ps：本颜色的字体是后续添加内容

参考链接：

大话数据结构.pdf

[**图解数据结构（6）——树及树的遍历**](http://www.cnblogs.com/yc_sunniwell/archive/2010/06/27/1766233.html)

http://www.cnblogs.com/yc\_sunniwell/archive/2010/06/27/1766233.html

**1.什么是树？**

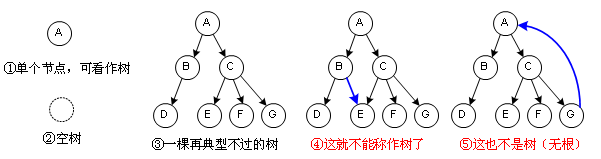
是一种数据结构，可以用来表示层次关系，因表示的样子很像一颗倒立的树而得名。

在数据结构中的特点，是一对多（链表是一对一，图是多对多）

最上面；树根

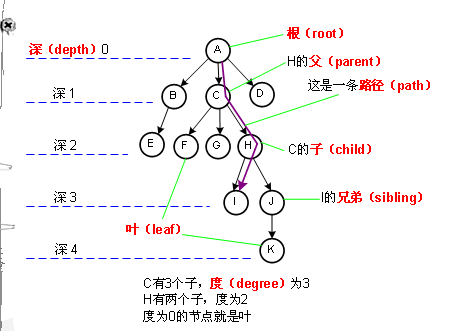
中间：树枝

最下：树叶



**2.树的定义。**

如图：



**树定义（无根树）：**

①一个无根树是一个二元组(V,E); //V是顶点集（非空集合），E是边集（V中元素构成的无序二元组的集合）

（注：个人理解，就是有至少1个点，可以有很多个，这些点的集合，就是V，称为顶点集，我觉得可以理解为结点的集合；

然后这些结点，将会连城一些线段，并且每条线段都没有方向(这里的方向，我认为可以理解为从一个点连向另外一个点，至于为什么说没有方向，我觉得是因为没有起点和终点，线段只是将两个点连起来而已，所以称为没有方向)，而这些线段的结合，就是E，称为边集）

在离散数学中，无根树指无环连通无向图。

所谓的 **无向图**，就是一个图中若每条边都是没有方向的，则称为无向图。

参考链接：http://baike.baidu.com/view/93110.htm

无向图中的边，均是顶点的无序对，无序对通常用圆括号表示。

（注：据我理解，所谓的无序对，就是拿出某个图中的两个点，例如v1和v2，然后(v1,v2)表示的边，和(v2,v1)表示的边是一样的。而像(v1,v2)这样表示，就是指的是无序对，正是因为边没有方向，所以两种表示方法表示的线段才是一样的）

无根树要求每个顶点之间都是直接或者间接相连，且图中没有环（即只有简单路径）。

（注：因为不能有环，所以任意两个点之间，只有一条路径能走的通，假如有两条路径能走得通，就形成了一个环了）

由于树是图的子集（图是什么？推测指的是数据结构？），这一类图具有树的特征，但不具备数的形式，没有特定的根结点，故称为无根树。

（注：这说明，无根树和有根树几乎是一样的，除了有根树有根结点，而根结点又是指定的，即给一个无根树，然后指定一个结点为根，那么这个无根树就成为了有根树）

树是n（n≥0）个结点的有限集。n=0时称为 **空树** ，在任意一棵 **非空树** 中：

（1）有且仅有一个特定的称为 **根（root）**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T1、T2、T3......、Tm，其中每个集合本身又是一棵树，并且称为根的子树。

用上图举例子的话，就是A为根时，是一棵树，然后这棵树从根部分离出以B、C、D三个结点为根的子树，其中B子树有2个结点，C子树有7个结点，D有1个结点。

注：

①当n>0时，即至少有一个结点时，那么树必然有一个根结点，且只有一个根结点；

②子树之间必然是互不相交的（否则就会出现多个上级结点指向同一个下级结点，或者是同级结点之间互相连接）

**有根树：**

①树根：在无根树中，可取树中任意一个结点为根r，使树变为一颗有根树。

②父子关系：（1）对于任意非根结点v，v到树根r的路径上的第一个结点p，定义为v的父结点，而v是p的子结点。

（2）一个结点可以有多个子结点（也可以是0个，例如这个树中，只有一条线连接着这个结点）；

（3）非根结点只能有一个父结点（如果是根出来的第一个结点，那么这个结点就是自己的父结点么？）

（4）树根没有父结点。

数据结构中的树一般是有根树。

**有根树：**

①有根树的递归定义：

（1）没有结点是一棵树（可称为空结点）；

（2）单独的一个结点是一颗树；

（3）如果t0、t1、t2、……、tk-1是一颗树，t是一个结点，则将t0、t1、t2、……、tk-1作为t的子结点后，组成的树t，也是一棵树，树根为t。（注意：一般有根树使用根结点来标识）

——我猜测，从t0到tk-1之间应该也可以将其分别视为一个结点，然后进行连接（以树的形式，即任意两个结点之间只有一条路径），就形成了一个有根树（树根为t）

**树的深度和高度：**

树的高度和树的深度似乎不一样，貌似说法有几种（比如某些大学教科书）：

有的说高度和深度一样；

有的说高度比深度少一；

有的说根结点从0开始计算；

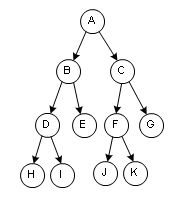
有的说根结点从1开始计算。

根据我个人理解，如图那样，有5层，根直接相连的，其深度则为1，高度为3；最下层的，其深度为4，高度为0。

推测，深度就是指从根开始的距离（即从根开始计算，假如根为第0个结点，那么连接他和根的线段中，他是第几个结点，深度就是几）；

而高度就是指，一个结点，和最底层距离有多远，假如最多是5层，这个结点和根的距离只有1（即位于第4层），那么他的高度就是4。

**3.二叉树**



二叉树是树的一种。

二叉树的特征是，除了叶以外的结点，都有两个子。（叶就是在它和根的路径上，没有比他更远的结点了，也可以理解为，只有一个结点和它相连，并且它不是根，那么他就是叶）

简单来说，就是在有根树的基础上，一个结点，往更深的地方延伸时，最多只能延伸出来零个，或者两个结点（只能是0个或者2个，不能是1个。如果是0个就是叶，否则就是结点），那么他就是二叉树。

例如：图中延伸出0个结点（叶）有：E，G,H,I,J,K

而延伸出2个结点的有：A,B,C,D,F

**4.树的遍历**

所谓树的遍历，是指对树中所有结点的信息的访问。

其重点有2个：

①对所有结点进行访问（假如有结点没有被访问，肯定算不上遍历了）

②对每个结点只访问一次；

说起遍历，那么前提是树必须有顺序。

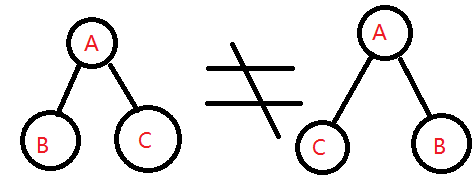
假如A结点是根结点，B和C是他的子（二叉树）。那么如何遍历呢？

先访问A再访问B最后C？还是A->C->B？又或者是先访问B或者C？

这些都是不一样的。

只有有了顺序，才能决定怎么访问。同样以A、B、C三个结点为例，A是根结点，B左C右，或者是B右C左，其并非是同一个树。

根据我个人所知推断，假如有这样一个树，他最终是要存储到硬盘里的，如果B在A左边，C在A右边，其必然是有一定原因和规律的。例如，三个数都是int值，B=10，A=20，C=30。那么以数组形式存储，其为{B,A,C}。比A小的放A左边，比A大的放A右边，数组以增序排列。假如值不变，形式为{C,A,B}，那么数组就变为以减序排列了。增序和减序，显然是不一样的。



因此，树的左右顺序是不能颠倒的。

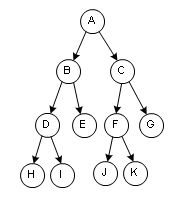
疑问：我找的资料上说的是二叉树其左右是不能颠倒的，那么三叉树，或者其他树可以颠倒么？我推测应该也是不能的吧，至于为什么，需要查查资料。

**二叉树的遍历方法主要有三种：**

①前序遍历；（先访问根结点，再访问子结点）

②后序遍历；（先访问子结点，再访问根结点）

③中序遍历（只适用于二叉树）

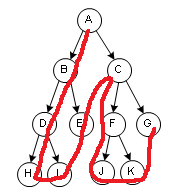


**前序遍历：**

顺序：访问根结点——》访问左子树——》访问右子树

在访问子树，依然按该顺序进行访问。

以上图中的二叉树为例，前序遍历的顺序是



其逻辑是：

（1）先访问根；

（2）当前如果是根，且无子结点，则结束；如果是根，有子结点，第二次返回根的时候结束；（在编程时如何做到这一点？）

（3）查看有没有没有访问过的子结点（问题，如何判断是不是访问过的）：

（3.1）如果有，则访问子结点的左边部分，重复（2）；

（3.2）如果没有，则判断当前结点是不是子结点的左结点：

（3.2.1）如果是，则访问右结点，重复（2）；

（3.2.2）如果不是，返回父结点，重复（2）

**后序遍历：**

逻辑是：

（1）结点有子节点的话，先访问子节点（并且先左后右再根）；

（2）然后（1）循环。

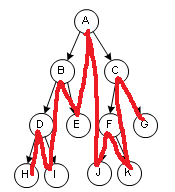
**中序遍历：**

所谓的中序遍历，就是指，先访问左子树，再访问根，再访问右子树。

具体到每一个子树时，也是如此。

与前序遍历和后序遍历不同的是，中序遍历在访问子树的过程中，并不直接访问路径上的，而是先要确定该子树还有没有子结点，如果没有了，才访问它，如果有，则继续访问子树的子树。

例如：



以图为例，由于B是A的子，D是B的子，H是D的子，而H没有子了。因此，先访问H。

然后访问H的父（D），访问D。

然后D的右子是I，因此访问I。

I没有子了，返回D；D的子树访问完了，因此返回D的父（B），访问B；

B的右子，因此访问E；

E没有子了，因此返回B，而B的子树也访问完了，因此返回A，访问A；

A的右子树是C，而C有子（F），F有子（J），J无子，因此访问J；

返回J的父F，访问F；

F的右子是K，访问K；

K无子，返回F，F的子树访问完了，返回C，C的右子是G，G无子，访问G。

树的遍历结束。其顺序是：H-D-I-B-E-A-J-F-K-C-G。

形象的说，就是先访问左子树，假如左子树没有子，则访问结点，并返回，访问父，再访问父的右子树。然后重复即可。——左-父-右的顺序

**树的 结点：**

包括一个数据元素，和从这个元素，指向其各个子树的分支（但不包括指向其父树的分支）。

结点拥有的子树数，称为 **结点的度**（Degree），度为0的结点，称为**叶结点**（Leaf）或终端节点；

度不为0的结点，称为**非终端结点**或**分支结点**。

除根结点外，分支结点也称为内部结点。

**树的度**为树内各节点的度的最大值。

简单来说，树的每个结点都有度，看他往下连向几个结点（只考虑下一层，不考虑更深部分），其度就是几，没有就是0。

没有继续往下连接的（叶），就是叶结点；

不是根也不是叶的，就是内部结点；

每个结点都有度，所有结点中，度最大的结点的度，就是树的度。

**树的顺序：**

之前说过，二叉树的结点的左右子树是不能交换的，也就是说，有顺序的。

假如一个树，其结点的子树之间的左右顺序，是有次序的，不能互相交换，那么就说这个树是**有序树**，否则就是**无序树**。

（三）

**树常用的基本方法：**

①构建一个空树；

②销毁一个树；

③按给的树的定义，来构造一个树（不懂，不太明白这个如何给）；

④若树存在，将树清为一个空树；

⑤若T为空树，返回true，否则返回false；

⑥返回树的深度；

⑦返回树的根节点；

⑧某结点cur\_e是树T的一个结点，返回此结点的值（应该说的是结点的数据部分的值）；

⑨给树T的结点cur\_e赋值为value（这个value是我们给的）；

⑩若cur\_e是树T的非根结点，则返回它的父结点，否则返回空；（原文是双亲，但是树只有一个父结点，不会有2个）；

（11）若cur\_e是树T的非叶结点，则返回它的最左孩子，否则返回空；

（12）若cur\_e有右兄弟，则返回它的右兄弟，否则返回空（但若有多个怎么办？）；

（13）指针p指向树T的某个结点，i为所指向的结点的度+1，非空树C与T不相交，操作结果：将树C插入树T，位置是，指针p指向的结点的第i棵子树的位置（即假如有结点有二棵子树，那么C就是结点的第三棵子树）；

（14）p指向树T的某个结点，i为所指结点p的度，操作结果：删除树T中p所指的第i棵子树。

**树的存储结构：**

需要考虑两部分：

①树的整体结构如何表示；

因为实际中，树的整体结构，最终往往是以数组／链表的形式来存储（不可能像树的图形图那样存储）。因此，需要有一个结构，用于表示树的整体结构。

②树的某一个结点如何表示；

需要有一个结构，用于表示这个结点在该树的数组／链表中的位置。

由于之前没有学过树的存储结构，因此特别说明：

树的存储结构分为两部分：①数据域；②指针域；

所谓的**数据域**，就是这个数组的数据内容部分。如之前学过的链表结构一样，

1. **struct** Node //node是节点的意思
2. {
3. Item item;
4. Node\* next; //struct Node表示struct Node的指针，貌似只用Node\*next也可以
5. };

Item是数据部分，用于储存数据（也许是int，也许是char\*）

而Node\*是**指针域**，其用于表示这个结点和其他结点之间的关系。注意，指针域不一定非要用指针类型，也许是非指针类型。

（1）双亲表示法：

如果树的整体结构以数组来存储的话。

则是

struct Node

{

string mm;

int parent;

};

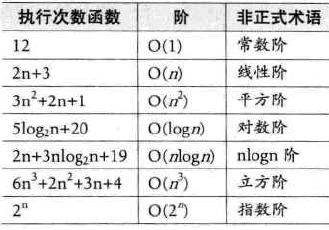
其中，mm是数据域，parent是其父结点下标的int值。由于根节点没有父结点，因此可以规定根结点的parent值为-1（数组下标最小为0）

由于每一个结点都保存了其父结点的值，因此，当我们有一个子结点时，可以轻易找到其父结点。而父结点又能继续找到他的父结点，直到parent值为-1时，找到了根节点。

具体过程可以自己画一个树形图，很简单。

时间复杂度为O（1）。

**关于时间复杂度：**



这里的执行函数次数，指的是最差情况下的次数。

①当固定若干次（执行函数的次数），则为常数；

②当数据的数量越多，次数成正比上升（可能会乘以一个固定值），那么是线性阶，比如数组；

③数据越多，需要的次数是乘方级的，则是平方阶；

④数据越多，但需要的次数比线性慢，例如二分查找法，数量翻一倍，次数增加一次，这种是对数阶；

⑤数据越多，需要的时间是三次方上升，是立方阶（还有4次方5次方等）；

⑥数据越多，需要的时间指数级别上升，是指数阶；

一般情况下，需要时间从少到多的顺序是：常数——》对数阶——》线性阶——》nlongn阶——》平方阶——》立方阶——》阶乘——》指数

后面几种比较少见，因为其效率太低，一般不考虑。

**缺点：**如果要找一个结点的子节点的话，会很麻烦，需要遍历整个树。

改进办法：再设置一个int类型变量：int lchild; 用于描述其最左子结点。

之所以是最左，是因为他可能有多个子结点，也可能没有子结点。没有子结点很简单，值为-1即可。如果有多个子结点，必然只能指向其中的一个，因此设置为最左子结点（当然，最右也可以）。

这样的话，如果树只有0个结点或者1个结点，那么找起来是很简单的。

但显然，树不可能只有0个结点或者1个结点，如果有2个结点，甚至多个结点呢？书上说，有2个子结点的话，在知道左子结点的时候，很容易找到右子结点，但我觉得不现实啊？你怎么知道有2个子结点，怎么知道右子结点是哪个？

改进办法2nd：设置一个int类型变量：int lbrother; 其效果是指向该结点右边的兄弟结点。这样的话，即使有多个子结点，也很容易找到某个子结点。

（2）孩子表示法

数组太麻烦了，不如上链表吧。

链表的特点是指针，他可以指向某个结点。父节点有若干个指针，分别指向自己的各个子节点。

指针的表示方法有三种：

①找树中最大的度，最大的度为指针的数量；

②一个指针，指向一个指针数组，指针数组的成员数量，是这个结点的子节点数量；

③每一条路径都是一个链表，其中链表的第一个指针，存储在一个二维数组之中，然后调用指针即可。

（3）兄弟表示法

每个结点，有两个指针。

第一个指针，指向它的最左边的孩子；

第二个指针，指向他右边的兄弟；

假如没有孩子，则为空；

假如右边的兄弟已经没有更右边的兄弟了，则为空；

**二叉树的遍历：**

所谓二叉树的遍历，就是指访问二叉树每一个结点。

用途有例如：①查找指定数据；②输出每一个遍历的结点的内容；

遍历方式分为前序、中序、后续。

以输出结点内容为例：

①二叉树的数据结构：

struct Tree

{

data m;

Tree\* Lchild = NULL, \*Rchild = NULL;

};

②前序遍历函数：

void PreOrderTraverse(Tree\*T) //前序遍历

{

if (T == NULL) //如果指向的是空结点，则返回

return;

std::cout << T->m << std::endl;

PreOrderTraverse(T->Lchild); //继续遍历左子结点

PreOrderTraverse(T->Rchild); //最后遍历右子结点

}

③中序遍历函数：

void MidOrderTraverse(Tree\*T) //中序遍历

{

if (T == nullptr)

return;

MidOrderTraverse(T->Lchild);

std::cout << T->m << std::endl;

MidOrderTraverse(T->Rchild);

}

④后序遍历函数：

void LastOrderTraverse(Tree\*T) //后序遍历

{

if (T == nullptr)

return;

LastOrderTraverse(T->Lchild);

LastOrderTraverse(T->Rchild);

std::cout << T->m << std::endl;

}

假如是查找，则需要修改遍历的逻辑，并且函数也应该加入条件判断。

例如，

void PreOrderTraverse(Tree\*T,data &p,Tree\* q) //前序遍历查找，p是需要查找的数据，q是指向该结点的指针

{

if (T == NULL) //如果指向的

return;

if (p == T->m)

q = T;

PreOrderTraverse(T->Lchild, p, q);

PreOrderTraverse(T->Rchild, p, q);

}

以上是我自己写的，也许还有更好的。

当然，假如二叉树是有序的（即左子树的值必然比根小，右子树的值必然比根大），那么必然存在一个更好的查找算法。

**二叉树的建立和插入：**

书上列的仅仅是按自己预想设计那样建立一个二叉树。如下面代码：

void CreateTree(Tree\*\*T,int p)

{

data temp;

if (p == 0)return;

std::cout << "输入一项内容：";

std::cin >> temp;

if (temp == 0) //这里假设遇见0时，认为是空结点（即所有结点的内容必然是非0的）

{

\*T = NULL; //让指向当前指针的是空指针

return;

}

else //采用前序遍历方法创建一个新树

{

Tree \*q = new Tree;

\*T = q;

(\*T)->m = temp;

CreateTree(&(\*T)->Lchild,p-1); //因为要传递指向该指针的地址，因此要在实际地址（(\*T)->Lchild)之前加地址运算符&

CreateTree(&(\*T)->Rchild,p-1);

}

}

以上这个树是无序的。

也可以通过创建一个空树：

void CreateTree(Tree\* T)

{

T->Lchild = T->Rchild = NULL;

}

然后插入一个子结点：

void InsertTree(Tree\*\*T,Tree\*\*m) //在树T内，插入树m

{

if (\*T == nullptr)

\*T = \*m;

else

{

if ((\*m)->m < (\*T)->m)

InsertTree(&(\*T)->Lchild, m);

else

InsertTree(&(\*T)->Rchild, m);

}

}

这个插入是有序的（依次判断，比当前结点小，则移动到当前结点的左子，否则右子，直到遇见空结点，然后放入空结点）。

二叉树的查找方法之一：

Tree\* FindTree(Tree\*T, data value) //查找某个值，返回其遍历查到的一个结点（设置此方法为前序遍历，具体可以修改）

{

if (T->m == value)

return T;

Tree\*temp;

temp=FindTree(T->Lchild, value);

if (temp == NULL)

{

temp=FindTree(T->Rchild, value);

if (temp == NULL)

return NULL;

else return temp;

}

else return temp;

}

清空一个树：

void DeleteTree(Tree\*T)

{

if (T == NULL) //如果是空指针，则返回（停止递归）

return;

Tree \*templeft = T->Lchild; //临时指针，用于指向左右子节点

Tree \*tempright = T->Rchild;

delete T;

DeleteTree(templeft);

DeleteTree(tempright);

}

**赫夫曼树：**

特点是：

①是一个二叉树；

②所有字符都在其二叉树的某个叶之中；

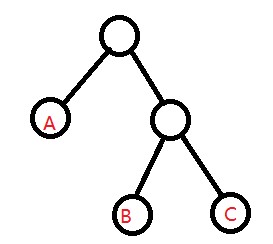
③根据出现频率，来规定字符的位置；

④出现频率越高的，其出现字符的位置越高；

⑤是一种压缩算法；

其原理是，根据字符出现的频率，将其置于二叉树的不同深度的叶上，在遍历的时候，以位的形式来尝试命中某个叶结点，当命中时，即为要找的字符，然后从下一位开始寻找下一个字符。

例如，二叉树是这样的：



ABC分别在某个叶结点上。

当我们正常表示ABC三个子节点时，至少要用2位来表示，例如：A00，B01，C10

而在遍历赫夫曼树时，找左子树记为0，找右子树记为1。其表示为：A0，B10，C11。

在最优情况：AAAA，正常表示法需要8位（00 00 00 00），但赫夫曼树表示法只需要4位（0 0 0 0）即可。

即使最差情况：表示BCBC，正常和赫夫曼树都需要8位。

**注：赫夫曼树需要位数并非就一定比正常表示法小，只是大部分都比正常表示法要小。**

因此，便起到了压缩占用空间的好处。

并且，由于必然命中，假如有某些字符无法命中，那么便可能是文件损坏了。