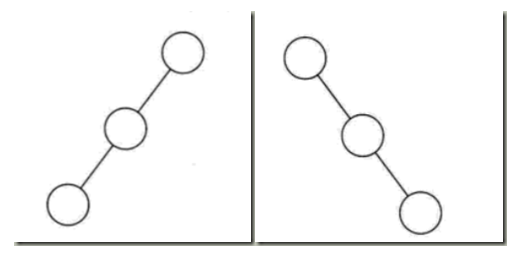
二叉树

二叉树是树的特殊一种，具有如下特点：1、每个结点最多有两颗子树，结点的度最大为2。2、左子树和右子树是有顺序的，次序不能颠倒。3、即使某结点只有一个子树，也要区分左右子树。

**一、特殊的二叉树及特点**

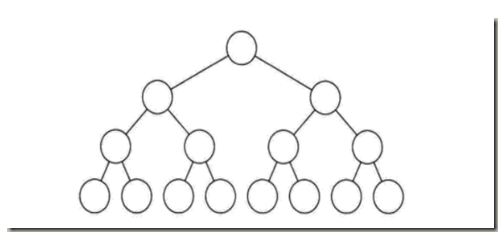
**1、斜树**

所有的结点都只有左子树（左斜树），或者只有右子树（右斜树）。这就是斜树，应用较少



**2、满二叉树**

所有的分支结点都存在左子树和右子树，并且所有的叶子结点都在同一层上，这样就是满二叉树。就是完美圆满的意思，关键在于树的平衡。



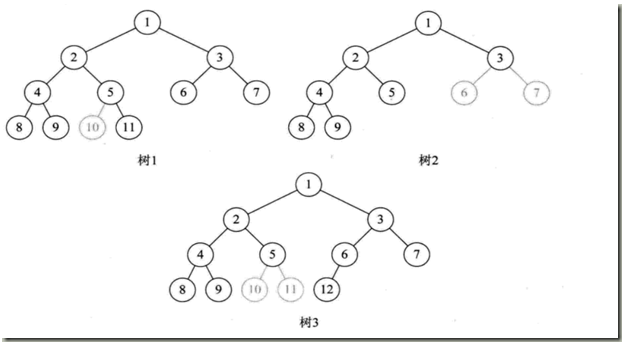
根据满二叉树的定义，得到其特点为：

1. 叶子只能出现在最下一层。
2. 非叶子结点度一定是2.
3. 在同样深度的二叉树中，满二叉树的结点个数最多，叶子树最多。

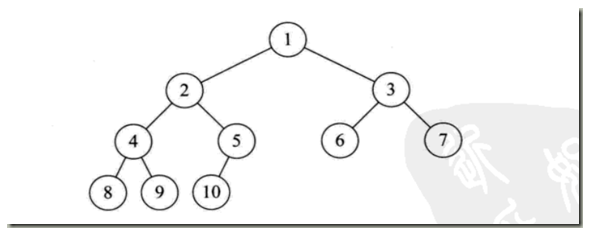
**3、完全二叉树**

对一棵具有n个结点的二叉树按层序排号，如果编号为i的结点与同样深度的满二叉树编号为i结点在二叉树中位置完全相同，就是完全二叉树。满二叉树必须是完全二叉树，反过来不一定成立。

其中关键点是按层序编号，然后对应查找。



在上图中，树1，按层次编号5结点没有左子树，有右子树，10结点缺失。树2由于3结点没有字数，是的6,7位置空挡了。树3中结点5没有子树。



上图就是一个完全二叉树。

结合完全二叉树定义得到其特点：

1. 叶子结点只能出现在最下一层（满二叉树继承而来）
2. 最下层叶子结点一定集中在左 部连续位置。
3. 倒数第二层，如有叶子节点，一定出现在右部连续位置。
4. 同样结点树的二叉树，完全二叉树的深度最小（满二叉树也是对的）。

根据下图加深理解，什么时候是完全二叉树。

**二、二叉树性质**

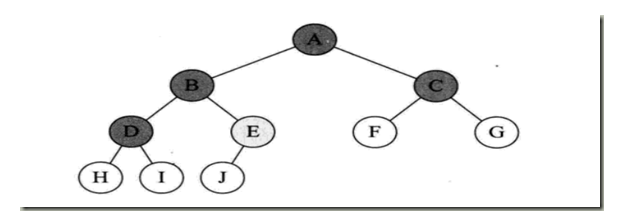
**1、一般二叉树性质**

1、在非空二叉树的i层上，至多有2i-1个节点(i>=1)。通过归纳法论证。

2、在深度为K的二叉树上最多有2k-1个结点（k>=1)。通过归纳法论证。

3、对于任何一棵非空的二叉树,如果叶节点个数为n0，度数为2的节点个数为n2，则有: n0 = n2 + 1

在一棵二叉树中，除了叶子结点（度为0）之外，就剩下度为2(n2)和1(n1)的结点了。则树的结点总数为T = n0+n1+n2;在二叉树中结点总数为T，而连线数为T-1.所以有：n0+n1+n2-1 = 2\*n2 +n1;最后得到n0 = n2+1;



上图中结点总数是10，n2为4，n1为1，n0为5。

**2、完全二叉树性质**

a、具有n的结点的完全二叉树的深度为log2n+1.

满二叉树是完全二叉树，对于深度为k的满二叉树中结点数量是2k-1 = n，完全二叉树结点数量肯定最多2k-1,同时完全二叉树倒数第二层肯定是满的（倒数第一层有结点，那么倒是第二层序号和满二叉树相同），所以完全二叉树的结点数最少大于少一层的满二叉树，为2k-1-1。

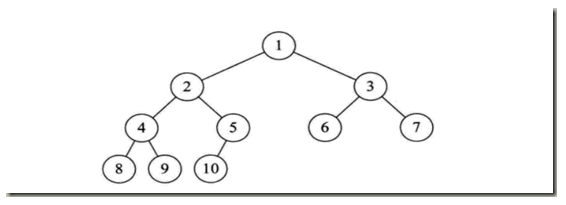
根据上面推断得出： 2k-1-1< n=<2k-1，因为结点数Nn为整数那么n<=2k-1可以推出n<=2k,n>2k-1-1可以推出 n>=2k-1,所以2k-1<n<=2k  。即可得k-1<=log2n<k 而k作为整数因此k=[log2n]+1。

b、如果有一颗有n个节点的完全二叉树的节点按层次序编号，对任一层的节点i（1<=i<=n）有

    1.如果i=1，则节点是二叉树的根，无双亲，如果i>1，则其双亲节点为[i/2]，向下取整

    2.如果2i>n那么节点i没有左孩子，否则其左孩子为2i

    3.如果2i+1>n那么节点没有右孩子，否则右孩子为2i+1



在上图中验证

第一条：

当i=1时，为根节点。当i>1时，比如结点为7，他的双亲就是7/2= 3；结点9双亲为4.

第二条：

结点6,6\*2 = 12>10，所以结点6无左孩子，是叶子结点。结点5，5\*2 = 10，左孩子是10,结点4，为8.

第三条：

结点5，2\*5+1>10,没有右孩子，结点4，则有右孩子。

**三、二叉树遍历**

二叉树遍历：从树的根节点出发，按照某种次序依次访问二叉树中所有的结点，使得每个结点被访问仅且一次。

这里有两个关键词：访问和次序。

**1、前序遍历**

基本思想：先访问根结点，再先序遍历左子树，最后再先序遍历右子树即根—左—右。

图中前序遍历结果是：1，2，4，5，7，8，3，6。

a/前序递归遍历的代码实现，如下所示

//前序递归遍历

void PreOrderTraverse(BiTree t)

{

//注意跳出条件

if(t != NULL)

{

//注意访问语句顺序

printf("%c ", t->data);

PreOrderTraverse(t->lchild);

PreOrderTraverse(t->rchild);

}

}

前序非递归遍历:

对于任一结点p：

        a. 访问结点p，并将结点p入栈；

        b. 判断结点p的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点p，循环置a；若不为空，则将p的左孩子置为当前结点p；

        c. 直到p为空，并且栈为空，则遍历结束。

//前序非递归遍历

int NoPreOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree tmp = t;

if(tmp == NULL)

{

fprintf(stdout, "the tree is null.\n");

return ERROR;

}

//现将左子树压入栈，当到叶子结点后，出栈，获取右子树，然后在压入右子树的左子树。

//顺序不能变

while((tmp != NULL) || (IsEmpty(&s) != 1))

{

while(tmp != NULL)

{

Push(&s, tmp);

printf("%c ", tmp->data);

tmp = tmp->lchild;

}

if(IsEmpty(&s) != 1)

{

Pop(&s, &tmp);

tmp = tmp->rchild;

}

}

return OK;

}

**2、中序遍历**

基本思想：先中序遍历左子树，然后再访问根结点，最后再中序遍历右子树即左—根—右。

图中中序遍历结果是：4，2，7，8，5，1，3，6。

中序遍历迭代代码

//中序递归遍历

void InOrderTraverse(BiTree t)

{

if(t != NULL)

{

InOrderTraverse(t->lchild);

printf("%c ", t->data);

InOrderTraverse(t->rchild);

}

}

2）中序非递归遍历

    根据中序遍历的顺序，对于任一结点，优先访问其左孩子，而左孩子结点又可以看做一个根结点，然后继续访问其左孩子结点，直到遇到左孩子结点为空的结点才停止访问，然后按相同的规则访问其右子树。其处理过程如下：

       对于任一结点：

       a. 若其左孩子不为空，则将p入栈，并将p的左孩子设置为当前的p，然后对当前结点再进行相同的操作；

       b. 若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的p置为栈顶结点的右孩子；

       c. 直到p为空并且栈为空，则遍历结束。

//中序非递归遍历二叉树

int NoInOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree tmp = t;

if(tmp == NULL)

{

fprintf(stderr, "the tree is null.\n");

return ERROR;

}

while(tmp != NULL || (IsEmpty(&s) != 1))

{

while(tmp != NULL)

{

Push(&s, tmp);

tmp = tmp->lchild;

}

if(IsEmpty(&s) != 1)

{

Pop(&s, &tmp);

printf("%c ", tmp->data);

tmp = tmp->rchild;

}

}

return OK;

}

**3、后序遍历**

基本思想：先后序遍历左子树，然后再后序遍历右子树，最后再访问根结点即左—右—根。

图中后序遍历结果是：4，8，7，5，2，6，3，1。

后序递归遍历代码实现，如下所示。

//后序递归遍历

void PostOrderTraverse(BiTree t)

{

if(t != NULL)

{

PostOrderTraverse(t->lchild);

PostOrderTraverse(t->rchild);

printf("%c ", t->data);

}

}

  后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子和右孩子都已被访问，并且左孩子在右孩子之前访问才能访问根结点，这就为流程控制带来了难题。下面介绍一种思路。

     要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点p，先将其入栈。若p不存在左孩子和右孩子，则可以直接访问它，或者p存在左孩子或右孩子，但是其左孩子和右孩子都已经被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将p的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子之前别访问，左孩子和右孩子都在根结点前面被访问。

//后序非递归遍历二叉树

int NoPostOrderTraverse(BiTree t)

{

SqStack s;

InitStack(&s);

BiTree cur; //当前结点

BiTree pre = NULL; //前一次访问的结点

BiTree tmp;

if(t == NULL)

{

fprintf(stderr, "the tree is null.\n");

return ERROR;

}

Push(&s, t);

while(IsEmpty(&s) != 1)

{

GetTop(&s, &cur);//

if((cur->lchild == NULL && cur->rchild == NULL) || (pre != NULL && (pre == cur->lchild || pre == cur->rchild)))

{

printf("%c ", cur->data); //如果当前结点没有孩子结点或者孩子结点都已被访问过

Pop(&s, &tmp);

pre = cur;

}

else

{

if(cur->rchild != NULL)

{

Push(&s, cur->rchild);

}

if(cur->lchild != NULL)

{

Push(&s, cur->lchild);

}

}

}

return OK;

}

**四、二叉树的建立**

其实而二叉树的建立就是二叉树的遍历，只不过将输入内容改为建立结点而已，比如，利用前序遍历建立二叉树

//创建树

//按先后次序输入二叉树中结点的值(一个字符),#表示空树

//构造二叉链表表示的二叉树

BiTree CreateTree(BiTree t)

{

char ch;

scanf("%c", &ch);

if(ch == '#')

{

t = NULL;

}

else

{

t = (BitNode \*)malloc(sizeof(BitNode));

if(t == NULL)

{

fprintf(stderr, "malloc() error in CreateTree.\n");

return;

}

t->data = ch; //生成根结点

t->lchild = CreateTree(t->lchild); //构造左子树

t->rchild = CreateTree(t->rchild); //构造右子树

}

return t;

}