17 Binary Trees

17.1 Introduction

As mentioned in the last chapter, binary trees are ordered trees whose vertices have at most two children, the left child and the right child. Although other kinds of ordered trees arise in computing, binary trees are especially common and have been especially well studied. In this chapter we discuss the binary tree abstract data type and binary trees as an implementation mechanism.

17.2 The Binary Tree ADT

Binary trees hold values of some type, so the ADT is *binary tree of T*, where *T* is the type of the elements in the tree. The carrier set of this type is the set of all binary trees whose vertices hold a value of type *T*. The carrier set thus includes the empty tree, the trees with only a root holding a value of type *T*, the trees with a root and a left child, the trees with a root and a right child, and so forth. Operations in this ADT include the following.

size(*t*)—Return the number of vertices in the tree *t*.

height(*t*)—Return the height of tree *t*.

empty?(*t*)—Return true just in case *t* is the empty tree.

contains?(*t*,*v*)—Return true just in case the value *v* is present in tree *t*.

buildTree(v,t_l,t_r)—Create and return a new binary tree whose root holds the value *v* and whose left and right subtrees are t_l and t_r.

emptyTree()—Return the empty binary tree.

rootValue(*t*)—Return the value of type *T* stored at the root of the tree *t*. Its precondition is that *t* is not the empty tree.

leftSubtree(*t*)—Return the tree whose root is the left child of the root of *t*. Its precondition is that *t* is not the empty tree.

rightSubtree(*t*)—Return the tree whose root is the right child of the root of *t*. Its precondition is that *t* is not the empty tree.

This ADT allows us to create arbitrary binary trees and examine them. For example, consider the binary tree in Figure 1.

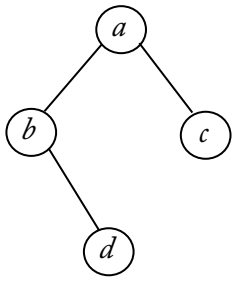


Figure 1: A Binary Tree

This tree can be constructed using the expression below.

```
buildTree(a,
        buildTree(b,
                emptyTree(),
                buildTree(d,
                        emptyTree(),
                        emptyTree())),
        buildTree(c,
                emptyTree(),
                emptyTree()))
```

To extract a value from the tree, such as the bottom-most vertex *d*, we could use the folowing expression, where *t* is the tree in Figure 1.

$$rootValue(rightSubtree(leftSubtree(t)))$$

As with the ADTs we have studied before, an object-oriented implementation of these operations as instance methods will include the tree as an implicit parameter, so the signatures of these operations vary somewhat when they are implemented. Furthermore, there are several operations that are very useful for a binary tree implementation that are not present in the ADT and several operations in the ADT that are not needed (more about this below).

17.3 The Binary Tree Class

We could treat binary trees as a kind of collection, adding it to our container hierarchy, but we won't do this for two reasons:

- In practice, binary trees are used to implement other collections, not as collections in their own right. Usually clients are interested in using basic Collection operations, not in the intricacies of building and traversing trees. Adding binary trees to the container hierarchy would complicate the hierarchy with a container that not many clients would use.

- Although binary trees have a contiguous implementation (discussed below), it is not useful except for heaps. Providing such an implementation in line with our practice in the container hierarchy to make both contiguous and linked implementations for all interfaces would create a class without much use.

We will make a BinaryTree class but its role will be to provide an implementation mechanism for other collections. Thus BinaryTree is not part of the container hierarchy though it includes several container operations. It also includes operations for creating and traversing trees in various ways, as well as several kinds of iterators. The Binary Tree class is pictured in Figure 2.

Note that there is no *buildTree*() operation and no *emptyTree*() operation in the BinaryTree class, though there is one in the ADT. The BinaryTree class constructor does the job of these two operations, so they are not needed as separate operations in the class.

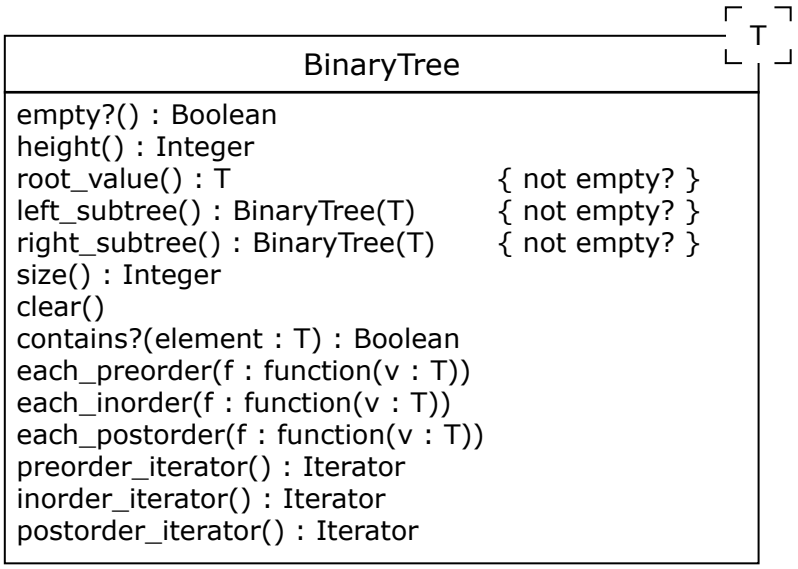


Figure 2: The BinaryTree Class

To *visit* or *enumerate* the vertices of a binary tree is to traverse or iterate over them one at a time, processing the values held in each vertex. This requires that the vertices be traversed in some order. There are three fundamental orders for traversing a binary tree. All are most naturally described in recursive terms.

Preorder: When the vertices of a binary tree are visited in preorder, the root vertex of the tree is visited first, then the left sub-tree (if any) is visited in preorder, then the right sub-tree (if any) is visited in preorder.

Inorder: When the vertices of a binary tree are visited inorder, the left sub-tree (if any) is visited inorder, then the root vertex is visited, then the right sub-tree is visited inorder.

Postorder: When the vertices of a binary tree are visited in postorder, the left sub-tree is visited in postorder, then the right sub-tree is visited in postorder, and then the root vertex is visited.

To illustrate these traversals, consider the binary tree in Figure 3 below. An inorder traversal of the tree in Figure 3 visits the vertices in the order *m, b, p, k, t, d, a, g, c, f, h*. A preorder traversal visits the vertices in the order *d, b, m, k, p, t, c, a, g, f, h*. A postorder traversal visits the vertices in the order *m, p, t, k, b, g, a, h, f, c, d*.

The BinaryTree class has internal iterators for visiting the vertices of a tree in the three orders listed above and applying the function passed in as an argument to each vertex of the tree. For examples, suppose that a print(v : T) operation prints the value v. If t is a binary tree, then the call t.each_inorder(print) will cause the values in the tree t to be printed out inorder, the call t.each_preorder(print) will cause them to be printed in preorder, and the call t.each_postorder(print) will cause them to be printed in postorder.

In addition, the BinaryTree class has three operations that return external iterators that provide access to the values in the tree in each of the three orders above.

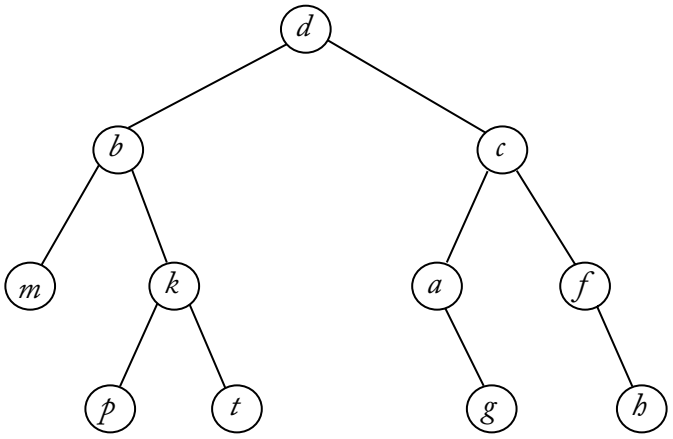


Figure 3: A Binary Tree

When implementing the BinaryTree class in Ruby, it will mix in `Enumerator`, and the `Enumerator.each()` operation will be an alias for the each_inorder(), which is the most common way to visit the vertices of a binary tree.

17.4 Contiguous Implementation of Binary Trees

We have already considered how to implement binary trees using an array when we learned about heapsort. The contiguous implementation is excellent for complete or even full binary trees because it wastes no space on references and it provides a quick and easy way to navigate in the tree. Unfortunately, in most applications binary trees are far from complete, so many array locations are never used, which wastes a lot of space. Even if our binary trees were always full, there is still the problem of having to predict the size of the tree ahead of time so that an array could be allocated that is big enough to hold all the tree vertices. The array could be reallocated if the tree becomes too large, but this is an expensive operation.

This is why it is not particularly useful to have a contiguous implementation of binary trees. Instead we will implement our BinaryTree class as a linked data structure and use it as the linked structure implementation mechanism for several of the collections we will add to our container hierarchy later on.

17.5 Linked Implementation of Binary Trees

A linked implementation of binary trees resembles implementations of other ADTs using linked structures. A BinaryTreeNode class require three attributes: one for the data held at the node and two for references to the nodes that re the roots of the left and right sub-trees. In addition, it is useful to have a BinaryTree class acting as the host for the graph formed by the linked nodes. This host structure has a reference to the tree's root node and other attributes as needed. For example, a count attribute might be useful to keep track of how many nodes are in the tree. Figure 4 shows how this works for a small example. The BinaryTree class has root and count attributes and the BinaryTreeNode class has a value attribute to store the data at the node and two reference attributes for the left and right sub-trees. These reference attributes are nil when their respective sub-trees are empty.

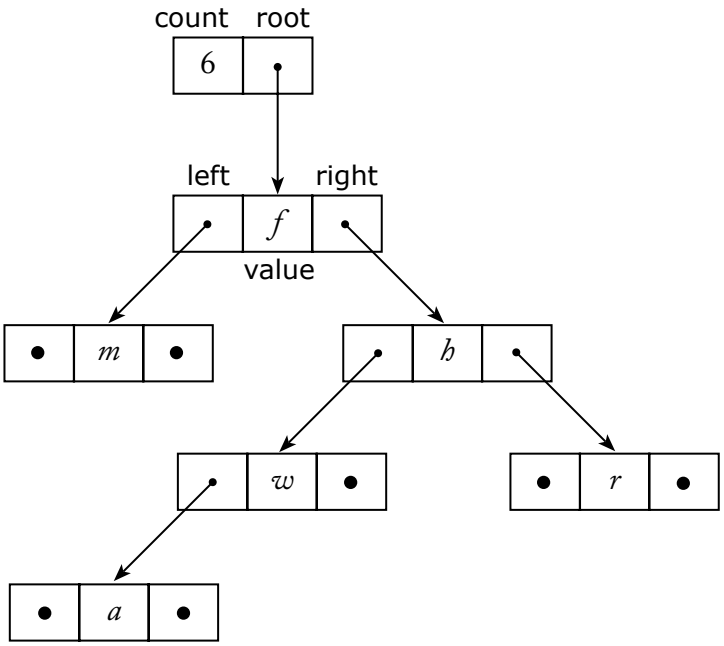


Figure 4: A Linked Representation of a Binary Tree

Trees are inherently recursive structures so it is natural to write many BinaryTree class operations recursively. For example, to implement the size() operation the BinaryTree can call an internal size(r : BinaryTreeNode) operation on the root node. This operation returns zero if its parameter is nil, and one plus the sum of recursive calls on the left and right sub-trees of its parameter node. Many other operations, particularly the internal iterator operations that apply functions to the data held at each node, can be implemented just as easily.

Implementing external iterators is more challenging, however. The problem is that external iterators cannot be written recursively because they have to be able to stop every time a new node is visited to deliver the value at the node to the client. There are two ways to solve this problem:

- Write a recursive operation to copy node values into a queue in the correct order and then extract items from the data structure one at a time as the client requests them.
- Don't user recursion to implement iterators: use a stack instead.

The second alternative, though harder to do, it clearly better because it uses much less space.

17.6 Summary and Conclusion

The binary tree ADT describes basic operations for building and examining binary trees whose vertices hold values of type *T*. A BinaryTree class has several operations not in the ADT, in particular, visitor operations for traversing the vertices of the tree and applying a function to the data stored in each node. External iterators are also made available by this class.

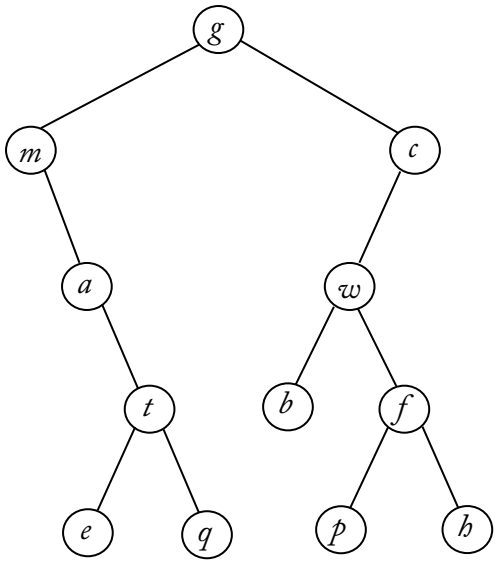
Contiguous implementations of the binary tree ADT are possible and useful in some special circumstances, such as in heapsort, but the main technique for implementing the binary tree ADT uses a linked representation. Recursion is a very useful tool for implementing most BinaryTree operations but it cannot be used as easily for implementing external iterators. The BinaryTree class implemented using a linked structure will be used as an implementation mechanism for container classes to come.

17.7 Review Questions

1. Where does the BinaryTree class fit in the Container class hierarchy?
2. Why does the BinaryTree class not include a buildTree() operation?
3. Why is the contiguous implementation of binary trees not very useful?
4. What is the relationship between the BinaryTree and BinaryTreeNode classes?

17.8 Exercises

1. Write the values of the vertices in the following tree in the order they are visited when the tree is traversed inorder, in preorder, and in postorder.



2. Write the size() operation for BinaryTree class in Ruby.
3. Write the height() operation for the BinaryTree class in Ruby.
4. Write the each_preorder(), each_inorder(), and each_postorder() operations for the BinaryTree class as internal iterator operations in Ruby.
5. Write a PreorderIterator class whose instances are iterators over for the BinaryTree class. The PreorderIterator class will need a stack attribute to hold the nodes that have not yet been visited, with the current node during iteration at the top of the stack.

17.9 Review Question Answers

1. We have decided not to include the BinaryTree class in the Container class hierarchy because it is usually not used as a container in its own right, but rather as an implementation mechanism for other containers.
2. The BinaryTree class does not include a buildTree() operation because it may have a constructor that does the very same job.
3. The contiguous implementation of binary trees is not very useful because it only uses space efficiently if the binary tree is at least full, and ideally complete. In practice, this is rarely the case, so the linked implementation uses space more efficiently.
4. The BinaryTree class has an attribute that stores a reference to the root of the tree, which is a BinaryTreeNode instance. The root node (if any) stores references to the left and right sub-trees of the root, which are also references to instances of the BinaryTreeNode class. Although the BinaryTree and BinaryTreeNode classes are not related by inheritance, they are intimately connected, just as the LinkedList class is closely connected to the LinkedListNode class.