数据结构

第八讲 树与树林

孙猛
http://www.math.pku.edu.cn/teachers/sunm

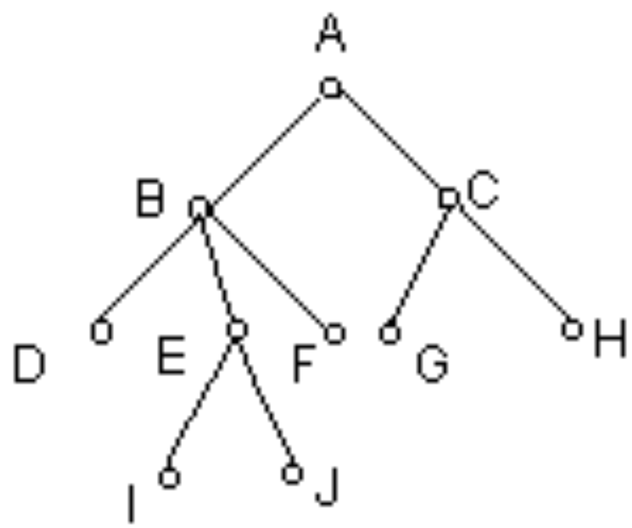
2017年10月26日

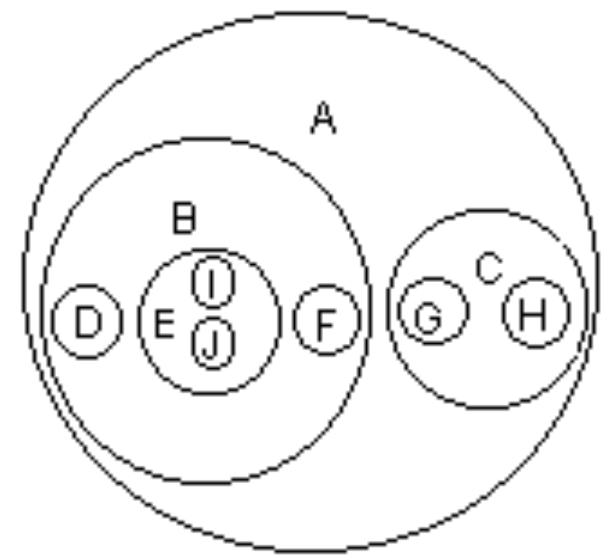
课程内容

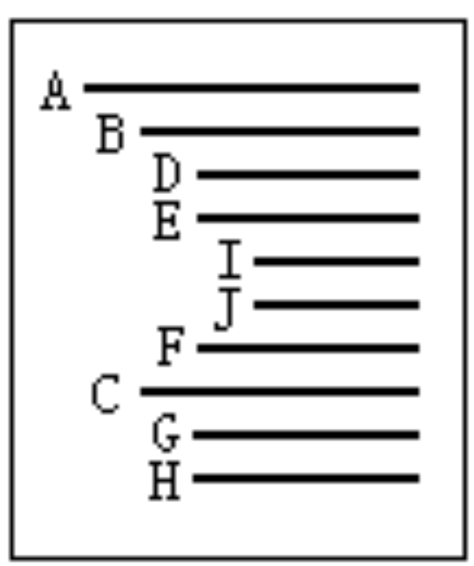
- 树及其抽象数据类型
- 树的实现
- 树林

树及其抽象数据类型

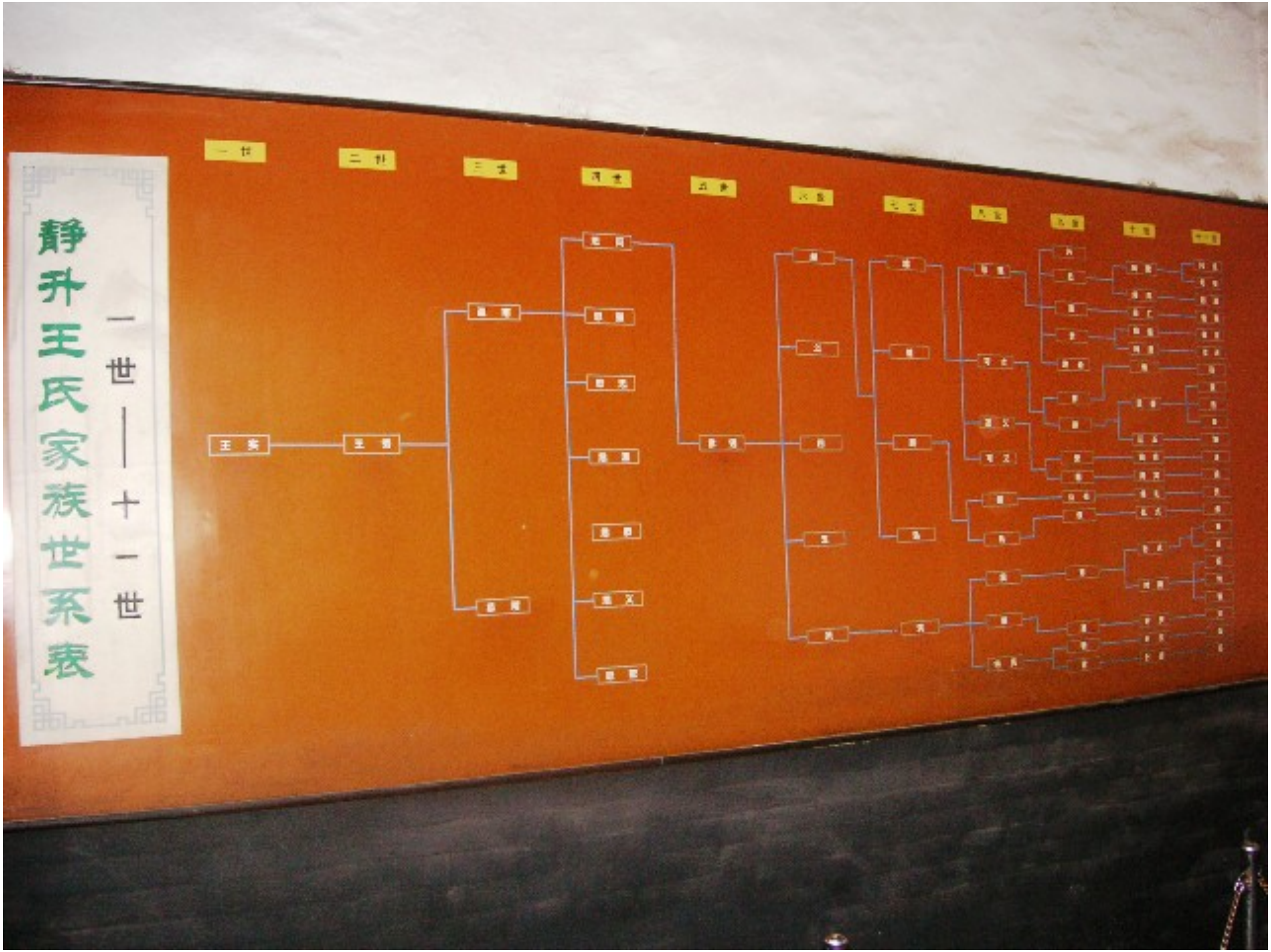
- 树的几种不同表现形式



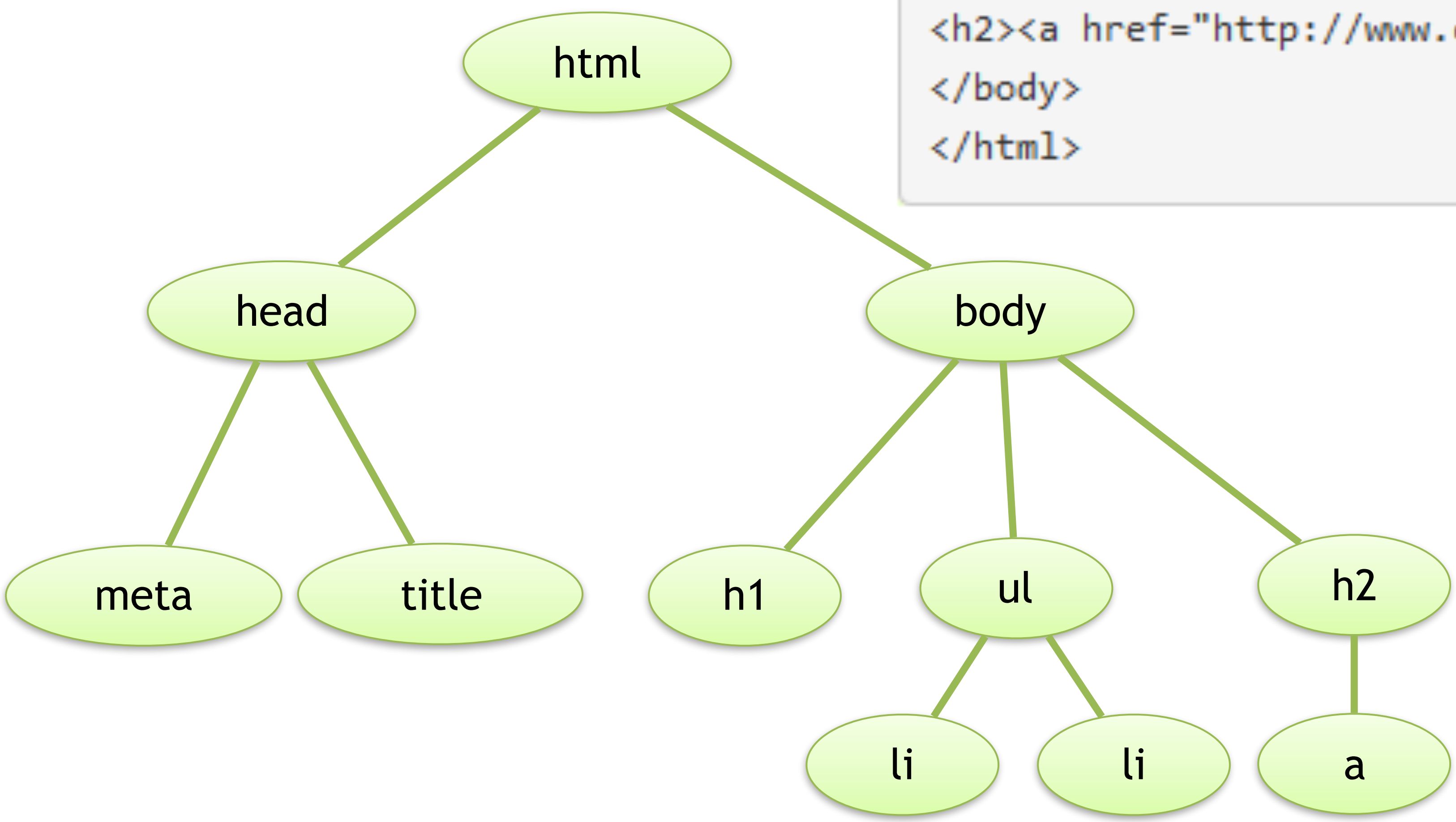




(A(B(D)(E(I)(J))(F))(C(G)(H)))


静升王氏家族世系表
一世——十一世

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```



基本定义

- 树是 $n(n \geq 0)$个结点的有限集T，T非空时满足：
  - 有且仅有一个特殊的称为根(root)的结点r；
  - 除根结点外，其余结点可分为m($m \geq 0$)个互不相交的非空有限集T_1，T_2，…，T_m，其中每个集合又是一棵非空树，称为r的子树(subtree)。
- 关于子树
  - 结点数为0的树称为空树。
  - 一棵非空树可以没有子树($m=0$)，即这棵树只包含一个根结点。
  - 如有子树，则子树非空。

主要概念

- 空树
- 父结点、子结点、路径、路径长度
- 结点的度数
- 树的度数
- 无序树、有序树
- 结点的次序
- 最左子结点、长子、次子
- 右兄弟

抽象数据类型

ADT Tree **is**

operations

```
Tree createEmptyTree(void);
Tree constree(Node p, Tree t1, … Tree ti);
int isNull(Tree t);
Node root(Tree t);
Node parent(Node p);
Tree leftchild(Tree t);
Tree rightSibling(Tree t);
……
```

End ADT Tree

树的周游

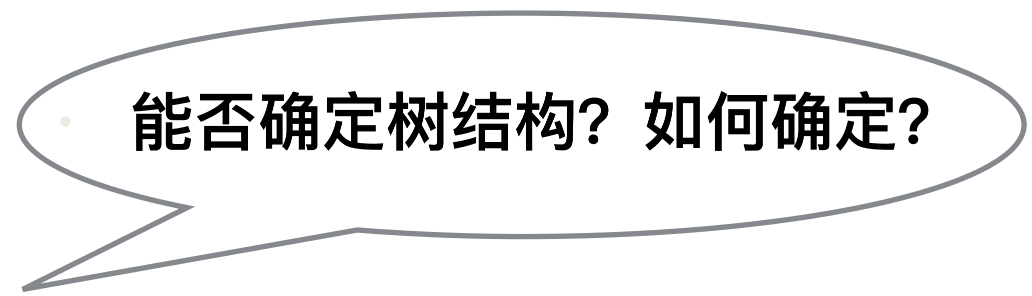
- 深度优先周游：

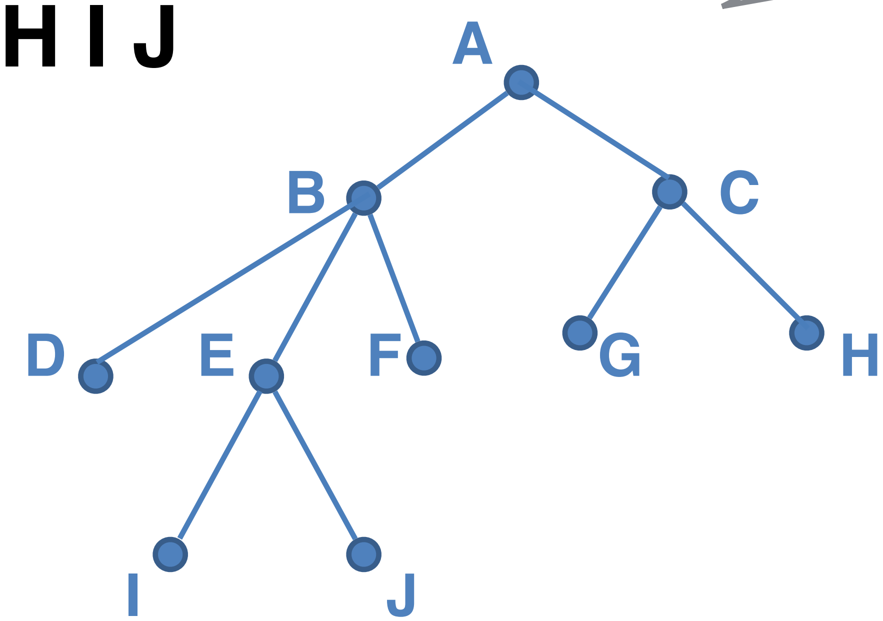
  先根次序 ： A B D E I J F C G H

  后根次序 ： D I J E F B G H C A

- 广度优先周游：

  A B C D E F G H I J





深度优先周游：先根次序

```
/*按先根次序周游树的递归算法*/
void preOrder(Tree  t) {
    Tree  c;
    if (t==NULL) return;
    visit( root(t) );
    c = leftChild ( t );
    while (c!=NULL)
        {preOrder ( c );  c = rightSibling ( c ); }
}
```

按先根次序周游树的非递归算法

```
void nPreOrder ( Tree t ) {
Tree c = t;
Stack  s = createEmptyStack ( );
          /* 栈元素的类型是Tree  */
do{
     while ( c!=NULL ) {
        visit ( root(c) ); push ( s, c ); c = leftChild ( c );
      }
     while ((c==NULL)&&(!isEmptyStack(s)))  {
        c = rightSibling(top(s)); pop(s);
     }
   } while(c!=NULL);
}
```

深度优先周游：后根次序

```
/*按后根次序周游树的递归算法*/
void postOrder( Tree t ) {
Tree  c;
if (t==NULL) return;
c = leftChild ( t );
while (c!=NULL)
   { postOrder ( c );c = rightSibling ( c ); }
visit(root(t) );
}
```

广度优先周游

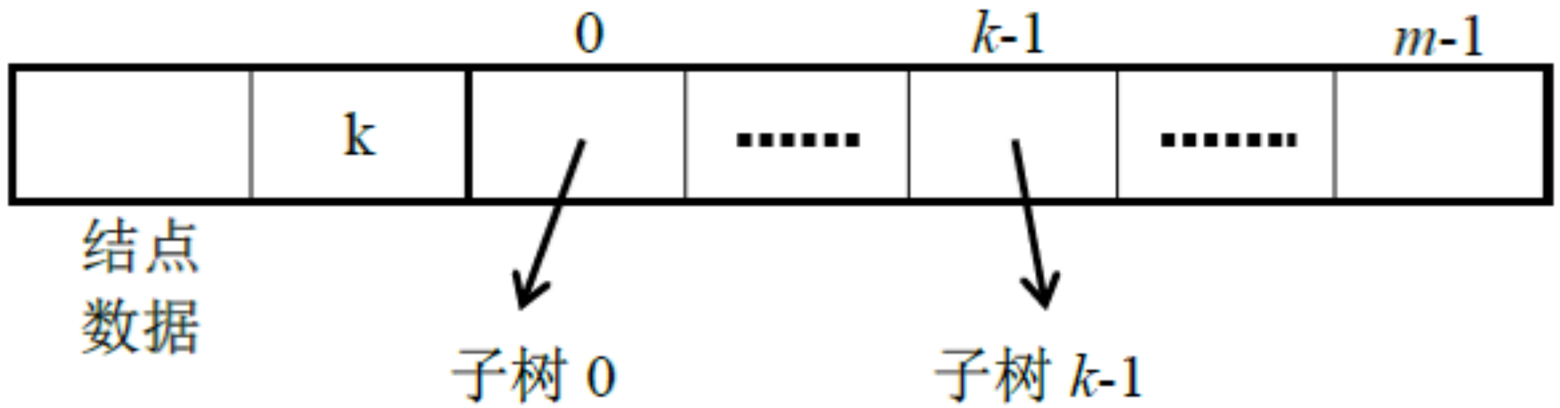
```
void levelOrder(Tree  t) {
  Tree  c;
  Queue q;                                /* 队列元素的类型为Tree */
  q = createEmptyQueue();  c =t;
  if (c==NULL)  return;
  enQueue(q,c);
  while (!isEmptyQueue(q)) {
     c = frontQueue(q);  deQueue(q);
     visit( root(c) );  c = leftChild( c );
     while (c!=NULL){
          enQueue(q,c);  c = rightSibling( c );
      }
   }
}
```

树的实现

- 子结点表示法
- 父指针表示法
- 子表表示法
- 长子-兄弟表示法

子结点表示法

- 树的最基本表示方法是子结点表示法，其基本设计与二叉树的左右指针表示法类似：用一个数据单元表示结点，通过结点间的指针表示树结构。



- 在m度结点表示的n个结点的树中，恰有$n\times(m-1)+1$个空树引用域。

父指针表示法

- 树中除根以外的每个结点都有唯一的一个父结点。
- 根据这一特性，可用一组连续的存储空间，即用一个顺序表存储树中的各个结点。
- 表中的每个元素包括结点本身的信息以及本结点的父结点在表中的下标。

父指针表示法

结点的结构可定义为：

```
struct  ParTreeNode{
    DataType  info;                              /* 结点中的元素 */
    int parent;                                  /* 结点的父结点位置 */
};
```

树的类型定义为：

```
struct  ParTree{
    int MAXNUM;                                  /* 树中最大结点个数 */
    int n;                                       /* 树中已有结点的个数 */
    struct ParTreeNode * nodelist;               /* 存放树中结点的数组 */
};
typedef  struct  ParTree  * PParTree;    / 树类型的指针类型 */
```

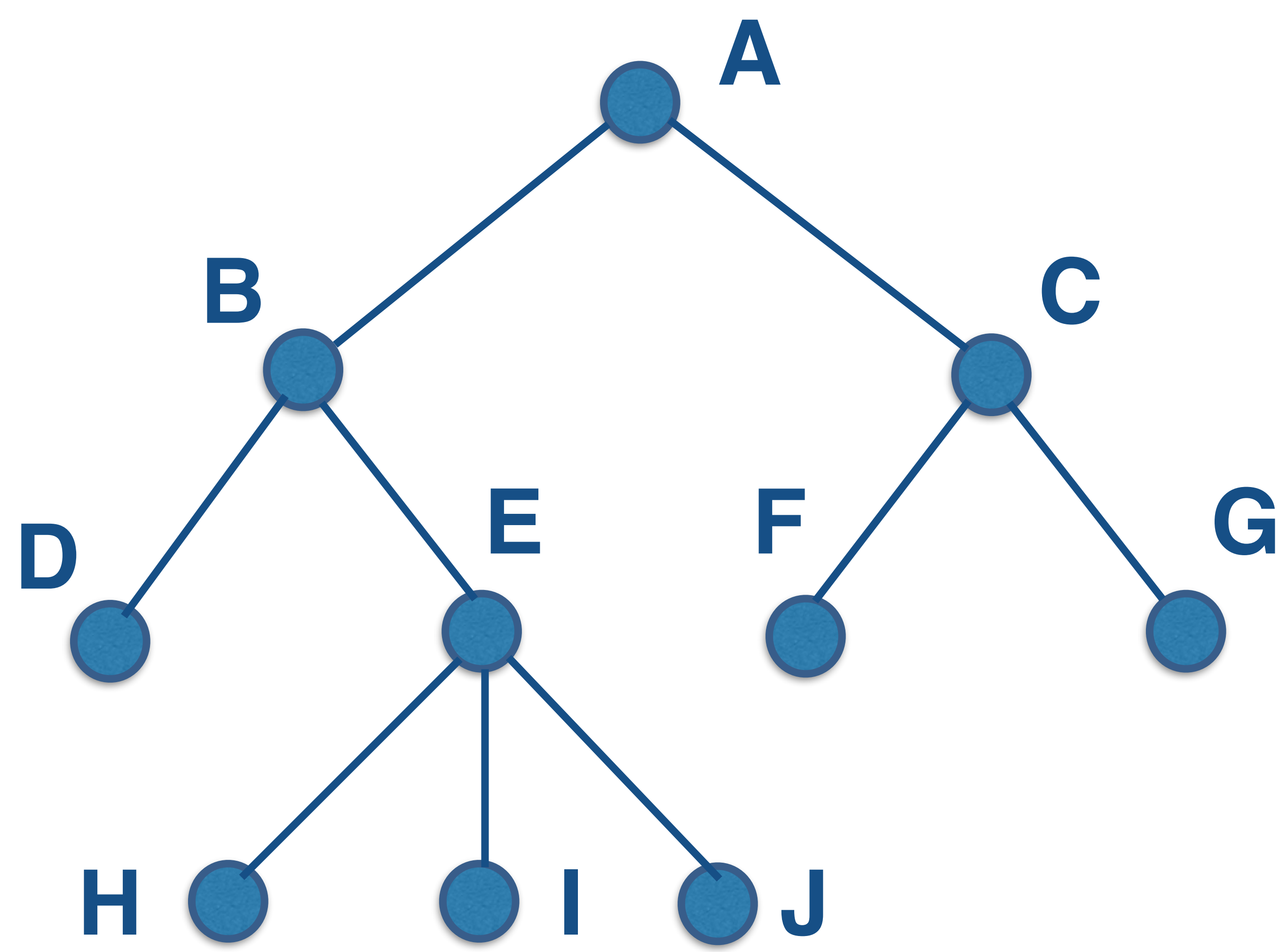
父指针表示法的改进

问题：

1）求结点的子结点和兄弟就需要查询整个表。

2）没有表示出结点之间的左右次序，无法求树中某个指定结点的最左子结点和右兄弟结点等基本运算。

改进方法是按一种周游次序在表中存放结点，其中较常见的一种方法是依次存放树的先根序列。



	info	parent
0	A	-1
1	B	0
2	D	1
3	E	1
4	H	3
5	I	3
6	J	3
7	C	0
8	F	7
9	G	7

父指针表示法的算法

```
/*在父指针表示的树中求右兄弟结点的位置*/
int rightSibling_partree(PParTree t, int p) {
  int i;
  if (p>=0 && p<t->n)
    for (i=p+1;i<t->n;i++)
      if (t->nodelist[i].parent==t->nodelist[p].parent)  return i ;
  return -1;
}

/*在父指针表示的树中求最左子结点的位置*/
int leftChild_partree(PParTree t, int p) {
  if (t->nodelist[p+1].parent==p)  return(p+1);
  else  return -1 ;
}
```

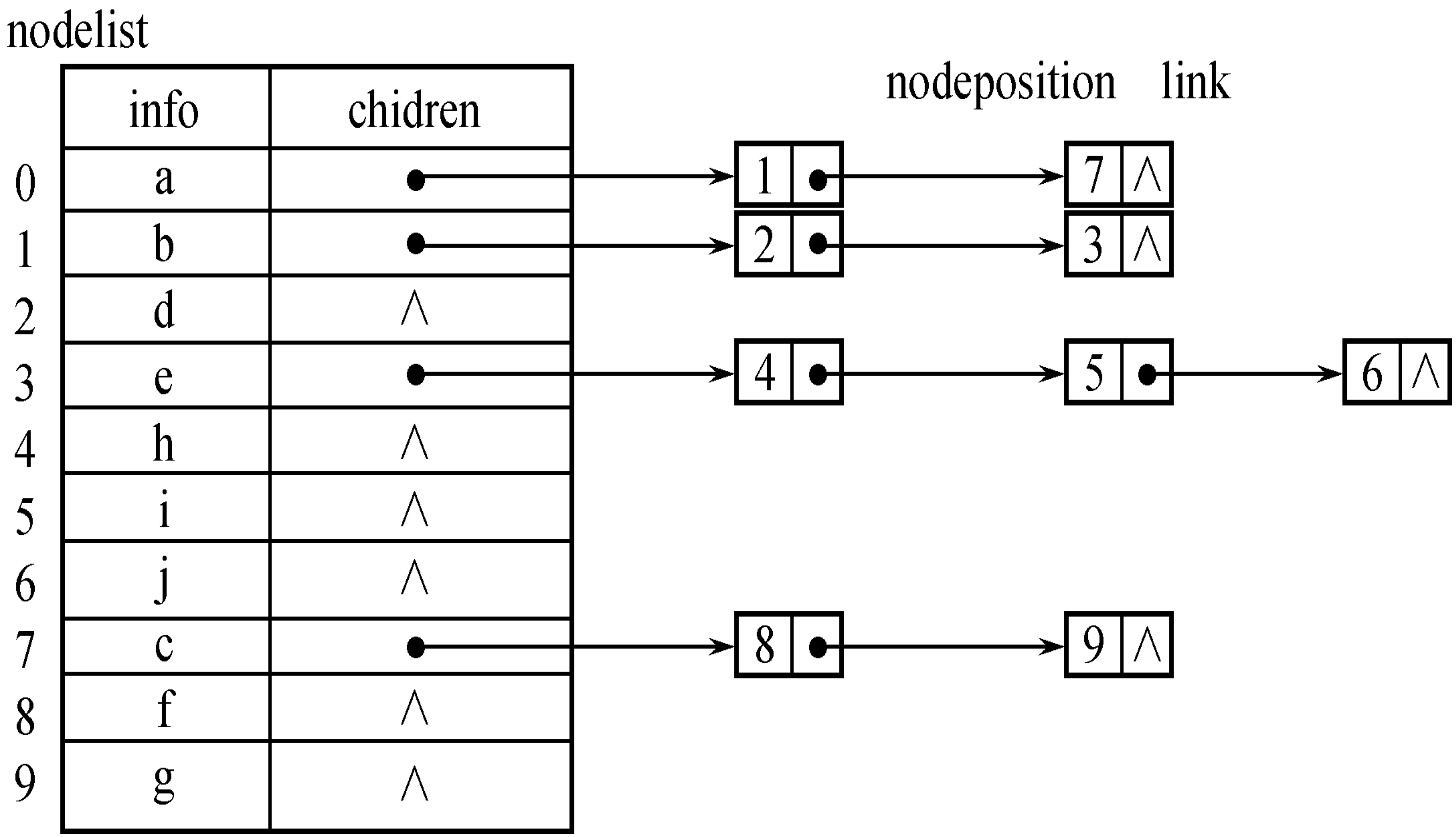
讨论

- 父指针表示方法的主要优点是存储空间比较节省，对求某个结点的父母和求某个结点的最左子结点操作都很方便，但是对求某个结点的右兄弟运算比较慢。

子表表示法

- 把整棵树表示成一个结点表。
- 结点表中的每个元素又包含一个表，它记录了这个结点的所有子结点的位置，称为子表（或者称为边表）。
- 结点表的长度即树中结点的个数，一般用一维数组顺序存储；结点表中除了要保存元素本身的信息外，还要保存子表的表头链接。
- 而子表的长度依赖于各结点的度数，所以各不相同，一般用单链表表示；子表中结点的链接顺序是按其在树中从左到右的次序进行的。

前面所示的树的子表表示



结点结构定义

```
struct  EdgeNode {           /* 子表中结点的结构 */
  int nodeposition;          /* 子结点在结点表中的位置 */
  struct EdgeNode  *link;   /* 指向下一个子表中的结点 */
};

struct  ChiTreeNode  {             /* 结点表中结点的结构 */
  DataType info;                   /* 结点本身的信息 */
  struct EdgeNode  *children;  /* 本结点子表的头指针 */
};
```

树结构定义

```
struct  ChiTree  {                              /* 树结构 */
    int  MAXNUM                                 /* 树中最大结点个数 */
    int  root;                                  /* 根结点的下标 */
    int  n;                                     /* 实际结点个数 */
   struct ChiTreeNode  * nodelist;  /*结点表 */
 };

typedef  struct  ChiTree *  PChiTree;
```

在树的子表表示中实现操作

- 与父指针表示法类似，通常树结点表中的结点也按某种遍历序排列。由于还有子结点表，通过它们可以直接找到相关子结点，各种找子结点的操作（求最左子结点、找全部子结点等）实现都比较方便。
- 但求某个结点的父结点和右兄弟实现起来比较费事，因为要找某结点的父结点必须依次检查哪个结点的子表中是否包含该结点，而要找某结点的右兄弟时，则首先要找到其父结点，然后再从父结点的子表中找寻它的右兄弟结点。

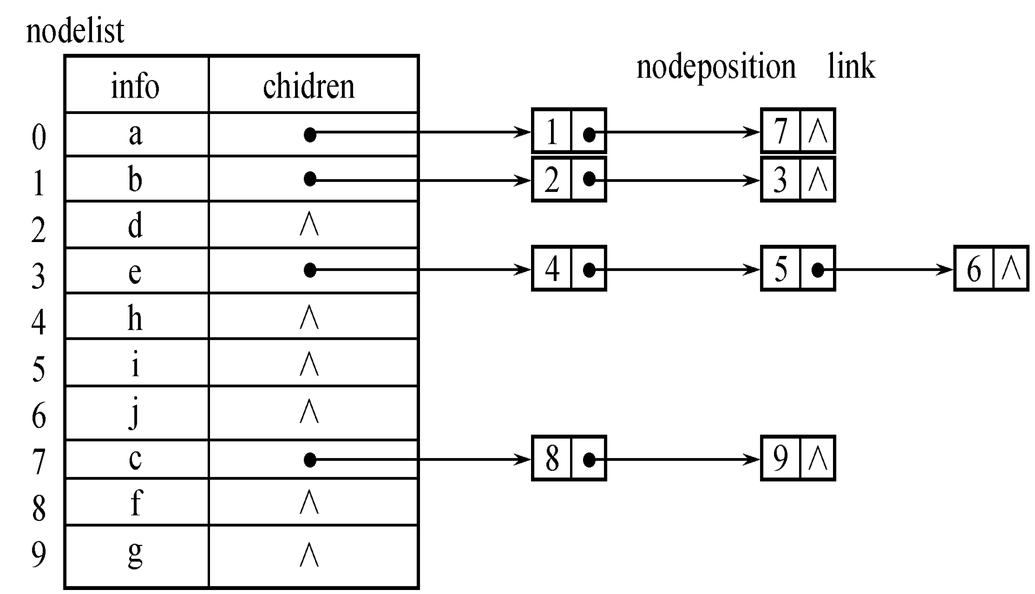
在树的子表表示上求父结点位置

```
int parent_chitree(PChiTree t, int p) {
  int i;  struct EdgeNode  *v;
  for (i=0;i<t->n;i++) {
     /* 逐个检查树的各个结点，是不是父结点 */
    v=t->nodelist[i].children;
    /* 若检查的结点子表中有p，则返回值是该结点的位置 */
    while (v!=NULL)
     if (v->nodeposition==p)
        return i ;
     else  v=v->link;
  }
  return(-1);  /* 无父结点，则返回值为-1 */
}
```

在树的子表表示上

求右兄弟结点位置

```
int rightSibling_chitree(PChiTree t, int p) {
int i;  struct EdgeNode *v;
for (i=0;i<t->n;i++){
   v = t->nodelist[i].children;
   while (v!=NULL)
    if (v->nodeposition==p)
      if (v->link==NULL)return(-1);
      else  return(v->link->nodeposition);
    else  v=v->link;
  }
return(-1);
}
```

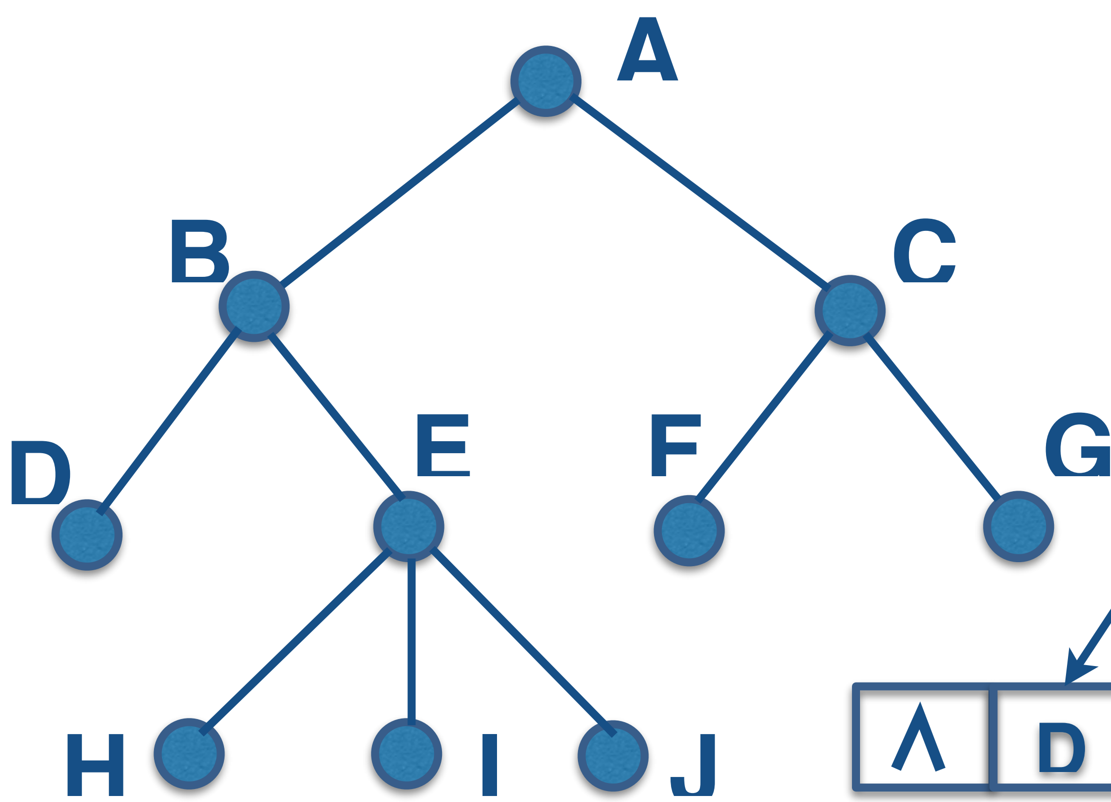


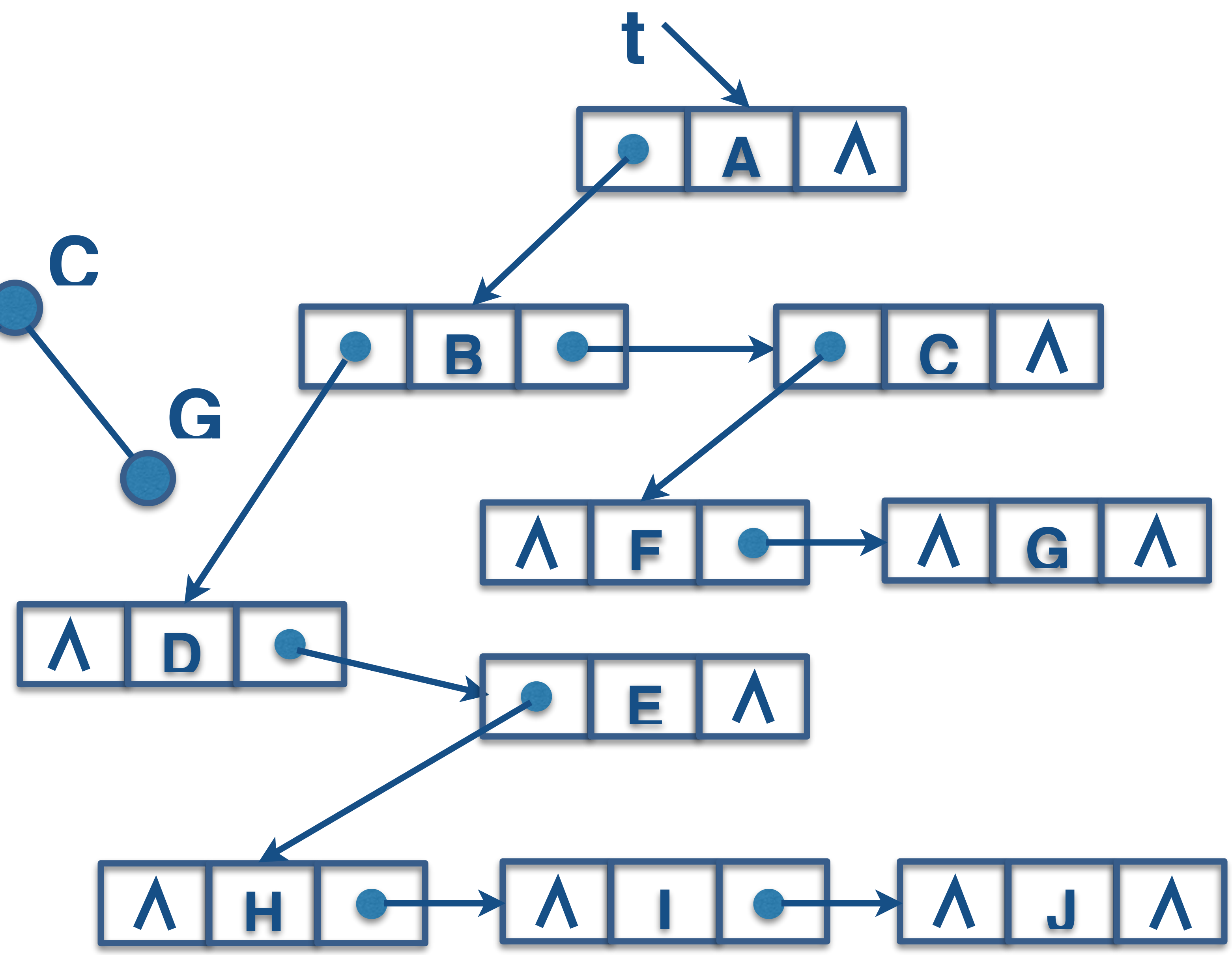
长子－兄弟表示法

在这种存储结构中，类型定义如下：

```
struct  CSNode;                          /* 树中结点结构 */
typedef  struct  CSNode  * PCSNode;  /* 结点的指针类型 */
struct  CSNode  {                        /* 结点结构定义 */
   DataType   info;                      /* 结点中的元素 */
   PCSNode  lchild;          /* 结点的最左子结点的指针 */
   PCSNode  rsibling;                    /* 结点的右兄弟的指针 */
 };
typedef  struct CSNode * CSTree;      /* 树类型定义 */
```

长子－兄弟表示法





长子－兄弟表示法

- 采用长子-兄弟表示法表示树时，除找父结点和从子树构造树之外，其余的基本运算实现起来都是比较简单的。甚至可以用一个简单的表达式或赋值语句代替函数调用。
- 找结点的全部子结点也很容易：先找到长子，再由子结点的rsibling字段逐个地找子结点的右兄弟。
- 长子-兄弟表示法的缺点是寻找某个指定结点的父结点比较麻烦，需要对树进行周游，周游时检查被访问的结点的各个子结点的位置是不是指定结点。

树林

- 树林是由零个或多个不相交的树所组成的集合。
- 树林中所有树也是有序的，彼此称为兄弟。
- 与自然界的树林有所不同，这里的树林可以是一个空集，也可以只由一棵树构成。
- 如果从一棵树中将根结点（连同根结点到各子结点的边）删除，便得到一个树林。

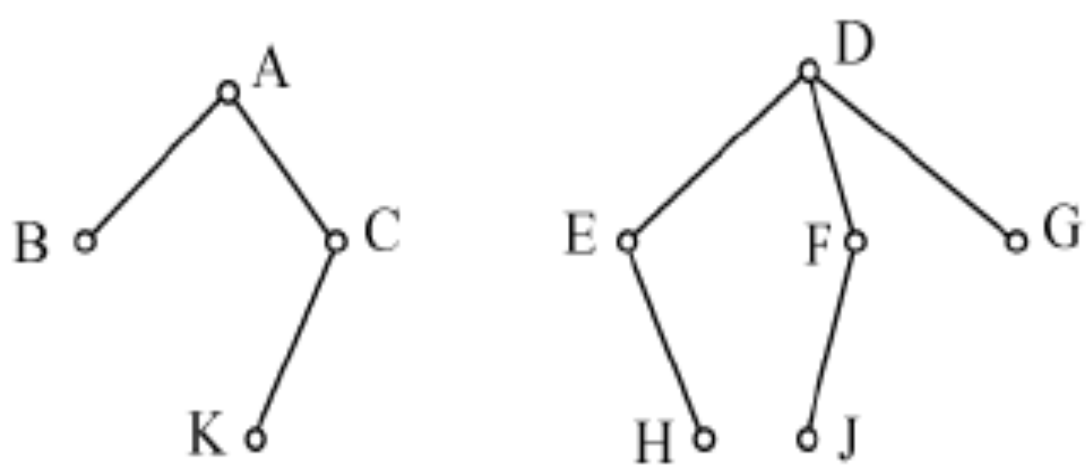
树林的周游

- 先根次序周游：
  - 首先访问树林中第一棵树的根结点；
  - 然后先根次序周游第一棵树除去根结点剩下的所有子树构成的树林；
  - 最后先根次序周游除去第一棵树之后剩下的树林。
- 后根次序周游：
  - 首先后根次序周游第一棵树的根结点的所有子树构成的树林；
  - 然后访问树林中第一棵树的根结点；
  - 最后后根次序周游除去第一棵树之后剩下的树林。

树林的存储表示

- 前面所有树的表示方法都可以推广到树林。下面给出用这些方法表示一个树林的例子。

树林的父指针表示

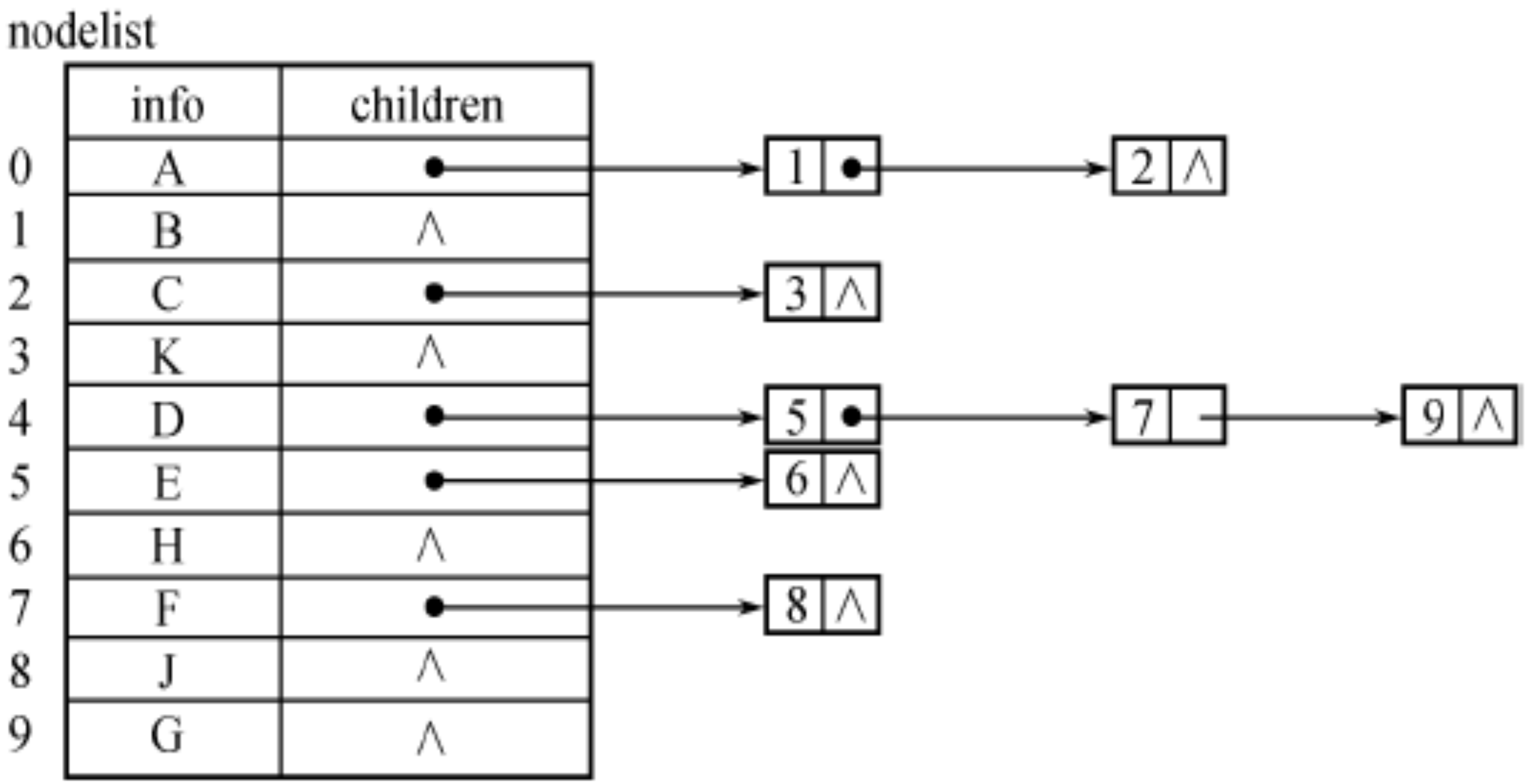


(a) 一个由两棵树构成的树林

	info	parent
0	A	−1
1	B	0
2	C	0
3	K	2
4	D	−1
5	E	4
6	H	5
7	F	4
8	J	7
9	G	7

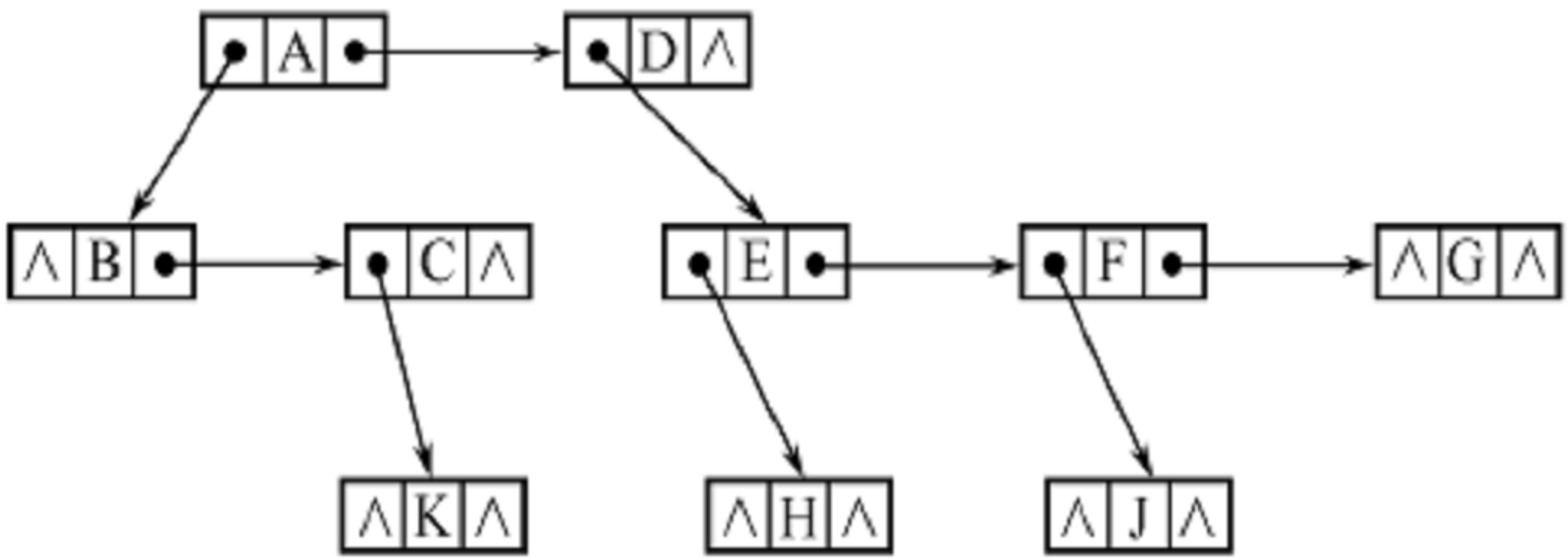
(b) 父指针表示法

树林的子表表示



(c) 子表表示法

树林的长子－兄弟表示



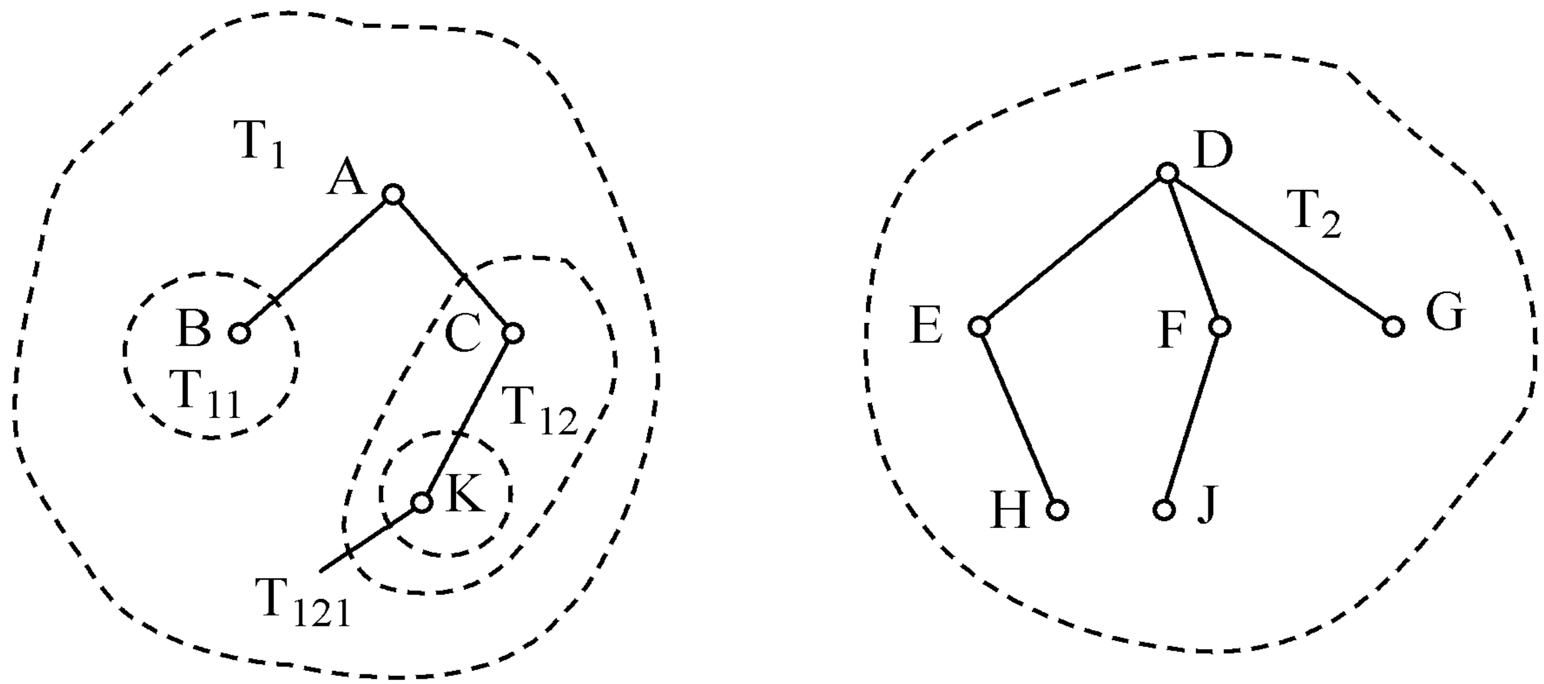
(d) 长子兄弟表示法

树林与二叉树的转换

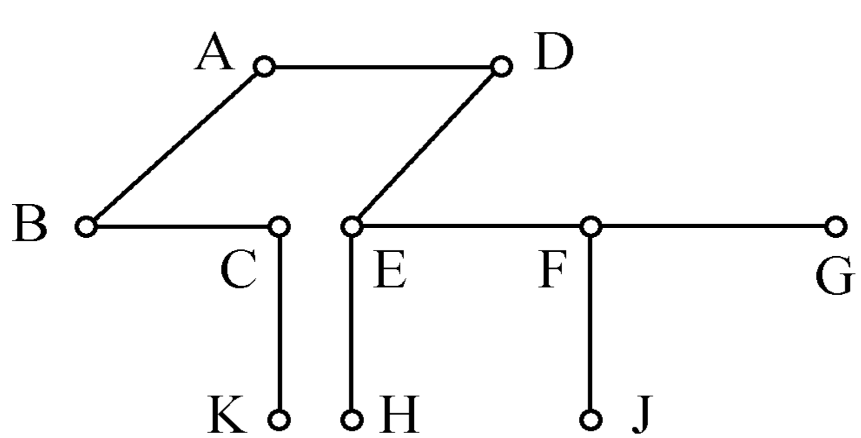
在树林（包括树）与二叉树之间有一个自然的一一对应的关系。

任何树林都唯一地对应到一棵二叉树；

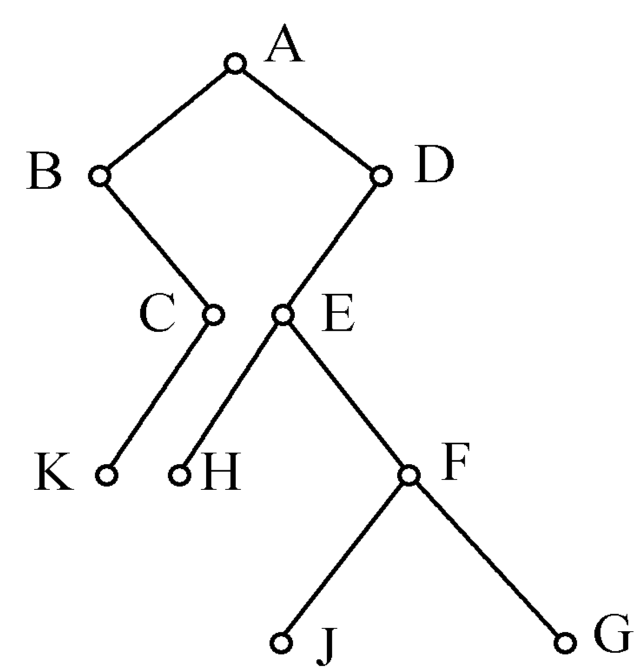
任何二叉树也都唯一地对应到一个树林。



(a) 转换前的树林



(b) 在兄弟结点间加一连线并去掉与
最左子结点外的其他原来连线



(c) 顺时针旋转45°

树林转换为二叉树

- 首先在所有相邻的兄弟结点之间加一条线；
- 然后 对每个非终端结点，只保留它到其最左子结点的连线，删去它与其它子结点之间原有的连线；
- 最后以根结点为轴心，将整棵树顺时针旋转一定角度（如45度），使其层次分明。

树林转换为二叉树

把树林F看作树的序列：$F = T_1$，T_2，…，T_n，

对构造应于F的二叉树B(F)的定义可以递归描述如下：

若$n = 0$，则B(F)为空；

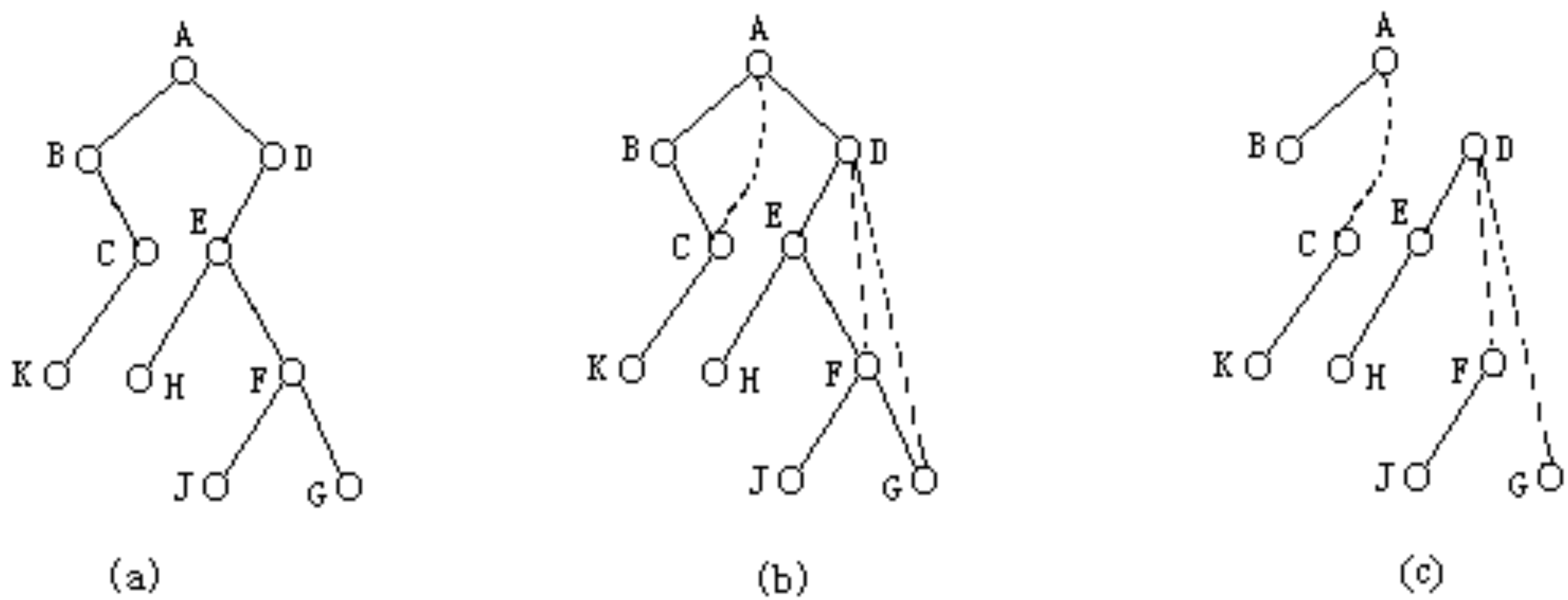
若$n > 0$，则B(F)的根是T_1的根w_1，

B(F)的左子树是B(T_{11}，T_{12}，…，T_{1m})，

其中T_{11}，T_{12}，…，T_{1m}是T_1的子树；

B(F)的右子树是B(T_2，…，T_n)。

二叉树转换为树林



二叉树转换为树林

(1)若某结点是其父母的左子结点，则把该结点的右子结点，右子结点的右子结点……，都与该结点的父母用（虚）线连起来；

(2)去掉原二叉树中所有父母到右子结点的连线；

整理由(1)、(2)两步所得到的树或树林，使之结构层次分明。

二叉树转换为树林

设B是一棵二叉树，r是B的根，L是r的左子树，R是r的右子树，则构造对应于B的树林F(B)　可作如下严格的递归描述：

若B为空二叉树，则F(B)是空的树林；

若B不为空二叉树，

则F(B)是一棵树T1加上树林F(R)，

其中树T1的根为r，r的子树为F(L)。

本讲重点

- 树是一种常见的树形结构，它常用的存储表示有四种：子结点表示法、父结点表示法、子表表示法和长子－兄弟表示法。周游是树上的重要操作，主要有深度优先和广度优先两种方式，在深度优先中又分为先根次序周游和后根次序周游。
- 由于树/树林与二叉树之间存在对应关系，所以常常把树与树林转换为二叉树处理。这使得二叉树的讨论和它的llink-rlink表示更加重要。
- 二叉树和树的各种存储方式都应该包（隐）含所有的结点和结点之间关系的信息，不同的表示原则上应该可以相互转换。