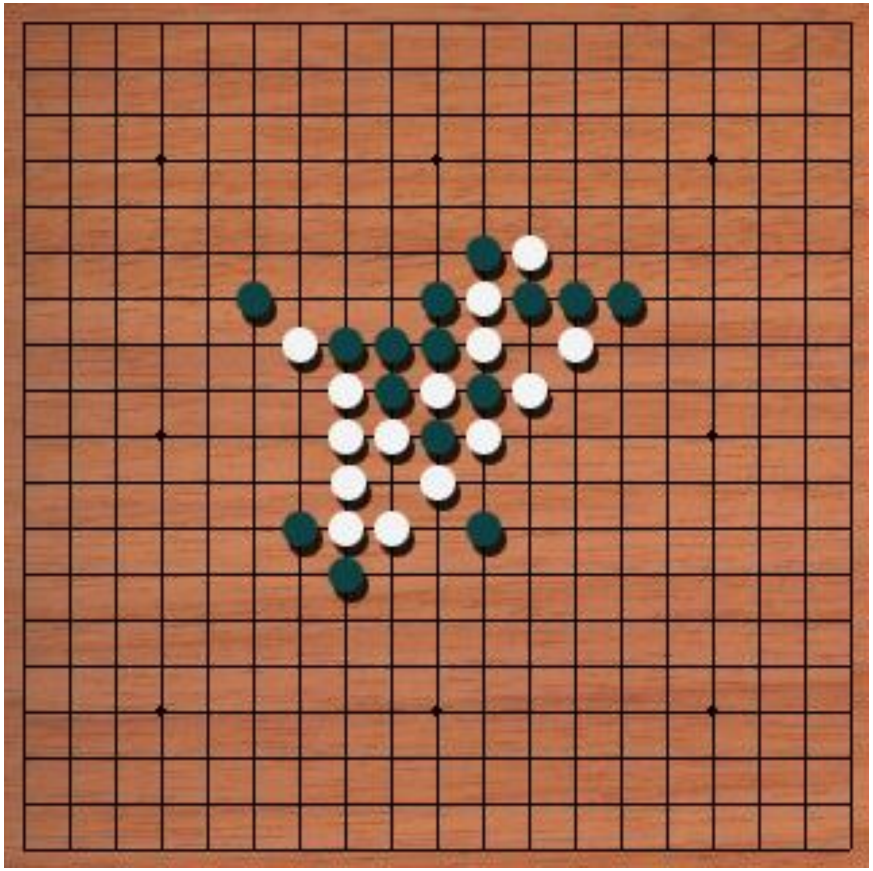
第五章 树

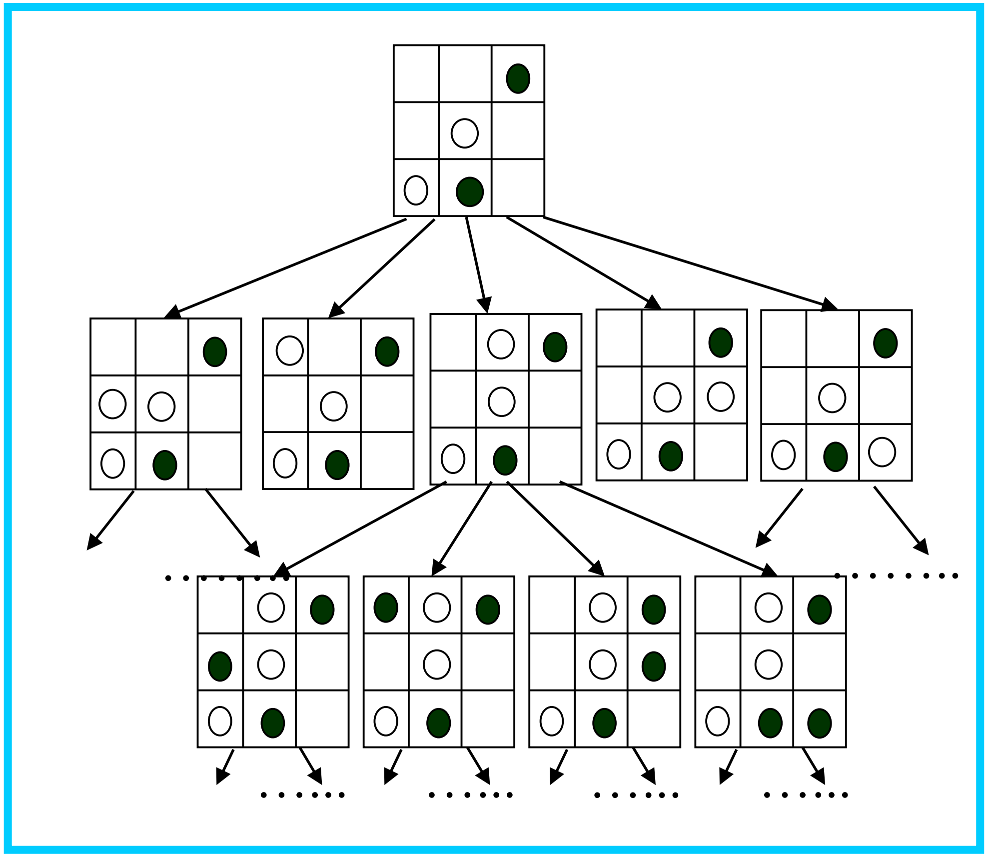
- 实例
- 树的定义及术语
- 二叉树的定义及性质
- 树型结构的存储
- 树型结构的遍历
- 二叉树的应用

本章小结 课后作业

树的实例

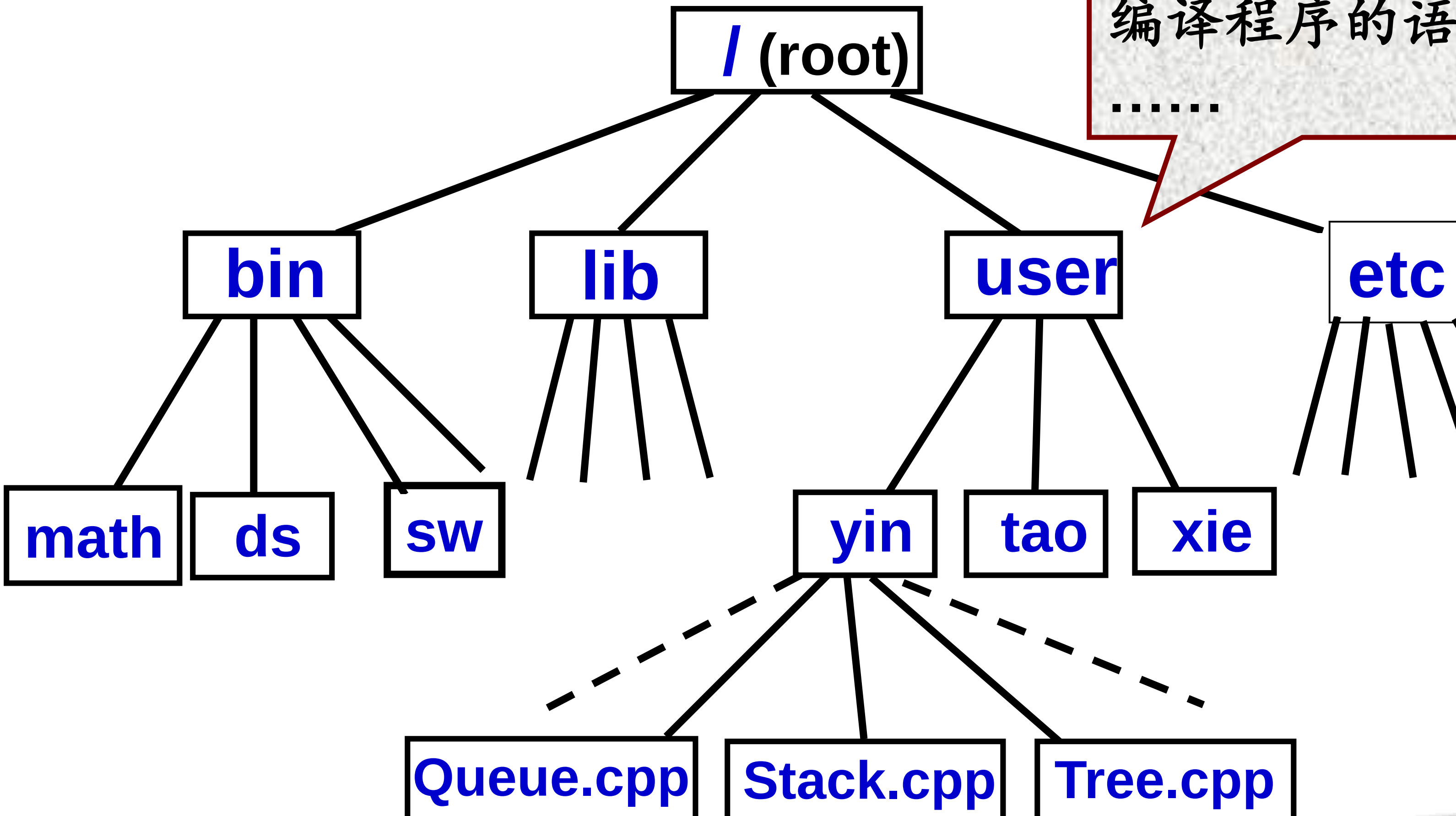
- 五子棋游戏





栈和队列实例

- UNIX 文件系统结构



人类的族谱
各种社会关系
各类分类编码
编译程序的语法树
……

Back

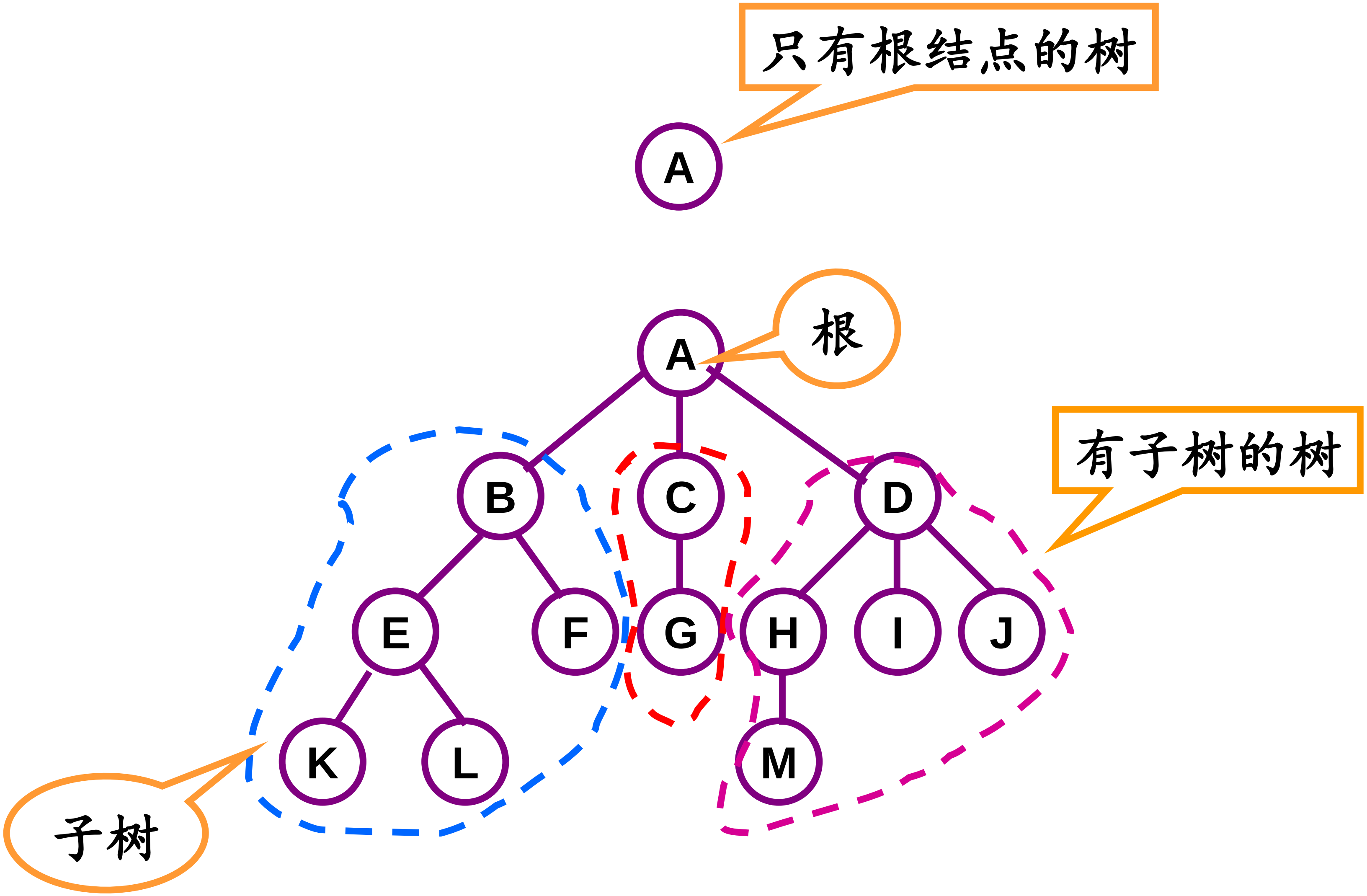
树的定义及术语

- 树是一类重要的非线性数据结构，是以分支关系定义的层次结构。

至少存在一个数据元素有两个或两个以上的直接前驱（或直接后继）

- 定义：树(tree)是n(n>=0)个结点的有限集T，其中：
  - ☺有且仅有一个特定的结点，称为树的根(root)；
  - ☺当n>1时，其余结点可分为m(m>0)个互不相交的有限集T1,T2,…,Tm，其中每一个集合本身又是一棵树，称为根的子树(subtree)
- 特点：
  - ☺非空树中，只有一个根结点
  - ☺非空树中，各子树是互不相交的集合

树的定义及术语



树的定义及术语

- 结点(node)——树中的元素，包括数据项及指向其子树的分支
- 结点的度(degree)——结点的子树数
- 叶子(leaf)——度为0的结点
- 孩子(child)——结点子树的根
- 双亲(parents)——孩子结点的上层结点
- 兄弟(sibling)——同一双亲的孩子
- 树的度——一棵树中最大的结点度数
- 结点的层次(level)——从根结点算起，根为第一层，它的孩子为第二层……
- 深度(depth)——树中结点的最大层次数
- 森林(forest)——m(m≥0)棵互不相交的树的集合

树的定义及术语

结点A的度：3

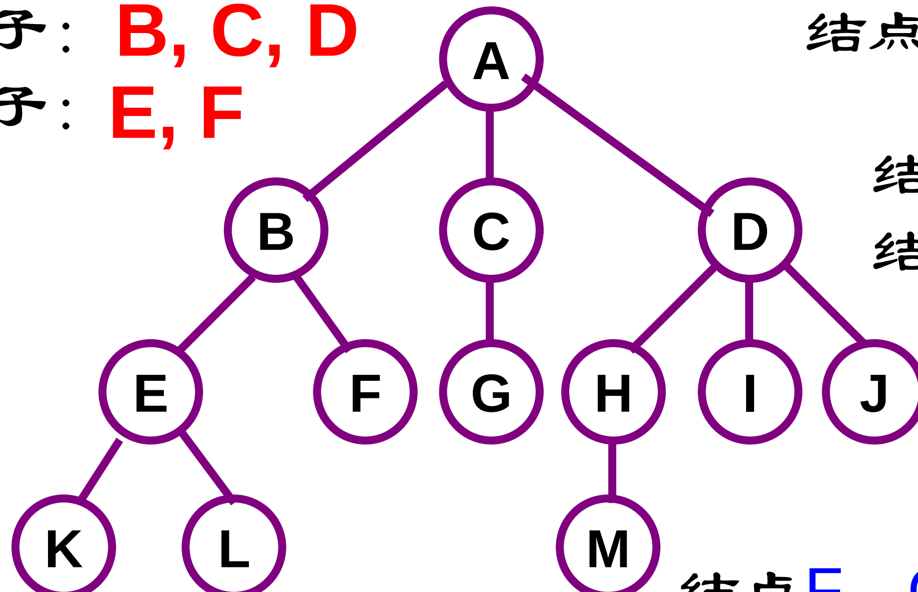
结点B的度：2

结点M的度：0

结点A的孩子：B, C, D

结点B的孩子：E, F

树的度：3



结点A的层次：1

结点M的层次：4

叶子：K, L, F, G, M, I, J

结点I的双亲：D

结点L的双亲：E

结点B，C，D为兄弟

结点K，L为兄弟

树的深度：4

结点F，G为堂兄弟

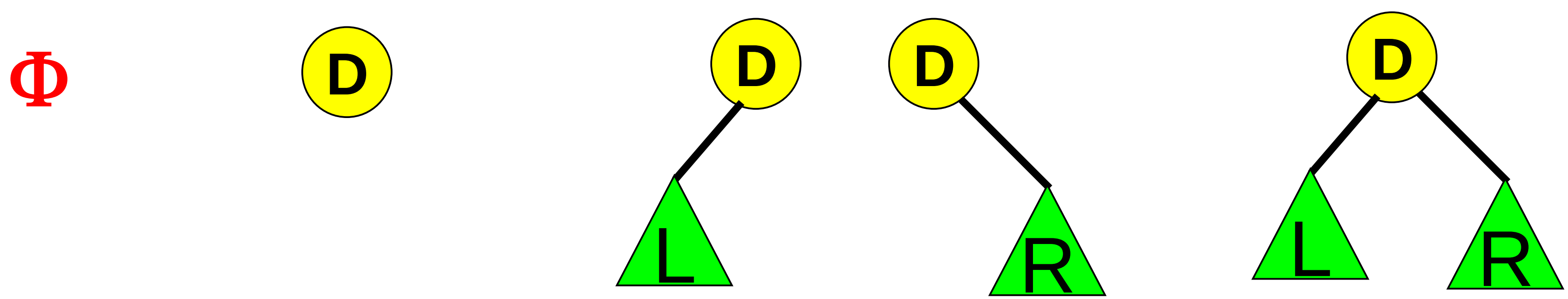
结点A是结点F，G的祖先

结点F，G是结点A的子孙

Back

二叉树的定义及性质

- 定义：二叉树是n(n≥0)个结点的有限集，它或为空树(n=0)，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树构成。
- 特点：
  - 每个结点至多有二棵子树，即不存在度大于2的结点；
  - 二叉树的子树有左、右之分。
- 二叉树的基本形态：



空二叉树　只有根结点的二叉树　右子树为空的二叉树　左子树为空的二叉树　左、右子树均非空的二叉树

二叉树的定义及性质

性质1：在二叉树的第 i 层上至多有2^{i-1} 个结点(i≥1)

证明：采用归纳法

归纳基： $i = 1$ 层时，只有一个根结点：

$$2^{i-1} = 2^0 = 1;$$

归纳假设：假设对所有的 j，$1 \le j < i$，命题成立，即第j层上至多有2^{j-1}个结点，

则第$i - 1$ 层上至多有2^{i-2} 个结点；

归纳证明：由于二叉树上每个结点至多有两棵子树，

则第 i 层的结点数 $= 2^{i-2} \times 2 = 2^{i-1}$ 。

二叉树的定义及性质

性质2：深度为 ***k*** 的二叉树上至多有 ***2^k-1*** 个结点（**k≥1**）

证明：由性质1可知，深度为k 的二叉树最大结点数为

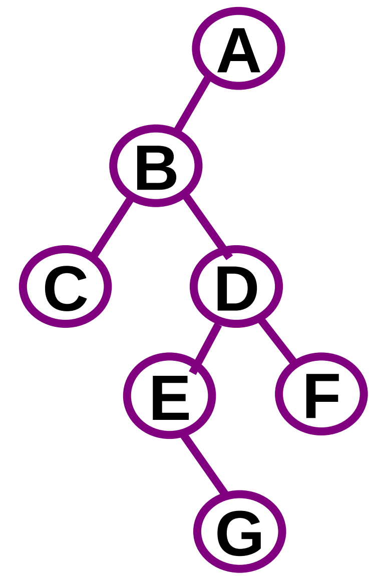
$$\sum_{i=1}^{k}(\text{第}i\text{层的最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = \frac{1-2^{k-1}*2}{1-2} = 2^k - 1$$

等比数列求和： $S_n = \dfrac{a_1 - a_n q}{1-q}$

二叉树的定义及性质

性质3：对任意一棵二叉树，若其叶子结点数为n_0，度为2的结点数为$n2$，则$n0=n2+1$。

证明：



考虑结点数

由于二叉树中结点的度均≤ 2，

设$n1$为二叉树中度为1的结点数，

$\therefore$ 二叉树结点总数$n= n0+n1+n2$

考虑分支数

设二叉树的分支总数为B，

除根结点外，其余结点都只有一个分支进入，则$n=B+1$

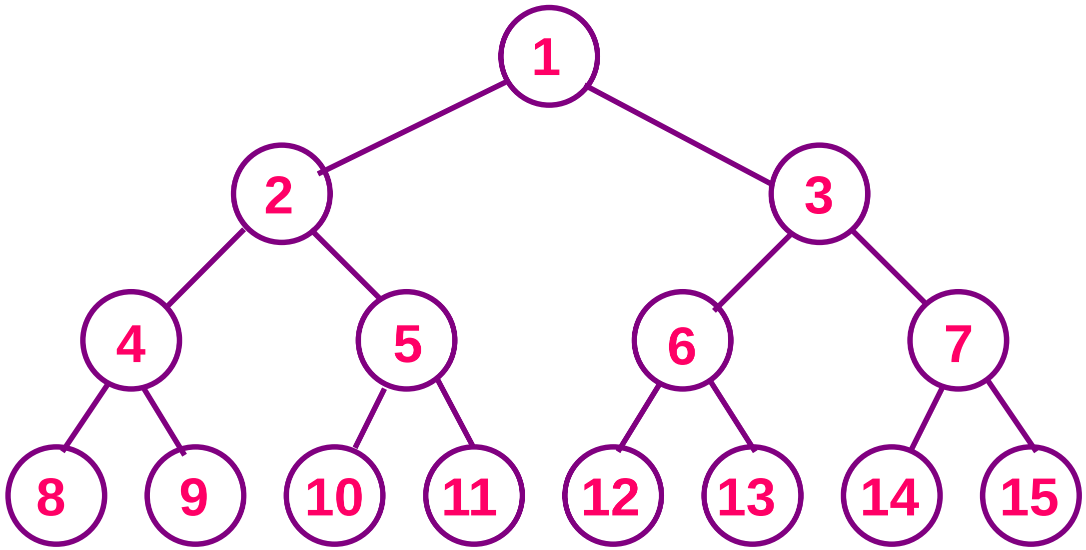
由于分支由度为1和度为2的结点射出，则$B=n1+2*n2$

$\therefore n=B+1=\cancel{n1}+\cancel{2}*n2+1 \; =n0+\cancel{n1}+\cancel{n2}$

证得：$n0=n2+1$

特殊形态的二叉树

满二叉树：深度为k且有2^k-1个结点的二叉树。



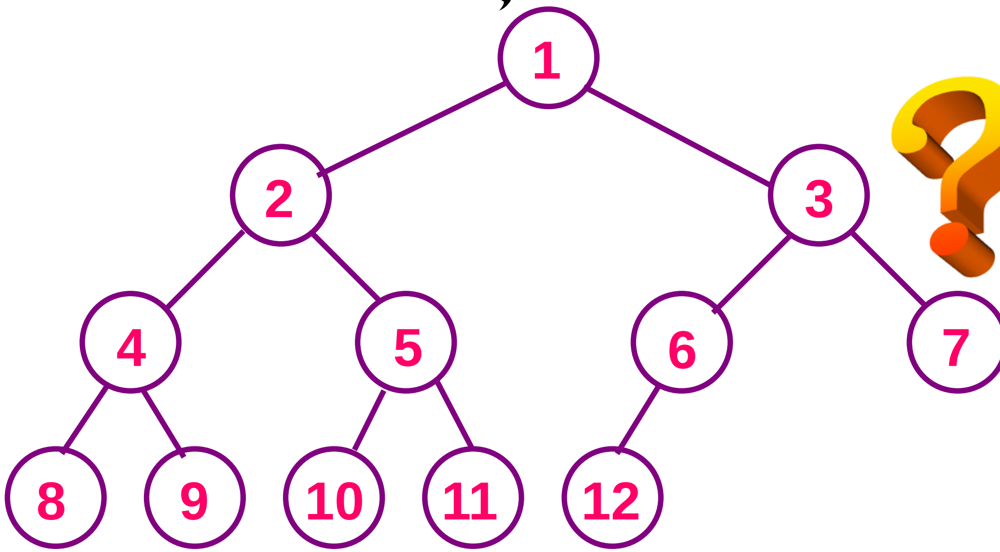
深度为4的满二叉树及结点编号

特点：

- 每一层上的结点数都是最大结点数；
- 除最下一层的叶子结点外，其余结点的度都为2，即不存在度为1的结点。

特殊形态的二叉树

完全二叉树：深度为k、有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称为~



完全二叉树中可能出现度为1的结点？

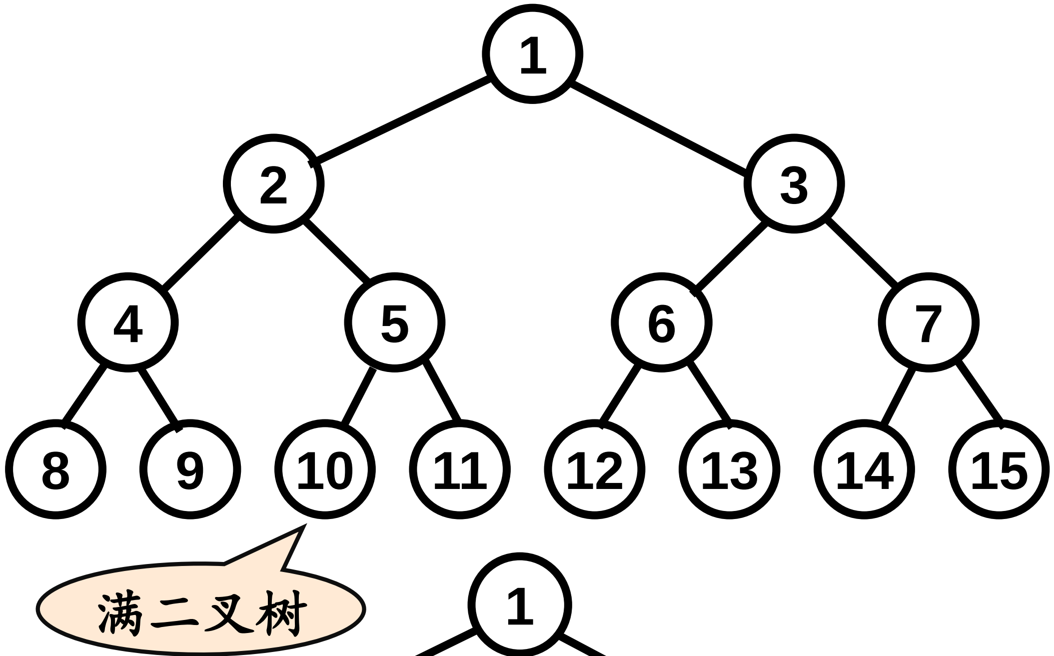
度为1的结点只能出现在倒数第二层上

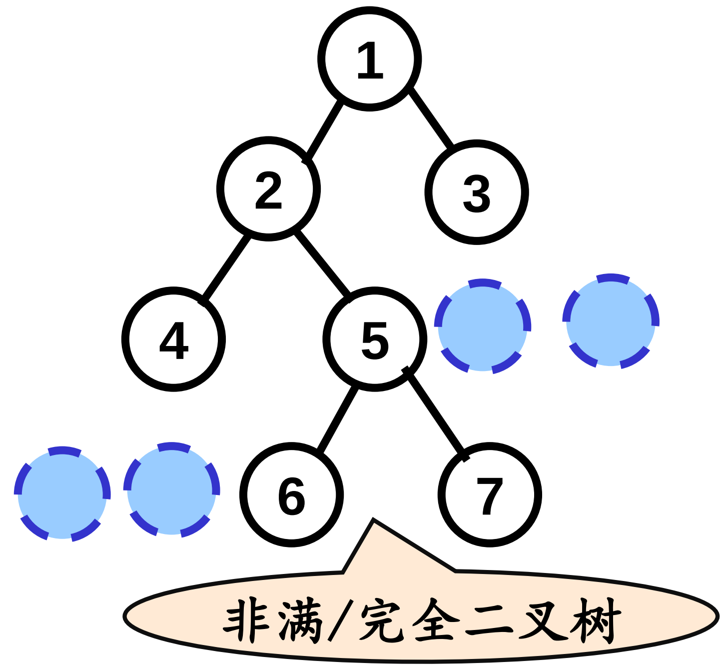
特点：

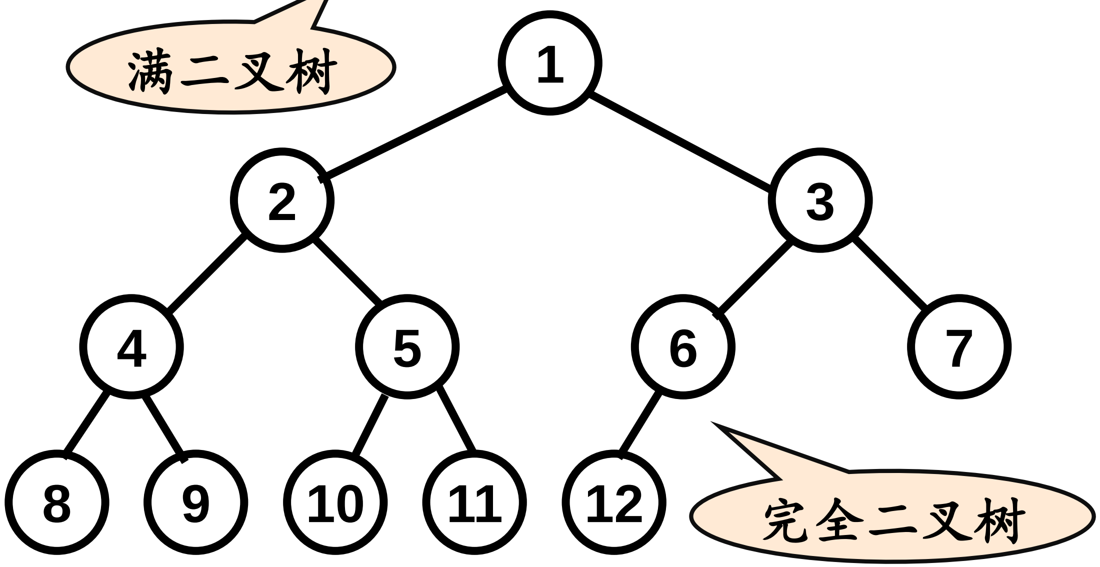
- ✓ 除了最下一层，其它层都达到了结点的最大数目，最下一层结点都集中在该层的最左边；
- ✓ 度<2的结点只能出现在最下两层上，若某结点没有左孩子，则一定没有右孩子，即为叶子结点。

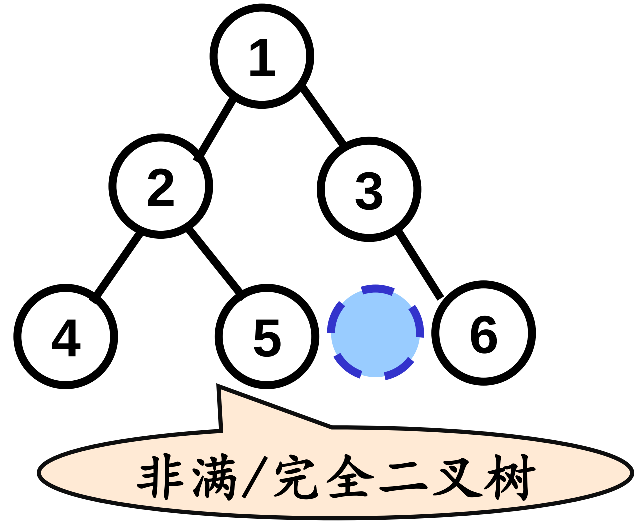
满二叉树

特殊形态的二叉树









完全二叉树的性质

性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$

证明：设有n个结点的完全二叉树的深度为k，根据二叉树的性质2：

深度为 k 的二叉树上至多有 2^k-1 个结点

深度为k-1的满二叉树结点数

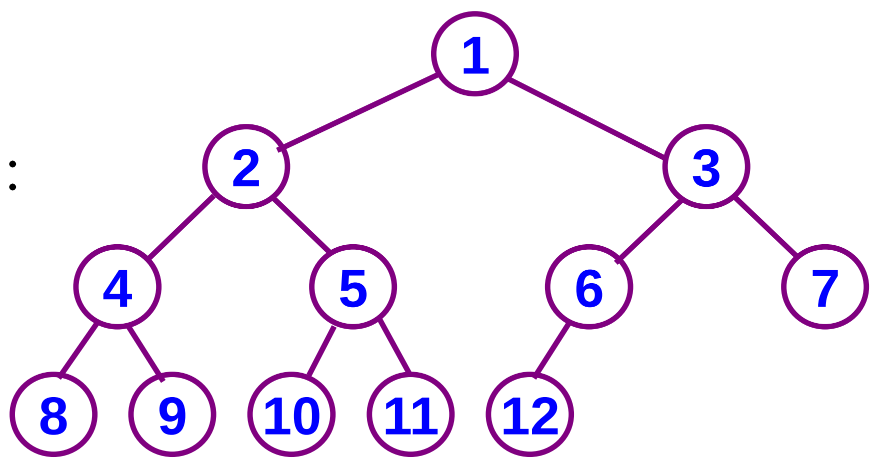
$\therefore\ 2^{k-1}-1 < n \le 2^k-1$

即 $2^{k-1} \le n < 2^k$

$\therefore k-1 \le \log_2 n < k$

由于k是整数，因此得：

$$k = \lfloor \log_2 n \rfloor + 1$$



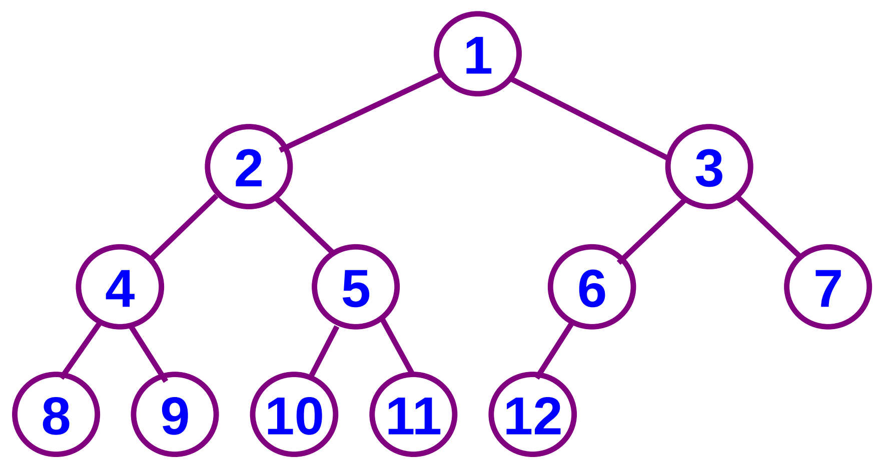
完全二叉树的性质

性质5：对n个结点的完全二叉树进行结点编号(从上到下、从左向右的顺序)，则对编号为i(1≤i≤n)的结点，有：

❶ 若i=1，则该结点为二叉树的根，无双亲；若i>1，则其双亲结点的编号为$\lfloor i/2 \rfloor$；

❷ 若2i≤n，则该结点的左孩子编号为2i；若2i>n，则该结点无左孩子；

❸ 若2i+1≤n，则该结点的右孩子编号为2i+1；若2i+1>n，则该结点无右孩子。



Back

树型结构的存储

树的存储结构

双亲表示法

孩子表示法

孩子兄弟表示法

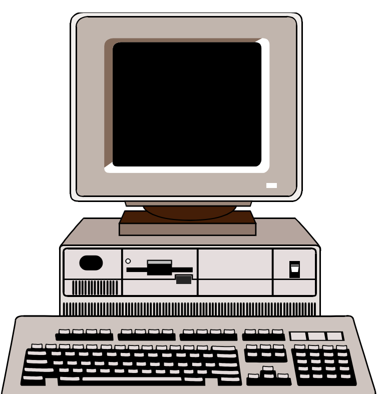
二叉树的存储结构

顺序存储结构

二叉链表

三叉链表

树与二叉树的相互转换



Back

树的存储结构

双亲表示法 ——存储结点之间的"双亲关系"

结点结构

data	parent

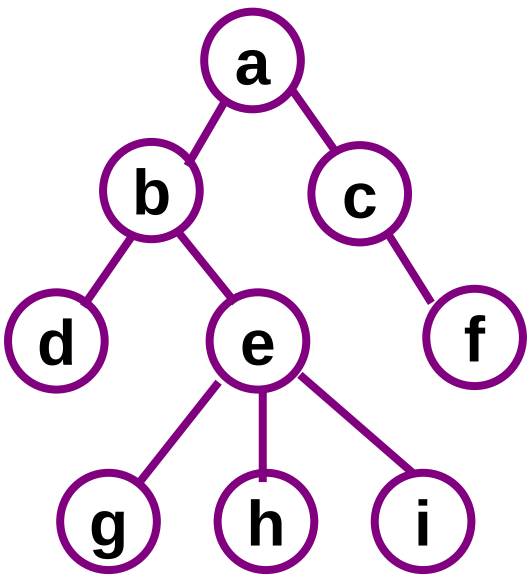
```
typedef struct PTNode
{   TElemType  data;    //数据域
     int    parent;   // 双亲域
 } PTNode;
```

树结构

```
#define TreeSize  100    //树中结点数目
typedef struct
{   PTNode  nodes[TreeSize];    //结点数组
     int  r, n;   //根结点的下标和结点总数
 } PTree;
```

树的存储结构

双亲表示法 ——存储结点之间的"双亲关系"



-1不在数组下标范围内，可用于表示根结点无双亲

	data	parent
0	a	-1
1	b	0
2	c	0
3	d	1
4	e	1
5	f	2
6	g	4
7	h	4
8	i	4

T.nodes

```
#define TreeSize  100
typedef struct
{ PTNode  nodes[TreeSize];
  int   r, n;
}PTree;
```

```
PTree T;
T.r=0; //根结点位置
T.n=9; //结点个数
```

找双亲容易
找孩子难☹

如何找孩子结点？

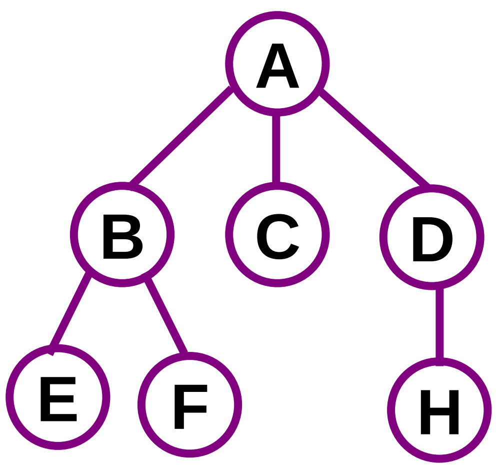
树的存储结构

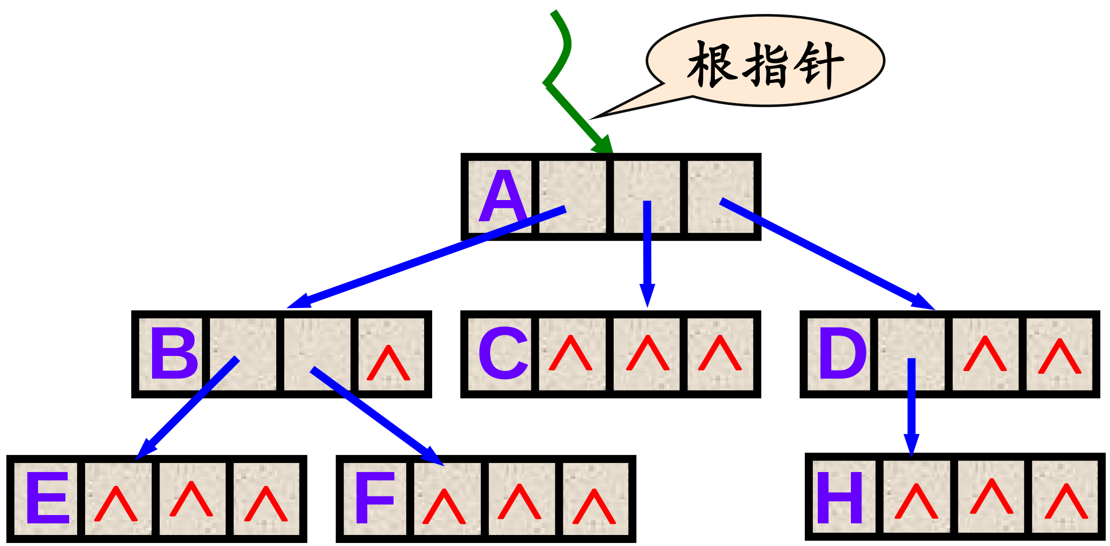
- 孩子表示法 ——存储结点之间的"孩子关系"
  - 多重链表：每个结点有多个指针域，每个指针指向一个孩子结点
    - ✓ 结点同构：结点的指针个数相等，为树的度D

data	child1	child2	…	childD

    - ✓ 结点不同构：结点指针个数不等，为该结点的度d

data	degree	child1	child2	…	childd





树的存储结构

孩子表示法 ——存储结点之间的"孩子关系"

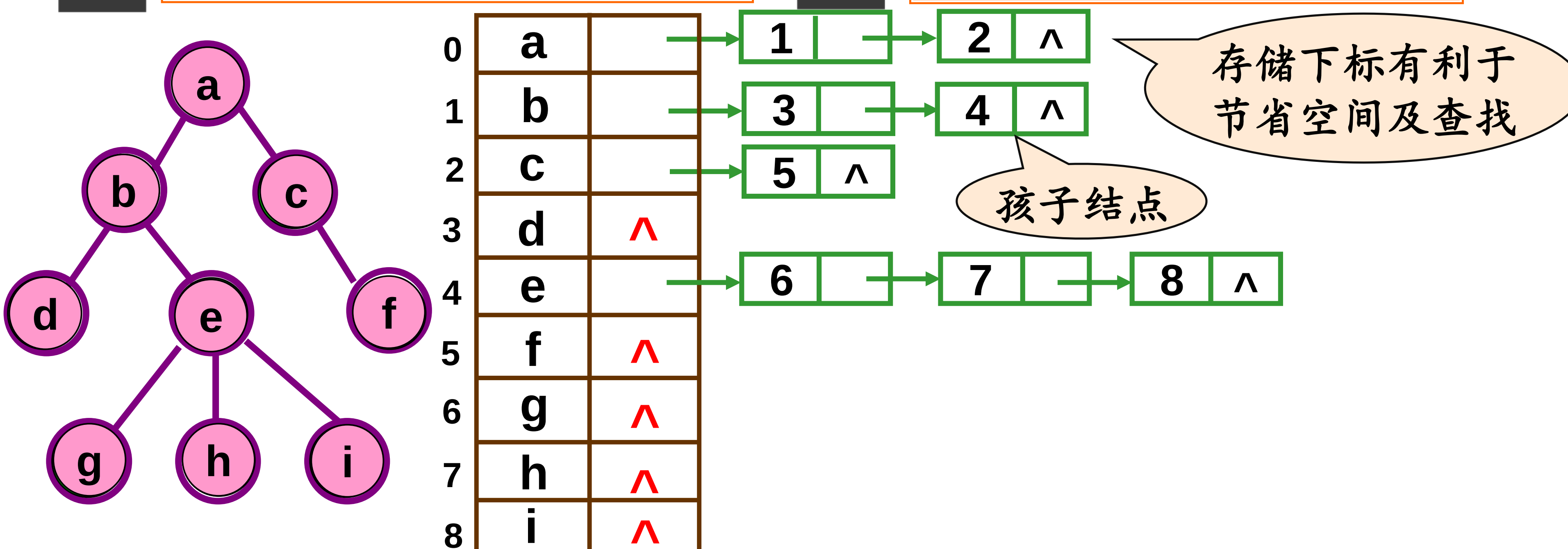
孩子链表：每个结点的孩子用单链表存储，用长度为n的数组存放每个结点的数据域及其单链表的头指针

孩子结点

```
typedef struct CTNode
{  int  child;
   struct  CTNode  *next;
}*ChildPtr;
```

数组结点

```
typedef   struct
{   TElemType   data;
    ChildPtr   firstchild;
}CTBox;
```



树的存储结构

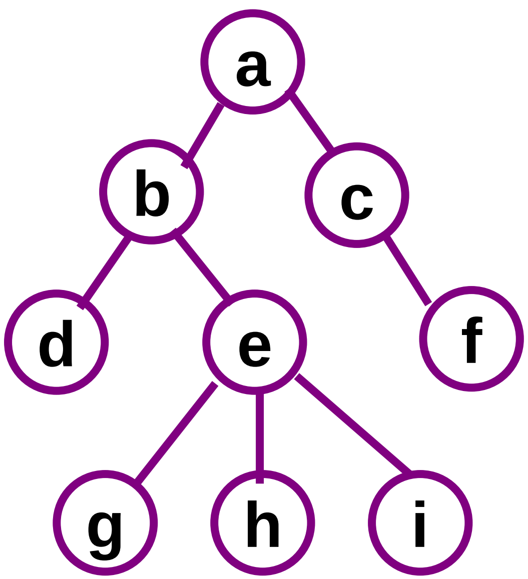
- 孩子表示法 ——存储结点之间的"孩子关系"
  - 孩子链表：每个结点的孩子用单链表存储，用长度为n的数组存放每个结点的数据域及其单链表的头指针

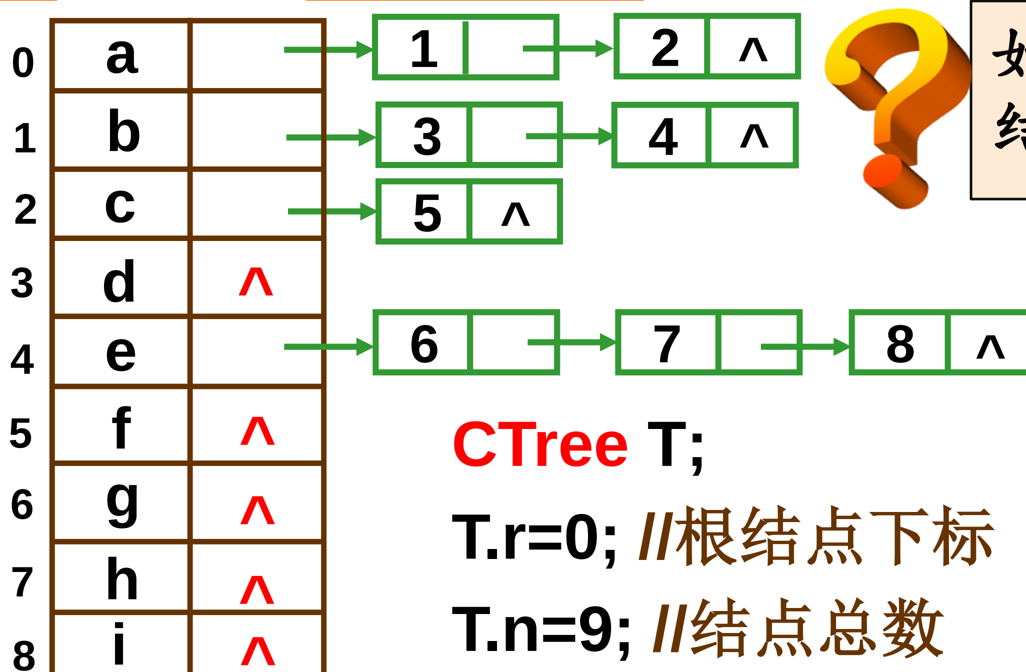
树结构

```
typedef   struct
{ CTBox  nodes[TreeSize];
   int r, n; //根结点下标、结点总数
}CTree;
```

找孩子容易
找双亲难☹

T.nodes





如何找双亲结点？

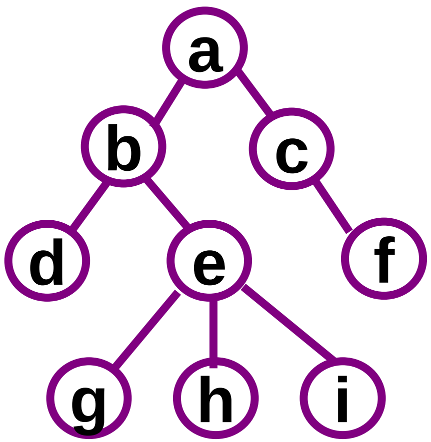
CTree T;

T.r=0; //根结点下标

T.n=9; //结点总数

树的存储结构

带双亲的孩子链表——结合"双亲表示法"和"孩子链表"



		parent	
0	a	-1	1 → 2 ^
1	b	0	3 → 4 ^
2	c	0	5 ^
3	d	1	^
4	e	1	6 → 7 → 8 ^
5	f	2	^
6	g	4	^
7	h	4	^
8	i	4	^

找孩子容易
找双亲容易☺

使用树解决实际问题时，需要根据经常对树中结点进行的操作选取合适的存储方式。

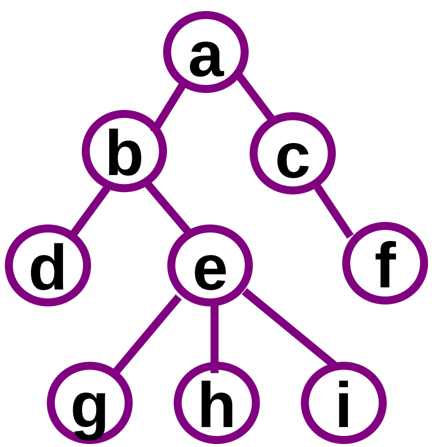
树的存储结构

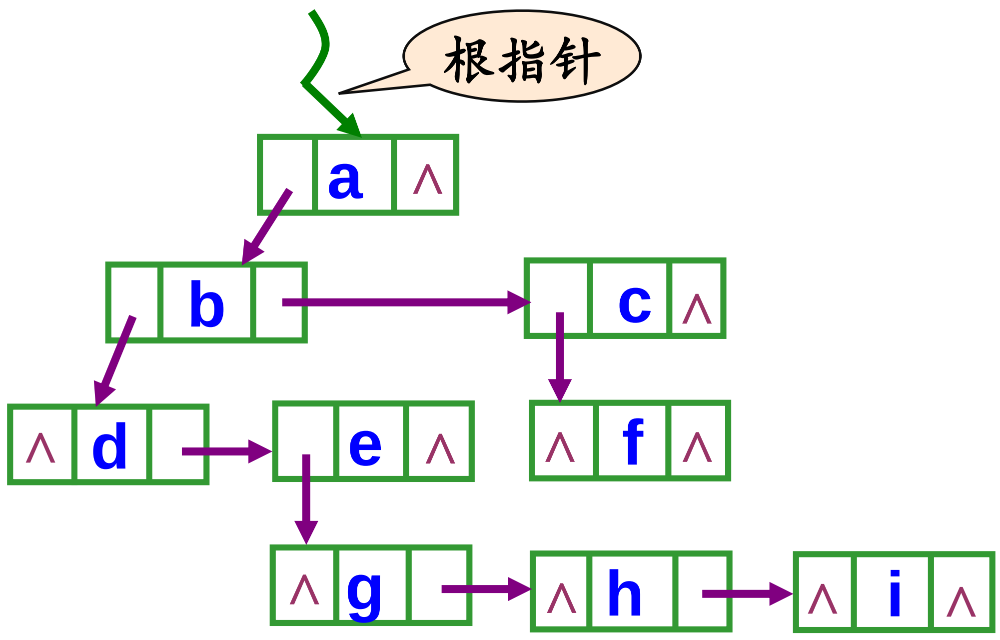
- 孩子-兄弟表示法——存储结点之间的”孩子及兄弟关系”

结点结构

firstchild	data	nextsibling

```
typedef struct CSnode
{   ElemType   data;       //数据域
    struct CSnode   *firstchild, *nextsibling;
                              //指向第一个孩子、下一个兄弟
} CSNode, *CSTree;
```

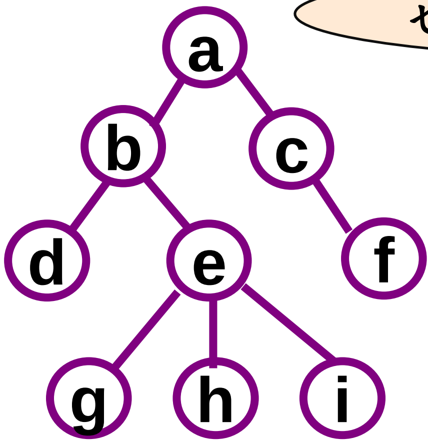




树的存储结构

孩子-兄弟表示法——存储结点之间的"孩子及兄弟关系"

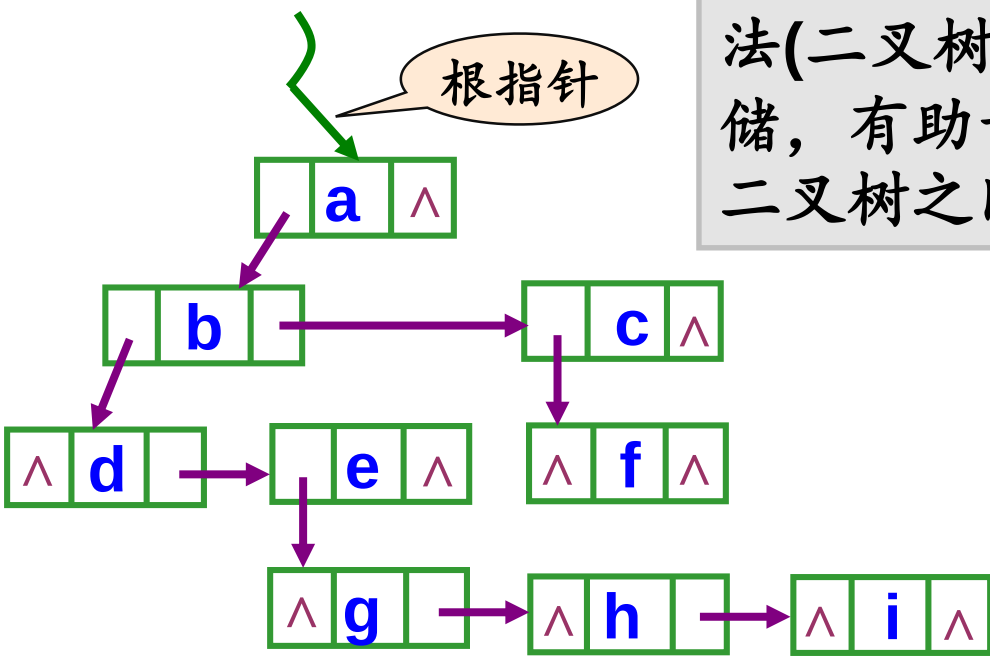
也称为"二叉树表示法"

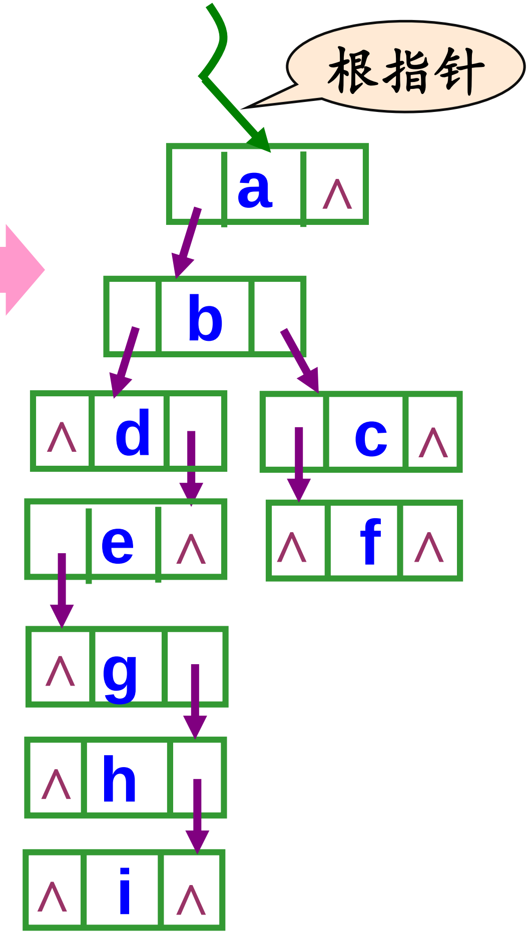


特点：

- 操作简单
- 破坏了树的层次

将树用孩子-兄弟表示法(二叉树表示法)存储，有助于实现树与二叉树之间的转换。





Back

二叉树的存储结构

◆顺序存储结构

——将二叉树"补成"完全二叉树，以完全二叉树中的结点编号为下标，将结点存储在顺序存储结构中。

二叉树的顺序存储结构

```
#define  Tree_Size  100     //二叉树中结点数目
typedef TElemType  SqBiTree[Tree_Size];
```

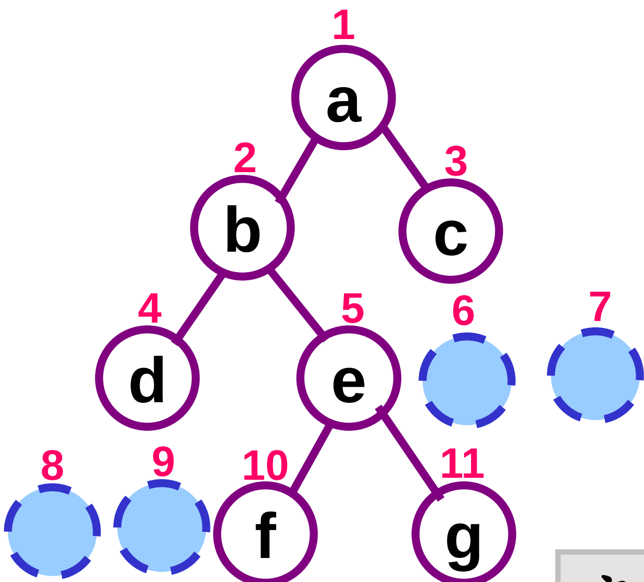
定义新类型名的基本步骤——

①写出定义变量的普通形式	例 int a[100];
②将变量名替换为"新类型名"	int ARRAY[100];
③前面加上typedef	typedef int ARRAY[100];
即可使用"新类型名"定义变量	ARRAY b;

二叉树的存储结构

顺序存储结构

——将二叉树"补成"完全二叉树，以完全二叉树中的结点编号为下标，将结点存储在顺序存储结构中。



SqBiTree T;

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g

该存储结构能否反映结点之间的关系？

完全二叉树中，能够判断编号为i的结点是否有双亲及左、右孩子；若有，能确定其编号

特点：

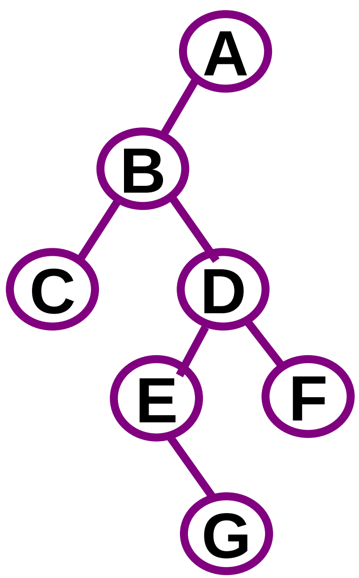
- ☺结点间的父子关系蕴含在其存储位置中
- ☹适于存储满/完全二叉树，存储其它二叉树则浪费空间

二叉树的存储结构

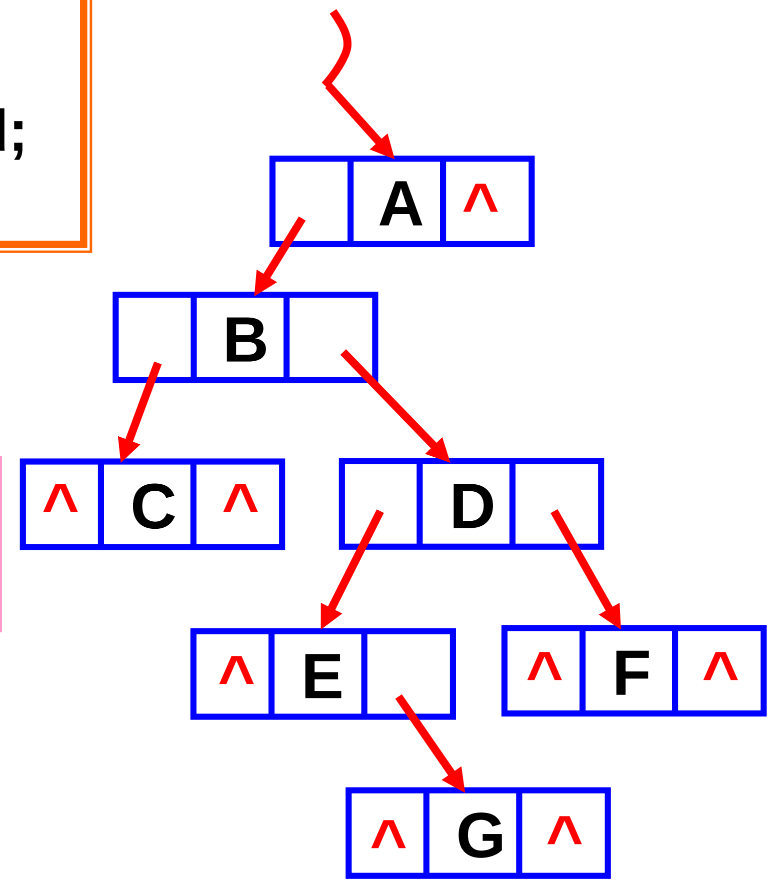
链式存储——二叉链表

lchild	data	rchild

```
typedef struct BiTNode
{   TElemType      data;    //数据域
    struct BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
```



找孩子容易
找双亲难☹



有n个结点的二叉链表中，有几个空指针域？

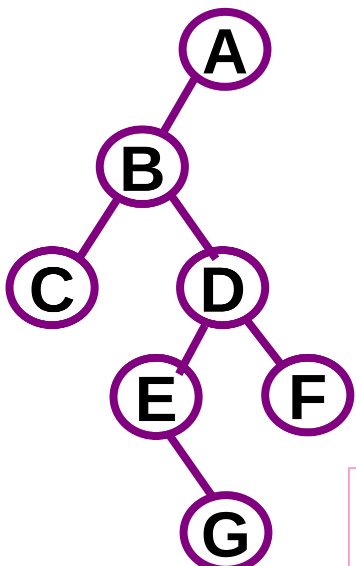
n+1个

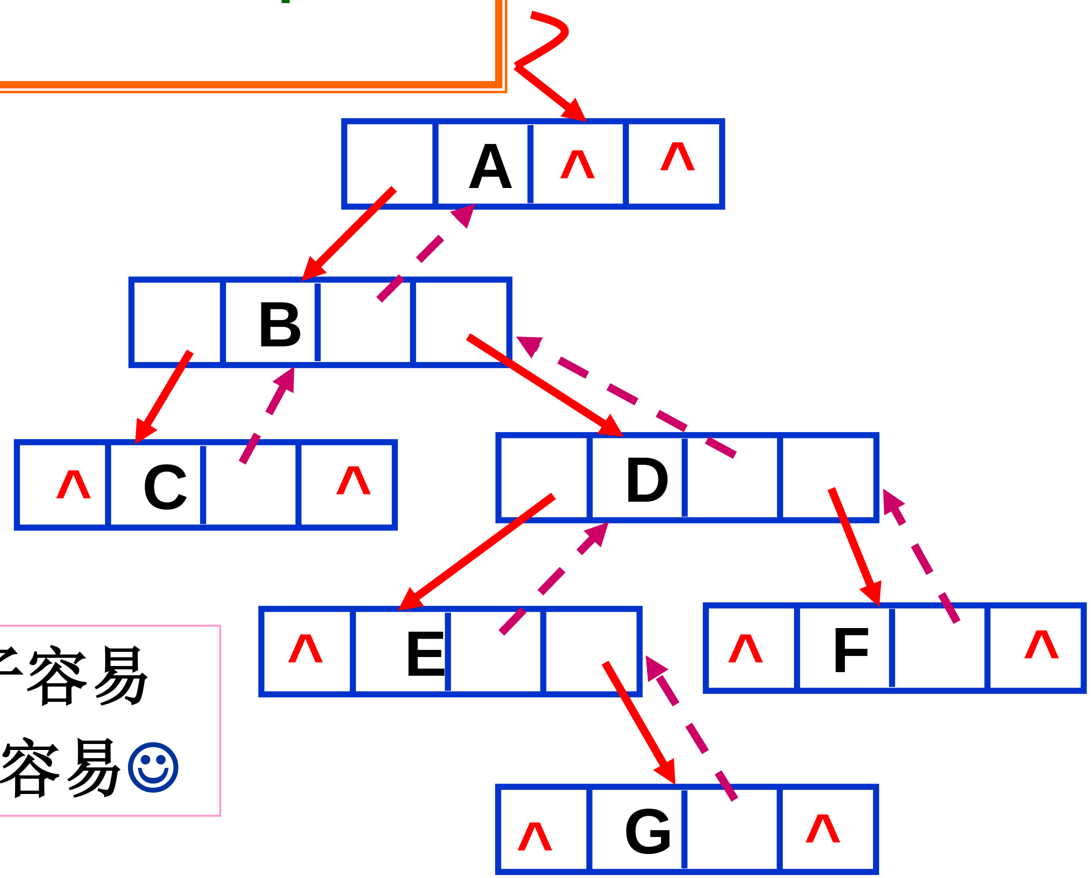
二叉树的存储结构

链式存储——三叉链表

lchild	data	parent	rchild

```
typedef struct TriTNode
{   TElemType   data;       //数据域
     struct TriTNode  *lchild, *rchild, *parent;
 } TriTNode, *TriTree;
```



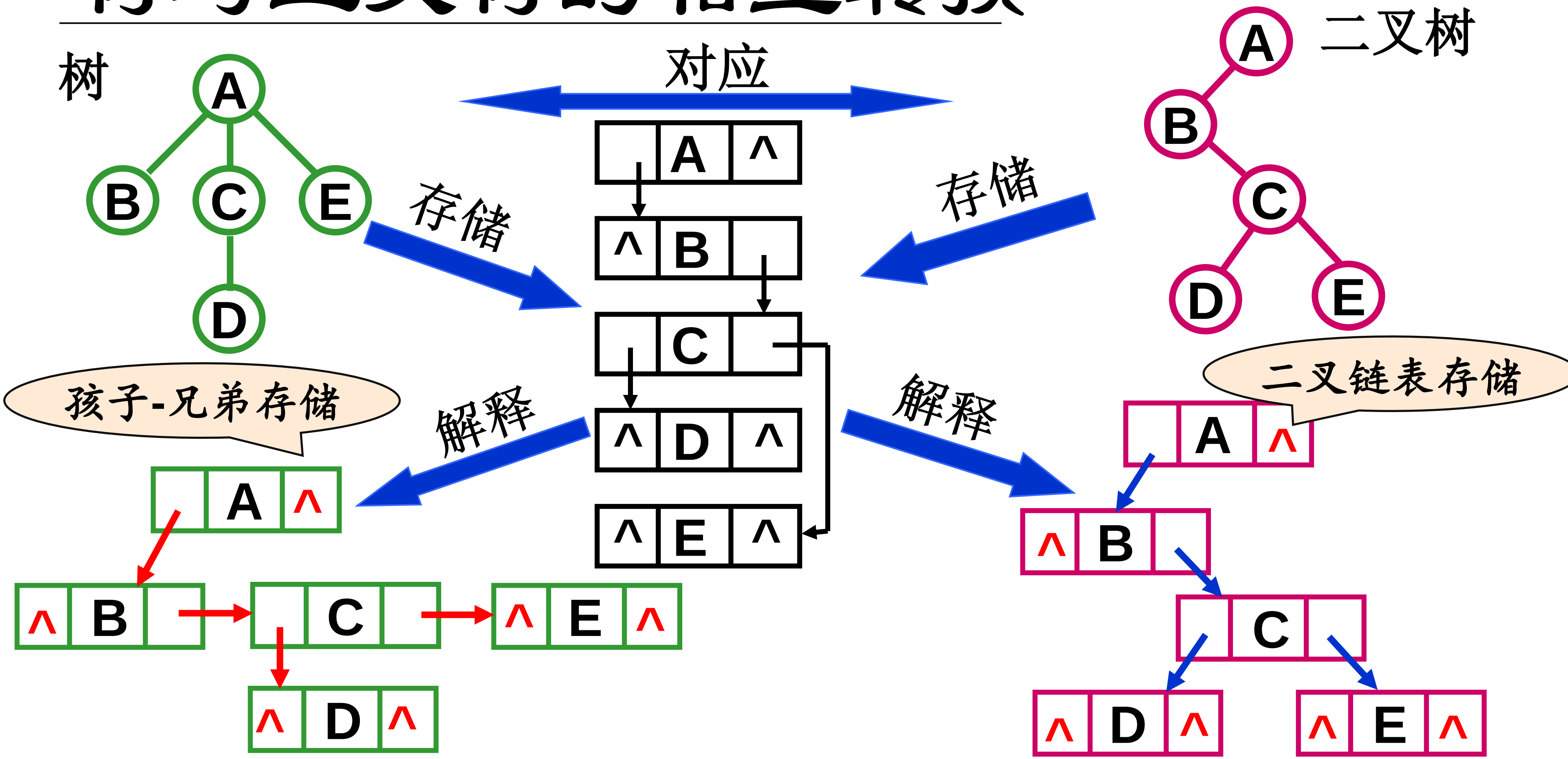


找孩子容易

找双亲容易☺

Back

树与二叉树的相互转换



树与二叉树的转换：

- 以二叉链表为对应关系，一棵树可与**唯一**的一棵二叉树对应。
- 用二叉树表示一棵树，可将对**复杂的树的操作**转换为对**简单的二叉树的操作**。

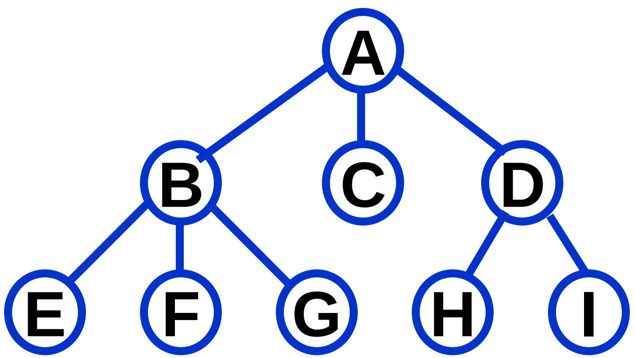
树与二叉树的相互转换

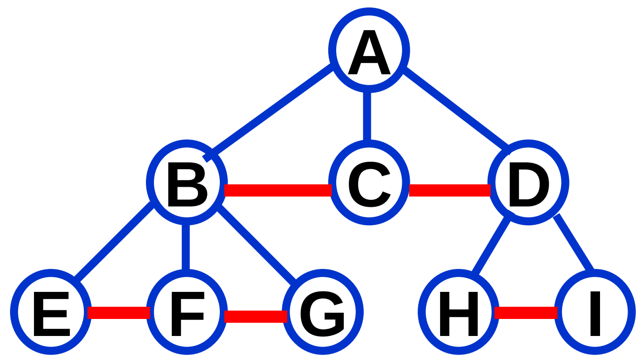
将树转换为二叉树

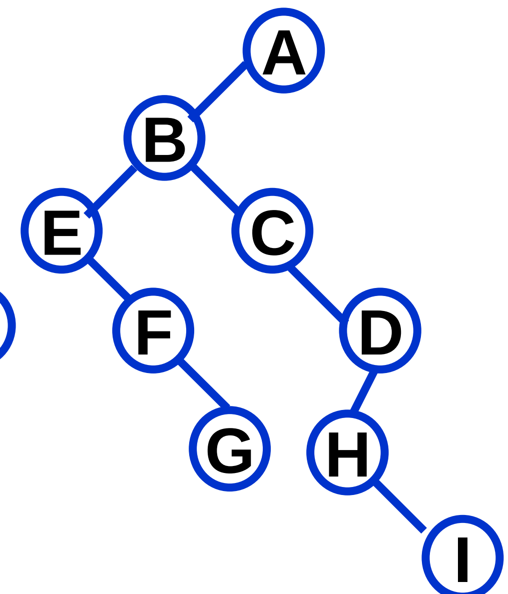
❶加线(连兄弟)：在兄弟之间加一连线；

❷抹线(断父子)：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系；

❸旋转：以树的根结点为轴心，将整树顺时针转45°。







为什么要"连兄弟、断父子"？

转换成二叉树以后，结点的下一个兄弟"变成"其右孩子

树转换得到的二叉树右子树一定为空

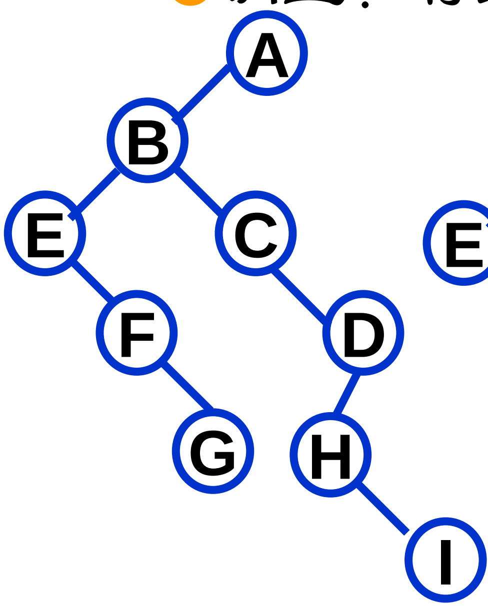
树与二叉树的相互转换

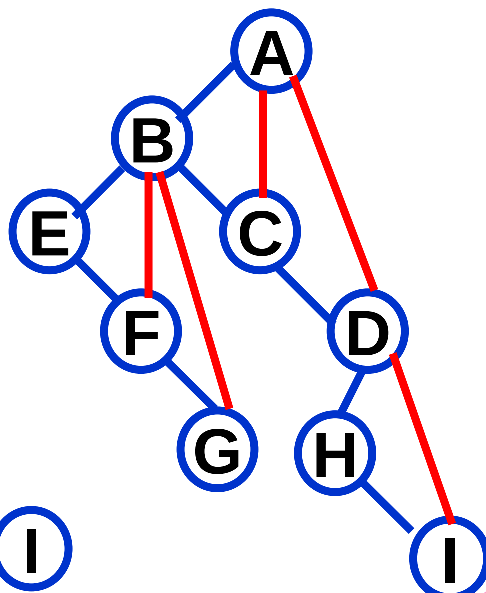
将二叉树还原成树

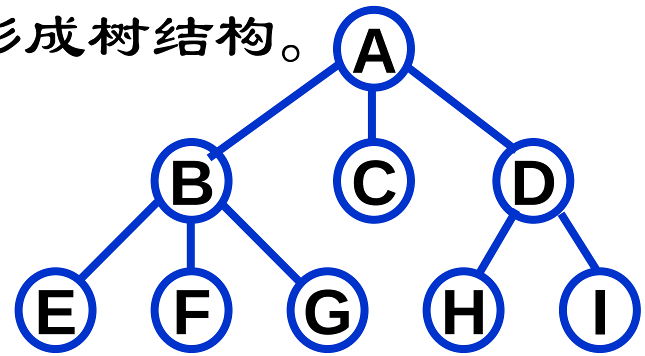
❶加线(连祖孙)：若p是双亲结点的左孩子，则将p的右孩子、右孩子的右孩子，…沿分支找到的所有右孩子，都与p的双亲用线连起来；

❷抹线(断父子)：抹掉原二叉树中双亲与右孩子之间的连线；

❸调整：将结点按层次排列，形成树结构。







为什么要"连祖孙、断父子"?

只能将根的右子树为空的二叉树还原成树

转换成树后，某结点的左孩子p的右孩子、右孩子的右孩子，……都"变成"p的兄弟

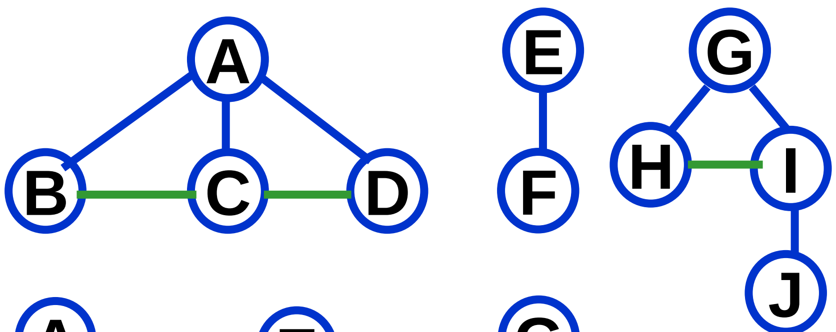
森林与二叉树的相互转换

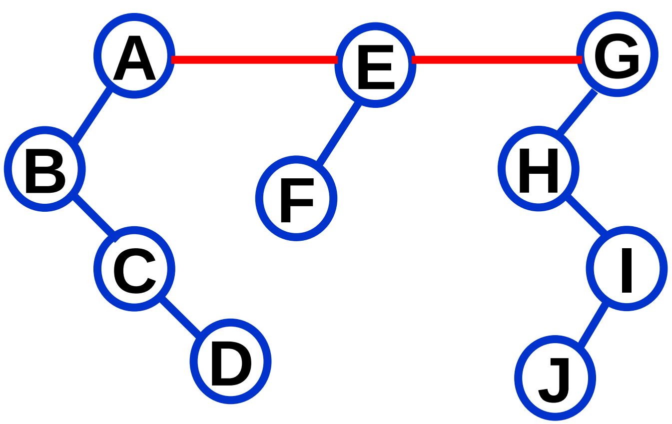
将森林树转换为二叉树

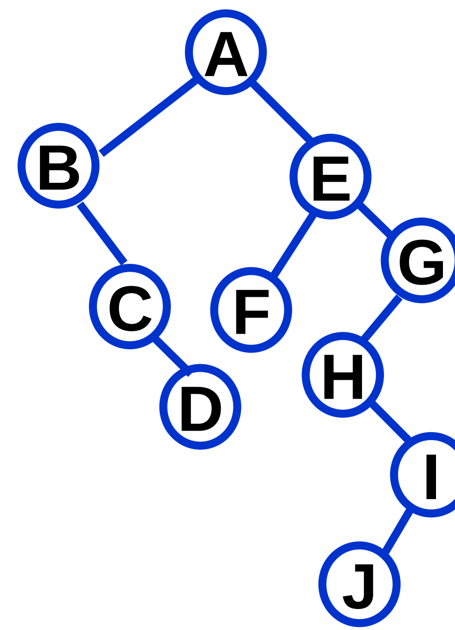
❶将**各棵树**分别转换成二叉树；

❷将每棵树的根结点**用线相连**；

❸以第一棵树的根结点为二叉树的根，再以根结点为轴心，顺时针**旋转**，构成二叉树的结构。







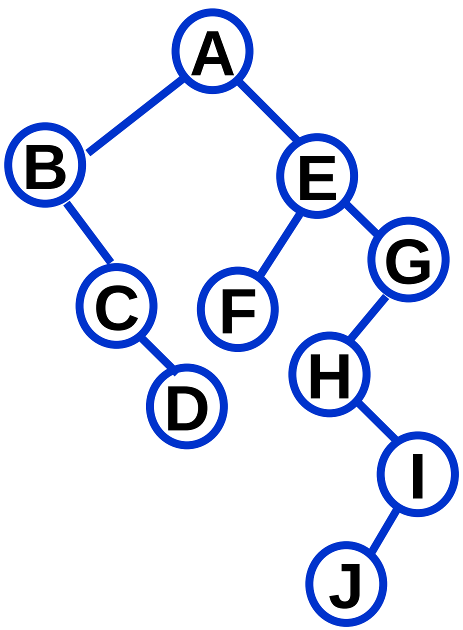
森林转换成的二叉树**右子树非空**

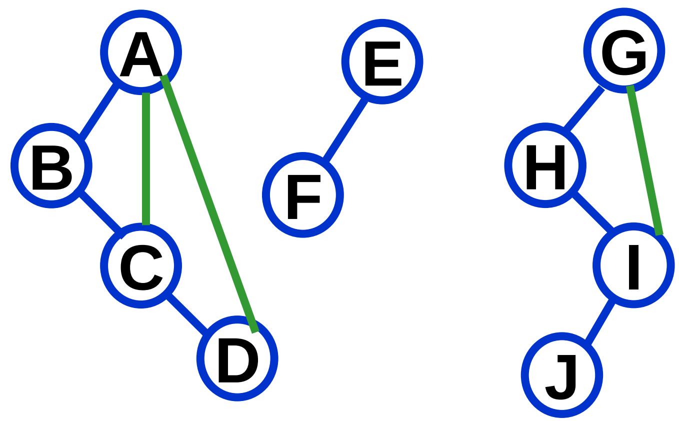
森林与二叉树的相互转换

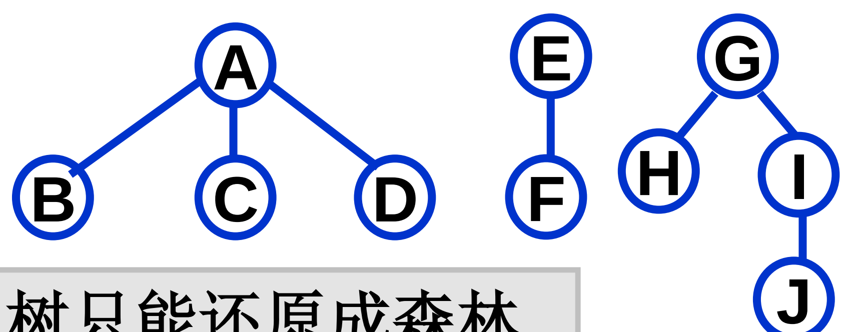
将二叉树还原成森林

❶抹线：将二叉树中根结点与其右孩子连线，及沿右分支找到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树；

❷还原：将孤立的二叉树分别还原成树。







右子树不为空的二叉树只能还原成森林

Back

树型结构的遍历

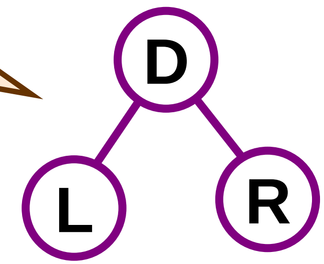
遍历——按一定规律访问树型结构的各个结点，且每个结点只访问一次。

即找到一个完整而有规律的走法，得到所有结点的线性排列；

遍历的关键在于访问结点的次序。

二叉树的遍历

DLR、LDR、LRD
DRL、RDL、RLD



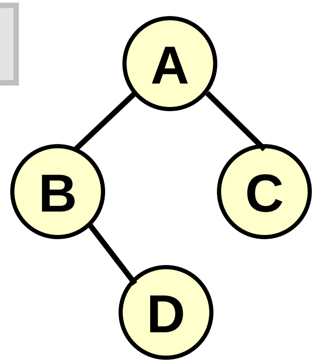
三个子任务——

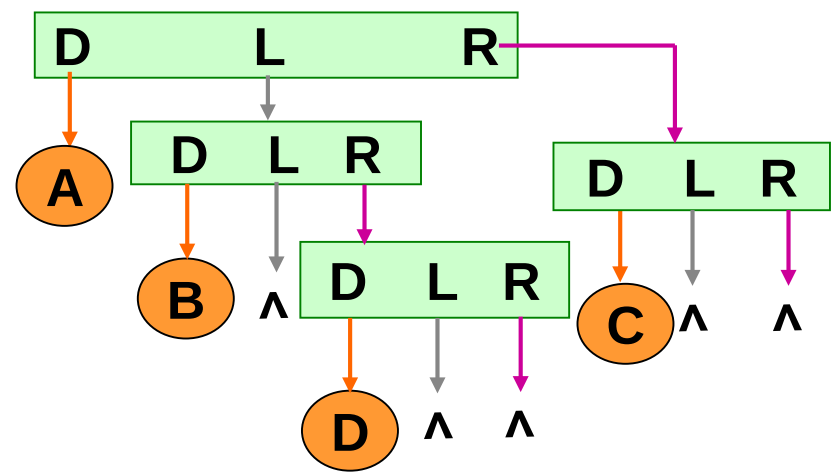
访问根结点(D)、遍历左子树(L)、遍历右子树(R)

常用遍历方法

❶ 先序遍历：先访问根结点，再分别先序遍历左、右子树

DLR



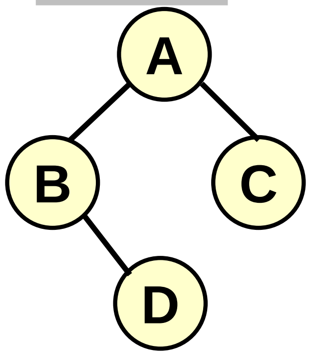


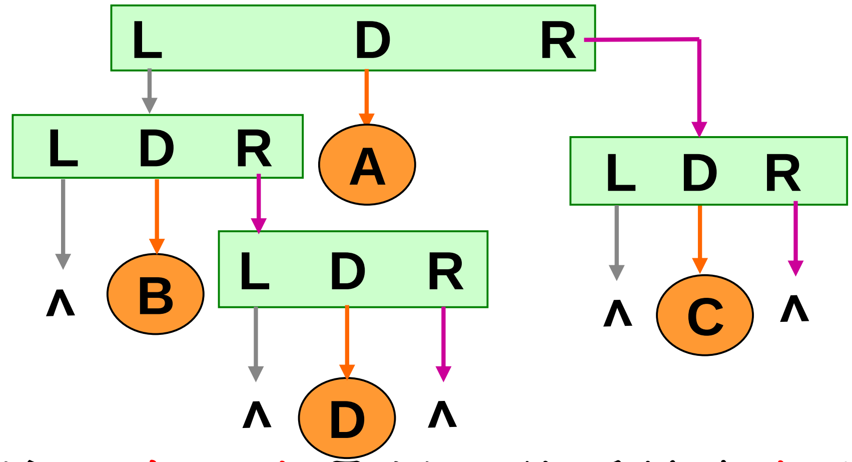
先序遍历序列：ABDC

树型结构的遍历

②中序遍历：先中序遍历左子树，然后访问根结点，最后中序遍历右子树

LDR

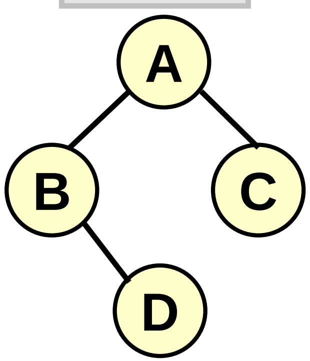


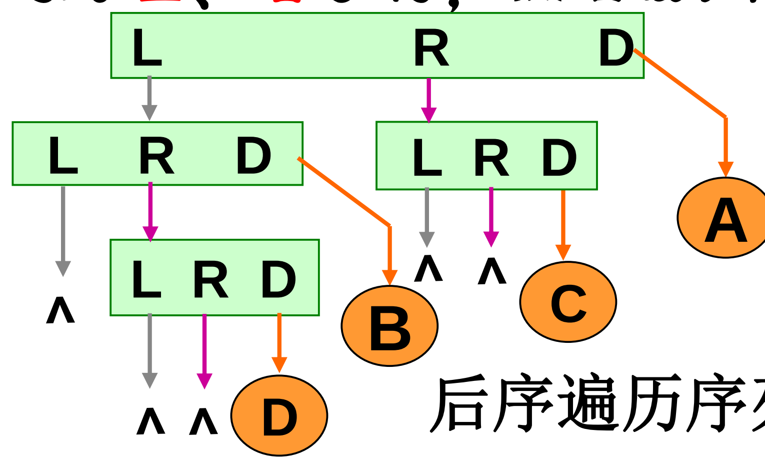


中序遍历序列：BDAC

③后序遍历：先后序遍历左、右子树，然后访问根结点

LRD



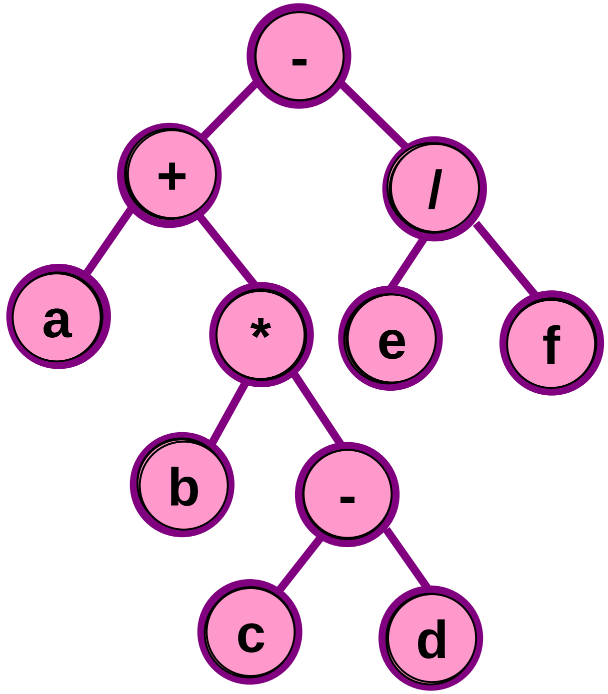


后序遍历序列：DBCA

④按层次遍历：从上到下、从左到右访问各结点

树型结构的遍历

二叉树的遍历



先序遍历：- + a * b - c d / e f

中序遍历：a + b * c - d - e / f

后序遍历：a b c d - * + e f / -

层次遍历：- + / a * e f b - c d

树型结构的遍历

二叉树的递归遍历算法

输出根结点

在左、右子树上进行*相同的遍历*

先序遍历	根	左子树	右子树
中序遍历	左子树	根	右子树
后序遍历	左子树	右子树	根

✓ 因此，上述遍历均是"递归"的过程。

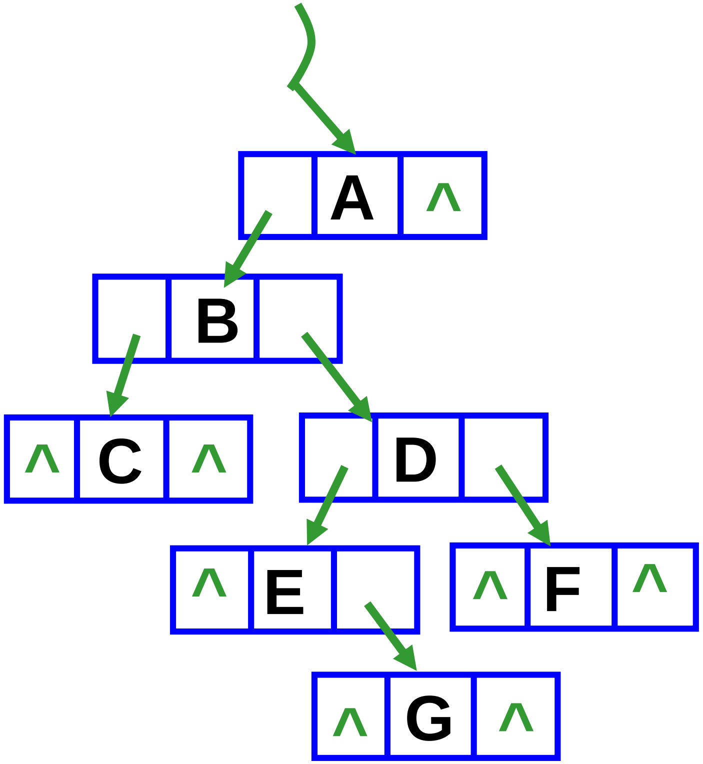
应选择哪种存储方法?

遍历过程中进行的主要操作——找结点的左、右孩子

∴应选择二叉链表存储方法

树型结构的遍历

二叉树的递归遍历算法




```
void  PreOrderTraverse(BiTree T)
{    if(T)
     {  printf("%d\t",T->data);
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
     }
}
```

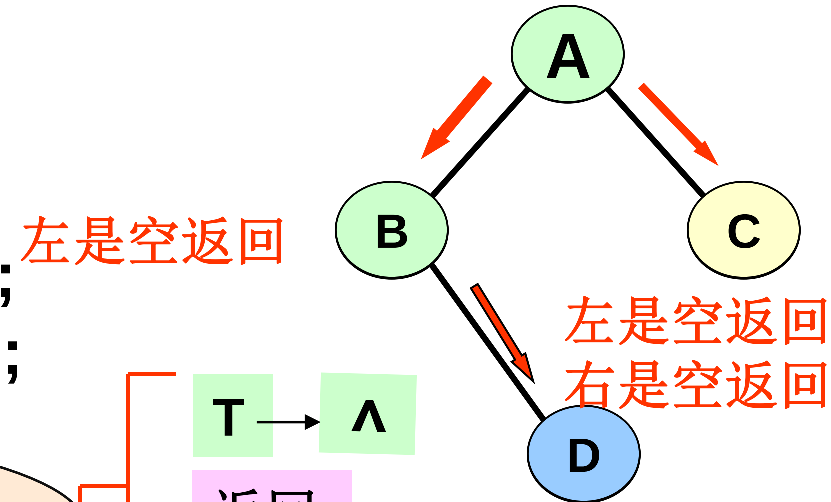
根

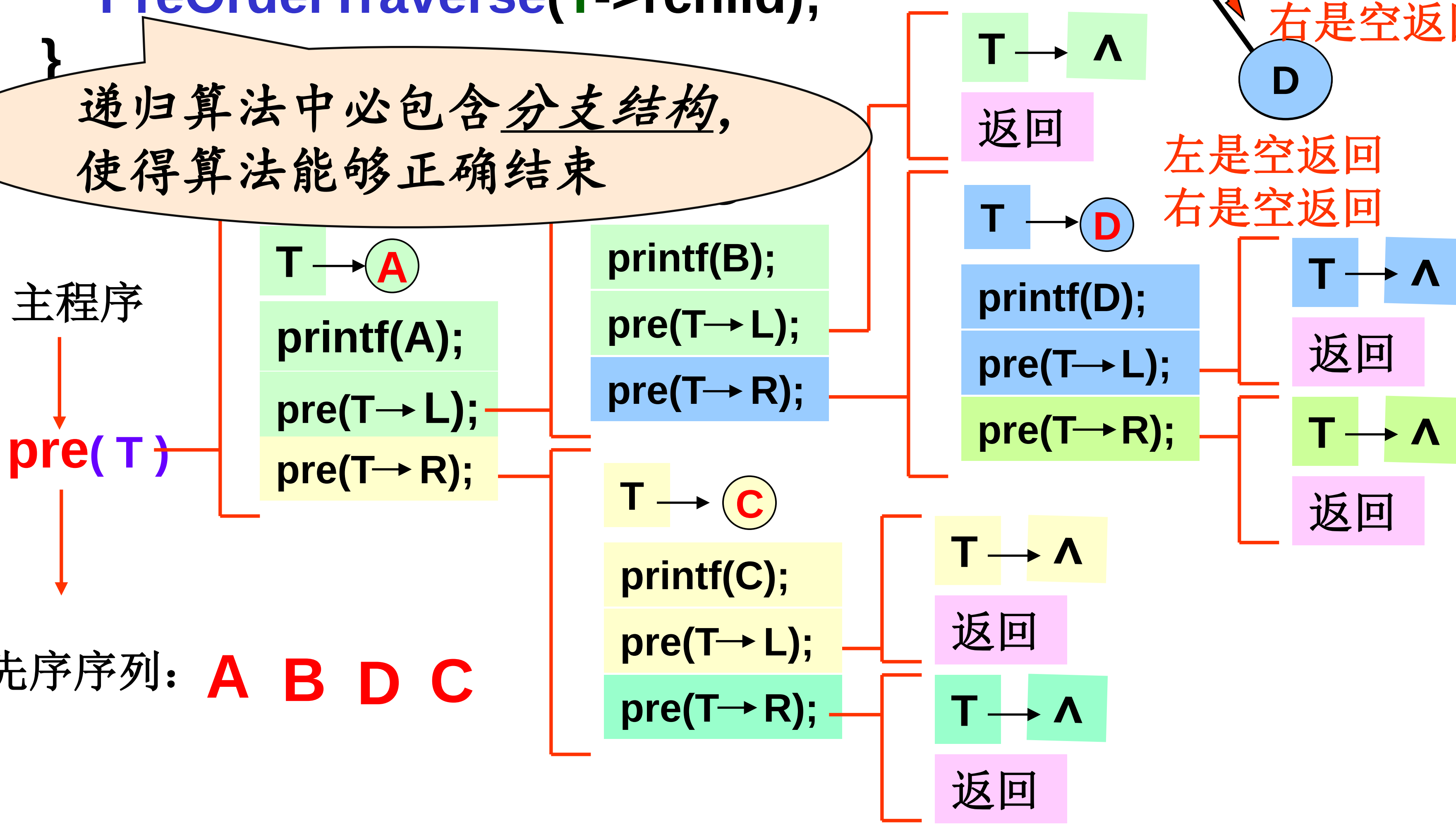
左

右

```
void  PreOrderTraverse(BiTree T)
{   if(T)
    {  printf("%d\t",T->data);
       PreOrderTraverse(T->lchild);
       PreOrderTraverse(T->rchild);
    }
}
```

递归算法中必包含*分支结构*，
使得算法能够正确结束

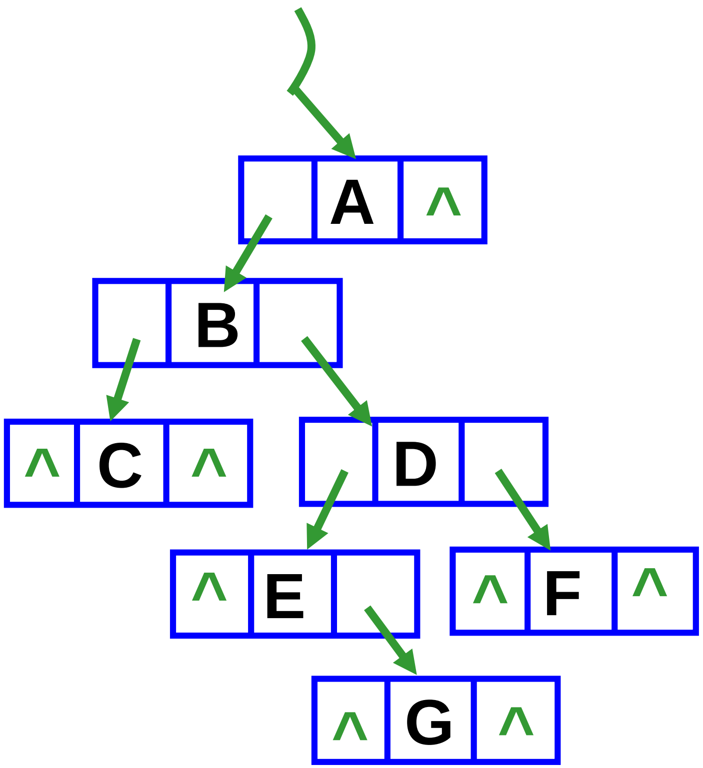




先序序列：A B D C

树型结构的遍历

二叉树的递归遍历算法

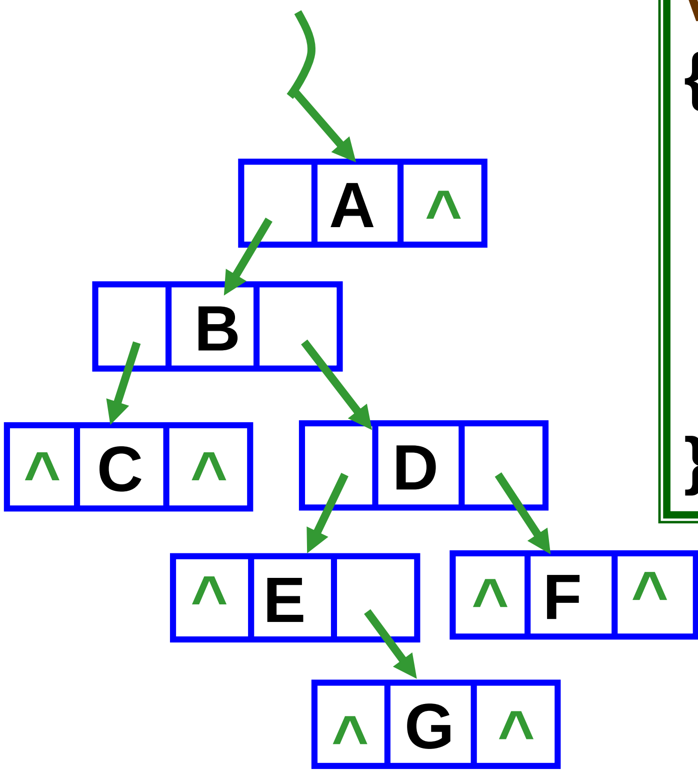



```
void  PreOrderTraverse(BiTree T)
{    if(T)
     {  printf("%d\t",T->data);          根
        PreOrderTraverse(T->lchild);     左
        PreOrderTraverse(T->rchild);     右
     }
}
```

```
void  InOrderTraverse(BiTree T)
{    if(T)
     {  InOrderTraverse(T->lchild);      左
        printf("%d\t",T->data);          根
        InOrderTraverse(T->rchild);      右
     }
}
```

树型结构的遍历

二叉树的递归遍历算法




```
void  PostOrderTraverse(BiTree T)
{    if(T)
     {  PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        printf("%d\t",T->data);
     }
}
```

左
右
根

可修改该语句，实现对每个结点的操作

遍历是二叉树各种操作的基础，可在*遍历过程中*对结点进行各种操作。

若能将某问题分解到二叉树的左、右子树上进行*相同操作*，则可将*递归*遍历算法进行适当改写来实现。

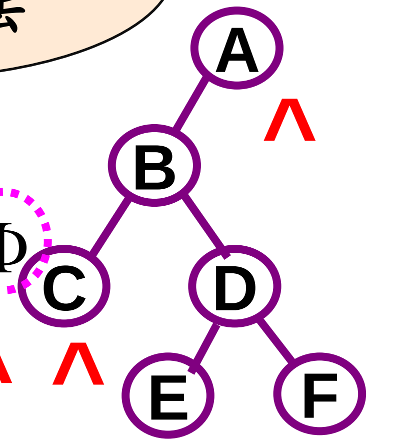
树型结构的遍历

改写先序递归遍历算法

二叉树递归遍历算法的应用

按先序遍历序列建立二叉树的二叉链表

先序序列：A B C Φ Φ D E Φ Φ F Φ Φ Φ



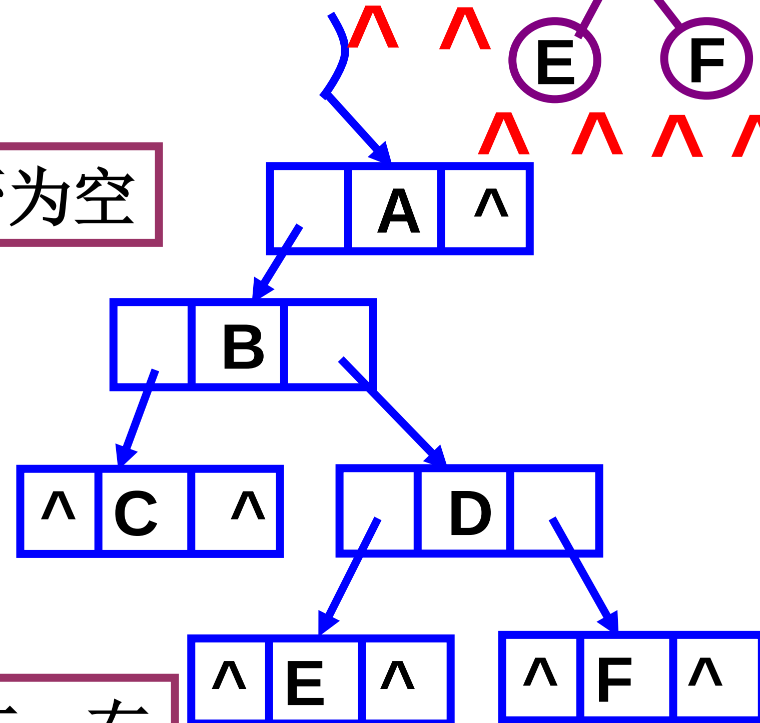

```
void  PreOrderTraverse(BiTree T)
{   ……
    if(T)
    {  printf("%d\t",T->data);
       PreOrderTraverse(T->lchild);
       PreOrderTraverse(T->rchild);
    }
}
```

接收一个字符

判断字符是否为空

字符不为空，则建立新结点并为数据域赋值

再建立以其左、右孩子为根的子树



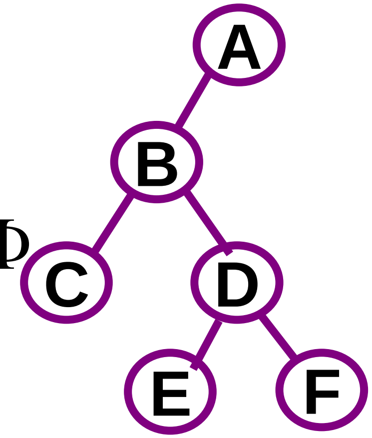
如何改写？

树型结构的遍历

二叉树递归遍历算法的应用

按先序遍历序列建立二叉树的二叉链表

先序序列： A B C Φ Φ D E Φ Φ F Φ Φ Φ



```
Status CreateBiTree(BiTree &T)
{  char ch;
   scanf("%c",&ch);
   if (ch==' ')   T = NULL;
   else
   {  T = (BiTNode *)malloc(sizeof(BiTNode));
      T->data = ch;
      CreateBiTree(T->lchild);
      CreateBiTree(T->rchild);
    }
    return OK;
}
```

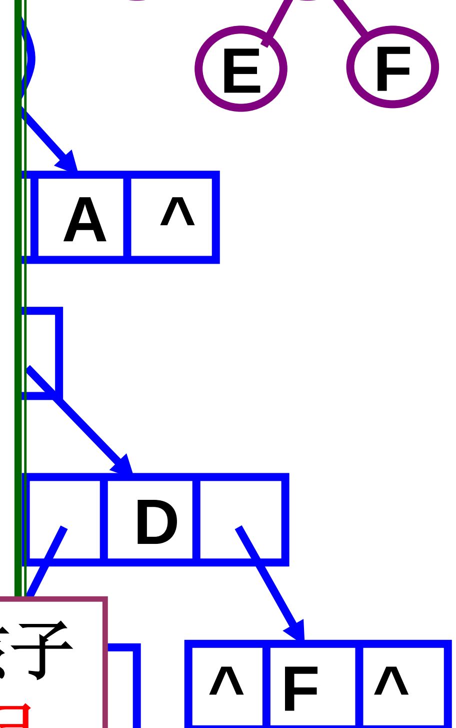
定义变量

接收字符

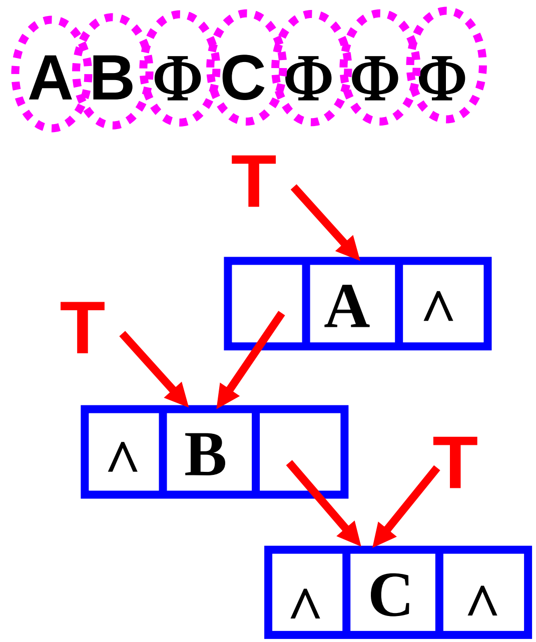
若为空格，则树的根为空

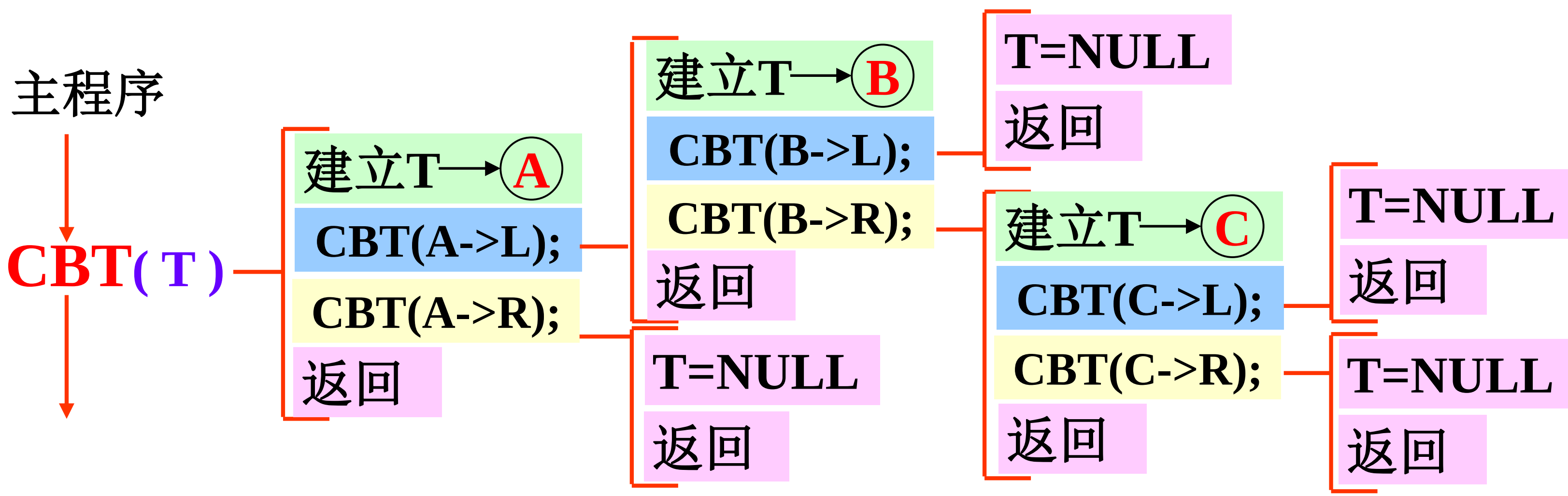
建立结点并赋值

建立以T左、右孩子为根的子树，递归

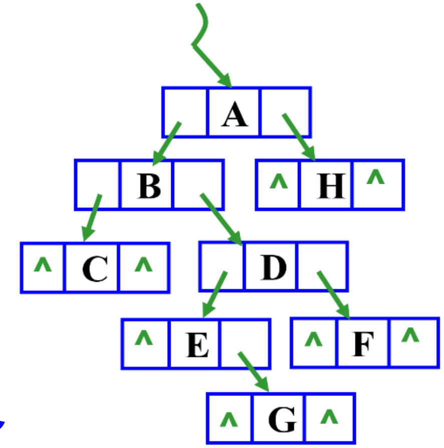


```
Status CreateBiTree(BiTree &T)
{  char ch;    scanf("%c",&ch);
   if (ch=='  ')   T = NULL;
   else
   { T = (BiTNode *)malloc(…));
     T->data = ch;
     CreateBiTree(T->lchild);
     CreateBiTree(T->rchild);    }
   return OK;     }
```





树型结构的遍历



二叉树递归遍历算法的应用

统计二叉树中的叶子结点总数

——左子树叶子数+右子树叶子数

问题分析——

局部静态变量、引用型参数

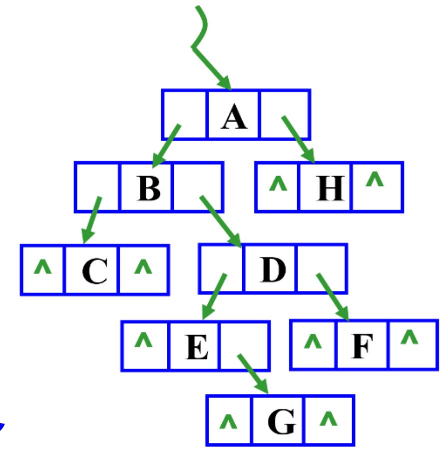
遍历过程中，若某结点为叶子，则原统计值count++。

∴要求count在递归过程中始终占用存储空间，保存最新值。

```
int LeafCount1 ( BiTree T)
{  static int count;
   if ( T )
   {    if ((T->lchild==NULL)&& (T->rchild==NULL))
                count++;
        else {   count=LeafCount1( T->lchild );
                 count=LeafCount1( T->rchild );
             }
   }
   return count;
}
```

可使用中序和后序?

树型结构的遍历



二叉树递归遍历算法的应用

统计二叉树中的叶子结点总数

——左子树叶子数+右子树叶子数

问题分析——

局部静态变量、引用型参数

遍历过程中，若某结点为叶子，则原统计值count++。

∴要求count在递归过程中始终占用存储空间，保存最新值。

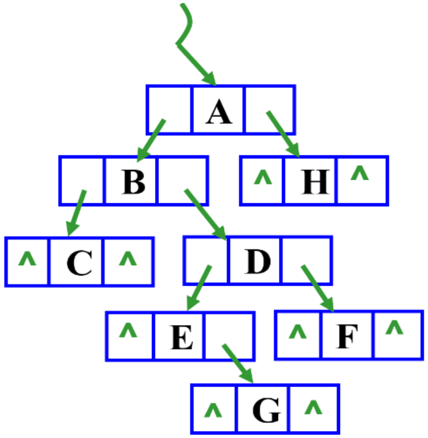
```
int LeafCount2 ( BiTree T, int &count )
{
    if ( T )
    {   if ((T->lchild==NULL)&& (T->rchild==NULL))
                count++;
        else {   LeafCount2( T->lchild, count);
                 LeafCount2( T->rchild, count);
             }
    }
}
```

树型结构的遍历

二叉树递归遍历算法的应用

求二叉树的深度

——max{左子树深度+1,右子树深度+1}



问题分析——

遍历过程中，先统计出子树深度，再求整棵树的深度。

∴只能改写后序遍历算法。

```
int Depth( BiTree T)
{  int depth, depthleft, depthright;
   if (T==NULL)    depth=0;
   else{  depthleft = Depth( T->lchild );
          depthright = Depth( T->rchild );
          depth= depthleft>depthright?depthleft +1: depthright+1;
       }
    return depth;
}
```

T为空时，以T为根的树深度为0

树型结构的遍历

二叉树的遍历

根据遍历序列，构造二叉树

先序遍历 根 左子树 右子树

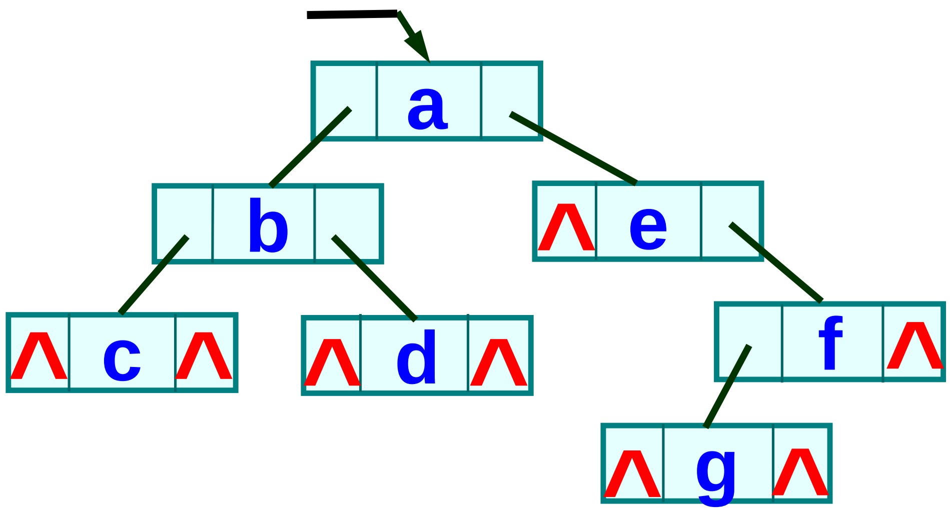
中序遍历 左子树 根 右子树

后序遍历 左子树 右子树 根

层次遍历 根 第二层 第三层 …

例 已知先序序列 a b c d e f g

中序序列 c b d a e g f



树型结构的遍历

二叉树的遍历

根据遍历序列，构造二叉树

先序遍历 根 左子树 右子树

中序遍历 左子树 根 右子树

后序遍历 左子树 右子树 根

层次遍历 根 第二层 第三层 …

☺ 已知先序+中序序列，能构造唯一的二叉树。

已知其它的序列组合呢?

😐 已知中序+后序序列，能构造唯一的二叉树?

😐 已知先序+后序序列，能构造唯一的二叉树?

😐 已知中序+层次序列，能构造唯一的二叉树?

树型结构的遍历

二叉树的遍历

根据遍历序列，构造二叉树

先序遍历 根 左子树 右子树

中序遍历 左子树 根 右子树

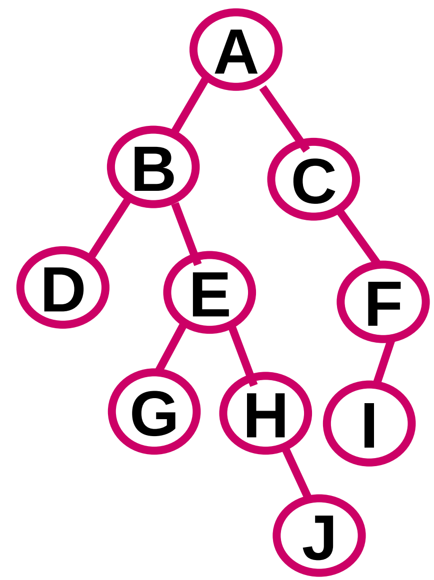
后序遍历 左子树 右子树 根

层次遍历 根 第二层 第三层 ...

已知中序+层次序列，能构造唯一的二叉树？

例 已知层次序列 A B C D E F G H I J

中序序列 D B G E H J A C I F



树型结构的遍历

二叉树的层次遍历非递归算法

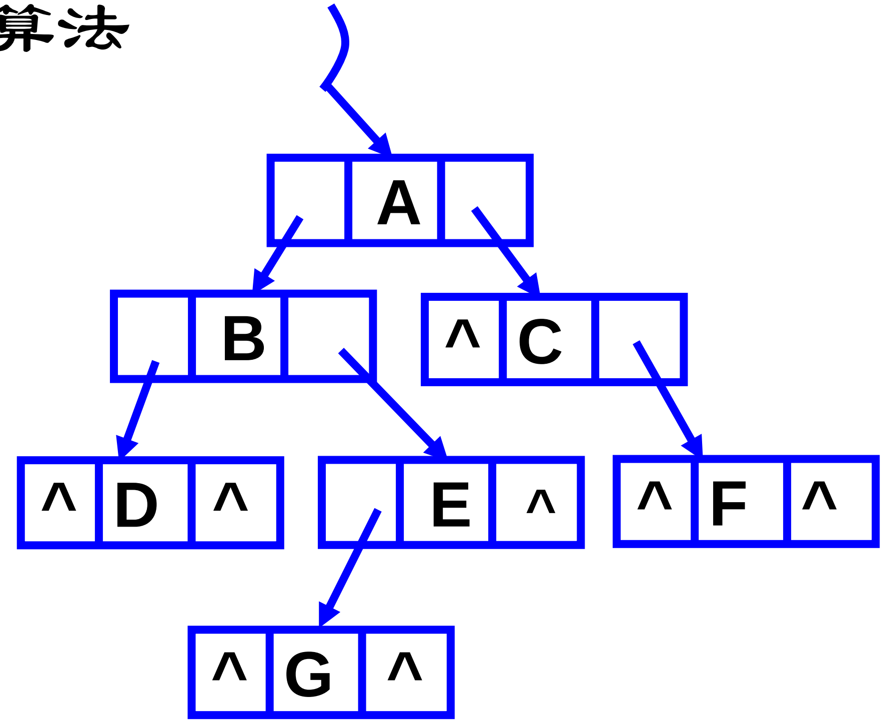
★算法思想：

利用队列实现二叉树的层次遍历非递归算法

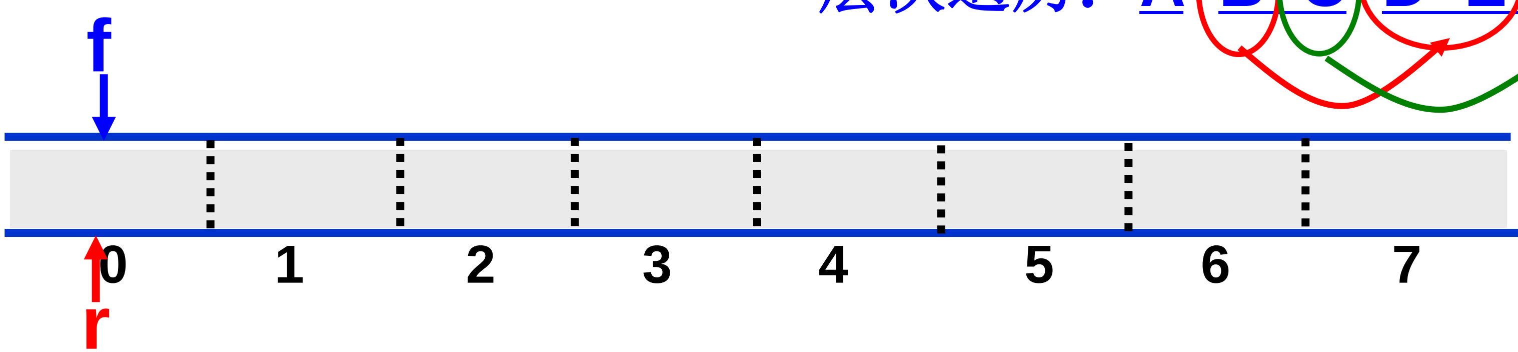
❶将二叉树的根结点入队

❷队头元素出队并访问，将其非空左、右孩子入队（即以从左向右的顺序将下一层结点保存在队列中）

❸重复❷直到队空为止



层次遍历：A B C D E F G



树型结构的遍历

二叉树的层次遍历非递归算法

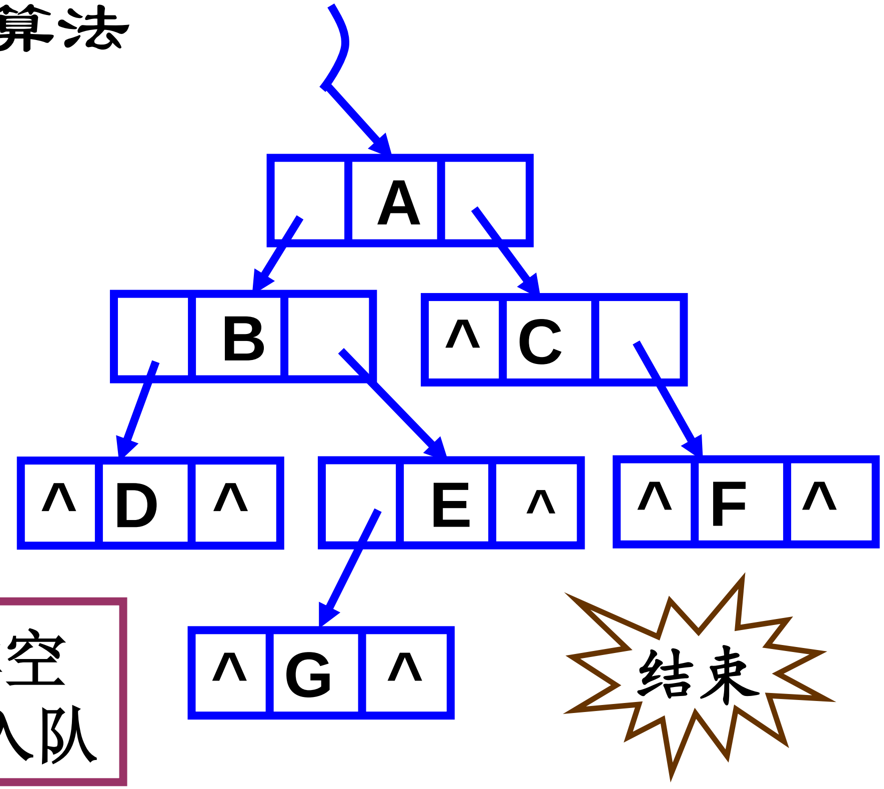
```
void  LevelTrave(BiTree T)
{   int  f=0,r=0;
    BiTree p, q[M];
    q[r++]=T;
    while(f<r)
    {    p=q[f++];
         printf("%c\t",p->data);
         if(p->lchild)
              q[r++]=p->lchild;
         if(p->rchild)
              q[r++]=p->rchild;
     }
}
```

根结点入队

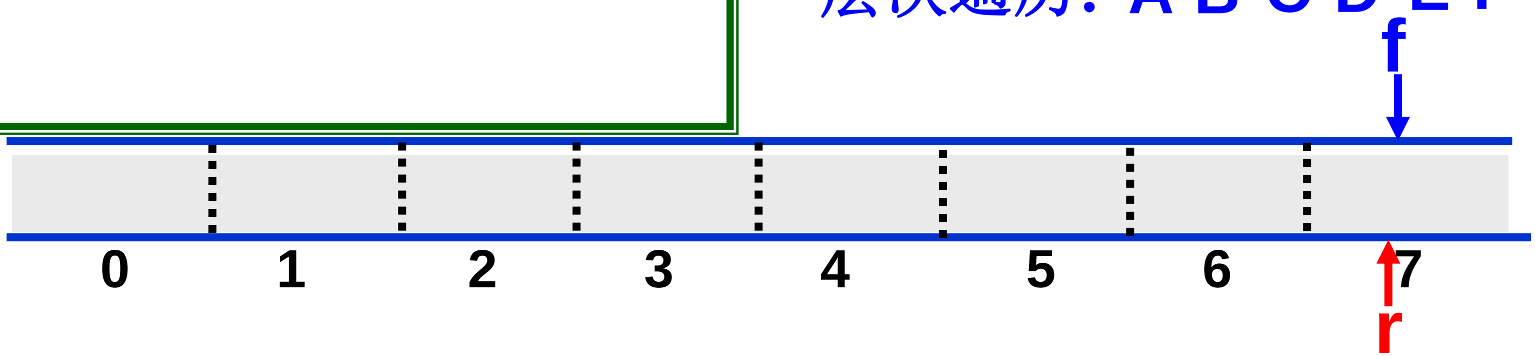
当队列不为空时

出队

将非空孩子入队



层次遍历：A B C D E F G



树型结构的遍历

树的遍历

常用遍历方法

❶ **先根遍历**：访问树的根结点，然后依次先根遍历根的每棵子树

根 | 先根遍历最左边一棵子树 | 先根遍历下一棵子树 | ……

❷ **后根遍历**：先依次后根遍历每棵子树，然后访问根结点

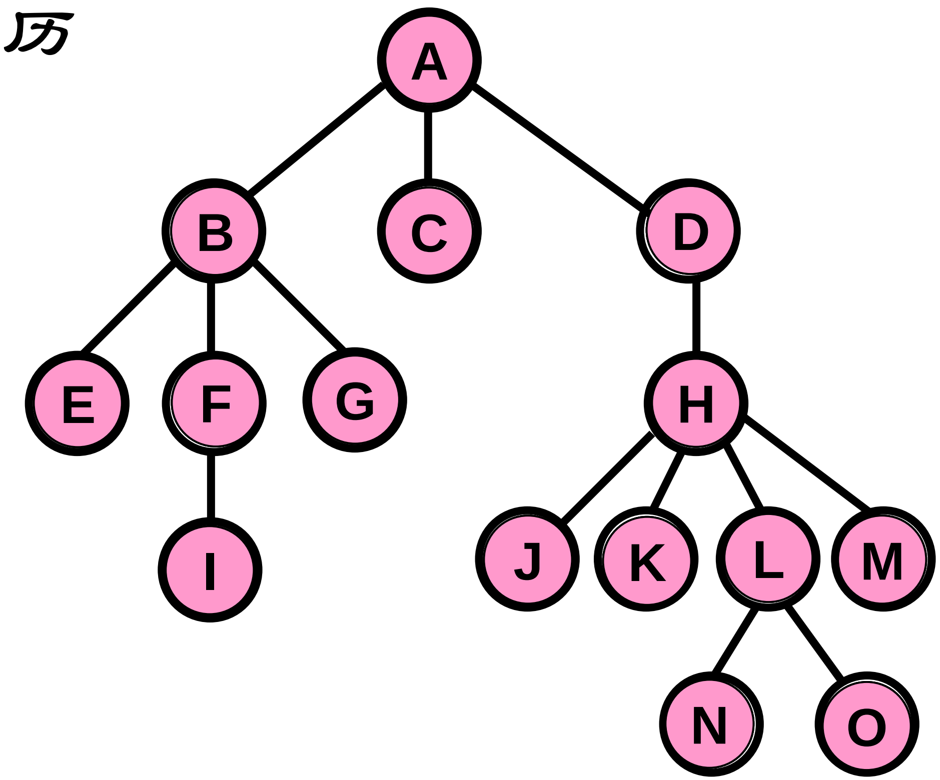
后根遍历最左边一棵子树 | 后根遍历下一棵子树 | …… | 根

❸ **按层次遍历**：先访问根结点，然后依次遍历第二层、……、第n层的结点

根 | 第二层 | 第三层 | ……

树型结构的遍历

树的遍历



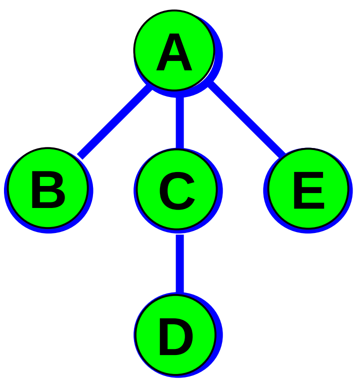
先根遍历：ABEFI GCDHJ KLNOM

后根遍历：EI FGBCJKNOLMHDA

层次遍历：ABCDEFGHI JKLMNO

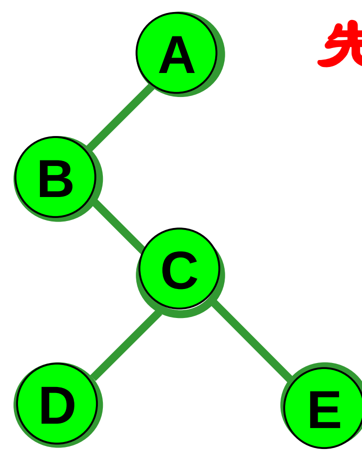
树型结构的遍历

- 树的递归遍历算法
  - 二叉树的先序、中序、后序遍历的递归算法较为简单。
  - 使用孩子-兄弟表示法作为树的存储结构，通过对应二叉树的遍历实现树的遍历。



先根： ABCDE

后根：BDCEA



先序：ABCDE

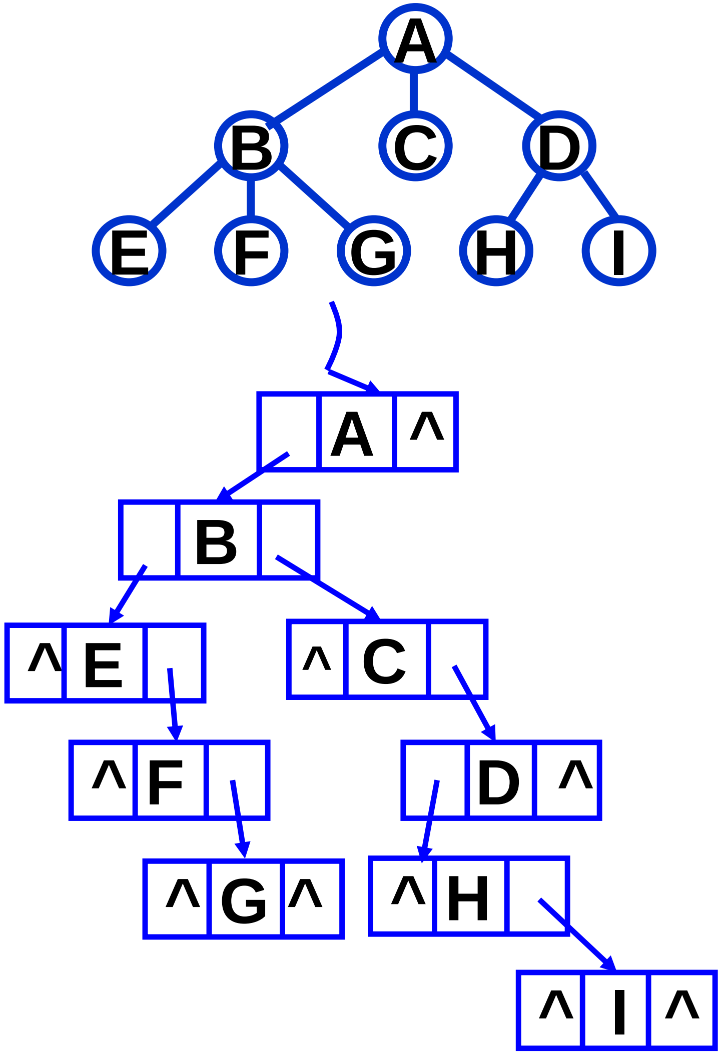
中序：BDCEA

先根遍历树 ⟷ 先序遍历对应的二叉树

后根遍历树 ⟷ 中序遍历对应的二叉树

树型结构的遍历

- 树的递归遍历算法的应用
  - 求树的深度



如何求树的深度?

树的深度= 最大子树深度+1

递归

适合选择哪种存储结构?

适于选用孩子-兄弟表示法

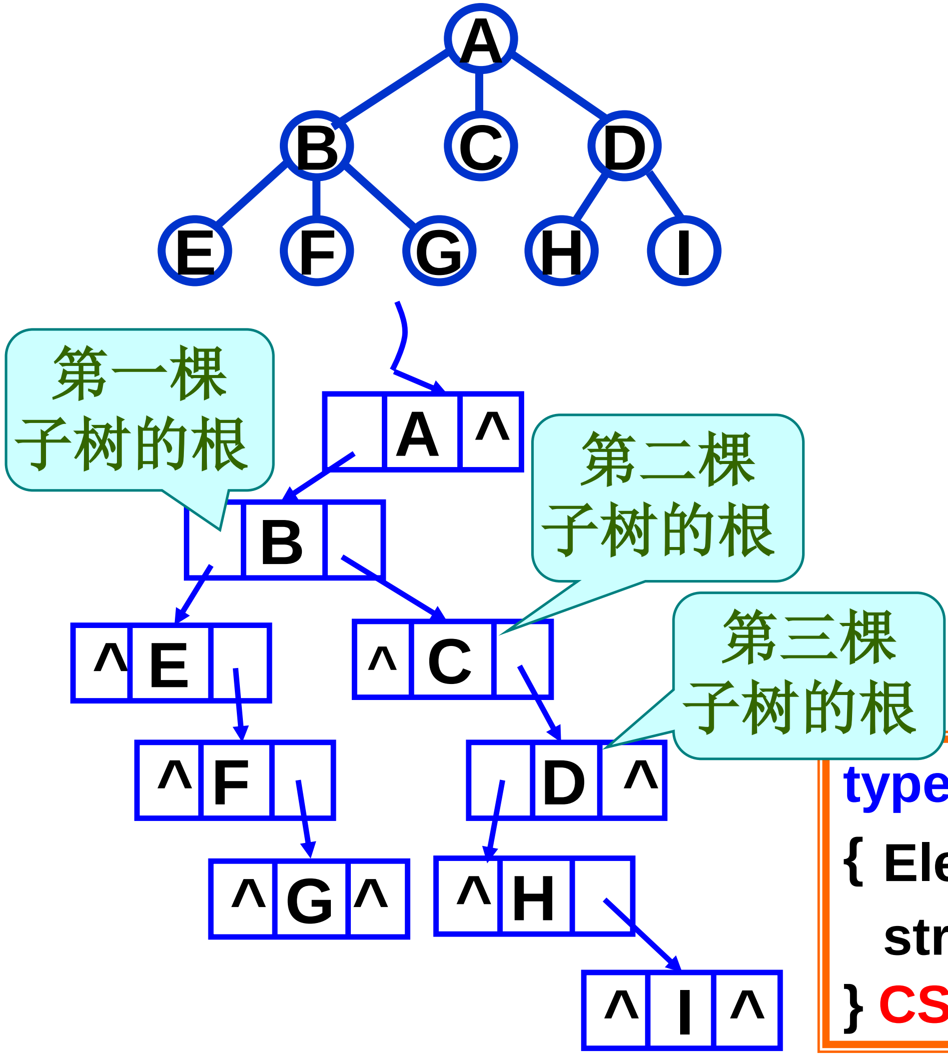
将问题转化为对二叉树的操作，且易于实现递归

```
typedef struct CSnode
{ ElemType   data;
  struct CSnode   *firstchild, *nextsibling;
} CSNode, *CSTree;
```

树型结构的遍历

树的递归遍历算法的应用

求树的深度



int CSTreeDepth(CSTree T)

1. 设depth为T的最大子树深度，初始为0；
2. 若T=NULL，则返回0；
3. 否则p=T->firstchild
4. p不为空，则求以p为根的子树深度d（递归）
5. 若d>depth，则depth=d
6. p=p->nextsibling
7. 返回depth+1

```
typedef struct CSnode
{ ElemType   data;
  struct CSnode   *firstchild, *nextsibling;
} CSNode, *CSTree;
```

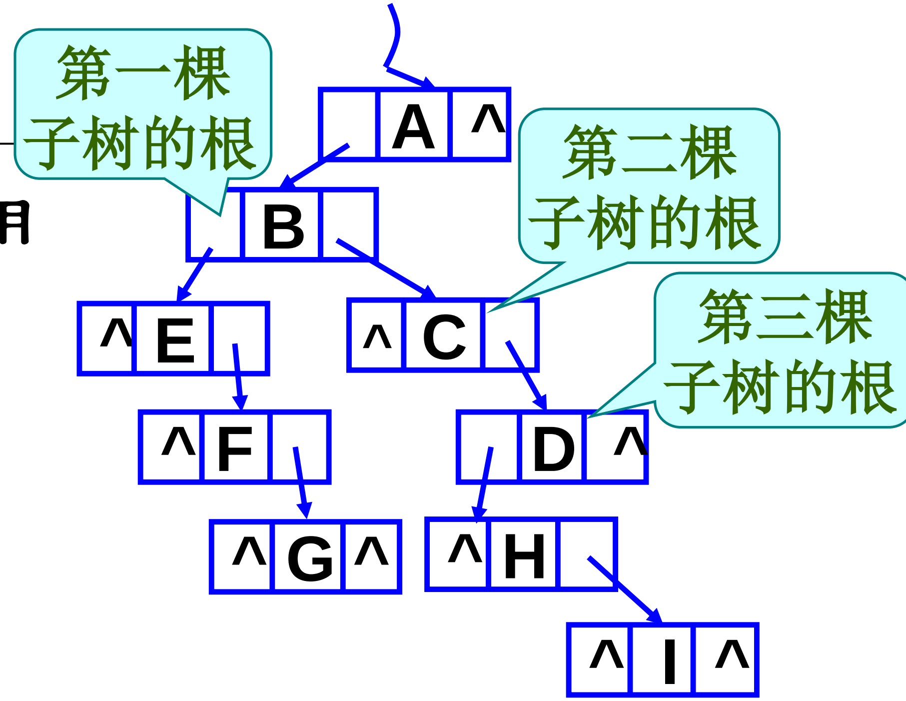
树型结构的遍历

树的递归遍历算法的应用

求树的深度

```
int CSTreeDepth(CSTree T)
{ int depth = 0, d;
   CSTree p;
   if (!T) return 0;
   p = T->firstchild;
   while(p)
   { d=CSTreeDepth(p);
     if (d > depth)
         depth = d;
     p = p->nextsibling;
    }
    return depth + 1;
}
```

空树深度为0



1.设depth为T的最大子树深度，初始为0；
2.若T=NULL，则返回0；
3. 否则p=T->firstchild
4. p不为空，则求以p为根的子树深度d（递归）
5.若d>depth，则depth=d
6.p=p->nextsibling
7.返回depth+1

树型结构的遍历

- 森林的遍历
  - 常用遍历方法

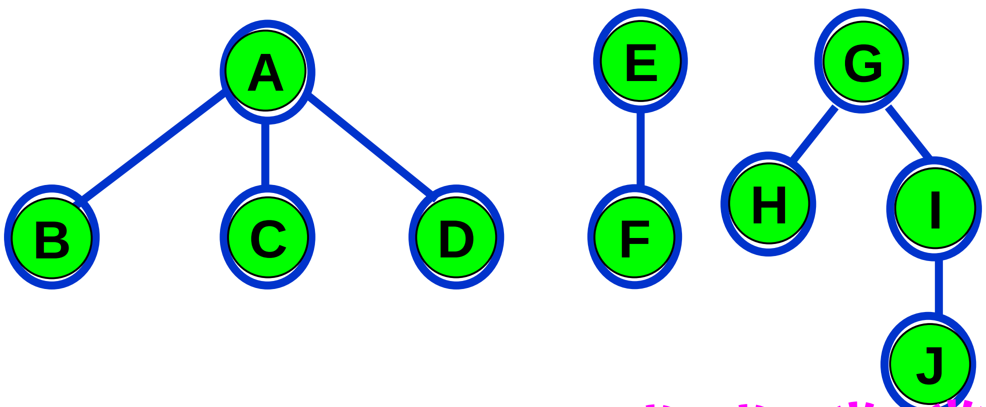
❶ 先序遍历：若森林不空，则

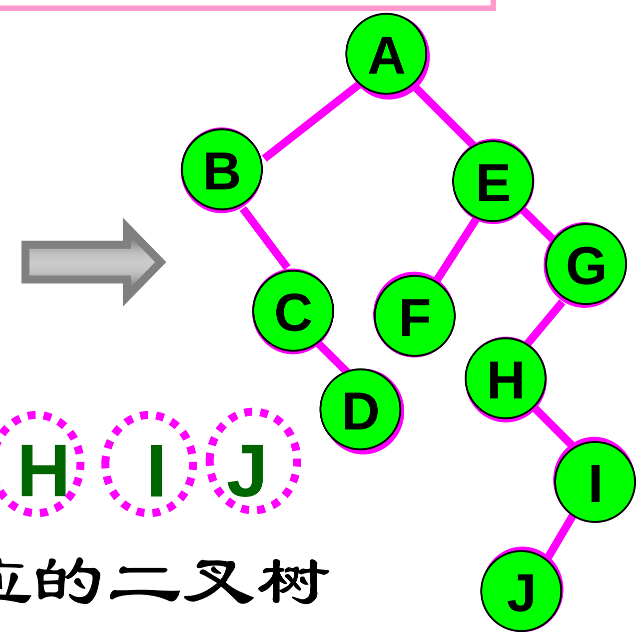
访问森林中第一棵树的根结点；

先序遍历森林中第一棵树的子树森林；

先序遍历森林中其余树构成的森林。

即：从左至右先根遍历森林中每一棵树。





先序遍历序列：A B C D E F G H I J

先序遍历森林 ⟷ 先序遍历对应的二叉树

树型结构的遍历

- 森林的遍历
  - 常用遍历方法

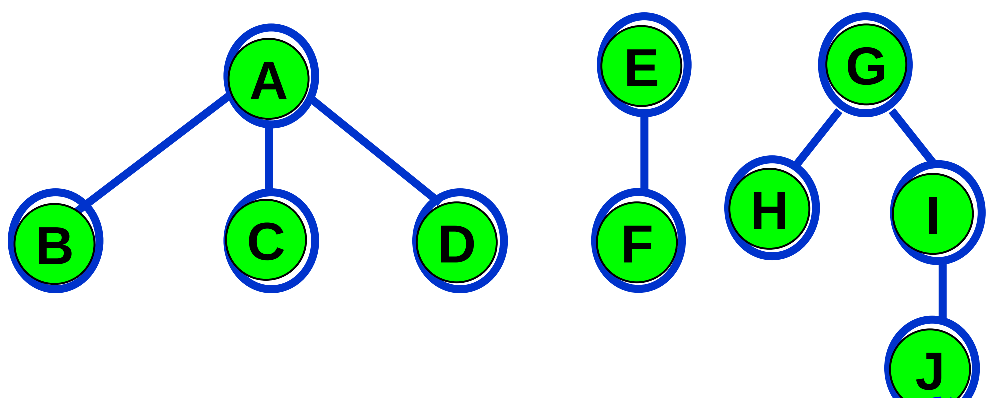
❷ 中序遍历：若森林不空，则

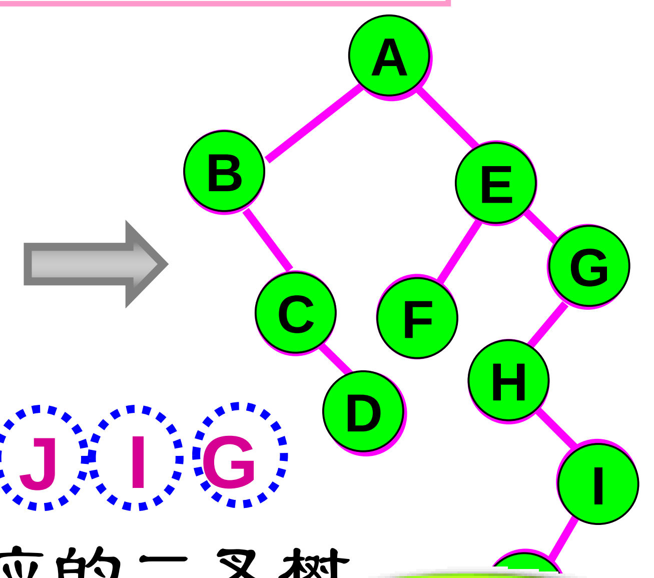
中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中其余树构成的森林。

即：从左至右后根遍历森林中每一棵树。





中序遍历序列：B C D A F E H J I G

中序遍历森林 ⟷ 中序遍历对应的二叉树

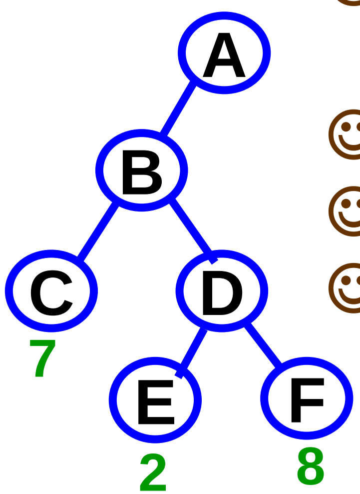
Back

二叉树的应用

哈夫曼(Huffman)树——带权路径长度最短的树

定义

- 路径：树中一个结点到另一个结点的分支构成这两个结点间的路径
- 路径长度：路径上的分支数
- 树的路径长度：从树根到各结点的路径长度之和
- 树的带权路径长度：树中所有带权结点的路径长度之和



记作：$wpl = \sum_{k=1}^{n} w_k l_k$

其中：w_k—权值

l_k—结点到根的路径长度

带权路径长度wpl——

Weighted **P**ath **L**ength

- Huffman树——设有n个权值$\{w_1, w_2, \ldots, w_n\}$，构造一棵有n个叶子结点的二叉树，每个叶子的权值为w_i，则wpl最小的二叉树叫~

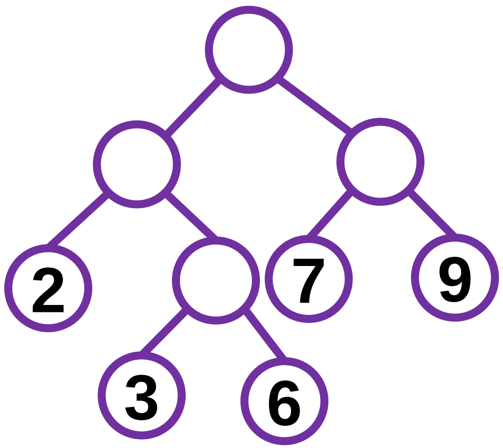
二叉树的应用

哈夫曼(Huffman)树——带权路径长度最短的树

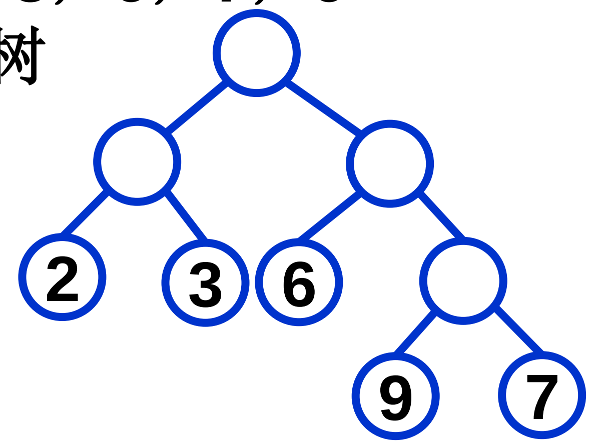
例　有5个结点，权值分别为2，3，6，7，9

构造有5个叶子结点的二叉树

$$wpl = \sum_{k=1}^{n} w_k l_k$$

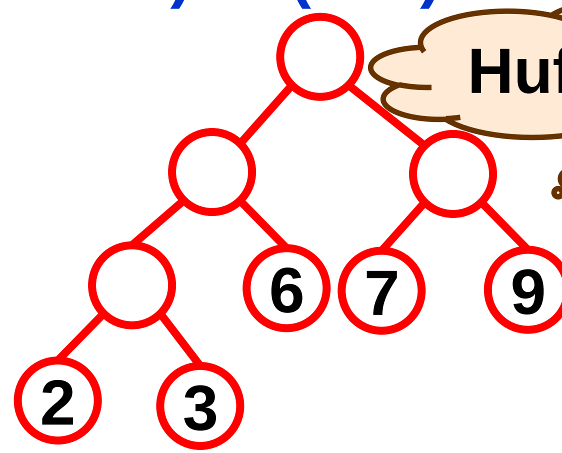


WPL=(2+7+9)*2+(3+6)*3=63



WPL=(2+3+6)*2+(9+7)*3 =70

Huffman树



WPL=(6+7+9)*2+(2+3)*3=59

Huffman树的结构特点?

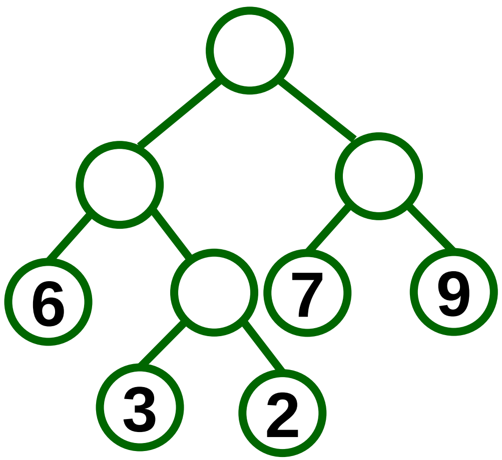
权值大的结点更靠近根结点

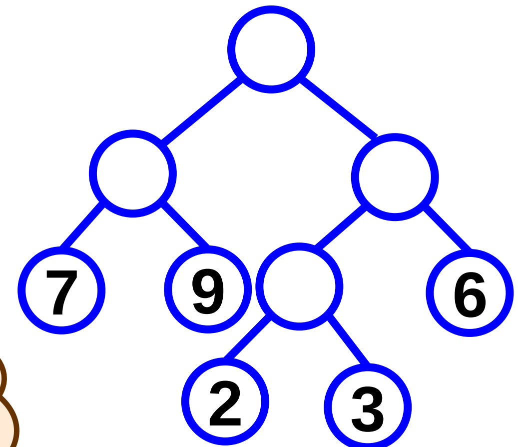
二叉树的应用

哈夫曼(Huffman)树——带权路径长度最短的树

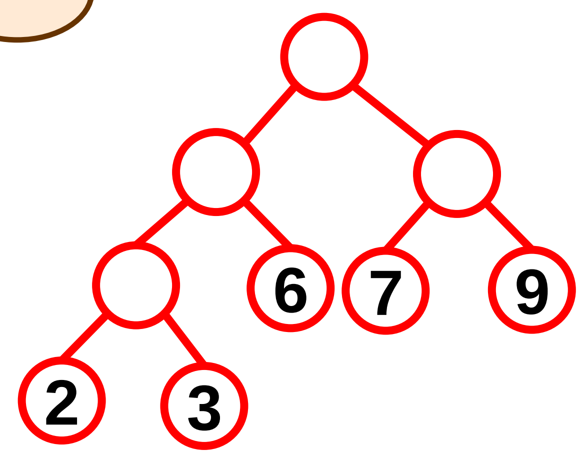
例　有5个结点，权值分别为2，3，6，7，9
　　构造有5个叶子结点的二叉树

$$wpl = \sum_{k=1}^{n} w_k l_k$$





Huffman树
不唯一



WPL=(6+7+9)*2+(2+3)*3=59

Huffman树的结构特点?

权值大的结点更靠近根结点

二叉树的应用

哈夫曼树的构造

问题：已知n个权值{$w_1,w_2, \ldots ,w_n$}，构造一棵有n个叶子结点的Huffman树，每个叶子的权值为w_i。

步骤：

❶ 初始，构造n棵只有根结点的二叉树，根结点的权值分别为w_i，形成有n棵树的森林；

❷ 从森林中选取根结点权值最小的2棵二叉树作为左右子树，构造一棵新的二叉树，其根结点的权值等于左、右孩子的权值之和；

❸ 在森林中删除选中的2棵二叉树，加入新的二叉树；

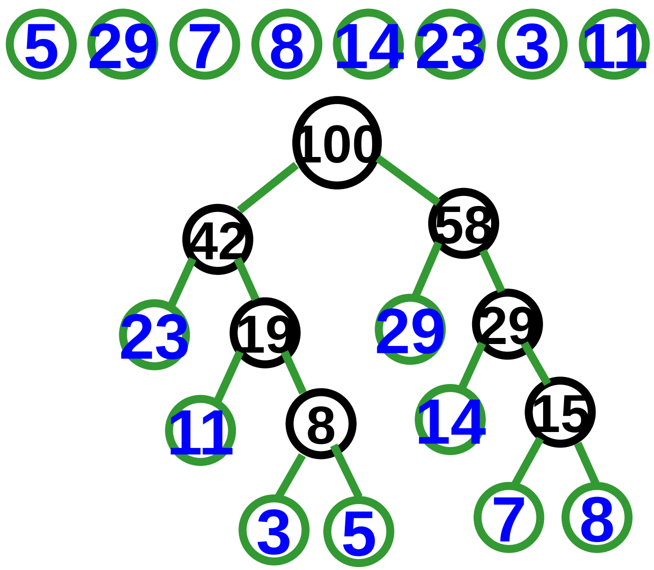
❹ 重复❷❸步，直到森林中只有1棵二叉树为止，即得到哈夫曼树。

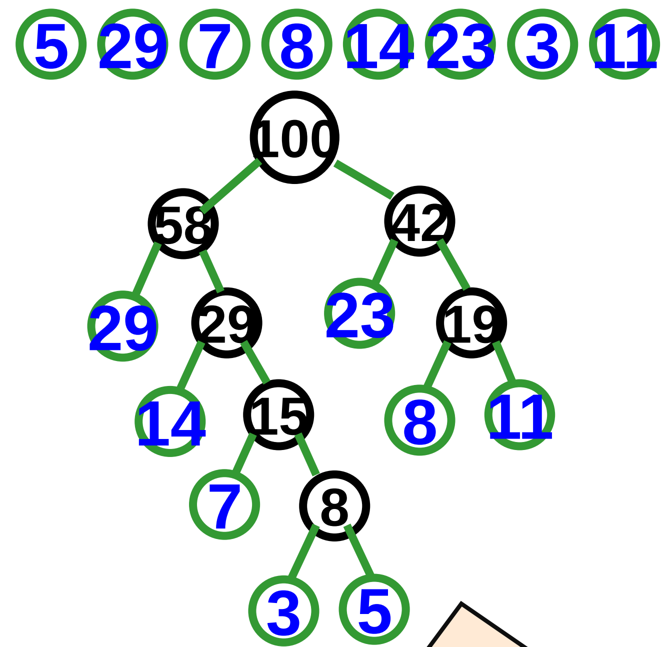
二叉树的应用

一棵有**n**个叶子结点的
Huffman树共有**2n-1**个结点

哈夫曼树的构造

例 **w={5, 29, 7, 8, 14, 23, 3, 11}**





若有根权值相同
的二叉树，任选一棵

带权路径长度

WPL=(23+29)*2+(11+14)*3+(3+5+7+8)*4
=271

WPL=(23+29)*2+(11+8+14)*3+7*4+(3+5)*5
=271

二叉树的应用

哈夫曼树的应用

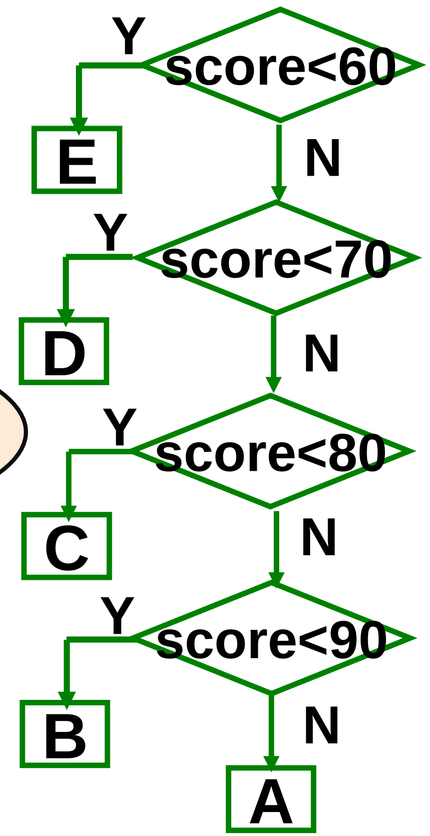
最佳判定问题

例 将百分制成绩转换为五分制。

分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10
等级	E	D	C	B	A

```
if(score<60)
    degree=‘E’;
else if(score<70)
    degree=‘D’;
else if(score<80)
    degree=‘C’;
else if(score<90)
    degree=‘B’;
else   degree=‘A’;
```

80%以上的学生要
经过2次以上的比较



二叉树的应用

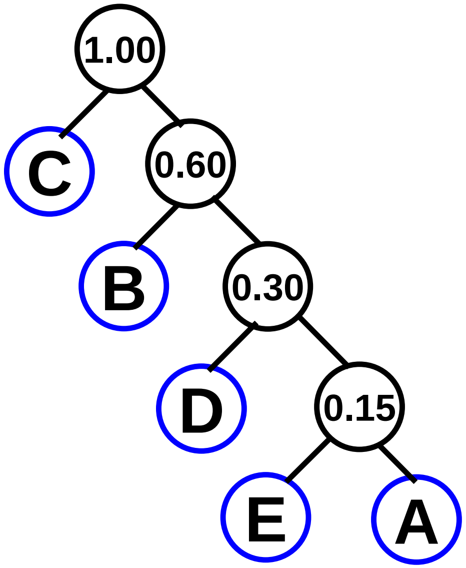
哈夫曼树的应用

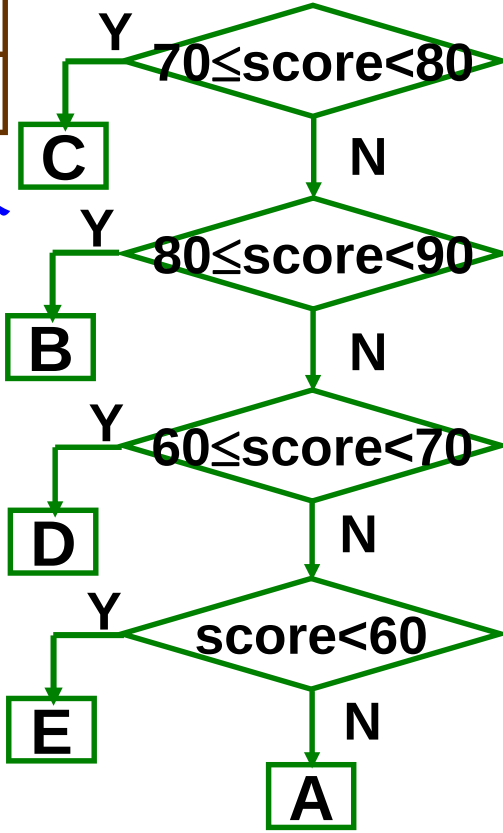
最佳判定问题

例 将百分制成绩转换为五分制。

分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10
等级	E	D	C	B	A

应利用各分数段学生的比例，以其为权值构造哈夫曼树，实现最佳判定。





二叉树的应用

哈夫曼树的应用

Huffman编码——数据通信使用的二进制编码

原文 $\xrightarrow{编码}$ 电文(二进制字符串) $\xrightarrow{译码}$ 原文

发送方　　　　　　　　接收方

Huffman编码目的：接收方能够译出正确、唯一的原文，且传输电文的长度最短。

例 要传输的原文为ABACCDA

能正确译码，但非最短

❶ 若设计等长编码—— A:00 B:01 C:10 D:11

发送方 将ABACCDA编码为：00 01 0010 10 1100

接收方 将00010010101100译为：ABA CCDA

❷ 若设计不等长编码—— A:0 B:00 C:1 D:01

发送方 将ABACCDA编码为：000011010

接收方 将000011010译为：AAAACCDA
BBCCDA

不能正确译码原因：A的编码是B的前缀

二叉树的应用

哈夫曼树的应用

Huffman编码——数据通信使用的二进制编码

原文 —编码→ 电文(二进制字符串) —译码→ 原文

发送方　　　　接收方

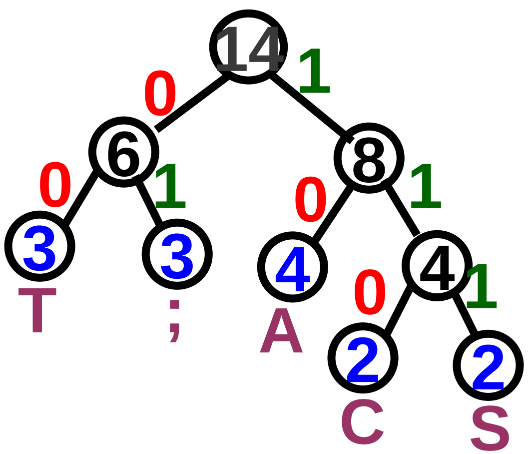
☺编码：

❶根据**字符出现频率构造Huffman树**；

❷将树中结点引向左孩子的分支标"0"，引向右孩子的分支标"1"；

❸每个字符的编码即为从根到叶子的路径上得到的0、1序列。

例：传输的电文集D={C,A,S,T, ; }，字符出现频率w={2,4,2,3,3}。



T： 00

;： 01

A： 10

C： 110

S： 111

Huffman编码如何保证：

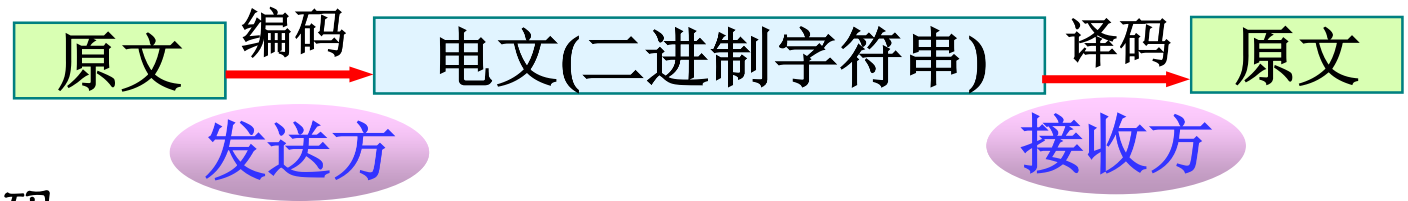
①某字符编码不是其它字符编码的前缀，即使得接收方能够正确译码？

②电文长度最短？

二叉树的应用

哈夫曼树的应用

Huffman编码——数据通信使用的二进制编码

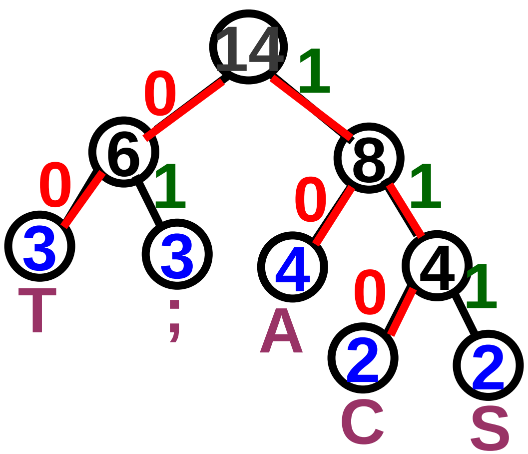


译码：

❶从Huffman树的根结点开始，从电文中逐位取码；

❷若取码为"0"，则向左走；若取码为"1"，则向右走，直到到达叶子结点，则译出一个字符；

❸再重新从根结点出发，按❷译码，直到电文结束。



T : 00

; : 01

A : 10

C : 110

S : 111

例 若发送CAS;CAT

则电文为:11010111011101000

若接收电文为1101000

译文只能是C A T

本 章 小 结

- 掌握树的基本概念和术语
- 掌握二叉树的定义、性质和几种特殊形态的二叉树
- 掌握二叉树的存储结构，遍历的定义及其递归算法的实现（及应用）
- 掌握树的存储结构及其遍历过程
- 掌握森林的遍历过程
- 掌握树、森林与二叉树的相互转换
- 掌握哈夫曼树的实现方法及带权路径长度的计算，掌握哈夫曼编码的构造方法

Back