```
/*
二叉搜索树
    针对动态查找组织的数据结构，方便插入和删除
    非空左子树所有结点的值都小于其根结点
    非空右子树所有结点的值都大于其根结点

平衡二叉树(AVL树)
    搜索树结点不同的插入次序，将导致不同的深度和平均查找长度
    平衡因子：定义为左右两个子树的高度差，左 - 右
    平衡二叉树：任一结点平衡因子小于等于 1 的非空二叉搜索树
    给定结点数为 n 的平衡二叉树的最大高度为 log2(n)

平衡二叉树的调整
    RR旋转：麻烦结点在被破坏结点右子树的右子树上
    LL旋转：麻烦节点在被破坏结点左子树的左子树上，调整的时候考虑最下方的被破坏结点
    LR旋转：基本同上
    RL旋转：基本同上
*/

#include<iostream>
#include<malloc.h>
using namespace std;
typedef int ElementType;
typedef struct TreeNode *BinTree;
struct TreeNode{
    ElementType Data;
    BinTree Left;
    BinTree Right;
};

// 在二叉搜索树种插入结点
BinTree Insert(ElementType X, BinTree BST)
{
    if(!BST)
    {   // 如果当前结点为空，对其初始化
        BST = (BinTree)malloc(sizeof(struct TreeNode));
        BST->Data = X;
        BST->Left = NULL;
        BST->Right = NULL;
    }
    else
    {   // 不为空
        if(X < BST->Data)        // 如果小，挂在左边
            BST->Left = Insert(X, BST->Left);
        else if(X > BST->Data)  // 如果大，挂在右边
            BST->Right = Insert(X, BST->Right);
        // 如果相等，什么都不用做
    }
    return BST;
}

// 查找递归实现
BinTree Find(ElementType X, BinTree BST)
{
    if(!BST)           // 如果根结点为空，返回 NULL
        return NULL;
    if(X < BST->Data) // 比根结点小，去左子树查找
        return Find(X, BST->Left);
    else if(X > BST->Data)  // 比根结点大，去右子树查找
        return Find(X,BST->Right);
    else if(X == BST->Data) // 找到了
        return BST;
}

// 查找非递归实现
BinTree IterFind(ElementType X, BinTree BST)
{
    while(BST)
    {
        if(X < BST->Data)
            BST = BST->Left;
        else if(X > BST->Data)  // 比根结点大，去右子树查找
            BST = BST->Right;
```

```
74          else if(X == BST->Data) // 找到了
75              return BST;
76      }
77      return NULL;
78  }
79
80  // 查找最小值的递归实现
81  BinTree FindMin(BinTree BST)
82  {
83      if(!BST) // 如果为空了，返回 NULL
84          return NULL;
85      else if(BST->Left) // 还存在左子树，沿左分支继续查找
86          return FindMin(BST->Left);
87      else // 找到了
88          return BST;
89  }
90
91  // 查找最大值的非递归实现
92  BinTree FindMax(BinTree BST)
93  {
94      if(BST) // 如果不空
95          while(BST->Right) // 只要右子树还存在
96              BST = BST->Right;
97      return BST;
98  }
99
100 // 删除
101 BinTree Delete(ElementType X, BinTree BST)
102 {
103     BinTree tmp;
104     if(!BST)
105         cout<<"要删除的元素未找到";
106     else if(X < BST->Data) // X 比当前结点值小，在左子树继续查找删除
107         BST->Left = Delete(X, BST->Left);
108     else if(X > BST->Data) // x 比当前结点值大，在右子树继续查找删除
109         BST->Right = Delete(X, BST->Right);
110     else
111     {   // 找到被删除结点
112         if(BST->Left && BST->Right)
113         {   // 被删除结点有两个孩子结点
114             tmp = FindMin(BST->Right); // 找到右子树中值最小的
115             BST->Data = tmp->Data;     // 用找到的值覆盖当前结点
116             BST->Right = Delete(tmp->Data, BST->Right); //
                把前面找到的右子树最小值结点删除
117         }
118         else
119         {   // 被删除结点只有一个孩子结点或没有孩子结点
120             tmp = BST;
121             if(!BST->Left && !BST->Right) // 没有孩子结点
122                 BST = NULL;
123             else if(BST->Left && !BST->Right) // 只有左孩子结点
124                 BST = BST->Left;
125             else if(!BST->Left && BST->Right) // 只有右孩子结点
126                 BST = BST->Right;
127         }
128         free(tmp);
129     }
130     return BST;
131 }
132
133 // 中序遍历
134 void  InOrderTraversal(BinTree BT)
135 {
136     if(BT)
137     {
138         InOrderTraversal(BT->Left); // 进入左子树
139         cout<<BT->Data; // 打印根
140         InOrderTraversal(BT->Right); // 进入右子树
141     }
142 }
143
144 int main()
145 {
```

```
    BinTree BST = NULL;
    BST = Insert(5, BST);
    BST = Insert(7, BST);
    BST = Insert(3, BST);
    BST = Insert(1, BST);
    BST = Insert(2, BST);
    BST = Insert(4, BST);
    BST = Insert(6, BST);
    BST = Insert(8, BST);
    BST = Insert(9, BST);
    /*
                5
               /\
              3  7
             /\  /\
            1 4 6  8
            \       \
             2       9
    */
    cout<<"中序遍历的结果是：";
    InOrderTraversal(BST);
    cout<<endl;
    cout<<"查找最小值是："<<FindMin(BST)->Data<<endl;
    cout<<"查找最大值是："<<FindMax(BST)->Data<<endl;
    cout<<"查找值为3的结点左子树结点值为："<<Find(3, BST)->Left->Data<<endl;
    cout<<"查找值为7的结点右子树结点值为："<<IterFind(7, BST)->Right->Data<<endl;
    cout<<"删除值为5的结点"<<endl;
    Delete(5, BST);
    /*
                6
               /\
              3  7
             /\   \
            1 4    8
            \       \
             2       9
    */
    cout<<"中序遍历的结果是：";
    InOrderTraversal(BST);
    cout<<endl;
    return 0;
}
```