7.3.4 A Linked Structure for Binary Trees

As with a general tree, a natural way to realize a binary tree T is to use a ***linked structure***, where we represent each node v of T by a position object (see Figure 7.14a) with fields providing references to the element stored at v and to the position objects associated with the children and parent of v. If v is the root of T, then the parent field of v is null. If v has no left child, then the left field of v is null. If v has no right child, then the right field of v is null. Also, we store the number of nodes of T in a variable, called size. We show the linked structure representation of a binary tree in Figure 7.14b.

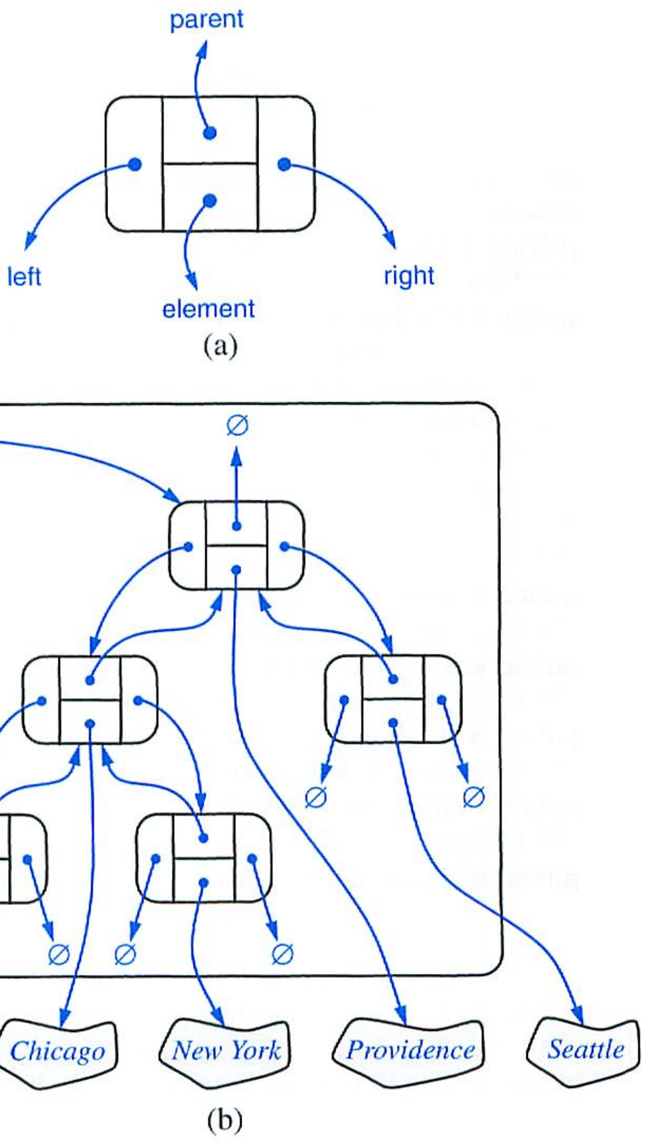

Figure 7.14: A node (a) and a linked structure (b) for representing a binary tree.

Java Implementation of a Binary Tree Node

We use a Java interface BTPosition (not shown) to represent a node of a binary tree. This interfaces extends Position, thus inheriting method element, and has additional methods for setting the element stored at the node (setElement) and for setting and returning the left child (setLeft and getLeft), right child (setRight and getRight), and parent (setParent and getParent) of the node. Class BTNode (Code Fragment 7.15) implements interface BTPosition by an object with fields element, left, right, and parent, which, for a node v, reference the element at v, the left child of v, the right child of v, and the parent of v, respectively.

```
/**
 * Class implementing a node of a binary tree by storing references to
 * an element, a parent node, a left node, and a right node.
 */
public class BTNode<E> implements BTPosition<E> {
  private E element;  // element stored at this node
  private BTPosition<E> left, right, parent;  // adjacent nodes
  /** Main constructor */
  public BTNode(E element, BTPosition<E> parent,
              BTPosition<E> left, BTPosition<E> right) {
    setElement(element);
    setParent(parent);
    setLeft(left);
    setRight(right);
  }
  /** Returns the element stored at this position */
  public E element() { return element; }
  /** Sets the element stored at this position */
  public void setElement(E o) { element=o; }
  /** Returns the left child of this position */
  public BTPosition<E> getLeft() { return left; }
  /** Sets the left child of this position */
  public void setLeft(BTPosition<E> v) { left=v; }
  /** Returns the right child of this position */
  public BTPosition<E> getRight() { return right; }
  /** Sets the right child of this position */
  public void setRight(BTPosition<E> v) { right=v; }
  /** Returns the parent of this position */
  public BTPosition<E> getParent() { return parent; }
  /** Sets the parent of this position */
  public void setParent(BTPosition<E> v) { parent=v; }
}
```

Code Fragment 7.15: Auxiliary class BTNode for implementing binary tree nodes.

Java Implementation of the Linked Binary Tree Structure

In Code Fragments 7.16–7.18, we show portions of class LinkedBinaryTree that implements the BinaryTree interface (Code Fragment 7.14) using a linked data structure. This class stores the size of the tree and a reference to the BTNode object associated with the root of the tree in internal variables. In addition to the BinaryTree interface methods, LinkedBinaryTree has various other methods, including accessor method sibling(v), which returns the sibling of a node v, and the following update methods:

addRoot(e):	Create and return a new node r storing element e and make r the root of the tree; an error occurs if the tree is not empty.
insertLeft(v,e):	Create and return a new node w storing element e, add w as the the left child of v and return w; an error occurs if v already has a left child.
insertRight(v,e):	Create and return a new node z storing element e, add z as the the right child of v and return z; an error occurs if v already has a right child.
remove(v):	Remove node v, replace it with its child, if any, and return the element stored at v; an error occurs if v has two children.
attach(v,T_1,T_2):	Attach T_1 and T_2, respectively, as the left and right subtrees of the external node v; an error condition occurs if v is not external.

Class LinkedBinaryTree has a constructor with no arguments that returns an empty binary tree. Starting from this empty tree, we can build any binary tree by creating the first node with method addRoot and repeatedly applying the insertLeft and insertRight methods and/or the attach method. Likewise, we can dismantle any binary tree T using the remove operation, ultimately reducing such a tree T to an empty binary tree.

When a position v is passed as an argument to one of the methods of class LinkedBinaryTree, its validity is checked by calling an auxiliary helper method, checkPosition(v). A list of the nodes visited in a preorder traversal of the tree is constructed by recursive method preorderPositions. Error conditions are indicated by throwing exceptions InvalidPositionException, BoundaryViolationException, EmptyTreeException, and NonEmptyTreeException.

```
/**
 * An implementation of the BinaryTree interface by means of a linked structure.
 */
public class LinkedBinaryTree<E> implements BinaryTree<E> {
  protected BTPosition<E> root; // reference to the root
  protected int size;           // number of nodes
  /** Creates an empty binary tree. */
  public LinkedBinaryTree() {
    root = null;  // start with an empty tree
    size = 0;
  }
  /** Returns the number of nodes in the tree. */
  public int size() {
    return size;
  }
  /** Returns whether a node is internal. */
  public boolean isInternal(Position<E> v) throws InvalidPositionException {
    checkPosition(v);           // auxiliary method
    return (hasLeft(v) || hasRight(v));
  }
  /** Returns whether a node is the root. */
  public boolean isRoot(Position<E> v) throws InvalidPositionException {
    checkPosition(v);
    return (v == root());
  }
  /** Returns whether a node has a left child. */
  public boolean hasLeft(Position<E> v) throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);
    return (vv.getLeft() != null);
  }
  /** Returns the root of the tree. */
  public Position<E> root() throws EmptyTreeException {
    if (root == null)
      throw new EmptyTreeException("The tree is empty");
    return root;
  }
  /** Returns the left child of a node. */
  public Position<E> left(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException {
    BTPosition<E> vv = checkPosition(v);
    Position<E> leftPos = vv.getLeft();
    if (leftPos == null)
      throw new BoundaryViolationException("No left child");
    return leftPos;
  }
```

Code Fragment 7.16: Portions of the LinkedBinaryTree class, which implements the BinaryTree interface. (Continues in Code Fragment 7.17.)

```
/** Returns the parent of a node. */
public Position<E> parent(Position<E> v)
  throws InvalidPositionException, BoundaryViolationException {
  BTPosition<E> vv = checkPosition(v);
  Position<E> parentPos = vv.getParent();
  if (parentPos == null)
    throw new BoundaryViolationException("No parent");
  return parentPos;
}
/** Returns an iterable collection of the children of a node. */
public Iterable<Position<E>> children(Position<E> v)
  throws InvalidPositionException {
 PositionList<Position<E>> children = new NodePositionList<Position<E>>();
  if (hasLeft(v))
    children.addLast(left(v));
  if (hasRight(v))
    children.addLast(right(v));
  return children;
}
/** Returns an iterable collection of the tree nodes. */
public Iterable<Position<E>> positions() {
 PositionList<Position<E>> positions = new NodePositionList<Position<E>>();
  if(size != 0)
    preorderPositions(root(), positions);  // assign positions in preorder
  return positions;
}
/** Returns an iterator of the elements stored at the nodes */
public Iterator<E> iterator() {
  Iterable<Position<E>> positions = positions();
 PositionList<E> elements = new NodePositionList<E>();
 for (Position<E> pos: positions)
   elements.addLast(pos.element());
  return elements.iterator();  // An iterator of elements
}
/** Replaces the element at a node. */
public E replace(Position<E> v, E o)
  throws InvalidPositionException {
  BTPosition<E> vv = checkPosition(v);
  E temp = v.element();
  vv.setElement(o);
  return temp;
}
```

Code Fragment 7.17: Portions of the LinkedBinaryTree class, which implements the BinaryTree interface. (Continues in Code Fragment 7.18.)

```
// Additional accessor method
/** Return the sibling of a node */
public Position<E> sibling(Position<E> v)
  throws InvalidPositionException, BoundaryViolationException {
    BTPosition<E> vv = checkPosition(v);
    BTPosition<E> parentPos = vv.getParent();
    if (parentPos != null) {
      BTPosition<E> sibPos;
      BTPosition<E> leftPos = parentPos.getLeft();
      if (leftPos == vv)
        sibPos = parentPos.getRight();
      else
        sibPos = parentPos.getLeft();
      if (sibPos != null)
        return sibPos;
    }
    throw new BoundaryViolationException("No sibling");
}
// Additional update methods
/** Adds a root node to an empty tree */
public Position<E> addRoot(E e) throws NonEmptyTreeException {
  if(!isEmpty())
    throw new NonEmptyTreeException("Tree already has a root");
  size = 1;
  root = createNode(e,null,null,null);
  return root;
}
/** Inserts a left child at a given node. */
public Position<E>  insertLeft(Position<E> v, E e)
  throws InvalidPositionException {
  BTPosition<E> vv = checkPosition(v);
  Position<E> leftPos = vv.getLeft();
  if (leftPos != null)
    throw new InvalidPositionException("Node already has a left child");
  BTPosition<E> ww = createNode(e, vv, null, null);
  vv.setLeft(ww);
  size++;
  return ww;
}
```

Code Fragment 7.18: Portions of the LinkedBinaryTree class, which implements the BinaryTree interface. (Continues in Code Fragment 7.19.)

```
/** Removes a node with zero or one child. */
public E remove(Position<E> v)
  throws InvalidPositionException {
  BTPosition<E> vv = checkPosition(v);
  BTPosition<E> leftPos = vv.getLeft();
  BTPosition<E> rightPos = vv.getRight();
  if (leftPos != null && rightPos != null)
    throw new InvalidPositionException("Cannot remove node with two children");
  BTPosition<E> ww;  // the only child of v, if any
  if (leftPos != null)
    ww = leftPos;
  else if (rightPos != null)
    ww = rightPos;
  else                // v is a leaf
    ww = null;
  if (vv == root) {   // v is the root
    if (ww != null)
      ww.setParent(null);
    root = ww;
  }
  else {              // v is not the root
    BTPosition<E> uu = vv.getParent();
    if (vv == uu.getLeft())
      uu.setLeft(ww);
    else
      uu.setRight(ww);
    if(ww != null)
      ww.setParent(uu);
  }
  size--;
  return v.element();
}
```

Code Fragment 7.19: Portions of the LinkedBinaryTree class, which implements the BinaryTree interface. (Continues in Code Fragment 7.20.)

```
/** Attaches two trees to be subtrees of an external node. */
public void attach(Position<E> v, BinaryTree<E> T1, BinaryTree<E> T2)
  throws InvalidPositionException {
  BTPosition<E> vv = checkPosition(v);
  if (isInternal(v))
    throw new InvalidPositionException("Cannot attach from internal node");
  if (!T1.isEmpty()) {
    BTPosition<E> r1 = checkPosition(T1.root());
    vv.setLeft(r1);
    r1.setParent(vv);         // T1 should be invalidated
  }
  if (!T2.isEmpty()) {
    BTPosition<E> r2 = checkPosition(T2.root());
    vv.setRight(r2);
    r2.setParent(vv);         // T2 should be invalidated
  }
}
/** If v is a good binary tree node, cast to BTPosition, else throw exception */
protected BTPosition<E> checkPosition(Position<E> v)
  throws InvalidPositionException {
  if (v == null || !(v instanceof BTPosition))
    throw new InvalidPositionException("The position is invalid");
  return (BTPosition<E>) v;
}
/** Creates a new binary tree node */
protected BTPosition<E> createNode(E element, BTPosition<E> parent,
                                  BTPosition<E> left, BTPosition<E> right) {
  return new BTNode<E>(element,parent,left,right); }
/** Creates a list storing the the nodes in the subtree of a node,
  * ordered according to the preorder traversal of the subtree. */
protected void preorderPositions(Position<E> v, PositionList<Position<E>> pos)
  throws InvalidPositionException {
  pos.addLast(v);
  if (hasLeft(v))
    preorderPositions(left(v), pos);   // recurse on left child
  if (hasRight(v))
    preorderPositions(right(v), pos);  // recurse on right child
}
```

Code Fragment 7.20: Portions of the LinkedBinaryTree class, which implements the BinaryTree interface. (Continued from Code Fragment 7.19.)

Performance of the LinkedBinaryTree Implementation

Let us now analyze the running times of the methods of class LinkedBinaryTree, which uses a linked structure representation:

- Methods size() and isEmpty() use an instance variable storing the number of nodes of T, and each take $O(1)$ time.
- The accessor methods root, left, right, sibling and parent take $O(1)$ time.
- Method replace(v, e) takes $O(1)$ time.
- Methods iterator() and positions() are implemented by performing a preorder traversal of the tree (using the auxiliary method preorderPositions). The nodes visited by the traversal are stored in a position list implemented by class NodePositionList (Section 6.2.4) and the output iterator is generated with method iterator() of class NodePositionList. Methods iterator() and positions() take $O(n)$ time and methods hasNext() and next() of the returned iterators run in $O(1)$ time.
- Method children uses a similar approach to construct the returned iterable collection, but it runs in $O(1)$ time, since there are at most two children for any node in a binary tree.
- The update methods insertLeft, insertRight, attach, and remove all run in $O(1)$ time, as they involve constant-time manipulation of a constant number of nodes.

Considering the space required by this data structure for a tree with n nodes, note that there is an object of class BTNode (Code Fragment 7.15) for every node of tree T. Thus, the overall space requirement is $O(n)$. Table 7.2 summarizes the performance of the linked structure implementation of a binary tree.

Operation	Time
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent, children, left, right, sibling	$O(1)$
hasLeft, hasRight, isInternal, isExternal, isRoot	$O(1)$
insertLeft, insertRight, attach, remove	$O(1)$

Table 7.2: Running times for the methods of an n-node binary tree implemented with a linked structure. Methods hasNext() and next() of the iterators returned by iterator(), positions().iterator(), and children(v).iterator() run in $O(1)$ time. The space usage is $O(n)$.

7.3.5 An Array-List Representation of a Binary Tree

An alternative representation of a binary tree T is based on a way of numbering the nodes of T. For every node v of T, let $p(v)$ be the integer defined as follows.

- If v is the root of T, then $p(v) = 1$.
- If v is the left child of node u, then $p(v) = 2p(u)$.
- If v is the right child of node u, then $p(v) = 2p(u) + 1$.

The numbering function p is known as a ***level numbering*** of the nodes in a binary tree T, for it numbers the nodes on each level of T in increasing order from left to right, although it may skip some numbers. (See Figure 7.15.)

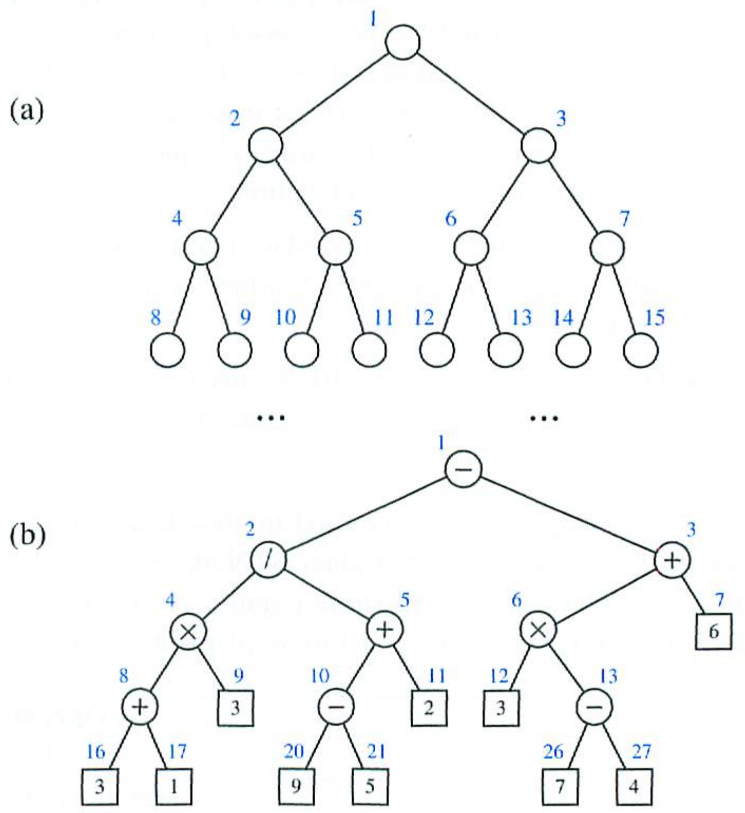

Figure 7.15: Binary tree level numbering: (a) general scheme; (b) an example.

The level numbering function p suggests a representation of a binary tree T by means of an array list S such that node v of T is the element of S at index $p(v)$. As mentioned in the previous chapter, we realize the array list S by means of an extendable array. (See Section 6.1.4.) Such an implementation is simple and efficient, for we can use it to easily perform the methods root, parent, left, right, hasLeft, hasRight, isInternal, isExternal, and isRoot by using simple arithmetic operations on the numbers $p(v)$ associated with each node v involved in the operation. We leave the details of this implementation as an exercise (R-7.26).

We show an example array-list representation of a binary tree in Figure 7.16.

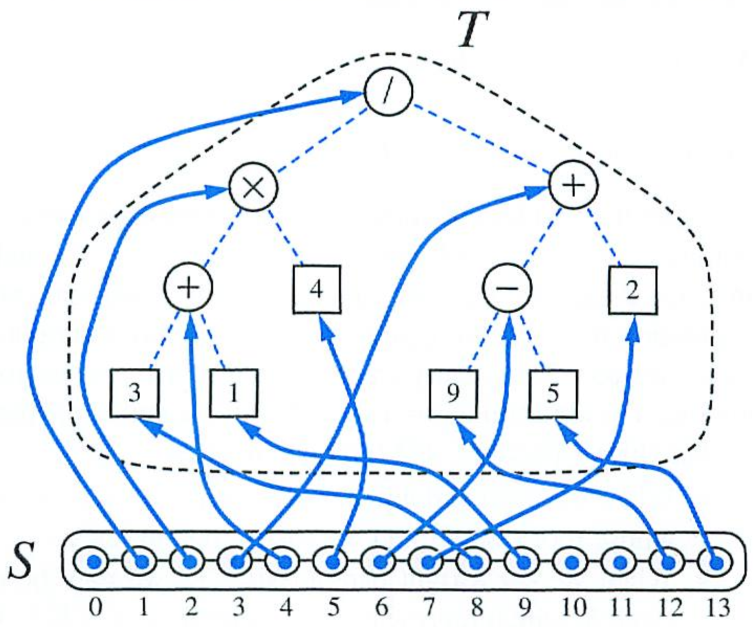

Figure 7.16: Representation of a binary tree T by means of an array list S.

Let n be the number of nodes of T, and let p_M be the maximum value of $p(v)$ over all the nodes of T. The array list S has size $N = p_M + 1$ since the element of S at index 0 is not associated with any node of T. Also, S will have, in general, a number of empty elements that do not refer to existing nodes of T. In fact, in the worst case, $N = 2^n$, the justification of which is left as an exercise (R-7.23). In Section 8.3, we will see a class of binary trees, called "heaps" for which $N = n + 1$. Thus, in spite of the worst-case space usage, there are applications for which the array-list representation of a binary tree is space efficient. Still, for general binary trees, the exponential worst-case space requirement of this representation is prohibitive.

Table 7.3 summarizes running times of the methods of a binary tree implemented with an array list. We do not include any tree update methods here.

Operation	Time
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent, children, left, right	$O(1)$
hasLeft, hasRight, isInternal, isExternal, isRoot	$O(1)$

Table 7.3: Running times for a binary tree T implemented with an array list S. We denote the number of nodes of T with n, and N denotes the size of S. The space usage is $O(N)$, which is $O(2^n)$ in the worst case.

7.3.6 Traversals of Binary Trees

As with general trees, binary tree computations often involve traversals.

Building an Expression Tree

Consider the problem of constructing an expression tree from a fully parenthesized arithmetic expression of size n. (Recall Example 7.9 and Code Fragment 7.24.) In Code Fragment 7.21, we give algorithm buildExpression for building such an expression tree, assuming all arithmetic operations are binary and variables are not parenthesized. Thus, every parenthesized subexpression contains an operator in the middle. The algorithm uses a stack S while scanning the input expression E looking for variables, operators, and right parentheses.

- When we see a variable or operator x, we create a single-node binary tree T, whose root stores x and we push T on the stack.
- When we see a right parenthesis, ")", we pop the top three trees from the stack S, which represent a subexpression $(E_1 \circ E_2)$. We then attach the trees for E_1 and E_2 to the one for $\circ$, and push the resulting tree back on S.

We repeat this until the expression E has been processed, at which time the top element on the stack is the expression tree for E. The total running time is $O(n)$.

```
Algorithm buildExpression($E$):
  Input: A fully-parenthesized arithmetic expression $E = e_0, e_1, \ldots, e_{n-1}$, with
      each $e_i$ being a variable, operator, or parenthetic symbol
  Output: A binary tree $T$ representing arithmetic expression $E$
   $S \leftarrow$ a new initially-empty stack
   for $i \leftarrow 0$ to $n-1$ do
     if $e_i$ is a variable or an operator then
       $T \leftarrow$ a new empty binary tree
       $T$.addRoot($e_i$)
       $S$.push($T$)
     else if $e_i$ ='(' then
       Continue looping
     else {$e_i$ =')'}
       $T_2 \leftarrow S$.pop()          {the tree representing $E_2$}
       $T \leftarrow S$.pop()           {the tree representing $\circ$}
       $T_1 \leftarrow S$.pop()          {the tree representing $E_1$}
       $T$.attach($T$.root(), $T_1$, $T_2$)
       $S$.push($T$)
   return $S$.pop()
```

Code Fragment 7.21: Algorithm buildExpression.

Preorder Traversal of a Binary Tree

Since any binary tree can also be viewed as a general tree, the preorder traversal for general trees (Code Fragment 7.8) can be applied to any binary tree. We can simplify the algorithm in the case of a binary tree traversal, however, as we show in Code Fragment 7.22.

```
Algorithm binaryPreorder($T, v$):
  perform the "visit" action for node $v$
  if $v$ has a left child $u$ in $T$ then
    binaryPreorder($T, u$)        {recursively traverse left subtree}
  if $v$ has a right child $w$ in $T$ then
    binaryPreorder($T, w$)        {recursively traverse right subtree}
```

Code Fragment 7.22: Algorithm binaryPreorder for performing the preorder traversal of the subtree of a binary tree T rooted at a node v.

As is the case for general trees, there are many applications of the preorder traversal for binary trees.

Postorder Traversal of a Binary Tree

Analogously, the postorder traversal for general trees (Code Fragment 7.11) can be specialized for binary trees, as shown in Code Fragment 7.23.

```
Algorithm binaryPostorder($T, v$):
  if $v$ has a left child $u$ in $T$ then
    binaryPostorder($T, u$)        {recursively traverse left subtree}
  if $v$ has a right child $w$ in $T$ then
    binaryPostorder($T, w$)        {recursively traverse right subtree}
  perform the "visit" action for node $v$
```

Code Fragment 7.23: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node v.

Expression Tree Evaluation

The postorder traversal of a binary tree can be used to solve the expression tree evaluation problem. In this problem, we are given an arithmetic expression tree, that is, a binary tree where each external node has a value associated with it and each internal node has an arithmetic operation associated with it (see Example 7.9), and we want to compute the value of the arithmetic expression represented by the tree.

Algorithm evaluateExpression, given in Code Fragment 7.24, evaluates the expression associated with the subtree rooted at a node v of an arithmetic expression tree T by performing a postorder traversal of T starting at v. In this case, the "visit" action consists of performing a single arithmetic operation. Note that we use the fact that an arithmetic expression tree is a proper binary tree.

```
Algorithm evaluateExpression($T, v$):
  if $v$ is an internal node in $T$ then
    let $\circ$ be the operator stored at $v$
    $x \leftarrow$ evaluateExpression($T, T$.left($v$))
    $y \leftarrow$ evaluateExpression($T, T$.right($v$))
    return $x \circ y$
  else
    return the value stored at $v$
```

Code Fragment 7.24: Algorithm evaluateExpression for evaluating the expression represented by the subtree of an arithmetic expression tree T rooted at node v.

The expression-tree evaluation application of the postorder traversal provides an $O(n)$-time algorithm for evaluating an arithmetic expression represented by a binary tree with n nodes. Indeed, like the general postorder traversal, the postorder traversal for binary trees can be applied to other "bottom-up" evaluation problems (such as the size computation given in Example 7.7) as well.

Inorder Traversal of a Binary Tree

An additional traversal method for a binary tree is the ***inorder*** traversal. In this traversal, we visit a node between the recursive traversals of its left and right subtrees. The inorder traversal of the subtree rooted at a node v in a binary tree T is given in Code Fragment 7.25.

```
Algorithm inorder($T, v$):
  if $v$ has a left child $u$ in $T$ then
    inorder($T, u$)          {recursively traverse left subtree}
  perform the "visit" action for node $v$
  if $v$ has a right child $w$ in $T$ then
    inorder($T, w$)          {recursively traverse right subtree}
```

Code Fragment 7.25: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node v.

The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right." Indeed, for every node v, the inorder traversal visits v after all the nodes in the left subtree of v and before all the nodes in the right subtree of v. (See Figure 7.17.)

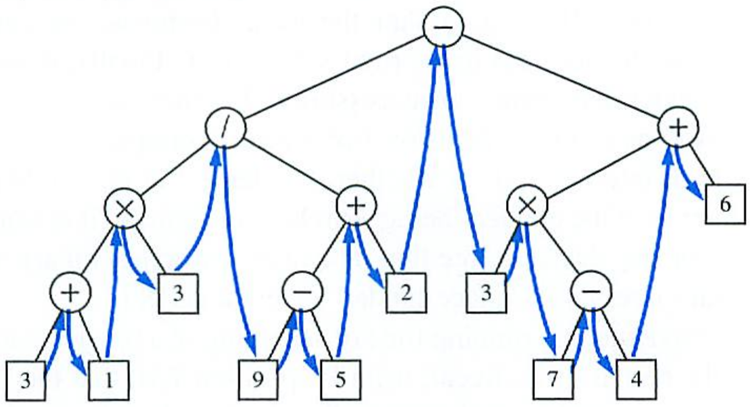

Figure 7.17: Inorder traversal of a binary tree.

Binary Search Trees

Let S be a set whose elements have an order relation. For example, S could be a set of integers. A ***binary search*** tree for S is a proper binary tree T such that

- Each internal node v of T stores an element of S, denoted with $x(v)$.
- For each internal node v of T, the elements stored in the left subtree of v are less than or equal to $x(v)$ and the elements stored in the right subtree of v are greater than or equal to $x(v)$.
- The external nodes of T do not store any element.

An inorder traversal of the internal nodes of a binary search tree T visits the elements in nondecreasing order. (See Figure 7.18.)

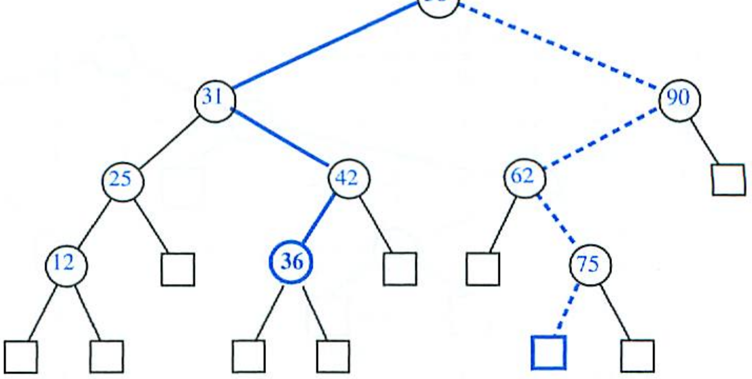

Figure 7.18: A binary search tree storing integers. The blue solid path is traversed when searching (successfully) for 36. The blue dashed path is traversed when searching (unsuccessfully) for 70.

We can use a binary search tree T for set S to find whether a given search value y is in S, by traversing a path down the tree T, starting at the root. (See Figure 7.18.) At each internal node v encountered, we compare our search value y with the element $x(v)$ stored at v. If $y \le x(v)$, then the search continues in the left subtree of v. If $y = x(v)$, then the search terminates successfully. If $y \ge x(v)$, then the search continues in the right subtree of v. Finally, if we reach an external node, the search terminates unsuccessfully. In other words, a binary search tree can be viewed as a binary decision tree (recall Example 7.8), where the question asked at each internal node is whether the element at that node is less than, equal to, or larger than the element being searched for. Indeed, it is exactly this correspondence to a binary decision tree that motivates restricting binary search trees to be proper binary trees (with "place-holder" external nodes).

Note that the running time of searching in a binary search tree T is proportional to the height of T. Recall from Proposition 7.10 that the height of a proper binary tree with n nodes can be as small as $\log(n+1)-1$ or as large as $(n-1)/2$. Thus, binary search trees are most efficient when they have small height. We illustrate an example search operation in a binary search tree in Figure 7.18, and we study binary search trees in more detail in Section 10.1.

Using Inorder Traversal for Tree Drawing

The inorder traversal can also be applied to the problem of computing a drawing of a binary tree. We can draw a binary tree T with an algorithm that assigns x- and y-coordinates to a node v of T using the following two rules (see Figure 7.19):

- $x(v)$ is the number of nodes visited before v in the inorder traversal of T
- $y(v)$ is the depth of v in T.

In this application, we take the convention common in computer graphics that x-coordinates increase left to right and y-coordinates increase top to bottom. So the origin is in the upper left corner of the computer screen.

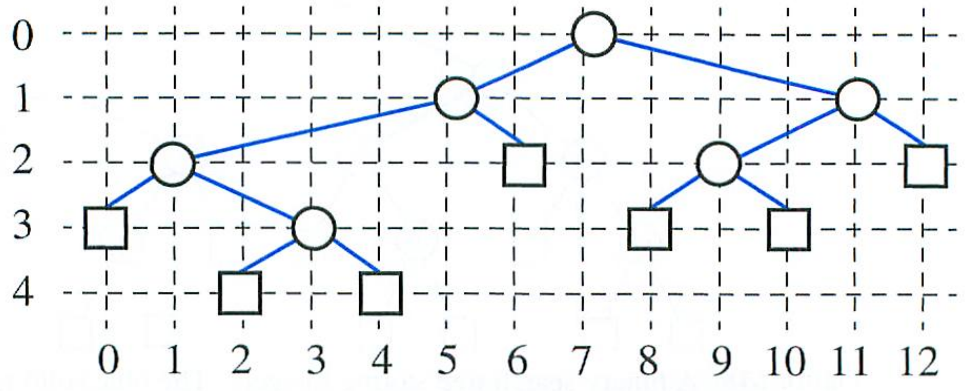

Figure 7.19: An inorder drawing of a binary tree.