人工智能学院

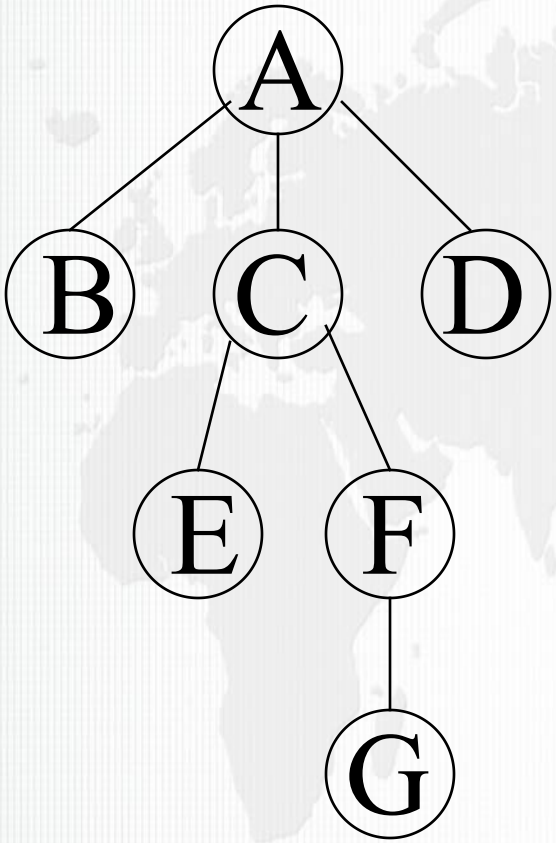
Data Structures and Algorithms

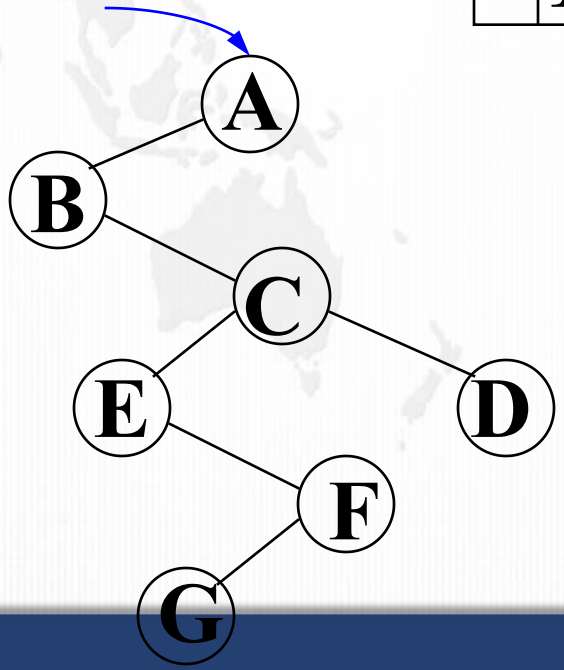
Trees and Binary Trees 3

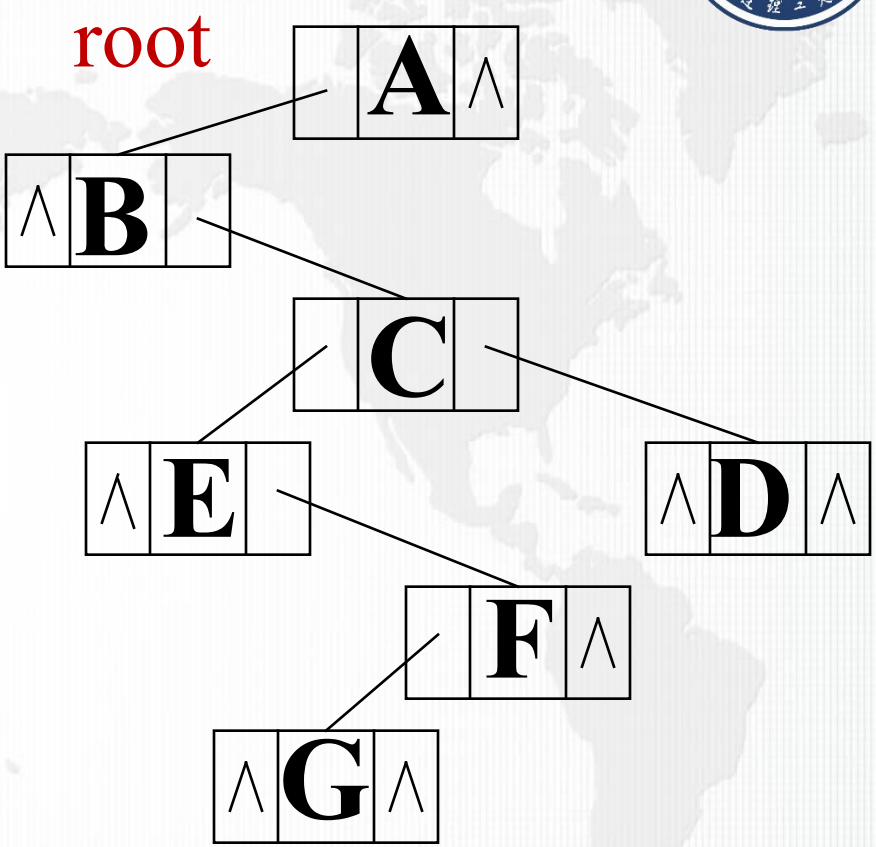
Preview

1. Review
2. Application
3. Summary

Preview

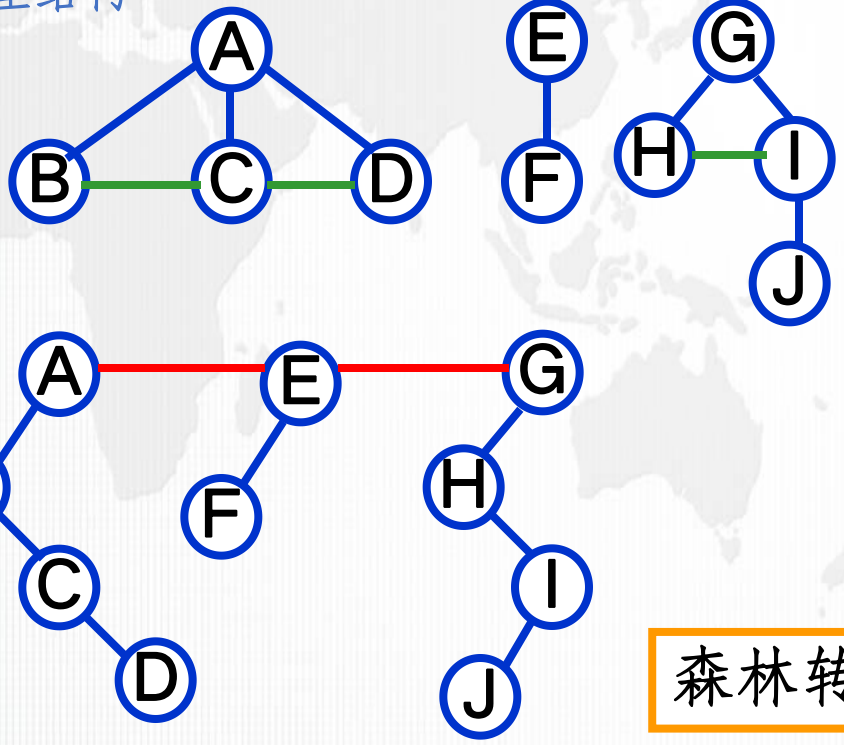


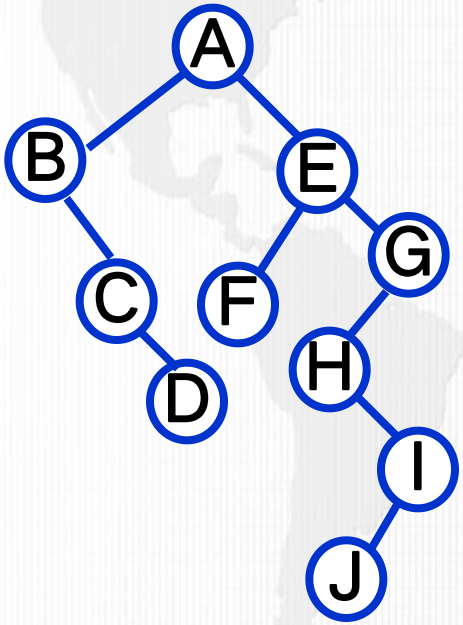




Preview

- 森林转化为二叉树：
  1）将各棵树分别转换成二叉树
  2）将每棵树的根结点用线相连
  3）以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构





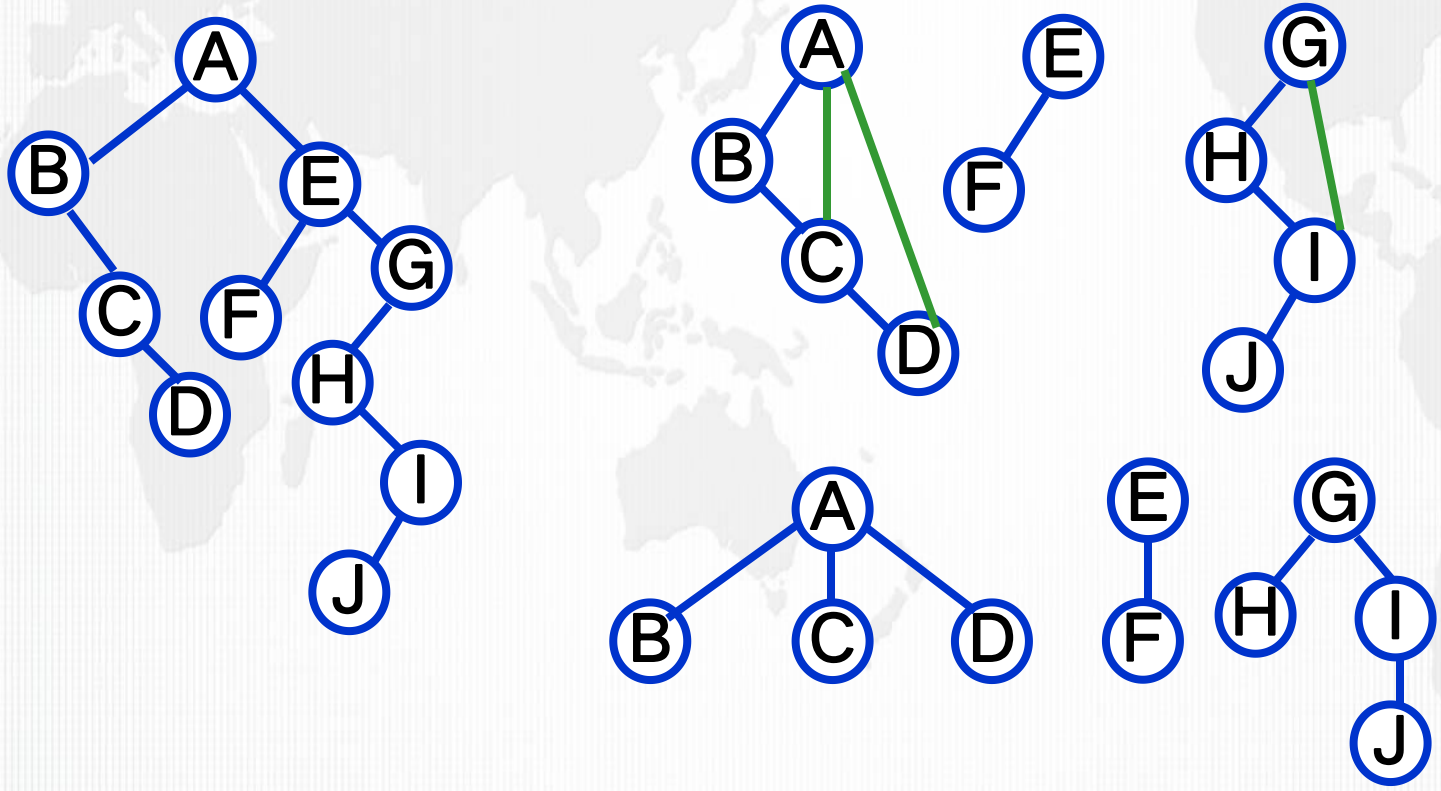
森林转换成的二叉树其右子树非空

Preview

- 二叉树转化为森林：

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

  2）还原：将孤立的二叉树还原成树



Preview

- 二叉树遍历的几种算法：

  先（根）序的遍历算法：先访问根结点，然后分别先序遍历左子树、右子树

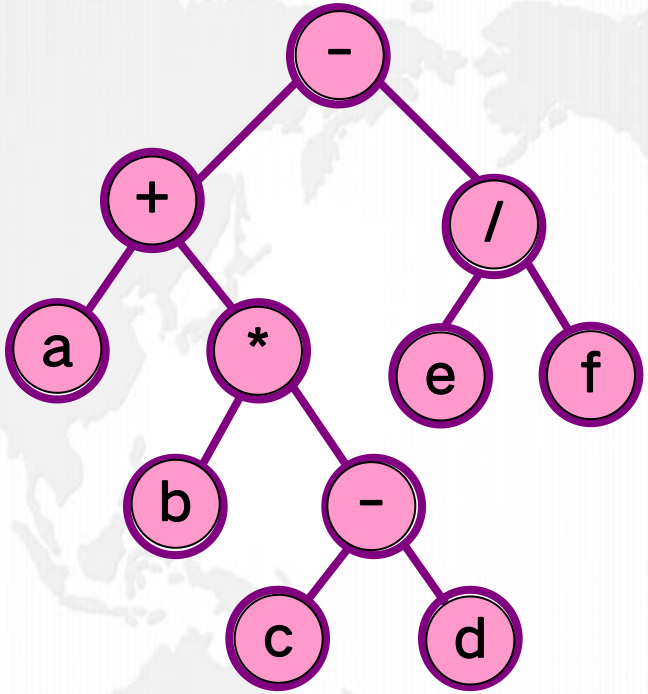
  中（根）序的遍历算法：先中序遍历左子树，然后访问根结点，最后中序遍历右子树

  后（根）序的遍历算法：先后序遍历左、右子树，然后访问根结点

  层次遍历算法：从上到下、从左到右访问各结点

Preview

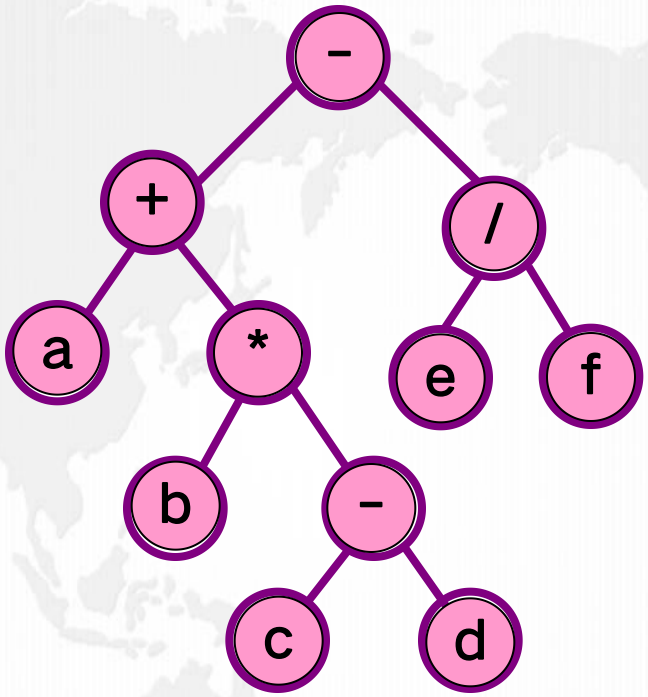
◆ 遍历举例：



先序遍历：-+a*b-cd/ef

Preview

◆ 遍历举例：

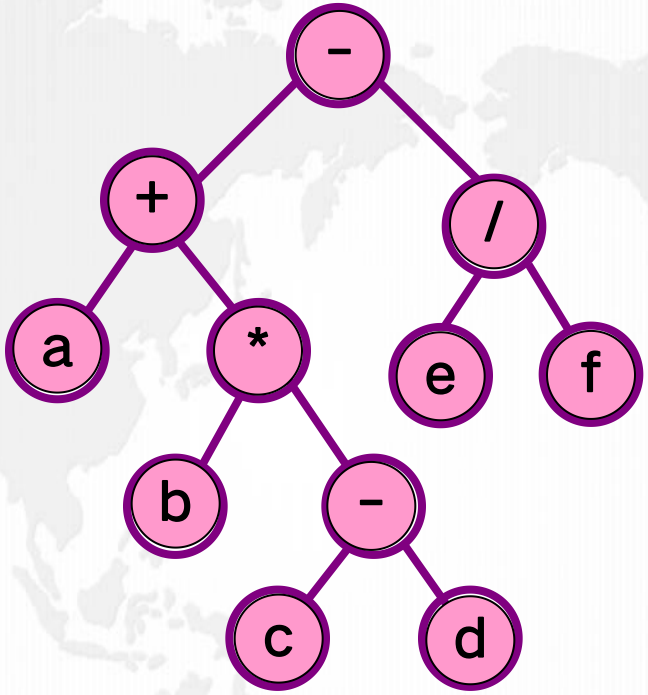


先序遍历：-+a*b-cd/ef

中序遍历：a+b*c-d-e/f

Preview

◆ 遍历举例：



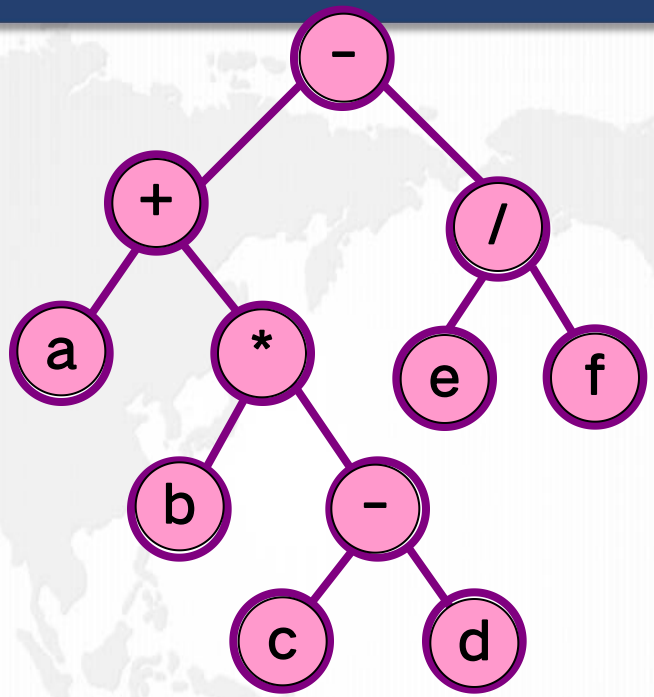
先序遍历：-+a*b-cd/ef

中序遍历：a+b*c-d-e/f

后序遍历：abcd-*+ef/-

Preview

◆ 遍历举例：



先序遍历：-+a*b-cd/ef

中序遍历：a+b*c-d-e/f

后序遍历：abcd-*+ef/-

层次遍历：-+/a*efb-cd

Preview

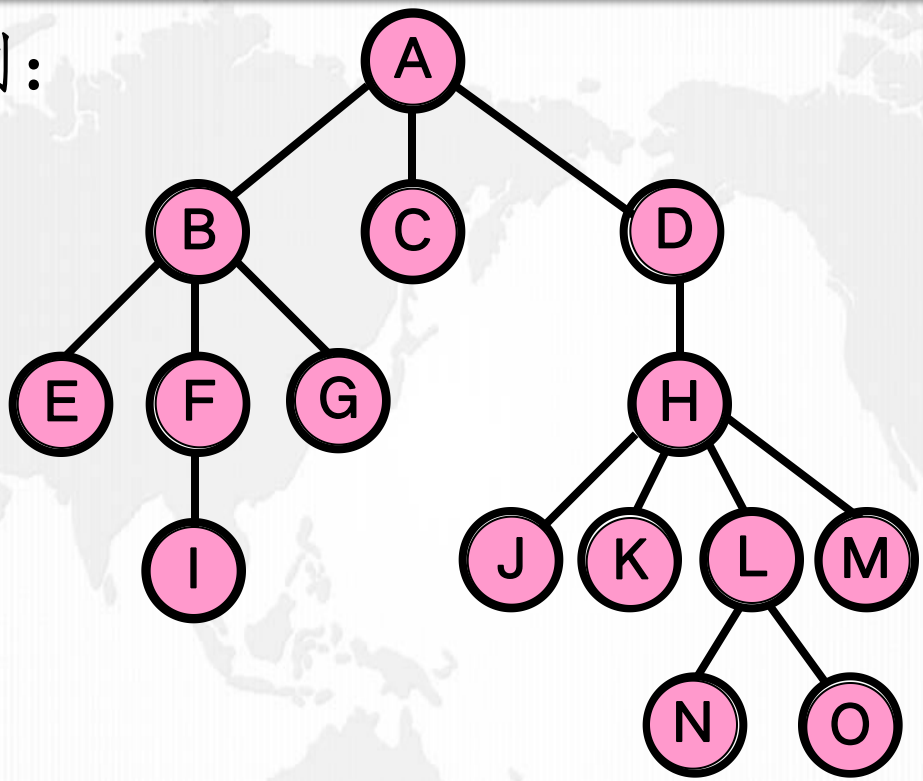
- 树的遍历常用方法：
  - 先根（序）遍历：先访问树的根结点，然后依次先根遍历根的每棵子树　　“根左右”

  - 后根（序）遍历：先依次后根遍历每棵子树，然后访问根结点　　“左右根”

  - 按层次遍历：先访问第一层上的结点，然后依次遍历第二层，……第n层的结点

Preview

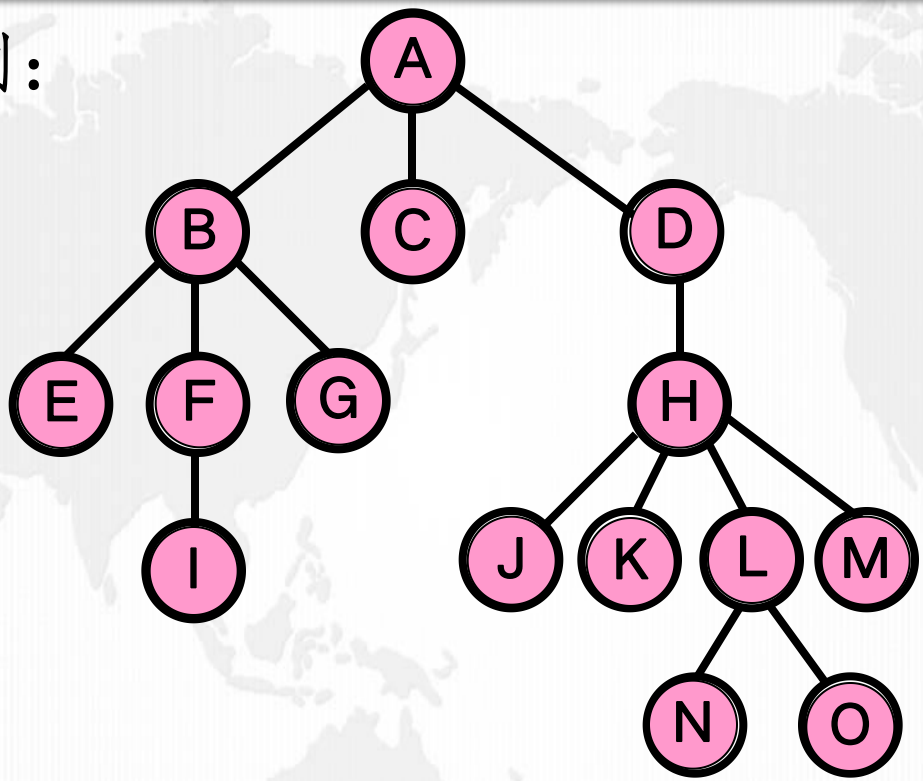
- 树的遍历案例：



先根遍历：ABEFI GCDHJ KL NOM

Preview

- 树的遍历案例：

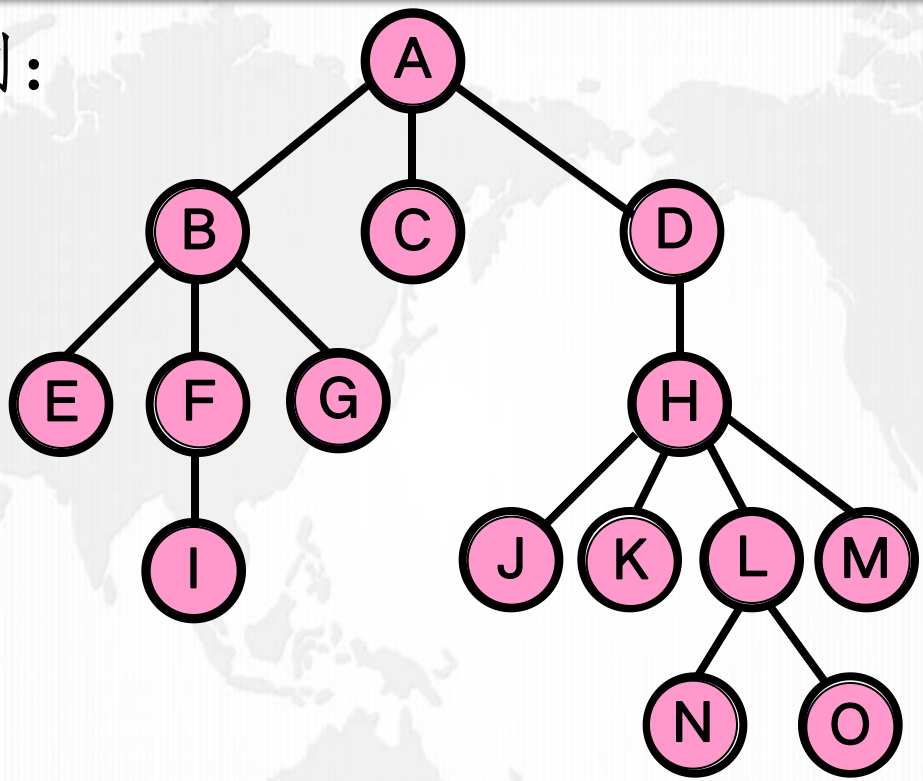


先根遍历：ABEFI GCDHJ KL NOM

后根遍历：EI FGBCJK NOL MHDA

Preview

◆ 树的遍历案例：



先根遍历：ABEFI GCDHJ KL NOM

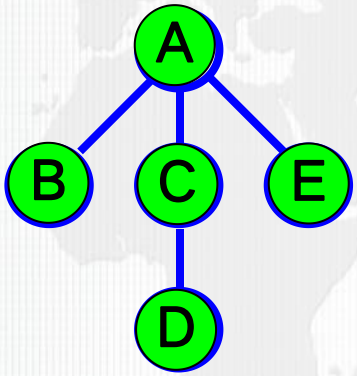
后根遍历：EI FGBCJK NOL MHDA

层次遍历：ABCDE FGHI J KL MNO

Preview

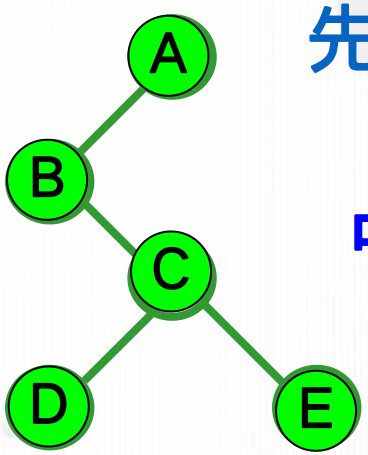
- 树的遍历算法——递归算法（采用孩子—兄弟存储结构）：

  将孩子—兄弟链表看作某棵二叉树的二叉链表表示



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

Preview

- 森林的组成部分：
  1. 森林中第一棵树的根结点；
  2. 森林中第一棵树的子树森林；
  3. 森林中其它树构成的森林。

- 森林遍历的常用方法：
- 1）先序遍历森林。若森林为非空，则按如下规则进行遍历:
  - 访问森林中第一棵树的根结点。
  - 先序遍历第一棵树中根结点的子树森林。
  - 先序遍历除去第一棵树之后剩余的树构成的森林。

- 2）中序遍历森林。森林为非空时，按如下规则进行遍历:
  - 中序遍历森林中第一棵树的根结点的子树森林。
  - 访问第一棵树的根结点。
  - 中序遍历除去第一棵树之后剩余的树构成的森林。

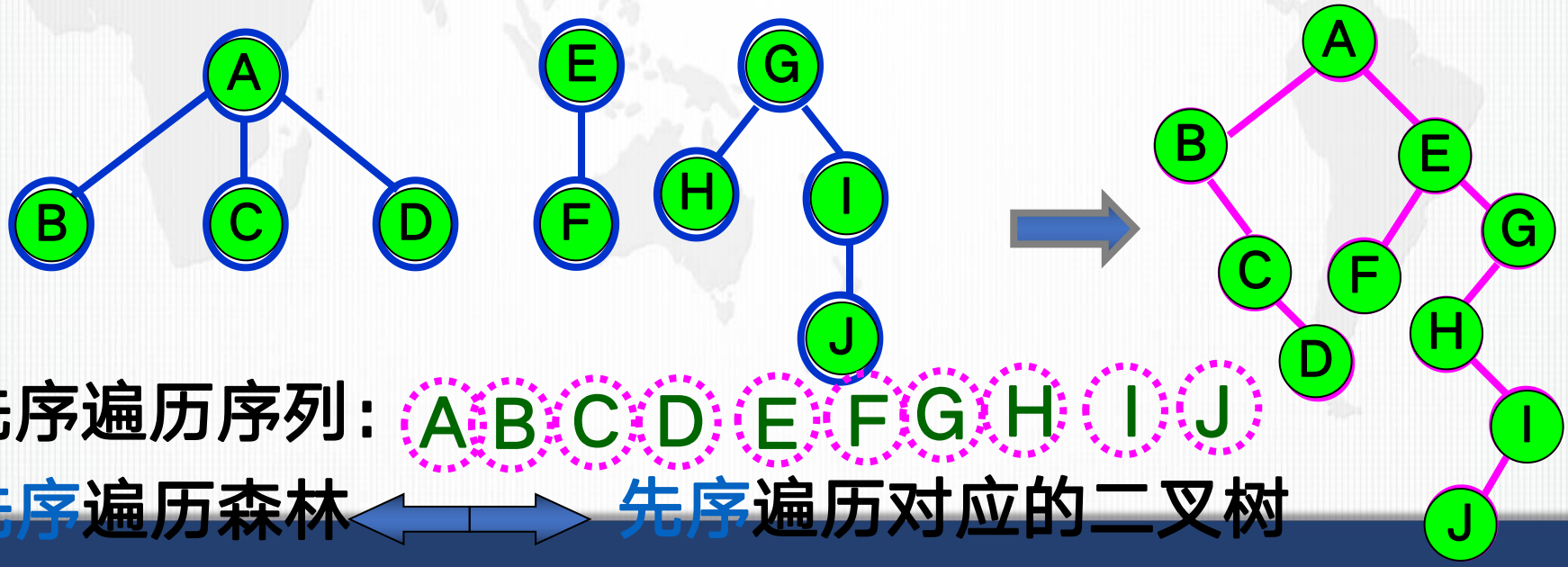
Preview

◆ 森林的遍历——先序遍历：

若森林为非空，则按如下规则进行遍历:

- 访问森林中第一棵树的根结点。
- 先序遍历第一棵树中根结点的子树森林。
- 先序遍历除去第一棵树之后剩余的树构成的森林。

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

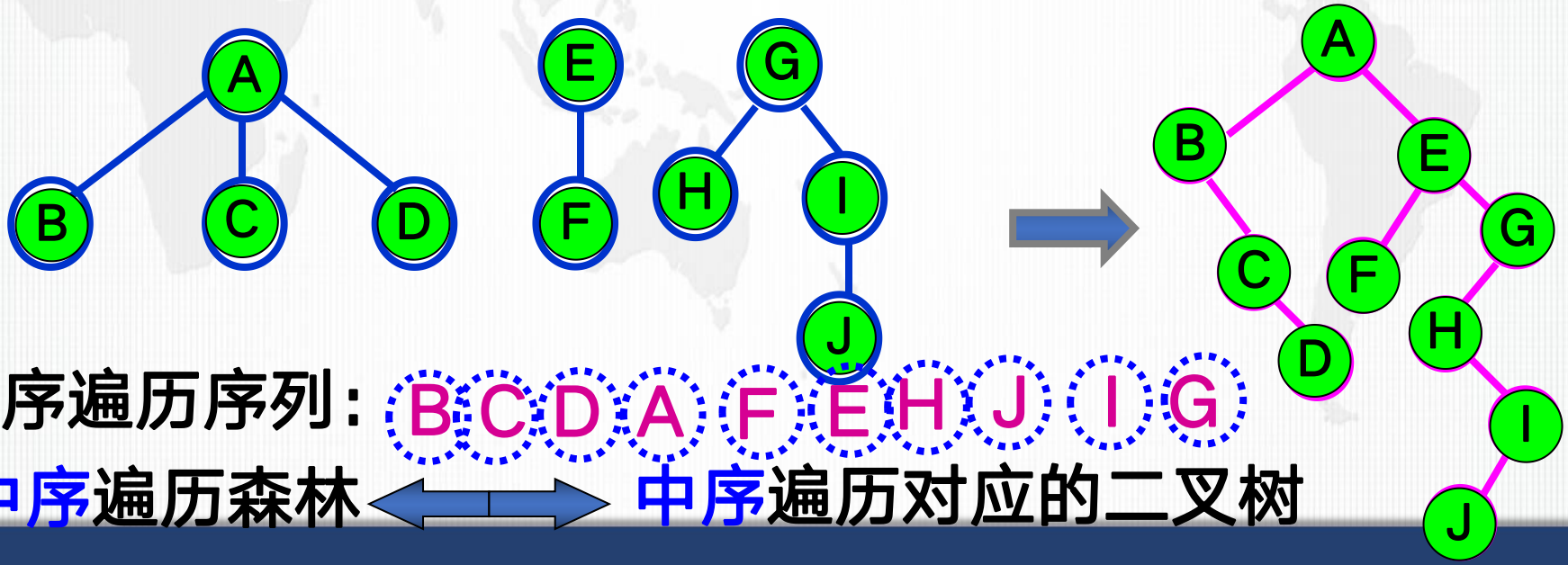
Preview

◆ 森林的遍历——中序遍历：

森林为非空时，按如下规则进行遍历:

- 中序遍历森林中第一棵树的根结点的子树森林。
- 访问第一棵树的根结点。
- 中序遍历除去第一棵树之后剩余的树构成的森林。

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

Preview

树的遍历、森林的遍历与二叉树遍历的对应关系？

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	后序遍历	中序遍历

Preview

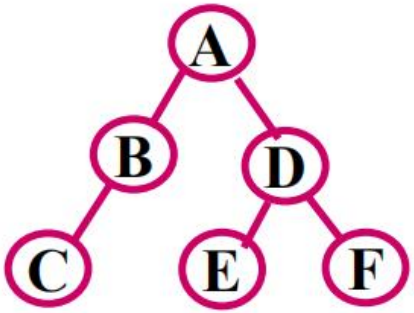
- 由遍历序列确定二叉树：
  - 给定先序、中序遍历序列可唯一确定二叉树。
  - 给定后序、中序遍历序列可唯一确定二叉树。
  - 给定层次、中序遍历序列可唯一确定二叉树。

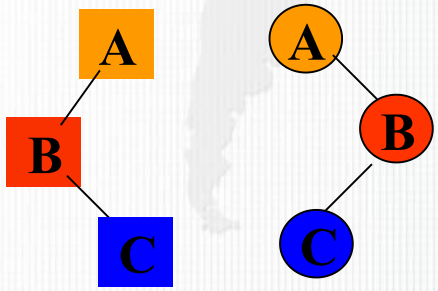
- 由二叉树先序序列和中序序列，构造二叉树

例如，先序序列为：A B C D E F

中序序列为：C B A E D F

请构造二叉树

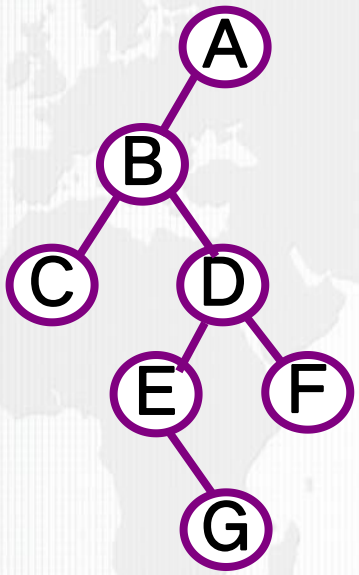


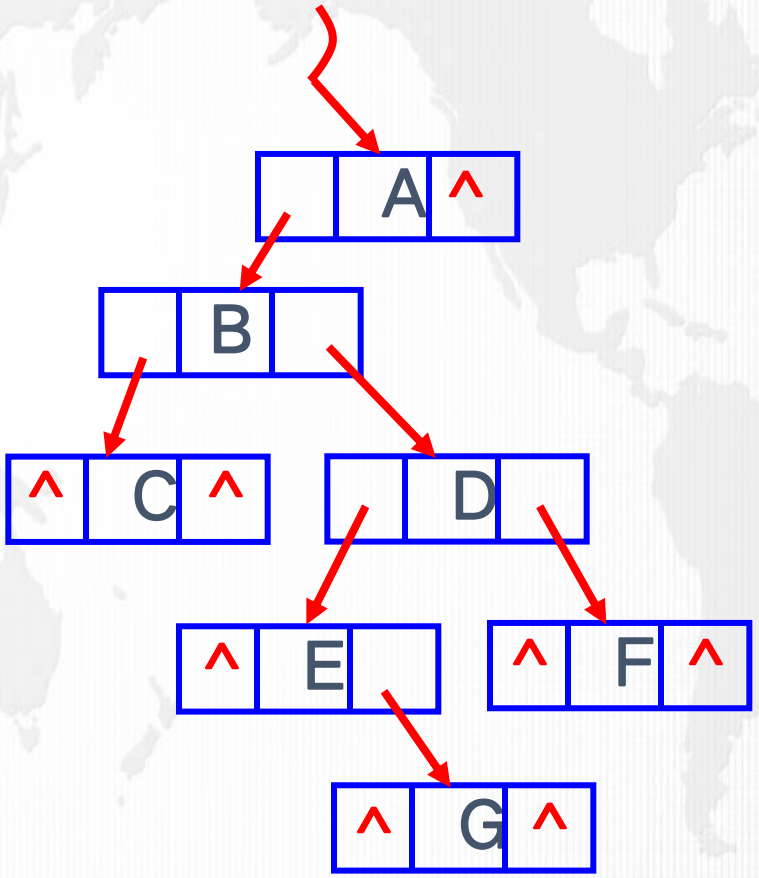


Application

◆ 链式存储实现——二叉链表：

lchild	data	rchild





在n个结点的二叉链表中，
有 n+1 个空指针域

Application

◆ 链式存储实现——二叉链表：

在n个结点的二叉链表中，有 n+1 个空指针域

外部节点（或空指针节点）：这是指不存在的节点，或者说是一个空的指针域。在实际的链式存储结构中，这是一个空指针。
内部节点：这是实际存在的节点，也是我们通常所说的二叉树的节点。

对于任何非空二叉树：

- 一个节点只有一个到达自己的指针（除了根节点，它没有到达自己的指针）。
- 每个节点都有2个指针，指向左孩子和右孩子。这些孩子要么是内部节点，要么是外部节点。

具有 n 个内部节点的二叉树：

- 有n−1 个内部指针，因为除了根节点，每个节点都有一个指针指向它。
- 有 2n 个指针总共从所有的内部节点出来，因为每个节点有2个指针。
- 这 2n 个指针分为两种：指向内部节点的和指向外部节点的。我们已经知道有 n−1 个指针指向内部节点，因此有 2n-(n-1)=n+1 个指针指向外部节点（即空指针域）。

Application

线索二叉树：

- 遍历二叉树是以一定规则将二叉树中的结点排列成一个线性序列，这实质上是对一个非线性结构进行线性化操作，使每个结点(除第一个和最后一个外)在这些线性序列中有且仅有一个直接前驱和直接后继。

- 但是，当以二叉链表作为存储结构时，只能找到结点的左、右孩子信息，而不能直接得到结点在任一序列中的前驱和后继信息，这种信息只有在遍历的动态过程中才能得到，为此引入线索二叉树来保存这些在动态过程中得到的有关前驱和后继的信息。

- 虽然可以在每个结点中增加两个指针域来存放在遍历时得到的有关前驱和后继信息，但这样做使得结构的存储密度大大降低。由于有n个结点的二叉链表中必定存在n+1个空链域，因此可以充分利用这些空链域来存放结点的前驱和后继信息。

Application

- 线索二叉树：
- 试做如下规定：若结点有左子树，则其Ichild域指示其左孩子，否则令Ichild域指示其前驱；若结点有右子树，则其rchild域指示其右孩子，否则令rchild域指示其后继。为了避免混淆，尚需改变结点结构，增加两个标志域。

leftChild	leftTag	element	rightTag	rightChild

其中：

$$leftTag = \begin{cases} 0 & \text{lchild域指示结点的左儿子} \\ 1 & \text{lchild域指示结点的前驱结点} \end{cases}$$

$$rightTag = \begin{cases} 0 & \text{rchild域指示结点的右儿子} \\ 1 & \text{rchild域指示结点的后继结点} \end{cases}$$

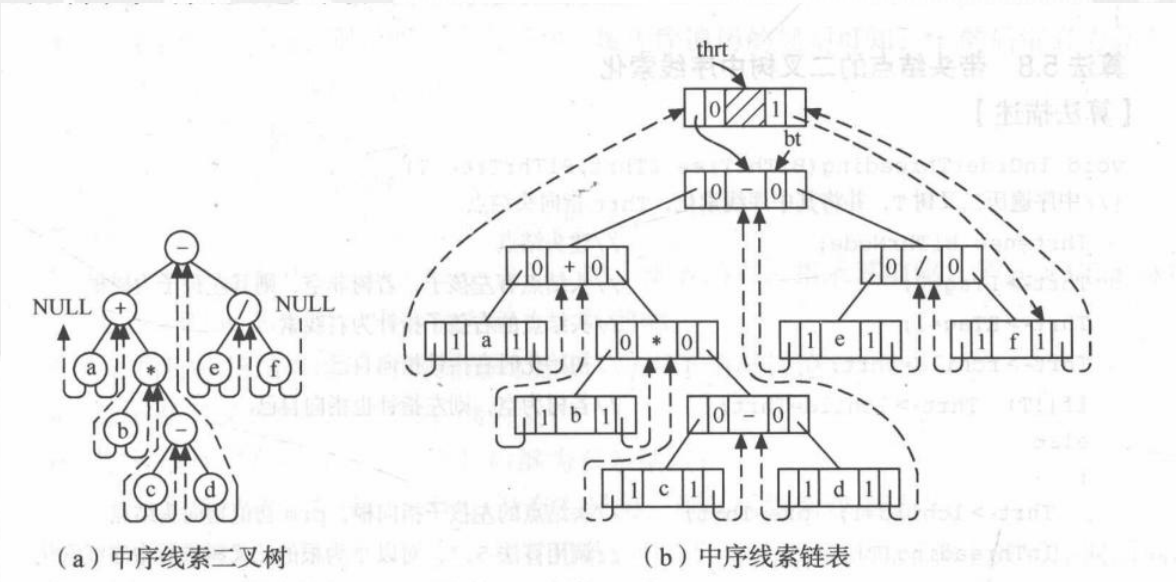
Application

- 线索链表的类型描述：

```
typedef int TElemType;
typedef enum {
  Link, Thread
} PointerTag;

typedef struct BiThrNode {
  TElemType data;
  struct BiThrNode *lchild, *rchild;
  PointerTag LTag, Rtag;
} BiThrNode, *BiThrTree;
```

Application

- 线索二叉树的构造——中序序列构造线索二叉树：
  - 二叉树的线索化是将二叉链表中的空指针改为指向前驱或后继的线索。而前驱或后继的信息只有在遍历时才能得到，因此线索化的实质就是遍历一次二叉树。
  - 以中序线索二叉树的建立为例。附设指针pre指向刚刚访问过的结点，指针p指向正在访问的结点，即pre指向p的前驱。在中序遍历的过程中，检查p的左指针是否为空，若为空就将它指向pre；检查pre的右指针是否为空，若为空就将它指向p。



Application

- 线索二叉树的构造——中序序列构造线索二叉树：

- 中序遍历的特点：在中序遍历中，对于任何一个节点，其前驱是其左子树的最右下角的节点，其后继是其右子树的最左下角的节点。

- 算法步骤（假设已经建立了普通二叉树）：

  1）如果p非空，左子树递归线索化。

  2）如果p的左孩子为空，则给p加上左线索，将其LTag置为1,让p的左孩子指针指向pre（前驱）；否则将p的LTag置为0。

  3）如果pre的右孩子为空，则给pre加上右线索，将其RTag置为1,让pre的右孩子指针指向p（后继）；否则将pre的RTag置为0。

  4）将pre指向刚访问过的结点p,即pre＝p。

  5）右子树递归线索化。

Application

◆ 线索二叉树的构造——中序序列构造线索二叉树：

```
// 中序遍历并线索化
void inOrderThreading(BiThrTree node) {
    if (node) {
        inOrderThreading( node: node->lchild);
        // 如果当前节点的左子树为空，设置lchild为前驱
        if (!node->lchild) {
            node->LTag = Thread;
            node->lchild = pre;
        }
        // 如果前一个节点的右子树为空，设置rchild为当前节点
        // pre为NULL时，说明当前节点是第一个被访问的节点，没有前驱
        if (pre && !pre->rchild) {
            pre->RTag = Thread;
            pre->rchild = node;
        }
        pre = node;
        inOrderThreading( node: node->rchild);
    }
}
```

当我们中序遍历到某个节点node时，这意味着我们已经访问了它的左子树（如果它有的话），并且pre是node的前驱。此时，我们要看看pre是否有右子节点。如果没有，这意味着pre的右指针是空闲的，可以被用作线索。因此，我们将pre的右指针指向node，即node是pre的后继。

Application

- 线索链表的遍历算法——遍历中序二叉树：

  1.指针p指向根结点。

  2.P为非空树或遍历未结束时，循环执行以下操作：

  1）沿左孩子向下，到达最左下结点*p，它是中序的第一个结点；

  2）访问*p；沿右线索反复查找当前结点*p的后继结点并访问后继结点，直至右线索为0或者遍历结束；

  3）转向p的右子树。

- 中序线索化链表的第一个结点：

  左子树上处于“最左下”（没有左子树）的结点。

- 中序线索化链表中结点的后继：

  若无右子树，则为后继线索所指结点；

  否则为对其右子树进行中序遍历时访问的第一个结点。

Application

◆ 线索链表的遍历算法——遍历中序二叉树：

```
54  void InOrderTraverse(BiThrTree T) {
55      BiThrTree p = T;
56      // 找到最左侧的节点，这是中序遍历的起点
57      while (p && p->LTag == Link) {
58          p = p->lchild;
59      }
60      while (p) {
61          printf("%d ", p->data);
62          // 如果右指针是线索，直接指向后继节点
63          if (p->RTag == Thread) {
64              p = p->rchild;
65          } else { // 否则，找到右子树的最左侧节点
66              p = p->rchild;
67              while (p && p->LTag == Link) {
68                  p = p->lchild;
69              }
70          }
71      }
72      printf("\n");
73  }
```

Application

◆ 线索二叉树应用：

- 非递归遍历: 在资源受限的环境中，如嵌入式系统或早期的计算机系统，递归可能会导致堆栈溢出。在这种情况下，线索二叉树提供了一种避免递归的遍历方法。
- 数据库和文件系统: 在某些数据库和文件系统的实现中，线索二叉树可以作为索引结构。通过线索化，可以快速找到一个节点的前驱和后继，这在某些查询和更新操作中很有用。
- GUI渲染: 在图形用户界面的某些渲染算法中，元素的遍历顺序很重要。线索二叉树可以帮助确保正确的遍历顺序，而无需额外的数据结构。
- 快速查找前驱和后继: 在某些应用中，快速查找一个节点的前驱和后继是很重要的。例如，文本编辑器可能需要快速导航到文档的前一个或后一个段落，而这些段落的结构可以用线索二叉树表示。
- 交互式应用: 在某些交互式应用中，用户可能会频繁地在数据结构中前进和后退。线索二叉树提供了一种简洁的方式来实现这种导航。

Application

◆ 统计二叉树中叶子结点个数算法：

局部静态变量具有继承性——
编译时分配内存，程序结束时释放

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

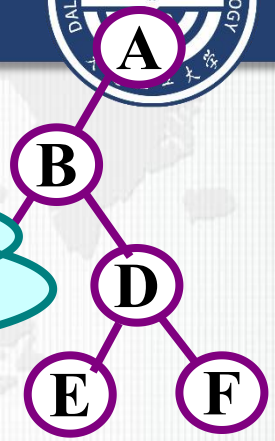
定义变量

T不为空时
求叶子结点数

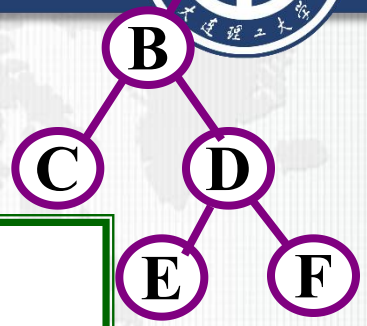
判断T是否为叶子结点

求T左、右子树
上的叶子结点数

每次递归调用——
当T为叶子时，count自加



Application



◆ 统计二叉树中叶子结点个数算法：

——左子树叶子数+右子树叶子数

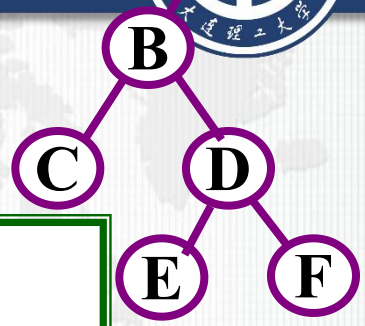
```
void LeafCount2 (BiTree T,  int *count)
{
  if ( T )
  {
      if ((T->lchild==NULL)&& (T->rchild==NULL))
         count++;
      LeafCount2( T->lchild, count);
      LeafCount2( T->rchild, count);
  }
}
```

T不为空时
求叶子结点数

判断T是否为叶子结点

求T左、右子树
上的叶子结点数

Application

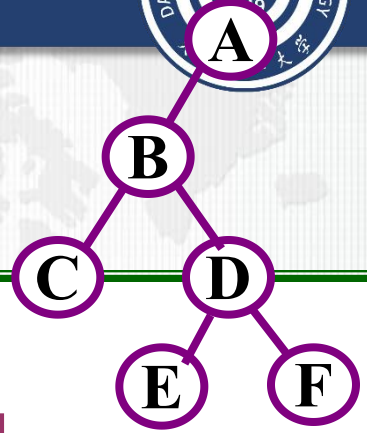


◆ 统计二叉树中叶子结点个数算法：

——左子树叶子数+右子树叶子数

```
int LeafCount3(BiTree T)
{  if (!T) return 0;
     //空树叶子结点个数为0
  if (!T->lchild && !T->rchild) return 1;
     //若根节点没有左右孩子叶子结点个数为1
  return LeafCount(T->lchild) +
            LeafCount(T->rchild);
     //否则为左右叶子结点个数之和
}
```

Application



◆ 求二叉树深度算法：

——max{左子树深度+1，右子树深度+1}

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

定义变量

T为空时，以T为根的树深度为0

分别求T左、右子树的深度

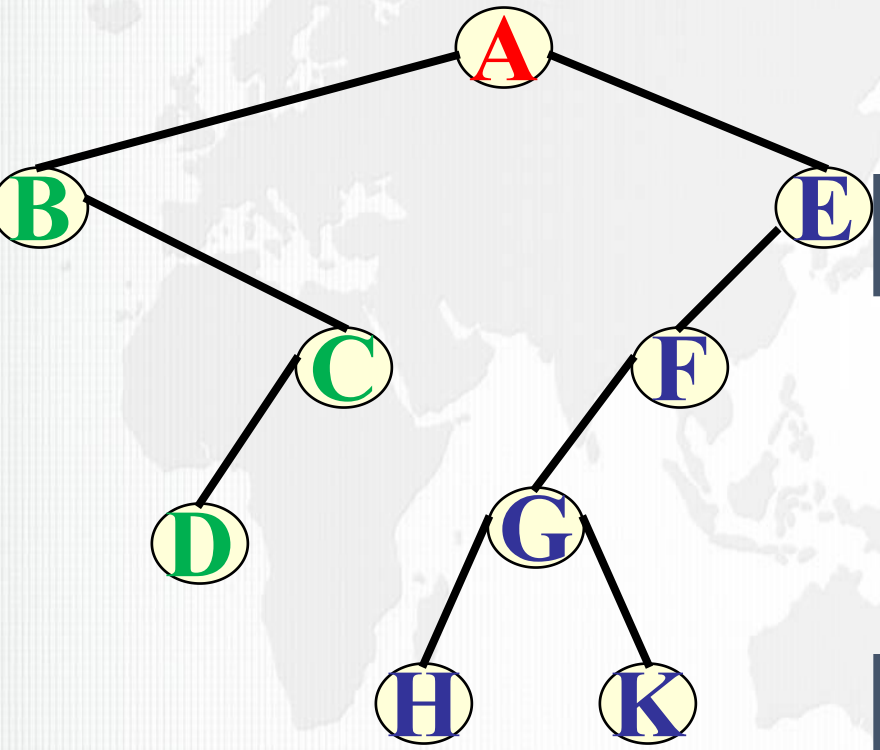
Application

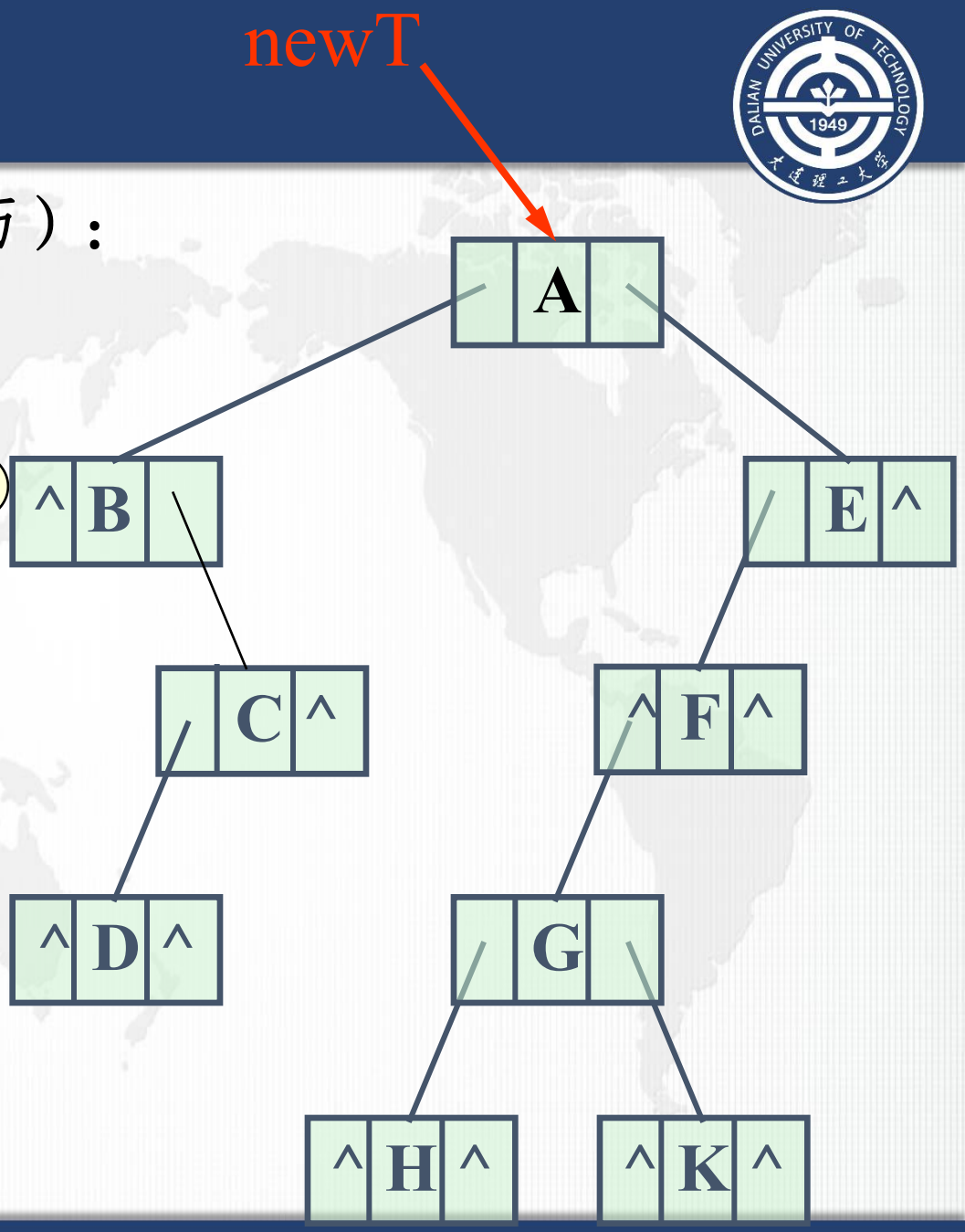
- 复制二叉树算法（后序遍历）：

```
// 后序遍历复制二叉树
Node* copyTreePostOrder(Node* root) {
    if (root == NULL) {
        return NULL;
    }

    // 递归复制左子树和右子树
    Node* leftCopy = copyTreePostOrder( root: root->left);
    Node* rightCopy = copyTreePostOrder( root: root->right);

    // 创建当前节点的副本
    Node* copyNode = newNode(root->data);
    copyNode->left = leftCopy;
    copyNode->right = rightCopy;

    return copyNode;
}
```

Application

- 复制二叉树算法（后序遍历）：





二叉树的复制过程

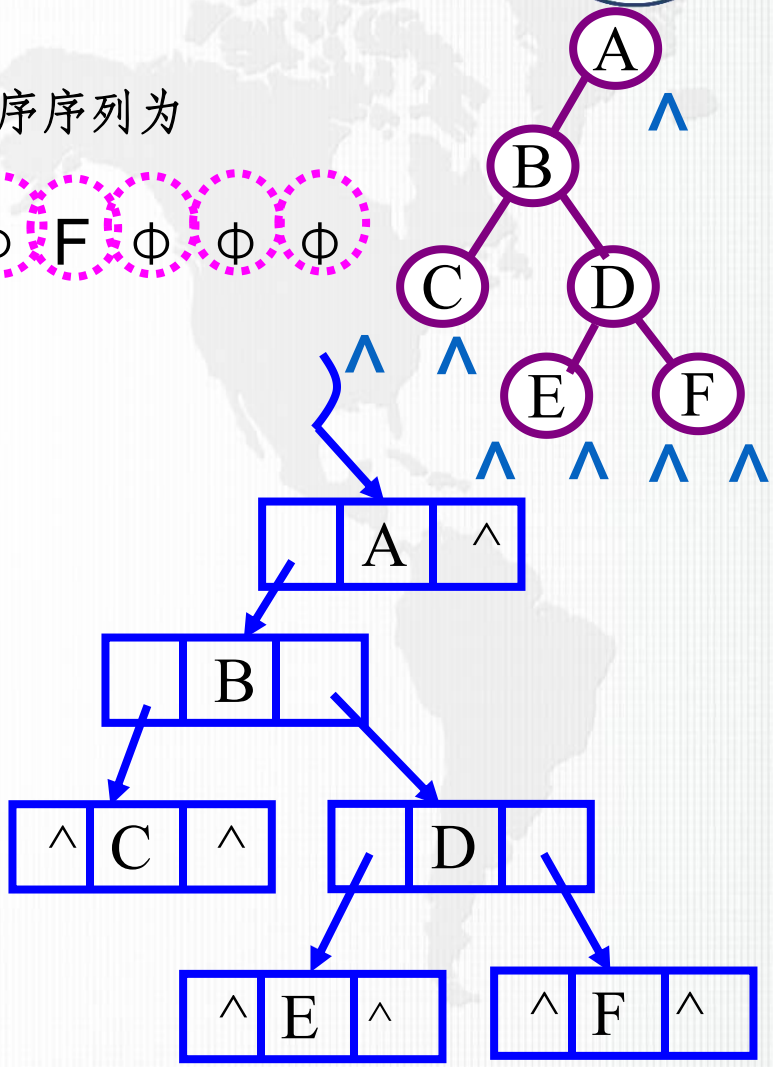
Application

- 建立二叉树的存储结构：

  按先序遍历序列建立二叉树的二叉链表，已知先序序列为

A B C Φ Φ D E Φ Φ F Φ Φ Φ

对于二叉树，若能将某问题分解到其左、右子树上分别进行相同操作，则可将*递归*遍历算法进行适当改写来实现。



Application

- 建立二叉树的存储结构：

  按先序遍历序列建立二叉树的二叉链表，已知先序序列为

  A B C Φ Φ D E Φ Φ F Φ Φ Φ

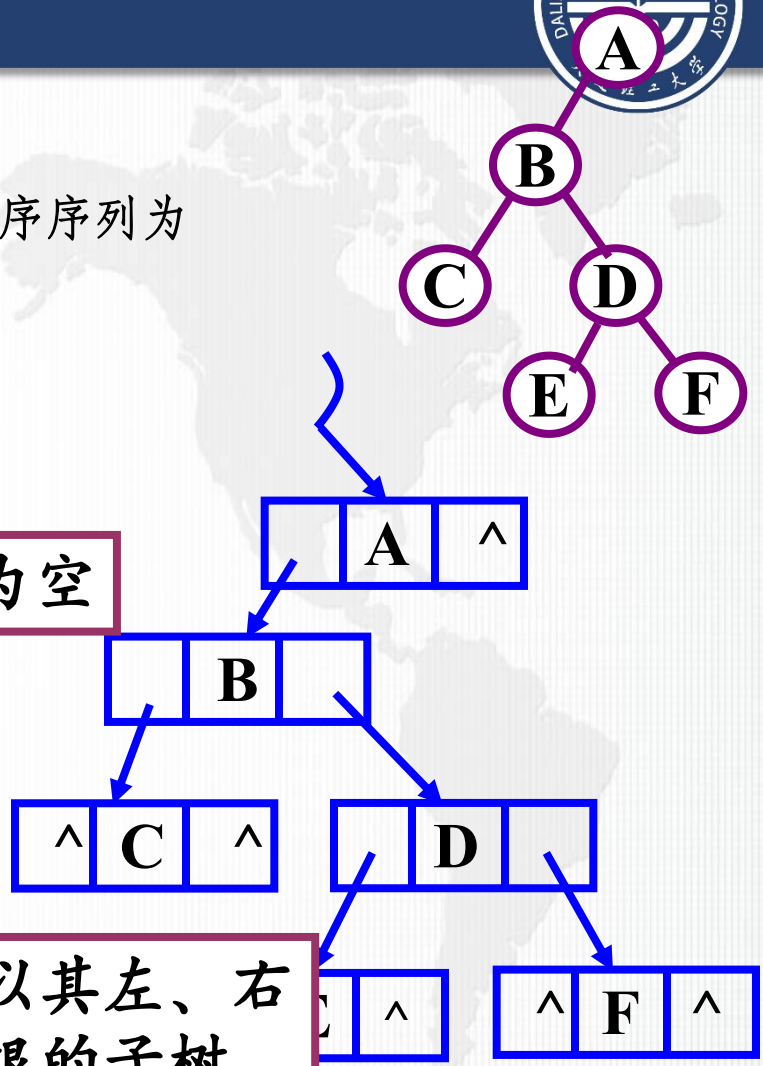
```
void  PreOrderTraverse(BiTree T)
{   ……
    if(T)
    {
      printf("%d\t",T->data);
      PreOrderTraverse(T->lchild);
      PreOrderTraverse(T->rchild);
    }
}
```

接收一个字符

判断字符是否为空

字符不为空，则建立一个新结点并赋值

再建立以其左、右孩子为根的子树



遍历算法——每次递归调用输出一个结点的数据域

Application

◆ 建立二叉树的存储结构：

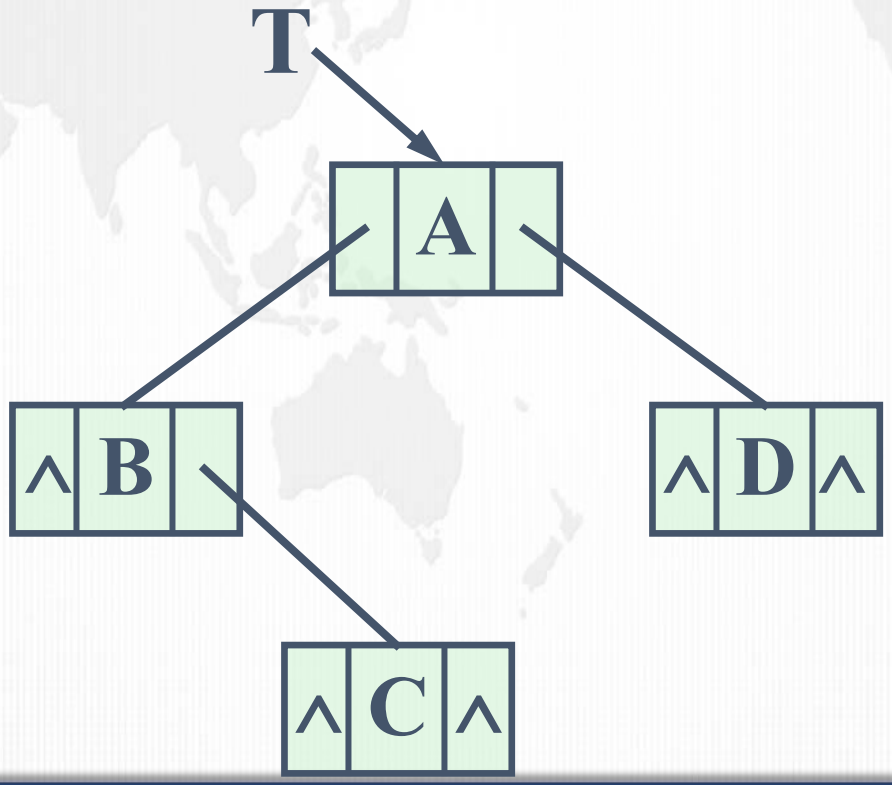
```
28      // 基于先序遍历建立二叉树
29      Node* buildTreeFromPreOrder(char* input) {
30          if (input[idx] == '\0') {
31              return NULL;
32          }
33
34          if (input[idx] == ' ') {
35              idx++;  // 跳过空格并继续处理下一个字符
36              return NULL;
37          }
38
39          Node* node = newNode( data: input[idx]);
40          idx++;  // 移动到下一个字符
41
42          node->left = buildTreeFromPreOrder(input);
43          node->right = buildTreeFromPreOrder(input);
44
45          return node;
46      }
```

Application

- 建立二叉树的存储结构：

  举例如下：

  A B ■ C ■ ■ D ■ ■



Application

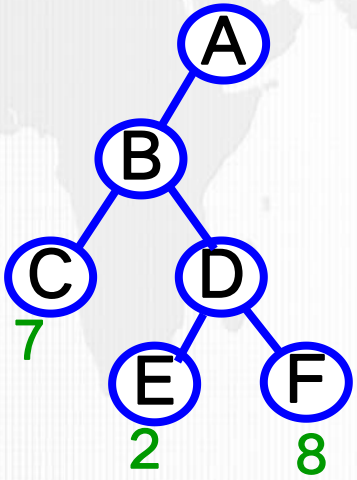
- 哈夫曼树：带权路径长度最短的树

路径：从树中一个结点到另一个结点之间的分支

路径长度：路径上的分支数

树的路径长度：从树根到每一个结点的路径长度之和

树的带权路径长度：树中所有叶子结点的带权路径长度之和



记作：$wpl = \sum_{k=1}^{n} w_k l_k$

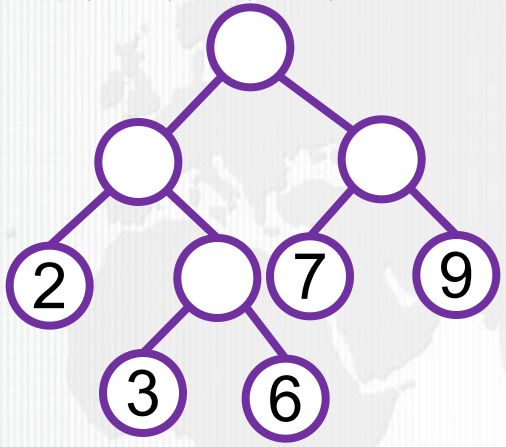
其中：w_k ——权值

l_k ——结点到根的路径长度

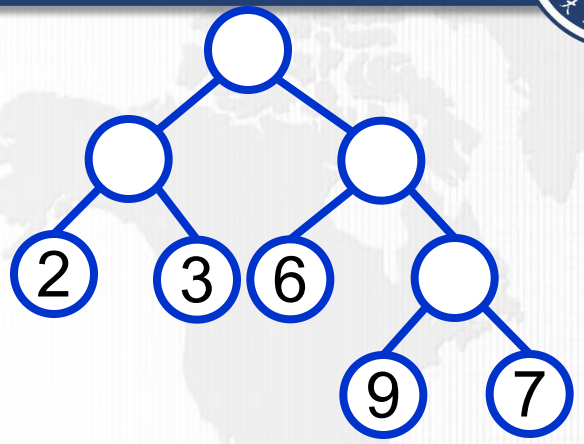
带权路径长度wpl——Weighted Path Length

Application

◆ 哈夫曼树：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



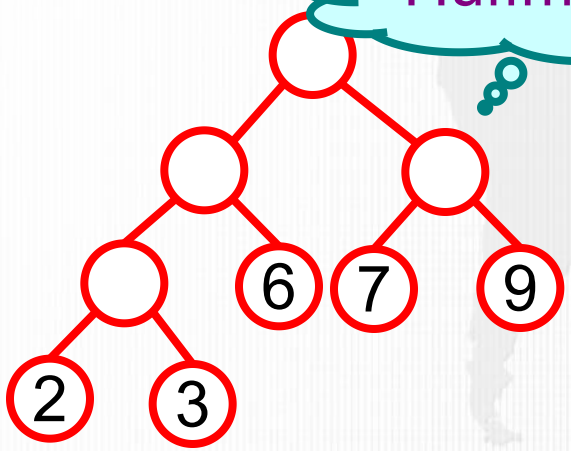
$$wpl = \sum_{k=1}^{n} w_k l_k$$



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

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

Huffman树



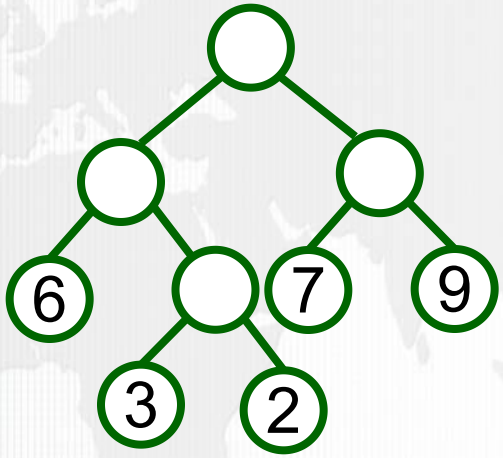
Huffman树的结构特点？

权值大的结点更靠近根结点

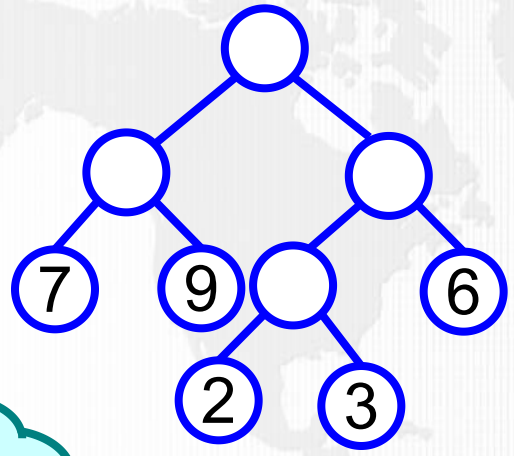
WPL= (6+7+9)*2+(2+3)*3 =59

Application

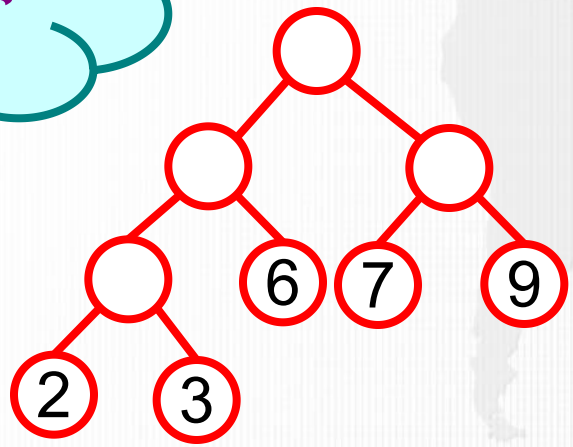
◆ 哈夫曼树：例有5个结点，权值分别为2，3，6，7，9构造有5个叶子结点的二叉树



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



Huffman树不唯一



Huffman树的结构特点？

权值大的结点更靠近根结点

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

Application

- 哈夫曼树的构造：

步骤如下：

1）根据给定的n个权值{w1,w2,……wn}，构造n棵只有根结点的二叉树(森林)，根结点的权值分别为wj

2）在森林中选取两棵根结点权值最小的二叉树作为左右子树，构造一棵新的二叉树，新二叉树根结点的权值为其左右孩子的权值之和

3）在森林中删除这两棵二叉树，并将新二叉树加入森林中

4）重复2、3步，直到森林中只含一棵二叉树为止，这棵树即哈夫曼树

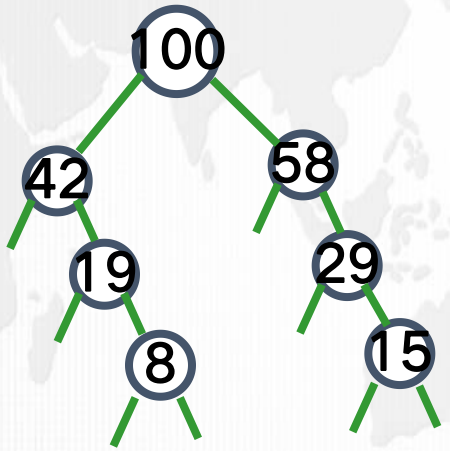
2、3步重复次数：n-1次

Application

- 哈夫曼树的构造：

例　w={5, 29, 7, 8, 14, 23, 3, 11}

5 29 7 8 14 23 3 11



一棵有n个叶子结点的Huffman树有2n-1个结点

n-1次合并，每次生成1个结点

带权路径长度

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

Application

◆ 哈夫曼树的应用——最佳判定树：

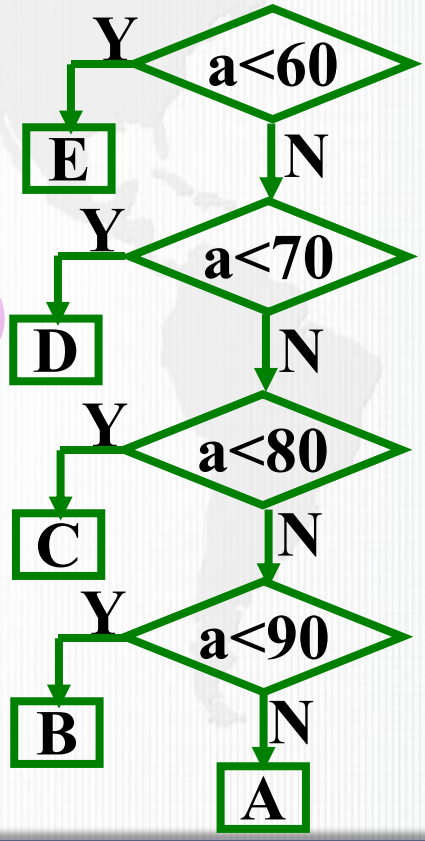
例 编制一个将百分制转换成五分制的程序

等级	E	D	C	B	A
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

```
if(a<60) b="E";
else if(a<70) b="D";
```

80%以上的学生
要经过2次以上的比较

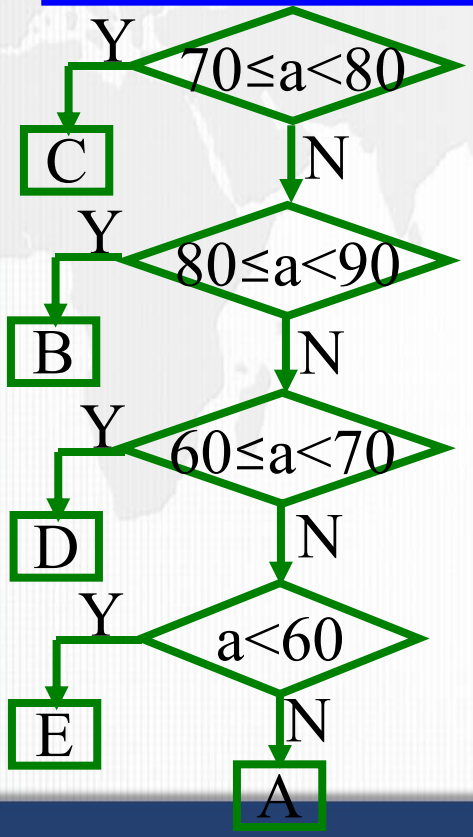
```
else if(a<80) b="C";
else if(a<90) b="B";
else  b="A";
```

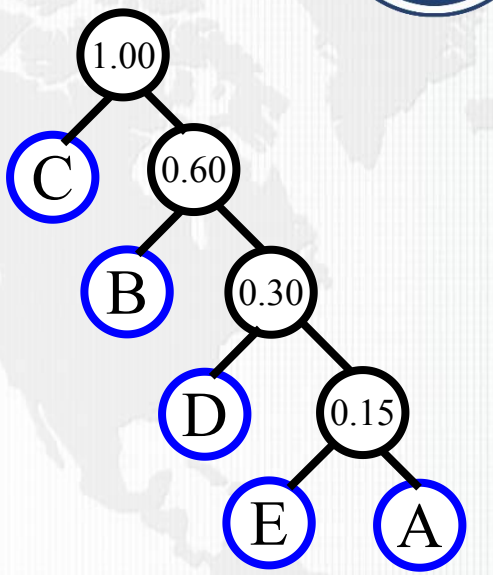


Application

◆ 哈夫曼树的应用——最佳判定树：

等级	E	D	C	B	A
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





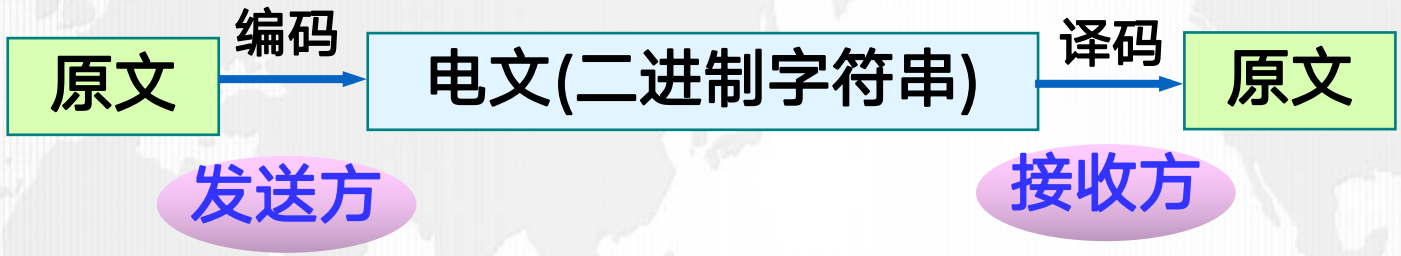
构造*以分数的分布比例为权值*的哈夫曼树可得最佳判定算法

Application

- 哈夫曼树的应用——哈夫曼编码：

  在数据通信中涉及到数据编码问题

  思想：根据字符出现频率编码，使电文总长最短



Application

◆ 哈夫曼树的应用——哈夫曼编码：

在数据通信中涉及到数据编码问题

思想：根据字符出现频率编码，使电文总长最短

例 要传输的原文为ABACCDA

等长编码 A：00 B：01 C：10 D：11

发送方：将ABACCDA 转换成 00010010101100

接收方：将00010010101100 还原为ABACCDA

不等长编码 A：0 B：00 C：1 D：01

发送方：将ABACCDA 转换成 000011010

接收方：000011010 转换成 AAAACCDA

BBCCDA

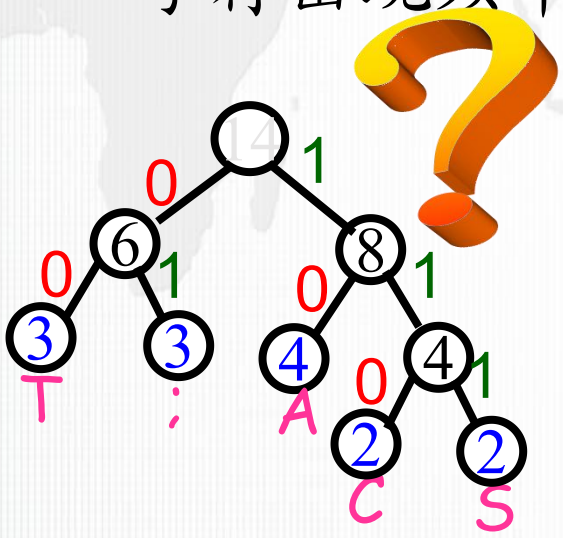
Application

- 哈夫曼树的应用——哈夫曼编码：

编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列

例 要传输的字符集 D={C,A,S,T,;}

字符出现频率 w={2,4,2,3,3}



哈夫曼编码为何能够保证：

①一个字符编码不是其它字符编码的前缀，即能够保证正确解码？

②同时，编码长度最短？

Application

◆ 哈夫曼树的应用——哈夫曼编码：

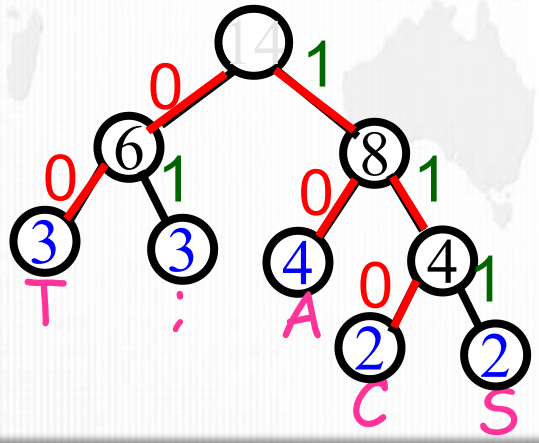
译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束

例 若发送电文是{CAS;CAT;SAT;AT}

其编码 “110 10 111 01 110100001111100011000”

若收到电文为 “1101000”

译文只能是 “CAT”



T : 00

; : 01

A : 10

C : 110

S : 111

Application

◆ 哈夫曼树的应用——哈夫曼编码：

- 二叉树的结构：在Huffman树中，字符（或其它数据项）仅存在于叶子节点中，非叶子节点不代表任何字符。由于编码是根据从根节点到叶子节点的路径生成的（例如，左分支代表0，右分支代表1），所以只有到达叶子节点时才完成了一个完整的字符编码。

- 编码路径：考虑任何给定的叶子节点（代表一个字符）。其编码是从根节点到该叶子节点的路径。由于其他字符也是其他叶子节点，所以它们的路径必然在某个点有所不同。而且，由于非叶子节点不代表字符，所以不存在一个字符的编码是另一个字符编码的前缀的情况。

- 构建过程：在构建Huffman树的过程中，我们始终是合并频率最低的两个节点。这确保了高频字符位于树的较高层，而低频字符位于较低层。但无论如何，由于叶子节点的独特性，我们不可能得到一个字符的编码是另一个字符编码的前缀。

Summary

- 二叉树是一种最常用的树形结构，二叉树具有一些特殊的性质共5种，而满二叉树和完全二叉树又是两种特殊形态的二叉树。

- 二叉树有两种存储表示：顺序存储和链式存储。顺序存储就是把二叉树的所有结点按照层次顺序存储到连续的存储单元中，这种存储更适用于完全二叉树。链式存储又称二叉链表，每个结点包括两个指针，分别指向其左孩子和右孩子。链式存储是二叉树常用的存储结构。

- 树的存储结构有三种：双亲表示法、孩子表示法和孩子兄弟表示法，孩子兄弟表示法是常用的表示法，任意一棵树都能通过孩子兄弟表示法转换为二叉树进行存储。森林与二叉树之间也存在相应的转换方法，通过这些转换，可以利用二叉树的操作解决一般树的有关问题。

Summary

- 二叉树的遍历算法是其他运算的基础，通过遍历得到了二叉树中结点访问的线性序列，实现了非线性结构的线性化。根据访问结点的次序不同可得三种遍历：先序遍历、中序遍历、后序遍历，时间复杂度均为O(n)。

- 在线索二叉树中，利用二叉链表中的n+1个空指针域来存放指向某种遍历次序下的前驱结点和后继结点的指针，这些附加的指针就称为“线索”。引入二叉线索树的目的是加快查找结点前驱或后继的速度。

- 哈夫曼树在通信编码技术上有广泛的应用，只要构造了哈夫曼树，按分支情况在左路径上写代码0,右路径上写代码1，然后从上到下叶结点相应路径上的代码序列就是该叶结点的最优前缀码，即哈夫曼编码。