CHAPTER 26

AVL Trees and Splay Trees

Objectives

- To describe what an AVL tree is (§26.1).
- To rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§26.2).
- To design the **`AVLTree`** class (§26.3).
- To insert elements into an AVL tree (§26.4).
- To implement node rebalancing (§26.5).
- To delete elements from an AVL tree (§26.6).
- To implement the **`AVLTree`** class (§26.7).
- To test the **`AVLTree`** class (§26.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§26.9).
- To know what a splay tree is and how to insert and delete elements in a splay tree (§26.10).

26.1 Introduction

Key Point: *AVL Tree is a balanced binary search tree.*

Chapter 21 introduced binary trees. The search, insertion, and deletion time for a binary tree depends on the height of the tree. In the worst case, the height is $O(n)$. If a tree is *perfectly balanced*, i.e., it is a complete binary tree, its height is $\log n$. Can we maintain a perfectly balanced tree? Yes. But it will be costly to do so. The compromise is to maintain a well-balanced tree—i.e., the heights of two subtrees for every node are about the same.

AVL trees are well balanced. AVL trees were invented by two Russian computer scientists, G. M. Adelson-Velsky and E. M. Landis, in 1962. In an AVL tree, the difference between the heights of two subtrees for every node is `0` or `1`. It can be shown that the maximum height of an AVL tree is $O(\log n)$.

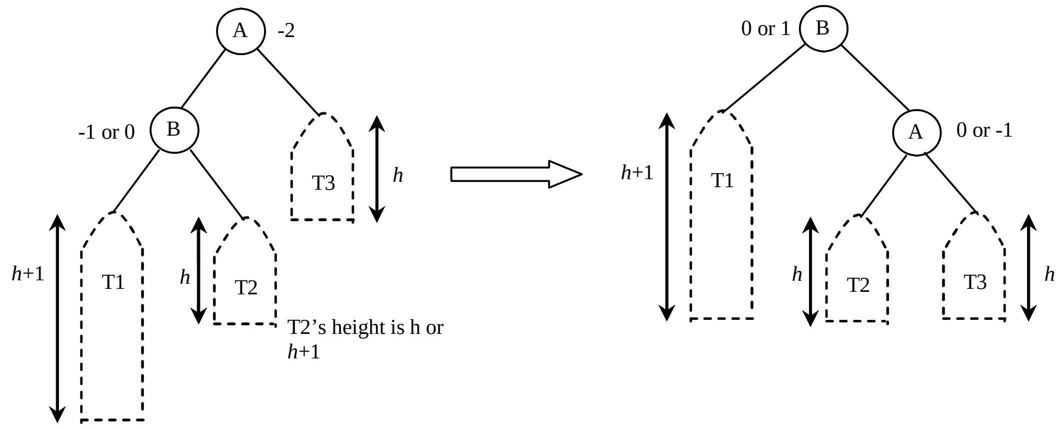
The process for inserting or deleting an element in an AVL tree is the same as for a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. A node is said to be *balanced* if its balance factor is `-1`, `0`, or `1`. A node is said to be *left-heavy* if its balance factor is `-1`. A node is said to be *right-heavy* if its balance factor is `+1`.

26.2 Rebalancing Trees

Key Point: *After inserting or deleting an element from an AVL tree, if the tree becomes unbalanced, perform a rotation operation to rebalance the tree.*

If a node is not balanced after an insertion or deletion operation; you need to rebalance it. The process of rebalancing a node is called a *rotation*. There are four possible rotations.

> **LL Rotation**: An *LL imbalance* occurs at a node `A` such that `A` has a balance factor `-2` and a left child `B` with a balance factor `-1` or `0`, as shown in Figure 26.1a. This type of imbalance can be fixed by performing a single right rotation at `A`, as shown in Figure 26.1b.

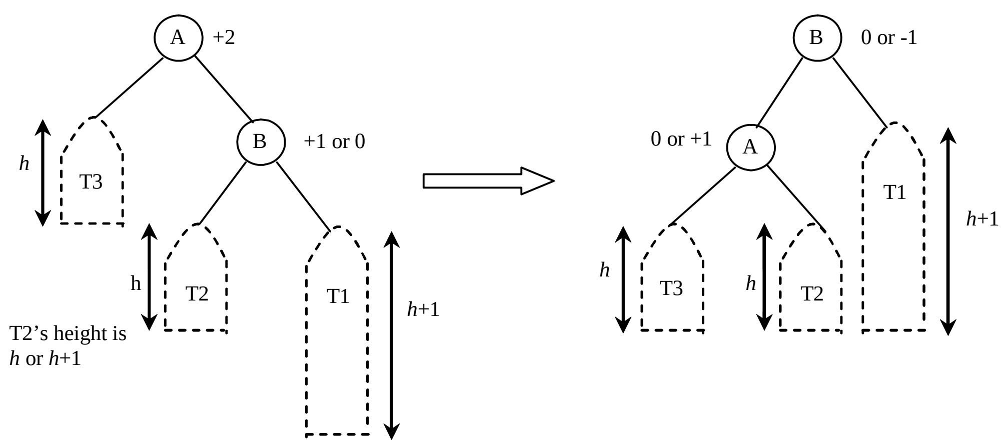


(a) **(b)**

Figure 26.1

LL rotation fixes LL imbalance.

RR Rotation: An *RR imbalance* occurs at a node **A** such that **A** has a balance factor **+2** and a right child **B** with a balance factor **+1** or **0**, as shown in Figure 26.2a. This type of imbalance can be fixed by performing a single left rotation at **A**, as shown in Figure 26.2b.

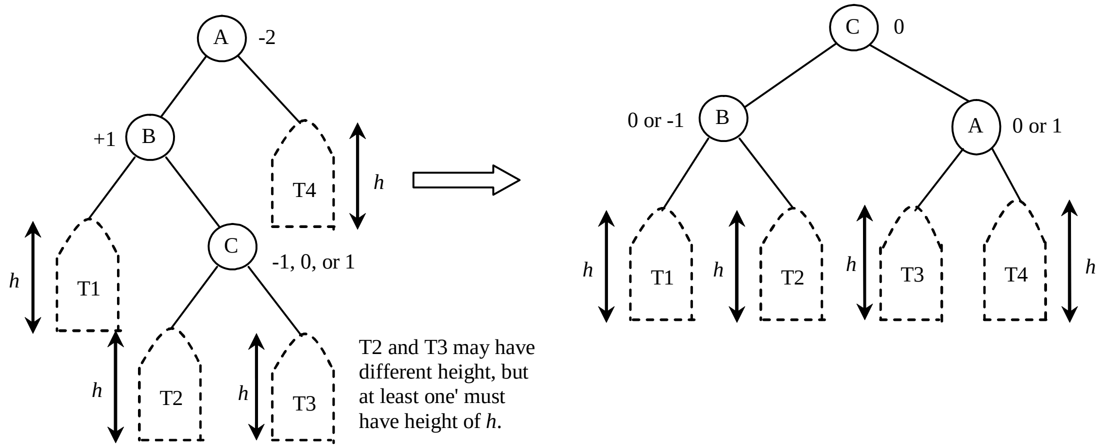


(a) **(b)**

Figure 26.2

RR rotation fixes RR imbalance.

LR Rotation: An *LR imbalance* occurs at a node **A** such that **A** has a balance factor **-2** and a left child **B** with a balance factor **+1**, as shown in Figure 26.3a. Assume **B**'s right child is **C**. This type of imbalance can be fixed by performing a double rotation at **A** (first a single left rotation at **B** and then a single right rotation at **A**), as shown in Figure 26.3b.

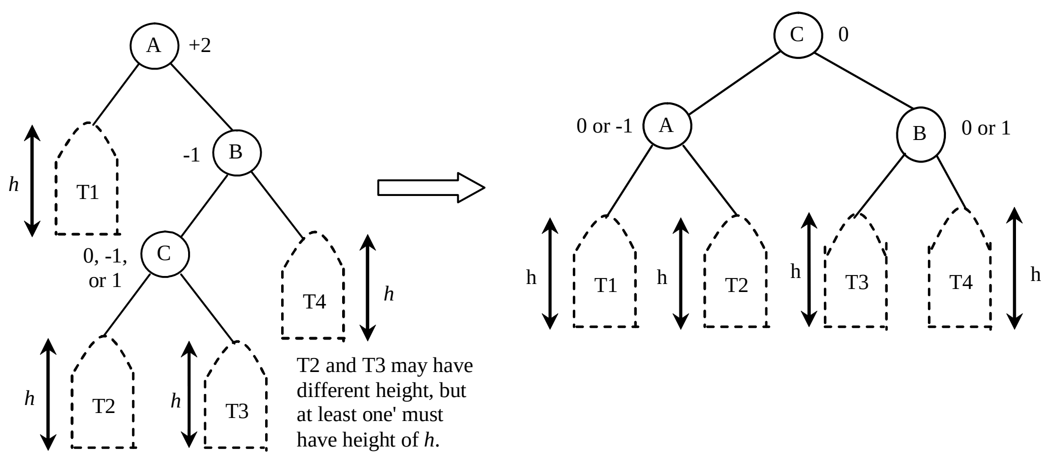


(a) **(b)**

Figure 26.3

LR rotation fixes LR imbalance.

RL Rotation: An *RL imbalance* occurs at a node **A** such that **A** has a balance factor **+2** and a right child **B** with a balance factor **-1**, as shown in Figure 26.4a. Assume **B**'s left child is **C**. This type of imbalance can be fixed by performing a double rotation at **A** (first a single right rotation at **B** and then a single left rotation at **A**), as shown in Figure 26.4b.



(a) **(b)**

Figure 26.4

RL rotation fixes RL imbalance.

Check point

26.1 What is an AVL tree? Describe the terms balance factor, left-heavy, and right-heavy.

26.2 Describe LL rotation, RR rotation, LR rotation, and RL rotation for an AVL tree.

26.3 Designing Classes for AVL Trees

Key Point: Since an AVL tree is a binary search tree, **AVLTree** is designed as a subclass of **BST**.

An AVL tree is a binary tree. So you can define the **`AVLTree`** class to extend the **`BST`** class, as shown in Figure 26.5. The **`BST`** and **`TreeNode`** classes are defined in §21.2.6.

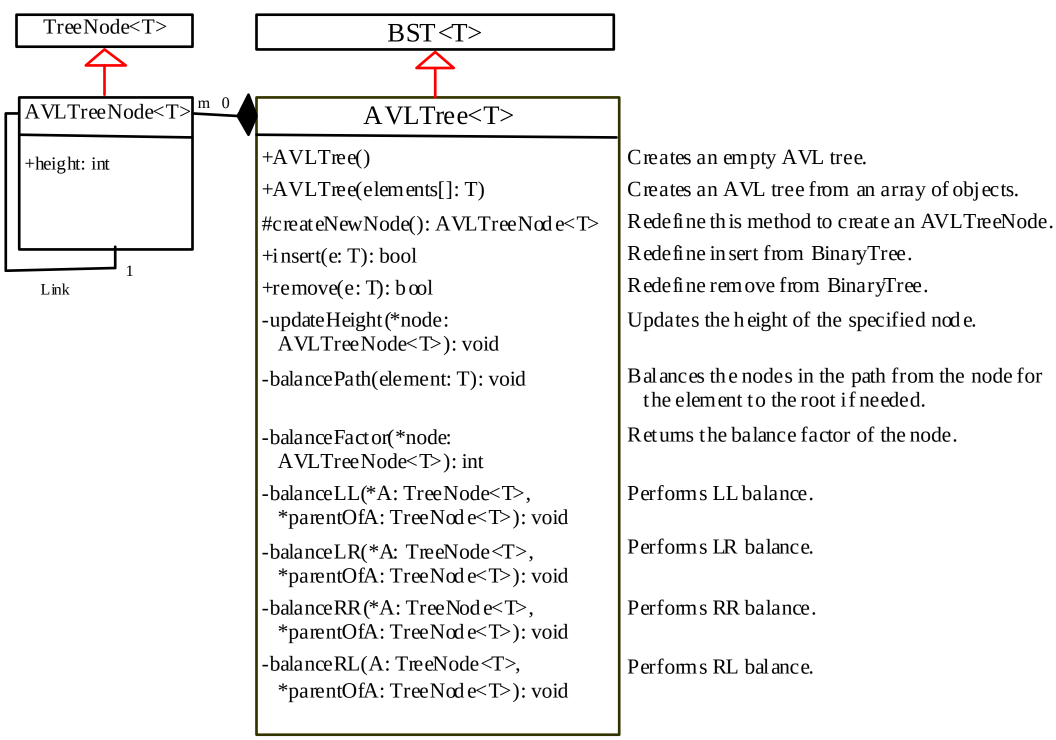


Figure 26.5

The `AVLTree` class extends `BST` with new implementations for the `insert` and `remove` functions.

In order to balance the tree, you need to know each node's height. For convenience, store the height of each node in `AVLTreeNode` and define `AVLTreeNode` to be a subclass of `TreeNode`, defined in lines 8-22 in Listing 21.3. Note that `TreeNode` contains the data fields `element`, `left`, and `right`, which are inherited in `AVLTreeNode`. So, `AVATreeNode` contains four data fields, as pictured in Figure 26.6.

node: AVLTreeNode<T>
#element: T #height: int #left: TreeNode<T> #right: TreeNode<E>

Figure 26.6

An `AVLTreeNode` contains protected data fields `element`, `height`, `left`, and `right`.

In the `BST` class, the `createNewNode()` function creates a `TreeNode` object. This function is overridden in the `AVLTree` class to create an `AVLTreeNode`. Note that the return type of the `createNewNode()` function in the `BianryTree` class is `TreeNode`, but the return type of the `createNewNode()` function in `AVLTree` class is `AVLTreeNode`. This is fine, since `AVLTreeNode` is a subtype of `TreeNode`.

Searching an element in an AVL tree is the same as searching in a regular binary tree. So, the `search` function defined in the `BST` class also works for `AVLTree`.

The `insert` and `remove` functions are overridden to insert and delete an element and perform rebalancing operations if necessary to ensure that the tree is balanced.

Pedagogical NOTE

Run from www.cs.armstrong.edu/liang/animation/AVLTreeAnimation.html to see how an AVL tree works, as shown in Figure 26.7.

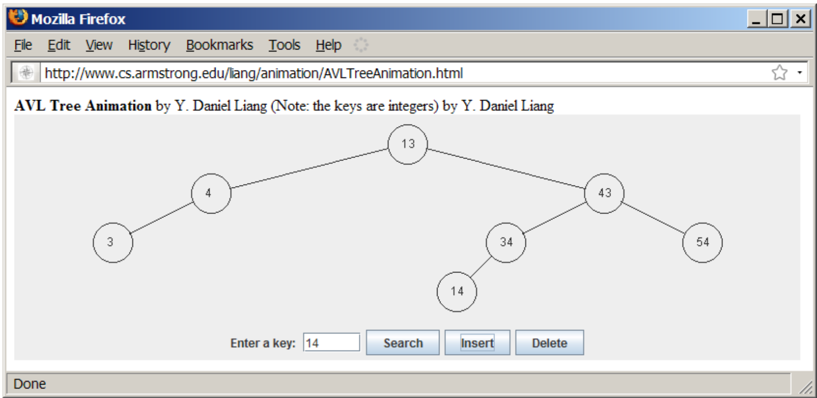


Figure 26.7

The animation tool enables you to insert, delete, and search elements visually.

26.4 Overriding the `insert` Function

Key Point: *Inserting an element into an AVL tree is the same as inserting it to a BST, except that the tree may need to be rebalanced.*

A new element is always inserted as a leaf node. As a result of adding a new node, the heights of the ancestors of the new node may increase. After insertion, check the nodes along the path from the new leaf node up to the root. If a node is found unbalanced, perform an appropriate rotation using the following algorithm:

Listing 26.1 Balancing Nodes on a Path

```
balancePath(T e)
{
  Get the path from the node that contains element e to the root,
    as illustrated in Figure 26.8;
  for each node A in the path leading to the root
  {
    Update the height of A;
    Let parentOfA denote the parent of A,
      which is the next node in the path, or NULL if A is the root;

    switch (balanceFactor(A))
    {
      case -2: if balanceFactor(A.left) = -1 or 0
                 Perform LL rotation; // See Figure 26.1
               else
```

```
                  Perform LR rotation; // See Figure 26.3
                break;
       case +2: if balanceFactor(A.right) = +1 or 0
                  Perform RR rotation; // See Figure 26.2
                else
                  Perform RL rotation; // See Figure 26.4
      } // End of switch
    } // End of for
  } // End of function
```

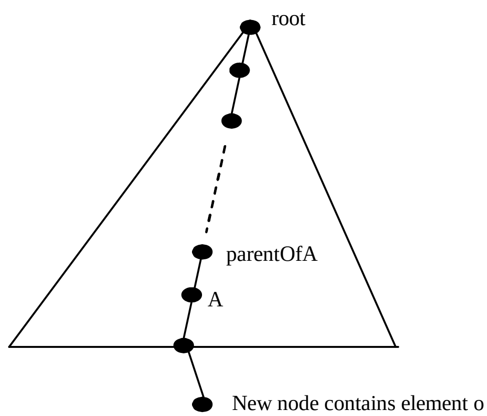


Figure 26.8

The nodes along the path from the new leaf node may become unbalanced.

The algorithm considers each node in the path from the new leaf node to the root. Update the height of the node on the path. If a node is balanced, no action is needed. If a node is not balanced, perform an appropriate rotation.

26.5 Implementing Rotations

Key Point: *An unbalanced tree becomes balanced by performing an appropriate rotation operation.*

Section 26.2, “Rebalancing Tree,” illustrated how to perform rotations at a node. Listing 26.2 gives the algorithm for the LL rotation, as pictured in Figure 26.1.

Listing 26.2 LL Rotation Algorithm

```
balanceLL(TreeNode A, TreeNode parentOfA) {
   Let B be the left child of A.

```

```
    if (A is the root)
      Let B be the new root
    else {
      if (A is a left child of parentOfA)
        Let B be a left child of parentOfA;
      else
        Let B be a right child of parentOfA;
    }

    Make T2 the left subtree of A by assigning B.right to A.left;
    Make A the right child of B by assigning A to B.right;
    Update the height of node A and node B;
  } // End of method
```

Note that the height of nodes **A** and **B** may be changed, but the heights of other nodes in the tree are not changed. Similarly, you can implement the RR rotation, LR rotation, and RL rotation.

26.6 Implementing the `remove` Function

Key Point: *Deleting an element from an AVL tree is the same as deleing it from a BST, except that the tree may need to be rebalanced.*

As discussed in §21.3, "Deleting Elements in a BST," to delete an element from a binary tree, the algorithm first locates the node that contains the element. Let **current** point to the node that contains the element in the binary tree and **parent** point to the parent of the **current** node. The **current** node may be a left child or a right child of the **parent** node. Two cases arise when deleting an element:

Case 1: The current node does not have a left child, as shown in Figure 21.9a. To delete the current node, simply connect the parent with the right child of the current node, as shown in Figure 21.9b.

The heights of the nodes along the path from the parent up to the root may have decreased. To ensure that the tree is balanced, invoke

```
balancePath(parent.element); // Defined in Listing 26.1
```

Case 2: The **current** node has a left child. Let **rightMost** point to the node that contains the largest element in the left subtree of the **current** node and **parentOfRightMost** point to the parent node of the **rightMost** node, as shown in Figure 21.11a. The **rightMost** node cannot have a right child but may have a left child. Replace the element value in the **current** node with the one in the **rightMost** node, connect the **parentOfRightMost** node with the left child of the **rightMost** node, and delete the **rightMost** node, as shown in Figure 21.11b.

The height of the nodes along the path from **parentOfRightMost** up to the root may have decreased. To ensure that the tree is balanced, invoke

```
balancePath(parentOfRightMost); // Defined in Listing 26.1
```

26.7 The AVLTree Class

Key Point: *The **AVLTree** class extends the **BST** class to override the **insert** and **delete** methods to rebalance the tree if necessary.*

Listing 26.3 gives the complete source code for the **AVLTree** class.

Listing 26.3 AVLTree.h

```
#ifndef AVLTREE_H
#define AVLTREE_H

#include "BinaryTree.h"
#include <vector>
#include <stdexcept>
using namespace std;

template<typename T>
class AVLTreeNode : public TreeNode<T>
{
public:
  int height; // height of the node

  AVLTreeNode(T element) : TreeNode<T>(element) // Constructor
  {
    height = 0;
  }
};

template <typename T>
class AVLTree : public BinaryTree<T>
{
public:
  AVLTree();
```

```
  AVLTree(T elements[], int arraySize);
  //  AVLTree(BinaryTree &tree); left as exercise
  //  ~AVLTree(); left as exercise

  bool insert(T element); // Redefine insert defined in BinaryTree
  bool remove(T element); // Redefine remove defined in BinaryTree

  // Redefine createNewNode defined in BinaryTree
  AVLTreeNode<T> * createNewNode(T element);

  /** Balance the nodes in the path from the specified
   * node to the root if necessary */
  void balancePath(T element);

  /** Update the height of a specified node */
  void updateHeight(AVLTreeNode<T> *node);

  /** Return the balance factor of the node */
  int balanceFactor(AVLTreeNode<T> *node);

  /** Balance LL (see Figure 26.1) */
  void balanceLL(TreeNode<T> *A, TreeNode<T> *parentOfA);

  /** Balance LR (see Figure 26.3) */
  void balanceLR(TreeNode<T> *A, TreeNode<T> *parentOfA);

  /** Balance RR (see Figure 26.2) */
  void balanceRR(TreeNode<T> *A, TreeNode<T> *parentOfA);

  /** Balance RL (see Figure 26.4) */
  void balanceRL(TreeNode<T> *A, TreeNode<T> *parentOfA);

private:
  int height;
};

template <typename T>
AVLTree<T>::AVLTree()
{
  height = 0;
}

template <typename T>
AVLTree<T>::AVLTree(T elements[], int arraySize)
{
  root = NULL;
  size = 0;

  for (int i = 0; i < arraySize; i++)
  {
    insert(elements[i]);
  }
}

template <typename T>
AVLTreeNode<T> * AVLTree<T>::createNewNode(T element)
{
  return new AVLTreeNode<T>(element);
}

```

```
template <typename T>
bool AVLTree<T>::insert(T element)
{
  bool successful = BinaryTree<T>::insert(element);
  if (!successful)
    return false; // element is already in the tree
  else
    // Balance from element to the root if necessary
    balancePath(element);

  return true; // element is inserted
}

template <typename T>
void AVLTree<T>::balancePath(T element)
{
  vector<TreeNode<T>* > *p = path(element);
  for (int i = (*p).size() - 1; i >= 0; i--)
  {
    AVLTreeNode<T> *A = static_cast<AVLTreeNode<T>*>((*p)[i]);
    updateHeight(A);
    AVLTreeNode<T> *parentOfA = (A == root) ? NULL :
      static_cast<AVLTreeNode<T>*>((*p)[i - 1]);

    switch (balanceFactor(A))
    {
      case -2:
        if (balanceFactor(
            static_cast<AVLTreeNode<T>*>(((*A).left))) <= 0)
          balanceLL(A, parentOfA); // Perform LL rotation
        else
          balanceLR(A, parentOfA); // Perform LR rotation
        break;
      case +2:
        if (balanceFactor(
             static_cast<AVLTreeNode<T>*>(((*A).right))) >= 0)
          balanceRR(A, parentOfA); // Perform RR rotation
        else
          balanceRL(A, parentOfA); // Perform RL rotation
    }
  }
}

template <typename T>
void AVLTree<T>::updateHeight(AVLTreeNode<T> *node)
{
  if (node->left == NULL && node->right == NULL) // node is a leaf
    node->height = 0;
  else if (node->left == NULL) // node has no left subtree
    node->height =
      1 + (*static_cast<AVLTreeNode<T>*>((node->right))).height;
  else if (node->right == NULL) // node has no right subtree
    node->height =
      1 + (*static_cast<AVLTreeNode<T>*>((node->left))).height;
  else
    node->height = 1 +
      max((*static_cast<AVLTreeNode<T>*>((node->right))).height,
      (*static_cast<AVLTreeNode<T>*>((node->left))).height);
}
```

```
template <typename T>
int AVLTree<T>::balanceFactor(AVLTreeNode<T> *node)
{
  if (node->right == NULL) // node has no right subtree
    return -node->height;
  else if (node->left == NULL) // node has no left subtree
    return +node->height;
  else
    return (*static_cast<AVLTreeNode<T>*>((node->right))).height -
      (*static_cast<AVLTreeNode<T>*>((node->left))).height;
}

template <typename T>
void AVLTree<T>::balanceLL(TreeNode<T> *A, TreeNode<T> *parentOfA)
{
  TreeNode<T> *B = (*A).left; // A is left-heavy and B is left-heavy

  if (A == root)
    root = B;
  else
    if (parentOfA->left == A)
      parentOfA->left = B;
    else
      parentOfA->right = B;

  A->left = B->right; // Make T2 the left subtree of A
  B->right = A; // Make A the left child of B
  updateHeight(static_cast<AVLTreeNode<T>*>(A));
  updateHeight(static_cast<AVLTreeNode<T>*>(B));
}

template <typename T>
void AVLTree<T>::balanceLR(TreeNode<T> *A, TreeNode<T> *parentOfA)
{
  TreeNode<T> *B = A->left; // A is left-heavy
  TreeNode<T> *C = B->right; // B is right-heavy

  if (A == root)
    root = C;
  else
    if (parentOfA->left == A)
      parentOfA->left = C;
    else
      parentOfA->right = C;

  A->left = C->right; // Make T3 the left subtree of A
  B->right = C->left; // Make T2 the right subtree of B
  C->left = B;
  C->right = A;

  // Adjust heights
  updateHeight(static_cast<AVLTreeNode<T>*>(A));
  updateHeight(static_cast<AVLTreeNode<T>*>(B));
  updateHeight(static_cast<AVLTreeNode<T>*>(C));
}

template <typename T>
void AVLTree<T>::balanceRR(TreeNode<T> *A, TreeNode<T> *parentOfA)
```

```
{
  // A is right-heavy and B is right-heavy
  TreeNode<T> *B = A->right;

  if (A == root)
    root = B;
  else
    if (parentOfA->left == A)
      parentOfA->left = B;
    else
      parentOfA->right = B;

  A->right = B->left; // Make T2 the right subtree of A
  B->left = A;
  updateHeight(static_cast<AVLTreeNode<T>*>(A));
  updateHeight(static_cast<AVLTreeNode<T>*>(B));
}

template <typename T>
void AVLTree<T>::balanceRL(TreeNode<T> *A, TreeNode<T> *parentOfA)
{
  TreeNode<T> *B = A->right; // A is right-heavy
  TreeNode<T> *C = B->left; // B is left-heavy

  if (A == root)
    root = C;
  else
    if (parentOfA->left == A)
      parentOfA->left = C;
    else
      parentOfA->right = C;

  A->right = C->left; // Make T2 the right subtree of A
  B->left = C->right; // Make T3 the left subtree of B
  C->left = A;
  C->right = B;

  // Adjust heights
  updateHeight(static_cast<AVLTreeNode<T>*>(A));
  updateHeight(static_cast<AVLTreeNode<T>*>(B));
  updateHeight(static_cast<AVLTreeNode<T>*>(C));
}

template <typename T>
bool AVLTree<T>::remove(T element)
{
  if (root == NULL)
    return false; // Element is not in the tree

  // Locate the node to be deleted and also locate its parent node
  TreeNode<T> *parent = NULL;
  TreeNode<T> *current = root;
  while (current != NULL)
  {
    if (element < current->element)
    {
      parent = current;
      current = current->left;
    }
    else if (element > current->element)
```

```
    {
      parent = current;
      current = current->right;
    }
    else
      break; // Element is in the tree pointed by current
  }

  if (current == NULL)
    return false; // Element is not in the tree

  // Case 1: current has no left children (See Figure 23.6)
  if (current->left == NULL)
  {
    // Connect the parent with the right child of the current node
    if (parent == NULL)
      root = current->right;
    else
    {
      if (element < parent->element)
        parent->left = current->right;
      else
        parent->right = current->right;

      // Balance the tree if necessary
      balancePath(parent->element);
    }
  }
  else
  {
    // Case 2: The current node has a left child
    // Locate the rightmost node in the left subtree of
    // the current node and also its parent
    TreeNode<T> *parentOfRightMost = current;
    TreeNode<T> *rightMost = current->left;

    while (rightMost->right != NULL)
    {
      parentOfRightMost = rightMost;
      rightMost = rightMost->right; // Keep going to the right
    }

    // Replace the element in current by the element in rightMost
    current->element = rightMost->element;

    // Eliminate rightmost node
    if (parentOfRightMost->right == rightMost)
      parentOfRightMost->right = rightMost->left;
    else
      // Special case: parentOfRightMost is current
      parentOfRightMost->left = rightMost->left;

    // Balance the tree if necessary
    balancePath(parentOfRightMost->element);
  }

  size--;
  return true; // Element inserted
}

#endif
```

The **AVLTree** class extends **BST** (line 22). Like the **BST** class, the **AVLTree** class has a no-arg constructor that constructs an empty **AVLTree** (lines 62–66) and a constructor that creates an initial **AVLTree** from an array of elements (lines 68–78).

The **createNewNode()** function defined in the **BST** class creates a **TreeNode**. This function is overridden to return an **AVLTreeNode** (lines 80–84). Note that this function is dynamically invoked from the insert function defined in **BST** (see lines 200, 222, 224, in Listing 21.3, BST.h).

The **insert** function in **AVLTree** is overridden in lines 86–97. The function first invokes the **insert** function in **BST**, and then invokes **balancePath(element)** (line 94) to ensure that tree is balanced.

The **balancePath** function first gets the nodes on the path from the node that contains the element to the root (line 102). For each node in the path, update its height (line 106), check its balance factor (line 110), and perform appropriate rotations if necessary (lines 112–125).

Four functions for performing rotations are defined in lines 158–245. Each function is invoked with two **TreeNode<T>** arguments **A** and **parentOfA** to perform an appropriate rotation at node **A**. How each rotation is performed is pictured in Figures 26.1–26.4. After the rotation, the height of these nodes is updated.

The **remove** function in **AVLTree** is overridden in lines 247–322. The function is the same as the one implemented in the **BST** class, except that you have to rebalance the nodes after deletion in lines 289 and 317.

26.8 Testing the AVLTree Class

Key Point: *This section gives an example of using the AVLTree class.*

Listing 26.4 gives a test program. The program creates an **AVLTree** initialized with an array of integers **25**, **20**, and **5** (lines 22–23), inserts elements in lines 28-39, and deletes elements in lines 41–49.

Listing 26.4 TestAVLTree.cpp

```
 1  #include <iostream>
 2  #include "AVLTree.h"
 3  using namespace std;
 4
 5  template <typename T>
 6  void printTree(AVLTree<T> &tree)
 7  {
 8    // Traverse tree
 9    cout << "\nInorder (sorted): " << endl;
10    tree.inorder();
11    cout << "\nPostorder: " << endl;
12    tree.postorder();
13    cout << "\nPreorder: " << endl;
14    tree.preorder();
15    cout << "\nThe number of nodes is " << tree.getSize();
16    cout << endl;
17  }
18
19  int main()
20  {
21    // Create an AVL tree
22    int numbers[] = {25, 20, 5};
23    AVLTree<int> tree(numbers, 3);
24
25    cout << "After inserting 25, 20, 5:" << endl;
26    printTree<int>(tree);
27
28    tree.insert(34);
29    tree.insert(50);
30    cout << "\nAfter inserting 34, 50:" << endl;
31    printTree<int>(tree);
32
33    tree.insert(30);
34    cout << "\nAfter inserting 30" << endl;
35    printTree<int>(tree);
36
37    tree.insert(10);
38    cout << "\nAfter inserting 10" << endl;
39    printTree(tree);
40
41    tree.remove(34);
42    tree.remove(30);
43    tree.remove(50);
44    cout << "\nAfter removing 34, 30, 50:" << endl;
45    printTree<int>(tree);
46
47    tree.remove(5);
48    cout << "\nAfter removing 5:" << endl;
49    printTree<int>(tree);
50
```

```
  return 0;
}
```

Sample output

```
After inserting 25, 20, 5:
Inorder (sorted): 5 20 25
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 50 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7

After removing 34, 30, 50:
Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3
```

Figure 26.9 shows how the tree evolves as elements are added to it. After **25** and **20** are added, the tree is as shown in Figure 26.9a. **5** is inserted as a left child of **20**, as shown in Figure 26.9b. The tree is not balanced. It is left-heavy at node **25**. Perform an LL rotation to produce an AVL tree, as shown in Figure 26.9c.

After inserting **34**, the tree is as shown in Figure 26.9d. After inserting **50**, the tree is as shown in Figure 26.9(e). The tree is not balanced. It is right-heavy at node **25**. Perform an RR rotation to produce an AVL tree, as shown in Figure 26.9(f).

After inserting **30**, the tree is as shown in Figure 26.9(g). The tree is not balanced. Perform an RL rotation to produce an AVL tree, as shown in Figure 26.9(h).

After inserting **10**, the tree is as shown in Figure 26.9(i). The tree is not balanced. Perform an LR rotation to produce an AVL tree, as shown in Figure 26.9(j).

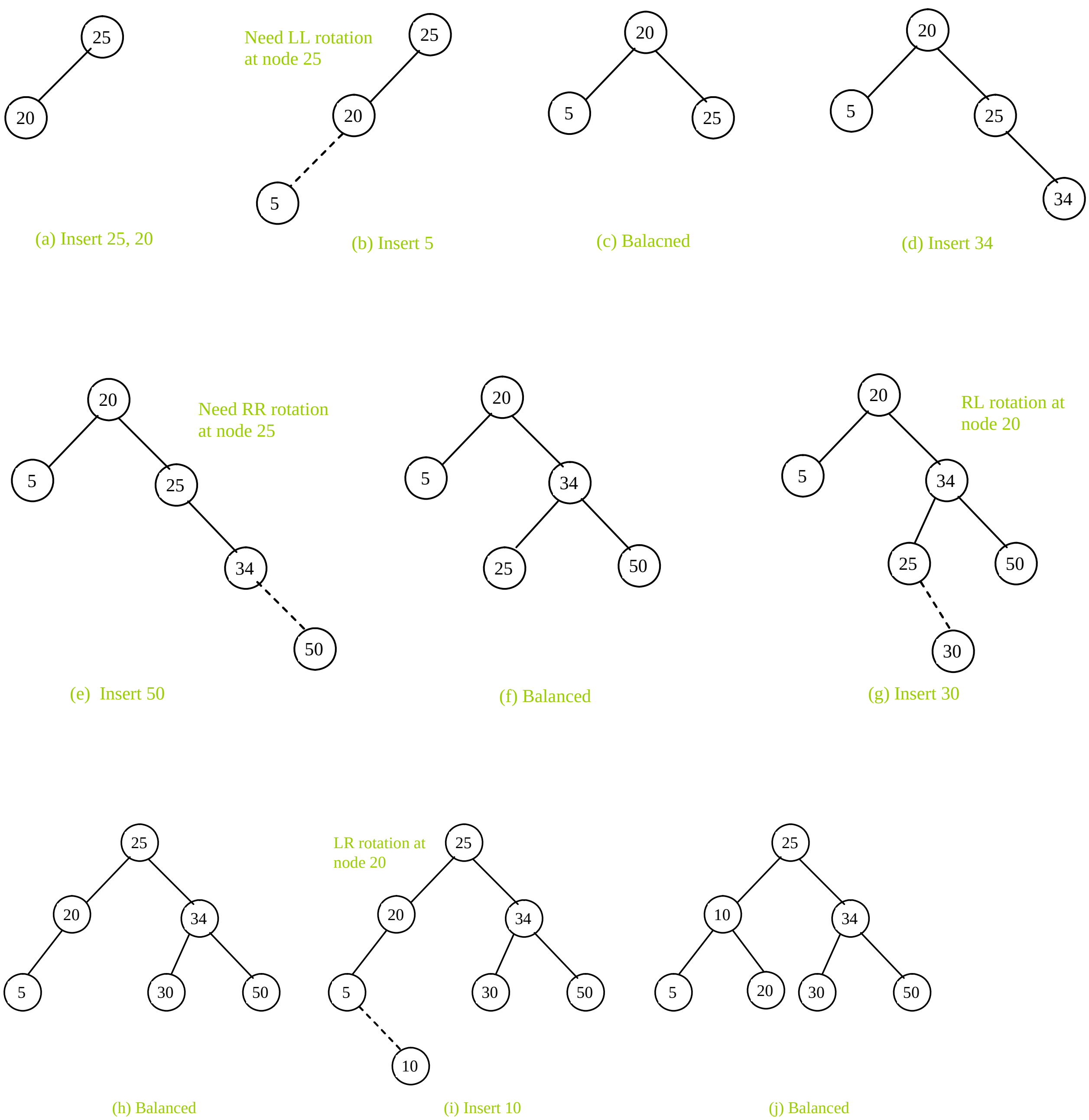


Figure 26.9

The tree evolves as new elements are inserted.

Figure 26.10 shows how the tree evolves as elements are deleted. After deletion of **`34`**, **`30`**, and **`50`**, the tree is as shown in Figure 26.10b. The tree is not balanced. Perform an LL rotation to produce an AVL tree, as shown in Figure 26.10c.

After deleting **`5`**, the tree is as shown in Figure 26.10d. The tree is not balanced. Perform an RL rotation to produce an AVL tree, as shown in Figure 26.10(e).

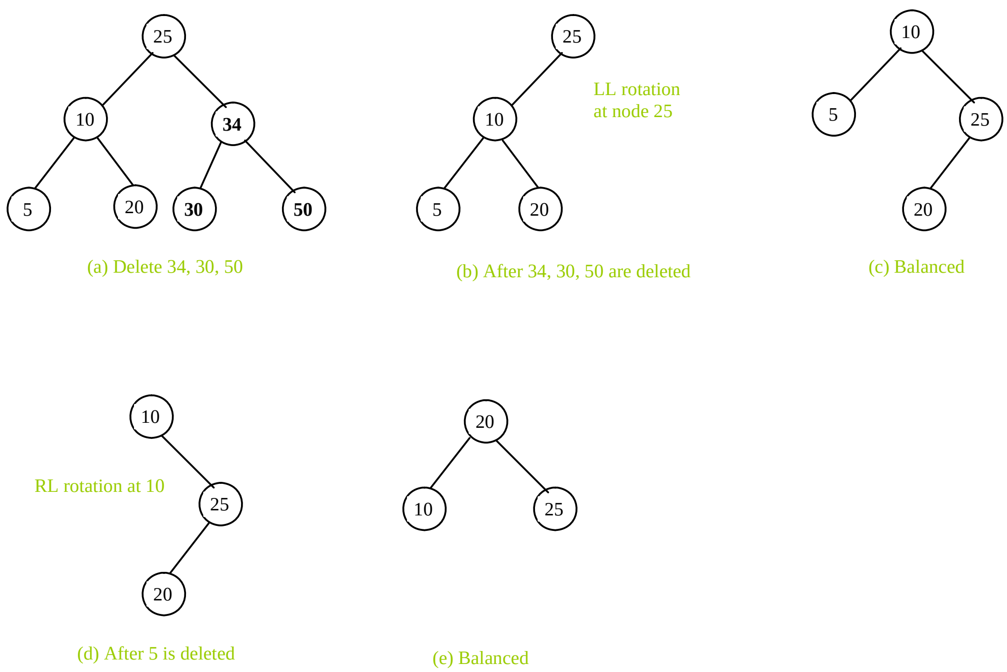


Figure 26.10

The tree evolves as the elements are deleted from it.

Check point

26.3 Why is the **`createNewNode`** function protected?

26.4 When is the **`updateHeight`** function invoked? When is the **`balanceFactor`** function invoked? When is the **`balancePath`** function invoked?

26.5 What are the data fields in the **`AVLTreeNode`** class? What are the data fields in the **`AVLTree`** class?

26.6 In the **`insert`** and **`remove`** functions, once you have performed a rotation to balance a node in the tree, is it possible that there are still unbalanced nodes?

26.7 Show the change of an AVL tree when inserting **`1`**, **`2`**, **`3`**, **`4`**, **`10`**, **`9`**, **`7`**, **`5`**, **`8`**, **`6`** into the tree, in this order.

26.8 For the tree built in the preceding question, show the change of the tree after deleting **`1`**, **`2`**, **`3`**, **`4`**, **`10`**, **`9`**, **`7`**, **`5`**, **`8`**, **`6`** from the tree in this order.

26.9 AVL Tree Time Complexity Analysis

Key Point: *Since the height of an AVL tree is O(log n), the time complexity of the **search**, **insert**, and **delete** methods in **AVLTree** is O(log n).*

The time complexity of the **`search`**, **`insert`**, and **`delete`** functions in **`AVLTree`** depends on the height of the tree. We can prove that the height of the tree is $O(\log n)$.

Let $G(h)$ denote the minimum number of the nodes in an AVL tree with height h. Obviously, $G(1)$ is 1 and $G(2)$ is 2.

An AVL tree with height $h \geq 3$ must have at least two subtrees: one with height $h-1$ and the other with height $h-2$. So, $G(h) = G(h-1) + G(h-2) + 1$. Recall that a Fibonacci number at index i can be described using the recurrence relation $F(i) = F(i-1) + F(i-2)$. So, the function $G(h)$ is essentially the same as $F(i)$. It can be proven that

$h < 1.4405 \log(n+2) - 1.3277$, where n in the number of nodes in the tree. Therefore, the height of an AVL tree is $O(\log n)$.

The **search**, **insert**, and **delete** functions involve only the nodes along a path in the tree. The **updateHeight** and **balanceFactor** functions are executed in a constant time for each node in the path. The **balancePath** function is executed in a constant time for a node in the path. So, the time complexity for the **search**, **insert**, and **delete** functions is $O(\log n)$.

26.10 Splay Trees

Key Point: *A splay tree is a binary search tree with an average time complexity of O(n) for search, insertion, and deletion.*

If the elements you access in a BST are near the root, it would take just $O(1)$ time to search for them. Can we design a BST that place the frequently accessed elements near the root? *Splay trees*, invented by Sleator and Tarjan, are a special type of BST for just this purpose. A splay tree is a self-adjusting BST. When an element is accessed, it is moved to the root under the assumption that it will very likely be accessed again in the near future. If this turns out to be the case, subsequent accesses to the element will be very efficient.

An AVL tree applies the rotation operations to keep it balanced. A splay tree does not enforce the height explicitly. However, it uses the move-to-root operations, called *splaying*, after every access, in order to move the newly-accessed element to the root and keep the tree balanced. An AVL tree guarantees the height to be $O(\log n)$. A splay does not guarantee it. Interestingly, the splaying operation guarantees the average time for search, insertion, and deletion to be $O(\log n)$.

The splaying operation is performed at the last node reached during a search, insertion, or deletion operation. Through a sequence of restructuring operations, the node is moved to the root. The specific rule for determining which node to splay is as follows:

- `search(element)`: If the element is found in a node u, we splay u. Otherwise, we splay the leaf node where the search terminates unsuccessfully.
- `insert(element)`: We splay the newly created node that contains the element.
- `remove(element)`: We splay the parent of the node that contains the element. If the node is the root, we splay its left child or right child. If the element is not in the tree, we splay the leaf node where the search terminates unsuccessfully.

How do you splay a node? Can it be done in an arbitrary fashion? No. To achieve the average $O(\log n)$ time, splaying must be performed in certain ways. The specific operations we perform to move a node u up depend on its relative position to its parent v and its grandparent w. Consider three cases:

zig-zig Case: u and v are both left children or right children, as shown in Figures 26.10a and 26.11a. Restructure u, v, and w to make u the parent of v and v the parent of w, as shown in Figures 26.10b and 26.11b.

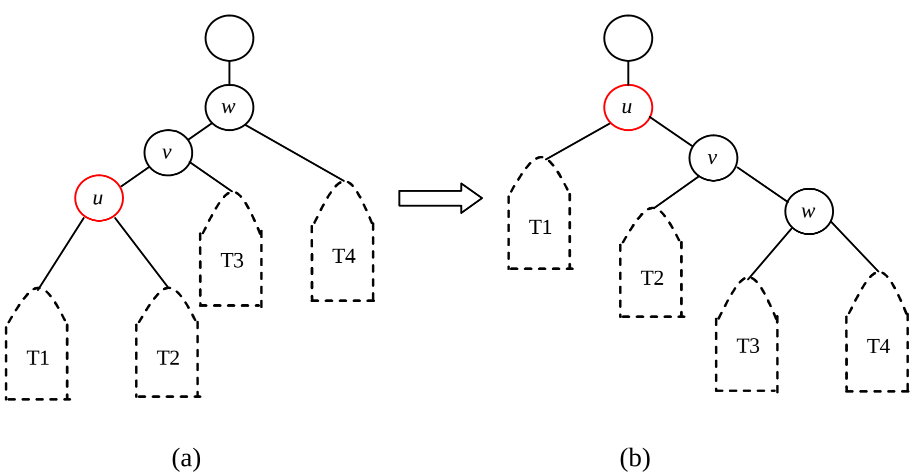


Figure 26.11

Left zig-zig restructure.

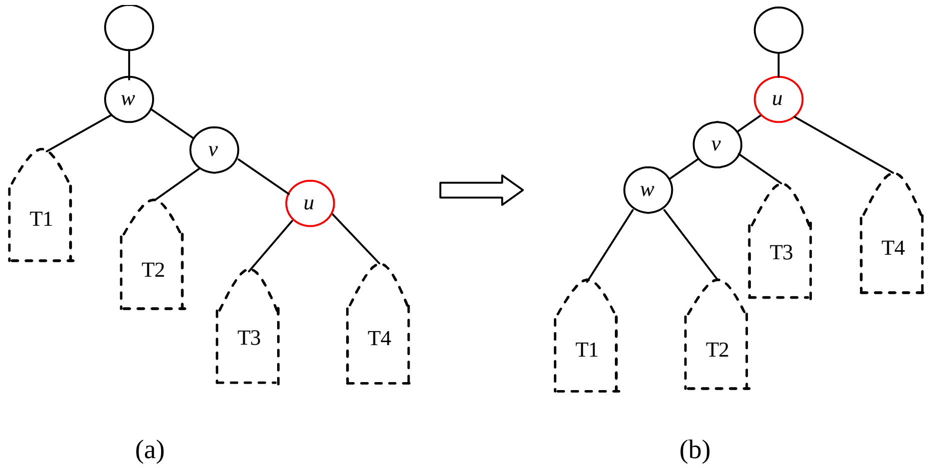


Figure 26.12

Right zig-zig restructure.

zig-zag Case: u is the right child of v and v is the left child of w, as shown in Figure 26.12a, or u is the left child of v and v is the right child of w, as shown in Figure 26.13a. Restructure u, v, and w to make u the parent of v and *w*, as shown in Figures 26.12b and 26.13b.

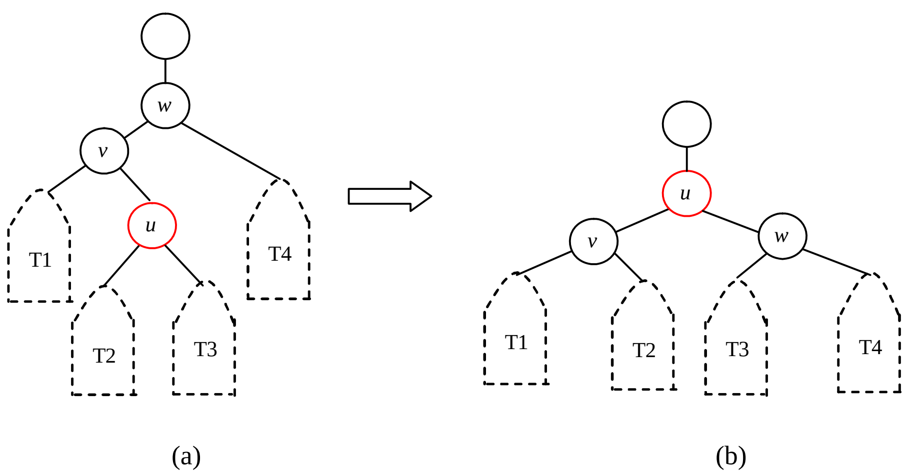


Figure 26.13

Left zig-zag restructure.

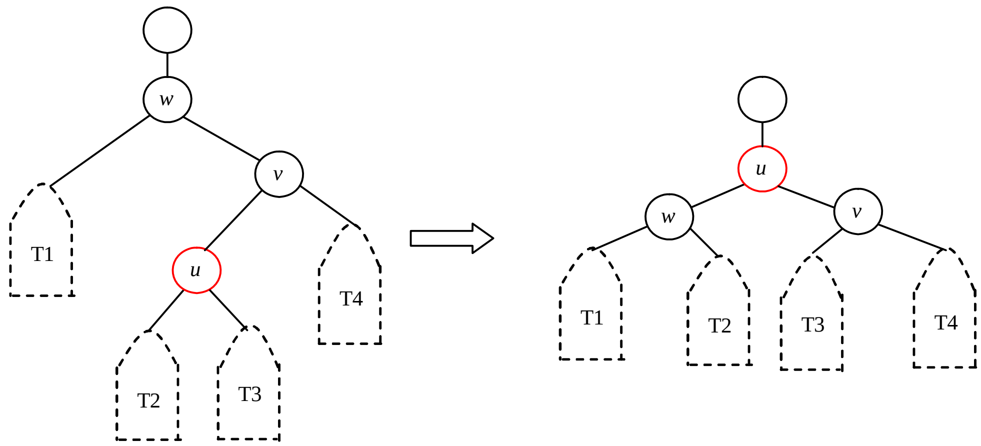


Figure 26.14

Right zig-zag restructure.

zig Case: v is the root, as shown in Figures 26.14a and 26.15a. Restructure u and v and make u the root, as shown in Figures 26.14b and 26.15b.

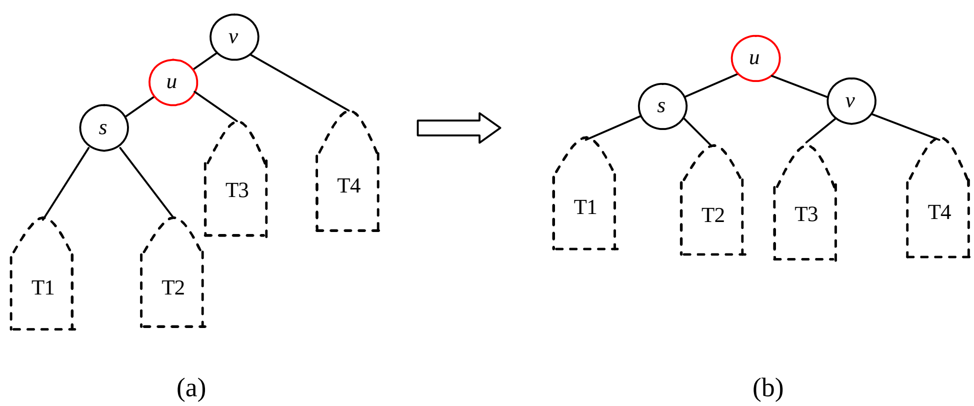
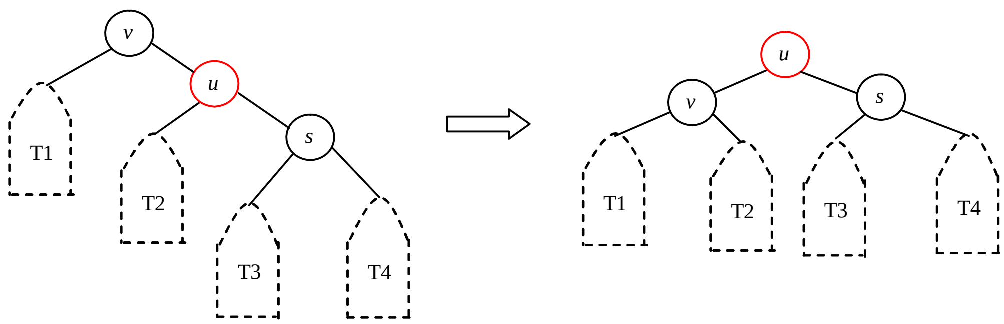


Figure 26.15

Left zig restructure.



(a) (b)

Figure 26.16

Right zig restructure.

Pedagogical NOTE

Run from www.cs.armstrong.edu/liang/animation/SplayTreeAnimation.html to see how a Splay tree works, as shown in Figure 26.17.

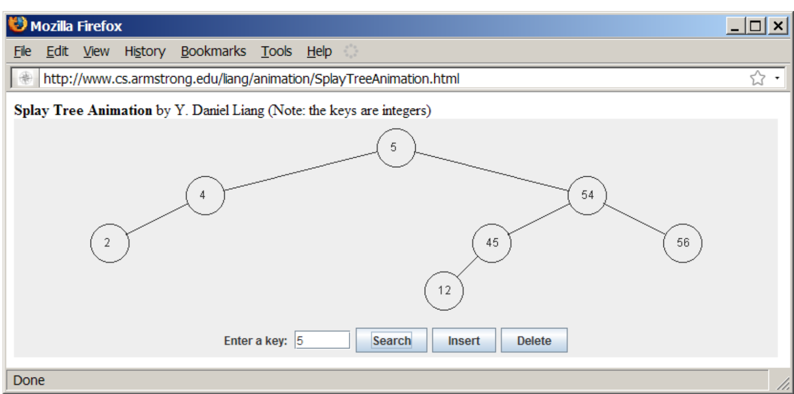


Figure 26.17

The animation tool enables you to insert, delete, and search elements visually.

The algorithm for search, insert, and delete in a splay tree is the same as in a regular binary search tree. The difference is that you have to perform a splay operation from the target node to the root. The splay operation consists of a sequence of restructurings. Figure 26.18 shows how the tree evolves as elements **25**, **20**, **5**, and **34** are inserted to it.

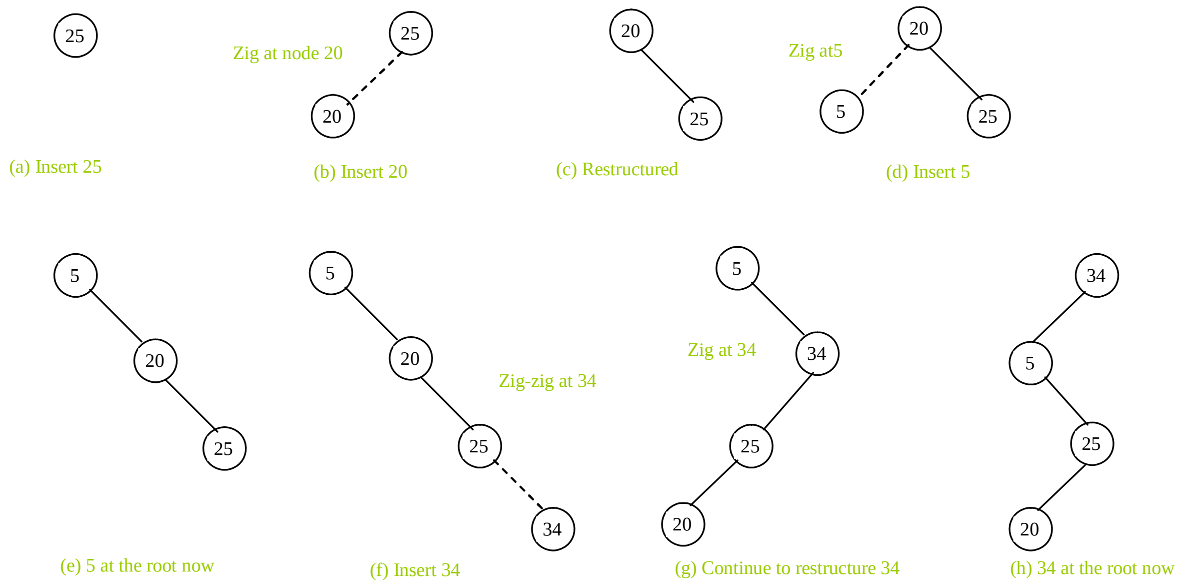


Figure 26.18

The tree evolves as new elements are inserted.

Suppose you perform a search for element **20** for the tree in Figure 26.17(h). Since **20** is in the tree, splay the node for **20**; the resulting tree is shown in Figure 26.18.

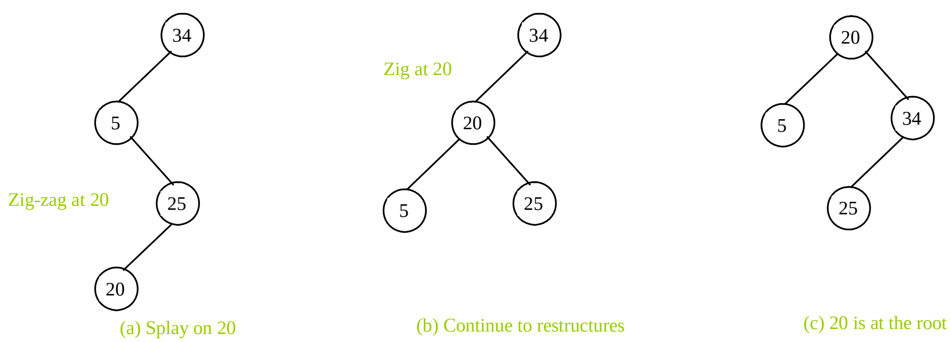


Figure 26.19

The tree is adjusted after searching ***20***.

Suppose you perform a search for element **21** for the tree in Figure 26.19c. Since **21** is not in the tree and the last node reached in the search is **25**, splay the node for **25**; the resulting tree is shown in Figure 26.20.

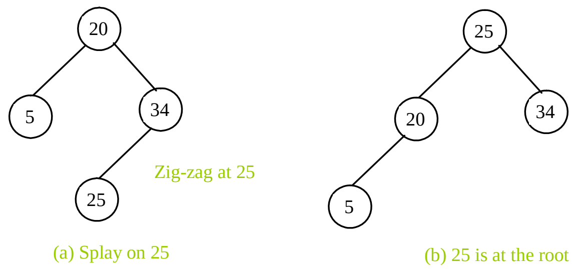


Figure 26.20

The tree is adjusted after searching ***`21`***.

Suppose you delete element **`5`** from the tree in Figure 26.20b. Since the node for **`20`** is the parent node for the node that contains **`5`**, splay the node for **`20`**; the resulting tree is shown in Figure 26.21.

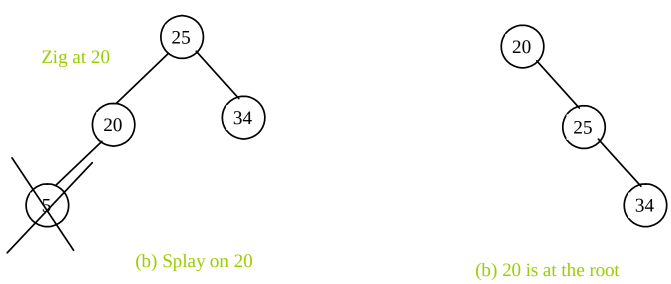


Figure 26.21

The tree is adjusted after deleting ***`5`***.

When moving a node u up, we perform a zig-zig, or a zig-zag if u has a grandparent, and perform a zig otherwise. After a zig-zig or a zig-zag is performed on u, the depth of u is decreased by 2 and after a zig is performed, the depth of u is decreased by 1. Let d denote the depth of u. If d is odd, a final zig is performed. If d is even, no zig operation is performed. Since a single zig-zig, zig-zag, or zig operation can be done in constant time, the overall time for a splay operation is $O(d)$. Though the runtime for a single access to a splay tree may be $O(1)$ or $O(n)$, it has been proven that the average time complexity for all accesses is $O(\log n)$. Splay trees are easier to implement than AVL trees. The implementation of splay trees is left as an exercise (see Programming Exercise 26.7).

Check point

26.9 Show the changes in a splay tree when **`1`**, **`2`**, **`3`**, **`4`**, **`10`**, **`9`**, **`8`**, **`6`** are inserted into it, in this order.

26.10 For the tree built in the preceding question, show the changes in the tree after **`1`**, **`9`**, **`7`**, **`5`**, **`8`**, **`6`** are deleted from it, in this order. (Note that **`7`** and **`5`** are not in the tree.)

26.11 Show an example with all nodes in one chain after inserting six elements.

Key Terms

- **AVL tree**
- **LL rotation**
- **LR rotation**
- **RR rotation**
- **RL rotation**
- **balance factor**
- **left-heavy**
- **right-heavy**
- **rotation**
- **perfectly balanced**
- **well balanced**
- **splay tree**

Chapter Summary

1. An AVL tree is a well-balanced binary tree.

2. In an AVL tree, the difference between the heights of two subtrees for every node is **0** or **1**.
3. The process for inserting or deleting an element in an AVL tree is the same as for a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation.
4. Imbalances in the tree caused by insertions and deletions are rebalanced through subtree rotations at the node of the imbalance.
5. The process of rebalancing a node is called a *rotation*. There are four possible rotations: LL rotation, LR rotation, RR rotation, and RL rotation.
6. The height of an AVL tree is $O(\log n)$. So, the time complexities for the search, insert, and delete functions are $O(\log n)$.
7. Splay trees are a special type of BST that provide quick access for frequently accessed elements.
8. The process for inserting or deleting an element in a splay tree is the same as for a regular binary search tree. The difference is that you have to perform a sequence of restructuring operations to move a node up to the root.
9. AVL trees are guaranteed to be well balanced. Splay trees may not be well balanced, but their average time complexity is $O(\log n)$.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

26.1* (*Store characters*) Write a program that inserts **26** lowercase letters from a to z into a **BST** and an **AVLTree** in this order, and displays the characters in the trees in inorder, preorder and postorder, respectively.

26.2 (*Compare performance*) Write a test program that randomly generates 500,000 numbers and inserts them into a **`BST`**, reshuffles the 500000 numbers and performs search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for **`AVLTree`**. Compare the execution time of these two programs.

26.3 (*Revise **`AVLTree`***) Revise the **`AVLTree`** class by adding the copy constructor and destructor.

26.4** (*Parent reference for **`BST`***) Suppose that the **`TreeNode`** class defined in **`BST`** contains a reference to the node's parent, as shown in Programming Exercise 21.7. Implement the **`AVLTree`** class to support this change. Write a test program that adds numbers **`1`**, **`2`**, ..., **`100`** to the tree and displays the paths for all leaf nodes.

26.5** (*The* k*th smallest element*) You can find the *k*th smallest element in a BST in $O(n)$ time from an inorder iterator. For an AVL tree, you can find it in $O(\log n)$ time. To achieve this, add a new data field named **`size`** in **`AVLTreeNode`** to store the number of nodes in the subtree rooted at this node. Note that the size of a node v is one more than the sum of the sizes of its two children. Figure 26.11 shows an AVL tree and the **`size`** value for each node in the tree.

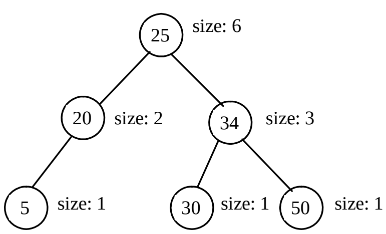


Figure 26.11

*The **`size`** data field in **`AVLTreeNode`** stores the number of nodes in the subtree rooted at the node.*

In the **`AVLTree`** class, add the following function to return the *k*th smallest element in the tree.

```
T find(int k)
```

The function returns **`NULL`** if **`k < 1`** or **`k > the size of the tree`**. This function can be implemented using a recursive function **`find(k, root)`** that returns the *k*th smallest element in the tree with the specified root. Let **`A`** and **`B`** be the left and right children of the root, respectively. Assuming that the tree is not empty and $k \le root.size$, **`find(k, root)`** can be recursively defined as follows:

$$\text{find(k, root)} = \begin{cases} \text{root.element, if A is null and k is 1;} \\ \text{B.element, if A is null and k is 2;} \\ \text{f(k, A), if k <= A.size;} \\ \text{root.element, if k} = \text{A.size} + 1; \\ \text{f(k - A.size - 1, B), if k} > \text{A.size} + 1; \end{cases}$$

Modify the **`insert`** and **`delete`** functions in **`AVLTree`** to set the correct value for the **`size`** property in each node. The **`insert`** and **`delete`** functions will still be in $O(\log n)$ time. The **`find(k)`** function can be implemented in $O(\log n)$ time. Therefore, you can find the *k*th smallest element in an AVL tree in $O(\log n)$ time.

26.6** (*Closest pair of points*) §18.8 introduced an algorithm for finding a closest pair of points in $O(n \log n)$ time using a divide-and-conquer approach. The algorithm was implemented using recursion with a lot of overhead. Using the plain-sweep approach along with an AVL tree, you can solve the same problem in $O(n \log n)$ time. Implement the algorithm using an **`AVLTree`**.

26.7*** (*The **`SplayTree`** class*) §26.10 introduced the splay tree. Implement the **`SplayTree`** class by extending the **`BST`** class and overriding the **`search`**, **`insert`**, and **`remove`** functions.

26.8**(*Compare performance*) Write a test program that randomly generates 500,000 numbers and inserts them into an **`AVLTree`**, reshuffles the 500,000 numbers and performs search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for **`SplayTree`**. Compare the execution times of these two programs.

*****26.9** (*Find u with smallest cost[u] efficiently*) The **`getShortestPath`** function in Listing 25.2 finds a **`u`** with the smallest **`cost[u]`** using a linear search, which takes O(|V|). The search time can be reduced to O(log|V|) using an AVL tree. Modify the function using an AVL to store the vertices in **`V-T`** and use Listing 25.9 TestShortestPath.cpp to test your new implementation.