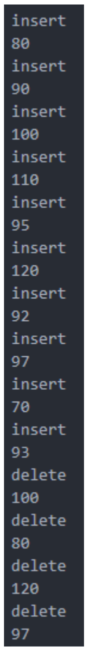
| | |
|---|---|
| Author 1: | Jonathan Mabery |
| Email 1: | jmabery@uco.edu |
| Author 2: | Andrew Aprile |
| Email 2: | aaprile@uco.edu |
| Course: | ADS. CMSC3613. CRN10040 |
| Assignment | Project 4 |

**Give a brief discussion of your implementation:**
For rotate_right() and left_balance(), we took the premade rotate_left() and right_balance() and switched the left and right. For the right_balance() and left_balance(), we changed the return type from void to bool so that it could be better used by the avl_delte() function.
For the avl delete function, we recursively go through the left branch and right branch until the target is found. This uses the same logic as the search_and_delete function in the binary search tree.
When the node is found, there are three cases to take care of

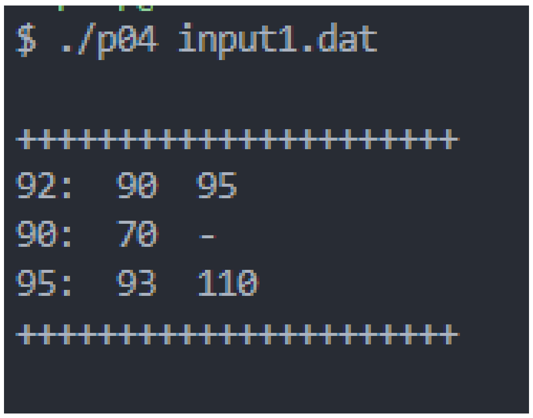
1. No left child. Replace the deleted node with the right subtree
2. No right child. Replace deleted node with left subtree
3. Two children. Replace deleted node with inorder successor.

The flag "shorter" is used to indicate if the height requires a rebalancing.
By using this avl_delete function, we keep the delete functionality to a O(log-n) time complexity

A screenshot of a test run.
Input 1. Output 1. Input 2. Output 2

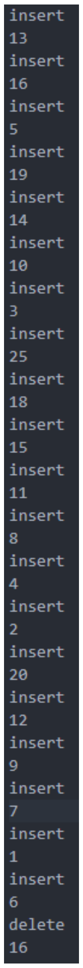


```
insert
80
insert
90
insert
100
insert
110
insert
95
insert
120
insert
92
insert
97
insert
70
insert
93
delete
100
delete
80
delete
120
delete
97
```

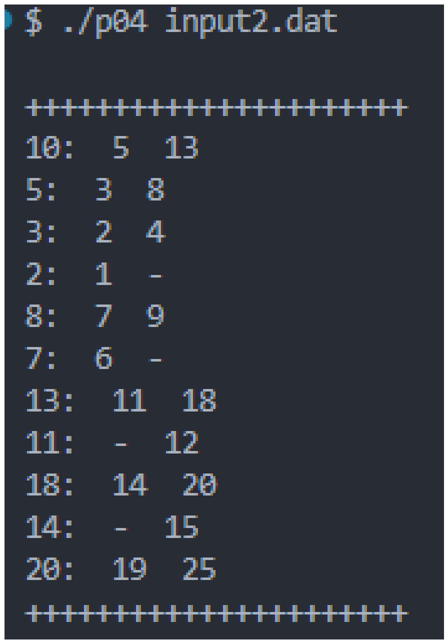


```
$ ./p04 input1.dat

++++++++++++++++++++++
92:  90  95
90:  70  -
95:  93  110
++++++++++++++++++++++
```



```
insert
13
insert
16
insert
5
insert
19
insert
14
insert
10
insert
3
insert
25
insert
18
insert
15
insert
11
insert
8
insert
4
insert
2
insert
20
insert
12
insert
9
insert
7
insert
1
insert
6
delete
16
```



```
$ ./p04 input2.dat

++++++++++++++++++++++
10:  5  13
5:  3  8
3:  2  4
2:  1  -
8:  7  9
7:  6  -
13:  11  18
11:  -  12
18:  14  20
14:  -  15
20:  19  25
++++++++++++++++++++++
```

# Source Code

Rotate Right

```
template <class Record>
void AVL_tree<Record>::rotate_right(Binary_node<Record> *&sub_root)
/*
Pre:  sub_root points to a subtree of the AVL_tree.
      This subtree has a nonempty left subtree.
Post: sub_root is reset to point to its former left child, and the former
      sub_root node is the right child of the new sub_root node.
*/
{
   // TODO 1
   if (sub_root == NULL || sub_root->left == NULL)
   {
      cout << "WARNING: program error detected in rotate_right" << endl;
   }
   else
   {
      Binary_node<Record> *pivot = sub_root->left;
      sub_root->left = pivot->right;
      pivot->right = sub_root;
      sub_root = pivot;
   }
}
```

Left Balance

```

template <class Record>
bool AVL_tree<Record>::left_balance(Binary_node<Record> *&sub_root)
/*
Pre:  sub_root points to a subtree of an AVL_tree that
      is doubly unbalanced on the left.
Post: The AVL properties have been restored to the subtree.
*/
{
   // TODO 2
   Binary_node<Record> *&left_tree = sub_root->left;

   switch (left_tree->get_balance())
   {

   // single right rotation
   case left_higher:
      sub_root->set_balance(equal_height);
      left_tree->set_balance(equal_height);
      rotate_right(sub_root); // pointer adjustment
      return true;

   case equal_height:
      // This is a valid case for deletion, not an error
      sub_root->set_balance(left_higher);
      left_tree->set_balance(right_higher);
      rotate_right(sub_root);
      return false;

   case right_higher:
      Binary_node<Record> *sub_tree = left_tree->right;

      switch (sub_tree->get_balance())
      {
      case equal_height:
         sub_root->set_balance(equal_height);
         left_tree->set_balance(equal_height);
         break;

      case right_higher:
         sub_root->set_balance(equal_height);
         left_tree->set_balance(left_higher);
         break;

      case left_higher:
         sub_root->set_balance(right_higher);
         left_tree->set_balance(equal_height);
         break;
      }

      // set balance of sub_tree after rotation
      sub_tree->set_balance(equal_height);

      // perform actual rotation
      rotate_left(left_tree);
      rotate_right(sub_root);
      return true;
   }
   return true;
}
```

Avl Delete

```
template <class Record>
Error_code AVL_tree<Record>::avl_delete(Binary_node<Record> *&sub_root,
                                        const Record &target, bool &shorter)
{
   Error_code result = success;

   // TODO 3
   // if node is not found : base case
   if (sub_root == nullptr)
   {
      shorter = false;
      return result = not_present;
   }

   // 1. Recursive search for target
   // left subtree case
   if (target < sub_root->data)
   {
      result = avl_delete(sub_root->left, target, shorter);

      if (shorter) // Left side got shorter, check for rebalance
      {
         switch (sub_root->get_balance())
         {
         case left_higher: // Was left-heavy, now balanced
            sub_root->set_balance(equal_height);
            // shorter stays true
            break;
         case equal_height: // Was balanced, now right-heavy
            sub_root->set_balance(right_higher);
            shorter = false; // Height change absorbed
            break;
         case right_higher:                    // Was right-heavy, now unbalanced
            shorter = right_balance(sub_root); // Call rebalance
            break;
         }
      }
   }

   // right subtree case
   else if (target > sub_root->data)
   {
      result = avl_delete(sub_root->right, target, shorter);
      if (shorter) // Right side got shorter, check for rebalance
      {
         switch (sub_root->get_balance())
         {
         case right_higher: // Was right-heavy, now balanced
            sub_root->set_balance(equal_height);
            // shorter stays true
            break;
      case equal_height: // Was balanced, now left-heavy
         sub_root->set_balance(left_higher);
         shorter = false; // Height change absorbed
         break;
      case left_higher:                    // Was left-heavy, now unbalanced
         shorter = left_balance(sub_root); // Call rebalance
         break;
      }
   }
}

// 2. Target found: Perform deletion
else
{
   // declare temporary variable that allows for deletion
   Binary_node<Record> *to_delete = sub_root;
   shorter = true; // deletion causes height to potentially be shorter

   // case 1: no left child
   if (sub_root->left == nullptr)
   {
      // since left is null we want try to assign sub_root to sub_root right, and if it is
      // also nullptr, it will go to the else if case below
      sub_root = sub_root->right;
      delete to_delete;
      return result = success;
   }

   // case 2 : no right child
   else if (sub_root->right == nullptr)
   {
      sub_root = sub_root->left;
      delete to_delete;
      return result = success;
   }

   else // case 3 : two children
   {
      // find inorder successor node
      Binary_node<Record> *successor = sub_root->right;
      while (successor->left != nullptr)
      {
         successor = successor->left;
      }

      // copy the successor's data to the current node
      sub_root->data = successor->data;

      // recursively delete the successor from the right subtree
```

```
         result = avl_delete(sub_root->right, successor->data, shorter);

         // if the tree needs rebalancing
         if (shorter)
         {
            switch (sub_root->get_balance())
            {
            case right_higher: // Was right-heavy, now balanced
               sub_root->set_balance(equal_height);
               // shorter stays true
               break;
            case equal_height: // Was balanced, now left-heavy
               sub_root->set_balance(left_higher);
               shorter = false; // Height change absorbed
               break;
            case left_higher:                    // Was left-heavy, now unbalanced
               shorter = left_balance(sub_root); // Call rebalance
               break;
            }
         }
      }
   }
   return result;
}
```