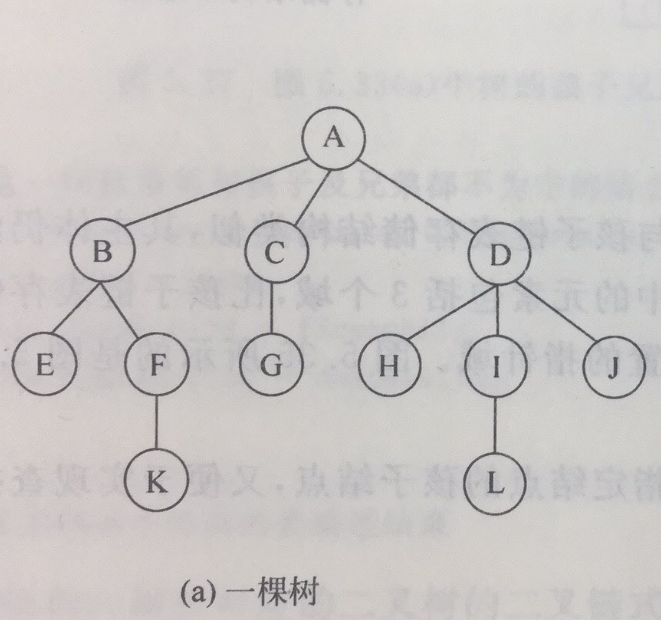
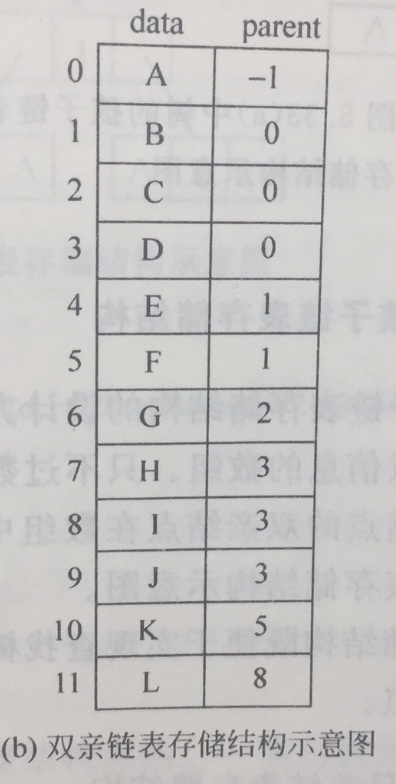
树的存储结构

（1）双亲链表存储结构

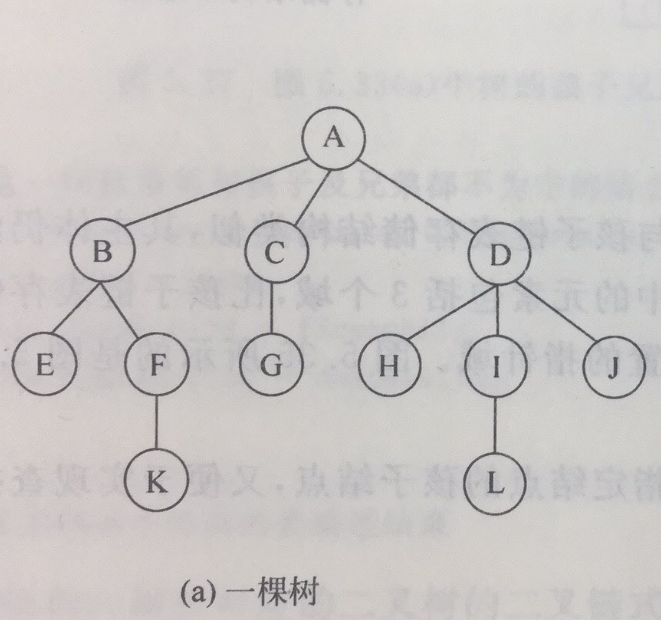
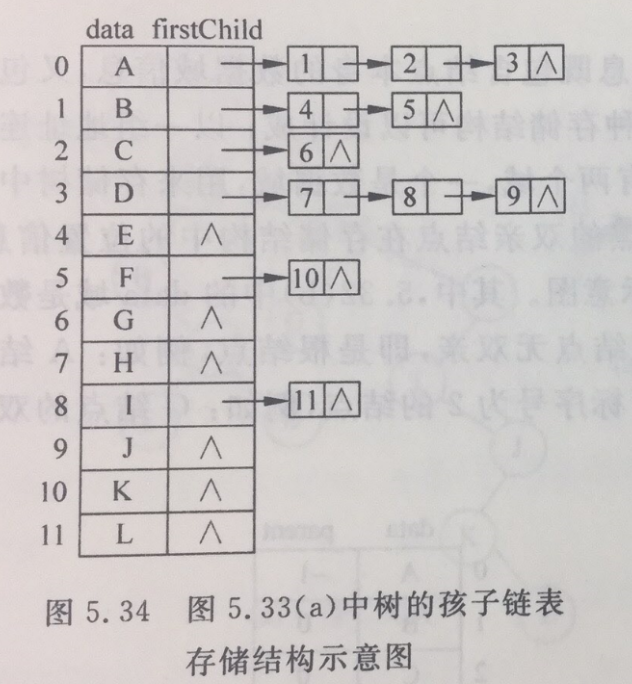
内部是一组连续的存储单元来存放树的各个节点，每个节点有两个域：1.数据域2.双亲节点的指针域。

（2）孩子链表存储结构

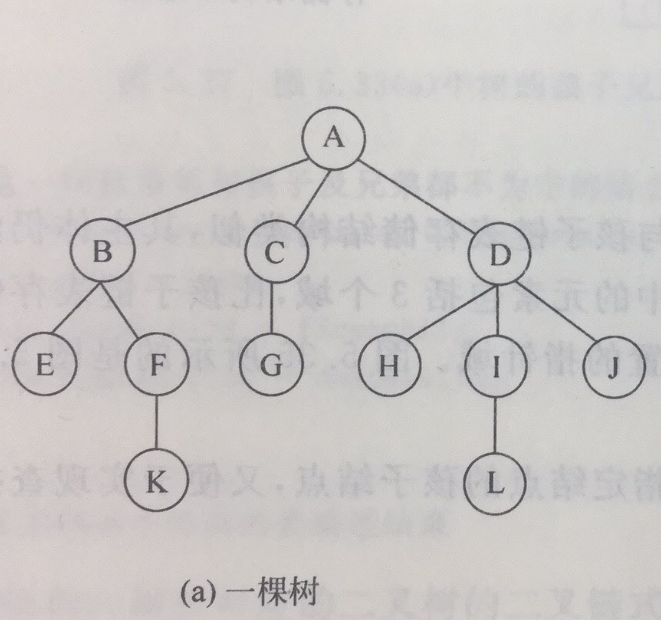
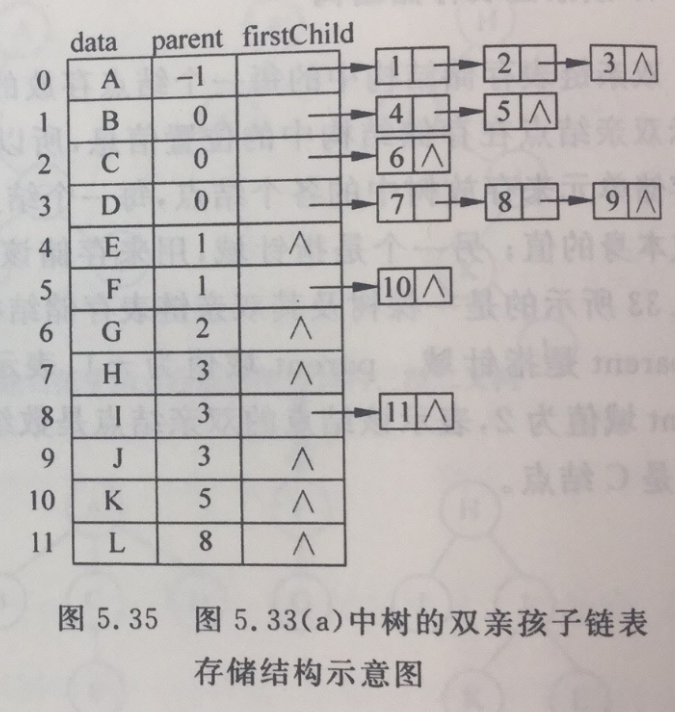
内部的结构：1.数据域 2.所有孩子节点的指针域

每一个节点的子节点数不同，则可将一个节点的所有孩子的位置信息从左到右的顺序连成一个单链表。

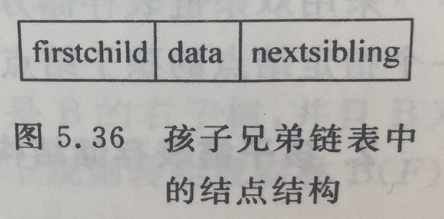
（3）双亲孩子链表存储结构

内部结构：1.数据域 2.双亲节点的指针域 3.所有孩子节点的指针域

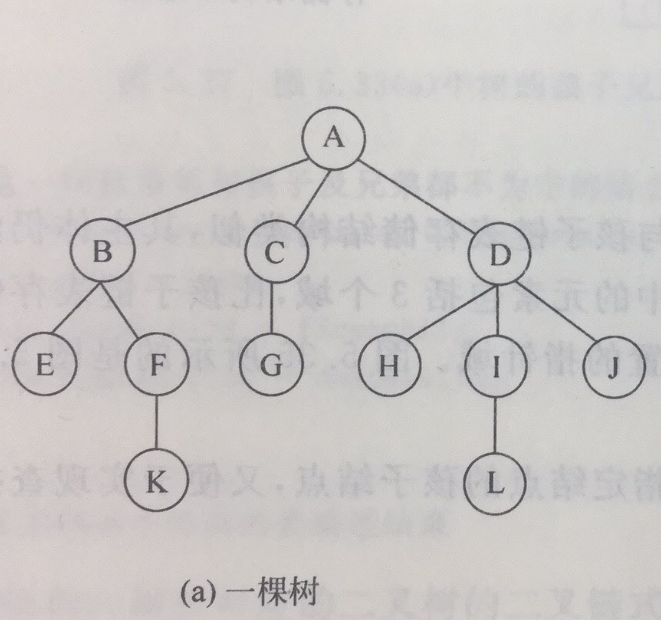
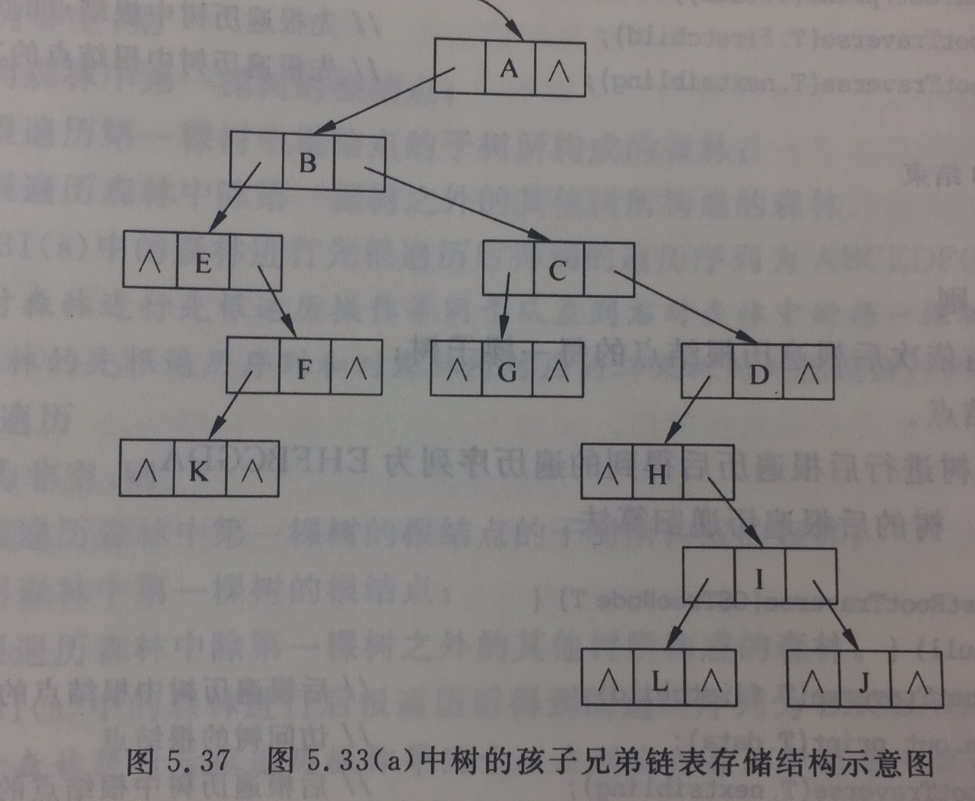
 

（4）孩子兄弟链表存储结构

内部结构：1.数据域 2.第一个孩子对的指针域 3.右邻兄弟的指针域



类似于二叉树的内部数据结构，只不过把二叉树的左孩子指针改为第一个孩子的指针，右孩子指针改为兄弟指针。一切对树的操作都能转化为二叉树的操作，锁着这种存储方式应用最为广泛。

结构代码：

public class CSTreeNode{

private Object date; //数据域

private CSTreeNode firstchild;//第一个孩子的指针域

private CSTreeNode nextsibling;//右邻兄弟的指针域

public CSTreeNode(){

date=null;

firstchild=nextsibling=null;

}

public CSTreeNode(Object date){

this.date=date;

firstchild=nextsibling=null;

}

}

遍历：

（1）先根递归遍历

public void preRootTraverse(CSTreeNode T){

if(T!=null){

System.out.print(T.date);//先访问树的根节点

preRootTraverse(T.firstchild);//先根遍历树中根节点的第一颗子树

preRootTraverse(T.nextsibling);//先根遍历树中根节点的其他子树

}

}

（2）后根遍历

public void postRootTraverse(CSTreeNode T){

if(T!=null){

postRootTraverse(T.firstchild);//后根遍历树中根节点的第一颗子树

System.out.print(T.date);//访问树的根节点

postRootTraverse(T.nextsibling);//后根遍历树中根节点的其他子树

}

}//注意树的后根遍历就是二叉树的中序遍历，树的遍历没有中根遍历

（3）层次遍历

public void levelTraverse(CSTreeNode T){ //层次遍历

if(T!=null){

LinkedList<CSTreeNode> q = new LinkedList<>();

q.offer(T); //根节点入队

while (!q.isEmpty()){ //队列不为空

for(T=q.poll();T!=null;T=T.nextsibling)//访问所有的兄弟节点

System.out.print(T.date);

if(T.firstchild!=null) //第一个孩子入队

q.offer(T.firstchild);

}

}

}