Lecture 29:
Data Structures

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 17th, 2010

20.6 Queues (cont.)

- The program of Figs. 20.16–20.17 creates a `Queue` class template (Fig. 20.16) through `private` inheritance (line 9) of the `List` class template (Fig. 20.4).
- The `Queue` has member functions `enqueue` (lines 13–16), `dequeue` (lines 19–22), `isQueueEmpty` (lines 25–28) and `printQueue` (lines 31–34).
- These are essentially the `insertAtBack`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template.

20.6 Queues (cont.)

- The `List` class template contains other member functions that we do not want to make accessible through the `public` interface to the `Queue` class.
- So when we indicate that the `Queue` class template is to inherit the `List` class template, we specify `private` inheritance.
- This makes all the `List` class template's member functions `private` in the `Queue` class template.
- When we implement the `Queue`'s member functions, we have each of these call the appropriate member function of the list class—`enqueue` calls `insertAtBack` (line 15), `dequeue` calls `removeFromFront` (line 21), `isQueueEmpty` calls `isEmpty` (line 27) and `printQueue` calls `print` (line 33).
- As with the `Stack` example in Fig. 20.13, this delegation requires explicit use of the `this` pointer in `isQueueEmpty` and `printQueue` to avoid compilation errors.

```
1  // Fig. 20.16: Queue.h
2  // Template Queue class definition derived from class List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // List class definition
7
8  template< typename QUEUETYPE >
9  class Queue : private List< QUEUETYPE >
10 {
11 public:
12    // enqueue calls List member function insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16    } // end function enqueue
17
18    // dequeue calls List member function removeFromFront
19    bool dequeue( QUEUETYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function dequeue
23
```

Fig. 20.16 | Queue class-template definition. (Part 1 of 2.)

```
24      // isQueueEmpty calls List member function isEmpty
25      bool isQueueEmpty() const
26      {
27         return this->isEmpty();
28      } // end function isQueueEmpty
29
30      // printQueue calls List member function print
31      void printQueue() const
32      {
33         this->print();
34      } // end function printQueue
35   }; // end class Queue
36
37   #endif
```

Fig. 20.16 | `Queue` class-template definition. (Part 2 of 2.)

```
1  // Fig. 20.17: Fig21_17.cpp
2  // Template Queue class test program.
3  #include <iostream>
4  #include "Queue.h" // Queue class definition
5  using namespace std;
6
7  int main()
8  {
9     Queue< int > intQueue; // create Queue of integers
10
11    cout << "processing an integer Queue" << endl;
12
13    // enqueue integers onto intQueue
14    for ( int i = 0; i < 3; i++ )
15    {
16       intQueue.enqueue( i );
17       intQueue.printQueue();
18    } // end for
19
```

Fig. 20.17 | Queue-processing program. (Part 1 of 5.)

```
20      int dequeueInteger; // store dequeued integer
21
22      // dequeue integers from intQueue
23      while ( !intQueue.isQueueEmpty() )
24      {
25         intQueue.dequeue( dequeueInteger );
26         cout << dequeueInteger << " dequeued" << endl;
27         intQueue.printQueue();
28      } // end while
29
30      Queue< double > doubleQueue; // create Queue of doubles
31      double value = 1.1;
32
33      cout << "processing a double Queue" << endl;
34
35      // enqueue floating-point values onto doubleQueue
36      for ( int j = 0; j < 3; j++ )
37      {
38         doubleQueue.enqueue( value );
39         doubleQueue.printQueue();
40         value += 1.1;
41      } // end for
42
```

Fig. 20.17 | Queue-processing program. (Part 2 of 5.)

```
43      double dequeueDouble; // store dequeued double
44
45      // dequeue floating-point values from doubleQueue
46      while ( !doubleQueue.isQueueEmpty() )
47      {
48         doubleQueue.dequeue( dequeueDouble );
49         cout << dequeueDouble << " dequeued" << endl;
50         doubleQueue.printQueue();
51      } // end while
52   } // end main
```

Fig. 20.17 | Queue-processing program. (Part 3 of 5.)

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3
```

Fig. 20.17 | Queue-processing program. (Part 4 of 5.)

```
2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

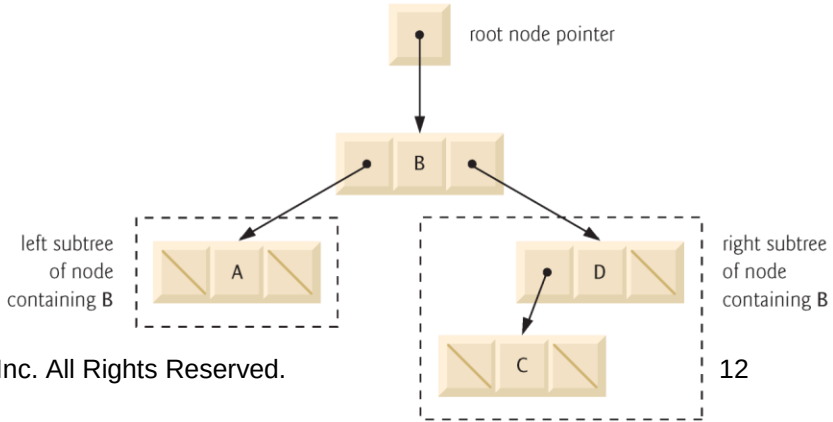
Fig. 20.17 | Queue-processing program. (Part 5 of 5.)

20.7 Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.
- This section discusses binary trees (Fig. 20.18)—trees whose nodes all contain two links (none, one or both of which may be null).

20.7 Trees (cont.)

- For this discussion, refer to nodes A, B, C and D in Fig. 20.18.
- The root node (node B) is the first node in a tree.
- Each link in the root node refers to a child (nodes A and D).
- The left child (node A) is the root node of the left subtree (which contains only node A), and the right child (node D) is the root node of the right subtree (which contains nodes D and C).
- The children of a given node are called siblings (e.g., nodes A and D are siblings).
- A node with no children is a leaf node (e.g., nodes A and C are leaf nodes).
- Computer scientists normally draw trees from the root node down—the opposite of how trees grow in nature.



20.7 Trees (cont.)

- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
- Figure 20.19 illustrates a binary search tree with 9 values.
- Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

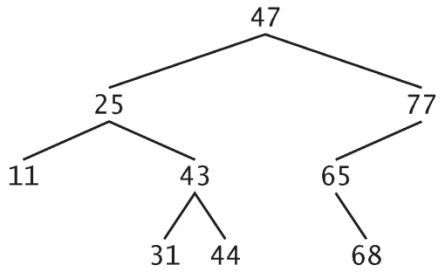


Fig. 20.19 | A binary search tree.

20.7 Trees (cont.)

- The program of Figs. 20.20–20.22 creates a binary search tree and traverses it (i.e., walks through all its nodes) three ways—using recursive inorder, preorder and postorder traversals.

```
1  // Fig. 20.20: TreeNode.h
2  // Template TreeNode class definition.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // forward declaration of class Tree
7  template< typename NODETYPE > class Tree;
8
9  // TreeNode class-template definition
10 template< typename NODETYPE >
11 class TreeNode
12 {
13    friend class Tree< NODETYPE >;
14 public:
15    // constructor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( 0 ), // pointer to left subtree
18         data( d ), // tree node data
19         rightPtr( 0 ) // pointer to right substree
20    {
21       // empty body
22    } // end TreeNode constructor
23
```

Fig. 20.20 | `TreeNode` class-template definition. (Part 1 of 2.)

```
24       // return copy of node's data
25       NODETYPE getData() const
26       {
27          return data;
28       } // end getData function
29    private:
30       TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
31       NODETYPE data;
32       TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
33    }; // end class TreeNode
34
35    #endif
```

Fig. 20.20 | `TreeNode` class-template definition. (Part 2 of 2.)

```
1  // Fig. 20.21: Tree.h
2  // Template Tree class definition.
3  #ifndef TREE_H
4  #define TREE_H
5
6  #include <iostream>
7  #include "TreeNode.h"
8  using namespace std;
9
10 // Tree class-template definition
11 template< typename NODETYPE > class Tree
12 {
13 public:
14    Tree(); // constructor
15    void insertNode( const NODETYPE & );
16    void preOrderTraversal() const;
17    void inOrderTraversal() const;
18    void postOrderTraversal() const;
```

Fig. 20.21 | `Tree` class-template definition. (Part 1 of 6.)

```
19  private:
20     TreeNode< NODETYPE > *rootPtr;
21
22     // utility functions
23     void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE & );
24     void preOrderHelper( TreeNode< NODETYPE > * ) const;
25     void inOrderHelper( TreeNode< NODETYPE > * ) const;
26     void postOrderHelper( TreeNode< NODETYPE > * ) const;
27  }; // end class Tree
28
29  // constructor
30  template< typename NODETYPE >
31  Tree< NODETYPE >::Tree()
32  {
33     rootPtr = 0; // indicate tree is initially empty
34  } // end Tree constructor
35
36  // insert node in Tree
37  template< typename NODETYPE >
38  void Tree< NODETYPE >::insertNode( const NODETYPE &value )
39  {
40     insertNodeHelper( &rootPtr, value );
41  } // end function insertNode
42
```

Fig. 20.21 | Tree class-template definition. (Part 2 of 6.)

```
43  // utility function called by insertNode; receives a pointer
44  // to a pointer so that the function can modify pointer's value
45  template< typename NODETYPE >
46  void Tree< NODETYPE >::insertNodeHelper(
47     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
48  {
49     // subtree is empty; create new TreeNode containing value
50     if ( *ptr == 0 )
51        *ptr = new TreeNode< NODETYPE >( value );
52     else // subtree is not empty
53     {
54        // data to insert is less than data in current node
55        if ( value < ( *ptr )->data )
56           insertNodeHelper( &( ( *ptr )->leftPtr ), value );
57        else
58        {
59           // data to insert is greater than data in current node
60           if ( value > ( *ptr )->data )
61              insertNodeHelper( &( ( *ptr )->rightPtr ), value );
62           else // duplicate data value ignored
63              cout << value << " dup" << endl;
64        } // end else
65     } // end else
66  } // end function insertNodeHelper
```

Fig. 20.21 | Tree class-template definition. (Part 3 of 6.)

```
67
68  // begin preorder traversal of Tree
69  template< typename NODETYPE >
70  void Tree< NODETYPE >::preOrderTraversal() const
71  {
72     preOrderHelper( rootPtr );
73  } // end function preOrderTraversal
74
75  // utility function to perform preorder traversal of Tree
76  template< typename NODETYPE >
77  void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *ptr ) const
78  {
79     if ( ptr != 0 )
80     {
81        cout << ptr->data << ' '; // process node
82        preOrderHelper( ptr->leftPtr ); // traverse left subtree
83        preOrderHelper( ptr->rightPtr ); // traverse right subtree
84     } // end if
85  } // end function preOrderHelper
86
```

Fig. 20.21 | `Tree` class-template definition. (Part 4 of 6.)

```
87  // begin inorder traversal of Tree
88  template< typename NODETYPE >
89  void Tree< NODETYPE >::inOrderTraversal() const
90  {
91     inOrderHelper( rootPtr );
92  } // end function inOrderTraversal
93
94  // utility function to perform inorder traversal of Tree
95  template< typename NODETYPE >
96  void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr ) const
97  {
98     if ( ptr != 0 )
99     {
100       inOrderHelper( ptr->leftPtr ); // traverse left subtree
101       cout << ptr->data << ' '; // process node
102       inOrderHelper( ptr->rightPtr ); // traverse right subtree
103    } // end if
104 } // end function inOrderHelper
105
```

Fig. 20.21 | Tree class-template definition. (Part 5 of 6.)

```
106 // begin postorder traversal of Tree
107 template< typename NODETYPE >
108 void Tree< NODETYPE >::postOrderTraversal() const
109 {
110    postOrderHelper( rootPtr );
111 } // end function postOrderTraversal
112
113 // utility function to perform postorder traversal of Tree
114 template< typename NODETYPE >
115 void Tree< NODETYPE >::postOrderHelper(
116    TreeNode< NODETYPE > *ptr ) const
117 {
118    if ( ptr != 0 )
119    {
120       postOrderHelper( ptr->leftPtr ); // traverse left subtree
121       postOrderHelper( ptr->rightPtr ); // traverse right subtree
122       cout << ptr->data << ' '; // process node
123    } // end if
124 } // end function postOrderHelper
125
126 #endif
```

Fig. 20.21 | `Tree` class-template definition. (Part 6 of 6.)

```
1  // Fig. 20.22: Fig21_22.cpp
2  // Tree class test program.
3  #include <iostream>
4  #include <iomanip>
5  #include "Tree.h" // Tree class definition
6  using namespace std;
7
8  int main()
9  {
10    Tree< int > intTree; // create Tree of int values
11    int intValue;
12
13    cout << "Enter 10 integer values:\n";
14
15    // insert 10 integers to intTree
16    for ( int i = 0; i < 10; i++ )
17    {
18       cin >> intValue;
19       intTree.insertNode( intValue );
20    } // end for
21
22    cout << "\nPreorder traversal\n";
23    intTree.preOrderTraversal();
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 1 of 4.)

```
24
25      cout << "\nInorder traversal\n";
26      intTree.inOrderTraversal();
27
28      cout << "\nPostorder traversal\n";
29      intTree.postOrderTraversal();
30
31      Tree< double > doubleTree; // create Tree of double values
32      double doubleValue;
33
34      cout << fixed << setprecision( 1 )
35         << "\n\n\nEnter 10 double values:\n";
36
37      // insert 10 doubles to doubleTree
38      for ( int j = 0; j < 10; j++ )
39      {
40         cin >> doubleValue;
41         doubleTree.insertNode( doubleValue );
42      } // end for
43
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 2 of 4.)

```
44      cout << "\nPreorder traversal\n";
45      doubleTree.preOrderTraversal();
46
47      cout << "\nInorder traversal\n";
48      doubleTree.inOrderTraversal();
49
50      cout << "\nPostorder traversal\n";
51      doubleTree.postOrderTraversal();
52      cout << endl;
53   } // end main
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 3 of 4.)

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 4 of 4.)

20.7 Trees (cont.)

- The `TreeNode` class template (Fig. 20.20) definition declares `Tree<NODETYPE>` as its `friend` (line 13).
  - This makes all member functions of a given specialization of class template `Tree` (Fig. 20.21) friends of the corresponding specialization of class template `TreeNode`, so they can access the `private` members of `TreeNode` objects of that type.
  - Because the `TreeNode` template parameter `NODETYPE` is used as the template argument for `Tree` in the `friend` declaration, `TreeNode`s specialized with a particular type can be processed only by a `Tree` specialized with the same type (e.g., a `Tree` of `int` values manages `TreeNode` objects that store `int` values).

20.7 Trees (cont.)

- Class template `Tree` (Fig. 20.21) has as `private` data `rootPtr` (line 20), a pointer to the tree's root node.
- Lines 15–18 declare the `public` member functions `insertNode` (that inserts a new node in the tree) and `preOrderTraversal`, `inOrderTraversal` and `postOrderTraversal`, each of which walks the tree in the designated manner.
- Each of these member functions calls its own recursive utility function to perform the appropriate operations on the internal representation of the tree, so the program is not required to access the underlying `private` data to perform these functions.
- Remember that the recursion requires us to pass in a pointer that represents the next subtree to process.

20.7 Trees (cont.)

- The `Tree` constructor initializes `rootPtr` to zero to indicate that the tree is initially empty.
- The `Tree` class's utility function `insertNodeHelper` (lines 45–66) is called by `insertNode` (lines 37–41) to recursively insert a node into the tree.
- *A node can only be inserted as a leaf node in a binary search tree.*
- If the tree is empty, a new `TreeNode` is created, initialized and inserted in the tree (lines 51–52).
- If the tree is not empty, the program compares the value to be inserted with the `data` value in the root node.
- If the insert value is smaller (line 55), the program recursively calls `insertNodeHelper` (line 56) to insert the value in the left subtree.
- If the insert value is larger (line 60), the program recursively calls `insertNodeHelper` (line 61) to insert the value in the right subtree.

20.7 Trees (cont.)

- If the value to be inserted is identical to the data value in the root node, the program prints the message `" dup"` (line 63) and returns without inserting the duplicate value into the tree.
- `insertNode` passes the address of `rootPtr` to `insertNodeHelper` (line 40) so it can modify the value stored in `rootPtr` (i.e., the address of the root node).
- To receive a pointer to `rootPtr` (which is also a pointer), `insertNodeHelper`'s first argument is declared as a pointer to a pointer to a `TreeNode`.

20.7 Trees (cont.)

- Member functions `inOrderTraversal` (lines 88–92), `preOrderTraversal` (lines 69–73) and `postOrderTraversal` (lines 107–111) traverse the tree and print the node values.
- For the purpose of the following discussion, we use the binary search tree in Fig. 20.23.

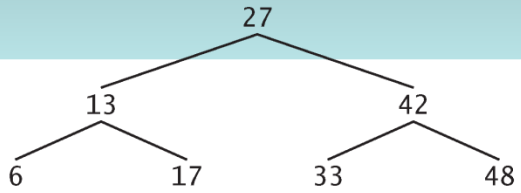


Fig. 20.23 | A binary search tree.

20.7 Trees (cont.)

- Function `inOrderTraversal` invokes utility function `inOrderHelper` to perform the inorder traversal of the binary tree.
- The steps for an inorder traversal are:
  - Traverse the left subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 100.)
  - Process the value in the node—i.e., print the node value (line 101).
  - Traverse the right subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 102.)
- The value in a node is not processed until the values in its left subtree are processed, because each call to `inOrderHelper` immediately calls `inOrderHelper` again with the pointer to the left subtree.

20.7 Trees (cont.)

- The inorder traversal of the tree in Fig. 20.23 is
  - `6 13 17 27 33 42 48`
- Note that the inorder traversal of a binary search tree prints the node values in ascending order.
- The process of creating a binary search tree actually sorts the data—thus, this process is called the binary tree sort.

20.7 Trees (cont.)

- Function `preOrderTraversal` invokes utility function `preOrderHelper` to perform the preorder traversal of the binary tree.
- The steps for an preorder traversal are:
  - Process the value in the node (line 81).
  - Traverse the left subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 82.)
  - Traverse the right subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 83.)
- The value in each node is processed as the node is visited.
- After the value in a given node is processed, the values in the left subtree are processed.
- Then the values in the right subtree are processed.
- The preorder traversal of the tree in Fig. 20.23 is
  - `27 13 6 17 42 33 48`

20.7 Trees (cont.)

- Function `postOrderTraversal` invokes utility function `postOrderHelper` to perform the postorder traversal of the binary tree.
- The steps for a postorder traversal are:
  - Traverse the left subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 120.)
  - Traverse the right subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 121.)
  - Process the value in the node (line 122).
- The value in each node is not printed until the values of its children are printed.
- The `postOrderTraversal` of the tree in Fig. 20.23 is
  - `6 17 13 33 48 42 27`

20.7 Trees (cont.)

- The binary search tree facilitates duplicate elimination.
- As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did when it was inserted in the tree.
- Thus, the duplicate will eventually be compared with a node containing the same value.
- The duplicate value may be discarded at this point.

20.7 Trees (cont.)

- Searching a binary tree for a value that matches a key value is also fast.
- If the tree is balanced, then each branch contains about half the number of nodes in the tree.
- Each comparison of a node to the search key eliminates half the nodes.
- This is called an O(log n) algorithm (Big O notation is discussed in Chapter 19).
- So a binary search tree with n elements would require a maximum of $\log_2 n$ comparisons either to find a match or to determine that no match exists.
- This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because $2^{10} > 1000$.
- When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because $2^{20} > 1{,}000{,}000$.

20.7 Trees (cont.)

- In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a level-order traversal of a binary tree.
- The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level.
- On each level of the tree, the nodes are visited from left to right.
- Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

Questions