CHAPTER 26  Trees and Binary Trees

26.1 Trees

A data structure is a container with a fundamental structural relationship associating the items in the container. For instance, a list can be thought of as a container in which the items are sequentially ordered. The ordering is a property of the container: it is not generally a relationship defined on the item class. It is easy to think of examples in which object A comes before object B on one list, but object B comes before A on another list. The ordering is a structural aspect of the list, rather than an inherent property of the objects.

We consider two fundamental and closely related data structures in this chapter: trees and binary trees. These structures are pervasive and commonly found in all areas of computing. Our treatment will be somewhat informal, and we’ll leave much of the detail to the reader.

We describe data structures in terms of abstract entities called “nodes.” We assume that each node carries (a reference to) an item in the container. The “structure” of the data structure is modeled by the relationship between the nodes. For example, we can define a list as a finite set of ordered nodes, which we might picture as follows:

(When we defined classes LinkedList and DoublyLinkedList, we used local classes named Node in each case. The Node objects defined the list structure and contained references to the list items. You can think of “node” concept as used here as an abstraction of these familiar Node objects.)

We could define out structures without introducing nodes, but this is a standard approach and helps distinguish structural properties of the container from properties of the component objects.

We define a tree informally as finite set of nodes, consisting of:

• a distinct root node, and
• a possibly empty list of disjoint trees, called the subtrees of the root.

We mean to imply that the root node is not an element of any of its subtrees, and two distinct subtrees of a root do not share any nodes.
A few observations are in order. First, we have define a tree recursively: in terms of trees. This does not cause problems: a tree with one node has only a root and empty list of subtrees; a tree with \( n > 1 \) nodes is defined in terms of trees with fewer than \( n \) nodes.

Second, note that a tree must contain at least one node. We do not allow an “empty tree” in the definition. However, it is not uncommon for trees to be defined in such as way as to include an empty tree.

Finally, the trees we have defined are sometimes called “rooted” and “ordered,” since there is a distinguished root and the subtrees are ordered by position on a list. If we replaced the list of subtrees with a set of subtrees, we would have an “unordered tree.”

We illustrate trees by representing the nodes with circle, and connecting the root to the roots of its subtrees with lines called “edges.” Figure 26.1 illustrates a tree with nine nodes. The root, \( A \), has two subtrees.

![Tree Illustration](image)

**Figure 26.1 A tree.**

Tree structures occur in many applications. For instance, the directory structure of a file system often exhibits such a structure, and the Java class hierarchy is tree structured.

(Aside: an arbitrary collection of nodes and edges is a graph. If it is possible to get from any node to any other node by following edges, the graph is connected. If it is possible to leave a node, follow edges, and return to the same node without traversing any edge more than once, the graph is cyclic. If the graph is not cyclic, it is acyclic. A tree is sometimes defined to be a connected acyclic graph.)

### 26.1.1 Tree terminology.

Next we introduce some fundamental terminology relating to trees. The examples here all refer to the tree shown in Figure 26.1.
We often use “genealogical” terminology when referring to trees. The root node of a tree is the parent of the root nodes of its subtrees, which are its children. Children of the same parent are siblings. In Figure 26.1, node A is parent of nodes B and C; C is parent of D, E, and F, and so forth. D, E, and F are siblings and children of C.

Transitive closures of the parent and child relations are ancestor and descendant, respectively. Thus, A and C are ancestors of D; D, E, F, G, H, and I are descendents of C.

The number of subtrees a node has is called its degree. Node A has degree 2, node C degree 3, and node E degree 0. A node of degree 0 is a leaf.

A non-empty list of nodes $N_0, \ldots, N_{k-1}$ in which each node $N_i$ is parent of the next node $N_{i+1}$, for $0 \leq i < k-1$, is a path. For instance, A, C, D is a path in the tree shown in Figure 26.1.

If $N_0, \ldots, N_{k-1}$ is a path, its length is $k-1$. The length of the path A, C, D, for example, is 2. Thus the length is the number of edges traversed in going from $N_0$ to $N_{k-1}$. A trivial path consisting of a single node has length 0. Note that there is a unique path from the root to every node in the tree.

The depth of a node is the length of the path from the root to the node. The height of a node is the length of a longest path from the node to a leaf. Thus the depth of node C is 1, and its height is 2. The root has depth 0, and a leaf has height 0. The height of the tree is the height of the root.

The set of nodes at the same depth comprise a level. Level 3, for example, consists of nodes G, H, and I. Siblings are always at the same level, but not all nodes at a given level are siblings.
26.2 Binary trees.

Rather than proceeding with the specification and implementation of classes modeling trees, we consider a closely related data structure called a binary tree. A binary tree is a finite set of nodes that is either empty, or consists of a root node and two disjoint binary trees, called the left and right subtrees of the root.

Notice that the way we have defined things, binary trees are not a subcategory of trees. In particular, we allow empty binary trees but not empty trees. And since we distinguish left and right subtrees, there are two distinct binary tree structures having two nodes: one in which the root has an empty left subtree, and one in which the root has an empty right subtree.

![Figure 26.3 Distinct binary tree structures with two nodes.](image)

We draw binary trees much like trees, occasionally putting in a “tic” to emphasize an empty subtree. We also use similar terminology, modifying the definition of node degree to mean the number of non-empty subtrees.

It is easy to see that a binary tree can have at most $2^n$ nodes at level $n$. A binary tree of height $n$ can have as many as $2^{n+1} - 1$ nodes, and as few as $n+1$. We call a binary tree full if every level, except possibly the last, has as many nodes as possible. The fundamental importance of binary trees is due largely to the fact we can construct binary trees containing $n$ nodes in which the length of the longest path is bounded by $\log_2 n$.

![Figure 26.4 Full and lean binary trees of height 3.](image)
26.3 Defining a class *BinaryTree*.

Let’s see what a class defining binary trees might look like. If we consider the informal recursive
definition given above, we would likely include constructors for building an empty binary tree and
for building a binary tree from a root element and two subtrees:

```java
/**
 * Construct an empty BinaryTree.
 */
public BinaryTree () { … }

/**
 * Construct a BinaryTree with the given root item and left and
 * right subtrees.
 */
public BinaryTree ( Object rootItem,
                   BinaryTree left,
                   BinaryTree right) { … }
```

We would include queries for determining if a binary tree was empty, and for accessing the
constituent parts of a non-empty binary tree:

```java
/**
 * This is the empty BinaryTree.
 */
public boolean isEmpty () { … }

/**
 * The item at the root of the BinaryTree.
 * require:
 * ! isEmpty()
 */
public Object root () { … }

/**
 * The left subtree of the BinaryTree.
 * require:
 * ! isEmpty()
 */
public BinaryTree left () { … }

/**
 * The right subtree of the BinaryTree.
 * require:
 * ! isEmpty()
 */
public BinaryTree right () { … }
```

We would also specify some form of iterator, and methods for modifying the binary tree:
adding and deleting nodes, changing the item at a node, and so on.
Rather than expanding the specification, let’s consider an implementation based directly on
the definition. We subclass BinaryTree to capture the two flavors, empty and non-empty:

```java
class EmptyBinaryTree extends BinaryTree ...
class NonEmptyBinaryTree extends BinaryTree ...
```

A NonEmptyBinaryTree has a root item and two subtrees as components:

```java
class NonEmptyBinaryTree extends BinaryTree {
  ...
  private Object root;  -- the item at the root
  private BinaryTree left;  -- the left subtree
  private BinaryTree right;  -- the right subtree
}
```

This is illustrated in Figure 26.5. (The root item is not shown to reduce clutter.)

![BinaryTree implementation diagram](image)

**Figure 26.5 BinaryTree implementation.**

If we develop this straightforward implementation, we encounter a difficulty: it will be nat-
ural to build the implementation in a way that binary trees share node structure. For instance, if \( t_1 \)
is a binary tree that has already been constructed, the statement

```java
BinaryTree t2 = t1.left();
```

is likely to lead to structure like this:

![Structure sharing diagram](image)

Now \( t_1 \) and \( t_2 \) are not independent in that they share *structure*. If \( t_2 \) is modified, by delet-
ing a node for instance, the modification affects \( t_1 \) as well. Notice that this is a different situation
from two containers containing common items. There is no fundamental problem with an object
simultaneously being on several lists, for instance. Here, we have structural elements in common. Such structural dependencies can become quite complex and lead to errors that are difficult to track down.

Furthermore, with the implementation implied above, we can easily build “binary trees” that fail to satisfy the “disjoint subtree” requirement of the definition. For instance, if \( t_1 \) and \( t_2 \) are as shown above, the statement

```java
BinaryTree t3 = new BinaryTree (o, t1.left(), t2);
```

will create a “binary tree” with identical left and right subtrees:

The situation becomes even worse if we include commands that allow a subtree of an existing tree to be replace with another existing tree, such as:

```java
/**
 * Replace the left subtree of this BinaryTree with the specified
 * BinaryTree.
 */
public void setLeft (BinaryTree t) { … }
```

With this command, we can create cycles in the tree structure. For example, with \( t_1 \) and \( t_2 \) as shown above, if we write:

```java
t2.setLeft(t1);
```

we end up with a structure like this:
How can we address this situation? There are several possible approaches.

1. Do not provide methods that modify structure in a problematic way. For instance, binary tree and iterator classes would not include methods for inserting or deleting nodes, or methods such as `setLeft` for modifying tree structure. Then even though binary trees might share nodes, and even though subtrees of a given tree might not be disjoint, no problems would arise since the structure sharing is not detectable by a client. While this approach is sometimes taken, most developers find such restrictions unacceptable in a library class.

2. Duplicate the subtree structure as necessary to avoid sharing nodes. For example, the constructor for a non-empty `BinaryTree` could make copies of the `BinaryTree`'s provided as arguments. While this would certainly solve the problem, the overhead is prohibitive.

3. Pass the responsibility for maintaining structural coherency to the client. That is, the implementor make no guarantees about structural independence, disjoint subtrees, etc. It is left to the clients to use or misuse instances of the class as they see fit. This is probably the most commonly adopted approach. It is straightforward, certainly easiest on the implementor, and not as dangerous as it first seems. Applications that employ binary trees tend to use them as private components of some more abstract construct, and structural manipulation of the trees is localized. Thus there is little actual danger of an unanticipated client destroying the coher-ence of a tree structure.

In the library associated with the text, we develop a somewhat different approach. Specifically, we guarantee that distinct `BinaryTree` instances do not share nodes. This is accomplished by either copying nodes, or by removing nodes from one tree if they are used to build another. For instance, one pair of methods for obtaining left and right subtrees remove the subtrees from the original tree:

```java
/**
 * Remove and return the left subtree of this BinaryTree.
 * This is not a proper query since it changes the state of this
 * BinaryTree to insure that two BinaryTree instances do not
 * share nodes.
 * require:
 *    !this.isEmpty()
 * ensure:
 *    this.emptyLeft()
 *    this.setLeft(result).equals(old)
 */
 public BinaryTree getLeft () { ... }
```

For example, the statement

```java
BinaryTree t2 = t1.getLeft();
```

results in the action shown at the top of the following page.

Another pair of methods copies of the subtrees:
public BinaryTree copyLeft () { … }

One pair of methods for setting left and right subtrees “steal” the nodes of the argument:

public void setLeft (BinaryTree left) { … }

For instance,

t1.setLeft(t2);

results in t1 capturing the nodes of t2, as shown on the following page. Another pair of methods make copies of the argument. For instance,

public void copyToLeft (BinaryTree left) { … }

/**
 * A copy of the left subtree of this BinaryTree.
 * require:
 * !this.isEmpty()
 * ensure:
 * this.copyLeft().equals(this.getLeft())
 */

/**
 * Replace the left subtree with of this BinaryTree with the
 * specified BinaryTree. To insure that two BinaryTree instances
 * do not share nodes, left.isEmpty() upon completion.
 * require:
 * !this.isEmpty()
 * left != null
 * left != this
 * ensure:
 * this.getLeft().equals(old.left)
 * left.isEmpty()
 */
This approach has its own set of problems. In particular, method arguments are modified, and methods like `getLeft` are “hybrid”: neither proper queries nor proper commands.

We postpone the complete specification until after discussing iterators.

### 26.4 Binary tree iterators.

As with other containers, we would like to have a way of sequentially accessing each item in a binary tree. The solution in the case of lists was obvious: start with the first element and proceed to the last. Binary trees, however, have a more complex structure and there is no single obvious way of sequencing through the items of a binary tree.
We consider three approaches based directly on the recursive definition of a binary tree. Recall that a binary tree is either empty, or consists of a root plus left and right subtrees. Traversing the empty binary tree is trivial – there is nothing to do. If the binary tree is not empty, we get different linearizations depending on the order in which we access the three components – root, left and right subtrees.

A **preorder** traversal of a non-empty binary tree

- visits the root; then
- preorder traverses the left subtree; then
- preorder traverses the right subtree.

This recursive description is sound since each subtree has fewer nodes than the binary tree of which it is a part, and traversing the empty binary tree is trivial. If we preorder traverse the following binary tree

![Binary Tree](image)

we access the nodes in the order $A, B, D, E, C, F, G$: $A$ is the root node; $B, D, E$ is the left subtree traversed in preorder; and $C, F, G$ is the right subtree in preorder.

A **postorder** traversal of a non-empty binary tree

- postorder traverses the left subtree; then
- postorder traverses the right subtree; then
- visits the root.

The binary tree shown traversed in postorder yields the linearization $D, E, B, G, F, C, A$.

A **inorder** traversal of a non-empty binary tree

- inorder traverses the left subtree; then
- visits the root; then
- inorder traverses the right subtree.

An inorder traversal of the above binary tree visits nodes in the order $D, B, E, A, C, G, F$.

We can, of course, produce further variations by traversing the right subtree before the left. And there are important orderings not based on the recursive definition. For instance, a *breadth-first* or *level-order* traversal visits the nodes of each level before moving to the next level. A level-order traversal of the binary tree illustrated above visits nodes in the order $A, B, C, D, E, F, G$.

Recall from Chapter 24 that the interface `Iterator` specified methods for setting the iterator to the first item in the container, and for advancing through the container:
We would like to define iterators that advance through a binary tree in either preorder, postorder, or inorder fashion. Thus we need to see how to find the “first” and “next” node of a binary tree in each of these ordering.

26.4.1 Inorder traversal.

We start with inorder, since it is probably the easiest to see. Recall that an inorder traversal is essentially:

- traverse the left subtree;
- visit the root;
- traverse the right subtree.

To find the first node, we move left down the binary tree as far as possible:

Note that at any point during the traversal, the left subtree of the current node will have been visited. In fact, the left subtree of any visited node has been visited. Also, any ancestor whose right subtree the current node is in will have been visited:
So how do we find the next node after the current?

- If the current node has a non-empty right subtree, the next node is the first node in that subtree:
  ![Diagram: current node with an empty right subtree]

- If the current node has an empty right subtree, the next node is the closest ancestor not yet visited; that is, the closest ancestor whose left subtree the current node is in:
  ![Diagram: current node with an empty right subtree]

If there is no such ancestor, the current node is the last node to be visited.

Advancing to the next node sometimes involves moving down to lower levels, and sometimes involves moving up to higher levels.

### 26.4.2 Preorder traversal.

Next, we consider preorder traversal. Recall that a preorder traversal is essentially:

- visit the root;
  - traverse the left subtree;
  - traverse the right subtree.

Finding the first preorder node is easy: it’s the root of the tree.

At any point in the traversal, all ancestors of the current node have been visited. Also, if the current node is in the right subtree of an ancestor node, the left subtree of that node has been visited:
To find the next node in a preorder traversal, there are three cases to consider. Two are easy to see; the third requires a little thought.

- If the current node has a non-empty left subtree, the next node is the root of the left subtree.
- If the current node has an empty left subtree, but a non-empty right subtree, it is the root of the right subtree.
- If the current node is a leaf, find the closest ancestor whose left subtree the current node is in, and that has a non-empty right subtree. The next node is the root of this right subtree.

26.4.3 Postorder traversal.

Finally, recall that a postorder traversal of a non-empty tree proceeds as:

- traverse the left subtree;
- traverse the right subtree;
- visit the root.

To find the first node, move down the tree from the root, moving left whenever possible, until a leaf is reached:

At any point in the traversal, all descendant of the current node have been visited, but none of the ancestors. If the current node is in the right subtree of an ancestor node, the left subtree of that node has been visited:
The are three cases to consider in finding the next node.

- If the current node is a right child, the next node is the parent.
- If the current node is a left child and the parent has an empty right subtree, the next node is the parent.
- If the current node is a left child, and the parent has a non-empty right subtree, the next node is the first node in that right subtree:

### 26.5 Implementing binary trees.

As with lists, there are many possible approaches to implementing binary trees. We sketch one array-based implementation several variations on a linked implementation. For the most part, we leave details to the reader.

#### 26.5.1 A simple array-based implementation.

To implement a binary tree with an array, we must linearize the tree, but in such a way that the tree structure is not lost. That is, we must map nodes to array elements, but in such a way that for any given node, we can determine whether or not its subtrees are empty, and if not, we can locate their roots.
One simple approach is based on a level-ordering of the binary tree. The root of the binary
tree is stored in array element with index 0. (Of course, what we mean is that a reference to the
root element is stored in the array.) If a node is stored in array element \( n \), its left child is stored in
element \( 2n+1 \), and its right child in element \( 2n+2 \). An empty subtree can be denoted by an array
element containing the null value.

It is easy to see that this approach allocates a unique array element to each tree node. For
instance, the tree pictured below would be stored as shown. Node \( B \), for instance, in stored in
array element 1. Thus \( B \)'s left child would be stored in element \( 2 \cdot 1 + 1 = 3 \), and \( B \)'s right child in
element \( 2 \cdot 1 + 2 = 4 \).

One disadvantage to this method is that space must be allocated for all possible nodes at a
given level, whether the level is fully populated or not.

### 26.5.2 Linked implementations.

Linked binary tree implementations are can be constructed in a similar manner to linked list
implementations. The implementation suggested in Section 26.3 is essentially a linked implement-
ation.

In the most straightforward approach, we define an implementation class \texttt{Node} that contains
the \texttt{Node} element and references to the children:

```java
private class Node {
    Object element;
    Node left;
    Node right;
}
```

The binary tree is constructed by linking \texttt{Node}s together. The
\texttt{BinaryTreeImplementation} class contains a reference to the root node of the binary tree, as
shown in Figure 26.9.

Just as there are various on the basic linked list structure – lists with header nodes, circular
lists, two-way linked lists, \textit{etc}. – there are variations on the simple linked structure for binary
trees. For instance, we could include an “upward pointer” in a \texttt{Node} referencing its parent:
private class Node {
    Object element;
    Node parent;
    Node left;
    Node right;
}

This reference could be null for the root Node, or could reference a header.

26.5.3 Threaded binary trees.

A useful variation arises if we try to find a binary tree structure analogous to circular lists. In building a circular list, we replaced the null reference at the end of the list with a reference back to the first node of the list. If we examine a linked implementation of a binary tree, we notice that more than half of the child-referencing components (right and left) contain null values. (If there are $n$ nodes in the tree, there are $2n$ child-referencing components. Only $n-1$ of these are non-null, since there are only $n-1$ children in the tree.)

We could replace these null references with references that point back up into the tree – but to what should they point? We could point them back to the root. But another alternative presents itself if we carefully examine the inorder traversal discussed above. A node’s right child is null exactly when the next inorder node (the inorder successor) is an ancestor. Similarly, a node’s left child is null exactly when the previous inorder node (the inorder predecessor) is an ancestor. We can obtain a useful structure by replacing null left references with references to a node’s inorder predecessor, and replacing null right references with references to a node’s inorder successor.
The upward references to ancestors are called *threads*, and a binary tree structured in this way is referred to as a *threaded binary tree*.

We need some way in a node to distinguish between references that point to a child, and references that point to an ancestor. We’ll adopt the simple expedient of using *boolean* components:

```java
declared private class Node {
    Object element;
    boolean hasLeftChild;
    boolean hasRightChild;
    Node left; // left child if hasLeftChild;
               // inorder predecessor if !hasLeftChild
    Node right; // right child if hasRightChild;
               // inorder successor if !hasRightChild
}
```

Threaded binary trees are typically implemented with head nodes. The head node serves as the inorder predecessor of the first inorder node in the binary tree, and as the inorder successor of the last inorder node. We would like to design the head node to avoid special cases in traversal methods. For instance, advancing to the “next” preorder node from the head should take us to the root, which is the first preorder node; advancing to the “next” preorder node from the last preorder node should take us back to the head.

Things work out nicely with the following head node structure, where head denotes the head node:

- for the empty binary tree:
  ```java
  head.hasLeftChild == false  head.left == head
  head.hasRightChild == true  head.right == head
  ```

- for a non-empty binary tree:
  ```java
  head.hasLeftChild == true    head.left == rootNode
  head.hasRightChild == true   head.right == head
  ```

### 26.6 Implementing traversers.

We sketch briefly how binary tree traversers might work for several implementations. Specifically, we consider how the method *advance* can be implemented for a traverser that is traversing a binary tree in preorder fashion. In each case, we assume that we are working in an implementation, and that we have direct access to the underlying node structure of the binary tree implementation.

We further assume that the traverser has a component variable that references the current node:

```java
private Node current;  // Node with the current item
```
The method `advance` requires that we have not completed the traversal, and updates the variable `current` to reference the next preorder node:

```java
/**
 * Advance to the next item in preorder traversal.
 * require:
 * !this.done()
 */
public void advance () {
    // modify current to reference the next preorder node.
}
```

The fundamental problem is finding nodes that are ancestors to the current node. We consider three implementations.

### 26.6.1 Simple linked implementation

In the most straightforward implementation, a node simply contains references to its children:

```java
private class Node {
    Object element;
    Node left;      // left child; null if no left child
    Node right;     // right child; null if no right child
}
```

In such a situation, it is handy to keep track of the path from the root to the current `Node`. We can do this with a stack of `Node` references, as shown in Figure 26.10.

![Figure 26.10 Node stack keeps path from root to current Node.](image)

In such a situation, it is handy to keep track of the path from the root to the current `Node`. We can do this with a stack of `Node` references, as shown in Figure 26.10.

Node references can be pushed onto the stack as we move down the binary tree, and can be popped off as we move back up.

If we look at the cases for finding the next preorder node, we see that we don’t need to keep the entire path on the stack. The only ancestor nodes we need to get to are those ancestors whose left subtree we’re in, and that have a non-empty right subtree. In fact, if these are the only nodes that get pushed onto the stack, the ancestor we’re looking for will be on the top of the stack exactly when we need it.
We can now write the method, assuming we have a class `NodeStack` and a `NodeStack` component variable named `path`:

```java
private class NodeStack extends Stack { … }

private NodeStack path; // path contains ancestors of current
// for which current is in the left subtree
// and that have non-empty right subtrees.
```

The algorithm simply handles the three cases for finding the next preorder `Node`:

```java
public void advance () {
    if (current.left != null) {
        if (current.right != null)
            path.push(current);
        current = current.left;
    } else if (current.right != null)
        current = current.right;
    else {
        Node parent = path.top();
        path.pop();
        current = parent.right;
    }
}
```

### 26.6.2 Nodes with backwards references.

If nodes contain references to their parent, moving back up the binary tree is trivial. The algorithm has a little poorer worst-case behavior, though.

Assume the following `Node` structure:

```java
private class Node {
    Object element;
    Node parent; // parent; null if root
    Node left; // left child; null if no left child
    Node right; // right child; null if no right child
}
```

The algorithm is straightforward.
26.6.3 Threaded binary tree.

Our last example uses a threaded binary tree, as described in Section 26.5.3. Note that in a threaded binary tree, the right link of a node with no right child references the closest ancestor whose left subtree the node is in. We use this to work our way back up the tree when necessary.

```
public void advance () {
    if (current.left != null)
        current = current.left;
    else if (current.right != null)
        current = current.right;
    else {
        Node previous = current;
        current = current.parent;
        while (current != null &&
               (current.right == previous || current.right == null)) {
            previous = current;
            current = current.parent;
        }
        if (current != null)
            current = current.right;
    }
}
```

Note that if current references the last preorder Node, advance moves current to the head Node.

26.6.4 Recursive internal traversers

Recall from Section 21.3.4 that an internal or passive iterator applies an operation to every item in a container that satisfied a specified condition. Given the specifications

```
public interface Predicate {
    public boolean execute (Object obj);
}
```

```
public interface Operation {
    public void execute (Object obj);
}
```
an internal iterator can be specified something like this:

```java
public class InternalIterator {

    /**
     * Create a new iterator for the specified Container,
     */
    public InternalIterator (Container l, Predicate p, Operation op) {
        ...
    }

    /**
     * Apply the operation to each element of the Container that
     * satisfies the predicate.
     */
    public void traverse () {
        ...
    }
}
```

Now we can define simple recursive implementations of the method `traverse` based directly on the recursive definition of a binary tree.

For instance, suppose we want an inorder traversal, and the binary tree implementation is as given in Section 26.5.2. Again, we assume that we are working in an implementation and have direct access to the underlying node structure. Let \( t \) be the binary tree we want to traverse, \( \text{op} \) the operation to perform on each item that satisfies the predicate \( p \):

```java
private LinkedBinaryTree t;
private Operation op;
private Predicate p;
```

We can write the following recursive method. Note that the recursive method is actually a private auxiliary method, in this case taking a `Node` as argument.

```java
public void traverse () {
    traverse(t.root);
}

/**
 * Inorder traverse the tree with root at Node n.
 */
private void traverse (Node n) {
    if (n != null) {
        traverse(n.left);
        if (p.execute(n.element))
            op.execute(n.element);
        traverse(n.right);
    }
}
```
26.7 Modifying tree structure: path traversers

In many binary tree applications, it is necessary to be able to manipulate the tree structure by adding, removing, and rearranging nodes. In order to accomplish this, and maintain our requirement that binary trees not share nodes, we introduce a traversing object called a *path traverser*. Rather than proceeding through every item in the tree, a path traverser starts at the root and makes its way down a path in the binary tree.

We specify the class essentially as follows. The complete specification can be found in the library documentation.

```java
public interface PathTraverser extends Cloneable {

/**
 * PathTraverser has advanced to an empty subtree; no current node.
 */
public boolean done ();

/**
 * Current node has empty left subtree.
 * require:
 *  !this.done()
 */
public boolean emptyLeft ();

/**
 * Current node has empty right subtree.
 * require:
 *  !this.done()
 */
public boolean emptyRight ();

/**
 * Item at the current node.
 * require:
 *  !this.done()
 */
public Object get ();

/**
 * Advance to the left child.
 * require:
 *  !this.done()
 * ensure:
 *  if old.emptyLeft(), then this.done()
 */
public void advanceLeft ();

```
/**
 * Advance to the right child.
 * require:
 * !this.done()
 * ensure:
 * if old.emptyRight(), then this.done()
 */
 public void advanceRight ();

} // end of interface PathTraverser

We use PathTraverser’s for modifying tree structure. Specifically, we equip a BinaryTree with methods for creating a PathTraverser, and for using a PathTraverser to modify its structure. (As before, the full specifications can be found in library documentation.)

public class BinaryTree {

 /**
 * Create a new pathTraverser for this BinaryTree.
 */
 public PathTraverser createPathTraverser () { …

 /**
 * Replace the item at the node referenced by the PathTraverser
 * to the specified item.
 */
 public void set (PathTraverser t, Object obj) { …

 /**
 * Insert a new node as the left child of the node referenced
 * by the PathTraverser.
 */
 public void insertLeft (PathTraverser t, Object obj) { …

 /**
 * Insert a new node as the right child of the node referenced
 * by the PathTraverser.
 */
 public void insertRight (PathTraverser t, Object obj) { …

 /**
 * Delete the node referenced by the PathTraverser.
 */
 public void remove (PathTraverser t) { …
Behaviors of the methods change, insertLeft, and insertRight are easy to see: change doesn't really affect the structure of the tree; existing left or right subtrees simply move down a level when a new node is inserted. For instance, if a PathTraverser is referencing node B in the tree shown below, and insertLeft is done, the old left child of B becomes the left child of the newly inserted node:

![Diagram](image)

The method remove is a little more problematic. If the node to be deleted is leaf, or has a single non-empty subtree, there is an obvious behavior.

![Diagram](image)

But how should we handle the case where the node to be deleted has two non-empty children? We will modify the tree so as to maintain its relative inorder sequence of nodes. (We must wait until the next chapter to see why this is a reasonable approach.)
For example, suppose we want to delete the node $B$ in the following tree.

![Tree Diagram]

The inorder sequence of this tree is $A, B, C, D, E, F, G, H, I, J$. We would like the inorder sequence of the resulting tree after the deletion to be $A, C, D, E, F, G, H, I, J$.

We can accomplish this by “replacing” the node to be deleted with the first inorder node of its right subtree. (Equally well, we could use the last inorder node of its left subtree.) This node, $C$ in the example, might have a non-empty right subtree, but has an empty left subtree. We let the node’s parent inherit its right subtree. For instance, after deletion, the example tree will look like this:

![Tree Diagram 2]

(There is a degenerative case in which the first inorder node is the root of the tree. Handling this case is rather obvious.)
26.8 Implementing trees as binary trees

Finally, we return to the general trees we considered at the start of the chapter. We will see a common method of implementing such trees with binary trees.

Recall that a tree has a root and possibly empty list of subtrees. We can represent a tree with a binary tree by letting the left child of the root reference the (root of the) first subtree, and letting the right children create the list. That is, the right child of a node is the root of the next tree on the list.

Consider, for example, the tree illustrated in Figure 26.1:

We create a binary tree with $A$ as the root. The left child of $A$ is the root of its first subtree, $B$. Similarly, the left child of $C$ is $D$, the left child of $D$ is $G$, etc.
Next, the list of subtrees of $A$ is constructed by linking the roots ($B$ and $C$) together as parent-right child. Similarly, the roots of the subtrees of $C$ and of $D$ are linked parent to right-child.

Thus, the left child of a node in the binary tree is the “first child” in the original tree. The right child in the binary tree is the “next sibling” in the original tree.