人工智能学院

Data Structures and Algorithms

Trees and Binary Trees2

本章内容

1. Review
2. Forest Conversion
3. Binary Tree Iteration
4. Tree Iteration
5. Forest Iteration
6. Application

Review

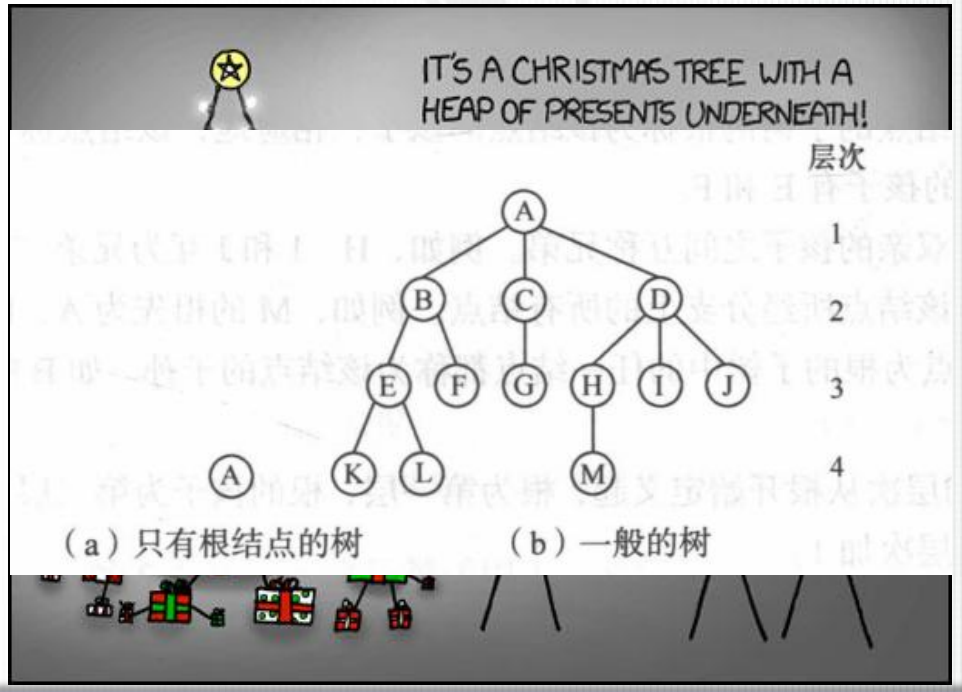
◆ 树的定义：

树是n（n>=0）个结点的有限集。当n =0时，称为空树。树的结构定义是一个递归的定义。在任意一棵非空树中应满足:

1）有且仅有一个特定的称为根的结点。

2）当n >1时，其余结点可分为m（m>0）个互不相交的有限集，其中每个集合本身又是一棵树，并且称为根的子树。

例如，在右图中，（a）是只有一个根结点的树；（b）是有13个结点的树，其中A是根，其余结点分成3个互不相交的子集：T1 = {B, E, F, K, L}, T2 = {C, G}, T3={D, H, I, J, M}。T1、T2、T3都是根A的子树，且本身也是一棵树。例如T1，其根为B,其余结点分为两个互不相交的子集：T11 = 、{E，K，L}, T12={F}。



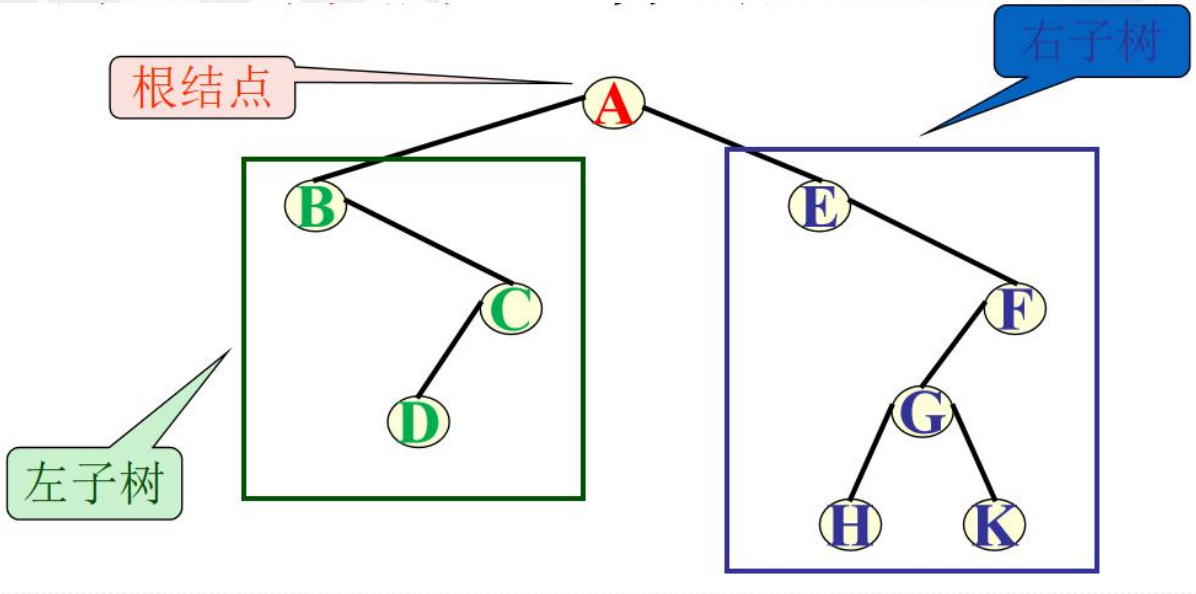
Review

◆ 二叉树的定义：

二叉树(Binary Tree)是n（n>=0）个结点所构成的集合，它或为空树（n=0）;或为非空树（n>0）, 对于非空树T:

1)有且仅有一个称之为根的结点；

2)除根结点以外的其余结点分为两个互不相交的子集T1和T2 分别称为T的左子树和右子树，且T1和T2本身又都是二叉树。

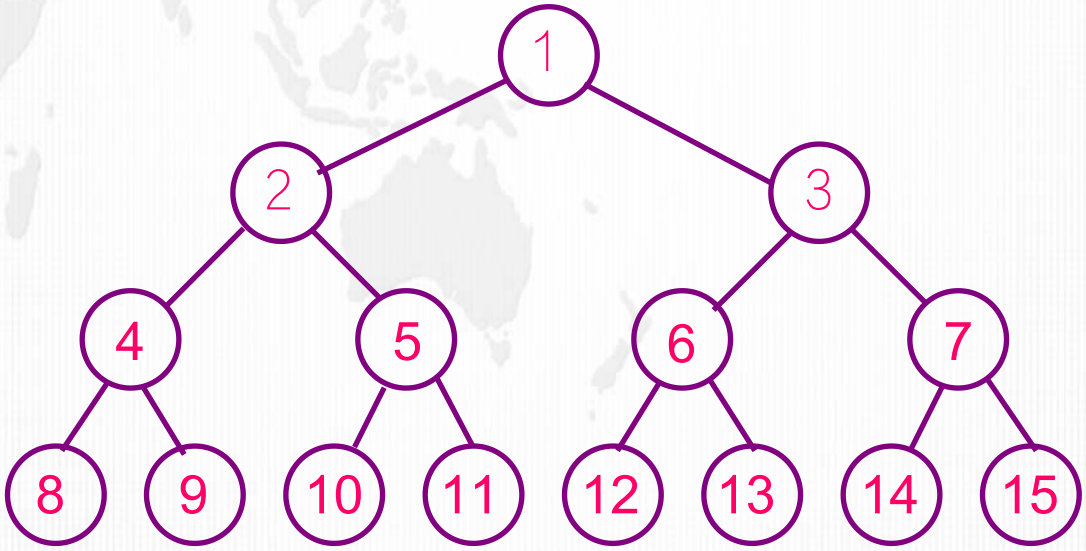


Review

- 几个特殊的二叉树：

满二叉树(Complete Binary Tree)

一棵高度为知且含有2^h-1个结点的二叉树称为满二叉树，即树中的每层都含有最多的结点，满二叉树的叶子结点都集中在二叉树的最下一层，并且除叶子结点之外的每个结点度数均为2。可以对满二叉树按层序编号：约定编号从根结点（根结点编号为1）起，自上而下，自左向右。这样，每个结点对应一个编号，对于编号为i的结点，若有双亲，则其双亲为$\lfloor i/2 \rfloor$，若有左孩子，则左孩子为2i；若有右孩子，则右孩子为2i+1。

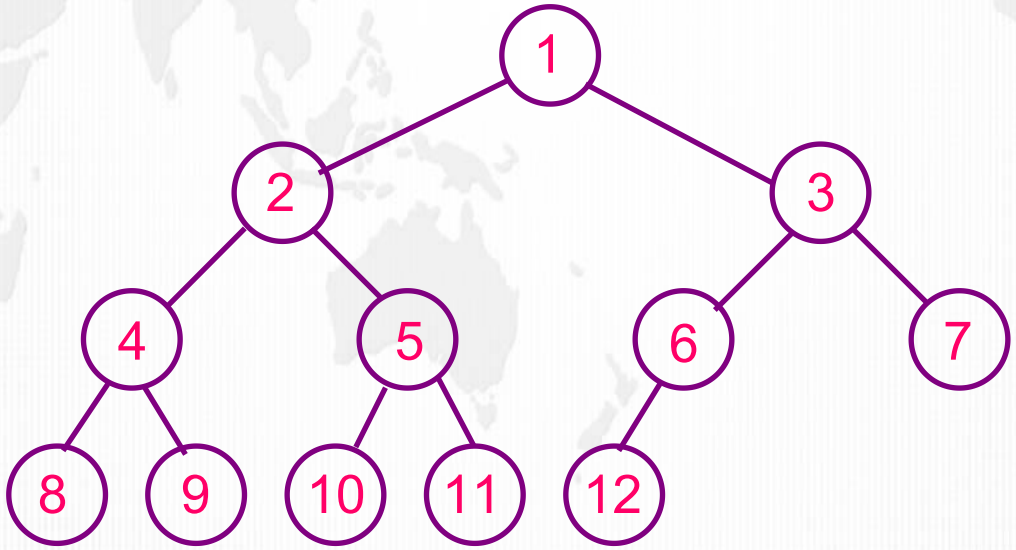


Review

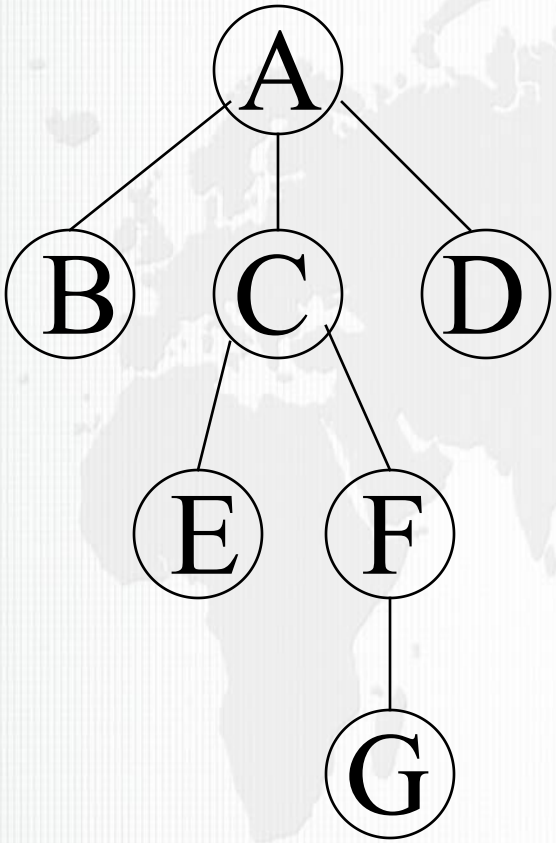
- 几个特殊的二叉树：

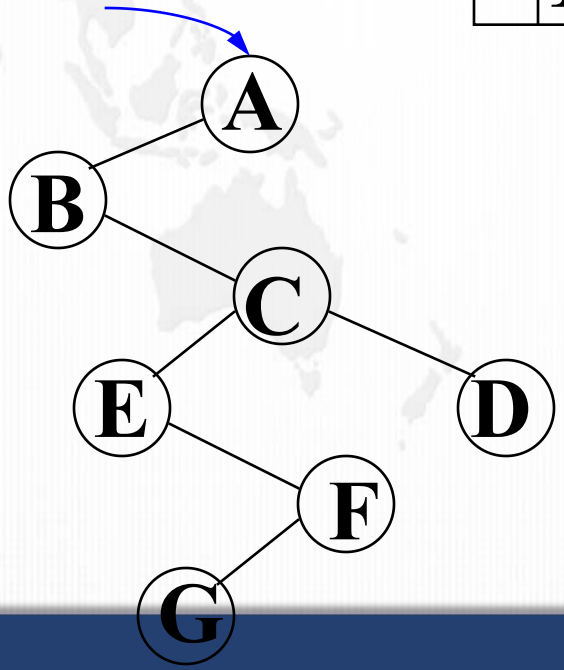
  完全二叉树

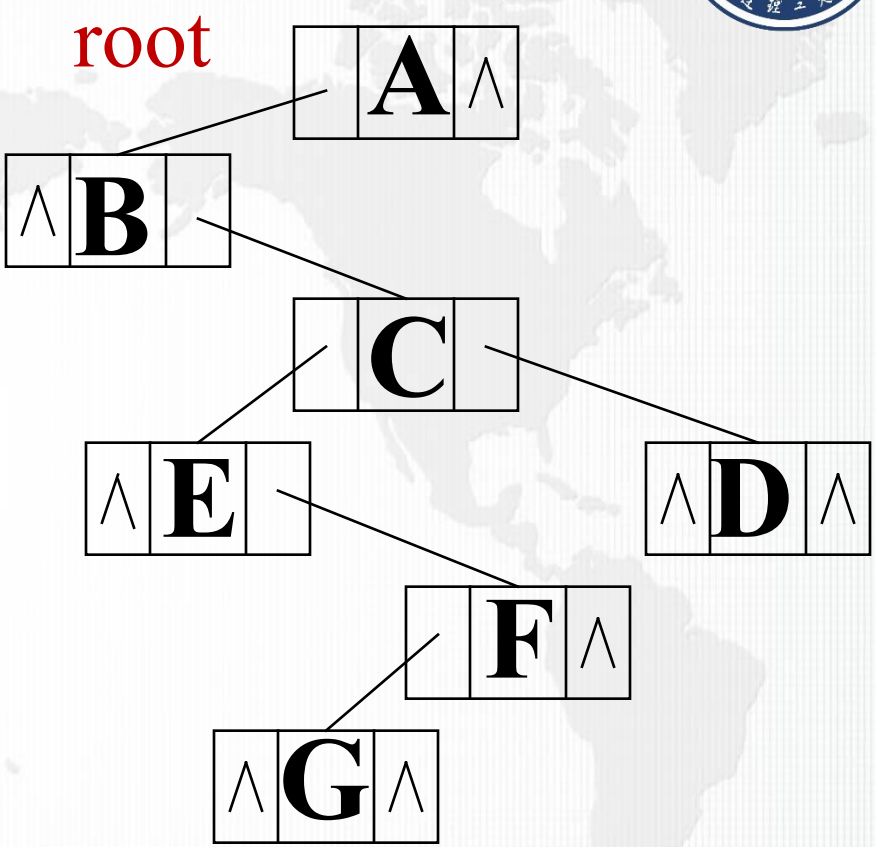
  高度为h、有n个结点的二叉树，当且仅当其每个结点都与高度为h的满二叉树中编号为 1-n的结点一一对应时，称为完全二叉树，（完全二叉树就是对应相同高度的满二叉树缺失最下层最右边的一些连续叶子结点）



Forest Conversion

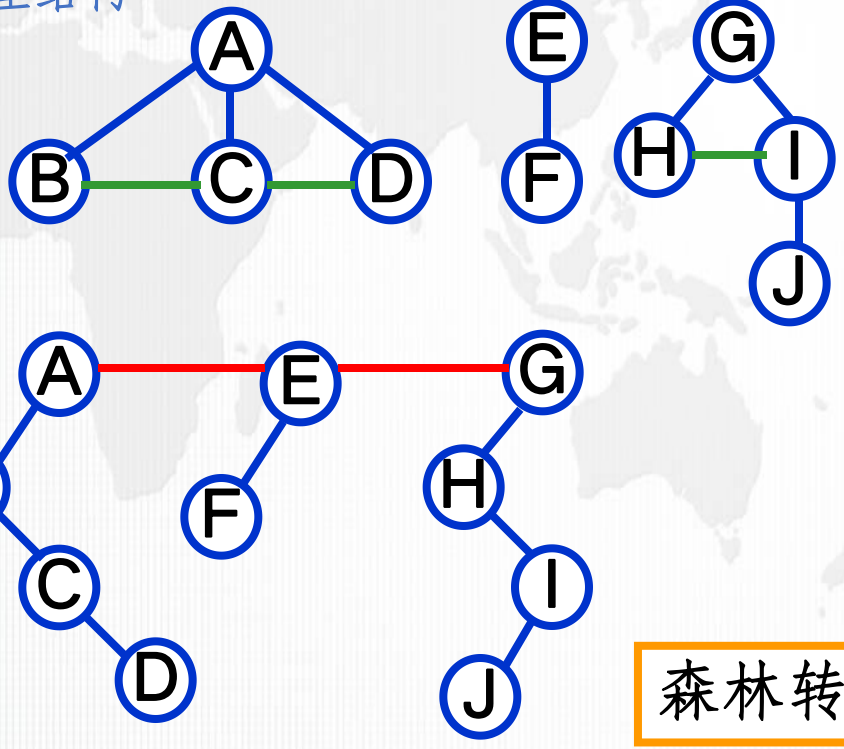


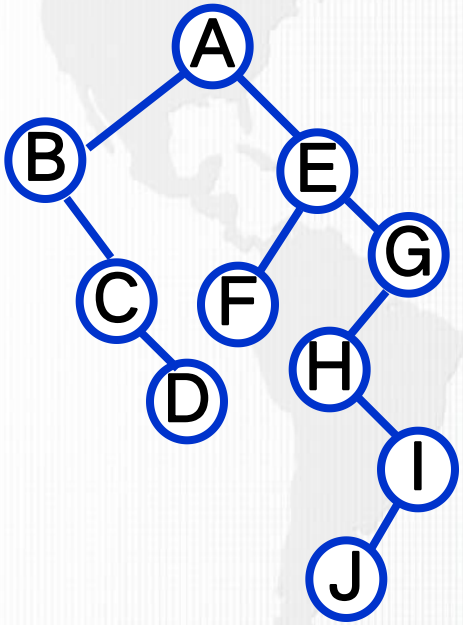




Forest Conversion

- 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





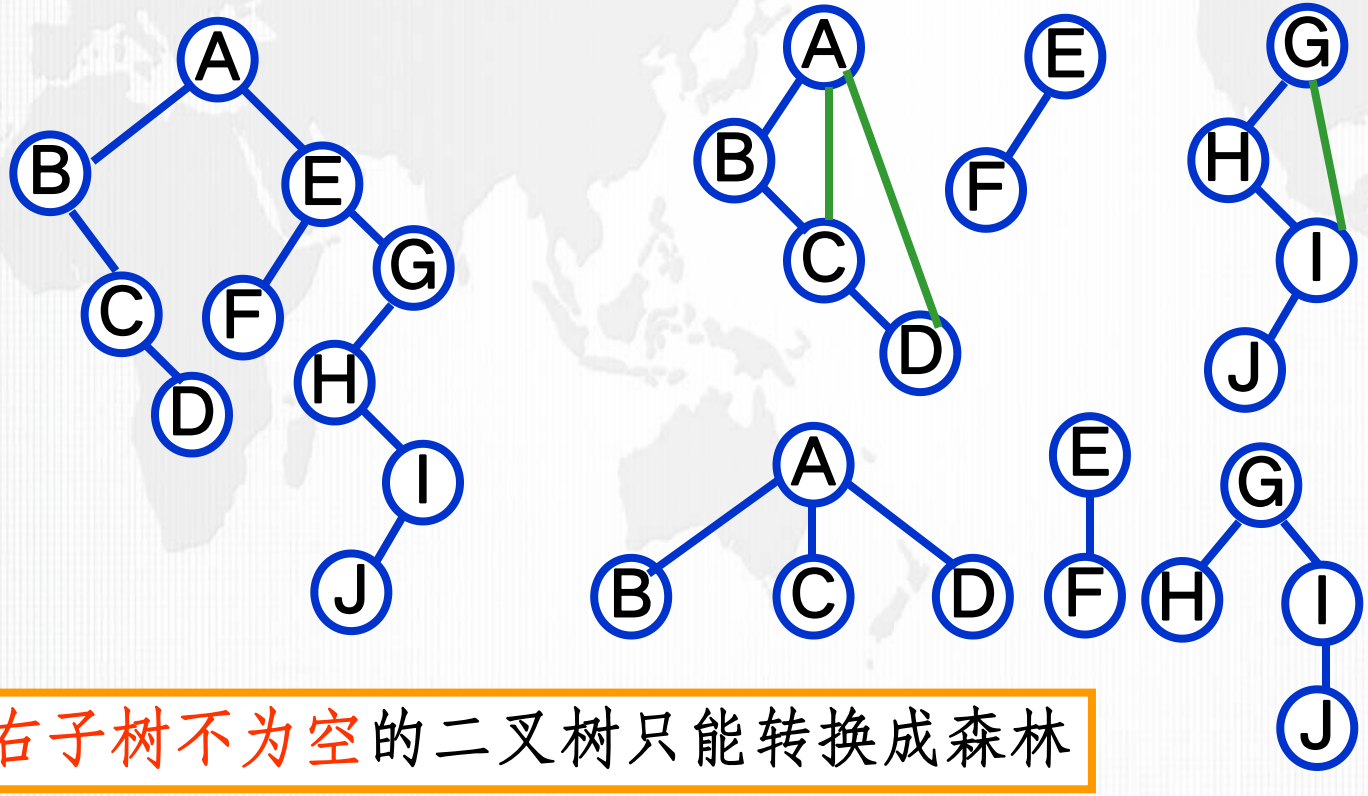
森林转换成的二叉树其右子树非空

Forest Conversion

◆ 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



右子树不为空的二叉树只能转换成森林

Forest Conversion

◆ 实际应用

数据表示和存储：

- 简化存储结构：在某些情况下，将复杂的森林结构转化为二叉树可以简化数据的存储和表示，节省存储空间，并使数据操作如插入、删除、修改等更加简洁高效
- 扁平化数据结构：在处理嵌套的树形数据结构（例如 JSON 对象）时，转换为二叉树可以将数据扁平化，便于进行进一步的数据处理和分析。

数据检索：

- 优化搜索算法：将森林转化为二叉树可以利用二叉搜索树或平衡二叉树（例如 AVL 树）的性质，实现更高效的数据检索。
- 查询性能的提升：对于数据库查询操作，通过使用二叉树结构可以加快数据的查询速度。

Binary Tree Iteration

- 问题提出：

  顺着某一条搜索路径巡访二叉树中的结点，使得每个结点均被访问一次，而且仅被访问一次。

  - 访问：“访问”的含义可以很广，如：对结点进行处理、输出结点的信息等
  - 遍历：“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，每个结点最多有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题。

Binary Tree Iteration

- 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

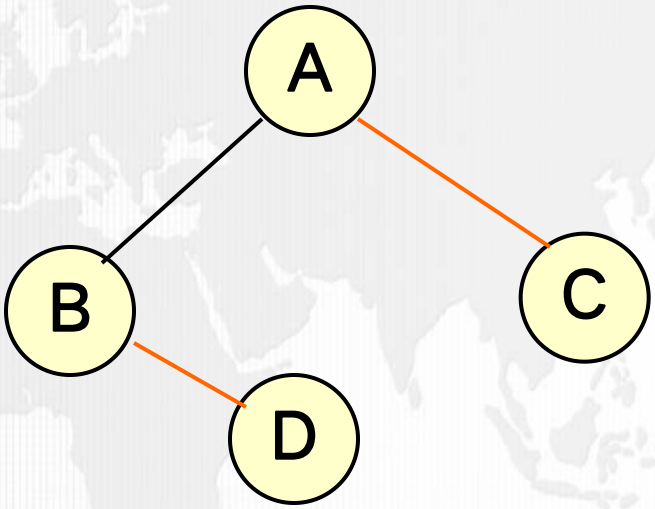
  中（根）序的遍历算法：先中序遍历左子树，然后访问根结点，最后中序遍历右子树

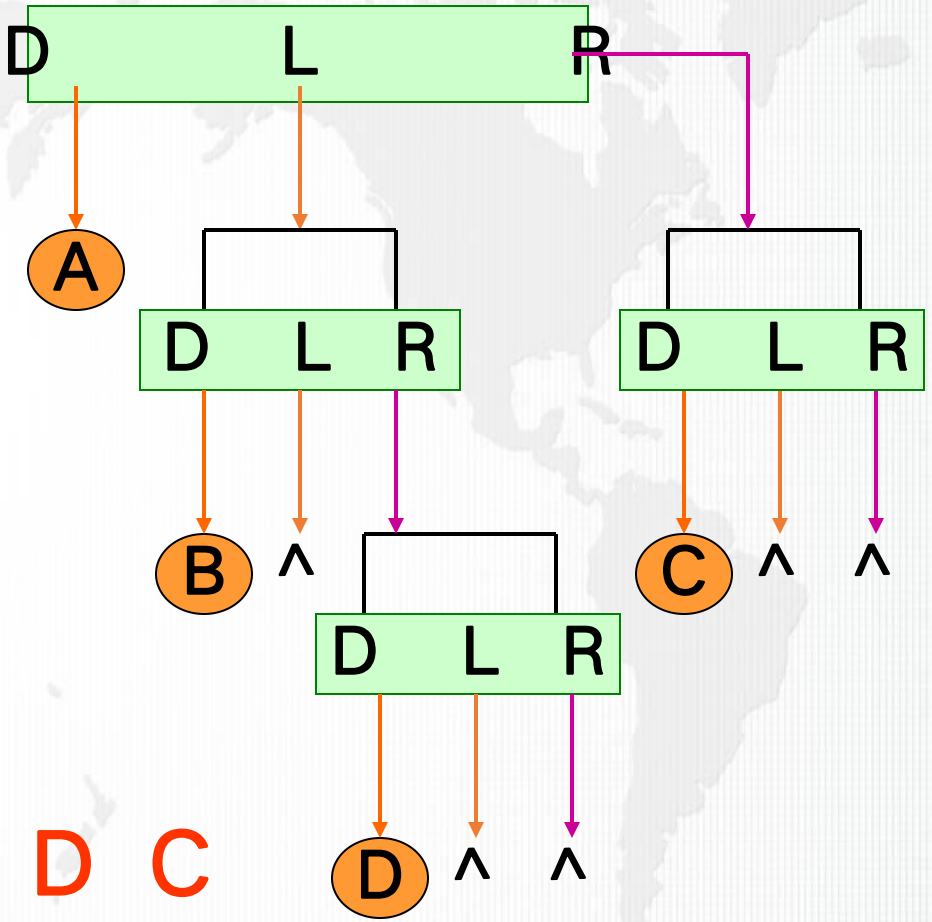
  后（根）序的遍历算法：先后序遍历左、右子树，然后访问根结点

  层次遍历算法：从上到下、从左到右访问各结点

Binary Tree Iteration

- 先序遍历：

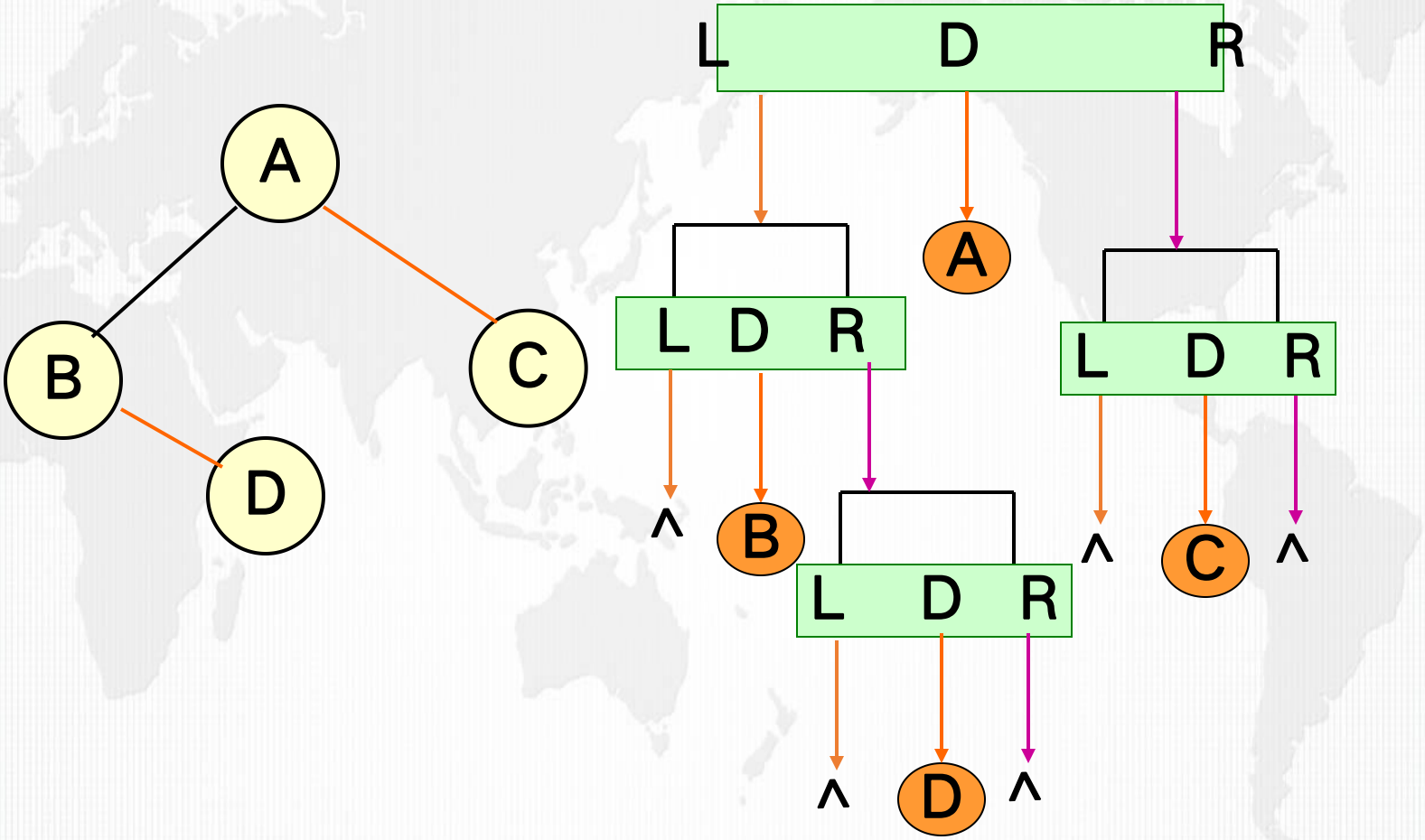




先序遍历序列：A B D C

Binary Tree Iteration

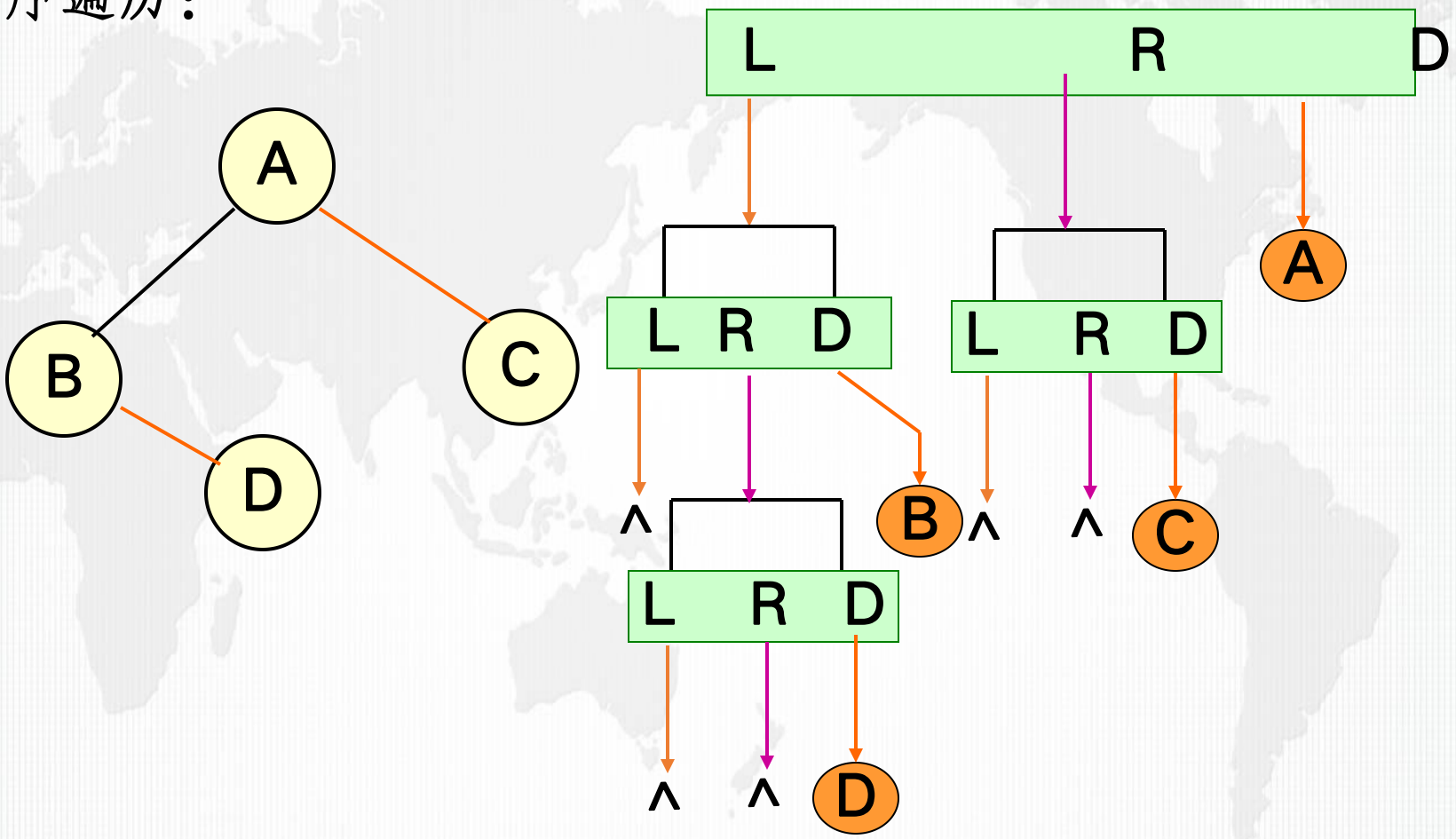
◆ 中序遍历：



中序遍历序列：B D A C

Binary Tree Iteration

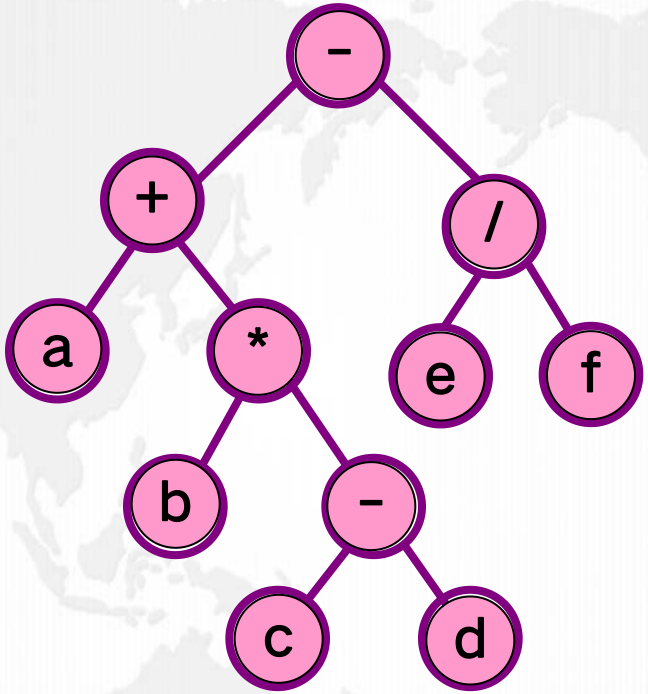
◆ 后序遍历：



后序遍历序列： D B C A

Binary Tree Iteration

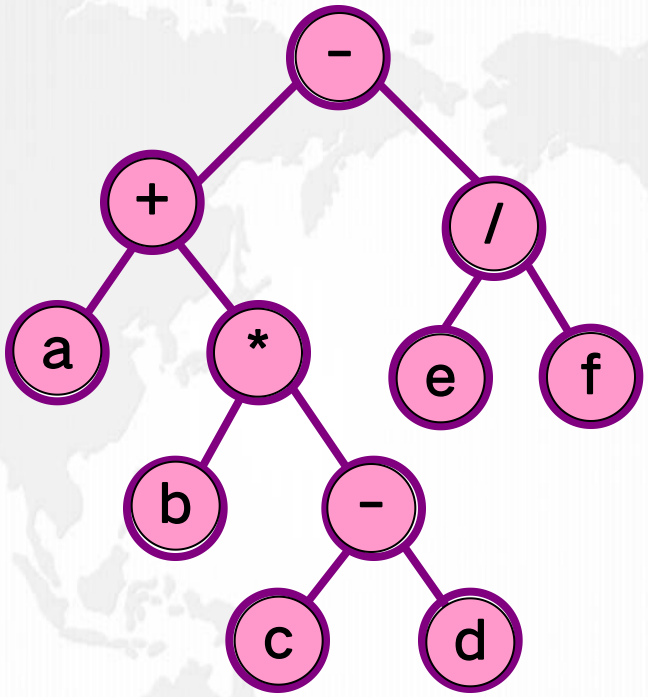
- 遍历举例：



先序遍历：-+a*b-cd/ef

Binary Tree Iteration

◆ 遍历举例：

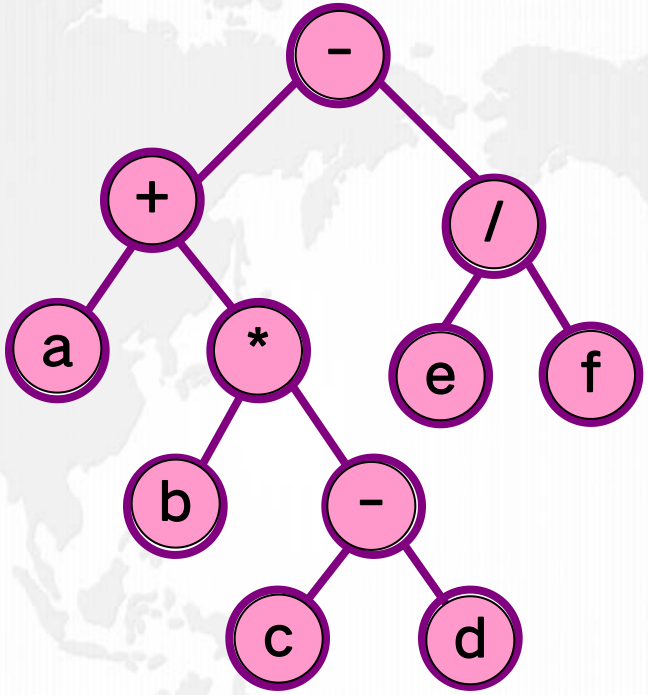


先序遍历：-+a*b-cd/ef

中序遍历：a+b*c-d-e/f

Binary Tree Iteration

◆ 遍历举例：



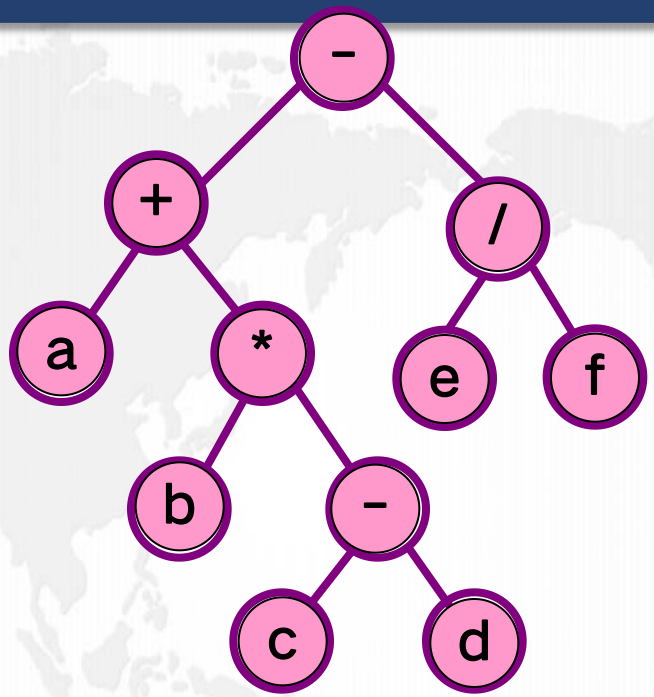
先序遍历：-+a*b-cd/ef

中序遍历：a+b*c-d-e/f

后序遍历：abcd-*+ef/-

Binary Tree Iteration

◆ 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

Binary Tree Iteration

◆ 算法的递归描述——先序遍历二叉树：

```
61
62  void preOrderTraversal(Node* root) {
63      if(root != NULL) {
64          printf("%c ", root->data); // 访问根节点
65          preOrderTraversal( root: root->left);  // 递归遍历左子树
66          preOrderTraversal( root: root->right); // 递归遍历右子树
67      }
68  }
69
```

Binary Tree Iteration

◆ 算法的递归描述——中序遍历二叉树：

```
70  void inOrderTraversal(Node* root) {
71      if(root != NULL) {
72          inOrderTraversal( root: root->left);  // 递归遍历左子树
73          printf("%c ", root->data); // 访问根节点
74          inOrderTraversal( root: root->right); // 递归遍历右子树
75      }
76  }
```

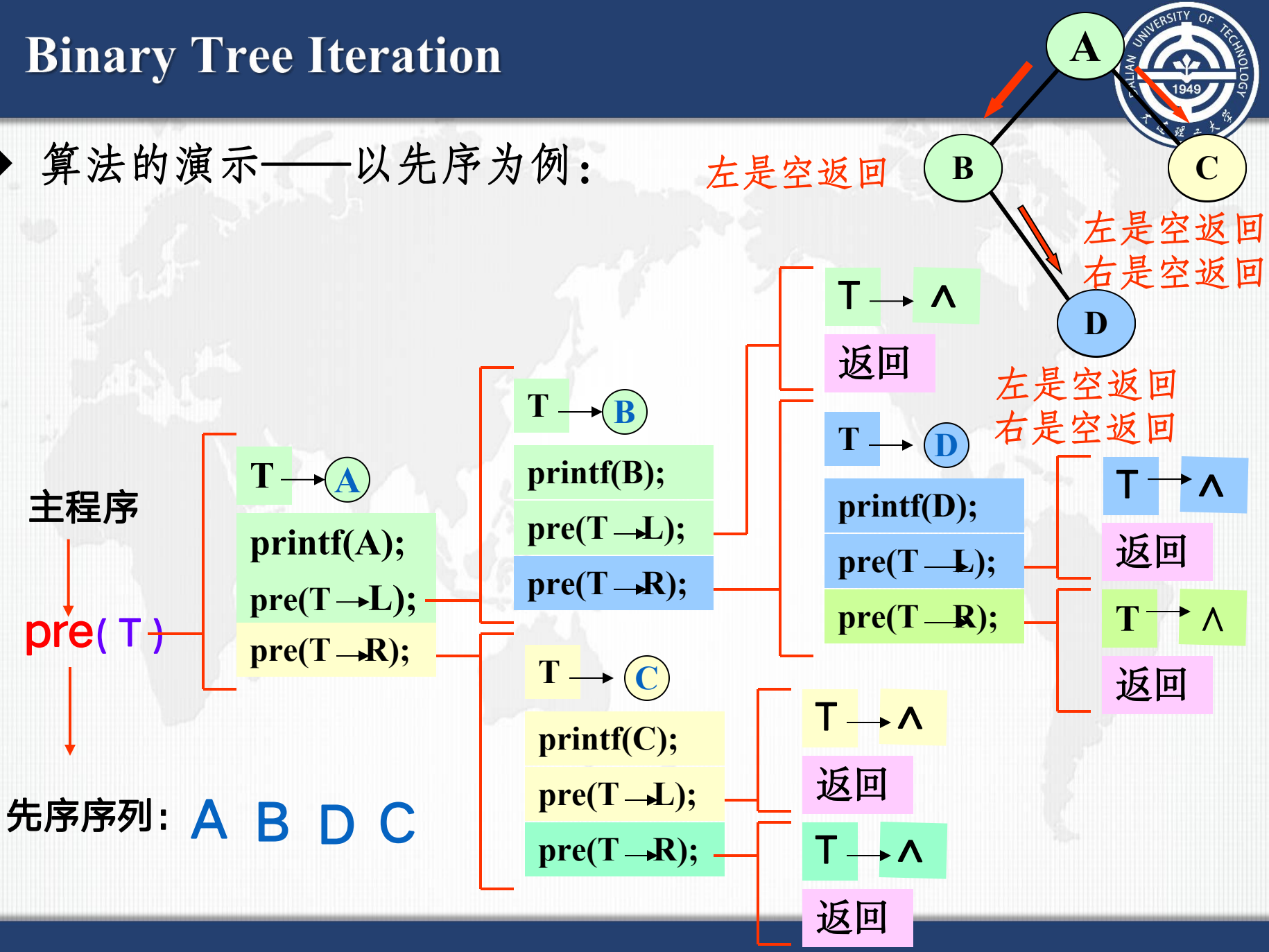
Binary Tree Iteration

◆ 算法的递归描述——后序遍历二叉树：

```
78  void postOrderTraversal(Node* root) {
79      if(root != NULL) {
80          postOrderTraversal( root: root->left);  // 递归遍历左子树
81          postOrderTraversal( root: root->right); // 递归遍历右子树
82          printf("%c ", root->data); // 访问根节点
83      }
84  }
```

Binary Tree Iteration

◆ 算法的演示——以先序为例：



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

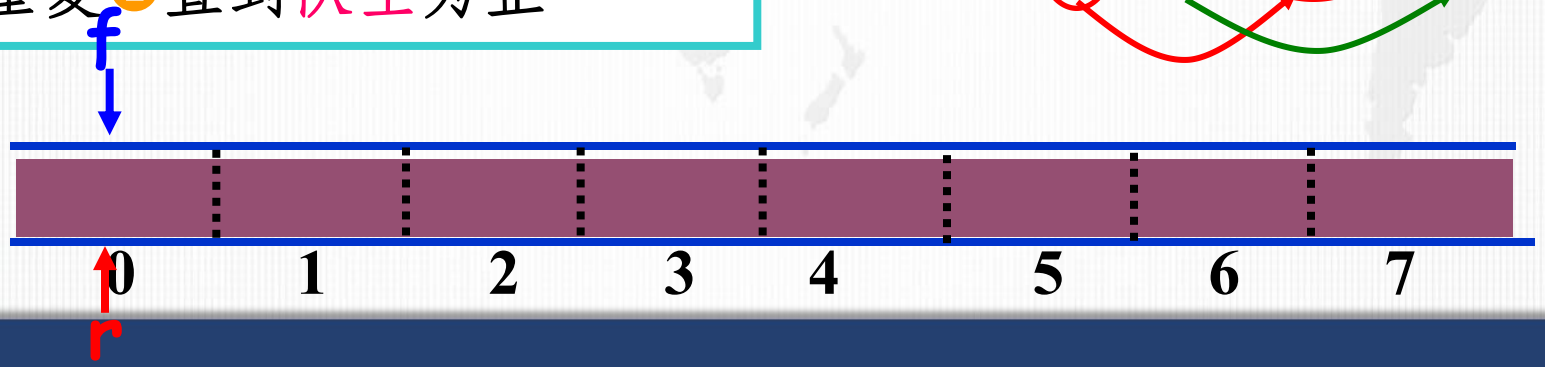
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D E F G



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

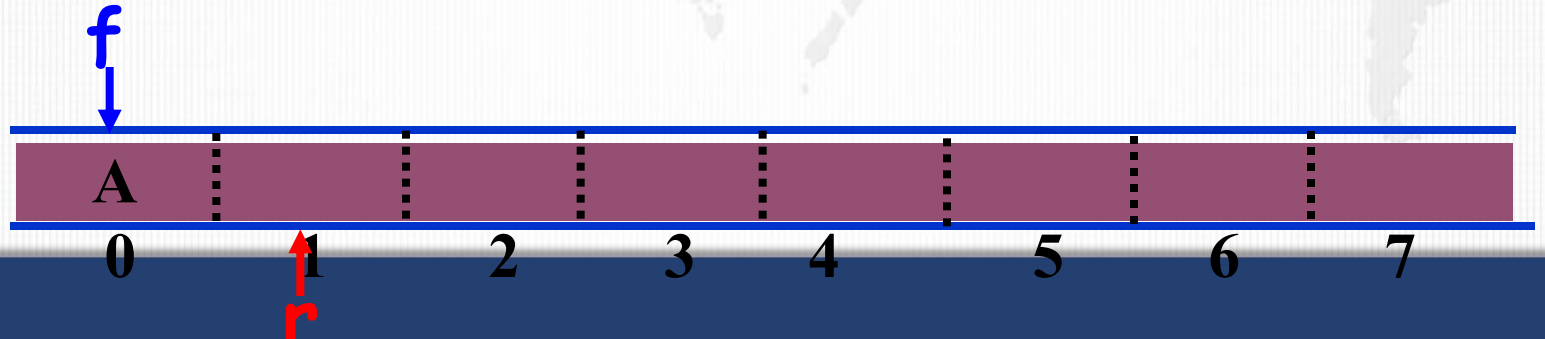
★算法思想：

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



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

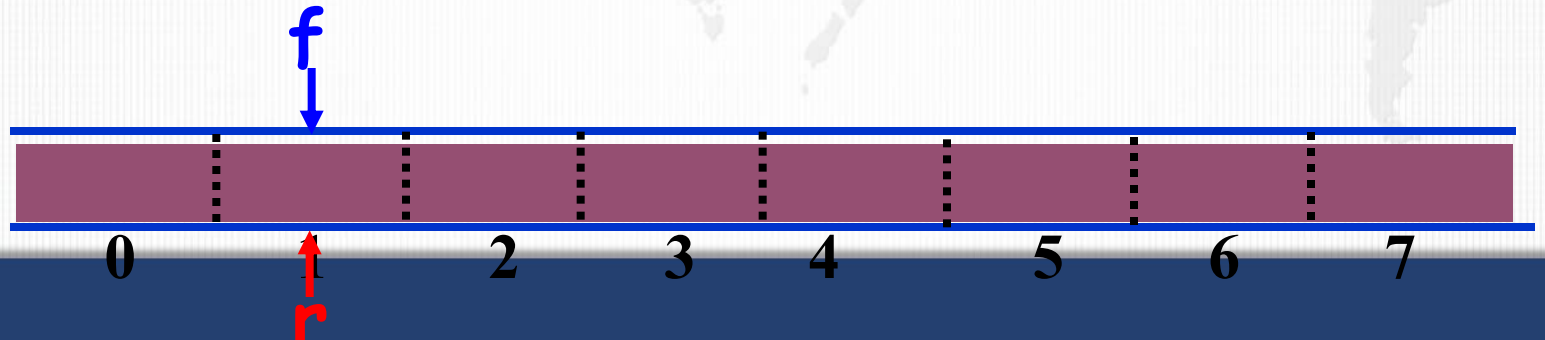
利用队列实现二叉树的层次遍历非递归算法

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

访问：A



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

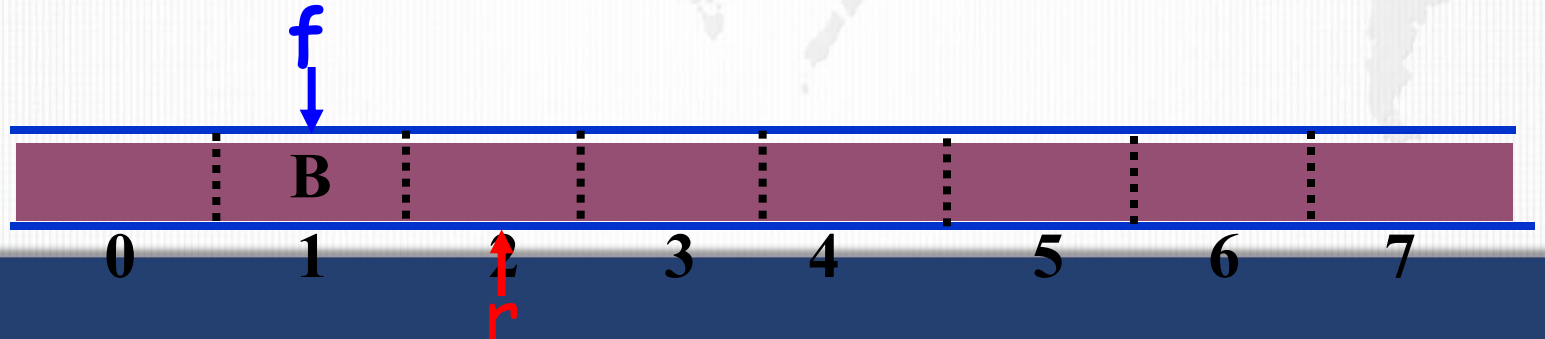
利用队列实现二叉树的层次遍历非递归算法

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

访问：A



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

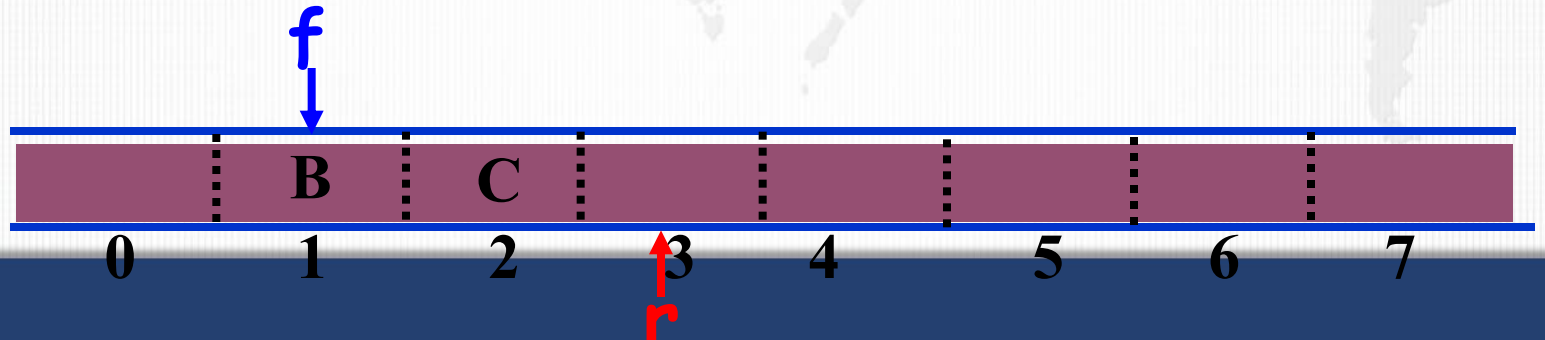
利用队列实现二叉树的层次遍历非递归算法

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

访问：A



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

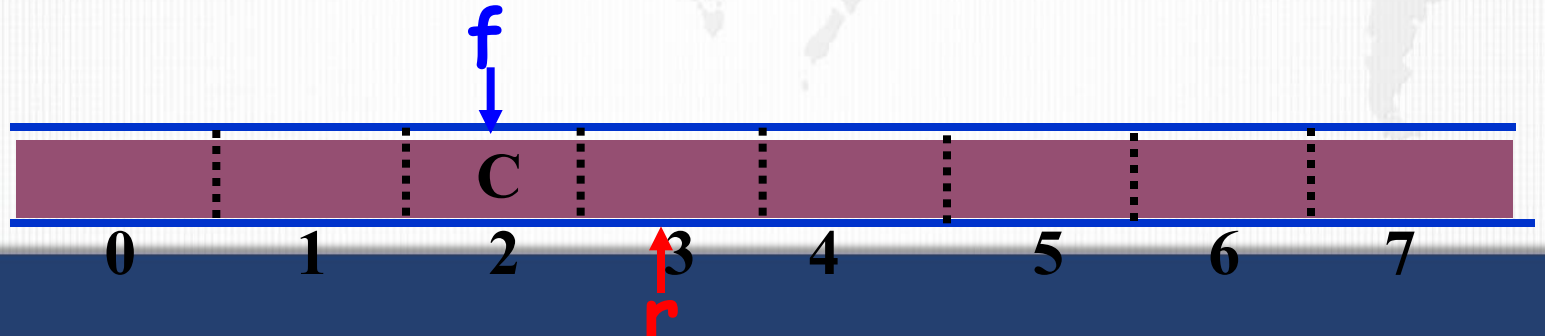
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

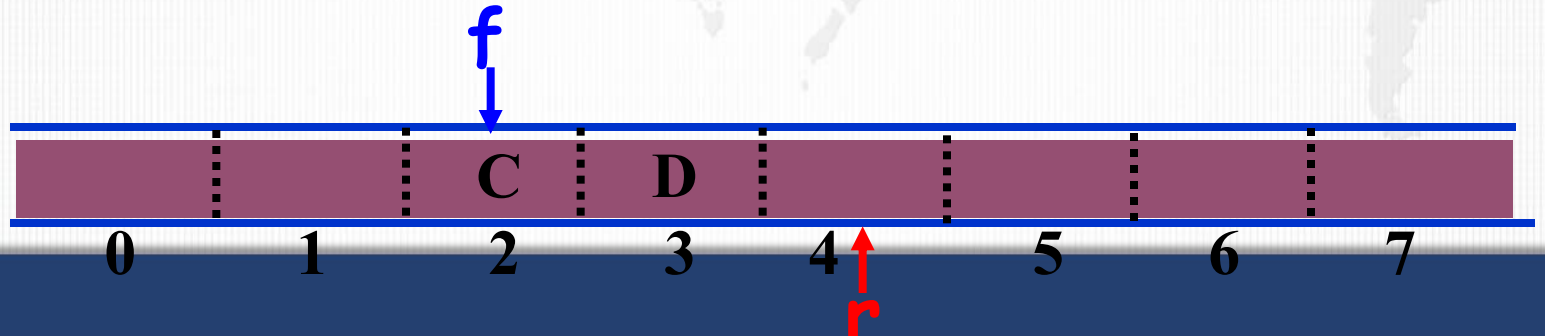
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

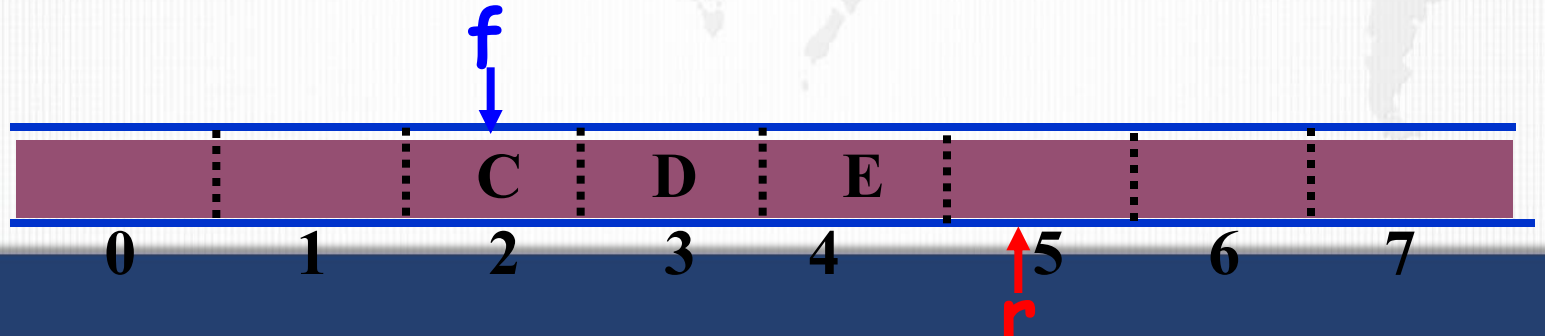
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

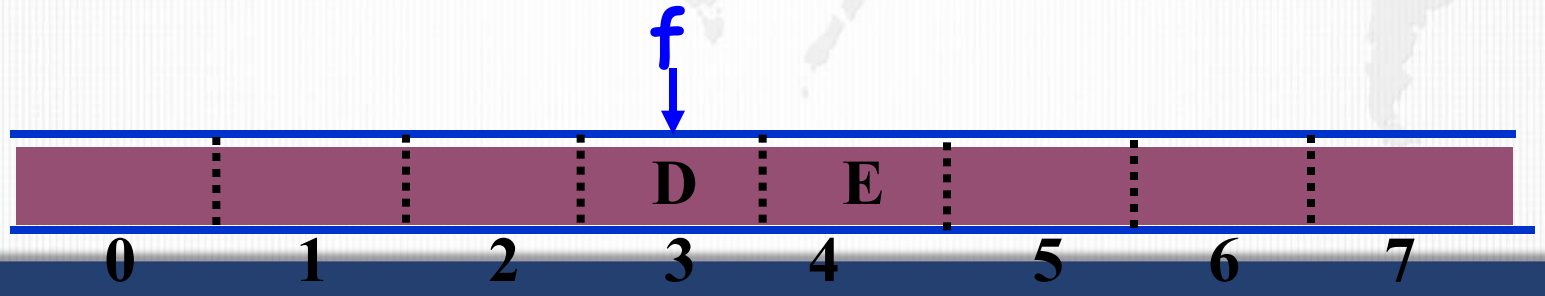
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

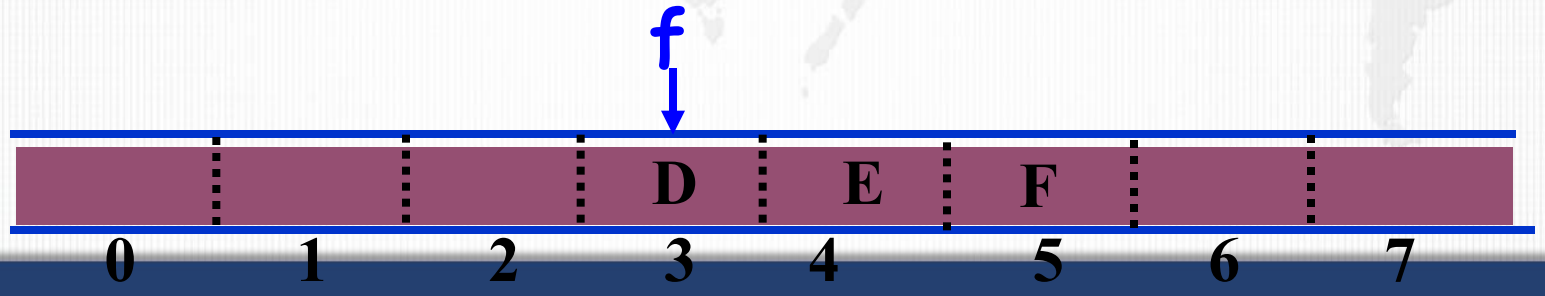
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

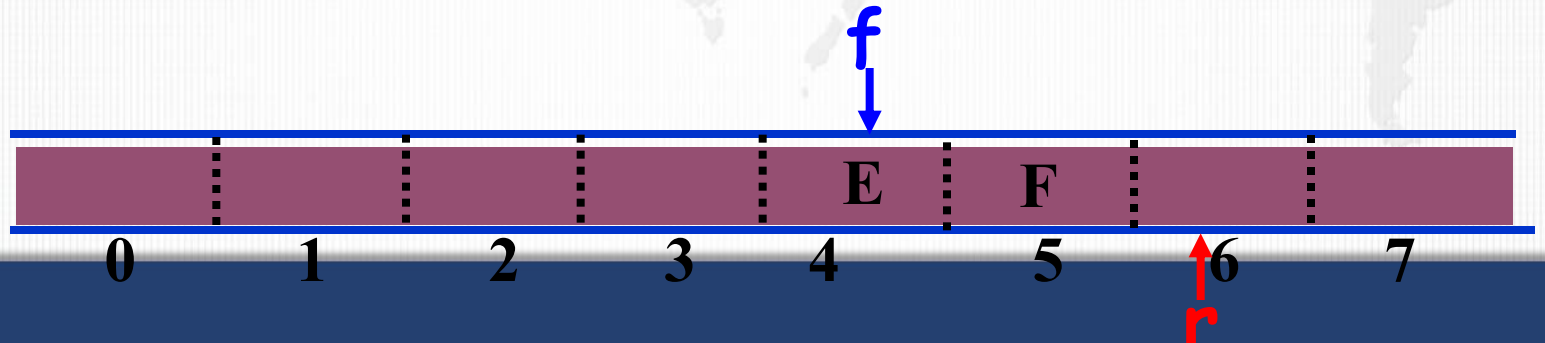
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

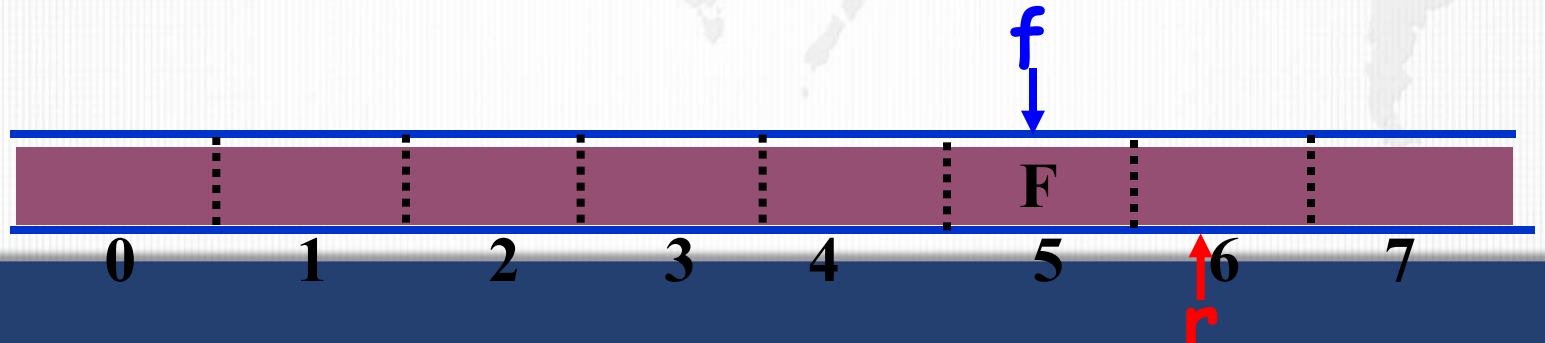
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D E



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

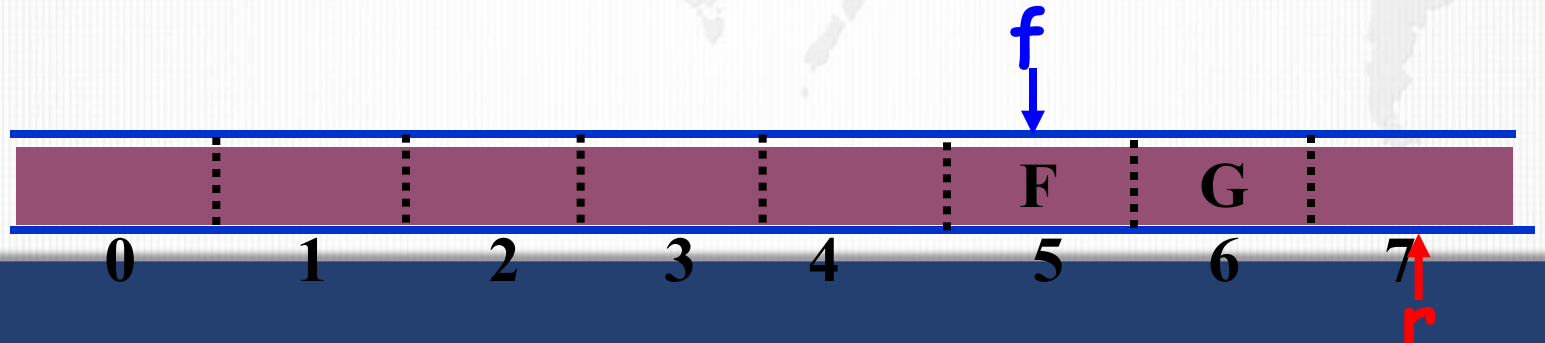
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D E



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

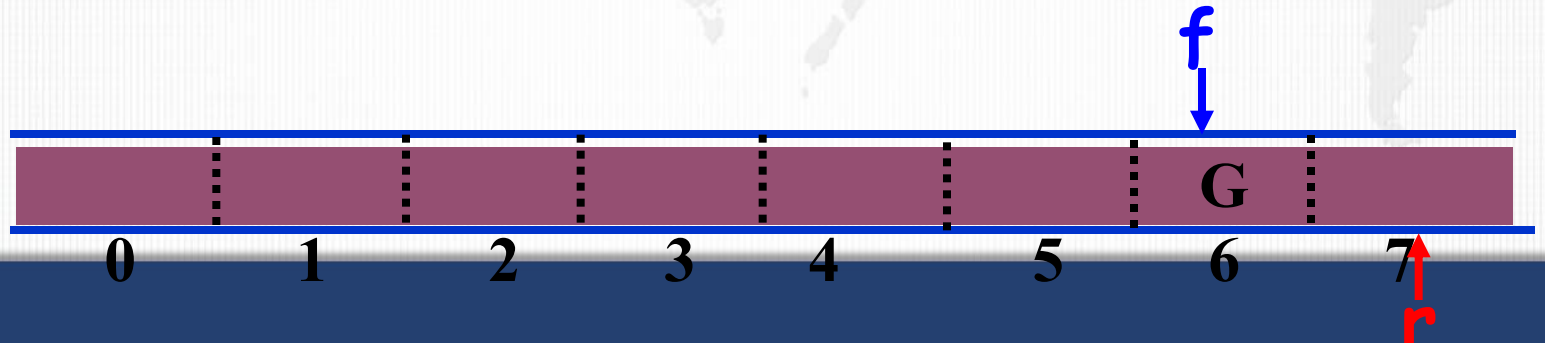
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D E F



Binary Tree Iteration

◆ 算法的非递归描述——层次遍历二叉树：

★算法思想：

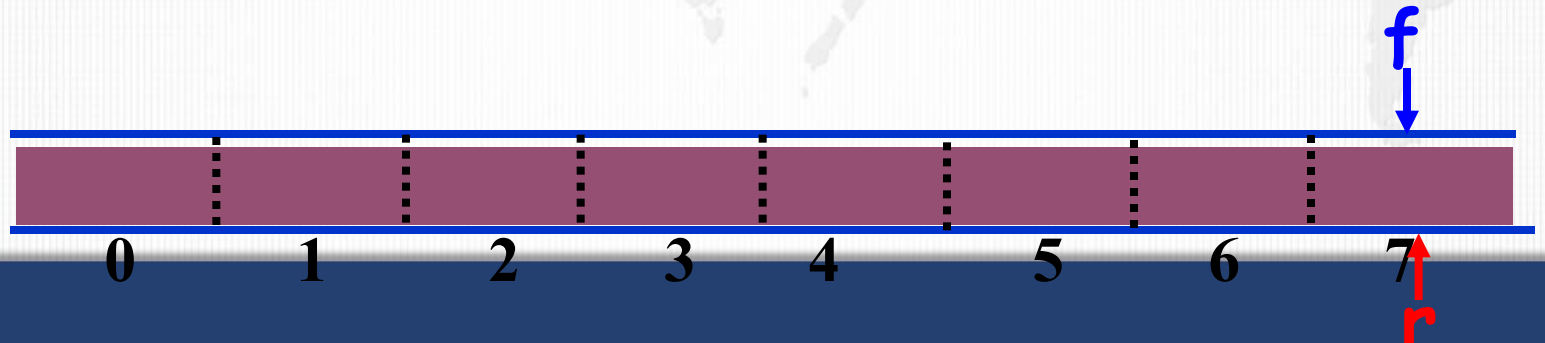
利用队列实现二叉树的层次遍历非递归算法

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

访问：A B C D E F G



Binary Tree Iteration

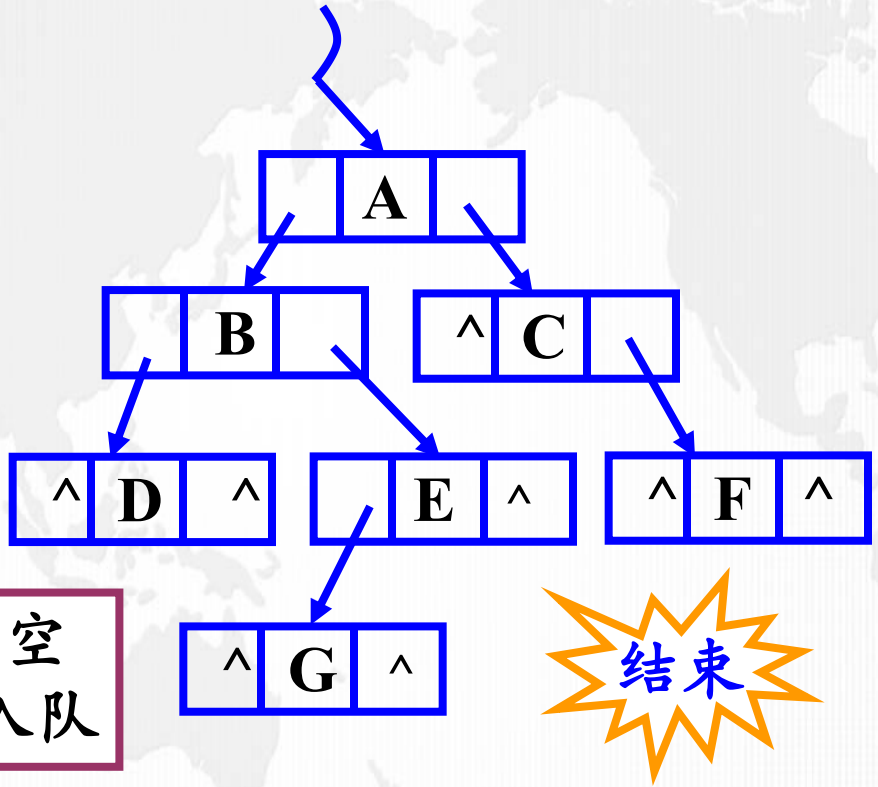
◆ 算法的非递归描述——层次遍历二叉树：

根结点入队

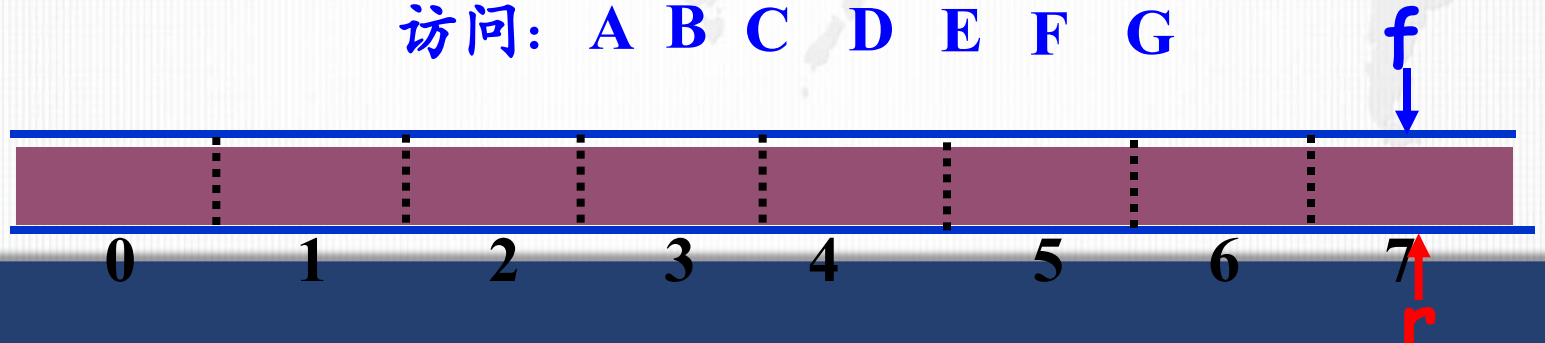
当队列不为空时

出队

将非空孩子入队



访问：A B C D E F G



Tree Iteration

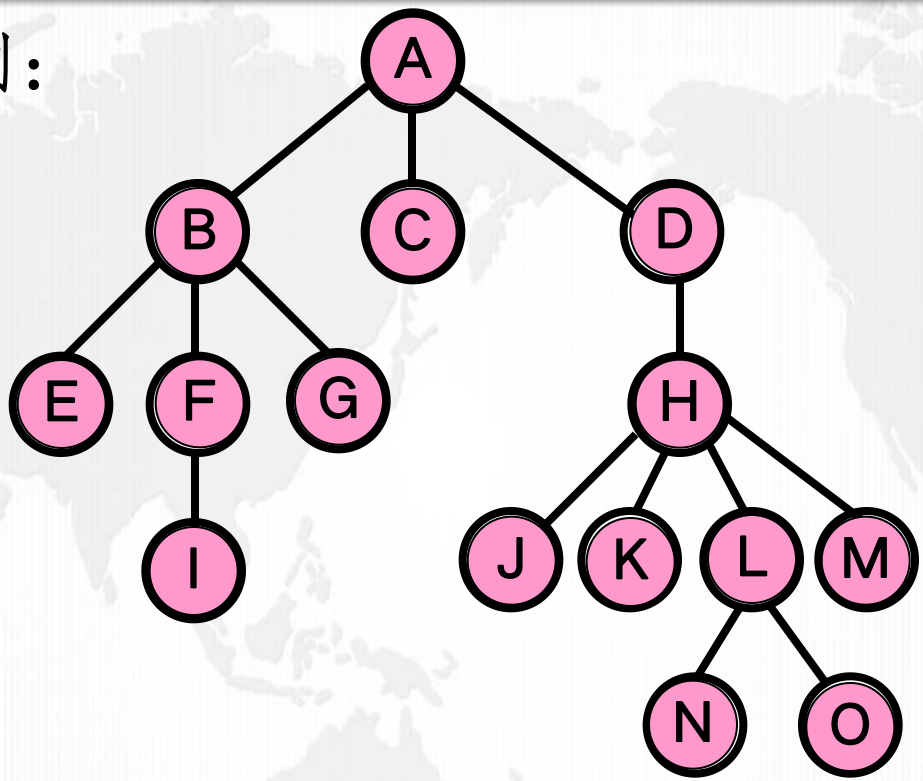
- 树的遍历常用方法：
  - 先根（序）遍历：先访问树的根结点，然后依次先根遍历根的每棵子树　　“根左右”

  - 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

Tree Iteration

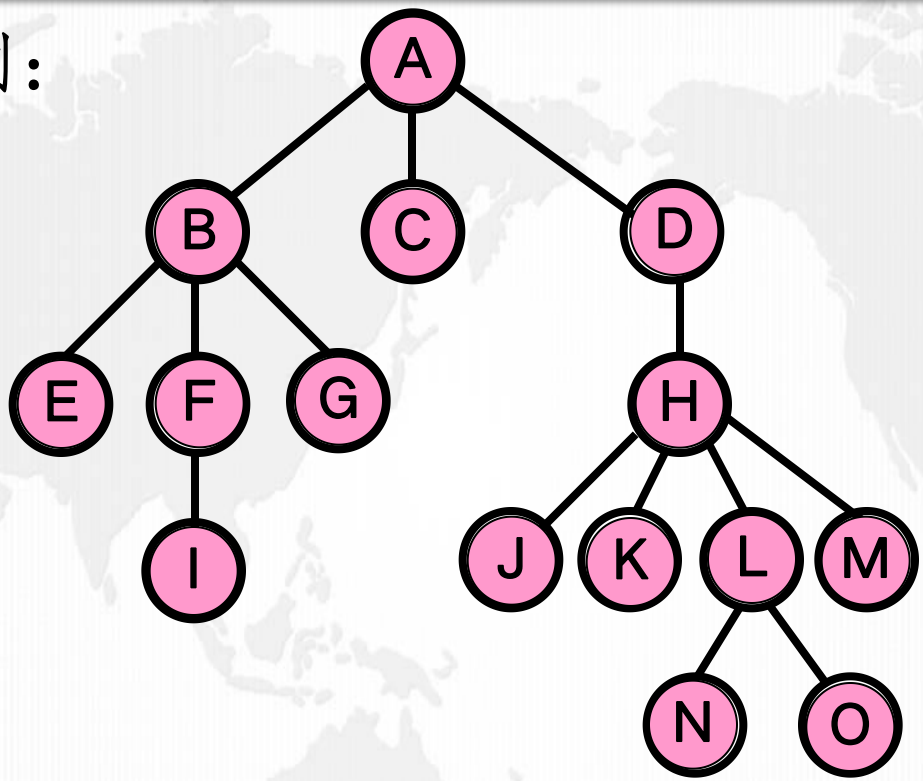
- 树的遍历案例：



先根遍历：ABEFI GCDHJ KL NOM

Tree Iteration

◆ 树的遍历案例：

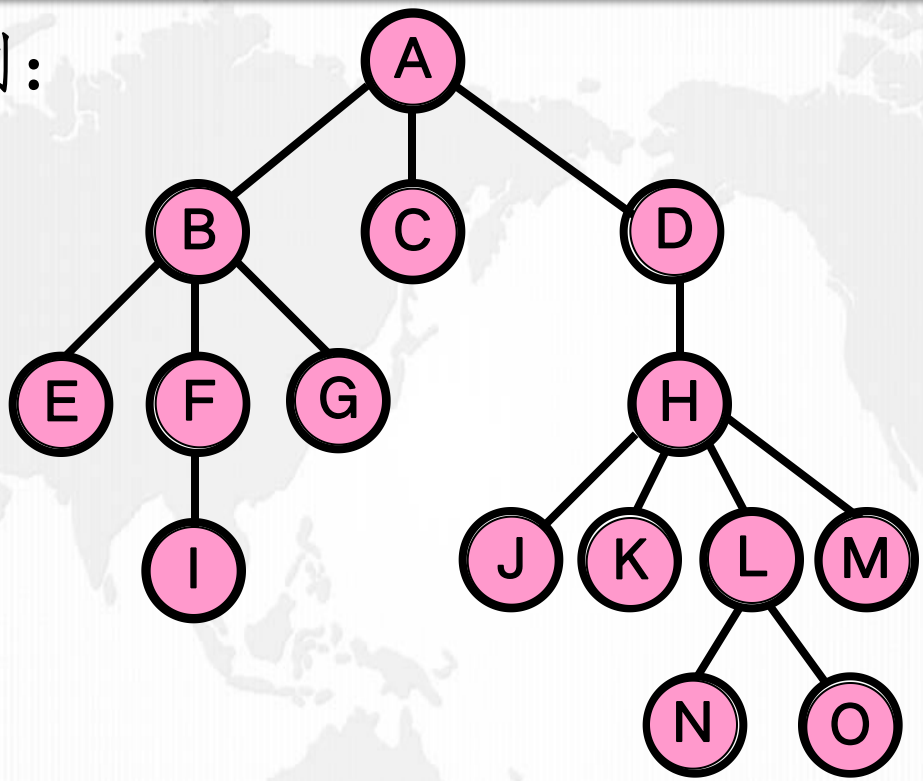


先根遍历：ABEFI GCDHJ KL NOM

后根遍历：EI FGBCJK NOL MHDA

Tree Iteration

◆ 树的遍历案例：



先根遍历：ABEFI GCDHJ KL NOM

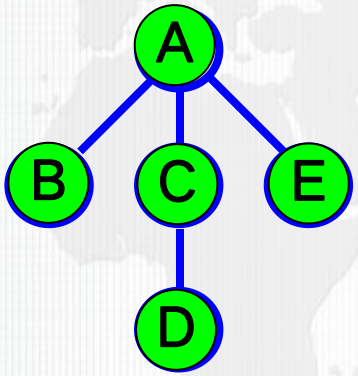
后根遍历：EI FGBCJK NOL MHDA

层次遍历：ABCDE FGHI J KL MNO

Tree Iteration

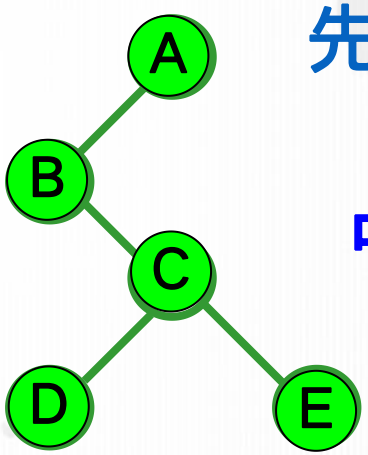
◆ 树的遍历算法——递归算法（采用孩子—兄弟存储结构）：

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

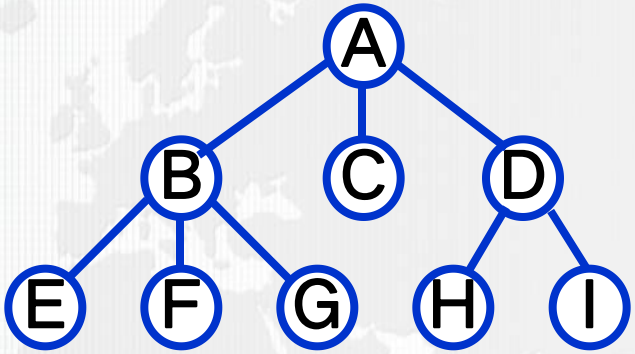
中序：BDCEA

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

Tree Iteration

◆ 树的遍历算法——求树的深度算法（采用孩子—兄弟存储结构）



如何求树的深度？

树的深度=最大子树深度+1

递归

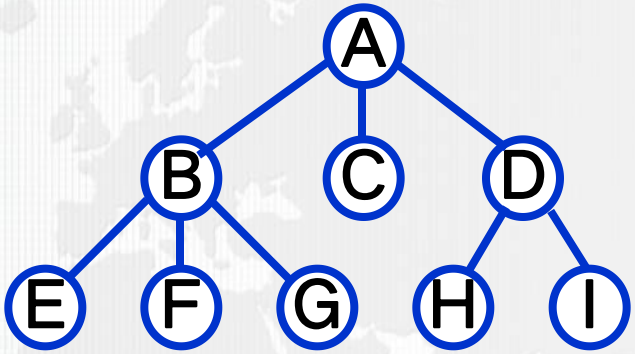
```
typedef struct CSnode
{
  ElemType   data;
  struct CSnode   *firstchild,
                  *nextsibling;
} CSNode, *CSTree;
```

如何选用孩子-兄弟存储结构？

该结构可看作对应二叉树的二叉链表，易于建立并适于递归

Tree Iteration

◆ 树的遍历算法——求树的深度算法（采用孩子—兄弟存储结构）



1.设depth为T的当前最大子树深度，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）

若d>depth，则depth=d

p=p->nextsibling

5.返回depth+1

Forest Iteration

- 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。

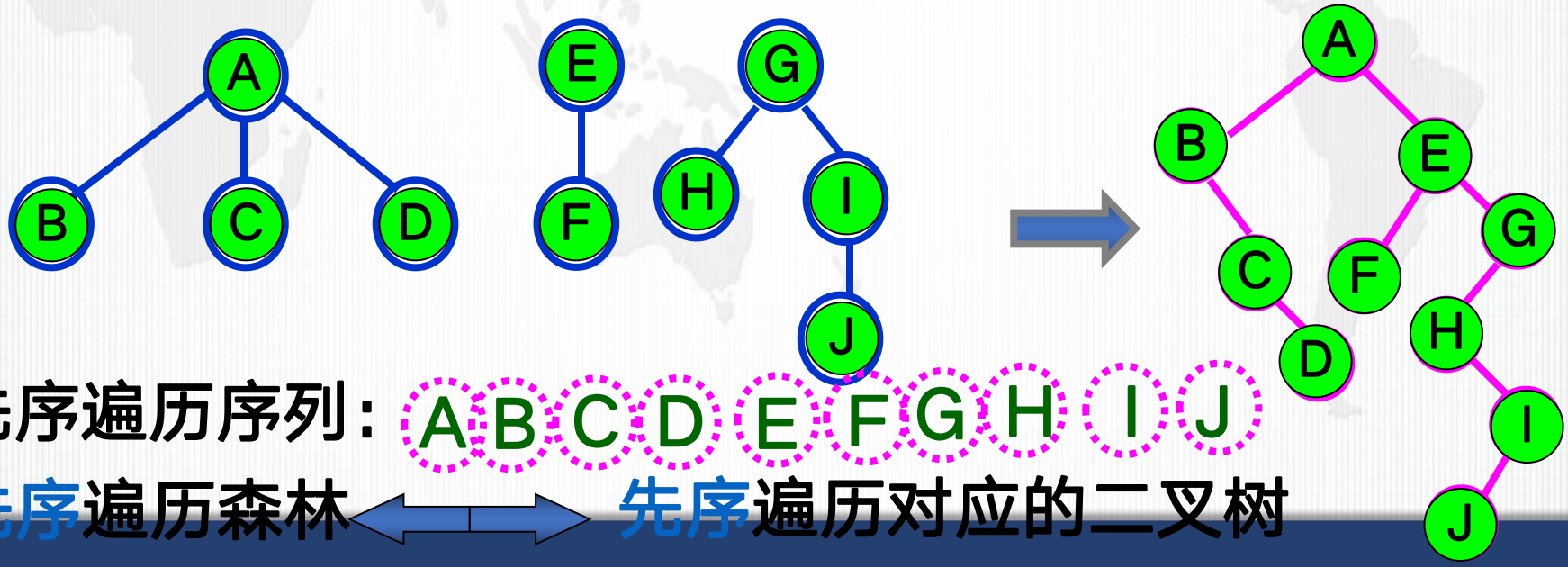
Forest Iteration

◆ 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

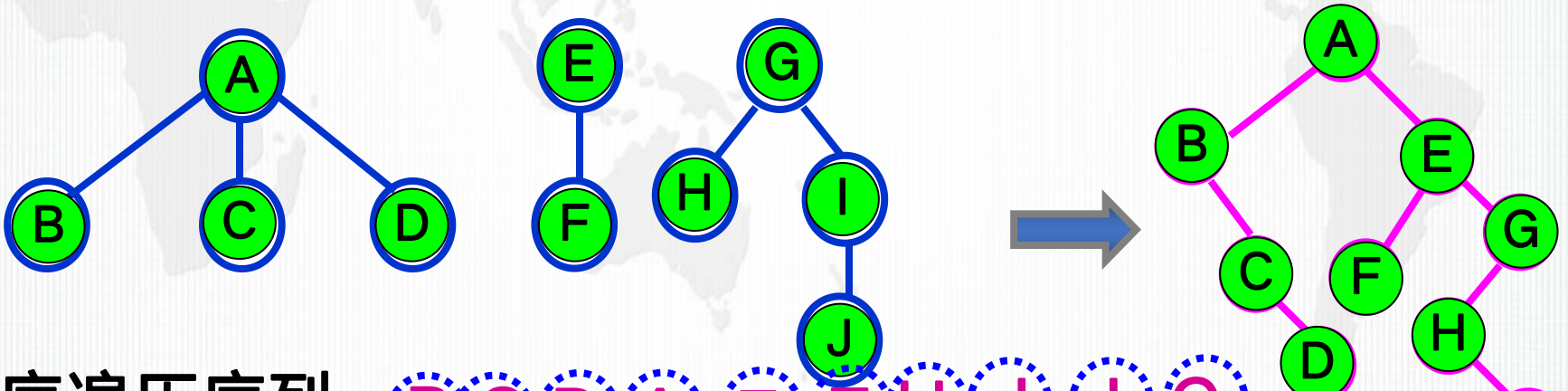
Forest Iteration

◆ 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

Application

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

Application

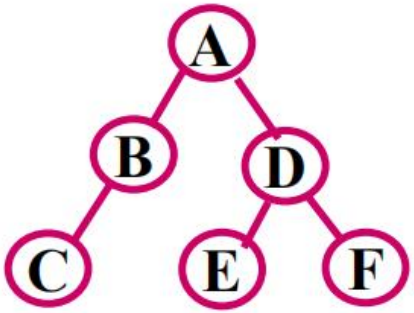
- 由遍历序列确定二叉树：
  - 给定先序、中序遍历序列可唯一确定二叉树。
  - 给定后序、中序遍历序列可唯一确定二叉树。
  - 给定层次、中序遍历序列可唯一确定二叉树。

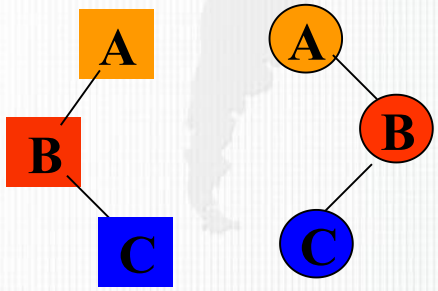
- 由二叉树先序序列和中序序列，构造二叉树

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

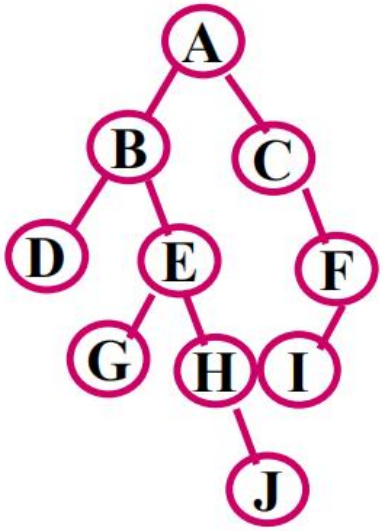
◆ 由遍历序列确定二叉树：

● 由二叉树中序序列和后序序列，构造二叉树

例如，后序序列为：D G J H E B I F C A

中序序列为：D B G E H J A C I F

请构造二叉树



Application

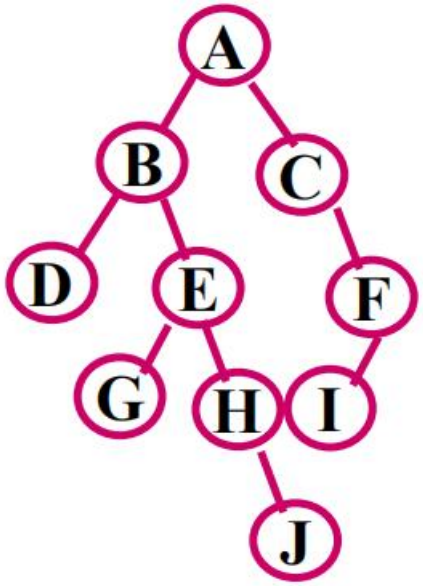
◆ 由遍历序列确定二叉树：

- 由二叉树层次序列和中序序列，构造二叉树

例如，层次序列为： A B C D E F G H I J

中序序列为： D B G E H J A C I F

请构造二叉树



Application

◆ 由遍历序列确定二叉树：

●任何n(n≥0)个不同结点的二叉树,都可由它的中序序列和先序序列唯一地确定。（另外两个定理留于大家自行证明

证明：

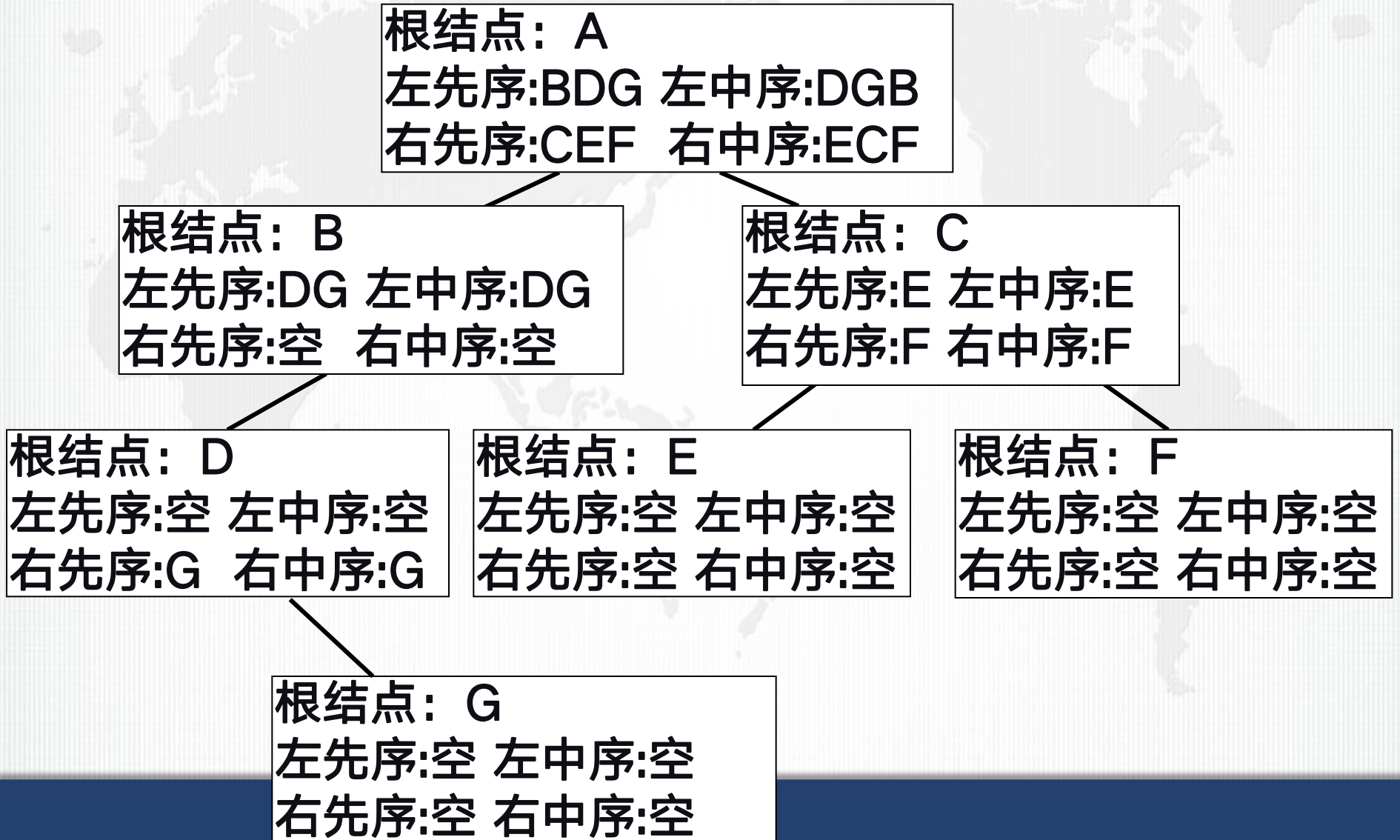
基础情况：对于只有一个节点的二叉树，先序和中序遍历均为该节点，因此可以唯一确定这个二叉树。

归纳步骤：

- 假设对于所有高度小于等于 k 的二叉树，先序和中序遍历可以唯一确定它。
- 考虑一个高度为 k+1 的二叉树。根据先序遍历，我们可以确定其根节点。通过中序遍历，我们可以确定哪些节点属于左子树和右子树。
- 根据归纳假设，左子树和右子树都可以通过它们的先序和中序遍历唯一确定。
- 因此，原始的二叉树也可以通过它的先序和中序遍历唯一确定。
- 这样，我们通过归纳法证明了对于所有的二叉树，先序和中序遍历可以唯一确定它（前提是没有重复的节点值）。

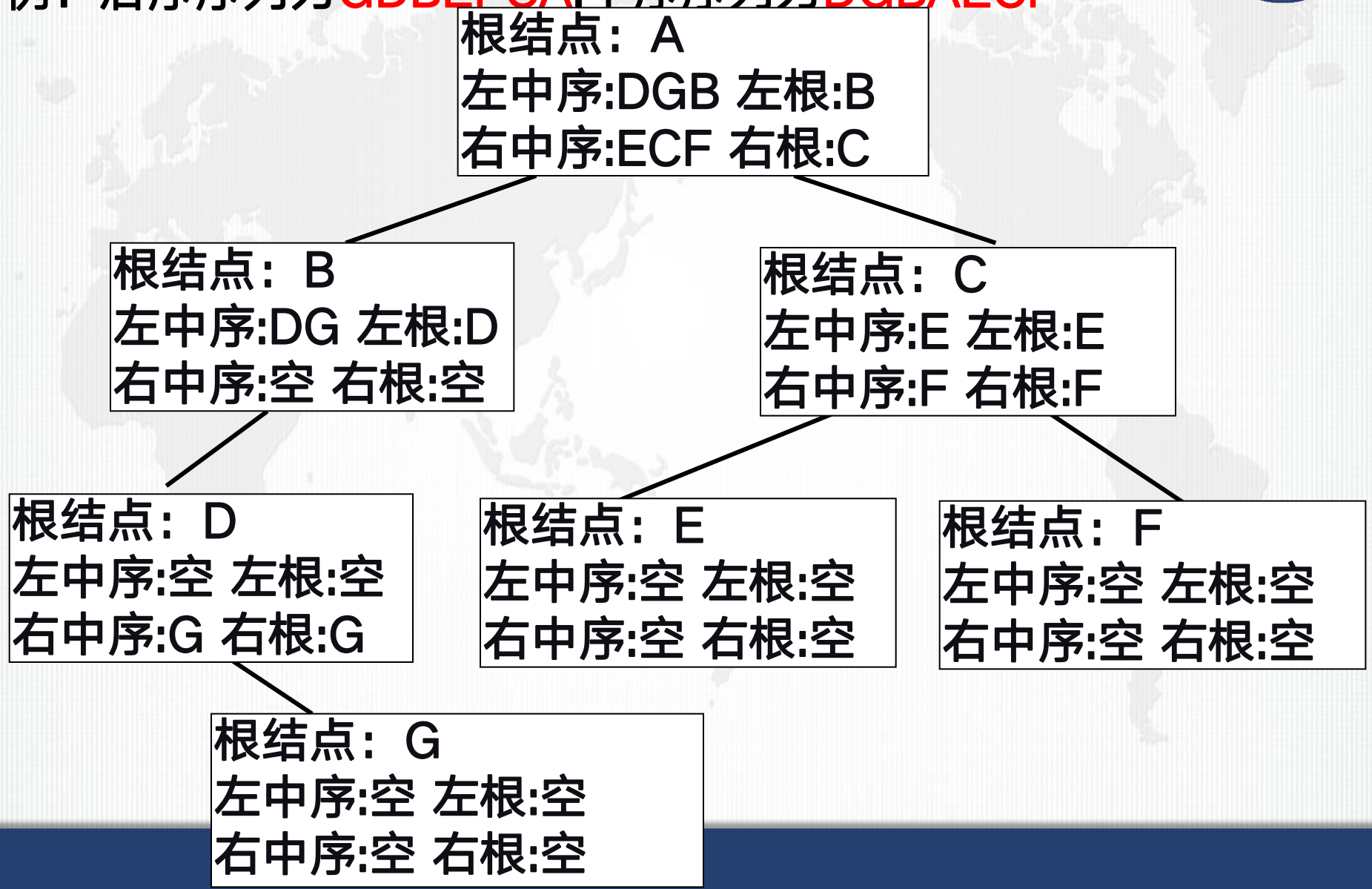
Application

例：已知先序序列为A B D G C E F,中序序列为D G B A E C F



Application

例：后序序列为GDBEFCA,中序序列为DGBAECF



Thanks!

See you in the next session!