Chapter 13 The General Binary Tree

In Chapter 6 we examined the heap, a partially ordered complete binary tree. We implemented the heap ADT as an array container because the special properties of a complete binary tree make possible an elegant array implementation. In general, though, binary trees are implemented as linked containers. In this chapter we will discuss the binary tree ADT and implement it as a linked container.

13.1 Our Current Model

Our current UML class diagram is shown below in Figure 13.1. It includes class TreeContainer and class BinaryTree, which has two nested types, class BinaryTreeNode and interface NodeProcessor. To simplify the diagram, nested classes SLNode and DLNode have been omitted.
Fig. 13.1. Our current software design, which includes a classifier for linked trees in general and a classifier for linked binary trees more specifically.
13.2 The Binary Tree ADT

The binary tree ADT appears below. An explanation of the *traversal* operations follows the ADT specification.

binary tree: See Chapter 6.

operations:

- `clear()` – Make the collection empty.
- `height()` – What is the height of the tree?
- `inorderTraverse(processor)` – Perform an inorder traversal of the tree, applying some operation to each node.
- `isEmpty()` – Is the collection empty?
- `postorderTraverse(processor)` – Perform a postorder traversal of the tree, applying some operation to each node.
- `preorderTraverse(processor)` – Perform a preorder traversal of the tree, applying some operation to each node.
- `size()` – How many elements are in the collection?

To *traverse* a binary tree is to visit each of the tree's nodes. It is not always necessary to visit the nodes in any particular order, but various applications of the binary tree ADT require one or more of three standard orders of traversal. We will examine these three standard traversals by way of example, using the tree in Figure 13.2.

![Fig. 13.2. An example tree for traversal](image-url)
A **preorder traversal** visits a given node before visiting either of the node's children (and their subtrees). A preorder traversal of the tree from Figure 13.2 visits the nodes in the following order: 1 2 4 5 3 6 7 8.

An **inorder traversal** visits a given node's left child (and the left child's subtree), then the node itself, then the node's right child (and the right child's subtree). An inorder traversal of the example tree visits the nodes in the following order: 4 2 5 1 6 3 8 7.

A **postorder traversal** visits a given node's two subtrees before visiting the node itself. A postorder traversal of the tree in Figure 13.2 visits the nodes in the following order: 4 5 2 6 8 7 3 1.

### 13.3 Binary Tree Implementation

Class `TreeContainer`, which class `BinaryTree` will specialize, is shown below.

```java
class TreeContainer extends LinkedContainer {
    protected Node root; // Any linked tree has a root node.

    public void clear() {
        root = null;
        super.clear();
    }
}
```

Class `BinaryTree` extends class `TreeContainer` and also contains static member class `BinaryTreeNode`, which subclasses `Node`, and member interface `NodeProcessor`. Any class that realizes interface `NodeProcessor` must offer a public method called `processNode`, which accepts an argument of type `BinaryTreeNode`. This gives our traversals great versatility; we can process all the nodes of a `BinaryTree` in any way we choose simply by coding a class that fulfills the contract set forth by the interface.
public abstract class BinaryTree extends TreeContainer {
    public static abstract class BinaryTreeNode extends Node {
        public BinaryTreeNode leftChild, rightChild, parent;

        public BinaryTreeNode(Object dat) {
            super(dat);
        }

        public BinaryTreeNode(Object dat, BinaryTreeNode lc, BinaryTreeNode rc, BinaryTreeNode par) {
            super(dat);
            leftChild = lc;
            rightChild = rc;
            parent = par;
        }

        public boolean isLeaf() {
            return leftChild == null && rightChild == null;
        }
    }

    public interface NodeProcessor {
        void processNode(BinaryTreeNode node);
    }

    /*
    All of the operations are ultimately carried out recursively. Our pattern of development is to write a recursive helper method and wrap a call(s) to that method in the public method that corresponds to the target operation of the ADT.
    precondition: N/A
    postcondition: The tree’s height has been returned.
    */

    protected int heightHelper(BinaryTreeNode current, int ht) {
        if (current == null)
            return ht;
        return Math.max(heightHelper(current.leftChild, ht + 1), heightHelper(current.rightChild, ht + 1));
    }

    /*
    Method height() corresponds to the binary tree operation of same name. height() simply calls heightHelper(), to which it passes the root node and the height of an empty tree.
    */

    public int height() {
        return heightHelper((BinaryTreeNode)root, -1);
    }
}
protected void inorder(BinaryTreeNode node,
              NodeProcessor processor)
{
    if (node != null)
        inorder(node.leftChild, processor);
    processor.processNode(node);
    if (node != null)
        inorder(node.rightChild, processor);
}

public void inorderTraverse(NodeProcessor processor)
{
    inorder((BinaryTreeNode)root, processor);
}

protected void postorder(BinaryTreeNode node,
              NodeProcessor processor)
{
    if (node != null)
    {
        postorder(node.leftChild, processor);
        postorder(node.rightChild, processor);
    }
    processor.processNode(node);
}

public void postorderTraverse(NodeProcessor processor)
{
    postorder((BinaryTreeNode)root, processor);
}

protected void preorder(BinaryTreeNode node,
            NodeProcessor processor)
{
    processor.processNode(node);
    if (node != null)
    {
        preorder(node.leftChild, processor);
        preorder(node.rightChild, processor);
    }
}

public void preorderTraverse(NodeProcessor processor)
{
    preorder((BinaryTreeNode)root, processor);
}
13.4 Binary Tree Applications

Several important kinds of binary tree are the Huffman coding tree (which is used in data compression), the expression tree, the decision tree (which is used in expert systems), and the binary search tree (BST) (which is the subject of the next chapter). Here we will discuss Huffman coding trees and expression trees.

13.4.1 Huffman Coding

The familiar ASCII and UNICODE character-encoding schemes use fixed-length codes to represent characters; each character of the ASCII set is represented by an eight-bit pattern, and each UNICODE character is represented by a 16-bit pattern. The idea behind Huffman coding is to reduce the space requirement for data by exploiting the fact that some characters are likely to appear more frequently than others. In English-language text, for example, the letter 'e' typically occurs far more often than does 'x' or 'z'. Huffman coding responds to these different frequencies of occurrence by assigning variable-length codes to the various characters; more frequently occurring characters are assigned shorter codes, and letters that occur less frequently are assigned longer codes.

Let us see how Huffman codes are assigned by using the example letter-frequency table shown below. The particular frequencies have been arbitrarily chosen and do not matter; what matters is the frequency of any given letter relative to the frequency of any other letter.

Table 13.1. A sample table of letter frequencies to be used in assigning Huffman codes

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>53</td>
</tr>
<tr>
<td>E</td>
<td>133</td>
</tr>
<tr>
<td>M</td>
<td>60</td>
</tr>
<tr>
<td>U</td>
<td>44</td>
</tr>
<tr>
<td>Z</td>
<td>8</td>
</tr>
</tbody>
</table>

The first step in building a Huffman coding tree is to sort the letters in ascending order by frequency. Each letter will be a leaf in the final tree.
Next, the first two letters, those with the lowest frequencies, are removed from the list and become the children of a new internal node. The internal node has no letter, but its frequency is the sum of the frequencies of its children. This new internal node is then placed on the list so as to maintain the list's sorted ordering.

This process continues until the list has only one element, the root of the final Huffman coding tree. The remaining steps of building our example tree are shown in the following diagrams.
13.4 Binary Tree Applications

![Binary Tree Diagram]

- 105
  - 52
    - 8
    - Z
    - 44
    - U
  - 53
    - C
  - 133
    - E
  - 100
    - M
    - A
- 100
  - 238
    - 105
      - 52
        - 8
        - Z
        - 44
        - U
      - 53
        - C
      - 133
        - E
    - 60
      - M
      - A
Fig. 13.3. The Huffman coding tree corresponding to the letter frequencies from Table 13.1

Notice that in the final diagram each edge to a left child has been labeled with a 0, and each edge to a right child has been labeled with a 1. This labeling of edges is used to determine the Huffman codes; the code for a given letter is the sequence of 0s and 1s encountered on the path from the root to that letter. The table below shows the Huffman codes obtained from the tree of Figure 13.3. This encoding clearly represents a significant savings compared to an ASCII or UNICODE encoding.

Table 13.2. The Huffman codes obtained from the Huffman coding tree of Figure 13.3

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
<th>Code</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>53</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>133</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td>60</td>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>U</td>
<td>44</td>
<td>1001</td>
<td>4</td>
</tr>
<tr>
<td>Z</td>
<td>8</td>
<td>1000</td>
<td>4</td>
</tr>
</tbody>
</table>

Huffman coding is often useful for compressing binary data as well as text. The popular MP3 format, for example, uses Huffman coding.
13.4 Binary Tree Applications

13.4.2 Expression Trees

The binary tree offers an alternative to the stack-based transformation and evaluation of arithmetic expressions described in Chapter 7. Expression trees for the expressions \((3 + 6) \% 7 \^ 5 - 1\) and \(3 + 6 \% 7 \^ 5 - 1\) are shown below in Figure 13.4.

![Expression Trees Diagram]

Fig. 13.4. Two example expression trees; an expression tree's leaves are operands, its internal nodes operators.

An expression tree need not store parentheses because the correct order of operations inheres in the tree's structure. Preorder, inorder, and postorder traversals of an expression tree yield prefix, infix, and postfix expressions, respectively.
Huffman coding trees and expression trees are examples of full binary trees. Any node of a full binary tree has either zero or two children. Some books reverse the definitions of full and complete binary trees presented here.

**Exercises**

1. Add attributes and operations to the class diagram shown at the beginning of the chapter.
2. We implemented `NodeProcessor` as a member interface. Suggest an equivalent alternative.
3. Design and code a class called `NodePrinter` that realizes interface `NodeProcessor`. Class `NodePrinter`'s `processNode` method should output the data field of its target node to a chosen `java.io.PrintStream`. In the next chapter we will implement a concrete subclass of `BinaryTree` that will allow you to test your `NodePrinter` class.
4. Design a node class for a tree whose nodes have an indefinite number of children.
5. Design an algorithm for building an expression tree from an infix expression.
6. How might an expression tree be evaluated?