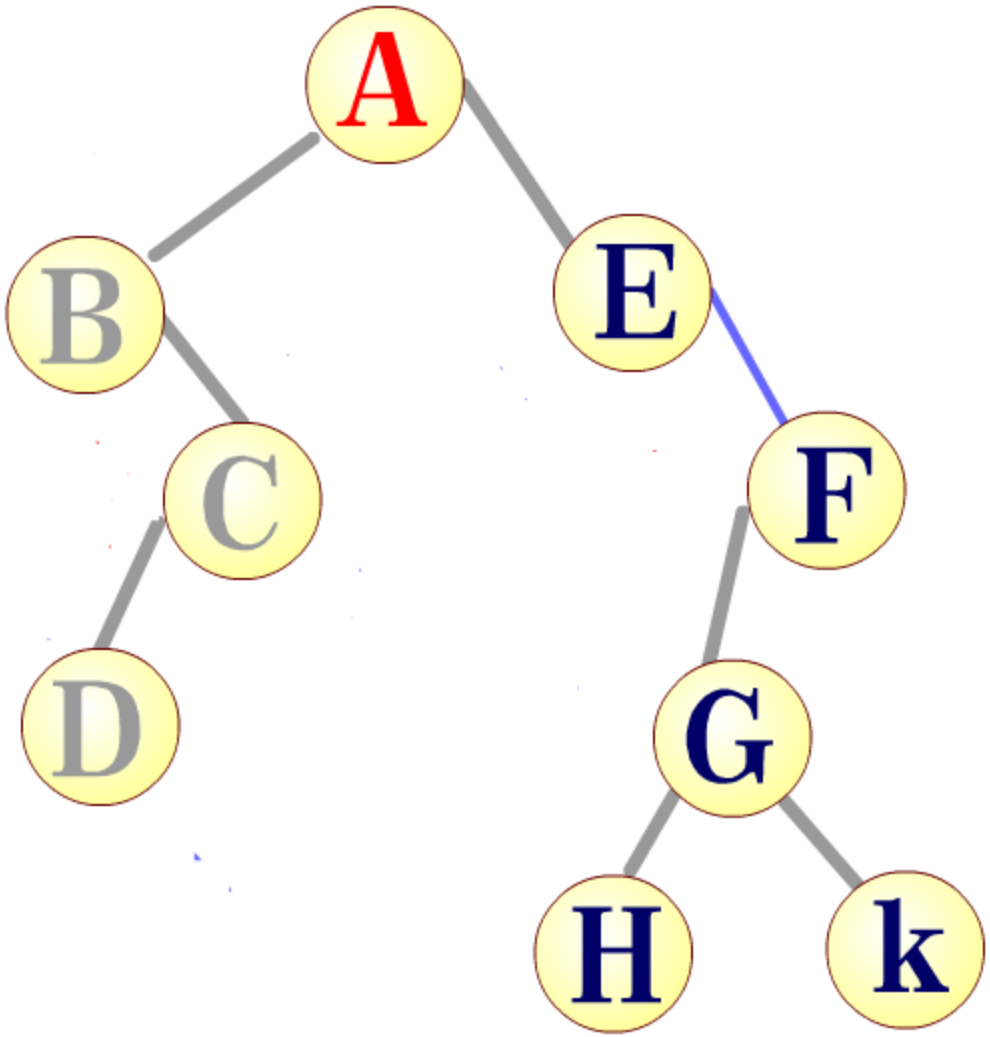
中序遍历的结果是：　D B E A C

后序遍历的结果是：　D E B C A

根在哪里

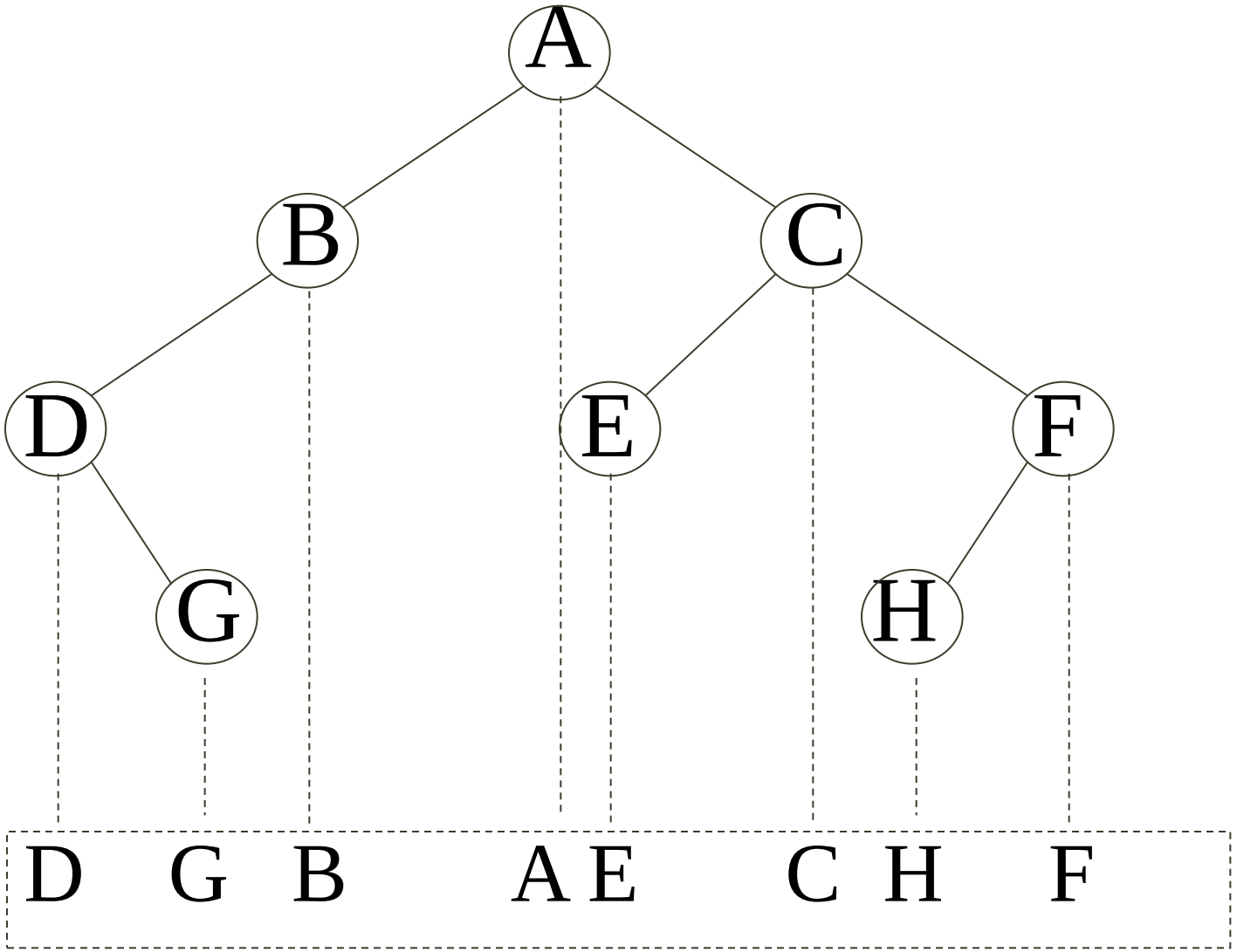

A
B
E
C
F
D
G
H
k


A
B
E
C
F
D
G
H
k

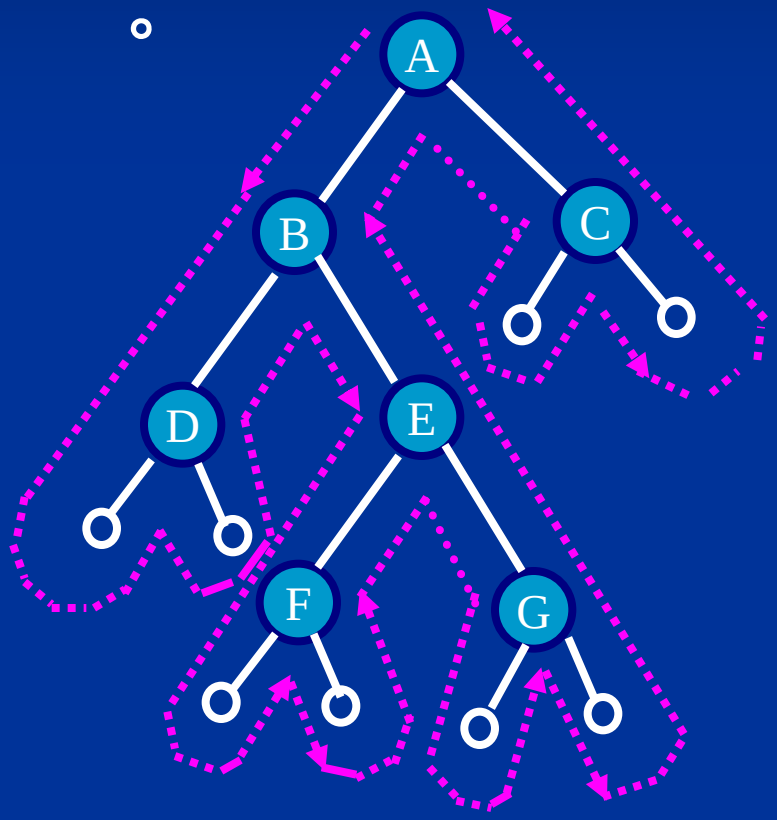

A
B
E
C
F
D
G
H
k

- 下面我们再给出两个遍历二叉树的技巧：

（1）对一棵二叉树**中序遍历**时，若我们将二叉树严格地按左子树的所有结点位于根结点的左侧，右子树的所有结点位于根右侧的形式绘制，就可以对每个结点做一条垂线，映射到下面的水平线上，由此得到的顺序就是该二叉树的中序遍历序列。


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

（2）任何一棵二叉树都可以将它的外部轮廓用一条线绘制出来，我们将它称为二叉树的包线，这条包线对于理解二叉树的遍历过程很有用。



从虚线的出发点到终点的路径上，每个结点经过**3**次。

第1次经过时访问＝**先序**遍历
第2次经过时访问＝**中序**遍历
第3次经过时访问＝**后序**遍历

- 由此可以看出：

（1）遍历操作实际上是将非线性结构线性化的过程，其结果为线性序列，并根据采用的遍历顺序分别称为先序序列、中序序列或后序序列；

（2）深度遍历操作是一个递归的过程，因此，这三种遍历操作的算法用递归函数实现最简单。

应用二叉树遍历的事例

```
template <class T>
int BinaryTree<T>::Size (BinTreeNode<T> *
    subTree) const {
//利用二叉树后序遍历算法计算二叉树的结点个数
    if (subTree == NULL) return 0;          //空树
    else return 1+Size (subTree->leftChild)
                + Size (subTree->rightChild);
};
```

应用二叉树遍历的事例

- int size(){ return size(root); }

```
template <class T>
int  BinaryTree<T>::Height ( BinTreeNode<T> *
   subTree) const {
//私有函数：利用二叉树后序遍历算法计算二叉
//树的高度或深度
     if (subTree == NULL) return 0;  //空树高度为0
    else {
        int i = Height (subTree->leftChild);
         int j = Height (subTree->rightChild);
        return (i < j) ? j+1 : i+1;
   };
```

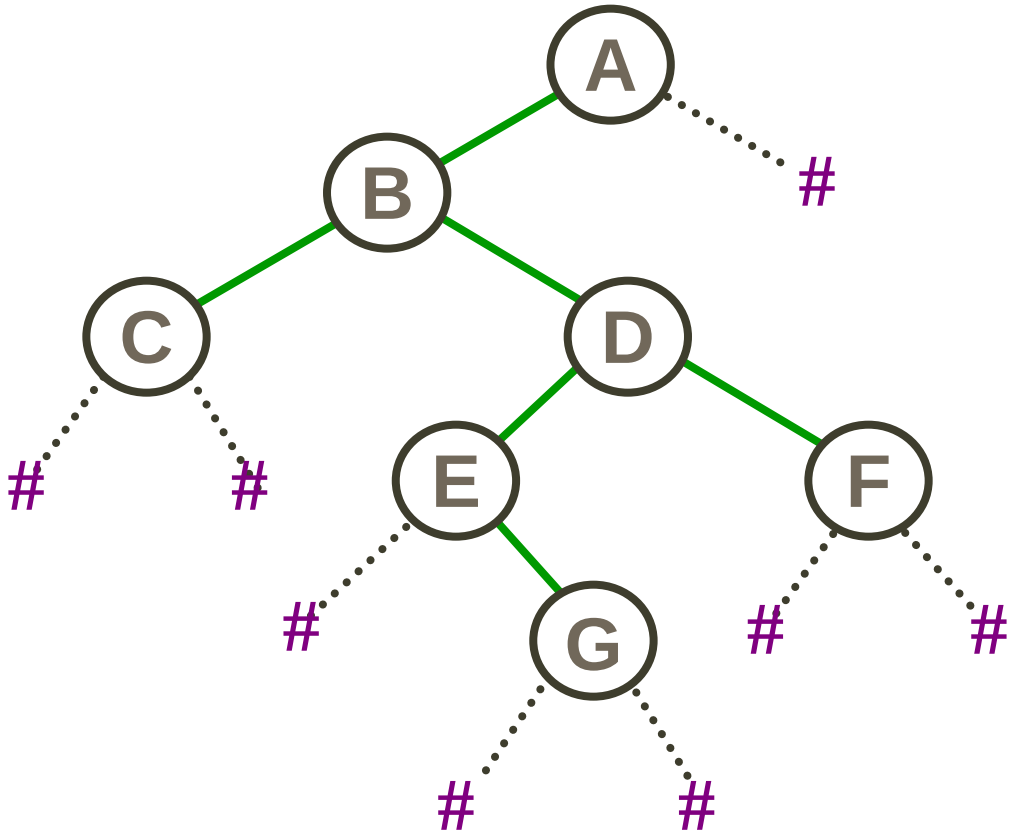
- int Height()
- {return Height(root);}

利用二叉树前序遍历建立二叉树

- 以递归方式建立二叉树。
- 输入结点值的顺序必须对应二叉树结点前序遍历的顺序。并约定以输入序列中不可能出现的值作为空结点的值以结束递归, 此值在RefValue中。例如用“#”或用“–1”表示字符序列或正整数序列空结点。

如图所示的二叉树的前序遍历顺序为

A B C # # D E # G # # F # # #



二叉树的存储结构及实现

```
template <class T>
void BiTree<T>::Creat(BiNode<T> * &bt)
{
    cin >> ch;                    //输入结点的数据信息，假设为字符
    if (ch == '# ') bt = NULL;              //建立一棵空树
    else {
      bt = new BiNode(ch);  //生成一个结点，数据域为ch

       if (bt == NULL) {cerr << "存储分配错!" << endl;  exit (1);}
     Creat(bt->lchild);          //递归建立左子树
     Creat(bt->rchild);          //递归建立右子树
   }}
```

```
binNode* BinTree::create() {
char temp ;
binNode* T;
 //读入字符，如是空格，结束否则生成节点，左，右子树
        cin>>temp ;
        if (temp=='@') return NULL ;
        else{
          T =new binNode(temp) ;
          T->lchild=create( );
          T->rchild=create();
          return T;
        }
}

BinTree::BinTree(void)
{   root=create();     }
```

融合操作符重载的代码

```
template<class T>
void BinaryTree<T>::CreateBinTree (istream& in,
   BinTreeNode<T> *& subTree) {
//私有函数: 以递归方式建立二叉树。
     T item;
     if ( !in.eof () ) {                  //未读完, 读入并建树
        in >> item;                       //读入根结点的值
        if (item != RefValue) {
           subTree = new BinTreeNode<T>(item); //建立根结点
           if (subTree == NULL)
             {cerr << "存储分配错!" << endl;  exit (1);}
         CreateBinTree (in, subTree->leftChild);
         CreateBinTree (in, subTree->rightChild);
         }
   else subTree = NULL;     //封闭指向空子树的指针
     }
};
```

部分成员函数的实现3-1

```
template<class T>
istream& operator >> (istream& in,
   BinaryTree<T>& Tree) {
//重载操作: 输入并建立一棵二叉树Tree。 in是输
//入流对象。
    CreateBinTree (in, Tree.root);     //建立二叉树
    return in;
};
```

部分成员函数的实现3-1

```
template <class T>
BinTreeNode<T> *BinaryTree<T>::
Parent (BinTreeNode <T> *subTree, BinTreeNode <T> *t) {
//从结点 subTree 开始, 搜索结点 t 的双亲, 若找
        //到则返回双亲结点地址, 否则返回NULL
    if (subTree == NULL) return NULL;
    if (subTree->leftChild == t || subTree->rightChild == t)
        return subTree;                    //找到, 返回父结点地址
    BinTreeNode <T> *p;
    if ((p = Parent (subTree->leftChild, t)) != NULL)
        return p;                          //递归在左子树中搜索
    else return Parent (subTree->rightChild, t);
                                           //递归在左子树中搜索
};
```

```
BinTreeNode<T> *Parent (BinTreeNode <T> *t)
    { return (root == NULL || root ==  t) ?
                NULL : Parent (root, t); }  //返回双亲结点
```

部分成员函数的实现3-2

```
template<class T>
void BinaryTree<T>::destroy (BinTreeNode<T> * subTree)
  {
                                  //私有函数: 删除根为subTree的子树
    if (subTree != NULL) {
        destroy (subTree->leftChild);     //删除左子树
       destroy (subTree->rightChild);   //删除右子树
       delete subTree;                          //删除根结点
    }
};
```

- ~ BinaryTree () { destroy(root); }　//析构函数