CHAPTER 20

Binary Trees

Chapter Objectives

This chapter discusses:

- Dynamic data structure concepts
- The nature of a binary tree and its derivative—the binary search tree (BST)
- How to process binary trees in general
- Those operations defined for general data collections that are relevant to binary trees or BSTs

Introduction

We move to a slightly more complex dynamic data structure—the binary tree. Where each node of a linked list contains only one child node, a binary tree has two child nodes, normally referred to as the left and right nodes, as shown in Figure 20.1.

Within this structure, we need to consider two cases: a binary tree, where the data in the tree exhibit no apparent organization, and a binary search tree, where the data are organized for efficient searching.

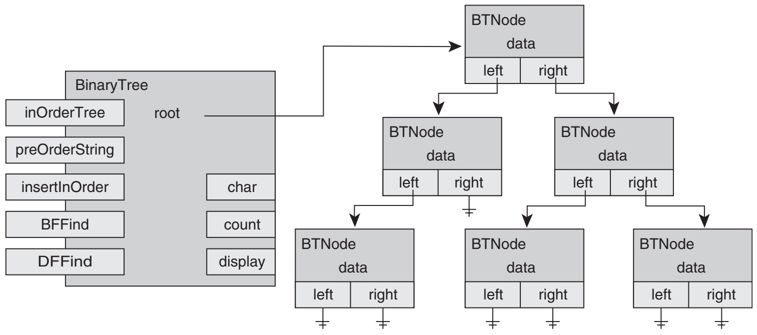

Figure 20.1 *Structure of a binary tree*

20.1 Dynamic Data Structure Concepts

20.1.1 Graphs

The most general form of dynamic data structure is a graph—an arbitrary collection of nodes connected by edges. A street map is a good example of a graph, where the intersections are the nodes and the streets are the edges. The edges may be directional to indicate that the graph can be traversed at that point in only one direction (like a one-way street). The edges may also have a value associated with them to indicate, for example, the cost of traversing that edge. For a street map, this could be either the distance or, perhaps in a more sophisticated system, the travel time—a function of the distance, the speed limit, and the traffic congestion. Graphs are not required to be completely connected, and they can obviously contain cycles—closed loops in which the unwary algorithm could get trapped. Graphs also have no obvious starting and stopping points.

20.1.2 Paths

A path on a graph is a connected list of edges that is the result of traversing a graph.

20.1.3 Directed Acyclic Graphs

On a graph, if we place the constraint that no path is permitted to wrap around and complete a cycle but child nodes can still have multiple parents, we have a directed, acyclic graph (DAG). These graphs are conceptually simpler to process than full graphs because they must have at least one node at which paths will originate.

20.1.4 Roots

No cycles are permitted in; therefore, there must be at least one starting point like there is for the head of a linked list. We refer to that starting point as the root.

20.1.5 Leaves

Just as there is one node that begins the tree, in general there will be many nodes that terminate the traversal of a tree because nodes have no child links. We refer to these nodes as the leaves. A linked list, by contrast, has only one leaf.

20.1.6 N-ary Trees

While a DAG is less complex to process than a graph, we must still apply another simplification. A tree is best thought of as a DAG in which nodes may have only one parent. A good example would be a family tree tracing the descendants from a common ancestor. In N-ary trees each node can contain an arbitrary number of links to other nodes.

20.1.7 Binary Trees

We make one more simplification before we can start analyzing the behavior of trees—it is convenient at the beginning to restrict the number of child nodes to two. This greatly reduces the amount of logic required to traverse a tree. An ancestral tree—tracing from a child back up through its lineage—is a good example of a binary tree.

20.1.8 Binary Search Trees

A binary search tree (BST) is an important specialization of a binary tree in which the data are ordered specifically to facilitate searching for information in the tree. We will discuss the impact of that ordering later. However, now is a good time to point out that if the data items in the tree are complex, one cannot order the data in the tree on all the components of the data. For example, it might be useful to organize student objects in a BST ordered by student IDs. As we will see, this greatly speeds the process of searching by ID for a student, but is not helpful when searching for other criteria, such as names or telephone numbers.

20.2 Processing Binary Trees in General

When discussing linked lists, we discovered that there are occasions in which one could iteratively process the list, especially if the list is not being changed. There are similar opportunities once we start processing graphs or trees. However, we first consider processing binary trees recursively. Figure 20.2 illustrates the content of the node with which we can construct a binary tree. Omitted for clarity are the `set/get` methods for the `data`, `left`, and `right` attributes.

We determine how to process a binary tree based on its formal definition:

- A binary tree is empty (`[ ]`)

or

- A `BTNode` contains a data item and two binary trees named `left` and `right`

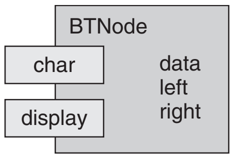

Figure 20.2 *A binary tree node*

Template 20.1 Template for processing a binary tree

```
<function and return> processBTNode(here)
% recursive processing
if isempty(here)
    <reached the end>
else
    ... getData(here) ...
     ... processBTNode(  getLeft(here) ) ...
     ... processBTNode( getRight(here) ) ...
end
```

Further, we could argue that we could model the template of a function for processing a binary tree after this definition, as shown in Template 20.1.

Figure 20.3 illustrates the operation of this recursive process on a particular `BTNode`. Just as with processing linked lists, the accomplishments of a recursive operation on a binary tree are governed by the logic implemented in the recursive call. In general, processing a node involves a recursive call to process the left child, a recursive call to process the right child, and then some logic to combine these two results with the data in this particular node. The results are then returned to the calling node.

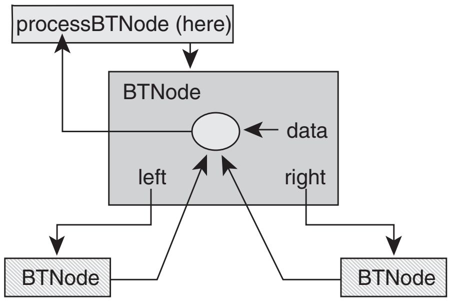

Figure 20.3 *Processing a binary tree*

20.3 Processing Binary Search Trees

The definition of a binary search tree (BST) is as follows:

- A BST is either empty (`[ ]`)

or

- A `BTNode` contains a data item and two BSTs named `left` and `right` where all the data in the left subtree are less[1] than the data at this node, and all the data in the right subtree are greater than this node

[1] This restricts the nature of the data items to those that can respond to `gt(...)`, `lt(...)`, and `eq(...)`.

- Equal data items are not permitted in BSTs[2] [3]

Notice that this is a recursive definition so that the ordering requirement continues throughout the BST.

The template and diagram for processing a BST is identical to that shown in Template 20.1 and Figure 20.3. As we see specific applications processing these trees, there are times when we can take advantage of the ordering of the data to reduce the amount of processing.

In each of the following sections, some of the discussions may apply only to one type of tree or the other. The script that tests these operations is shown in Section 20.9.4.

20.4 Traversing a Binary Tree

We will consider three different techniques for traversing a binary tree. Two are depth-first traversals—so-called because each traversal reaches recursively down paths to the leaves of the tree. The other is a breadth-first traversal that is performed iteratively and travels across the tree generations.

20.4.1 Two Depth-First Traversals

When traversing a linked list, the process is simple because there is only one child. The only choice you can make is to append the recursive result from the rest of the list before or after the local data value. When traversing a binary tree, there are three data items to combine: the local data item (D), the result from the left branch (L), and the result from the right branch (R). In general, they can be combined in six different combinations: D—L—R, L—D—R, L—R—D, D—R—L, R—D—L, and R—L—D. All six will give different results, but only two are actually useful to implement.

- The L—D—R sequence is referred to as an "in-order traversal" because if the tree is a BST, it will visit the nodes in their assigned order in the tree. This is the standard traversal method, unless there is a good reason not to use it.[4] Template 20.2 shows the template for in-order traversal.
- The D—L—R sequence is referred to as a "pre-order traversal" because it touches the data first before performing either recursion.

[2] This requirement drives the nature of the information stored in a BST—there needs to be a high probability that the index will be unique, so that equality becomes either a data item to be ignored or an error to be analyzed. Good indices would be a telephone number or student ID. Bad indices would be first and last names in any combination, since uniqueness is not guaranteed.

[3] It is technically possible, but beyond the scope of this text, to construct a BST where the data item stored is a collection of items, all of which satisfy a specific equality test. The structure of the BST would then continue to satisfy this requirement, while its implementation would add equal items to the local collection. Traversal and search then become considerably more complex, of course.

[4] For our current purposes, the only reason not to use an in-order traversal is if the traversal is for the purpose of serializing a BST (writing it to a file for archiving purposes). If you use an in-order traversal for this, the data will be in order. If you subsequently insert that data, an item at a time back into an initially empty BST, the result will not be the original BST, but rather a linear tree!

Template 20.2 Recursive `BinaryTree` traversal template

```
<function and return> inOrder(here)
% processing items in order
if isempty(here)
     <return the empty answer>
else
<combine the following>
... inOrder(getLeft(here)) ...
... getData(here) ...
... inOrder(getRight(here)) ...
end
```

This form of traversal is used when serializing a BST because it preserves the tree structure.

By convention in this text, the `char(...)` conversion of these more complex collections will not traverse the collection. Specific methods traversing the collections will be used. Listing 20.1 shows the `inOrderString(...)` method that directly matches the above template. For comparison:

- As usual, the wrapper function merely invokes the recursive process with the tree root
- Both functions return a character string
- The recursive function concatenates the result of the left recursion with the `char(...)` of the local node and the result of the right recursion
- The `BTNode char(...)` function takes care of presenting the string value of double types using the `%g` conversion

In Listing 20.1:

Lines 1–3: Show the header for the wrapper function.

Listing 20.1 The `BinaryTree inOrderString(...)` method

```
 1. function str = inOrderString(bt)
 2. % @BinaryTree\inOrderString
 3. %   str = inOrderString(bt)
 4. str = RinOrderString(bt.root);

 5. function str = RinOrderString(here)
 6. %  recursive in-order processing
 7. if isempty(here)
 8.     str = '';
 9. else
10.     str = [ RinOrderString(getLeft(here)) ' ' ...
            char(here) ' ' ...
            RinOrderString(getRight(here)) ];
11. end
```

Line 4: The wrapper function invokes the recursive helper function with the root of the tree.

Lines 5 and 6: Show the header for the recursive helper function.

Lines 7 and 8: At a leaf node, return an empty string.

Lines 9–11: Otherwise, concatenate a string using (in order) the left recursive call, the character conversion of the local `BTNode`, and the right recursive call.

Listing 20.2 shows a pre-order traversal that is identical to Listing 20.1 except for the order of the concatenation. In this case, the node character conversion precedes the left and right traversal.

20.4.2 Breadth-First Traversal

Breadth-first traversal permits data in the tree to be accessed a generation at a time. It uses a simple, but effective, iterative template that uses a queue to store the children of each generation. Template 20.3 shows the template for this process.

Listing 20.2 The `BinaryTree preOrderString(...)` method

```
 1. function str = preOrderString(bt)
 2. % @BinaryTree\preOrderString
 3. %   str = preOrderString(bt)
 4. str = RPreOrderString(bt.root);

 5. function str = RPreOrderString(here)
 6. %  recursive in-order processing
 7. if isempty(here)
 8.     str = '';
 9. else
10.     str = [ char(here) ' ' ...
            RPreOrderString(getLeft(here)) ' ' ...
            RPreOrderString(getRight(here)) ];
11. end
```

Template 20.3 Template for `BFTraversal`

```
<function and return> BFTraversal(btree)
% @BinaryTree\BFTraversal
q = Queue;     % create a queue
enqueue(q, getRoot(btree) );
< initialize the result>
while !isEmpty(q)
node = dequeue(q);
<operate on getData(node)>
enqueue(q, getLeft(node));
enqueue(q, getRight(node));
end
< return the result>
```

Listing 20.3 shows the code that produces a string representation of the tree using a breadth-first traversal. This is slightly complicated by the need to keep up with the layers in the tree. What is pushed onto the queue is a cell array containing a number—the level of the tree—with the node to be processed.

- All the children are pushed with the value of the next level
- A new line is inserted into the string when the level changes
- Whenever the level changes, we halve the number of spaces to be written between the items on a row
- Since it is possible for leaves to appear in the printout, they are marked by `'—'`.

In Listing 20.3:

Lines 1–3: Show the usual function header.

Lines 4 and 5: Create a queue and enqueue the tree root.

Lines 6–9: Initialize the data. We set the current level to –1 in order to force a new line at the beginning of the output.

Line 10: Continues as long as there are nodes in the queue.

Line 11: Removes an entry. Each entry is a cell array containing the current level (generation) in the tree and the current node to process.

Listing 20.3 `BFTString` generator

```
 1. function str = BFTString(bt)
 2. % @BinaryTree\BFTString
 3. %  tree traversal including formatting lines
 4. q = Queue;
 5. enqueue(q, {0, bt.root} )
 6. level = -1;
 7. sp='                                           ';
 8. str = '';
 9. nsp = 2*length(sp);
10. while ~isEmpty(q)
11.     v = dequeue(q);
12.     if v{1} ~= level      % set the tree level
13.         level = v{1};
14.         str = [str '\n'];
15.         nsp = ceil(nsp / 2);
16.     end
17.     node = v{2}; % insert what we just dequeued
18.     if isempty(node)
19.         str = [str sp(1:nsp) '—'];
20.     else
21.         str = [str sp(1:nsp) char(node)];
22.         % enqueue the children of this node
23.         enqueue(q, {level+1, getLeft(node)  });
24.         enqueue(q, {level+1, getRight(node) });
25.     end
26. end
```

Line 12: If the dequeued level is not the current level, we move to a new line of output.

Line 13: Sets the new tree level.

Line 14: Inserts a new-line character into the output string.

Line 15: Shows half of the number of spaces used to separate the tree data (a crude tree layout scheme that works tolerably well).

Line 17: Fetches the node we just dequeued.

Lines 18 and 19: If it is empty, add spaces and then `'–'` to the output string to mark the spot where a leaf node terminated.

Lines 21–24: Otherwise, append spaces and the character conversion for that node to the output string and enqueue both children of the node at the next level. This forces a new tree row to start when these children are processed.

20.5 Building a Binary Search Tree

Since the structure of ordinary binary trees usually has some specific meaning not evident from the data content itself, they are not built dynamically. For our purposes, they will be created in the test script. It is very common, however, to build a BST by inserting data items one at a time.

Building the BST consists of a repetitive insertion of data into an initially empty BST. The major technical challenge for inserting into a BST is where to place the new node and retain the ordering of the data. This ordering cannot be guaranteed when inserting at the root or at some intermediate

Listing 20.4 Inserting into a BST

```
 1. function insert(bst, n)
 2. % @BST\insert
 3. setRoot(bst, rInsert(getRoot(bst), n) );
 4. assignin('caller', inputname(1), bst);

 5. function node = rInsert(here, n)
 6. % recursive insertion into a BST
 7. if isempty(here)
 8.     node = BTNode(n);
 9. elseif getData(here) == n
10.     node = here;
11. elseif getData(here) > n
12.     node = BTNode(getData(here), ...
                  rInsert(getLeft(here), n), ...
                  getRight(here));
13. else
14.     node = BTNode(getData(here), ...
                  getLeft(here), ...
                  rInsert(getRight(here), n));
15. end
```

layer of the tree. The only place one can safely add to a BST and retain its ordering is at a leaf node, which is usually found recursively. This code is listed in Listing 20.4. This code will function well for any tree content object that can respond to the equality test and the greater than test (that is, an object that has `gt(...)` and `eq(...)` methods defined).

In Listing 20.4:

> Lines 1 and 2: Show the header for the wrapper function. As with any operation that changes a dynamic data structure, the whole structure is always replaced.
>
> Line 3: Passes the root to start the recursive helper, and stores the result as the new root.
>
> Line 4: Copies the local object back to the caller.
>
> Lines 5 and 6: Show the header for the recursive helper. Notice that a new `BTNode` is created and returned from each exit point.
>
> Lines 7 and 8: Show the terminating condition: we found a leaf node, and return a new `BTNode` containing the new data.
>
> Lines 9 and 10: We must test for equality to avoid adding duplicate keys. A policy decision is needed to decide whether to announce an error here or merely (as in this code) exit without adding a new node by just returning what we were given. Notice that the existing node must still be returned to add the rest of that structure to the new resulting structure.
>
> Line 11: We compare the current node's data to the value to insert to decide whether the insertion happens to the left or right of this node.
>
> Lines 12–14: Whichever direction is chosen, a new node is created with the same data value as the current node, the same left or right `subtree` to the side that is unchanged, and a recursive call to the side that is changed.

20.6 Mapping a Binary Tree

In general, mapping a binary tree is performed not as a general service to all `BinaryTree` users, but rather as a specialized utility written for derived classes that uses a `BinaryTree` as their parent class. For example, suppose a bank keeps all its interest-bearing accounts in a class derived from the `BinaryTree` class. Periodically, they need to traverse that tree and add the interest generated by each account. Since the size of the tree remains unchanged, but the contents change, this is a mapping of the original tree.

Listing 20.5 illustrates such an update method. In order to update all the accounts in a tree of `SavingsAccounts`, the mapping would follow the recursive template shown in Template 20.1 with the following tailoring required:

- As usual, the public method, `updateTree(...)`, merely launches the recursive helper with the current root of the tree and stores the root returned as the new root
- The helper returns an empty tree when it reaches a leaf of the old tree
- Otherwise, it returns a new `BTNode` containing the result of updating the old data, using the recursive `updateR` call to update the rest of the tree

In Listing 20.5:

Lines 1–4: Show the usual wrapper function header.

Line 5: Shows the wrapper body calling the recursive program, passing the tree root in, saving the new root, and then copying the local object back to the user.

Lines 7 and 8: Show the header for the recursive helper.

Lines 9 and 10: Show termination when you reach a leaf returning an empty node.

Line 12: Otherwise, create a new `BTNode` calling the `update(...)` method on the original node data, and use the recursive results from both left and right.

Listing 20.5 Mapping a `SavingsAccount` tree

```
 1. function updateTree(acctTree)
 2. % @SavingsAcctTree\update
 3. %    update to the accounts by depositing
 4. %    the interest due
 5.  setHead(acctTree, updateR(getRoot(acctTree)) );
 6.  assignin('caller', inputname(1), acctTree);

 7. function newTree = updateR(here)
 8. % recursive update helper
 9. if isempty(here)
10.     newTree = [];
11. else
12.     newTree = BTNode( update( getData(here) ), ...
                         updateR( getLeft(here) ), ...
                         updateR( getRight(here) ));
13. end
```

20.7 Deleting Nodes from a Binary Search Tree

We do not, in general, filter binary trees. These trees are static in nature, and rarely need maintenance. If maintenance is required, this is usually done manually in the code that reconstructs the tree. We can, however, filter a BST in a restricted sense—removing one node at a time, while preserving the ordering of the BST. Figure 20.4 illustrates this process.

When removing an item from a BST, there are three cases to consider depending on the location in the tree of the item to be removed.

No children node: Consider, for example, removing the node containing 13. Since it has no children at all, it can be removed simply by replacing its link in the node containing 9 with [ ].

One child node: Removing a node with one child node (such as those with value 9 or 71) is also simple. We merely replace its parent node link with a link to the surviving child node.

Two children nodes: The logic for removing a node with both children present is a little more challenging. Consider removing the node with value 50. First, we must find the largest node in its left subtree, the value 42, or the smallest in its right subtree, the value 66. We place its node data in the node we are removing, and then delete that node from the tree. This latter deletion is one of the simpler cases with fewer than two children.

Listings 20.6 and 20.7 show the code for this.

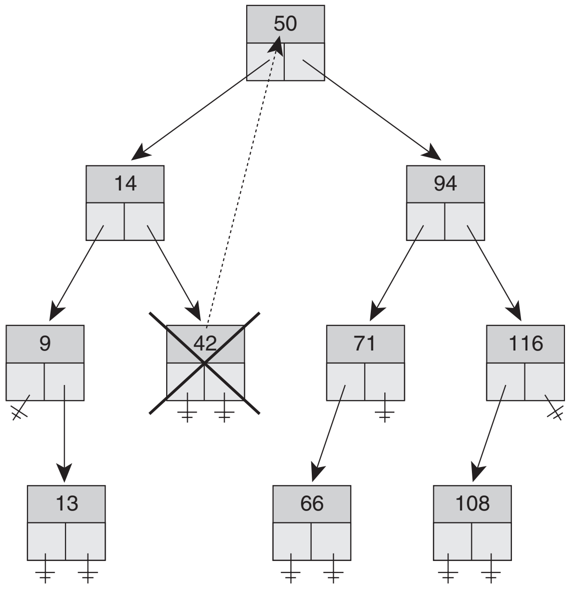

Figure 20.4 *Deleting from a BST*

Listing 20.6 Deleting from a BST Part 1—finding and repairing

```
 1. function delete(bst, n)
 2. % @BST\delete
 3. setRoot(bst, rDelete(getRoot(bst), n) );
 4. assignin('caller', inputname(1), bst);
 5. function node = rDelete(here, n)
 6. % recursive BST delete function
 7. if isempty(here)           % not in the tree
 8.     node = [];
 9. elseif getData(here) == n  % found it
10.     node = deleteThis(here);
11. elseif getData(here) > n   % look left
12.     node = BTNode(getData(here), ...
                  rDelete(getLeft(here), n), ...
                  getRight(here));
13. else                        % look right
14.     node = BTNode(getData(here), ...
                  getLeft(here), ...
                  rDelete(getRight(here), n));
15. end
```

In Listing 20.6:

Lines 1–4: Show the typical wrapper function passing the root to the recursive helper, saving the result as the new root and returning the local object to the caller.

Lines 5 and 6: Show the header for the recursive delete function.

Lines 7 and 8: Show the terminating condition—we did not find the value sought—and return an empty node.

Lines 9 and 10: Show successful termination—we found the node required, and call `deleteThis(...)` (Listing 20.7) to compute the new `subtree` below this node.

Lines 11–15: The usual BST search creates a new `BTNode`, substituting the results of a right or left recursive call.

Listing 20.7 Deleting from a BST Part 2—finding the replacement

```
function node = deleteThis(here)
% @BST\deleteThis – delete this node
left = getLeft(here);
right = getRight(here);
if isempty(left)
    node = right;
elseif isempty(right)
    node = left;
else
    [value rest] = deleteLargest(left);
    node = BTNode(value, rest, right);
end
```

continued on next page

```
13. function [value node] = deleteLargest(here)
14. % find the largest in this tree
15. right = getRight(here);
16. if isempty(right)
17.     value = getData(here);
18.     node = getLeft(here);
19. else
20.     [value, node] = deleteLargest(right);
21. end
```

In Listing 20.7:

Lines 1 and 2: Show the header.

Lines 3 and 4: Retrieve the left and right nodes.

Lines 5 and 6: Show one of the easy cases—if the left is empty, return the right node. That might also be empty, of course.

Lines 7 and 8: Otherwise, the right node's being empty also gives us one of the easy cases.

Line 10: No such luck, so we have to go to a recursive helper to give us two things: the value of the node we want to delete and the tree that remains when we delete it. We provide it with the left subtree and expect the largest value back. We could also have passed the right subtree to a similar function that looks for the smallest value.

Line 11: Creates a new node with the value returned, the remains of the left `subtree`, and the untouched right subtree.

Lines 13 and 14: Show the recursive helper function that finds and deletes the largest node in this `subtree`.

Line 15: Gets the right node. The largest in this `subtree` will be the node that has no right child (it might have a left child).

Line 16: Shows the terminating condition: the right child is empty.

Lines 17 and 18: If not, return the value of this node and its left child that might be empty.

Line 20: Otherwise, keep going recursively to the right.

20.8 Folding a Binary Tree

Unlike the previous tree operations, folding can be performed as a service in the `BinaryTree` class. We illustrate this by the `count(...)` method. To count items in a binary tree, we will again follow the recursive template shown in Template 20.1, with the following tailoring required:

- As usual, the public method, `count(...)`, merely launches the recursive helper, `rCount`, with the root of the tree and passes the total to its caller

- The helper returns `0` when it reaches a tree leaf
- Otherwise, it returns the result from adding `1` to the result of applying `rCount` recursively to left and right `subtrees`

The resulting code is shown in Listing 20.8.

In Listing 20.8:

Lines 1 and 2: Show the function header.

Line 3: Returns the answer from the recursive call. Since we are not changing the tree, we do not need to return our object to the caller.

Lines 4 and 5: Show the header for the recursive helper.

Lines 6 and 7: Show the terminating condition—a leaf node. Return a count of `0`.

Line 9: Otherwise, adds `1` for this node to the recursive counts for the left and right subtrees.

Listing 20.8 A generic `BinaryTree count` method

```
 1. function ans = count(bt)
 2. % @BinaryTree\count
 3. ans = rCount(bt.root);


 4. function ans = rCount(here)
 5. %  recursive counting a binary tree
 6. if isempty(here)
 7.     ans = 0;
 8. else
 9.     ans = 1 + rCount(getLeft(here)) ...
                + rCount(getRight(here));
10. end
```

20.9 Searching Binary Trees

We need to consider searching both the ordinary binary tree and the BST.[5] In the same way as we categorized traversal, searching an ordinary binary tree may be accomplished either breadth-first or depth-first.

20.9.1 Breadth-First Search

The breadth-first search method `BFFind(...)` will follow the iterative, breadth-first template shown in Template 20.3, with the following tailoring required:

- First, it creates a queue and enqueues the tree root

[5] We should observe again that a BST ordered by some particular data attribute can be searched efficiently for items with that particular attribute. However, to find items by some other criteria requires it to be treated as a regular binary tree.

- As long as that queue is not empty (indicating failure) or the dequeued item is not what we seek, it continues enqueueing the children of the current node

The resulting code is shown in Listing 20.9.

In Listing 20.9:

Lines 1 and 2: Show the function header.

Lines 3 and 4: Create a queue and enqueue the tree root.

Line 5: Initializes the loop control variable.

Line 6: Initializes the answer.

Line 7: Shows loop-and-a-half repeats until the answer is found (success), or the queue is empty (failure).

Line 8: Fetches a node.

Line 10: Checks for the target (invokes the `eq(...)` method).

Line 11: Continues the process only when this is not the target.

Line 13–16: Enqueue the left child if it is not empty.

Lines 17–20: Enqueue the right child if it is not empty.

Lines 23 and 24: Return the node found rather than `true` or `false`. With complex objects, this allows access to other parts of the data found.

Listing 20.9 `BFFind` method

```
 1. function ans = BFFind(bt, what)
 2. % @BinaryTree\BFFind
 3. q = Queue;
 4. enqueue(q, bt.root )
 5. found = false;
 6. ans = [];
 7. while ~found && ~isEmpty(q)
 8.     node = dequeue(q);
 9.     % check the data we just dequeued
10.     found = (node == what);
11.     if ~found
12.     % enqueue the children of this node
13.         left = getLeft(node);
14.         if ~isempty(left)
15.             enqueue(q, left );
16.         end
17.         right = getRight(node);
18.         if ~isempty(right)
19.             enqueue(q, right );
20.         end
21.     end
22. end
23. if found
24.     ans = getData(node);
25. end
```

20.9.2 Depth-First Search

Depth-first search can also be accomplished iteratively using an `MStack` instead of a `Queue`, as illustrated with the `DFFind(...)` method. The code illustrated in Listing 20.10 is similar in every way to Listing 20.9, except that an `MStack` class with `push(...)` and `pop(...)` is used in place of the `Queue` class with `enqueue(...)` and `dequeue(...)`.

20.9.3 Searching a Binary Search Tree

BST search takes advantage of the ordering of the data to eliminate half of the tree at each search step, as shown in Listing 20.11. Although this could be accomplished with the `while` loop template, the recursive template shown in Template 20.1 is a better choice for simpler code. Obviously, the data items in the tree must provide both an `eq(...)` method and a `gt(...)` method.

Listing 20.10 `DFFind` method

```
function ans = DFFind(bt, what)
% BinaryTree\DFFind
st = MStack;
push(st, bt.root )
found = false;
ans = [];
while ~found && ~isEmpty(st)
    node = pop(st);
    % check the data we just popped
    found = (node == what);
    if ~found
    % push the children of this node
        left = getLeft(node);
        if ~isempty(left)
            push(st, left );
        end
        right = getRight(node);
        if ~isempty(right)
            push(st, right );
        end
    end
end
if found
    ans = getData(node);
end
```

Listing 20.11 BST search

```
function ans = find(bst, n)
% @BST\find
ans = rFind( getRoot(bst), n);
```

continued on next page

```
function ans = rFind(here, n)
%
if isempty(here)
    ans = [];
elseif getData(here) == n
    ans = getData(here);
elseif getData(here) > n
    ans = rFind(getLeft(here), n);
else
    ans = rFind(getRight(here), n);
end
```

20.9.4 Testing Binary Search Tree Functionality

Listing 20.12 shows how to test the binary tree functions discussed so far in this chapter.

In Listing 20.12:

Line 1: Asks the user how many nodes for this tree.

Listing 20.12 Testing the BST functions

```
 1. nn = input('How many nodes? ')
 2. bst = BST( BTNode(50) );
 3. for in = 1:nn
 4.     insert(bst, ceil(100*rand));
 5. end
 6. fprintf('number is %d (started with %d)\n', ...
                            count(bst), nn+1 );
 7. fprintf('largest number is %d\n', ...
                            largest(bst) );
 8. fprintf('pre-order list is %s\n', ...
                         preOrderString(bst) );
 9. fprintf('in-order list is %s\n', ...
                         inOrderString(bst) );
10. fprintf('breadth-first list: %s\n', ...
                         BFTString(bst));
11. n = 1;
12. while n > 0
13.  n = input('number to remove: ');
14.  if n > 0
15.   ans = find(bst, n)
16.   BSTdelete(bst, n);
17.   fprintf('number is %d (started with %d)\n', ...
                        count(bst), nn+1 );
18.   fprintf('in-order list is %s\n', ...
                        inOrderString(bst) );
19.   fprintf(' bst with %d removed: %s\n', ...
                        n, BFTString(bst));
20.  end
21. end
```

Line 2: Constructs a BST with 50 as its root. This is actually a sneaky trick to give the BST some balance. Since we will be inserting values between 0 and 100, starting with a root of 50 gives the best chance of a roughly balanced tree.

Lines 3–5: Insert the requested number of values into the tree.

Line 6: Folds the tree by counting its nodes. Why might these numbers not match?

Line 7: Shows another fold to find the largest number. Would this function be different on a BST than on a binary tree? Would it be more efficient?

Lines 8–10: Show results from pre-order, in-order, and breadth-first traversals.

Lines 11–13: Show a loop-and-a-half, asking the user which node to delete from the tree.

Line 14: Shows that a negative entry terminates the loop.

Line 15: Searches the BST for the value requested.

Lines 16–19: Delete the value and display the count, in-order listing, and the breadth-first traversal of the result.

20.10 Combining Complex Operations

When the stated problem is a little more complex than the ordinary operations previously indicated, it is tempting to expand the code for a single operation. For example, suppose we were asked to put into a cell array the names of all foreigners in Prince William's ancestral tree. It would be possible to modify the `fold` method on the `FamilyTree` to apply the condition that the person's nationality be not British, and then add their name to a cell array of names. However, it seems that a regular traversal of the tree, extracting all the data into a cell array and then filtering that cell array to remove all the non-British subjects, might take a little longer but would be much more robust and comprehensible.

Chapter Summary

In this chapter, we have seen the following:

- Dynamic data structures can be created in many forms, the most general of which is the graph.
- Binary trees are defined recursively as dynamic collections where each node has at most two children.
- A binary search tree (BST) is a binary tree where at every node, all data less than that node value are in the left subtree, and all data greater than its value are in the right subtree.

- In general, binary trees are processed recursively, except when performing breadth-first operations where a queue can be used to organize an iterative solution.
- Because the organization of the data makes searching BSTs very efficient, most applications of interest relate to managing, traversing, and searching BSTs.