树的结构主要是为了查找，这个主要是为了搜索，树的结构关注的不是增删查改

**树**

广义上面的树的结构我们不知道树的一个节点是有几个子节点的，所以这个时候我们需要定义的一种结构就是，一个节点的孩子是可以动态的增加的，这个时候我们就可以把一个一个的节点放置在vector中，这个时候某一个节点的孩子节点是可以动态的增加的  
还有一种结构就是，我们可以把一个节点中定义两个节点指针，一个指向的是当前节点的兄弟，一个指向的是当前节点的第一个孩子节点，那么它的其他的孩子节点就由第一个孩子节点去标记了

**struct** **TreeNode**

{

int \_data;

TreeNode\* \_firstchild;

TreeNode\* \_nextSibling;

}

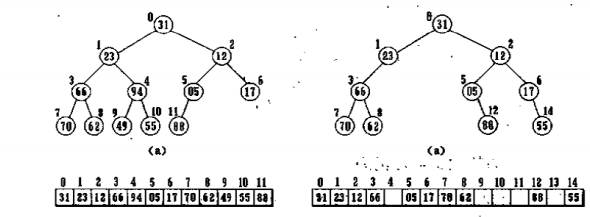
树的应用：文件管理

**二叉树的结构**

上面的广义上的树不是我们学习的重点，我们学习的重点的是二叉树，包括现实中的使用最多的也是二叉树，以及一些成熟的数据结构也是用二叉树的一些包装。

二叉树的结构

* 我们可以把一个数组看成是一个二叉树，这个在以后的堆里面会有很好的运用，虽然数组在实际物理上面是连续存储着的，但是我们在访问一些数据，以及对数据的一些操作的时候，是按照树的结构进行处理的，比如下面的一种情况，首先看一个图



上图中，实际在内存中存储数据的时候，是按照数组的方式存储的，但是我们对数据访问操作的时候是按照树的结构去访问的，比如要访问一个节点的子节点，而当前节点的下标是n，那么它的左孩子节点的下标就是n \* 2+1，右孩子节点就是n \* 2+2，同样的，一个的父亲节点的下标是(n-1)/2

左孩子 n \* 2 + 1  
右孩子 n \* 2 + 2  
父亲节 (n - 1)/2

分析：如果我们的数据很集中，比如满二叉树或者是完全二叉树的时候可以使用上面的结构，但是如果我们的数据不是连续的，甚至是中间有很多空闲的位置的时候，就不建议使用数组这样的结构，这回造成空间的很大的浪费。

* 常用的一种结构是一个节点有两个指向孩子的指针。  
  这种结构叫做二叉链

template<class T>

struct TreeNode

{

T \_data;

TreeNode<T>**\*** \_left;

TreeNode<T>**\*** \_right;

}

* 还有一种结构是有一个指向父亲的节点，有两个指向孩子的节点  
  这种结构是三叉链

template<class T>

struct TreeNode

{

T \_data;

TreeNode<T>**\*** \_father;

TreeNode<T>**\*** \_left;

TreeNode<T>**\*** \_right;

}

**创建一个树**

传入一个数组，然后使用前序（其他顺序），构建一棵树，传参的时候，需要传入的时候是数组，数组大小还有一个就是非法值  
从数组里面一个一个取数据，然后构建到树里面直到，遇到非法值或者是遇到了结尾的时候，这个时候偶就让当前节点的根为NULL

创建一个树，在构造函数中需要调用的函数是creatTree，然后参数是数组，数组大小，非法值，数组下标，返回值是node\*

把这个创建树的代码背下来

{1,2,3，0，0，4，0，0，5,6}  
这里我们的非法值是0，首先看下面的代码

**#include<iostream>**

**using** **namespace** std;

*// {}*

**template**<**class** T>

**struct** TreeNode

{

T \_data;

TreeNode<T>\* \_left;

TreeNode<T>\* \_right;

TreeNode(T data)

:\_data(data)

, \_left(NULL)

, \_right(NULL)

{}

};

**template**<**class** T>

**class** BinaryTree

{

**typedef** TreeNode<T> Node;

**public**:

BinaryTree()

:\_root(NULL)

{}

BinaryTree(**const** T\* arr,**const** T& inval,**const** **size\_t**& n)

{

**size\_t** index = 0;

\_root = \_CreatTree(arr,n,index,inval);

}

**private**:

Node\* \_CreatTree(**const** T\* arr,**const** **size\_t**& n,**size\_t**& index,**const** T& inval)

{

Node\* root = NULL;

**if** (arr[index] != inval && index < n)

{

root = **new** Node(arr[index]);

root->\_left = \_CreatTree(arr,n,++index,inval);

root->\_right = \_CreatTree(arr,n,++index,inval);

}

**return** root;

}

**private**:

Node\* \_root;

};

**void** **TestTree**()

{

**int** arr[] = {1,2,3,0,0,4,0,0,5,6};

BinaryTree<**int**> tree(arr,0,10);

}

下面是对上面代码的一个分析，首先应该注意的一个就是，我们需要在构造函数里面调用一个\_CreatTree()的函数来进行树的创建，因为在 创建树的时候，需要递归的调用，所以这里不能直接一直调用构造函数，这显然是不合理的

创建树的函数 -- \_CreatTree()

接下来主要需要讲述的就是创建树的函数，首先是函数的返回值，因为我们需要递归的调用，在调用的过程中的一个大的思路就是，**首先，让根节点等于函数的返回值，到函数的内部创建一个root的节点指针，然后是如果数组中的数据符合要求就开辟一个节点的空间，然后就是root指向它，接着是是root的左和右分别的调用\_CreatTree()函数，最后返回root。通过这样的操作之后，就可以首先是\_root指向一个节点，接下来是递归的使当前节点的左右孩子节点调用，就可以使左右孩子节点也指向相应的操作**

把上面的创建树的代码，包括下面的一些代码，其实可以考虑背下来，因为这些代码是最最基础的东西了，在理解的基础之上背下来，这些是以后深入学习的一个基础，不光光是要背，更加重要的是理解其中的一些

参数index

参数index为什么使用的是引用呢，这里如果我们不使用引用的话，就会出现一些问题，首先还是看上面的我们给出的例子

**int** arr[] = {1,2,3,0,0,4,0,0,5,6};

首先第一层递归里面的时候，我们的index是0，然后创建一个节点，把1放置进去了，然后进入到下一层递归，这个时候index是1，然后把数据'2'放置进去了，大家记住这层递归里面的index是1，然后进入到第三层递归的时候，index是2，这个时候把数据‘3’放置进去了，接着往下走的时候，index是3，这个时候数据是0，这是个非法值，于是再次递归，这个时候index是4，也是一个非法值，这个时候，关于数据3这一层的递归函数已经结束了，所以返回上一层递归，上一层递归的时候，index是1，就是又回到了数据2的那一层递归调用了，这个时候++index后，又变成了3，所以节点数据2的右孩子又是3，所以这里是错误，这个是我们应该注意的一个问题。

**二叉树的遍历**

前序遍历，中序遍历，后序遍历，层序遍历

看到一个二叉树的时候，想象成三个部分，根节点，左子树，右子树

前序：根左右  
中序：左根右  
后序：左右根  
层序：一层一层

**关于遍历的思想特别的重要**

一定要深刻的理解各种遍历的思想，因为下面的好多问题其实都是遍历思想的变形

二叉树的遍历的这一点的知识很重要，因为后面的很多的内容，很多的题目，很多的求解的方法都是使用了遍历的思想的

前序遍历

思想

**根左右：访问一个节点的时候先访问数据，然后访问左孩子，然后访问右孩子**

**public**:

**void** PrevOrder()

{

\_PrevOrder(\_root);

}

**private**:

void \_PrevOrder(Node\* root)

{

**if** (root != NULL)

{

cout << root->\_data << " ";

\_PrevOrder(root->\_left);

\_PrevOrder(root->\_right);

}

}

前序遍历和后面的很多的遍历包括其他的问题是需要有两个部分组成的，为什么一个设置成公有的，一个设置成私有的呢，因为我们的遍历的时候是需要访问 \_ root的，但是我们肯定不能在外面用户使用的时候直接让他们调用前序遍历的时候，将\_root，作为参数传递进去，这显然是实现不了的，所以这里需要在写一个私有的函数，让公有的函数去调用这个私有的函数，然后让这个私有的函数去实现遍历的功能。

关于函数的实现部分就是按照拿到一个节点的时候首先要做的就是如果当前节点不是NULL的话，就访问数据，如果是NULL的，就结束么，我上面写代码的时候，我把递归调用那里放置在了if判断的外面了，这是错误的，因为放置在if判断的外面的话，当前节点已经是NULL了，我们是无法访问它的左右节点的。

中序遍历

思想

**左根右：访问一个节点的时候，首先访问它的左孩子，然后访问数据，然后在访问访问右孩子**

**public**:

**void** **InOrder**()

{

\_InOrder(\_root);

cout << endl;

}

**private**:

void \_InOrder(Node\* root)

{

**if** (root != NULL)

{

\_InOrder(root->\_left);

cout << root->\_data << " ";

\_InOrder(root->\_right);

}

}

没什么好解释的了，结合着前序遍历就可以理解中序遍历，然后就可以理解后序遍历了

后序遍历

思想

**左右根：访问一个节点的时候，首先访问它的左孩子，然后在访问访问右孩子，然后访问数据**

**public**:

**void** **EndOrder**()

{

\_EndOrder(\_root);

cout << endl;

}

**private**:

void \_EndOrder(Node\* root)

{

**if** (root != NULL)

{

\_EndOrder(root->\_left);

\_EndOrder(root->\_right);

cout << root->\_data << " ";

}

}

层序遍历

思想  
一层一层的遍历  
层序遍历的代码中我们使用一个队列，首先我们把根节点放置在队列中去，这个时候队列里面至少有一个节点了 ，接下来要做的就是每次从队列中拿取一个数据，然后把这个节点的数据打印出来，接着就是判断这个节点的左右两个孩子是否为NULL，如果不是的就把左右两个节点指针放置在队列中去。

public:

void **LevelOrder**()

{

queue<**Node**\*> que;

**if** (\_root)

{

que.push(\_root);

}

**while** (!que.empty())

{

**Node**\* cur = que.front();

que.pop();

cout << cur->\_data << " ";

**if** (cur->\_left)

{

que.push(cur->\_left);

}

**if** (cur->\_right)

{

que.push(cur->\_right);

}

}

}

**二叉树的各种求解**

**二叉树的各种求解中都基本上使用了二叉树遍历的一些思想**

**需要用到递归的思想：就是我想要求当前节点的某个解，我需要求解它的左右子树的某个解**

求树的所有节点的个数

思想  
我需要求解以当前节点为根的节点的个数，我需要求解当前节点的子节点的个数  
返回条件就是，如果当前节点为NULL的话，就直接返回0，如果当前节点不是NULL，就返回当前节点的孩子节点的个数+1，这个1表示的就是当前节点的值。  
这个求解的过程可以理解为一个后序遍历的思想，就是我想要给当前节点的所有节点的个数，我首先应该求解的是当前节点的子节点的个数，然后才能给当前节点的个数加上+1。

**代码很重要，背下来，在这个代码的基础之上，思考其他的一些内容**

**int** **NodeNum**()

{

**return** **\_NodeNum**(\_root);

}

**private**:

**int** \_NodeNum(Node\* root)

{

**if** (root == NULL)

{

**return** 0;

}

**return** \_NodeNum(root->\_left) + \_NodeNum(root->\_right) + 1;

}

求树的叶子节点的个数

思想  
这个时候其实思想还是使用了后序遍历的思想，就是我想要求解当前节点的叶子节点的个数，我需要求解的是当前节点的孩子节点的叶子节点的个数

**public**:

**int** LeafNum()

{

**return** **\_LeafNum**(\_root);

}

**private**:

**int** \_LeafNum(Node\* root)

{

**if** (root == **NULL**)

{

**return** 0;

}

**if** ((root->\_left == **NULL**) && (root->\_right == **NULL**))

{

**return** 1;

}

**return** \_LeafNum(root->\_left) + \_LeafNum(root->\_right);

}

**这个内容我们需要和上面的求解节点的个数做一个详细的比较，我们都应该以当前节点为一个判断的依据。这个时候我们很容易纠结的一个问题就是，是不是需要单独的判断左右呢，单独的判断左右的时候，又如何进行相加呢，这个时候就容易除了问题了。看我们上面的代码就很好的解决了这个问题，如果当前节点是NULL，，当前节点很可能就是从上面的父节点传递过来的，所以很好的避免了我们需要单独分开判断的问题了。这个思想当中我们需要注意的问题就是我们的返回条件是两种的，一种就是返回0，一种是返回1。**

求第K层的个数

思想  
我们想要求第k层节点的个数，需要求解的是第k-1层节点的个数

**public**:

**int** LevelNodeNum(**int** k)

{

**return** **\_LevelNodeNum**(\_root,k);

}

**int** \_LevelNodeNum(Node\* root,**int** k)

{

**if** (root == NULL)

**return** 0;

**if** ((k == 1) && (NULL != root) )

**return** 1;

**else**

{

**return** \_LevelNodeNum(root->\_left, k - 1) + \_LevelNodeNum(root->\_right, k - 1);

}

}

上面的方式是一种很巧妙的方法，但是我们还是应该理解其中的道理，这个问题和上面的问题一样，我们还是需要考虑子问题还有几十需要考虑的是返回值，这里的返回值也是有两种返回条件，一种是返回0的时候，还有一种是返回1的状态。

**上面的问题让我直接想可能还有点困难，但是让我结合着代码理解的话，问题是不大的**

求二叉树的高度（深度）

思想  
这里还是子问题，如果我们想要求解当前节点的深度，首先需要求解的是当前节点的子节点的高度，然后+1，加的这个1就是当前的一个节点。但是这里返回的时候我哦们需要放回的应该是左右子树最大的那一个，但是如果用三目运算符，或者是使用如下的方式的话，

**if**(\_High(root->\_left) > \_High(root->\_right))

{

**return** \_High(root->\_left);

}

就可能造成了两次的调用了递归函数。所以这里我们采用的是首先用一个数据保存起来的方式。

**int** **High**()

{

**return** **\_High**(\_root);

}

**private**:

**int** \_High(Node\* root)

{

**if** (root == NULL)

{

**return** 0;

}

**int** l = \_High(root->\_left);

**int** r = \_High(root->\_right);

**if** (l > r)

**return** l + 1;

**else**

**return** r + 1;

}

查找某一个节点

思想  
这个问题和其他的问题有点出入的，我们需要从中间的某一个位置返回一个节点指针，但是这里返回就是一个问题，所以我们应该注意到这个问题

**private**:

Node\* Find(T data)

{

**return** **\_Find**(\_root,data);

}

Node\* \_Find(Node\* root,T data)

{

**if** (NULL == root)

{

**return** NULL;

}

**else**

{

**if** (root->\_data == data)

{

**return** root;

}

Node\* ret = \_Find(root->\_left,data);

**if** (ret)

{

**return** ret;

}

ret = \_Find(root->\_right,data);

}

}

**上面的代码背下来，这是学习其他东西和解决问题的一些基础，一开始学习人家的代码就是要后来更好的运用**