CHAPTER 21

Binary Search Trees

Objectives

- To represent and access the elements in a binary search tree (§21.2.2).
- To represent binary trees using linked data structures (§21.2.1).
- To search an element in binary search tree (§21.2.2).
- To insert an element into a binary search tree (§21.2.3).
- To traverse a binary tree in inorder, postorder, and preorder (§21.2.4).
- To define and implement the **`BST`** tree class (§21.2.5).
- To delete a node in a binary search tree (§21.3).
- To design and implement the iterator for traversing the elements in a binary tree (§21.4).

21.1 Introduction

Key Point: *A tree is a classic data structure with many important applications.*

The preceding chapter introduced several basic data structures such as linked lists, stacks, queues, heaps, and priority queues. This chapter introduces binary search trees.

21.2 Binary Search Trees

Key Point: *A binary search tree can be implemented using a linked structure.*

A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It either is empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. Examples of binary trees are shown in Figure 21.1.

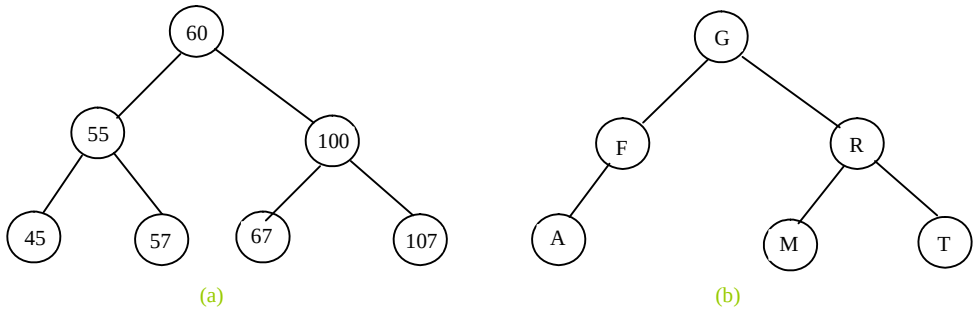


Figure 21.1
Each node in a binary tree has zero, one, or two branches.

The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a *level* of the tree. *Siblings* are nodes that share the same parent node. The root of a left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. The *height* of an empty tree is **0**. The height of a non-empty tree is the length of the path from the root node to its furthest leaf + 1. Consider the tree in Figure 21.1a. The length for the path from node 60 to 45 is **2**. The depth of node 60 is **0**, the depth of node 55 is **1**, and the depth of node 45 is **2**. The height of the tree is **3**. Nodes 45 and 57 are siblings. Nodes 45, 57, 67, and 107 are in the same level.

A special type of binary tree called a *binary search tree* (BST) is often useful. A BST (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less

than the value of the node, and the value of any node in its right subtree is greater than the value of the node. The binary trees in Figure 21.1 are all BSTs. This section is concerned with BSTs.

Pedagogical NOTE

Follow the link **www.cs.armstrong.edu/liang/cpp2e/animation/BSTAnimation.html** to see how a BST works, as shown in Figure 21.2.

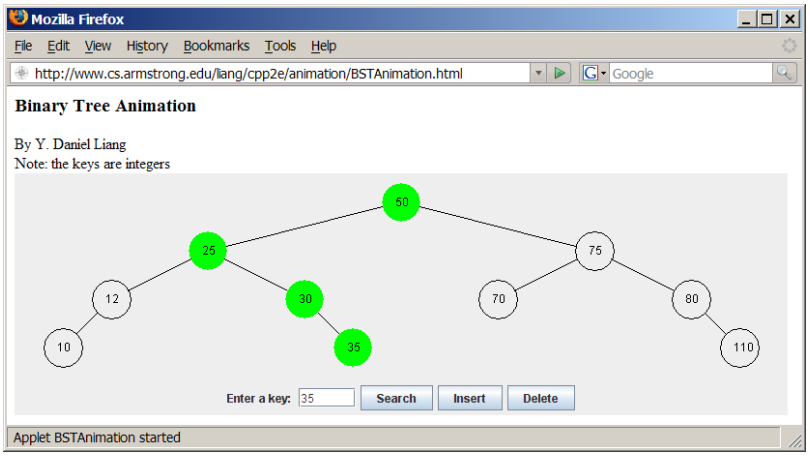


Figure 21.2
The animation tool enables you to insert, delete, and search elements visually.

21.2.1 Representing Binary Search Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure 21.3.

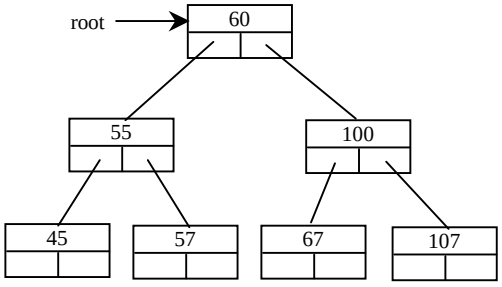


Figure 21.3
A binary tree can be represented using a set of linked nodes.

A node can be defined as a class, as follows:

```
template<typename T>
class TreeNode
{
public:
  T element; // Element contained in the node
  TreeNode<T>* left; // Pointer to the left child
  TreeNode<T>* right; // Pointer to the right child

  TreeNode() // No-arg constructor
  {
    left = NULL;
    right = NULL;
  }

  TreeNode(T element) // Constructor
  {
    this->element = element;
    left = NULL;
    right = NULL;
  }
};
```

The variable **root** refers to the root node of the tree. If the tree is empty, **root** is **NULL**. The following code creates the first three nodes of the tree in Figure 21.1:

```
// Create the root node
TreeNode<int>* root = new TreeNode<int>(60);

// Create the left child node
root->left = new TreeNode<int>(55);

// Create the right child node
root->right = new TreeNode<int>(100);
```

21.2.2 Accessing Nodes in Binary Trees

Suppose a tree with three nodes is created as in the preceding section. You can access the nodes in the tree through the **root** pointer. Here are the statements to display the elements at the root and its left and right nodes.

```
// Display the root element
cout << root->element << endl;

// Display the element in the left child of the root
cout << (root->left)->element << endl;

// Display the element in the right child of the root
cout << (root->right)->element << endl;
```

21.2.3 Searching for an Element

To search for an element in the BST, you start from the root and scan down from the root until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 21.1. Let **`current`** point to the root (line 4). Repeat the following steps until **`current`** is **`NULL`** (line 6) or the element matches **`current->element`** (line 16).

If element is less than current->element, assign current->left to current (line 9).

If **`element`** is greater than **`current->element`**, assign **`current->right`** to **`current`** (line 13).

If **`element`** is equal to **`current->element`**, return **`true`** (line 16).

If the **`current`** is **`null`**, the subtree is empty and the element is not in the tree (line 18).

Listing 21.1 Searching for an Element in a BST

```
template<typename T>
bool search(T element)
{
  TreeNode<T>* current = root; // Start from the root

  while (current != NULL)
    if (element < current->element)
    {
      current = current->left; // Go left
    }
    else if (element > current->element)
    {
      current = current->right; // Go right
    }
    else // Element matches current->element
      return true; // Element is found

  return false; // Element is not in the tree
}
```

21.2.3 Inserting an Element into a BST

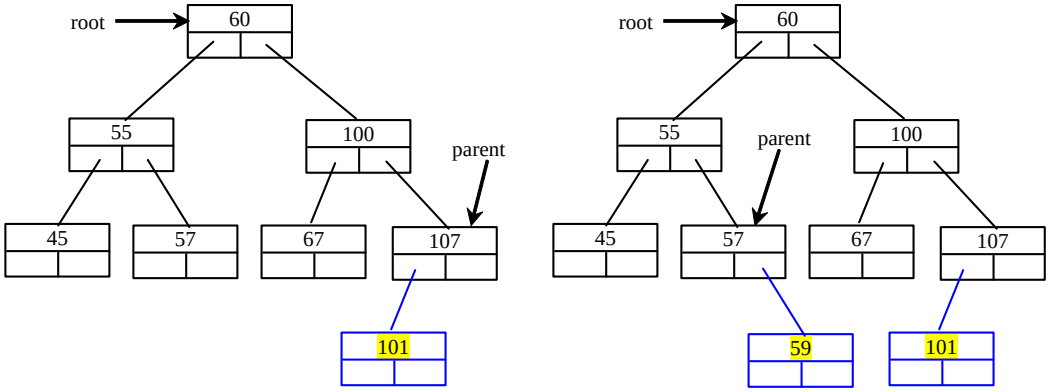
To insert an element into a BST, you need to locate where to insert it. The key idea is to locate the parent for the new node. Listing 21.2 gives the algorithm.

Listing 21.2 Inserting an Element into a BST

```
template<typename T>
bool insert(T element)
{
 if (root == NULL)
   root = new TreeNode<T>(element);
 else
 {
   // Locate the parent node
   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
       return false; // Duplicate node not inserted

    // Create the new node and attach it to the parent node
    if (element < parent->element)
      parent->left = new TreeNode<T>(element);
    else
      parent->right = new TreeNode<T>(element);

    return true; // Element inserted
  }
}
```

If the tree is empty, create a root node with the new element (lines 4–5). Otherwise, locate the parent node for the new element node (lines 9–22). Create a new node for the element and link this node to its parent node (lines 25–28). If the new element is less than the parent element, the node for the new element will be the left child of the parent (line 26). If the new element is greater than the parent element, the node for the new element will be the right child of the parent (line 28).

For example, to insert **`101`** into the tree in Figure 21.3, after the **`while`** loop finishes in the algorithm, **`parent`** points to the node for **`107`**, as shown in Figure 21.4a. The new node for **`101`** becomes the left child of the parent. To insert **`59`** into the tree, after the **`while`** finishes in the algorithm, the parent points to the node for **`57`**, as shown in Figure 21.4b. The new node for **`59`** becomes the right child of the parent.



(a) Inserting 101 **(b) Inserting 59**

Figure 21.4
Two new elements are inserted into the tree.

21.2.4 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *preorder*, *postorder*, *depth-first*, and *breadth-first* traversals.

With inorder traversal, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node. The inorder traversal displays all the nodes in a BST in increasing order.

With postorder traversal, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself. An application of postorder is to find the size of the directory in a file system. As shown in Figure 21.5, each directory is an internal node and a file is a leave node. You can apply postorder to get size of each file and subdirectory before finding the size of the root directory.

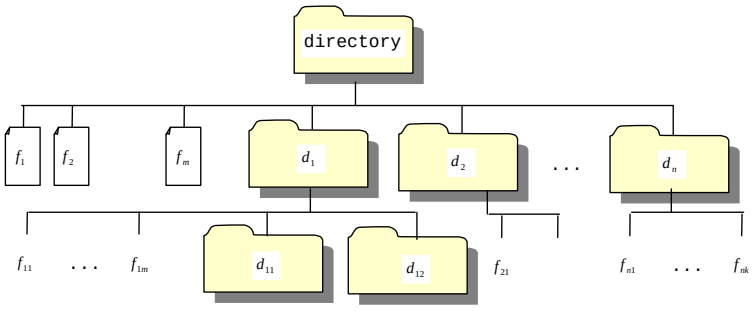


Figure 21.5

A directory contains files and subdirectories.

With preorder traversal, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. Depth-first traversal is the same as preorder traversal. An application of preorder is to print a structure document. As shown in Figure 21.6, you can print the table of contents in a book using the preorder.

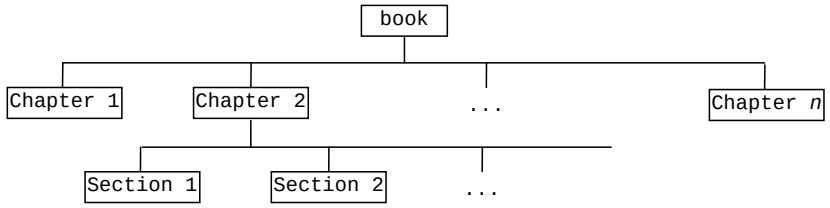


Figure 21.6

A tree can be used to represent a structured document such as a book, chapters, and sections.

NOTE

You can reconstruct a binary search tree by inserting the elements in their preorder.

The reconstructed tree preserves the parent and child relationship for the nodes in the original binary search tree.

With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 21.4b, the inorder is

```
45 55 57 59 60 67 100 101 107
```

The postorder is

```
45 59 57 55 67 101 107 100 60
```

The preorder is

```
60 55 45 57 59 100 67 107 101
```

The breadth-first traversal is

```
60 55 100 45 57 67 107 59 101
```

21.2.5 The `BST` Class

Let us define a binary search tree class named **`BST`**, as shown in Figure 21.7. Its implementation is given in Listing 21.3. The implementation of search, insert, inorder traversal, postorder traversal, and preorder traversal are discussed in the preceding sections. The **`remove`** function will be discussed in §21.3. The iterators for traversing the elements in a binary tree will be discussed in §21.4.

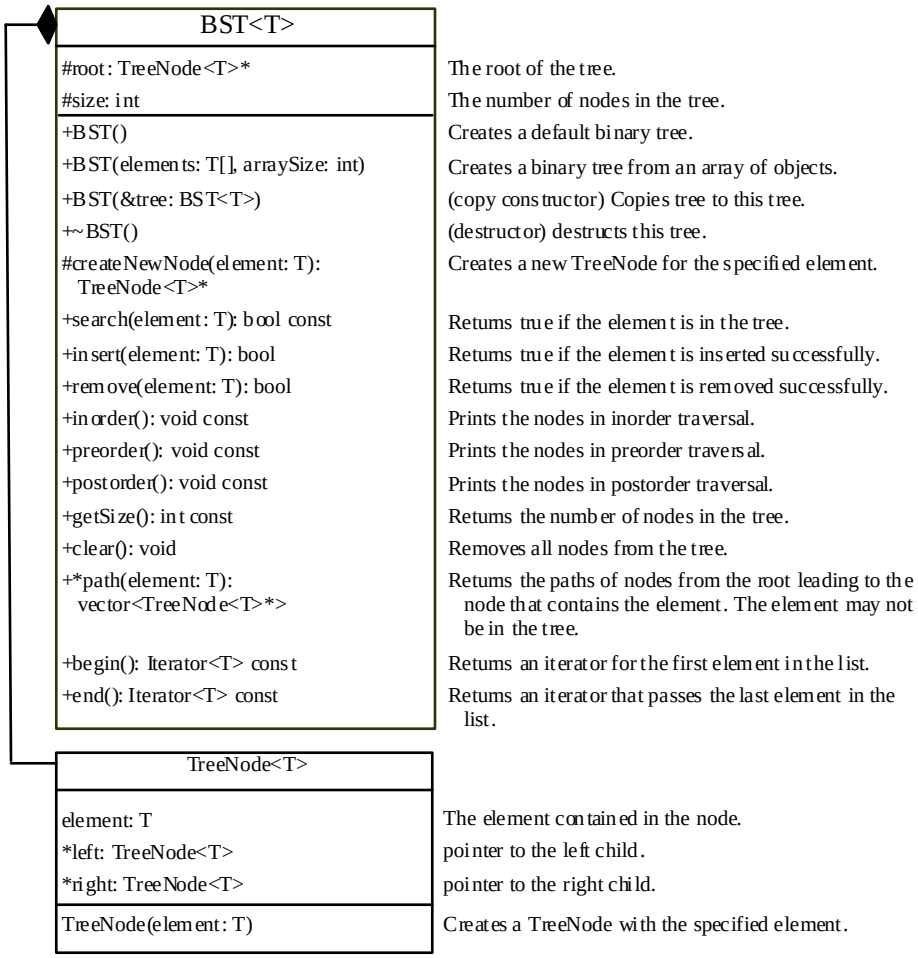


Figure 21.7
*The **BST** class supports many operations for a binary search tree.*

Listing 21.3 BST.h

```
 1  #ifndef BST_H
 2  #define BST_H
 3
 4  #include <vector>
 5  #include <stdexcept>
 6  using namespace std;
 7
 8  template<typename T>
 9  class TreeNode
10  {
11  public:
12    T element; // Element contained in the node
13    TreeNode<T>* left; // Pointer to the left child
14    TreeNode<T>* right; // Pointer to the right child
15
16    TreeNode(T element) // Constructor
17    {
18      this->element = element;
19      left = NULL;
```

```
    right = NULL;
  }
};

template <typename T>
class Iterator : public std::iterator<std::forward_iterator_tag,
T>
{
public:
  Iterator(TreeNode<T>* p)
  {
    if (p == NULL)
      current = -1; // The end
    else
    {
      // Get all the elements in inorder
      treeToVector(p);
      current = 0;
    }
  }

  Iterator operator++()
  {
    current++;
    if (current == v.size())
      current = -1; // The end
    return *this;
  }

  T &operator*()
  {
    return v[current];
  }

  bool operator == (const Iterator<T> &iterator) const
  {
    return current == iterator.current;
  }

  bool operator != (const Iterator<T> &iterator) const
  {
    return current != iterator.current;
  }

private:
  int current;
  vector<T> v;
  void treeToVector(TreeNode<T>* p)
  {
    if (p != NULL)
    {
      treeToVector(p -> left);
      v.push_back(p -> element);
      treeToVector(p -> right);
    }
  }
};

```

```
77  template <typename T>
78  class BST
79  {
80  public:
81    BST();
82    BST(T elements[], int arraySize);
83    BST(BST<T> &tree);
84    ~BST();
85    bool search(T element) const;
86    virtual bool insert(T element);
87    virtual bool remove(T element);
88    void inorder() const;
89    void preorder() const;
90    void postorder() const;
91    int getSize() const;
92    void clear();
93    vector<TreeNode<T>* >* path(T element) const;
94
95    Iterator<T> begin() const
96    {
97      return Iterator<T>(root);
98    };
99
100   Iterator<T> end() const
101   {
102     return Iterator<T>(NULL);
103   };
104
105 protected:
106   TreeNode<T>* root;
107   int size;
108   virtual TreeNode<T>* createNewNode(T element);
109
110 private:
111   void inorder(TreeNode<T>* root) const;
112   void postorder(TreeNode<T>* root) const;
113   void preorder(TreeNode<T>* root) const;
114   void copy(TreeNode<T>* root);
115   void clear(TreeNode<T>* root);
116 };
117
118 template <typename T>
119 BST<T>::BST()
120 {
121   root = NULL;
122   size = 0;
123 }
124
125 template <typename T>
126 BST<T>::BST(T elements[], int arraySize)
127 {
128   root = NULL;
129   size = 0;
130
131   for (int i = 0; i < arraySize; i++)
132   {
133     insert(elements[i]);
134   }
```

```
135 }
136
137 /* Copy constructor */
138 template <typename T>
139 BST<T>::BST(BST<T> &tree)
140 {
141   root = NULL;
142   size = 0;
143   copy(tree.root); // Recursively copy nodes to this tree
144 }
145
146 /* Copies the element from the specified tree to this tree */
147 template <typename T>
148 void BST<T>::copy(TreeNode<T>* root)
149 {
150   if (root != NULL)
151   {
152     insert(root->element);
153     copy(root->left);
154     copy(root->right);
155   }
156 }
157
158 /* Destructor */
159 template <typename T>
160 BST<T>::~BST()
161 {
162   clear();
163 }
164
165 /* Return true if the element is in the tree */
166 template <typename T>
167 bool BST<T>::search(T element) const
168 {
169   TreeNode<T>* current = root; // Start from the root
170
171   while (current != NULL)
172     if (element < current->element)
173     {
174       current = current->left; // Go left
175     }
176     else if (element > current->element)
177     {
178       current = current->right; // Go right
179     }
180     else // Element matches current.element
181       return true; // Element is found
182
183   return false; // Element is not in the tree
184 }
185
186 template <typename T>
187 TreeNode<T>*  BST<T>::createNewNode(T element)
188 {
189   return new TreeNode<T>(element);
190 }
191
192 /* Insert element into the binary tree
```

```
193   * Return true if the element is inserted successfully
194   * Return false if the element is already in the list
195   */
196  template <typename T>
197  bool BST<T>::insert(T element)
198  {
199    if (root == NULL)
200      root = createNewNode(element); // Create a new root
201    else
202    {
203      // Locate the parent node
204      TreeNode<T>* parent = NULL;
205      TreeNode<T>* current = root;
206      while (current != NULL)
207        if (element < current->element)
208        {
209          parent = current;
210          current = current->left;
211        }
212        else if (element > current->element)
213        {
214          parent = current;
215          current = current->right;
216        }
217        else
218          return false; // Duplicate node not inserted
219
220      // Create the new node and attach it to the parent node
221      if (element < parent->element)
222        parent->left = createNewNode(element);
223      else
224        parent->right = createNewNode(element);
225    }
226
227    size++;
228    return true; // Element inserted
229  }
230
231  /* Inorder traversal */
232  template <typename T>
233  void BST<T>::inorder() const
234  {
235    inorder(root);
236  }
237
238  /* Inorder traversal from a subtree */
239  template <typename T>
240  void BST<T>::inorder(TreeNode<T>* root) const
241  {
242    if (root == NULL) return;
243    inorder(root->left);
244    cout << root->element << " ";
245    inorder(root->right);
246  }
247
248  /* Postorder traversal */
249  template <typename T>
250  void BST<T>::postorder() const
```

```
251  {
252    postorder(root);
253  }
254
255  /** Inorder traversal from a subtree */
256  template <typename T>
257  void BST<T>::postorder(TreeNode<T>* root) const
258  {
259    if (root == NULL) return;
260    postorder(root->left);
261    postorder(root->right);
262    cout << root->element << " ";
263  }
264
265  /* Preorder traversal */
266  template <typename T>
267  void BST<T>::preorder() const
268  {
269    preorder(root);
270  }
271
272  /* Preorder traversal from a subtree */
273  template <typename T>
274  void BST<T>::preorder(TreeNode<T>* root) const
275  {
276    if (root == NULL) return;
277    cout << root->element << " ";
278    preorder(root->left);
279    preorder(root->right);
280  }
281
282  /* Get the number of nodes in the tree */
283  template <typename T>
284  int BST<T>::getSize() const
285  {
286    return size;
287  }
288
289  /* Remove all nodes from the tree */
290  template <typename T>
291  void BST<T>::clear()
292  {
293    // Left as exercise
294  }
295
296  /* Return a path from the root leading to the specified element */
297  template <typename T>
298  vector<TreeNode<T>*>* BST<T>::path(T element) const
299  {
300    vector<TreeNode<T>* >* v = new vector<TreeNode<T>* >();
301    TreeNode<T>* current = root;
302
303    while (current != NULL)
304    {
305      v->push_back(current);
306      if (element < current->element)
307        current = current->left;
308      else if (element > current->element)
```

```
309       current = current->right;
310     else
311       break;
312   }
313
314   return v;
315 }
316
317 /* Delete an element from the binary tree.
318  * Return true if the element is deleted successfully
319  * Return false if the element is not in the tree */
320 template <typename T>
321 bool BST<T>::remove(T element)
322 {
323   // Locate the node to be deleted and also locate its parent node
324   TreeNode<T>* parent = NULL;
325   TreeNode<T>* current = root;
326   while (current != NULL)
327   {
328     if (element < current->element)
329     {
330       parent = current;
331       current = current->left;
332     }
333     else if (element > current->element)
334     {
335       parent = current;
336       current = current->right;
337     }
338     else
339       break; // Element is in the tree pointed by current
340   }
341
342   if (current == NULL)
343     return false; // Element is not in the tree
344
345   // Case 1: current has no left children
346   if (current->left == NULL)
347   {
348     // Connect the parent with the right child of the current node
349     if (parent == NULL)
350     {
351       root = current->right;
352     }
353     else
354     {
355       if (element < parent->element)
356         parent->left = current->right;
357       else
358         parent->right = current->right;
359     }
360
361     delete current; // Delete current
362   }
363   else
364   {
365     // Case 2: The current node has a left child
366     // Locate the rightmost node in the left subtree of
```

```
367       // the current node and also its parent
368       TreeNode<T>* parentOfRightMost = current;
369       TreeNode<T>* rightMost = current->left;
370
371       while (rightMost->right != NULL)
372       {
373         parentOfRightMost = rightMost;
374         rightMost = rightMost->right; // Keep going to the right
375       }
376
377       // Replace the element in current by the element in rightMost
378       current->element = rightMost->element;
379
380       // Eliminate rightmost node
381       if (parentOfRightMost->right == rightMost)
382         parentOfRightMost->right = rightMost->left;
383       else
384         // Special case: parentOfRightMost->right == current
385         parentOfRightMost->left = rightMost->left;
386
387       delete rightMost; // Delete rightMost
388     }
389
390     size--;
391     return true; // Element inserted
392   }
393
394   #endif
```

A binary tree contains nodes defined in the **`TreeNode`** class (lines 8-22). You can obtain iterators for traversing the elements in a binary tree. The **`Iterator`** class (lines 24–75) will be discussed in §21.4.

The header of the **`BST`** class is defined in lines 77–116. Five private functions are defined in lines 110–115. These are supporting functions, which are used only for implementing the public functions. The data fields **`root`** and **`size`** (lines 106–107) are declared protected so they can be directly accessed from the child classes. Later, in Chapter 26, we will define the **`AVLTree`** class, which is derived from **`BST`**. The **`createNewNode`** function creates a new node (line 108). The function is defined virtual, because it will be redefined in the **`AVLTree`** class to create a new type of node. The **`insert`** and **`remove`** functions (lines 86–87) are also defined **`virtual`**, because they will be redefined the **`AVLTree`** class, so that they can be invoked dynamically for different types of trees.

The no-arg constructor (lines 118–123) constructs an empty binary tree with **`root NULL`** and **`size 0`**. The constructor (lines 125–135) constructs a binary tree initialized with the elements in the array.

The copy constructor (lines 138–144) creates a new binary tree by copying the contents from an existing tree. This is done by recursively inserting the elements from the existing tree to the new tree using the copy function (lines 147–156).

The destructor (lines 159–1631) removes all nodes from the tree.

The **`search(T element)`** function (lines 166–184) searches an element in the BST. It returns **`true`** (line 181) if an element is found, otherwise, it returns **`false`** (line 183).

The **`insert(T element)`** function (lines 196–229) creates a node for element and inserts it into the tree. If the tree is empty, the node becomes the root (line 200). Otherwise, the function finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the function returns **`false`** (line 218); otherwise it returns **`true`** (line 228).

The **`inorder()`** function (lines 232–236) invokes **`inorder(root)`** to traverse the entire tree. The function **`inorder(TreeNode root)`** traverses the tree with the specified root. This is a recursive function. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The **`postorder()`** function (lines 249–253) and the **`preorder()`** function (lines 266–270) are implemented similarly using recursion.

The **`path(T element)`** function (lines 297–315) finds a path of nodes from the root leading to the node that contains the element, or the parent node under which the element will be inserted.

Listing 21.4 gives an example that uses some of the functions in **`BST`**.

`Listing 21.4 TestBST.cpp`

```
1  #include <iostream>
```

```
 2  #include <vector>
 3  #include <string>
 4  #include "BST.h"
 5  using namespace std;
 6
 7  int main()
 8  {
 9    BST<string> tree;
10    tree.insert("George");
11    tree.insert("Michael");
12    tree.insert("Tom");
13    tree.insert("Adam");
14    tree.insert("Jones");
15    tree.insert("Peter");
16    tree.insert("Daniel");
17
18    cout << "Inorder (sorted): ";
19    tree.inorder();
20
21    cout << "\nPostorder: ";
22    tree.postorder();
23
24    cout << "\nPreorder: ";
25    tree.preorder();
26
27    cout << "\nThe number of nodes is " << tree.getSize() << endl;
28    cout << "search(\"Jones\") " << tree.search("Jones") << endl;
29    cout << "search(\"John\") " << tree.search("John") << endl;
30
31    cout << "A path from the root to Peter is: ";
32    vector<TreeNode<string>* > *v = tree.path("Peter");
33    for (unsigned i = 0; i < (*v).size(); i++)
34      cout << (*v)[i]->element << " ";
35
36    int numbers[] = {2, 4, 3, 1, 8, 5, 6, 7};
37    BST<int> intTree(numbers, 8);
38    cout << "\nInorder (sorted): ";
39    intTree.inorder();
40
41    return 0;
42  }
```

Sample output

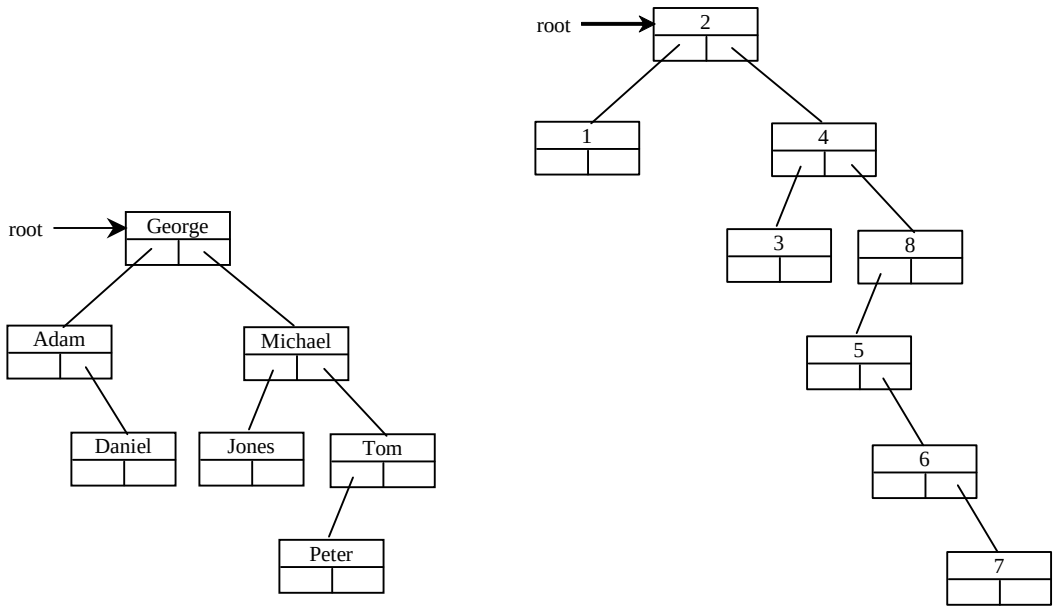
```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7
search("Jones") 1
search("John") 0
A path from the root to Peter is: George Michael Tom Peter
Inorder (sorted): 1 2 3 4 5 6 7 8
```

The program creates a binary tree for strings using **`BST<string>`** (line 9). The program adds strings into the binary tree (lines 10–16) and displays the elements in inorder (line 19), postorder (line 22), and preorder

(line 25). The **`getSize`** and **`search`** functions are invoked in lines 27–29. Lines 31–34 display a path from the root to **`Peter`**.

The program creates a binary tree for integers from an array of integers (line 36) and displays the numbers in increasing order (line 39).

After all the string elements are inserted, **`tree`** should appear as shown in Figure 21.8a. Tree **`intTree`** is created as shown in Figure 21.8b.



(a) (b)

Figure 21.8
The BSTs are pictured here after they are created.

If the elements are inserted in a different order, the tree will look different. However, the inorder traversal prints elements in the same order as long as the set of elements is the same. The inorder traversal displays a sorted list.

Check point

21.1 If a set of the same elements is inserted into a binary tree in two different orders, will the two corresponding binary trees look the same? Will the inorder traversal be the same? Will the postorder traversal be the same? Will the preorder traversal be the same?

21.2 What is wrong if the two highlighted lines are deleted in the following constructor for **BST**?

```
template<typename T>
BST<T>::BST(T elements[], int arraySize)
{
  root = NULL;
  size = 0;
  for (int i = 0; i < arraySize; i++)
  {
    insert(elements[i]);
  }
}
```

21.3 Add the elements **4**, **5**, **1**, **2**, **9**, **3** into a binary tree in this order. Draw the diagrams to show the binary tree as each element is added.

21.4 Show the inorder, postorder, preorder, and breadth-first traversals for the binary tree in Figure 21.1.

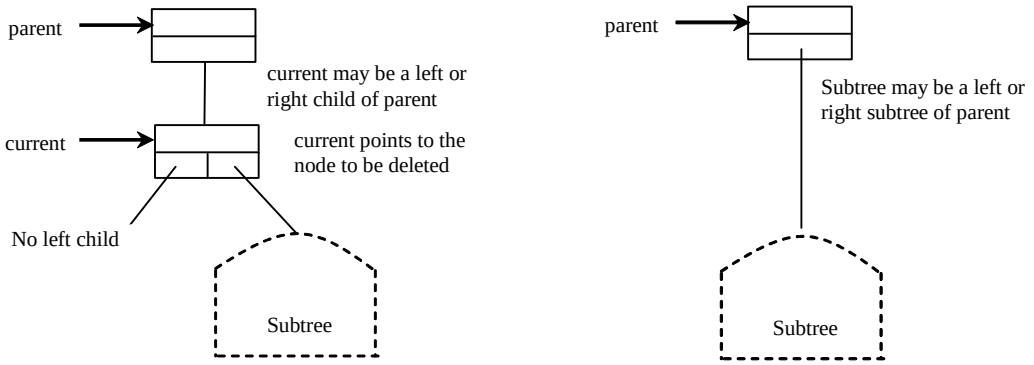
21.3 Deleting Elements in a BST

Key Point: *To delete an element from a BST, first locate it in the tree and then consider two cases—whether or not the node has a left child—before deleting the element and reconnecting the tree.*

The **`insert(element)`** function was presented in §21.2.4 to add an element to a binary tree. Often you need to delete an element from a binary tree. Deleting an element from a binary tree is far more complex than adding an element into a binary tree.

To delete an element from a binary tree, you need to first locate the node that contains the element and also its parent node. 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. Consider two cases:

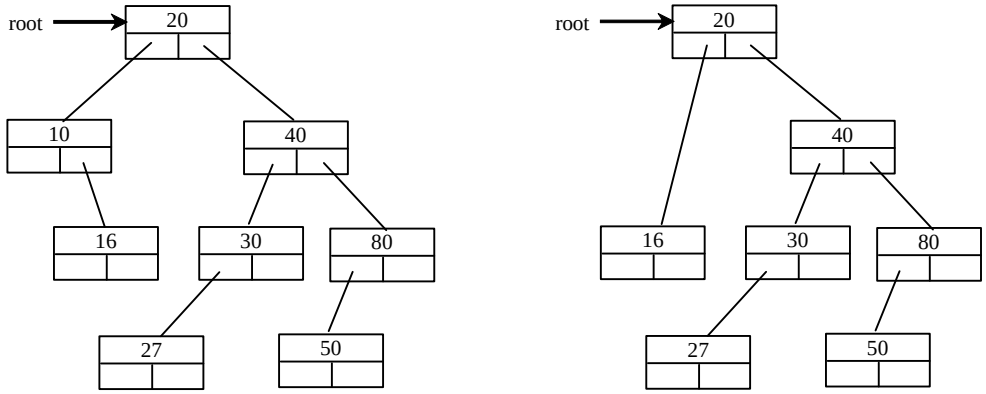
Case 1: The current node does not have a left child, as shown in Figure 21.9a. Simply connect the parent with the right child of the current node, as shown in Figure 21.9b.



(a) **(b)**

Figure 21.9
Case 1: the current node has no left child.

For example, to delete node **10** in Figure 21.10a. Connect the parent of node **10** with the right child of node **10**, as shown in Figure 21.10b.



(a) **(b)**

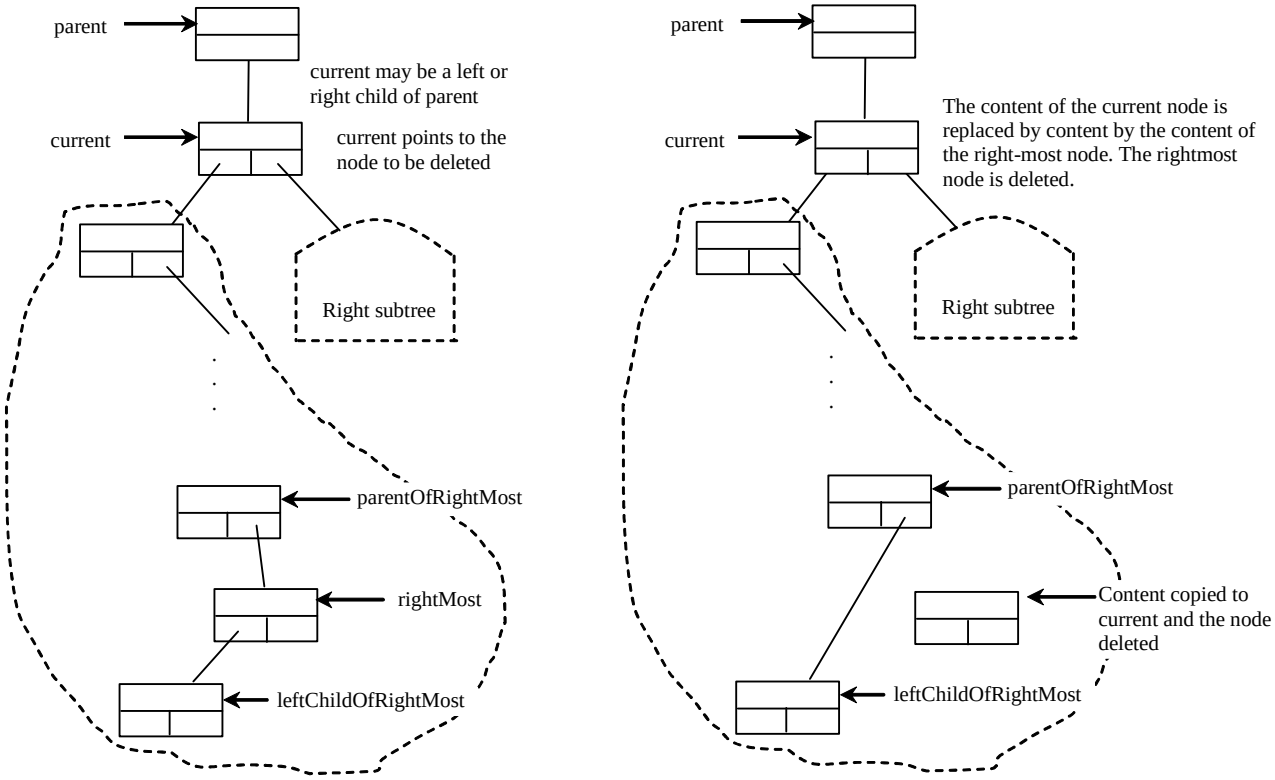
Figure 21.10
*Case 1: deleting node **10** from (a) results in (b).*

NOTE

If the current node is a leaf, it falls into Case 1. For example, to delete element **16** in Figure 21.10a, connect its right child to the parent of node **16**. In this case, the right child of node **16** is **NULL**.

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. Note that 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.



(a) (b)

Figure 21.11
Case 2: the current node has a left child.

For example, consider deleting node **20** in Figure 21.12a. The **rightMost** node has the element value **16**. Replace the element value **20** with **16** in the **current** node and make node **10** the parent for node **14**, as shown in Figure 21.12b.

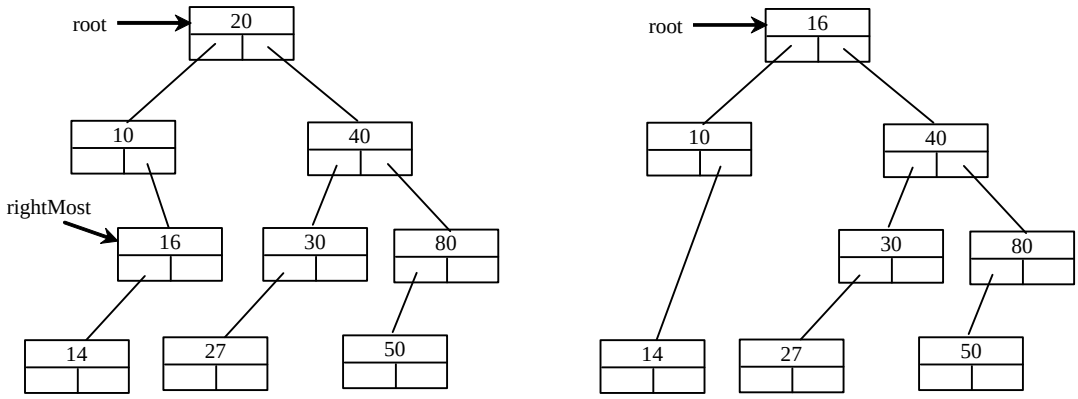


Figure 21.12
*Case 2: deleting node **20** from (a) results in (b).*

NOTE

If the left child of **current** does not have a right child, **current->left** points to the large element in the left subtree of **current**. In this case, **rightMost** is **current->left** and **parentOfRightMost** is **current**. You have to take care of this special case to reconnect the right child of **rightMost** with **parentOfRightMost**.

The algorithm for deleting an element from a binary tree is described in Listing 21.5.

Listing 21.5 Deleting an Element from a Binary Tree

```
bool remove(T e)
{
  Locate element e in the tree;
  if element e is not found
    return false;

  Let current be the node that contains e and parent be
    the parent of current;

  if (current has no left child) // Case 1
    Connect the right child of
      current with parent; now current is not referenced, so
    it is eliminated;
  else // Case 2
    Locate the rightmost node in the left subtree of current.
    Copy the element value in the rightmost node to current.
    Connect the parent of the rightmost to the left child
      of rightmost; Delete rightMost.

  return true; // Element deleted
```

```
}
```

You cannot name the function **delete** because **delete** is a C++ keyword.

The complete implementation of the **remove** function is given in lines 320–392 in Listing 21.3. The function locates the node (named **current**) to be deleted and also locates its parent (named **parent**) in lines 312–328. If **current** is **NULL** (line 342), the element is not in the tree. So, the funciton returns **false** (line 343). Please note that if **current** is **root**, **parent** is **NULL**. If the tree is empty, both **current** and **parent** are **NULL**.

Case 1 of the algorithm is covered in lines 346–362. In this case, the **current** node has no left child (i.e., **current->left** is **NULL**). If **parent** is **NULL**, assign **current->right** to **root** (lines 349–352). Otherwise, assign **current->right** to **parent->left** or **parent->right**, depending on whether **current** is a left or right child of **parent** (354–359).

Case 2 of the algorithm is covered in lines 352–376. In this case, **current** has a left child. The algorithm locates the rightmost node (named **rightMost**) in the left subtree of the current node and also its parent (named **parentOfRightMost**) (lines 371–375). Replace the element in **current** by the element in **rightMost** (line 378); assign **rightMost->left** to **parentOfRightMost->right** or **parentOfRightMost->right** (lines 381–385), depending on whether **rightMost** is a right or left child of **parentOfRightMost**.

Listing 21.6 is a test program that deletes the elements from the binary tree.

Listing 21.6 TestBSTDelete.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include "BST.h"
5  using namespace std;
6
7  template <typename T>
8  void printTree(const BST<T>& tree)
```

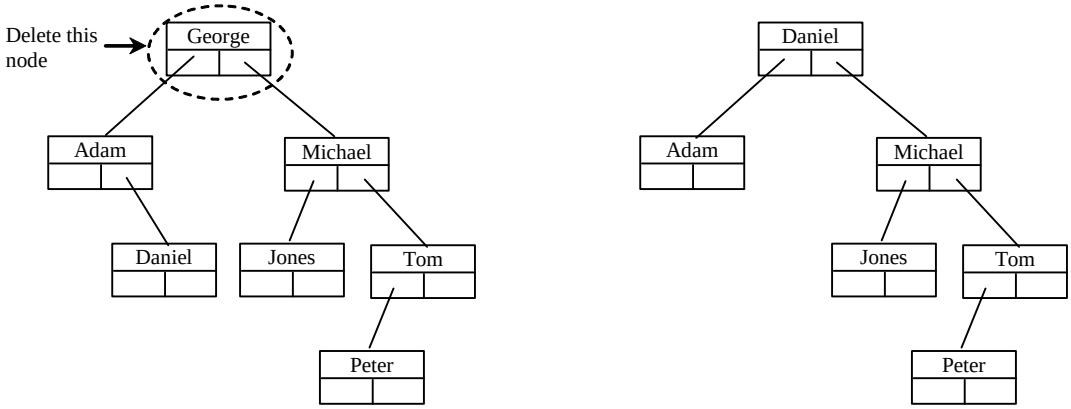
```
 9  {
10    // Traverse tree
11    cout << "Inorder (sorted): ";
12    tree.inorder();
13    cout << "\nPostorder: ";
14    tree.postorder();
15    cout << "\nPreorder: ";
16    tree.preorder();
17    cout << "\nThe number of nodes is " << tree.getSize() << endl;
18  }
19
20  int main()
21  {
22    BST<string> tree;
23    tree.insert("George");
24    tree.insert("Michael");
25    tree.insert("Tom");
26    tree.insert("Adam");
27    tree.insert("Jones");
28    tree.insert("Peter");
29    tree.insert("Daniel");
30    printTree(tree);
31
32    cout << "\nAfter delete George:";
33    tree.remove("George");
34    printTree(tree);
35
36    cout << "\nAfter delete Adam:";
37    tree.remove("Adam");
38    printTree(tree);
39
40    cout << "\nAfter delete Michael:";
41    tree.remove("Michael");
42    printTree(tree);
43
44    return 0;
45  }
```

Sample output

```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7

After delete George:
Inorder (sorted): Adam Daniel Jones Michael Peter Tom
Postorder: Adam Jones Peter Tom Michael Daniel
Preorder: Daniel Adam Michael Jones Tom Peter
The number of nodes is 6

After delete Adam:
Inorder (sorted): Daniel Jones Michael Peter Tom
Postorder: Jones Peter Tom Michael Daniel
Preorder: Daniel Michael Jones Tom Peter
The number of nodes is 5

After delete Michael:
Inorder (sorted): Daniel Jones Peter Tom
```

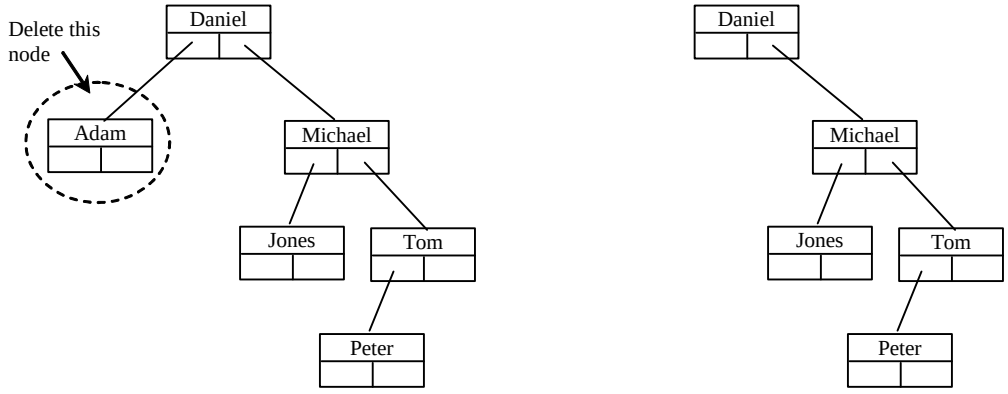
```
Postorder: Peter Tom Jones Daniel
Preorder: Daniel Jones Tom Peter
The number of nodes is 4
```

Figures 21.13–21.15 show how the tree evolves as the elements are deleted from it.



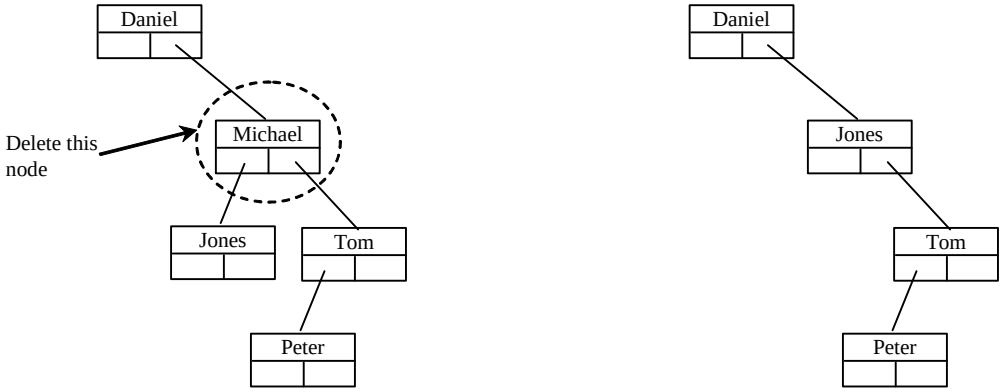
(a) Deleting George (b) After George is deleted

Figure 21.13
Deleting George falls in Case 2.



(a) Deleting Adam (b) After Adam is deleted

Figure 21.14
Deleting Adam falls in Case 1.



(a) Deleting Michael (b) After Michael is deleted

Figure 21.15
Deleting Michael falls in Case 2.

NOTE:

It is obvious that the time complexity for the inorder, preorder, and postorder is $O(n)$, since each node is traversed only once. The time complexity for search, insertion and deletion is the height of the tree. In the worst case, the height of the tree is $O(n)$.

Consider the following scenarios.

1. What will happen if you forget to delete the discarded node? Your program will continue to run, but it will suffer memory leak.
2. Does the program work if the **`rightMost`** node is a leaf? Yes. In this case, **`rightMost->left`** is **`NULL`**, which is assigned to **`parentOfRightMost->right`**.

Check point

21.5 Show the change of the binary tree as elements **`60`** and **`55`** are deleted from the tree in this order Figure 21.1.

21.4 Iterators for `BST`

Key Point: *You can use iterators to traverse a binary tree.*

§20.5 introduced iterators and defined an **`Iterator`** class for **`LinkedList`**. Iterators are very useful to provide a uniformed way for traversing the elements in a container. Every iterator class has the same pattern, so they are very similar. This section defines an **`Iterator`** class for traversing the elements in **`BST`**, as shown in Figure 21.16.

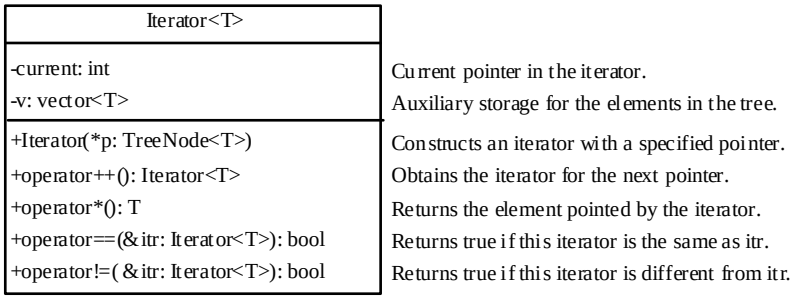


Figure 21.16
*The class defines an iterator for accessing the elements in **`BST`**.*

This class is implemented in lines 24–75 in Listing 21.3. For convenience, the constructors and functions are implemented as inline functions.

The **`Iterator`** class has two data fields, **`vector`** and **`current`**. The data field **`vector`** (line 65) is used as an auxiliary storage for the elements in the binary tree. The data field **`current`** points to the current element in vector (line 64).

The constructor (lines 28–38) creates an iterator. Invoking the **`treeToVector(p)`** function (line 35) stores all the elements in the tree rooted at **`p`** into the **`vector`** (lines 66–73). The data field **`current`** points to the current element in the vector. Initially, it is set to **`0`** (line 36).

The **`operator++()`** function moves **`current`** to point to the next element in the vector (line 42). **`current`** is set to **`-1`** if the pointer is moved past the last element in the vector (line 44).

To obtain iterators from a **`BST`**, the following two functions are defined and implemented in lines 95–103 in Listing 21.3.

```
Iterator<T> begin() const;
```

```
Iterator<T> end() const;
```

The **`begin()`** function returns the iterator for the first element to be traversed and the **`end()`** function returns the iterator that pasts the last element in the list.

Listing 21.7 gives an example that uses iterators to traverse the string elements in a tree and displays the strings in uppercase. The program creates a **`BST`** for strings in line 9, adds four strings to the list (lines 12–15), and traverses all the elements in the list using iterators and displays them in uppercase (lines 18–22).

Listing 21.7 TestBSTIterator.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include "BST.h"
 4  using namespace std;
 5
 6  string toUpperCase(string& s)
 7  {
 8    for (int i = 0; i < s.length(); i++)
 9      s[i] = toupper(s[i]);
10
11    return s;
12  }
13
14  int main()
15  {
16    // Create a binary search tree for strings
17    BST<string> tree;
18
19    // Add elements to the tree
20    tree.insert("America");
21    tree.insert("Canada");
22    tree.insert("Russia");
23    tree.insert("France");
24
25    // Traverse a binary tree using iterators
26    for (Iterator<string> iterator = tree.begin();
27      iterator != tree.end(); iterator++)
28    {
29      cout << toUpperCase(*iterator) << " ";
30    }
31
32    return 0;
33  }
```

Sample output

```
AMERICA CANADA FRANCE RUSSIA
```

Key Terms

- **binary search tree**
- **binary tree**
- **inorder traversal**
- **postorder traversal**
- **preorder traversal**
- **tree traversal**

Chapter Summary

- A binary tree can be implemented using linked nodes. Each node contains the element value and two pointers that point to the left and right children.
- Tree traversal is the process of visiting each node in the tree exactly once in a certain order. There are several ways to traverse a tree.
- With inorder traversal, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node. The inorder traversal displays all the nodes in a binary search tree in increasing order.
- With postorder traversal, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself.
- With preorder traversal, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. Depth-first traversal is the same as preorder traversal.
- With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

Quiz

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

Programming Exercises

Section 21.2

21.1* (*Search in **BST***) Add a function in **BST** to search an element in the tree.

```
// Search element in this binary tree
bool search(T element)
```

21.2* (*Breadth-first traversal in **BST***) Add a function in **BST** to traverse the tree in breadth-first order.

```
// Display the nodes in breadth-first traversal
```

```
void breadthFirstTraversal()
```

21.3* (*Depth of **BST***) Add a function in **BST** to return the depth of the tree.

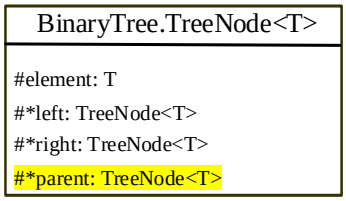
```
// Return the height of this binary tree. Height is the
// number of nodes in the longest path from the root
int height()
```

21.4** (*Implement inorder traversal using a stack*) Implement the **inorder** function in **BST** using a stack instead of recursion.

21.5** (*Implement preorder traversal using a stack*) Implement the **preorder** function in **BST** using a stack instead of recursion.

21.6** (*Implement postorder traversal using a stack*) Implement the **postorder** function in **BST** using a stack instead of recursion.

21.7*** (*Parent reference for **BST***) Revise **TreeNode** by adding a reference to a node's parent, as shown below:



Revise the **BST** class to ensure that each node has the proper parent link. Add the following function in **BST**:

```
// Returns the parent for the specified node
TreeNode<T>*  getParent(TreeNode<T> node)
```

Write a test program that adds numbers **1**, **2**, ..., **100** to the tree and displays the paths for all leaf nodes.

21.8* (*Find the leaves*) Add a function in **BST** to return the number of leaves in the tree:

```
// Returns the number of leaf nodes
int getNumberofLeaves()
```

21.9* (*Find the nonleaves*) Add a function in the **BST** class to return the number of the nonleaves as follows:

```
// Returns the number of non-leaf nodes
int getNumberofNonLeaves()
```

21.10* (*Test full binary tree*) A full binary tree is a binary tree with the leaves on the same level. Add a function in the **BST** class to return true if the tree is full. (Hint: the number of nodes in a full binary tree is $2^{depth} - 1$)

```
// Returns true if the tree is a full binary tree
bool isFullBST()
```