二叉树实现（数组和链表）

**树**

是一种数据结构，树的一些相关的术语：

结点的度：一个结点的后继结点的个数。

树的度：树中度值最大的结点的度被称为树的度。

树的深度：树的层次数。

分支结点：度值大于0的结点，分支结点至少含有一个后继，分支结点也称为非终端结点。

叶子结点：树中的度值为0的结点。

双亲结点：树中某个结点的前驱结点，也成为父节点。

子女结点：树中某结点的后继结点。

兄弟结点：树中同一层的结点。

**二叉树**

（1）二叉树是一种特殊的树，树的度值最大为2.

（2）二叉树的表示：

           A

         /      \

       B        C

      /     \     /   \

     D   E   F   G

二元表示：

Binary-tree = (D, S)

D={A, B, C, D, E, F, G}

S={<A,B>, <A,C>, <B,D>, <B,E>, <C,F>, <C,G>}

**3.二叉树的特点**

（1）任何一棵二叉树的叶子结点总比双分支结点个数多一个。

分别设叶子结点，单分子结点，双分子结点的个数为：n0, n1, n2，则有：

n0+n1+n2 = 2n2+n1+1 ==>  n0 = n2+1

2n2+n1+1：n2结点每个节点有两个叶子节点，n1结点每个结点有一个叶子结点，+1的原因是根结点不属于任何一个结点的子节点，所以根结点需要单独加上。

（2）二叉树的第i层最多有2^(i-1)个元素。

（3）一棵h层的二叉树最多有2^h - 1个元素。

**4.满二叉树**：h层的二叉树共有2^h - 1个元素。

**完全二叉树**：h层的二叉树若前h-1层是满的，最后一层是连续缺失右边的若干个结点。

用数组实现二叉树，父亲节点和孩子节点下标分二种情况,.

如果根节点下标从1开始存储，则编号为i的结点的主要关系为：  
双亲：下取整 （i/2）  
左孩子：2i  
右孩子：2i+1  
  
如果根节点下标从0开始存储，则编号为i的结点的主要关系为：  
双亲：下取整 （(i-1)/2）  
左孩子：2i+1  
右孩子：2i+2

二叉树存储的实现：

（1）顺序存储：将二叉树存储在内存中一块连续的存储单元中，数组的元素个数取决于二叉树的层数。

（2）将二叉树的每个结点的编号作为该结点在数组中的存储下标。

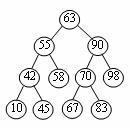
（3）将根节点存放在下标为1的单元里。

（4）若下标为i的单元存在左子女，则左子女的下标为2i，若存在右子女，右子女的下标为2i+1。

6.中根查找顺序

对于一个给定的二叉树，如果要对其进行遍历可以有前序遍历、中序遍历、后序遍历基本的三种遍历方式，这三种遍历方式都是相对于根而言的。比如中序遍历，就是先访问左子树，再访问根，最后访问右子树，意思就是根的访问顺序在中间。

对于一棵给定的二叉树：



中序遍历的顺序为：10, 42, 45, 55, 58, 63, 67, 70, 83, 90, 98。

从这个遍历顺序可以看出正好是一个从小到大的排过序的顺序，这是在构建二叉树的时候就可以做到的，在构建二叉树的时候，选择好根节点，比根节点小的数放到根节点的左子树，比根节点大的数放到根节点的右子树即可。

例如给出一个序列：63, 42, 45, 67, 70。

访问到63，将其作为根节点，访问到42<63则作为63的左子树，访问到45<63放到63的左子树上，而45>42所以放到42的右子树上，访问到67>63，则将67作为63的右子树，访问到70>63应该放在63的右子树上，而进一步有70>67，则70应该作为67的右子树，所以可以得到下面的二叉树：

                        63

                      /    \

                    42      67

                      \       \

                       45      70

7.二叉树的链式存储

结点类型：left域、data域、right域

left域：用来存放左子女的地址，若不存在左子女，left域为空。

data域：用来存放该结点的数据。

right域：用来存放右子女的地址，若不存在右子女，right域为空。

8.二叉搜索树

若树中左子女的值都小于根节点的值，右子女的值都大于根节点的值，这样的二叉树叫做二叉搜索树。

9.二叉树的数组实现方式

分析：首先给定一个二叉树之后，就可以知道该二叉树的层次数h，然后创建一个2^h = 2^h - 1 + 1个元素的数组，加1的目的是因为数组的0下标是不用的，根节点存放在下标为1的单元中。所有数组中没有用到的单元，均赋值为二叉树中数值域不包含的值，这样在遍历的时候就可以将这个值作为判断的依据。

#**include** <stdio.h>

**void** **create\_btree**(**int** list[], **int** bt[], **int** n) /\*n表示list数组中元素的个数\*/{

**int** i, order;

bt[1] = list[0];

**for**(i = 1; i < n; i++)

{

order = 1; /\*每次进来从根结点开始比较\*/

**while**(bt[order] != 0)

{

**if**(list[i] < bt[order])

order \*= 2;

**else**

order = order\*2 + 1;

}

bt[order] = list[i];

}

}

**int** **main**(){

**int** list[7] = {30, 18, 16, 25, 34, 7, 31};

**int** bt[16] = {0};

**int** i;

create\_btree(list, bt, 7);

**for**(i = 0; i < 16; i++) /\*按层输出\*/

**if**(bt[i] != 0)

printf("%4d", bt[i]);

printf("\n");

**return** 0;

}

程序的输出结果：

IMG_257

二叉树的链表实现：

#**include** <stdio.h>#**include** <malloc.h>

#**define** NULL 0

**typedef** **struct** tree

{

**int** data;

**struct** tree \*left, \*right;

}ElemBT;

**void** **create\_btree**(ElemBT \*root, **int** list[], **int** n) /\*n表示list数组中元素的个数\*/{

**int** i;

ElemBT \*current, \*parent, \*p;

**for**(i = 1; i < n; i++)

{

p = (ElemBT \*)malloc(**sizeof**(ElemBT));

p->left = p->right = NULL;

p->data = list[i];

current = root;

**while**(current != NULL)

{

parent = current;

**if**(current->data > p->data)

current = current->left;

**else**

current = current->right;

}

**if**(parent->data > p->data)

parent->left = p;

**else**

parent->right = p;

}

}

**int** **main**(){

**int** list[7] = {30, 18, 16, 25, 34, 7, 31};

ElemBT \*root;

root = (ElemBT \*)malloc(**sizeof**(ElemBT));

root->data = list[0];

root->left = root->right = NULL;

create\_btree(root, list, 7);

**return** 0;

递归建二叉树

1. **struct** binary\_tree\_node
2. {
3. **int** data;
4. binary\_tree\_node\* left\_child;
5. binary\_tree\_node\* right\_child;
6. };
8. **void** create\_tree(binary\_tree\_node\* &tree, **int** a[], **int** len, **int** index)
9. {
10. **if** (index >= len)
11. **return**;
12. tree = **new** binary\_tree\_node;
13. tree->data = a[index];
14. tree->left\_child = NULL;
15. tree->right\_child = NULL;
16. create\_tree(tree->left\_child, a, len, 2 \* index + 1);
17. create\_tree(tree->right\_child, a, len, 2 \* index + 2);
18. }
20. **void** Print(binary\_tree\_node\* p)
21. {
22. **if** (NULL == p)
23. **return**;
24. cout << p->data;
25. Print(p->left\_child);
26. Print(p->right\_child);
27. }
29. **int** main()
30. {
31. **int** p[7] = { 0, 1, 2, 3, 4, 5, 6 };
32. binary\_tree\_node\*tree;
33. create\_tree(tree, p, 7, 0);
34. Print(tree);
35. **return** 0;
36. }