这是面微软之前做的一些二叉树的算法题，很多都写好注释了，相信很容易看懂。
有问题可以发邮件给我,邮箱地址 long470884130@163.com
个人主页邓世龙的自留地 http://www.dengshilong.org/
邓世龙
2012年6月15日星期五

```
/*
    在二叉树中找出和为某一值的所有路径（树）
    题目：输入一个整数和一棵二叉树。
    从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。
    打印出和与输入整数相等的所有路径。
    例如输入整数和如下二叉树
      10
      / \
     5  12
    / \
   4   7
    则打印出两条路径：10, 5, 7和10, 12。
    二叉树节点的数据结构定义为：
    struct BSTreeNode  // a node in the binary tree
    {
        int m_nValue; // value of node
        BSTreeNode  *m_pLeft; // left child of node
        BSTreeNode  *m_pRight; // right child of node
    };
    void print_value_path(BSTreeNode  *root,int n)主要调用
    辅助函数void print_value_path(BSTreeNode  *root,int path[],int len, int n)
    程序的主要实现在这个函数中
    void print_value_path_assist(BSTreeNode  *root,int path[],int len, int n)
    数组path用于保存路径，
    len用于保存路径的长度，开始时len值为.
    n是给定整数
    基本步骤如下：
    用数组path保存经过的节点，首先将根节点加入路径中，然后判断根节点是否是叶子节点，
    如果是叶子节点，并且节点的值等于给定的值，则输出路径，
    如果不是，则递归调用函数，输出左子树和右子树的路径，
    递归调用函数时，给定的整数值要减去节点的值，得到新的整数值。
*/
#include <iostream>
#include <cassert>

using namespace std;
struct BSTreeNode {
```

```
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//辅助函数,输出从根节点到叶子节点的所有路径中，路径经过的节点值之和等于给定整数的路径
//数组path用于保存路径，len是路径的长度，n是给定整数
void print_value_path_assist(BSTreeNode  *root,int path[],int len, int n){
    if(root == NULL){
        return ;
    }else{//此节点非空，加入路径中，路径长度加
        path[len] = root->m_nValue;
```

```
        len++;
        if(root->m_nValue == n && root->m_pLeft == NULL
            && root->m_pRight == NULL){//叶子节点，并且节点值等于整数值，输出路
径
            for(int i = 0;i < len;i++)
                cout << path[i] << " ";
            cout << endl;
        }else{//非叶子节点
            int value = n - root->m_nValue;
            print_value_path_assist(root->m_pLeft,path,len,value);//输出左子树
路径
            print_value_path_assist(root->m_pRight,path,len,value);//输出右子树
路径
            len--;//回溯，找不经过此节点的路径，删除此节点，路径长度减，
        }
    }
}
//给定根节点和整数
//输出从根节点到叶子节点的所有路径中，路径经过的节点值之和等于给定整数的路径
void print_value_path(BSTreeNode  *root,int n){
    int path[30] = {0};
    int len = 0;//初始路径长度为
    print_value_path_assist(root,path,0,n);
}

void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}


int main(){
    BSTreeNode  *root = new BSTreeNode (10);
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (12));
    insert(root, new BSTreeNode (4));
    insert(root, new BSTreeNode (7));
    cout << "中序遍历的结果" << endl;
    inorder(root);
    cout << endl;
    int n = 22;
```

```
    cout << "all the path from root to leaf equal to " << n << " is " << endl;
    print_value_path(root,n);
    return 0;
}


/*给定一棵二叉树，判断是否是平衡二叉树
如下二叉排序树
      10
      / \
     5  12
    / \
   4   7
这是一棵平衡二叉树。
平衡二叉树定义：
左子树的深度和右子树的深度之差绝对值小于等于，
左子树是平衡树，有子树是平衡树

解法：首先判断，左子树的深度和右子树的深度
如果左子树深度和右子树深度的绝对值之差大于，则不是平衡二叉树，
如果小于，则继续判断左子树是否是平衡二叉树，右子树是否是平衡二叉树
    如果其中之一不是平衡二叉树，则这棵树不是平衡二叉树
    否则是平衡二叉树
*/

#include <iostream>
#include <cassert>
#include <cmath>
#include <queue>
#include <ctime>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
```

```
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//得到一棵树的深度
int depth(BSTreeNode  *root){
    if(root == NULL)
        return 0;
    int left = depth(root->m_pLeft);//左子树深度
    int right = depth(root->m_pRight);//右子树深度
    if(left > right)//左子树更深
        return left + 1;
    else
        return right + 1;
}
//判断是否是平衡二叉树
bool is_balance(BSTreeNode  *root){
    if(root == NULL)//空节点，是平衡树
        return true;
    int left = depth(root->m_pLeft);
    int right = depth(root->m_pRight);
    if(abs(left - right) > 1){//左右子树深度之差的绝对值大于，不是平衡树
        return false;
    }else{
```

```
        if(is_balance(root->m_pLeft) && is_balance(root->m_pRight))//左右子树都
是平衡树
            return true;
        else
            return false;
    }
}
//前序遍历
void preorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    cout << root->m_nValue << " ";
    preorder(root->m_pLeft);
    preorder(root->m_pRight);
}
int main(){
    BSTreeNode  *root = new BSTreeNode (10);
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (12));
    insert(root, new BSTreeNode (4));
    insert(root, new BSTreeNode (7));
    cout << "前序遍历的结果" << endl;
    preorder(root);
    cout << endl;
    cout << "the depth of the tree is " <<  depth(root) << endl;
    if(is_balance(root))
        cout << "The is balance " << endl;
    else
        cout << "The tree is not balance " << endl;
    return 0;
}
```

```
/*
判断整数序列是不是二元查找树的后序遍历结果
题目：输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果。
如果是返回true，否则返回false。
例如输入5 7 6 9 11 10 8，由于这一整数序列是如下树的后序遍历结果：
         8
       /  /
      6    10
     / /  / /
```

5 7 9 11
因此返回true。
如果输入7 4 6 5，没有哪棵树的后序遍历的结果是这个序列，因此返回false.

解题思路：
这题主要考察对后序遍历的理解，首先当元素少于或等于两个时，一定可以由后序遍历得到。

而在后序遍历中，最后一个节点是根节点。所以在整数序列[begin,end]中，
最后一个值就是根节点的值，设为end。

从整数序列第一个元素开始，找到第一个大于根节点的整数，设为index,
这个标示着右子树的开始那么在index后面的整数
是不允许小于end的，因为二叉排序树中，右子树中的元素都必须大于根节点。

而左子树的整数序列就是[begin,index-1],右子树的序列是[index,end-1],
左子树的整数序列和右子树整数序列也必须是由后序遍历得到的
*/

```
#include <iostream>
#include <cmath>
using namespace std;
bool is_postorder_assist(int a[], int begin, int end){
    if(abs(begin - end) <= 1)//当元素少于等于两个时，一定可以由后序遍历得到
        return true;
    int value = a[end];//根节点
    int index = begin;//用于标记第一个大于根节点的整数
    while(a[index] < value)
        index++;//找到第一个大于根节点的整数
    for(int i = index + 1;i < end;i++){//这个整数后面的整数不能有小于根节点的
        if(a[i] < a[end])
            return false;
    }
    //当左子树和右子树都能由后序遍历得到，返回true,否则返回false
    return is_postorder_assist(a, begin, index - 1) && is_postorder_assist(a, index
+ 1, end - 1);
}
//调用辅助函数，判断是否是后序遍历的结果
bool is_postorder(int a[],int n){
    return is_postorder_assist(a,0,n - 1);
}
int main(){
    int a[] = {5,7,6,9,11,10,8};
    //int a[] = {7,4,6,5};
    int n = sizeof(a) / sizeof(int);
```

```
    for(int i = 0;i < n;i++){
        cout << a[i] << " ";
    }
    cout << endl;
    if(is_postorder(a,n))
        cout << "can be postorder" << endl;
    else
        cout << "can't be postorder" << endl;
    return 0;
}
```

```
/*  层序遍历二叉树
    例如输入
   8
  / /
 6 10
/ / / /
5 7 9 11
输出8 6 10 5 7 9 11。
首先我们定义的二元查找树节点的数据结构如下：
 struct BSTreeNode
{
  int m_nValue; // value of node
  BSTreeNode *m_pLeft; // left child of node
  BSTreeNode *m_pRight; // right child of node
};

解题：一个广度遍历就行了，从根节点开始入队。之后将根节点出队，
根节点的非空左右孩子入队，再之后左右孩子出队，它们的非空左右孩子入队，
直到队列中没有元素
具体实现在void level_traversal(BSTreeNode  *root)中

如果题目要求按层次输出元素，即输出
```

8
6 10
5 7 9 11
这样用队列就不行了，因为你必须队列中的元素属于哪一层的，而队列无法保留这个信息。

这样我们必须自己维护层次信息
void print_level(BSTreeNode *root)中
我们用一个数组保存二叉树中的节点，从上到下，从左到右依次保存，

cur表示当前层第一个元素的位置，end表示当前层最后一个元素的下一个位置，

这样遍历当前层，将它们的非空孩子加入到数组中，遍历完这层后，cur就指向
下一层第一个元素的位置，而end要更新为下一层最后一个元素的下一个位置。

实现中用的是vector,因为它是动态扩张的，便于我们得到下一层最后一个元素的
下一个位置

*/
```
#include <iostream>
#include <cassert>
#include <cmath>
#include <queue>
#include <ctime>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
```

```
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}

void level_traversal(BSTreeNode  *root){//层次遍历二叉树
    if(root == NULL)
        return;
    queue<BSTreeNode  *> q;
    q.push(root);
    BSTreeNode  *temp = NULL;
    while(!q.empty()){
        temp = q.front();//得到队首元素
        q.pop();//将队首元素出队
        cout << temp->m_nValue << " ";
        if(temp->m_pLeft != NULL)//左子树非空，将左子树入队
            q.push(temp->m_pLeft);
        if(temp->m_pRight != NULL)//右子树非空，将右子树入队
            q.push(temp->m_pRight);
    }
    cout << endl;
}
void print_level(BSTreeNode  *root){
    if(root == NULL)
        return;
    vector<BSTreeNode  *> vec;
    vec.push_back(root);
    int end = 1;
    int cur = 0;//每一层的开始位置
    while(cur < (int)vec.size()){
```

```
        end = vec.size();//每一层的最后一个元素的下一个位置
        while(cur < end){
            cout << vec[cur]->m_nValue << " ";
            if(vec[cur]->m_pLeft != NULL)//左子树非空，将左子树加入
                vec.push_back(vec[cur]->m_pLeft);
            if(vec[cur]->m_pRight != NULL)//右子树非空，将右子树加入
                vec.push_back(vec[cur]->m_pRight);
            cur++;
        }
        cout << endl;
    }
}
int main(){
    BSTreeNode  *root = new BSTreeNode (8);
    insert(root, new BSTreeNode (6));
    insert(root, new BSTreeNode (10));
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (7));
    insert(root, new BSTreeNode (9));
    insert(root, new BSTreeNode (11));

    cout << "the level traversal is " << endl;
    level_traversal(root);
    cout << "print number by level is " << endl;
    print_level(root);
    return 0;
}
```

```
/*输入一颗二元查找树，完成树的镜像转换
即在转换后的二元查找树中，左子树的结点都大于右子树的结点。
用递归和循环两种方法完成树的镜像转换。
例如输入：
  8
  / /
  6 10
 // //
5 7 9 11
输出：
   8
  / /
```

```
 10 6
 // //
11 9 7 5
```
定义二元查找树的结点为：
```
struct BSTreeNode // a node in the binary search tree (BST)
{
  int m_nValue; // value of node
  BSTreeNode *m_pLeft; // left child of node
  BSTreeNode *m_pRight; // right child of node
};
```
解题：
递归方法：将根节点的左右孩子指针交换，也就是左孩子指针指向右子树，右孩子指针指向左子树
之后递归调用函数，将左子树和右子树也进行转化。

循环方法：用一个队列保存节点，从根节点往叶子节点，一层一层的交换节点的左右子树。
```
*/
#include <iostream>
#include <cassert>
#include <queue>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
```

```
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//求二叉树的镜像，递归方法
void mirror(BSTreeNode * &root){
    if(root == NULL)
        return;
    BSTreeNode *temp = root->m_pLeft;//交换左右子树
    root->m_pLeft = root->m_pRight;
    root->m_pRight = temp;
    mirror(root->m_pLeft);//求左子树的镜像
    mirror(root->m_pRight);//右子树的镜像
}
//求二叉树的镜像，非递归方法
void not_recursion_mirror(BSTreeNode * &root){
    if(root == NULL)
        return;
    queue<BSTreeNode *> s;//队列，用于保存节点
    s.push(root);//先将根节点入栈
    BSTreeNode *cur = NULL;
    BSTreeNode *temp = NULL;
    while(!s.empty()){
        cur = s.front();//得到栈顶元素
        s.pop();//删除栈顶元素
        BSTreeNode *temp = cur->m_pLeft;//交换左右子树
        cur->m_pLeft = cur->m_pRight;
        cur->m_pRight = temp;
        if(cur->m_pLeft != NULL)//左子树非空，入队
            s.push(cur->m_pLeft);
        if(cur->m_pRight != NULL)//右子树非空，入队
            s.push(cur->m_pRight);
    }
}
```

```
//中序遍历
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}
int main(){
    BSTreeNode  *root = new BSTreeNode (8);
    insert(root, new BSTreeNode (6));
    insert(root, new BSTreeNode (10));
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (7));
    insert(root, new BSTreeNode (9));
    insert(root, new BSTreeNode (11));
    cout << "inorder the tree is " << endl;
    inorder(root);
    cout << endl;
    cout << "after mirror is " << endl;
    mirror(root);//递归方法
    inorder(root);
    cout << endl;
    cout << "after mirror again is " << endl;
    not_recursion_mirror(root);//非递归方法
    inorder(root);
    cout << endl;
    return 0;
}
```

```
/*
    如下二叉排序树
      10
      / \
     5  12
    / \
   4   7
```

```
    前序遍历结果：10 5 4 7 12
    中序遍历结果：4 5 7 10 12
    后序遍历结果：4 7 5 12 10
     二叉排序树节点的数据结构定义为：
     struct BSTreeNode  // a node in the binary tree
     {
         int m_nValue; // value of node
         BSTreeNode  *m_pLeft; // left child of node
         BSTreeNode  *m_pRight; // right child of node
     };
这里是前序遍历，中序遍历，后序遍历的非递归实现。

前序遍历的非递归实现：
这个比较简单，先将根节点入栈。之后在栈非空的条件下，
将栈顶元素出栈，将非空右子树入栈，再将非空左子树入栈，

下次出栈的将是左子树，这样左子树继续上述过程，
直到左子树的每个元素都输出了，
再将右子树出栈，右子树继续上述过程。


中序遍历的非递归实现：
这个比前序遍历难。先从根节点出发，将沿途经过的节点入栈，
之后将栈顶元素出栈，输出它的值，并将它的右子树入栈，
右子树进行上述过程，


后序遍历的非递归实现：
这是三种遍历方式中最难的。与中序遍历时类似，
但后序遍历要先输出节点的右子树，之后才能输出节点

先从根节点出发，将沿途经过的节点入栈，之后得到栈顶元素，
判断它的右子树是否为空，
如果是空的则可以输出此节点，
如果非空，则将它的右子树
入栈，右子树进行上述过程。

这里要记录上一次输出的节点，
如果上一次输出的节点是现在节点的右孩
子，则可以输出现在的节点

*/
#include <iostream>
#include <cassert>
```

```
#include <stack>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//递归的前序遍历
void preorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    cout << root->m_nValue << " ";
```

```
    preorder(root->m_pLeft);
    preorder(root->m_pRight);
}
//非递归的前序遍历
void no_recursion_preorder(BSTreeNode  *root){
    if(root == NULL)
        return;
    stack<BSTreeNode  *> s;
    s.push(root);
    BSTreeNode  *current = NULL;
    while(!s.empty()){
        current = s.top();//得到栈顶第一个元素
        s.pop();//删除栈顶第一个元素
        cout << current->m_nValue << " ";
        if(current->m_pRight != NULL)//先将右子树入栈
            s.push(current->m_pRight);
        if(current->m_pLeft != NULL)//在将左子树入栈
            s.push(current->m_pLeft);
    }
}
//递归的中序遍历
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}
//非递归的中序遍历
void no_recursion_inorder(BSTreeNode  *root){
    if(root == NULL)
        return;
    stack<BSTreeNode  *> s;
    BSTreeNode  *current = root;
    while(!s.empty() || current != NULL){
        while(current != NULL){//往左子树走，将途中经过的节点入栈
            s.push(current);
            current = current->m_pLeft;
        }
        if(!s.empty()){
            current = s.top();//得到栈顶节点
            s.pop();//将栈顶的元素删除
            cout << current->m_nValue << " ";
            current = current->m_pRight;//往右子树走
```

```
        }
    }
}
//递归的后序遍历
void postorder(BSTreeNode  * root){
    if(root == NULL)
        return ;
    postorder(root->m_pLeft);
    postorder(root->m_pRight);
    cout << root->m_nValue << " ";
}
//非递归的后序遍历
void no_recursion_postorder(BSTreeNode  *root){
    if(root == NULL)
        return ;
    stack<BSTreeNode  *> s;
    BSTreeNode  *current = root;
    BSTreeNode *pre = NULL;//记录上一次输出的节点

    do{
        while(current != NULL){
            s.push(current);
            current = current->m_pLeft;
        }

        while(!s.empty()){
            current = s.top();
            //如果前一个输出节点等于当前节点的右孩子，则可以输出当前节点
            if(current->m_pRight == pre){
                cout << current->m_nValue << " ";
                pre = current;//更新输出的节点
                s.pop();//将这个元素出栈
            }else{//右孩子非空，或者上一次输出的节点是左孩子，当前节点不能输出
                if (current->m_pRight == NULL)
                    pre = NULL;

                current = current->m_pRight;//往右子树走
                break;
            }
        }
    }while(!s.empty());
}
int main(){
    BSTreeNode  *root = new BSTreeNode (10);
```

```
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (12));
    insert(root, new BSTreeNode (4));
    insert(root, new BSTreeNode (7));
    cout << "preorder is " << endl;
    preorder(root);
    cout << endl;
    cout << "preorder not using recursion" << endl;
    no_recursion_preorder(root);
    cout << endl;
    cout << "inorder is " << endl;
    inorder(root);
    cout << endl;
    cout << "inorder not using recursion" << endl;
    no_recursion_inorder(root);
    cout << endl;
    cout << "postorder is " << endl;
    postorder(root);
    cout << endl;
    cout << "postorder not using recursion" << endl;
    no_recursion_postorder(root);
    cout << endl;
    return 0;
}



/*给定有序数组order，用数组[begin,end]中的数构建一棵二叉排序树
用递归做会比较简单，先将order[(begin+end)/2]这个数加入到二叉树中，
这个节点就作为二叉排序树的根节点
那么根节点的左子树中的数字就是[begin,(begin+end)/2 - 1]中的数，
而右子树则是[(begin+end)/2 + 1,end]之间的数。
递归调用[begin,(begin+end)/2 - 1]后，返回的是左子树的根节点，
递归调用[begin,(begin+end)/2 - 1]后，返回的是右子树的根节点。
例如将[1,10]转化为而叉排序树，则会得到。。。不会画图，画的比较难看。
 前序遍历结果：5 2 1 3 4 8 6 7 9 10
 中序遍历结果：1 2 3 4 5 6 7 8 9 10
*/
#include <iostream>
#include <cassert>

using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
```

```
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//给定有序数组order，用数组[begin,end]中的数构建一棵二叉排序树
BSTreeNode  *array_to_tree(int order[], int begin, int end){
    if(begin > end)
        return NULL;
    int mid = (begin + end) / 2;
    BSTreeNode  *root = new BSTreeNode (order[mid]);//根节点
    root->m_pLeft = array_to_tree(order, begin, mid - 1);//左子树
    root->m_pRight = array_to_tree(order, mid + 1, end);//右子树
    return root;//返回根节点
}
//前序遍历
void preorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    cout << root->m_nValue << " ";
    preorder(root->m_pLeft);
    preorder(root->m_pRight);
}
//中序遍历
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}
int main(){
    int order[30];
    int begin = 1;
    int end = 10;
    for(int i = begin;i <= end;i++)
        order[i] = i;
    BSTreeNode  *root = array_to_tree(order,begin,end);
    cout << "preorder is " << endl;
    preorder(root);
    cout << endl;
```

```
    cout << "inorder is " << endl;
    inorder(root);
    cout << endl;
    return 0;
}
```

```
/*如下给定一棵二叉树的前序遍历和中序遍历结果，
    要求重建二叉树，并得到后序遍历。

   前序遍历结果：10 5 4 7 12
   中序遍历结果：4 5 7 10 12
   重建后的二叉树。
       10
      / \
     5  12
    / \
   4   7
   后序遍历结果：4 7 5 12 10
    二叉排序树节点的数据结构定义为：
    struct BSTreeNode  // a node in the binary tree
    {
        int m_nValue; // value of node
        BSTreeNode  *m_pLeft; // left child of node
        BSTreeNode  *m_pRight; // right child of node
    };

    解题思路：
    在前序遍历的结果中，第一个值就是根节点的值，在中序遍历中找到这个值的位置，
    设为index.
    则index的左边是左子树中序遍历的结果，index的右边是右子树中序遍历的结果，

    要得到左子树前序遍历的结果，可以在先求得左子树节点的个数，即中序遍历中
    index之前节点的个数，设为left,则左子树前序遍历的结果是
    从前序遍历结果第二个位置往后的left个节点。

    右子树前序遍历的结果同理可得。之后递归得到左子树和右子树的根节点，
    将当前的根节点的左右孩子指向左子树和右子树的根节点即可。

*/
#include <iostream>
#include <cassert>
```

```
#include <cmath>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//pre是前序遍历的结果，pb是前序遍历开始位置，pe是前序遍历结束位置，
//in是中序遍历的结果，ib是中序遍历开始位置,ie是中序遍历的结束位置
BSTreeNode *get_root(int pre[],int pb,int pe,int in[],int ib,int ie){
    int key = pre[pb];//根节点
    BSTreeNode *root = new BSTreeNode(key);
    int index;//标记根节点在中序遍历中的位置
    for(index = ib; index <= ie;index++){//找到根节点在中序遍历中的位置
        if(in[index] == key)
            break;
    }
    int left = index - ib;//左子树节点个数
    int right = ie - index;//右子树节点个数

    if(left > 0)//得到左子树根节点
        root->m_pLeft = get_root(pre,pb + 1,pb + left,in,ib,ib + left - 1);
    if(right > 0)
        //得到右子树根节点
        root->m_pRight = get_root(pre,pb + left + 1,pe,in,index + 1,ie);
    return root;
}
//根据前序遍历和中序遍历的结果，已经节点个数，重建二叉树
BSTreeNode *rebuild_tree(int pre[],int in[],int len){
    if(len <= 0)
        return NULL;
    return get_root(pre,0,len - 1,in,0,len - 1);
}
//前序遍历
void preorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    cout << root->m_nValue << " ";
    preorder(root->m_pLeft);
```

```
    preorder(root->m_pRight);
}
//中序遍历
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}
//后续遍历
void postorder(BSTreeNode  * root){
    if(root == NULL)
        return ;
    postorder(root->m_pLeft);
    postorder(root->m_pRight);
    cout << root->m_nValue << " ";
}


int main(){

    int pre[] = {10,5,4,7,12};
    int in[] = {4,5,7,10,12};
    int len = sizeof(pre) / sizeof(int);
    cout << "preorder is " << endl;
    for(int i = 0;i < len;i++)
        cout << pre[i] << " ";
    cout << endl;
    cout << "inorder is " << endl;
    for(int i = 0;i < len;i++)
        cout << in[i] << " ";
    cout << endl;
    cout << "rebuilding the tree " << endl;
    BSTreeNode *root = rebuild_tree(pre,in,len);
    cout << "postorder is " << endl;
    postorder(root);
    cout << endl;
    return 0;
}
```

```
/*
    在二叉排序树中输出所有路径（树）
    题目：输入一棵二叉排序树。
    从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。

    如下二叉排序树
      10
      / \
     5  12
    / \
   4   7
   则打印三条路径。
   10 5 4
   10 5 7
   10 12
    二叉排序树节点的数据结构定义为：
    struct BSTreeNode  // a node in the binary tree
    {
        int m_nValue; // value of node
        BSTreeNode  *m_pLeft; // left child of node
        BSTreeNode  *m_pRight; // right child of node
    };
    print_path(BSTreeNode  *root)主要是调用print_path_assist
    程序的主要实现在这个函数中
    void print_path_assist(BSTreeNode  *root,int path[],int len)
    数组path用于保存路径，
    len用于保存路径的长度，开始时len值为.
    基本步骤如下：
    用数组path保存经过的节点，首先将根节点加入路径中，然后判断根节点是否是叶子节点，
    如果是叶子节点，则输出路径，
    如果不是，则递归调用函数，输出左子树和右子树的路径
*/
#include <iostream>
#include <cassert>

using namespace std;
struct BSTreeNode {
```

```
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//辅助函数,输出从根节点到叶子节点的所有路径中
//数组path用于保存路径，len是路径的长度
void print_path_assist(BSTreeNode  *root,int path[],int len){
    if(root == NULL){
        return ;
    }else{//此节点非空，加入路径中，路径长度加
        path[len] = root->m_nValue;
        len++;
```

```
        if(root->m_pLeft == NULL
            && root->m_pRight == NULL){//叶子节点，并且节点值等于整数值，输出路
径
            for(int i = 0;i < len;i++)
                cout << path[i] << " ";
            cout << endl;
        }else{//非叶子节点
            print_path_assist(root->m_pLeft,path,len);//输出左子树路径
            print_path_assist(root->m_pRight,path,len);//输出右子树路径
            len--;//回溯，找不经过此节点的路径，删除此节点，路径长度减，
        }
    }
}
//输出从根节点到叶子节点的所有路径
void print_path(BSTreeNode  *root){
    int path[30] = {0};
    int len = 0;//初始路径长度为
    print_path_assist(root,path,0);
}
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}


int main(){
    BSTreeNode  *root = new BSTreeNode (10);
    insert(root, new BSTreeNode (5));
    insert(root, new BSTreeNode (12));
    insert(root, new BSTreeNode (4));
    insert(root, new BSTreeNode (7));
    cout << "中序遍历的结果" << endl;
    inorder(root);
    cout << endl;
    cout << "all the path from root to leaf is " << endl;
    print_path(root);
    return 0;
}
```

```
/*
输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。
要求不能创建任何新的结点，只调整指针的指向。
   10
  / /
  6  14
 / / / /
4  8 12 16
 转换成双向链表
4=6=8=10=12=14=16。

 首先我们定义的二元查找树节点的数据结构如下：
 struct BSTreeNode
{
  int m_nValue; // value of node
  BSTreeNode *m_pLeft; // left child of node
  BSTreeNode *m_pRight; // right child of node
};

解题思路：考虑到中序遍历的结果是4,6,8,10,12,14,16.
所以转化为双向链表可以用中序遍历的思想。
主要实现在这个函数中
void to_dlist_assist(BSTreeNode *root, BSTreeNode * &pre)
*/
#include <iostream>
#include <cstdio>
#include <cassert>
using namespace std;
struct BSTreeNode {
    int m_nValue;//节点的值
    BSTreeNode  *m_pLeft;//左孩子
    BSTreeNode  *m_pRight;//右孩子
    BSTreeNode (int value = 0){//构造函数，默认值为
        this->m_nValue = value;
        this->m_pLeft = NULL;
        this->m_pRight = NULL;
    }
};
```

```
//插入节点到二叉树中，如果插入成功，返回true,失败返回false
bool insert(BSTreeNode  * &root, BSTreeNode  * node){
    assert(node != NULL);
    if(root == NULL){//根节点为空，插入节点
        root = node;
        return true;
    }else{
        BSTreeNode  * parent = NULL;
        BSTreeNode  * current = root;
        while(current != NULL){
            if(current->m_nValue > node->m_nValue){//在左子树中找插入节点
                parent = current;
                current = current->m_pLeft;
            }else if(current->m_nValue < node->m_nValue){//在右子树中找插入节点
                parent = current;
                current = current->m_pRight;
            }else{//存在值相同的节点，插入失败
                return false;
            }
        }
        if(parent->m_nValue > node->m_nValue)//作为左孩子插入
            parent->m_pLeft = node;
        else//作为右孩子插入
            parent->m_pRight = node;
        return true;
    }
}
//中序遍历
void inorder(BSTreeNode  * root){
    if(root == NULL)
        return;
    inorder(root->m_pLeft);
    cout << root->m_nValue << " ";
    inorder(root->m_pRight);
}
//辅助函数，root是根节点，刚开始时pre存储中序遍历第一个节点的前一个节点
//运行这个函数后，pre存储中序遍历的最后一个节点
void to_dlist_assist(BSTreeNode *root, BSTreeNode * &pre){
    if(root == NULL)
        return;
    if(root->m_pLeft != NULL){
        to_dlist_assist(root->m_pLeft,pre);//运行后pre中存储左子树中序遍历的最
后一个节点
    }
```

```
    if(pre != NULL){
        pre->m_pRight = root;//左子树中序遍历的最后一个节点的下一个节点就是根节
点
    }
    root->m_pLeft = pre;
    pre = root;//右子树中序遍历第一个节点的前一个节点是根
    if(root->m_pRight != NULL){
        to_dlist_assist(root->m_pRight,pre);
    }
}
//将二叉树转化为双向链表，经过转化后，root指向双向链表的第一个节点
void to_dlist(BSTreeNode * &root){
    BSTreeNode *pre = NULL;//刚开始时中序遍历的第一个节点的前一个节点是NULL
    to_dlist_assist(root,pre);//运行这个函数后，pre存储中序遍历的最后一个节点
    pre->m_pRight = NULL;//中序遍历的最后一个节点的下一个节点是NULL
    BSTreeNode *head = root;
    while(head->m_pLeft != NULL)
        head = head->m_pLeft;//得到第一个节点
    root = head;
}


void print_dlist(BSTreeNode * head){
    if(head == NULL)
        return;
    BSTreeNode *cur = head;
    cout << "当前节点，前节点，后节点" << endl;
    while(cur != NULL){
        cout << cur->m_nValue << ",";
        if(cur->m_pLeft == NULL)
            cout << "NULL" << ",";
        else
            cout << cur->m_pLeft->m_nValue << ",";
        if(cur->m_pRight == NULL)
            cout << "NULL" << ",";
        else
            cout << cur->m_pRight->m_nValue << ",";
        cur = cur->m_pRight;
        cout << endl;
    }
}


int main(){
```

```
    BSTreeNode  *root = new BSTreeNode (10);
    insert(root, new BSTreeNode (6));
    insert(root, new BSTreeNode (14));
    insert(root, new BSTreeNode (4));
    insert(root, new BSTreeNode (8));
    insert(root, new BSTreeNode (12));
    insert(root, new BSTreeNode (16));
    cout << "中序遍历的结果" << endl;
    inorder(root);
    cout << endl;
    to_dlist(root);
    cout << "after change tree to double list is " << endl;
    print_dlist(root);
    return 0;
}
```