将二叉查找树类中的插入、删除和查找函数改成非递归函数。

【解】插入操作的实现见代码清单 8-13。先为被插入的元素申请一个结点（第 3 行）。最简单的情况是当前的树为空，那么插入元素被作为根结点（第 5 行）。第 7 行开始考虑一般的情况。从根结点开始往下寻找。如果 x 大于当前结点，x 应该插入到右子树上，所以检查当前结点有没有右孩子。如果有右孩子，则将当前结点设为右孩子，继续寻找，否则 x 插入为当前结点的右孩子。如果 x 不大于当前结点，x 应该插入到左子树上，所以检查当前结点有没有左孩子。如果有左孩子，则将当前结点设为左孩子，继续寻找，否则 x 插入为当前结点的左孩子。

代码清单 8-13 插入操作的非递归实现

```
1.  template <class Type>
2.  void BinarySearchTree<Type>::insert(const Type &x)
3.  {   BinaryNode* nd = new BinaryNode(x, NULL, NULL); // 为插入元素申请结点
4.
5.      if (root == NULL) { root = nd; return; }      // 在空树上插入
6.
7.      BinaryNode* now = root;
8.      while (true) {                  // 寻找插入位置并插入
9.          if (x > now->data){
10.             if( now->right == NULL ){
11.                 now->right = nd;
12.                 break;
13.             }
14.             now = now->right;
15.         }
16.         else if ( now->left == NULL ) {
17.             now->left = nd;
18.             break;
19.         }
20.             else now = now->left;
21.     }
22. }
```

查找操作是最简单的一个操作，它的实现见代码清单 8-14。它也是从根结点按层往下查找的过程。如果被查找的元素 x 大于当前结点，x 应该在右子树上，将当前结点设为右孩子，继续寻找。如果 x 小于当前结点，x 应该在左子树上，将当前结点设为左孩子，继续寻找。如果 x 等于当前结点，返回 true 表示找到。如果在寻找的过程中发现当前结点是空结点，表示 x 不存在，返回 false。

代码清单 8-14 查找操作的非递归实现

```
1.  template <class Type>
2.  bool BinarySearchTree<Type>::find(const Type &x) const
3.  {   BinaryNode* now = root;
```

```
4.          while(now != NULL){
5.              if (x > now->data) now = now->right;
6.              else if (x < now->data) now = now->left;
7.              else return true;
8.          }
9.          return false;
10.  }
```

删除是最复杂的一个操作，它的实现见代码清单 8-15。与其他两个操作一样，删除操作也必须先找到要删除的位置。第 7 到 19 行的 while 循环就是完成这个功能。从这个循环退出时，now 指向被删除的结点，now 的父结点是 parent。如果 now 是 parent 的左孩子，leftOrRight 等于 1。如果是右孩子，leftOrRight 等于 2。

代码清代 8-15 删除操作的非递归实现

```
1.   template <class Type>
2.   void BinarySearchTree<Type>::remove(const Type &x)
3.   {    BinaryNode *now = root;              // 当前正在检查的结点
4.        BinaryNode *parent;                  // now 的父结点
5.        int leftOrRight;              // now 是父亲的左孩子为 1，右孩子为 2
6.
7.        while(now != NULL) {                 // 寻找被删结点
8.            if (x > now->data) {
9.               leftOrRight = 2;
10.                parent= now;
11.                now = now->right;
12.               }
13.           else if (x < now->data) {
14.                leftOrRight = 1;
15.                     parent = now;
16.                now = now->left;
17.               }
18.           else break;
19.       }
20.
21.       // 删除 now
22.       if (now->left != NULL && now->right != NULL) {       //有两个孩子
23.           BinaryNode* tmp = now->right;
24.           if (tmp->left == NULL) {                 // 右儿子作为替身
25.               now->data = tmp->data;
26.               now->right = tmp->right;
27.               delete tmp;
28.               return;
29.           }
30.         while(tmp->left->left != NULL) tmp = tmp->left;  // 找右子树的最小值
```

```
31.         BinaryNode *deleted = tmp->left;
32.             now->data = deleted->data;
33.           tmp->left = deleted->right;
34.           delete deleted;
35.           return;
36.         }
37.
38.         if (now == root) {                          // 删除根结点
39.             if (root->left == NULL) root = root->right; else root = root->left;
40.             delete now;
41.             return;
42.         }
43.
44.         if (leftOrRight == 1)               // 删除只有一个或没有孩子的结点
45.          parent->left = (now->left == NULL ? now->right : now->left);
46.           else parent->right = (now->left == NULL ? now->right : now->left);
47.         delete now;
48.         return;
49.  }
```

从 21 行开始删除 now。第 22 到 35 行处理的是 now 有两个儿子的情况。当 now 有两个儿子时，首先在右子树上找最小元素。如果 now 的右儿子没有左儿子，那么 now 的右儿子就是右子树上的最小元素，将右儿子作为替身，删除右儿子（第 25 到 28 行）。如果 now 的右儿子有左儿子，则找到它最左边的儿子的父亲（第 30 行）。将最左边的儿子作为替身，删除替身结点（第 31 到 36 行）

第 38 行开始处理 now 有一个儿子或没有儿子的情况。先考虑 now 是根结点，此时可删除 now，并将它的非空的左儿子或右儿子作为树根（第 38 到 43 行）。如果 now 不是根结点，则必须将它的父亲和它的非空儿子连起来。于是判它是父亲的左儿子还是右儿子。如果是左儿子，将父亲的左指针指向 now 的非空儿子，否则将父亲的右指针指向 now 的非空儿子。最后删除 now，操作结束。