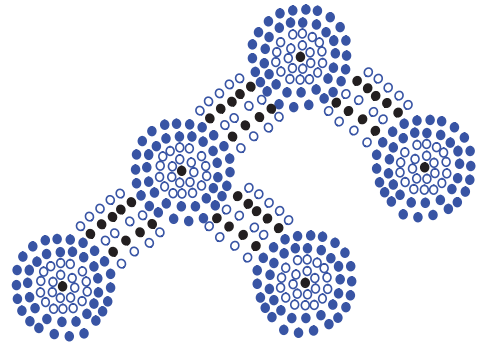
Chapter 7 Trees



Contents

7.1 General Trees

Productivity experts say that breakthroughs come by thinking "nonlinearly." In this chapter, we discuss one of the most important nonlinear data structures in computing—***trees***. Tree structures are indeed a breakthrough in data organization, for they allow us to implement a host of algorithms much faster than when using linear data structures, such as lists, vectors, and sequences. Trees also provide a natural organization for data, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems.

It is not always clear what productivity experts mean by "nonlinear" thinking, but when we say that trees are "nonlinear," we are referring to an organizational relationship that is richer than the simple "before" and "after" relationships between objects in sequences. The relationships in a tree are ***hierarchical***, with some objects being "above" and some "below" others. Actually, the main terminology for tree data structures comes from family trees, with the terms "parent," "child," "ancestor," and "descendant" being the most common words used to describe relationships. We show an example of a family tree in Figure 7.1.

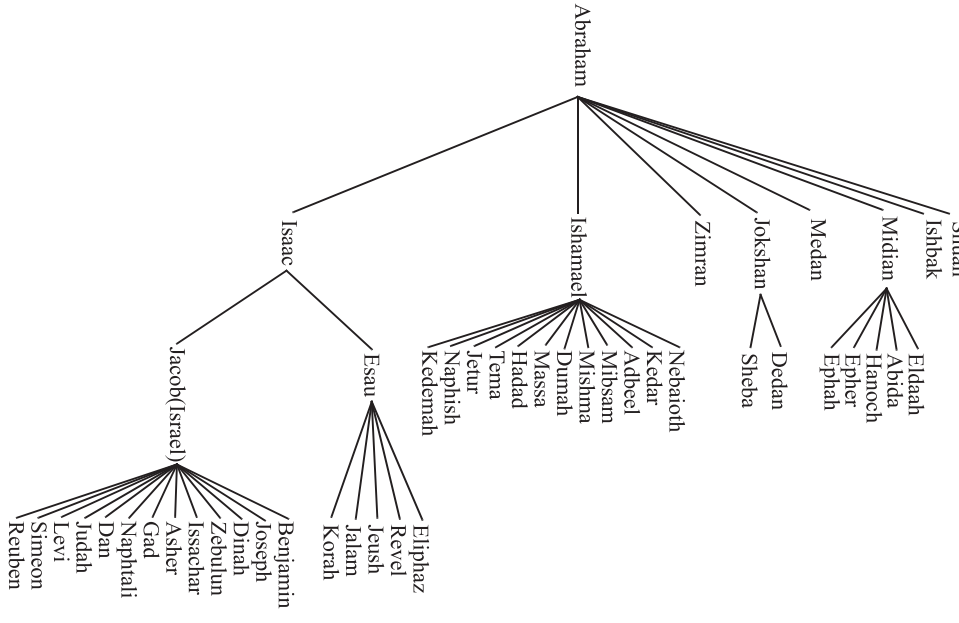


Figure 7.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

7.1.1 Tree Definitions and Properties

A ***tree*** is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a ***parent*** element and zero or more ***children*** elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. (See Figure 7.2.) We typically call the top element the ***root*** of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

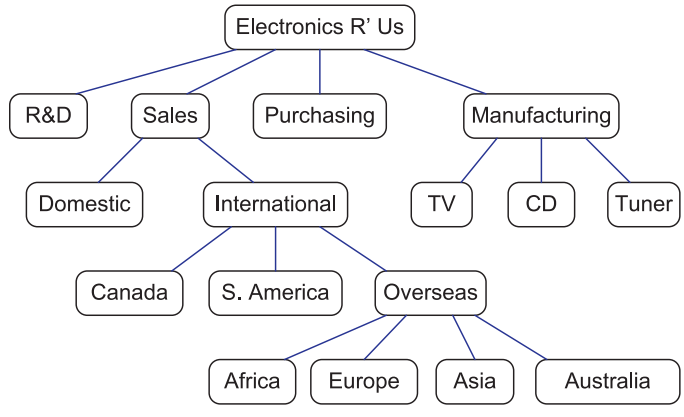


Figure 7.2: A tree with 17 nodes representing the organizational structure of a fictitious corporation. *Electronics R'Us* is stored at the root. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

Formal Tree Definition

Formally, we define ***tree*** T to be a set of ***nodes*** storing elements in a ***parent-child*** relationship with the following properties:

- If T is nonempty, it has a special node, called the ***root*** of T, that has no parent.
- Each node v of T different from the root has a unique ***parent*** node w; every node with parent w is a ***child*** of w.

Note that according to our definition, a tree can be empty, meaning that it doesn't have any nodes. This convention also allows us to define a tree recursively, such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of trees whose roots are the children of r.

Other Node Relationships

Two nodes that are children of the same parent are ***siblings***. A node v is ***external*** if v has no children. A node v is ***internal*** if it has one or more children. External nodes are also known as ***leaves***.

Example 7.1: *In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. (See Figure 7.3.) More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX and Linux operating systems, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/."*

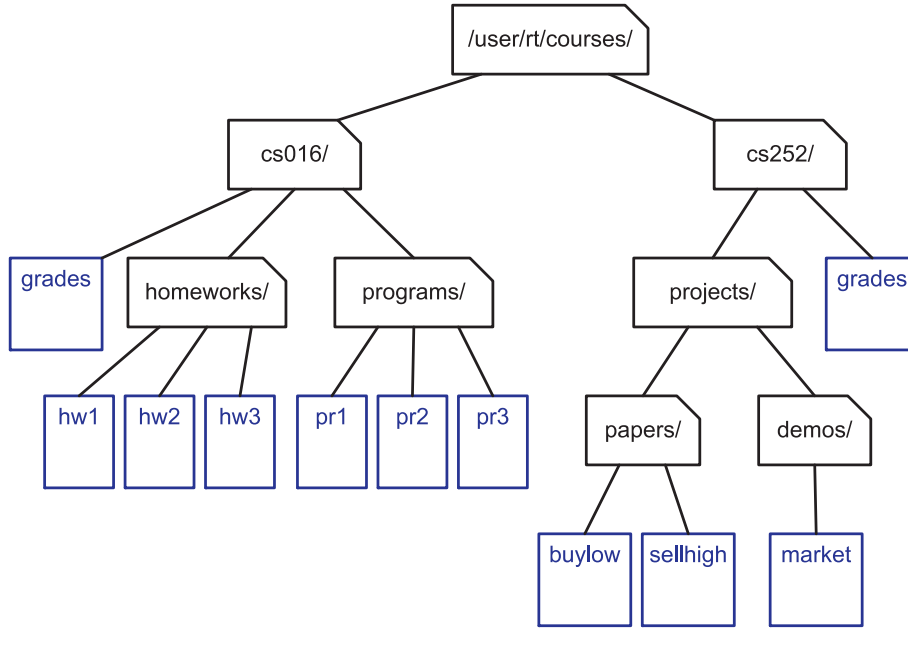


Figure 7.3: Tree representing a portion of a file system.

A node u is an ***ancestor*** of a node v if $u = v$ or u is an ancestor of the parent of v. Conversely, we say that a node v is a ***descendent*** of a node u if u is an ancestor of v. For example, in Figure 7.3, cs252/ is an ancestor of papers/, and pr3 is a descendent of cs016/. The ***subtree*** of T ***rooted*** at a node v is the tree consisting of all the descendents of v in T (including v itself). In Figure 7.3, the subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.

Edges and Paths in Trees

An ***edge*** of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa. A ***path*** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. For example, the tree in Figure 7.3 contains the path (cs252/, projects/, demos/, market).

Example 7.2: *When using single inheritance, the inheritance relation between classes in a C++ program forms a tree. The base class is the root of the tree.*

Ordered Trees

A tree is ***ordered*** if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, second, third, and so on. Such an ordering is determined by how the tree is to be used, and is usually indicated by drawing the tree with siblings arranged from left to right, corresponding to their linear relationship. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in a sequence or iterator in the correct order.

Example 7.3: *A structured document, such as a book, is hierarchically organized as a tree whose internal nodes are chapters, sections, and subsections, and whose external nodes are paragraphs, tables, figures, the bibliography, and so on. (See Figure 7.4.) The root of the tree corresponds to the book itself. We could, in fact, consider expanding the tree further to show paragraphs consisting of sentences, sentences consisting of words, and words consisting of characters. In any case, such a tree is an example of an ordered tree, because there is a well-defined ordering among the children of each node.*

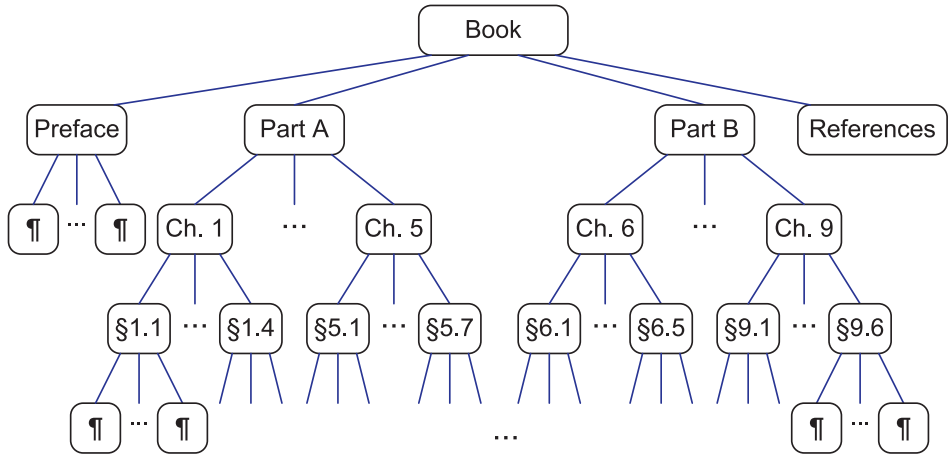


Figure 7.4: An ordered tree associated with a book.

7.1.2 Tree Functions

The tree ADT stores elements at the nodes of the tree. Because nodes are internal aspects of our implementation, we do not allow access to them directly. Instead, each node of the tree is associated with a ***position*** object, which provides public access to nodes. For this reason, when discussing the public interfaces of functions of our ADT, we use the notation p (rather than v) to clarify that the argument to the function is a position and not a node. But, given the tight connection between these two objects, we often blur the distinction between them, and use the terms "position" and "node" interchangeably for trees.

As we did with positions for lists in Chapter 6, we exploit C++'s ability to overload the dereferencing operator ("$*$") to access the element associated with a position. Given a position variable p, the associated element is accessed by $*p$. This can be used both for reading and modifying the element's value.

It is useful to store collections of positions. For example, the children of a node in a tree can be presented to the user as such a list. We define ***position list***, to be a list whose elements are tree positions.

The real power of a tree position arises from its ability to access the neighboring elements of the tree. Given a position p of tree T, we define the following:

p.parent(): Return the parent of p; an error occurs if p is the root.

p.children(): Return a position list containing the children of node p.

p.isRoot(): Return true if p is the root and false otherwise.

p.isExternal(): Return true if p is external and false otherwise.

If a tree T is ordered, then the list provided by p.children() provides access to the children of p in order. If p is an external node, then p.children() returns an empty list. If we wanted, we could also provide a function p.isInternal(), which would simply return the complement of p.isExternal().

The tree itself provides the following functions. The first two, size and empty, are just the standard functions that we defined for the other container types we already saw. The function root yields the position of the root and positions produces a list containing all the tree's nodes.

size(): Return the number of nodes in the tree.

empty(): Return true if the tree is empty and false otherwise.

root(): Return a position for the tree's root; an error occurs if the tree is empty.

positions(): Return a position list of all the nodes of the tree.

We have not defined any specialized update functions for a tree here. Instead, we prefer to describe different tree update functions in conjunction with specific applications of trees in subsequent chapters. In fact, we can imagine several kinds of tree update operations beyond those given in this book.

7.1.3 A C++ Tree Interface

Let us present an informal C++ interface for the tree ADT. We begin by presenting an informal C++ interface for the class Position, which represents a position in a tree. This is given in Code Fragment 7.1.

```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position parent() const;               // get parent
  PositionList children() const;         // get node's children
  bool isRoot() const;                   // root node?
  bool isExternal() const;               // external node?
};
```

Code Fragment 7.1: An informal interface for a position in a tree (not a complete C++ class).

We have provided a version of the derferencing operator ("*") that returns a standard (readable and writable) reference. (For simplicity, we did not provide a version that returns a constant reference, but this would be an easy addition.)

Next, in Code Fragment 7.2, we present our informal C++ interface for a tree. To keep the interface as simple as possible, we ignore error processing; hence, we do not declare any exceptions to be thrown.

```
template <typename E>                    // base element type
class Tree<E> {
public:                                  // public types
  class Position;                        // a node position
  class PositionList;                    // a list of positions
public:                                  // public functions
  int size() const;                      // number of nodes
  bool empty() const;                    // is tree empty?
  Position root() const;                 // get the root
  PositionList positions() const;        // get positions of all nodes
};
```

Code Fragment 7.2: An informal interface for the tree ADT (not a complete class).

Although we have not formally defined an interface for the class PositionList, we may assume that it satisfies the standard list ADT as given in Chapter 6. In our code examples, we assume that PositionList is implemented as an STL list of objects of type Position, or more concretely, "std::list<Position>." In particular, we assume that PositionList provides an iterator type, which we simply call Iterator in our later examples.

7.1.4 A Linked Structure for General Trees

A natural way to realize a tree T is to use a ***linked structure***, where we represent each node of T by a position object p (see Figure 7.5(a)) with the following fields: a reference to the node's element, a link to the node's parent, and some kind of collection (for example, a list or array) to store links to the node's children. If p is the root of T, then the *parent* field of p is NULL. Also, we store a reference to the root of T and the number of nodes of T in internal variables. This structure is schematically illustrated in Figure 7.5(b).

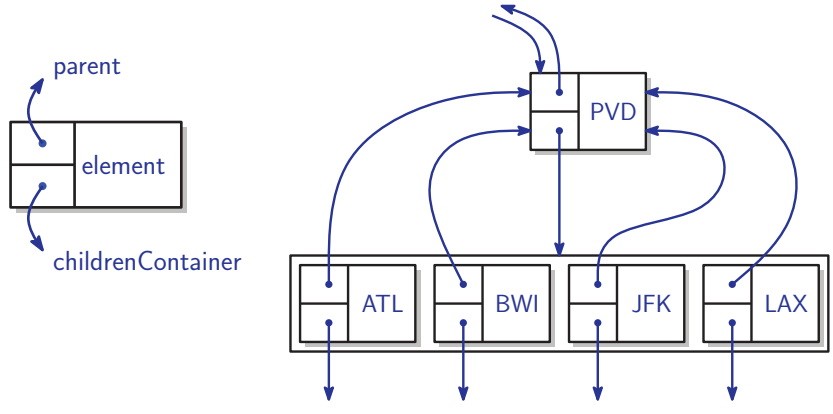


Figure 7.5: The linked structure for a general tree: (a) the node structure; (b) the portion of the data structure associated with a node and its children.

Table 7.1 summarizes the performance of the linked-structure implementation of a tree. The analysis is left as an exercise (C-7.27), but we note that, by using a container to store the children of each node p, we can implement the children(p) function by using the iterator for the container to enumerate its elements.

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Table 7.1: Running times of the functions of an n-node linked tree structure. Let c_p denote the number of children of a node p. The space usage is $O(n)$.

7.2 Tree Traversal Algorithms

In this section, we present algorithms for performing traversal computations on a tree by accessing it through the tree ADT functions.

7.2.1 Depth and Height

Let p be a node of a tree T. The ***depth*** of p is the number of ancestors of p, excluding p itself. For example, in the tree of Figure 7.2, the node storing *International* has depth 2. Note that this definition implies that the depth of the root of T is 0. The depth of p's node can also be recursively defined as follows:

- If p is the root, then the depth of p is 0
- Otherwise, the depth of p is one plus the depth of the parent of p

Based on the above definition, the recursive algorithm $\mathsf{depth}(T, p)$ shown in Code Fragment 7.3, computes the depth of a node referenced by position p of T by calling itself recursively on the parent of p, and adding 1 to the value returned.

Algorithm $\mathsf{depth}(T, p)$:
  if p.isRoot() **then**
    return 0
  else
    return $1 + \mathsf{depth}(T, p.\mathsf{parent}())$

Code Fragment 7.3: An algorithm to compute the depth of a node p in a tree T.

A simple C++ implementation of algorithm depth is shown in Code Fragment 7.4.

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                              // root has depth 0
  else
    return 1 + depth(T, p.parent());       // 1 + (depth of parent)
}
```

Code Fragment 7.4: A C++ implementation of the algorithm of Code Fragment 7.3.

The running time of algorithm $\mathsf{depth}(T, p)$ is $O(d_p)$, where d_p denotes the depth of the node p in the tree T, because the algorithm performs a constant-time recursive step for each ancestor of p. Thus, in the worst case, the depth algorithm runs in $O(n)$ time, where n is the total number of nodes in the tree T, since some nodes may have this depth in T. Although such a running time is a function of the input size, it is more accurate to characterize the running time in terms of the parameter d_p, since it is often much smaller than n.

The ***height*** of a node p in a tree T is also defined recursively.

- If p is external, then the height of p is 0
- Otherwise, the height of p is one plus the maximum height of a child of p

The ***height*** of a tree T is the height of the root of T. For example, the tree of Figure 7.2 has height 4. In addition, height can also be viewed as follows.

Proposition 7.4: *The height of a tree is equal to the maximum depth of its external nodes.*

We leave the justification of this fact to an exercise (R-7.7). Based on this proposition, we present an algorithm, height1, for computing the height of a tree T. It is shown in Code Fragment 7.5. It enumerates all the nodes in the tree and invokes function depth (Code Fragment 7.3) to compute the depth of each external node.

Algorithm height1(T):
  $h = 0$
  for each $p \in T$.positions() **do**
    if p.isExternal() **then**
      $h = \max(h, \text{depth}(T, p))$
  return h

Code Fragment 7.5: Algorithm height1(T) for computing the height of a tree T based on computing the maximum depth of the external nodes.

The C++ implementation of this algorithm is shown in Code Fragment 7.6. We assume that Iterator is the iterator class for PositionList. Given such an iterator q, we can access the associated position as $*q$.

```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions();          // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
    if (q->isExternal())
      h = max(h, depth(T, *q));                // get max depth among leaves
  }
  return h;
}
```

Code Fragment 7.6: A C++ implementation of the function height1.

Unfortunately, algorithm height1 is not very efficient. Since height1 calls algorithm depth(p) on each external node p of T, the running time of height1 is given by $O(n + \sum_p (1 + d_p))$, where n is the number of nodes of T, d_p is the depth of node p, and E is the set of external nodes of T. In the worst case, the sum $\sum_p (1 + d_p)$ is proportional to n^2. (See Exercise C-7.8.) Thus, algorithm height1 runs in $O(n^2)$ time.

Algorithm height2, shown in Code Fragment 7.7 and implemented in C++ in Code Fragment 7.8, computes the height of tree T in a more efficient manner by using the recursive definition of height.

Algorithm height2(T, p):
  if p.isExternal() **then**
    return 0
  else
    $h = 0$
    for each $q \in p$.children() **do**
      $h = \max(h, \text{height2}(T, q))$
    return $1 + h$

Code Fragment 7.7: A more efficient algorithm for computing the height of the subtree of tree T rooted at a node p.

```
int height2(const Tree& T, const Position& p) {
  if (p.isExternal()) return 0;                  // leaf has height 0
  int h = 0;
  PositionList ch = p.children();                // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q)
    h = max(h, height2(T, *q));
  return 1 + h;                                  // 1 + max height of children
}
```

Code Fragment 7.8: Method height2 written in C++.

Algorithm height2 is more efficient than height1 (from Code Fragment 7.5). The algorithm is recursive, and, if it is initially called on the root of T, it will eventually be called on each node of T. Thus, we can determine the running time of this method by summing, over all the nodes, the amount of time spent at each node (on the nonrecursive part). Processing each node in children(p) takes $O(c_p)$ time, where c_p denotes the number of children of node p. Also, the **while** loop has c_p iterations and each iteration of the loop takes $O(1)$ time plus the time for the recursive call on a child of p. Thus, algorithm height2 spends $O(1 + c_p)$ time at each node p, and its running time is $O(\sum_p (1 + c_p))$. In order to complete the analysis, we make use of the following property.

Proposition 7.5: *Let T be a tree with n nodes, and let c_p denote the number of children of a node p of T. Then $\sum_p c_p = n - 1$.*

Justification: Each node of T, with the exception of the root, is a child of another node, and thus contributes one unit to the above sum. ■

By Proposition 7.5, the running time of algorithm height2, when called on the root of T, is $O(n)$, where n is the number of nodes of T.

7.2.2 Preorder Traversal

A ***traversal*** of a tree T is a systematic way of accessing, or "visiting," all the nodes of T. In this section, we present a basic traversal scheme for trees, called preorder traversal. In the next section, we study another basic traversal scheme, called postorder traversal.

In a ***preorder*** traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children. The specific action associated with the "visit" of a node depends on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for this node. The pseudo-code for the preorder traversal of the subtree rooted at a node referenced by position p is shown in Code Fragment 7.9. We initially invoke this routine with the call preorder$(T, T.\text{root}())$.

Algorithm preorder(T, p):
  perform the "visit" action for node p
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling preorder(T, q)

Code Fragment 7.9: Algorithm preorder for performing the preorder traversal of the subtree of a tree T rooted at a node p.

The preorder traversal algorithm is useful for producing a linear ordering of the nodes of a tree where parents must always come before their children in the ordering. Such orderings have several different applications. We explore a simple instance of such an application in the next example.

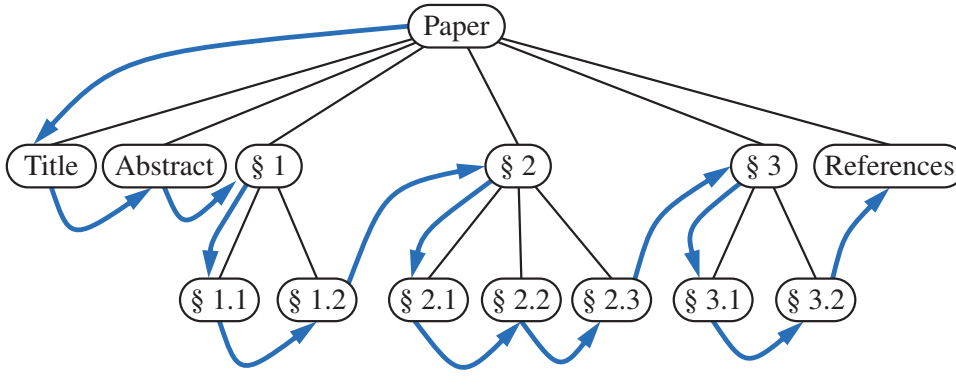


Figure 7.6: Preorder traversal of an ordered tree, where the children of each node are ordered from left to right.

Example 7.6: *The preorder traversal of the tree associated with a document, as in Example 7.3, examines an entire document sequentially, from beginning to end. If the external nodes are removed before the traversal, then the traversal examines the table of contents of the document. (See Figure 7.6.)*

The preorder traversal is also an efficient way to access all the nodes of a tree. To justify this, let us consider the running time of the preorder traversal of a tree T with n nodes under the assumption that visiting a node takes $O(1)$ time. The analysis of the preorder traversal algorithm is actually similar to that of algorithm height2 (Code Fragment 7.8), given in Section 7.2.1. At each node p, the nonrecursive part of the preorder traversal algorithm requires time $O(1+c_p)$, where c_p is the number of children of p. Thus, by Proposition 7.5, the overall running time of the preorder traversal of T is $O(n)$.

Algorithm preorderPrint(T,p), implemented in C++ in Code Fragment 7.10, performs a preorder printing of the subtree of a node p of T, that is, it performs the preorder traversal of the subtree rooted at p and prints the element stored at a node when the node is visited. Recall that, for an ordered tree T, function T.children(p) returns an iterator that accesses the children of p in order. We assume that Iterator is this iterator type. Given an iterator q, the associated position is given by $*q$.

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                                  // print element
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

Code Fragment 7.10: Method preorderPrint(T,p) that performs a preorder printing of the elements in the subtree associated with position p of T.

There is an interesting variation of the preorderPrint function that outputs a different representation of an entire tree. The ***parenthetic string representation*** $P(T)$ of tree T is recursively defined as follows. If T consists of a single node referenced by a position p, then

$$P(T) = *p.$$

Otherwise,

$$P(T) = *p + \text{"("} + P(T_1) + P(T_2) + \cdots + P(T_k) + \text{")"},$$

where p is the root position of T and $T_1, T_2, \ldots, T_k$ are the subtrees rooted at the children of p, which are given in order if T is an ordered tree.

Note that the definition of $P(T)$ is recursive. Also, we are using "+" here to denote string concatenation. (Recall the string type from Section 1.1.3.) The parenthetic representation of the tree of Figure 7.2 is shown in Figure 7.7.

Electronics R'Us (
  R&D
  Sales (
    Domestic
    International (
      Canada
      S. America
      Overseas (*Africa* *Europe* *Asia* *Australia*)))
  Purchasing
  Manufacturing (*TV* *CD* *Tuner*))

Figure 7.7: Parenthetic representation of the tree of Figure 7.2. Indentation, line breaks, and spaces have been added for clarity.

Note that, technically speaking, there are some computations that occur between and after the recursive calls at a node's children in the above algorithm. We still consider this algorithm to be a preorder traversal, however, since the primary action of printing a node's contents occurs prior to the recursive calls.

The C++ function parenPrint, shown in Code Fragment 7.11, is a variation of function preorderPrint (Code Fragment 7.10). It implements the definition given above to output a parenthetic string representation of a tree T. It first prints the element associated with each node. For each internal node, we first print "(", followed by the parenthetical representation of each of its children, followed by ")".

```
void parenPrint(const Tree& T, const Position& p) {
  cout << *p;                                  // print node's element
  if (!p.isExternal()) {
    PositionList ch = p.children();            // list of children
    cout << "( ";                              // open
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
      if (q != ch.begin()) cout << " ";        // print separator
      parenPrint(T, *q);                       // visit the next child
    }
    cout << " )";                              // close
  }
}
```

Code Fragment 7.11: A C++ implementation of algorithm parenPrint.

We explore a modification of Code Fragment 7.11 in Exercise R-7.10, to display a tree in a fashion more closely matching that given in Figure 7.7.

7.2.3 Postorder Traversal

Another important tree traversal algorithm is the ***postorder traversal***. This algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root. It is similar to the preorder traversal, however, in that we use it to solve a particular problem by specializing an action associated with the "visit" of a node p. Still, as with the preorder traversal, if the tree is ordered, we make recursive calls for the children of a node p according to their specified order. Pseudo-code for the postorder traversal is given in Code Fragment 7.12.

Algorithm postorder(T, p):
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling postorder(T, q)
  perform the "visit" action for node p

Code Fragment 7.12: Algorithm postorder for performing the postorder traversal of the subtree of a tree T rooted at a node p.

The name of the postorder traversal comes from the fact that this traversal method visits a node p after it has visited all the other nodes in the subtree rooted at p. (See Figure 7.8.)

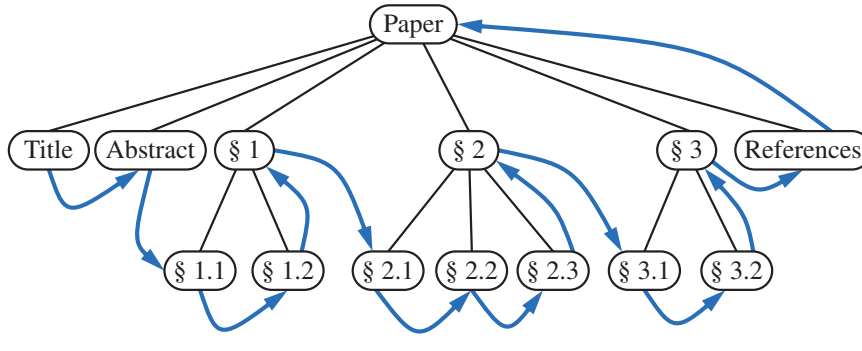


Figure 7.8: Postorder traversal of the ordered tree of Figure 7.6.

The analysis of the running time of a postorder traversal is analogous to that of a preorder traversal. (See Section 7.2.2.) The total time spent in the nonrecursive portions of the algorithm is proportional to the time spent visiting the children of each node in the tree. Thus, a postorder traversal of a tree T with n nodes takes $O(n)$ time, assuming that visiting each node takes $O(1)$ time. That is, the postorder traversal runs in linear time.

In Code Fragment 7.13, we present a C++ function postorderPrint which per-

forms a postorder traversal of a tree T. This function prints the element stored at a node when it is visited.

```
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                  // print element
}
```

Code Fragment 7.13: The function postorderPrint(T, p), which prints the elements of the subtree of position p of T.

The postorder traversal method is useful for solving problems where we wish to compute some property for each node p in a tree, but computing that property for p requires that we have already computed that same property for p's children. Such an application is illustrated in the following example.

Example 7.7: *Consider a file-system tree T, where external nodes represent files and internal nodes represent directories (Example 7.1). Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of the following (see Figure 7.9):*

- *The size of the directory itself*
- *The sizes of the files in the directory*
- *The space used by the children directories*

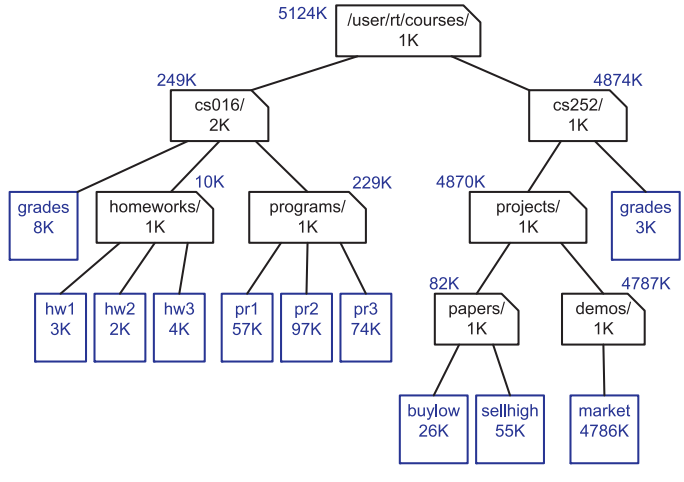


Figure 7.9: The tree of Figure 7.3 representing a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.

This computation can be done with a postorder traversal of tree T. After the subtrees of an internal node p have been traversed, we compute the space used by p by adding the sizes of the directory p itself and of the files contained in p, to the space used by each internal child of p, which was computed by the recursive postorder traversals of the children of p.

Motivated by Example 7.7, algorithm diskSpace, which is presented in Code Fragment 7.14, performs a postorder traversal of a file-system tree T, printing the name and disk space used by the directory associated with each internal node of T. When called on the root of tree T, diskSpace runs in time $O(n)$, where n is the number of nodes of the tree, provided the auxiliary functions name(p) and size(p) take $O(1)$ time.

```
int diskSpace(const Tree& T, const Position& p) {
  int s = size(p);                          // start with size of p
  if (!p.isExternal()) {                    // if p is internal
    PositionList ch = p.children();         // list of p's children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      s += diskSpace(T, *q);                // sum the space of subtrees
    cout << name(p) << ": " << s << endl; // print summary
  }
  return s;
}
```

Code Fragment 7.14: The function diskSpace, which prints the name and disk space used by the directory associated with p, for each internal node p of a file-system tree T. This function calls the auxiliary functions name and size, which should be defined to return the name and size of the file/directory associated with a node.

Other Kinds of Traversals

Preorder traversal is useful when we want to perform an action for a node and then recursively perform that action for its children, and postorder traversal is useful when we want to first perform an action on the descendents of a node and then perform that action on the node.

Although the preorder and postorder traversals are common ways of visiting the nodes of a tree, we can also imagine other traversals. For example, we could traverse a tree so that we visit all the nodes at depth d before we visit the nodes at depth $d+1$. Such a traversal, called a ***breadth-first traversal***, could be implemented using a queue, whereas the preorder and postorder traversals use a stack. (This stack is implicit in our use of recursion to describe these functions, but we could make this use explicit, as well, to avoid recursion.) In addition, binary trees, which we discuss next, support an additional traversal method known as the inorder traversal.

7.3 Binary Trees

A ***binary tree*** is an ordered tree in which every node has at most two children.

1. Every node has at most two children.
2. Each child node is labeled as being either a ***left child*** or a ***right child***.
3. A left child precedes a right child in the ordering of children of a node.

The subtree rooted at a left or right child of an internal node is called the node's ***left subtree*** or ***right subtree***, respectively. A binary tree is ***proper*** if each node has either zero or two children. Some people also refer to such trees as being ***full*** binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is ***improper***.

Example 7.8: *An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes-or-no questions. Each internal node is associated with a question. Starting at the root, we go to the left or right child of the current node, depending on whether the answer to the question is "Yes" or "No." With each decision, we follow an edge from a parent to a child, eventually tracing a path in the tree from the root to an external node. Such binary trees are known as* **decision trees**, *because each external node p in such a tree represents a decision of what to do if the questions associated with p's ancestors are answered in a way that leads to p. A decision tree is a proper binary tree. Figure 7.10 illustrates a decision tree that provides recommendations to a prospective investor.*

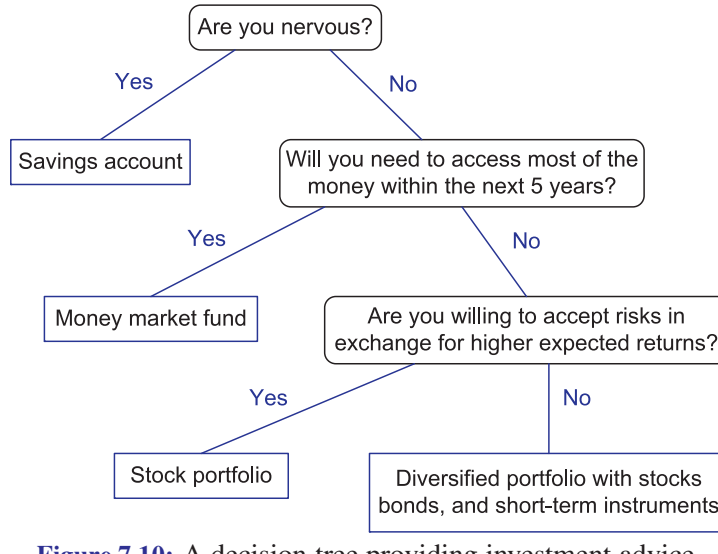


Figure 7.10: A decision tree providing investment advice.

Example 7.9: *An arithmetic expression can be represented by a tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators $+$, $-$, $\times$, and $/$. (See Figure 7.11.) Each node in such a tree has a value associated with it.*

- *If a node is external, then its value is that of its variable or constant.*
- *If a node is internal, then its value is defined by applying its operation to the values of its children.*

Such an arithmetic-expression tree is a proper binary tree, since each of the operators $+$, $-$, $\times$, and $/$ take exactly two operands. Of course, if we were to allow for unary operators, like negation ($-$), as in "$-x$," then we could have an improper binary tree.

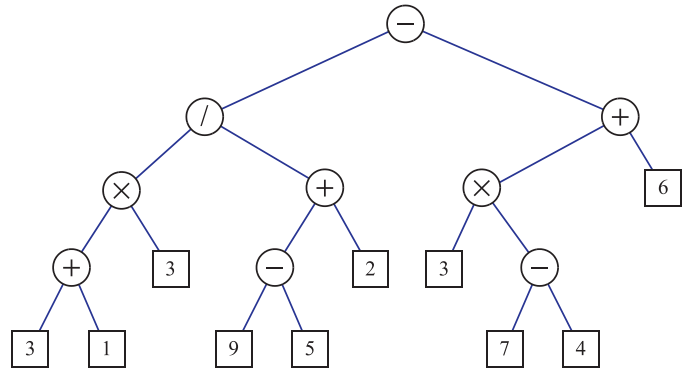


Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

A Recursive Binary Tree Definition

Incidentally, we can also define a binary tree in a recursive way such that a binary tree is either empty or consists of:

- A node r, called the ***root*** of T and storing an element
- A binary tree, called the ***left subtree*** of T
- A binary tree, called the ***right subtree*** of T

We discuss some of the specialized topics for binary trees below.

7.3.1 The Binary Tree ADT

In this section, we introduce an abstract data type for a binary tree. As with our earlier tree ADT, each node of the tree stores an element and is associated with a

position object, which provides public access to nodes. By overloading the dereferencing operator, the element associated with a position p can be accessed by $*p$. In addition, a position p supports the following operations.

p.left(): Return the left child of p; an error condition occurs if p is an external node.

p.right(): Return the right child of p; an error condition occurs if p is an external node.

p.parent(): Return the parent of p; an error occurs if p is the root.

p.isRoot(): Return true if p is the root and false otherwise.

p.isExternal(): Return true if p is external and false otherwise.

The tree itself provides the same operations as the standard tree ADT. Recall that a position list is a list of tree positions.

size(): Return the number of nodes in the tree.

empty(): Return true if the tree is empty and false otherwise.

root(): Return a position for the tree's root; an error occurs if the tree is empty.

positions(): Return a position list of all the nodes of the tree.

As in Section 7.1.2 for the tree ADT, we do not define specialized update functions for binary trees, but we consider them later.

7.3.2 A C++ Binary Tree Interface

Let us present an informal C++ interface for the binary tree ADT. We begin in Code Fragment 7.15 by presenting an informal C++ interface for the class Position, which represents a position in a tree. It differs from the tree interface of Section 7.1.3 by replacing the tree member function children with the two functions left and right.

```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position left() const;                 // get left child
  Position right() const;                // get right child
  Position parent() const;               // get parent
  bool isRoot() const;                   // root of tree?
  bool isExternal() const;               // an external node?
};
```

Code Fragment 7.15: An informal interface for the binary tree ADT (not a complete C++ class).

Next, in Code Fragment 7.16, we present an informal C++ interface for a binary tree. To keep the interface as simple as possible, we have ignored error processing, and hence we do not declare any exceptions to be thrown.

```
template <typename E>                          // base element type
class BinaryTree<E> {                          // binary tree
public:                                        // public types
  class Position;                              // a node position
  class PositionList;                          // a list of positions
public:                                        // member functions
  int size() const;                            // number of nodes
  bool empty() const;                          // is tree empty?
  Position root() const;                       // get the root
  PositionList positions() const;              // list of nodes
};
```

Code Fragment 7.16: An informal interface for the binary tree ADT (not a complete C++ class).

Although we have not formally defined an interface for the class PositionList, we may assume that it satisfies the standard list ADT as given in Chapter 6. In our code examples, we assume that PositionList is implemented as an STL list of objects of type Position.

7.3.3 Properties of Binary Trees

Binary trees have several interesting properties dealing with relationships between their heights and number of nodes. We denote the set of all nodes of a tree T, at the same depth d, as the ***level*** d of T. In a binary tree, level 0 has one node (the root), level 1 has, at most, two nodes (the children of the root), level 2 has, at most, four nodes, and so on. (See Figure 7.12.) In general, level d has, at most, 2^d nodes.

We can see that the maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree. From this simple observation, we can derive the following properties relating the height of a binary T to its number of nodes. A detailed justification of these properties is left as an exercise (R-7.16).

Proposition 7.10: *Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:*

1. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq n-1$

Also, if T is proper, then it has the following properties:

1. $2h+1 \leq n \leq 2^{h+1}-1$
2. $h+1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq (n-1)/2$

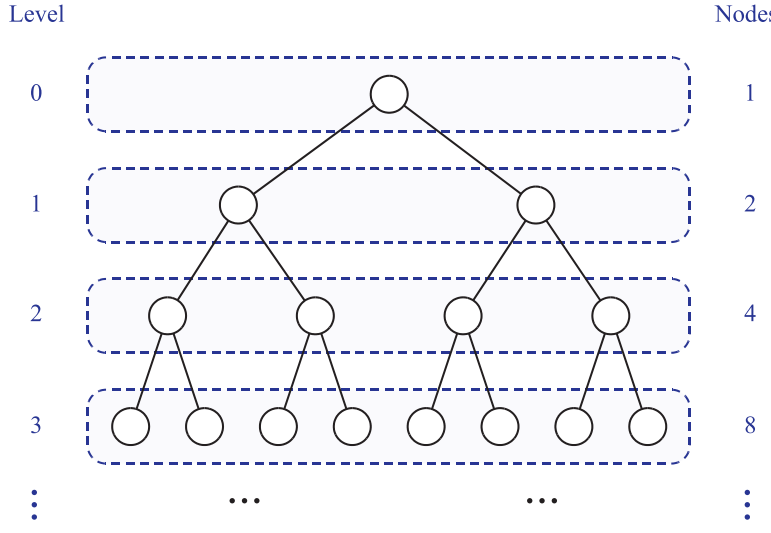


Figure 7.12: Maximum number of nodes in the levels of a binary tree.

In addition to the binary tree properties above, we also have the following relationship between the number of internal nodes and external nodes in a proper binary tree.

Proposition 7.11: *In a nonempty proper binary tree T, the number of external nodes is one more than the number of internal nodes.*

Justification: We can see this using an argument based on induction. If the tree consists of a single root node, then clearly we have one external node and no internal nodes, so the proposition holds.

If, on the other hand, we have two or more, then the root has two subtrees. Since the subtrees are smaller than the original tree, we may assume that they satisfy the proposition. Thus, each subtree has one more external node than internal nodes. Between the two of them, there are two more external nodes than internal nodes. But, the root of the tree is an internal node. When we consider the root and both subtrees together, the difference between the number of external and internal nodes is $2-1=1$, which is just what we want. ■

Note that the above relationship does not hold, in general, for improper binary trees and nonbinary trees, although there are other interesting relationships that can hold as we explore in an exercise (C-7.9).

7.3.4 A Linked Structure for Binary Trees

In this section, we present an implementation of a binary tree T as a ***linked structure***, called LinkedBinaryTree. We represent each node v of T by a node object storing the associated element and pointers to its parent and two children. (See Figure 7.13.) For simplicity, we assume the tree is ***proper***, meaning that each node has either zero or two children.

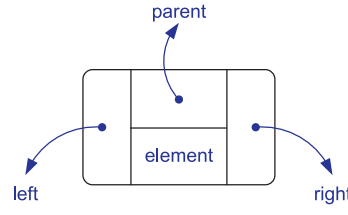


Figure 7.13: A node in a linked data structure for representing a binary tree.

In Figure 7.14, we show a linked structure representation of a binary tree. The structure stores the tree's size, that is, the number of nodes in the tree, and a pointer to the root of the tree. The rest of the structure consists of the nodes linked together appropriately. If v is the root of T, then the pointer to the parent node is NULL, and if v is an external node, then the pointers to the children of v are NULL.

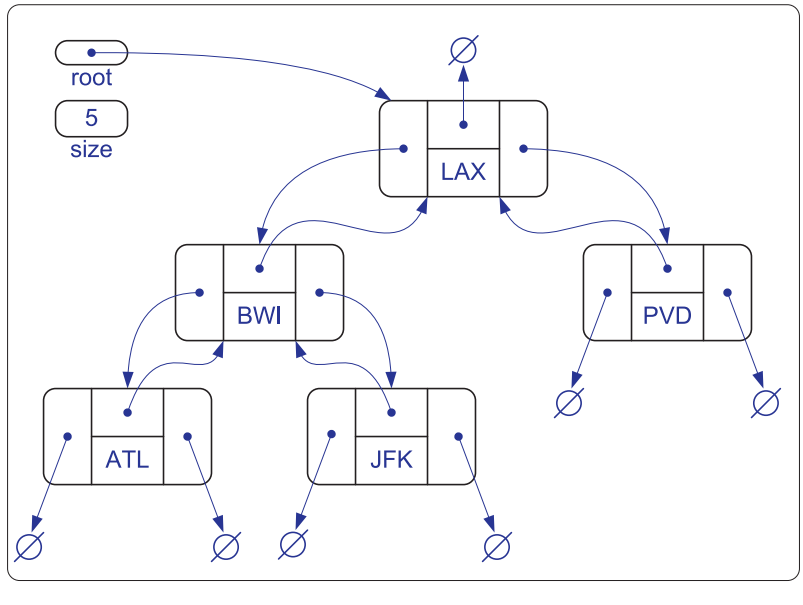


Figure 7.14: An example of a linked data structure for representing a binary tree.

We begin by defining the basic constituents that make up the LinkedBinaryTree class. The most basic entity is the structure Node, shown in Code Fragment 7.17, that represents a node of the tree.

```
struct Node {                                          // a node of the tree
  Elem   elt;                                          // element value
  Node*  par;                                          // parent
  Node*  left;                                         // left child
  Node*  right;                                        // right child
  Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
};
```

Code Fragment 7.17: Structure Node implementing a node of a binary tree. It is nested in the protected section of class BinaryTree.

Although all its members are public, class Node is declared within the protected section of the LinkedBinaryTree class. Thus, it is not publicly accessible. Each node has a member variable *elt*, which contains the associated element, and pointers *par*, *left*, and *right*, which point to the associated relatives.

Next, we define the public class Position in Code Fragment 7.18. Its data member consists of a pointer v to a node of the tree. Access to the node's element is provided by overloading the dereferencing operator ("*"). We declare LinkedBinaryTree to be a friend, providing it access to the private data.

```
class Position {                                       // position in the tree
private:
  Node* v;                                             // pointer to the node
public:
  Position(Node* _v = NULL) : v(_v) { }                // constructor
  Elem& operator*()                                    // get element
    { return v->elt; }
  Position left() const                                // get left child
    { return Position(v->left); }
  Position right() const                               // get right child
    { return Position(v->right); }
  Position parent() const                              // get parent
    { return Position(v->par); }
  bool isRoot() const                                  // root of the tree?
    { return v->par == NULL; }
  bool isExternal() const                              // an external node?
    { return v->left == NULL && v->right == NULL; }
  friend class LinkedBinaryTree;                       // give tree access
};
typedef std::list<Position> PositionList;              // list of positions
```

Code Fragment 7.18: Class Position implementing a position in a binary tree. It is nested in the public section of class LinkedBinaryTree.

Most of the functions of class Position simply involve accessing the appropriate members of the Node structure. We have also included a declaration of the class PositionList, as an STL list of positions. This is used to represent collections of nodes. To keep the code simple, we have omitted error checking, and, rather than using templates, we simply provide a type definition for the base element type, called Elem. (See Exercise P-7.2.)

We present the major part of the class LinkedBinaryTree in Code Fragment 7.19. The class declaration begins by inserting the above declarations of Node and Position. This is followed by a declaration of the public members, local utility functions, and the private member data. We have omitted housekeeping functions, such as a destructor, assignment operator, and copy constructor.

```
typedef int Elem;                                    // base element type
class LinkedBinaryTree {
protected:
  // insert Node declaration here...
public:
  // insert Position declaration here...
public:
  LinkedBinaryTree();                                // constructor
  int size() const;                                  // number of nodes
  bool empty() const;                                // is tree empty?
  Position root() const;                             // get the root
  PositionList positions() const;                    // list of nodes
  void addRoot();                                    // add root to empty tree
  void expandExternal(const Position& p);            // expand external node
  Position removeAboveExternal(const Position& p); // remove p and parent
  // housekeeping functions omitted...
protected:                                           // local utilities
  void preorder(Node* v, PositionList& pl) const;    // preorder utility
private:
  Node* _root;                                       // pointer to the root
  int n;                                             // number of nodes
};
```

Code Fragment 7.19: Implementation of a LinkedBinaryTree class.

The private data for class LinkedBinaryTree consists of a pointer *_root* to the root node and a variable *n*, containing the number of nodes in the tree. (We added the underscore to the name *root* to avoid a name conflict with the member function root.) In addition to the functions of the ADT, we have introduced a few update functions, addRoot, expandExternal, and removeAboveExternal, which provide the means to build and modify trees. They are discussed below. We define a utility function preorder, which is used in the implementation of the function positions.

In Code Fragment 7.20, we present the definitions of the constructor and sim-

pler member functions of class LinkedBinaryTree. The function addRoot assumes that the tree is empty, and it creates a single root node. (It should not be invoked if the tree is nonempty, since otherwise a memory leak results.)

```
LinkedBinaryTree::LinkedBinaryTree()                    // constructor
  : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const                      // number of nodes
  { return n; }
bool LinkedBinaryTree::empty() const                    // is tree empty?
  { return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
  { return Position(_root); }
void LinkedBinaryTree::addRoot()                        // add root to empty tree
  { _root = new Node; n = 1; }
```

Code Fragment 7.20: Simple member functions for class LinkedBinaryTree.

Binary Tree Update Functions

In addition to the BinaryTree interface functions and addRoot, the class LinkedBinaryTree also includes the following update functions given a position p. The first is used for adding nodes to the tree and the second is used for removing nodes.

expandExternal(p): Transform p from an external node into an internal node by creating two new external nodes and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.

removeAboveExternal(p): Remove the external node p together with its parent q, replacing q with the sibling of p (see Figure 7.15, where p's node is w and q's node is v); an error condition occurs if p is an internal node or p is the root.

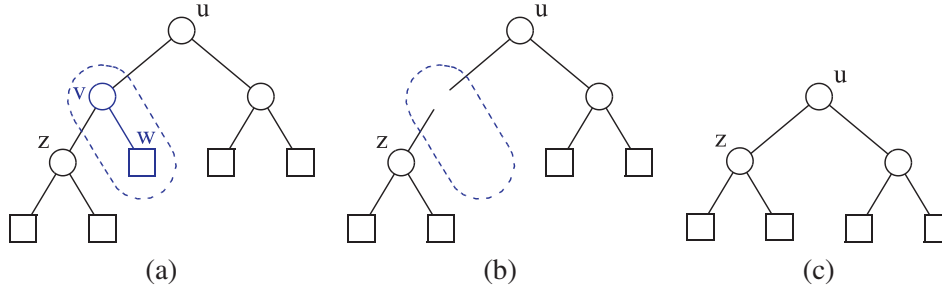


Figure 7.15: Operation removeAboveExternal(p), which removes the external node w to which p refers and its parent node v.

The function expandExternal(p) is shown in Code Fragment 7.21. Letting v be p's associated node, it creates two new nodes. One becomes v's left child and the other becomes v's right child. The constructor for Node initializes the node's pointers to NULL, so we need only update the new node's parent links.

```
                                                    // expand external node
void LinkedBinaryTree::expandExternal(const Position& p) {
  Node* v = p.v;                                    // p's node
  v->left = new Node;                               // add a new left child
  v->left->par = v;                                 // v is its parent
  v->right = new Node;                              // and a new right child
  v->right->par = v;                                // v is its parent
  n += 2;                                           // two more nodes
}
```

Code Fragment 7.21: The function expandExternal(p) of class LinkedBinaryTree.

The function removeAboveExternal(p) is shown in Code Fragment 7.22. Let w be p's associated node and let v be its parent. We assume that w is external and is not the root. There are two cases. If w is a child of the root, removing w and its parent (the root) causes w's sibling to become the tree's new root. If not, we replace w's parent with w's sibling. This involves finding w's grandparent and determining whether v is the grandparent's left or right child. Depending on which, we set the link for the appropriate child of the grandparent. After unlinking w and v, we delete these nodes. Finally, we update the number of nodes in the tree.

```
LinkedBinaryTree::Position                          // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p) {
  Node* w = p.v; Node* v = w->par;                  // get p's node and parent
  Node* sib = (w == v->left ? v->right : v->left);
  if (v == _root) {                                 // child of root?
    _root = sib;                                    // ...make sibling root
    sib->par = NULL;
  }
  else {
    Node* gpar = v->par;                            // w's grandparent
    if (v == gpar->left) gpar->left = sib;          // replace parent by sib
    else gpar->right = sib;
    sib->par = gpar;
  }
  delete w; delete v;                               // delete removed nodes
  n -= 2;                                           // two fewer nodes
  return Position(sib);
}
```

Code Fragment 7.22: An implementation of the function removeAboveExternal(p).

The function positions is shown in Code Fragment 7.23. It invokes the utility function preorder, which traverses the tree and stores the node positions in an STL vector.

```
                                                    // list of all nodes
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder(_root, pl);                              // preorder traversal
  return PositionList(pl);                          // return resulting list
}
                                                    // preorder traversal
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const {
  pl.push_back(Position(v));                        // add this node
  if (v->left != NULL)                              // traverse left subtree
    preorder(v->left, pl);
  if (v->right != NULL)                             // traverse right subtree
    preorder(v->right, pl);
}
```

Code Fragment 7.23: An implementation of the function positions.

We have omitted the housekeeping functions (the destructor, copy constructor, and assignment operator). We leave these as exercises (Exercise C-7.22), but they also involve performing a traversal of the tree.

Performance of the LinkedBinaryTree Implementation

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

- Each of the position functions left, right, parent, isRoot, and isExternal takes $O(1)$ time.
- By accessing the member variable n, which stores the number of nodes of T, functions size and empty each run in $O(1)$ time.
- The accessor function root runs in $O(1)$ time.
- The update functions expandExternal and removeAboveExternal visit only a constant number of nodes, so they both run in $O(1)$ time.
- Function positions is implemented by performing a preorder traversal, which takes $O(n)$ time. (We discuss three different binary-tree traversals in Section 7.3.6. Any of these suffice.) The nodes visited by the traversal are each added in $O(1)$ time to an STL list. Thus, function positions takes $O(n)$ time.

Table 7.2 summarizes the performance of this implementation of a binary tree. There is an object of class Node (Code Fragment 7.17) for each node of tree T. Thus, the overall space requirement is $O(n)$.

Operation	*Time*
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

Table 7.2: Running times for the functions of an n-node binary tree implemented with a linked structure. The space usage is $O(n)$.

7.3.5 A Vector-Based Structure for Binary Trees

A simple structure for representing a binary tree T is based on a way of numbering the nodes of T. For every node v of T, let $f(v)$ be the integer defined as follows:

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

The numbering function f is known as a ***level numbering*** of the nodes in a binary tree T, because it numbers the nodes on each level of T in increasing order from left to right, although it may skip some numbers. (See Figure 7.16.)

The level numbering function f suggests a representation of a binary tree T by means of a vector S, such that node v of T is associated with the element of S at rank $f(v)$. (See Figure 7.17.) Typically, we realize the vector S by means of an extendable array. (See Section 6.1.3.) Such an implementation is simple and efficient, for we can use it to easily perform the functions root, parent, left, right, sibling, isExternal, and isRoot by using simple arithmetic operations on the numbers $f(v)$ associated with each node v involved in the operation. That is, each position object v is simply a "wrapper" for the index $f(v)$ into the vector S. We leave the details of such implementations as a simple exercise (R-7.26).

Let n be the number of nodes of T, and let f_M be the maximum value of $f(v)$ over all the nodes of T. The vector S has size $N = f_M + 1$, 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. For a tree of height h, $N = O(2^h)$. In the worst case, this can be as high as $2^n - 1$. The justification is left as an exercise (R-7.24). In Section 8.3, we discuss 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 the running times of the functions of a binary tree implemented with a vector. We do not include any tree update functions here. The vector implementation of a binary tree is a fast and easy way of realizing the binary-tree ADT, but it can be very space inefficient if the height of the tree is large.

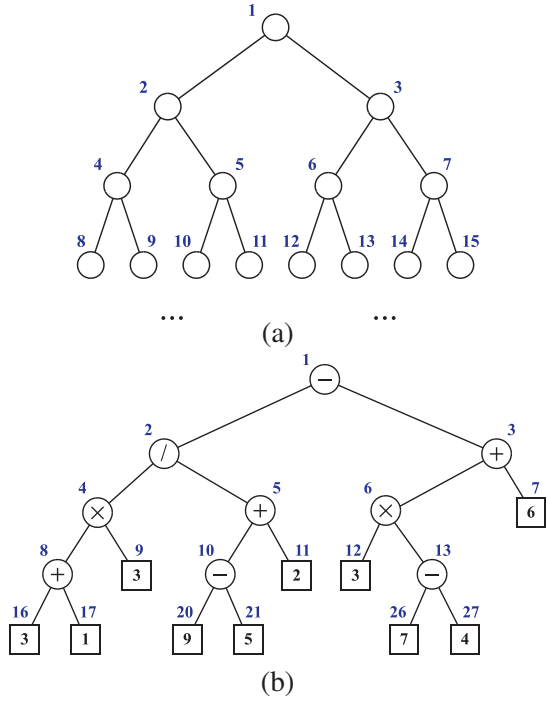


Figure 7.16: Binary tree level numbering: (a) general scheme; (b) an example.

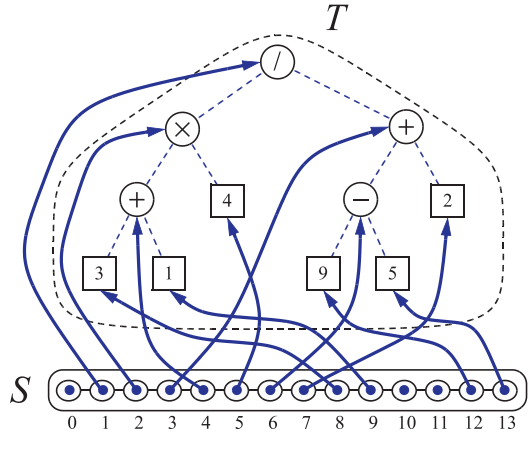


Figure 7.17: Representation of a binary tree T by means of a vector S.

Operation	Time
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

Table 7.3: Running times for a binary tree T implemented with a vector 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 a Binary Tree

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

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 7.9) 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 7.24. (Also see Code Fragment 7.23.)

Algorithm binaryPreorder(T, p):
  perform the "visit" action for node p
  if p is an internal node **then**
    binaryPreorder(T, p.left()) {recursively traverse left subtree}
    binaryPreorder(T, p.right()) {recursively traverse right subtree}

Code Fragment 7.24: Algorithm binaryPreorder, which performs the preorder traversal of the subtree of a binary tree T rooted at node p.

For example, a preorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle$LAX, BWI, ATL, JFK, PVD$\rangle$. 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 7.12) can be specialized for binary trees as shown in Code Fragment 7.25.

A postorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle$ATL, JFK, BWI, PVD, LAX$\rangle$.

Algorithm binaryPostorder(T, p):
  if p is an internal node **then**
    binaryPostorder(T, p.left()) {recursively traverse left subtree}
    binaryPostorder(T, p.right()) {recursively traverse right subtree}
  perform the "visit" action for the node p

Code Fragment 7.25: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node p.

Evaluating an Arithmetic Expression

The postorder traversal of a binary tree can be used to solve the expression 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 7.26, evaluates the expression associated with the subtree rooted at a node p of an arithmetic-expression tree T by performing a postorder traversal of T starting at p. In this case, the "visit" action consists of performing a single arithmetic operation.

Algorithm evaluateExpression(T, p):
  if p is an internal node **then**
    $x \leftarrow$ evaluateExpression(T, p.left())
    $y \leftarrow$ evaluateExpression(T, p.right())
    Let $\circ$ be the operator associated with p
    return $x \circ y$
  else
    return the value stored at p

Code Fragment 7.26: Algorithm evaluateExpression for evaluating the expression represented by the subtree of an arithmetic-expression tree T rooted at node p.

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. The specialization of the postorder traversal for binary trees simplifies that for general trees, however, because we use the left and right functions to avoid a loop that iterates through the children of an internal node.

Interestingly, the specialization of the general preorder and postorder traversal

methods to binary trees suggests a third traversal in a binary tree that is different from both the preorder and postorder traversals. We explore this third kind of traversal for binary trees in the next subsection.

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 p in a binary tree T is given in Code Fragment 7.27.

Algorithm inorder(T, p):
  if p is an internal node **then**
    inorder$(T, p.\text{left}())$ {recursively traverse left subtree}
  perform the "visit" action for node p
  if p is an internal node **then**
    inorder$(T, p.\text{right}())$ {recursively traverse right subtree}

Code Fragment 7.27: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node p.

For example, an inorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle \text{ATL}, \text{BWI}, \text{JFK}, \text{LAX}, \text{PVD} \rangle$. 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 p, the inorder traversal visits p after all the nodes in the left subtree of p and before all the nodes in the right subtree of p. (See Figure 7.18.)

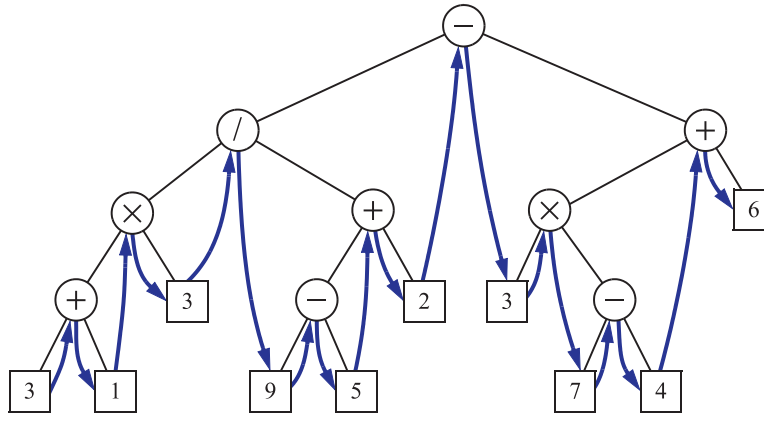


Figure 7.18: Inorder traversal of a binary tree.

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 p of T stores an element of S, denoted with $x(p)$
- For each internal node p of T, the elements stored in the left subtree of p are less than or equal to $x(p)$ and the elements stored in the right subtree of p are greater than or equal to $x(p)$
- 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 7.19.)

We can use a binary search tree T to locate an element with a certain value x by traversing down the tree T. At each internal node we compare the value of the current node to our search element x. If the answer to the question is "smaller," then the search continues in the left subtree. If the answer is "equal," then the search terminates successfully. If the answer is "greater," then the search continues in the right subtree. Finally, if we reach an external node (which is empty), then the search terminates unsuccessfully. (See Figure 7.19.)

Note that the time for searching in a binary search tree T is proportional to the height of T. Recall from Proposition 7.10 that the height of a tree with n nodes can be as small as $O(\log n)$ or as large as $\Omega(n)$. Thus, binary search trees are most efficient when they have small height. We illustrate an example search in a binary search tree in Figure 7.19. We study binary search trees in more detail in Section 10.1.

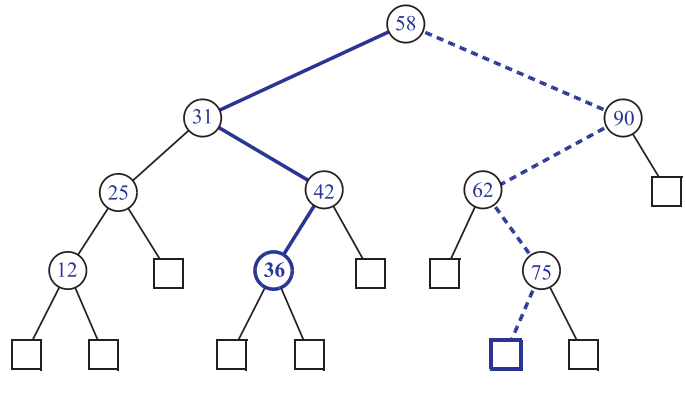


Figure 7.19: A binary search tree storing integers. The blue solid path is traversed when searching (successfully) for 36. The blue dashed path is traversed when searching (unsuccessfully) for 70.

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 p of T using the following two rules (see Figure 7.20).

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

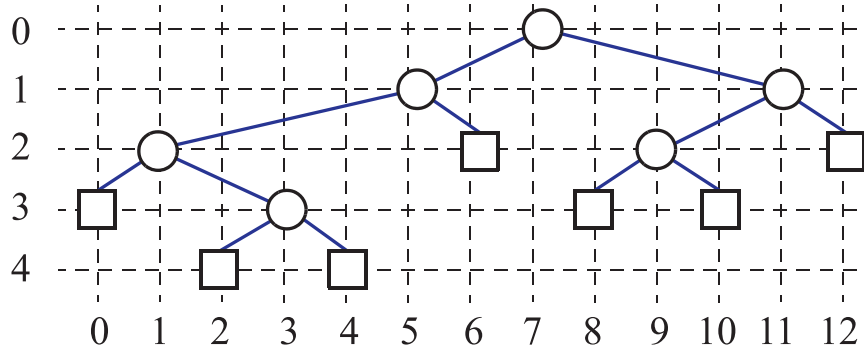


Figure 7.20: The inorder drawing algorithm for a binary tree.

The Euler Tour Traversal of a Binary Tree

The tree-traversal algorithms we have discussed so far are all forms of iterators. Each traversal visits the nodes of a tree in a certain order, and is guaranteed to visit each node exactly once. We can unify the tree-traversal algorithms given above into a single framework, however, by relaxing the requirement that each node be visited exactly once. The resulting traversal method is called the ***Euler tour traversal***, which we study next. The advantage of this traversal is that it allows for more general kinds of algorithms to be expressed easily.

The Euler tour traversal of a binary tree T can be informally defined as a "walk" around T, where we start by going from the root toward its left child, viewing the edges of T as being "walls" that we always keep to our left. (See Figure 7.21.) Each node p of T is encountered three times by the Euler tour:

- "On the left" (before the Euler tour of p's left subtree)
- "From below" (between the Euler tours of p's two subtrees)
- "On the right" (after the Euler tour of p's right subtree)

If p is external, then these three "visits" actually all happen at the same time.

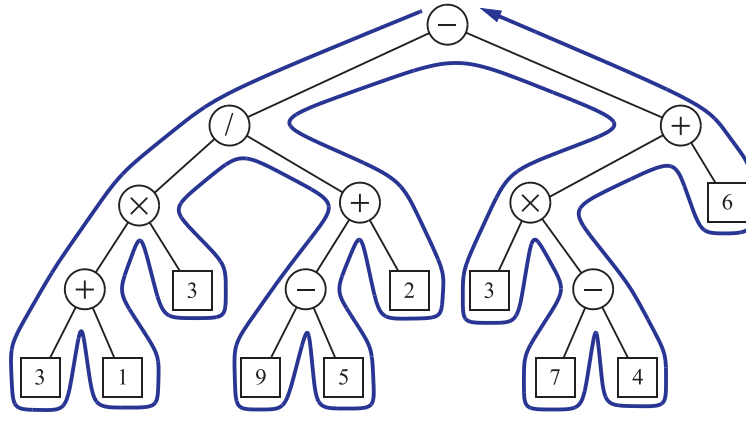


Figure 7.21: Euler tour traversal of a binary tree.

We give pseudo-code for the Euler tour of the subtree rooted at a node p in Code Fragment 7.28.

Algorithm eulerTour(T, p):
  perform the action for visiting node p on the left
  if p is an internal node **then**
    recursively tour the left subtree of p by calling eulerTour$(T, p.\text{left}())$
  perform the action for visiting node p from below
  if p is an internal node **then**
    recursively tour the right subtree of p by calling eulerTour$(T, p.\text{right}())$
  perform the action for visiting node p on the right

Code Fragment 7.28: Algorithm eulerTour for computing the Euler tour traversal of the subtree of a binary tree T rooted at a node p.

The preorder traversal of a binary tree is equivalent to an Euler tour traversal in which each node has an associated "visit" action occur only when it is encountered on the left. Likewise, the inorder and postorder traversals of a binary tree are equivalent to an Euler tour, where each node has an associated "visit" action occur only when it is encountered from below or on the right, respectively.

The Euler tour traversal extends the preorder, inorder, and postorder traversals, but it can also perform other kinds of traversals. For example, suppose we wish to compute the number of descendants of each node p in an n node binary tree T. We start an Euler tour by initializing a counter to 0, and then increment the counter each time we visit a node on the left. To determine the number of descendants of a node p, we compute the difference between the values of the counter when p is visited on the left and when it is visited on the right, and add 1. This simple rule gives us the number of descendants of p, because each node in the subtree rooted

at p is counted between p's visit on the left and p's visit on the right. Therefore, we have an $O(n)$-time method for computing the number of descendants of each node in T.

The running time of the Euler tour traversal is easy to analyze, assuming that visiting a node takes $O(1)$ time. Namely, in each traversal, we spend a constant amount of time at each node of the tree during the traversal, so the overall running time is $O(n)$ for an n node tree.

Another application of the Euler tour traversal is to print a fully parenthesized arithmetic expression from its expression tree (Example 7.9). Algorithm printExpression, shown in Code Fragment 7.29, accomplishes this task by performing the following actions in an Euler tour:

- "On the left" action: if the node is internal, print "("
- "From below" action: print the value or operator stored at the node
- "On the right" action: if the node is internal, print ")"

Algorithm printExpression(T, p):
  if p.isExternal() **then**
    print the value stored at p
  else
    print "("
    printExpression(T, p.left())
    print the operator stored at p
    printExpression(T, p.right())
    print ")"

Code Fragment 7.29: An algorithm for printing the arithmetic expression associated with the subtree of an arithmetic-expression tree T rooted at p.

7.3.7 The Template Function Pattern

The tree traversal functions described above are actually examples of an interesting object-oriented software design pattern, the ***template function pattern***. This is not to be confused with templated classes or functions in C++, but the principal is similar. The template function pattern describes a generic computation mechanism that can be specialized for a particular application by redefining certain steps.

Euler Tour with the Template Function Pattern

Following the template function pattern, we can design an algorithm, templateEulerTour, that implements a generic Euler tour traversal of a binary tree. When

called on a node p, function templateEulerTour calls several other auxiliary functions at different phases of the traversal. First of all, it creates a three-element structure r to store the result of the computation calling auxiliary function initResult. Next, if p is an external node, templateEulerTour calls auxiliary function visitExternal, else (p is an internal node) templateEulerTour executes the following steps:

- Calls auxiliary function visitLeft, which performs the computations associated with encountering the node on the left
- Recursively calls itself on the left child
- Calls auxiliary function visitBelow, which performs the computations associated with encountering the node from below
- Recursively calls itself on the right subtree
- Calls auxiliary function visitRight, which performs the computations associated with encountering the node on the right

Finally, templateEulerTour returns the result of the computation by calling auxiliary function returnResult. Function templateEulerTour can be viewed as a ***template*** or "skeleton" of an Euler tour. (See Code Fragment 7.30.)

Algorithm templateEulerTour(T,p):
  $r \leftarrow$ initResult()
  if p.isExternal() **then**
    $r.finalResult \leftarrow$ visitExternal(T,p,r)
  else
    visitLeft(T,p,r)
    $r.leftResult \leftarrow$ templateEulerTour$(T,p.\text{left}())$
    visitBelow(T,p,r)
    $r.rightResult \leftarrow$ templateEulerTour$(T,p.\text{right}())$
    visitRight(T,p,r)
  return returnResult(r)

Code Fragment 7.30: Function templateEulerTour for computing a generic Euler tour traversal of the subtree of a binary tree T rooted at a node p, following the template function pattern. This function calls the functions initResult, visitExternal, visitLeft, visitBelow, visitRight, and returnResult.

In an object-oriented context, we can then write a class EulerTour that:

- Contains function templateEulerTour
- Contains all the auxiliary functions called by templateEulerTour as empty place holders (that is, with no instructions or returning NULL)
- Contains a function execute that calls templateEulerTour$(T,T.\text{root}())$

Class EulerTour itself does not perform any useful computation. However, we can extend it with the inheritance mechanism and override the empty functions to do useful tasks.

Template Function Examples

As a first example, we can evaluate the expression associated with an arithmetic-expression tree (see Example 7.9) by writing a new class EvaluateExpression that:

- Extends class EulerTour
- Overrides function initResult by returning an array of three numbers
- Overrides function visitExternal by returning the value stored at the node
- Overrides function visitRight by combining *r.leftResult* and *r.rightResult* with the operator stored at the node, and setting *r.finalResult* equal to the result of the operation
- Overrides function returnResult by returning *r.finalResult*

This approach should be compared with the direct implementation of the algorithm shown in Code Fragment 7.26.

As a second example, we can print the expression associated with an arithmetic-expression tree (see Example 7.9) using a new class PrintExpression that:

- Extends class EulerTour
- Overrides function visitExternal by printing the value of the variable or constant associated with the node
- Overrides function visitLeft by printing "("
- Overrides function visitBelow by printing the operator associated with the node
- Overrides function visitRight by printing ")"

This approach should be compared with the direct implementation of the algorithm shown in Code Fragment 7.29.

C++ Implementation

A complete C++ implementation of the generic EulerTour class and of its specializations EvaluateExpressionTour and PrintExpressionTour are shown in Code Fragments 7.31 through 7.34. These are based on a linked binary tree implementation.

We begin by defining a local structure Result with fields *leftResult*, *rightResult*, and *finalResult*, which store the intermediate results of the tour. In order to avoid typing lengthy qualified type names, we give two type definitions, BinaryTree and Position, for the tree and a position in the tree, respectively. The only data member is a pointer to the binary tree. We provide a simple function, called initialize, that sets this pointer to an existing binary tree. The remaining functions are protected,

since they are not invoked directly, but rather by the derived classes, which produce the desired specialized behavior.

```
template <typename E, typename R>      // element and result types
class EulerTour {                      // a template for Euler tour
protected:
  struct Result {                      // stores tour results
    R leftResult;                      // result from left subtree
    R rightResult;                     // result from right subtree
    R finalResult;                     // combined result
  };
  typedef BinaryTree<E> BinaryTree;    // the tree
  typedef typename BinaryTree::Position Position; // a position in the tree
protected:                             // data member
  const BinaryTree* tree;              // pointer to the tree
public:
  void initialize(const BinaryTree& T) // initialize
    { tree = &T; }
protected:                             // local utilities
  int eulerTour(const Position& p) const;  // perform the Euler tour
                                       // functions given by subclasses
  virtual void visitExternal(const Position& p, Result& r) const {}
  virtual void visitLeft(const Position& p, Result& r) const {}
  virtual void visitBelow(const Position& p, Result& r) const {}
  virtual void visitRight(const Position& p, Result& r) const {}
  Result initResult() const { return Result(); }
  int result(const Result& r) const { return r.finalResult; }
};
```

Code Fragment 7.31: Class EulerTour defining a generic Euler tour of a binary tree. This class realizes the template function pattern and must be specialized in order to generate an interesting computation.

Next, in Code Fragment 7.32, we present the principal traversal function, called eulerTour. This recursive function performs an Euler traversal on the tree and invokes the appropriate functions as it goes. If run on the generic Euler tree, nothing interesting would result, because these functions (as defined in Code Fragment 7.31) do nothing. It is up to the derived functions to provide more interesting definitions for these generic functions.

In Code Fragment 7.33, we present our first example of a derived class using the template pattern, called EvaluateExpressionTour. It evaluates an integer arithmetic-expression tree. We assume that each external node of an expression tree provides a function called value, which returns the value associated with this node. We assume that each internal node of an expression tree provides a function called operation, which performs the operation associated with this node to the two operands arising from its left and right subtrees, and returns the result.

```
template <typename E, typename R>     // do the tour
int EulerTour<E, R>::eulerTour(const Position& p) const {
  Result r = initResult();
  if (p.isExternal()) {                          // external node
    visitExternal(p, r);
  }
  else {                                         // internal node
    visitLeft(p, r);
    r.leftResult = eulerTour(p.left());          // recurse on left
    visitBelow(p, r);
    r.rightResult = eulerTour(p.right());        // recurse on right
    visitRight(p, r);
  }
  return result(r);
}
```

Code Fragment 7.32: The principal member function eulerTour, which recursively traverses the tree and accumulates the results.

Using these two functions, we can evaluate the expression recursively as we traverse the tree. The main entry point is the function execute, which initializes the tree, invokes the recursive Euler tour starting at the root, and prints the final result. For example, given the expression tree of Figure 7.21, this procedure would output the string "The value is: -13".

```
template <typename E, typename R>
class EvaluateExpressionTour : public EulerTour<E, R> {
protected:                                       // shortcut type names
  typedef typename EulerTour<E, R>::BinaryTree BinaryTree;
  typedef typename EulerTour<E, R>::Position Position;
  typedef typename EulerTour<E, R>::Result Result;
public:
  void execute(const BinaryTree& T) {            // execute the tour
    initialize(T);
    std::cout << "The value is: " << eulerTour(T.root()) << "\n";
  }
protected:                                       // leaf: return value
  virtual void visitExternal(const Position& p, Result& r) const
    { r.finalResult = (*p).value(); }
                                                 // internal: do operation
  virtual void visitRight(const Position& p, Result& r) const
    { r.finalResult = (*p).operation(r.leftResult, r.rightResult); }
};
```

Code Fragment 7.33: Implementation of class EvaluateExpressionTour which specializes EulerTour to evaluate the expression associated with an arithmetic-expression tree.

Finally, in Code Fragment 7.34, we present a second example of a derived class, called PrintExpressionTour. In contrast to the previous function, which evaluates the value of an expression tree, this one prints the expression. We assume that each node of an expression tree provides a function called print. For each external node, this function prints the value associated with this node. For each internal node, this function prints the operator, for example, printing "+" for addition or "*" for multiplication.

```
template <typename E, typename R>
class PrintExpressionTour : public EulerTour<E, R> {
protected: // ...same type name shortcuts as in EvaluateExpressionTour
public:
  void execute(const BinaryTree& T) {      // execute the tour
    initialize(T);
    cout << "Expression: "; eulerTour(T.root()); cout << endl;
  }
protected:                                  // leaf: print value
  virtual void visitExternal(const Position& p, Result& r) const
    { (*p).print(); }
                                            // left: open new expression
  virtual void visitLeft(const Position& p, Result& r) const
    { cout << "("; }
                                            // below: print operator
  virtual void visitBelow(const Position& p, Result& r) const
    { (*p).print(); }
                                            // right: close expression
  virtual void visitRight(const Position& p, Result& r) const
    { cout << ")"; }
};
```

Code Fragment 7.34: A class that prints an arithmetic-expression tree.

When entering a subtree, the function visitLeft has been overridden to print "(" and on exiting a subtree, the function visitRight has been overridden to print ")." The main entry point is the function execute, which initializes the tree, and invokes the recursive Euler tour starting at the root. When combined, these functions print the entire expression (albeit with lots of redundant parentheses). For example, given the expression tree of Figure 7.21, this procedure would output the following string.

```
((((3 + 1) * 3) / ((9 - 5) + 2)) - ((3 * (7 - 4)) + 6))
```

7.3.8 Representing General Trees with Binary Trees

An alternative representation of a general tree T is obtained by transforming T into a binary tree T'. (See Figure 7.22.) We assume that either T is ordered or that it has been arbitrarily ordered. The transformation is as follows:

- For each node u of T, there is an internal node u' of T' associated with u
- If u is an external node of T and does not have a sibling immediately following it, then the children of u' in T' are external nodes
- If u is an internal node of T and v is the first child of u in T, then v' is the left child of u' in T
- If node v has a sibling w immediately following it, then w' is the right child of v' in T'

Note that the external nodes of T' are not associated with nodes of T, and serve only as place holders (hence, may even be null).

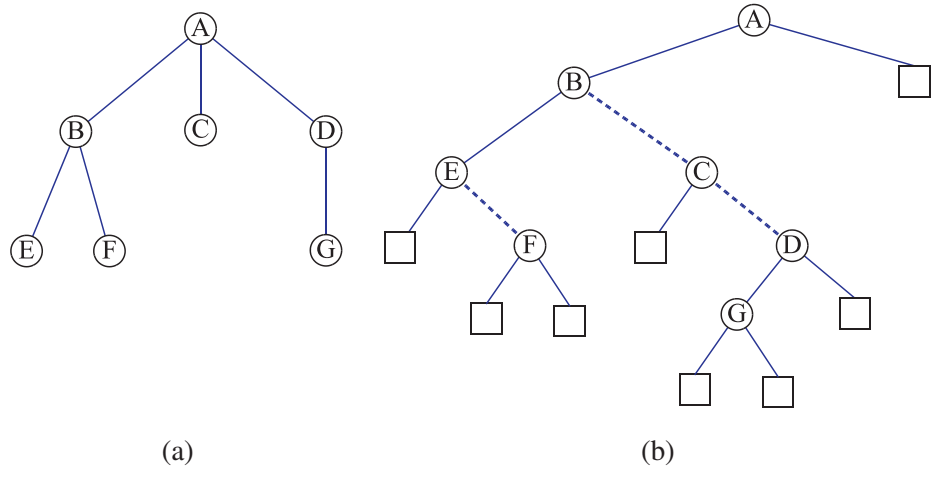


Figure 7.22: Representation of a tree by means of a binary tree: (a) tree T; (b) binary tree T' associated with T. The dashed edges connect nodes of T' associated with sibling nodes of T.

It is easy to maintain the correspondence between T and T', and to express operations in T in terms of corresponding operations in T'. Intuitively, we can think of the correspondence in terms of a conversion of T into T' that takes each set of siblings $\{v_1, v_2, \ldots, v_k\}$ in T with parent v and replaces it with a chain of right children rooted at v_1, which then becomes the left child of v.

7.4 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-7.1 Describe an algorithm for counting the number of left external nodes in a binary tree, using the Binary tree ADT.

R-7.2 The following questions refer to the tree of Figure 7.3.

a. Which node is the root?
b. What are the internal nodes?
c. How many descendents does node cs016/ have?
d. How many ancestors does node cs016/ have?
e. What are the siblings of node homeworks/?
f. Which nodes are in the subtree rooted at node projects/?
g. What is the depth of node papers/?
h. What is the height of the tree?

R-7.3 Find the value of the arithmetic expression associated with each subtree of the binary tree of Figure 7.11.

R-7.4 Let T be an n-node improper binary tree (that is, each internal node has one or two children). Describe how to represent T by means of a ***proper*** binary tree T' with $O(n)$ nodes.

R-7.5 What are the minimum and maximum number of internal and external nodes in an improper binary tree with n nodes?

R-7.6 Show a tree achieving the worst-case running time for algorithm depth.

R-7.7 Give a justification of Proposition 7.4.

R-7.8 What is the running time of algorithm height2(T, v) (Code Fragment 7.7) when called on a node v distinct from the root of T?

R-7.9 Let T be the tree of Figure 7.3.

a. Give the output of preorderPrint$(T, T.\text{root}())$ (Code Fragment 7.10).
b. Give the output of parenPrint$(T, T.\text{root}())$(Code Fragment cod:paren:Print).

R-7.10 Describe a modification to the parenPrint function given in Code Fragment 7.11, so that it uses the size function for string objects to output the parenthetic representation of a tree with line breaks and spaces added to display the tree in a text window that is 80 characters wide.

R-7.11 Draw an arithmetic-expression tree that has four external nodes, storing the numbers 1, 5, 6, and 7 (with each number stored in a distinct external node, but not necessarily in this order), and has three internal nodes, each storing an operator from the set $\{+, -, \times, /\}$, so that the value of the root is 21. The operators may return and act on fractions, and an operator may be used more than once.

R-7.12 Let T be an ordered tree with more than one node. Is it possible that the preorder traversal of T visits the nodes in the same order as the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur. Likewise, is it possible that the preorder traversal of T visits the nodes in the reverse order of the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur.

R-7.13 Answer the previous question for the case when T is a proper binary tree with more than one node.

R-7.14 Let T be a tree with n nodes. What is the running time of the function parenPrint$(T, T.\text{root}())$? (See Code Fragment 7.11.)

R-7.15 Draw a (single) binary tree T, such that:

- Each internal node of T stores a single character
- A *preorder* traversal of T yields EXAMFUN
- An *inorder* traversal of T yields MAFXUEN

R-7.16 Answer the following questions so as to justify Proposition 7.10.

a. What is the minimum number of external nodes for a binary tree with height h? Justify your answer.

b. What is the maximum number of external nodes for a binary tree with height h? Justify your answer.

c. Let T be a binary tree with height h and n nodes. Show that

$$\log(n+1) - 1 \leq h \leq (n-1)/2.$$

d. For which values of n and h can the above lower and upper bounds on h be attained with equality?

R-7.17 Describe a generalization of the Euler tour traversal of trees such that each internal node has three children. Describe how you could use this traversal to compute the height of each node in such a tree.

R-7.18 Modify the C++ function preorderPrint, given in Code Fragment 7.10, so that it will print the strings associated with the nodes of a tree one per line, and indented proportionally to the depth of the node.

R-7.19 Let T be the tree of Figure 7.3. Draw, as best as you can, the output of the algorithm postorderPrint$(T, T.\text{root}())$ (Code Fragment 7.13).

R-7.20 Let T be the tree of Figure 7.9. Compute, in terms of the values given in this figure, the output of algorithm diskSpace$(T, T.\text{root}())$. (See Code Fragment 7.14.)

R-7.21 Let T be the binary tree of Figure 7.11.

a. Give the output of preorderPrint$(T, T.\text{root}())$ (Code Fragment 7.10).

b. Give the output of the function printExpression$(T, T.\text{root}())$ (Code Fragment 7.29).

R-7.22 Describe, in pseudo-code, an algorithm for computing the number of descendents of each node of a binary tree. The algorithm should be based on the Euler tour traversal.

R-7.23 Let T be a (possibly improper) binary tree with n nodes, and let D be the sum of the depths of all the external nodes of T. Show that if T has the minimum number of external nodes possible, then D is $O(n)$ and if T has the maximum number of external nodes possible, then D is $O(n \log n)$.

R-7.24 Let T be a binary tree with n nodes, and let f be the level numbering of the nodes of T as given in Section 7.3.5.

a. Show that, for every node v of T, $f(v) \leq 2^n - 1$.

b. Show an example of a binary tree with seven nodes that attains the above upper bound on $f(v)$ for some node v.

R-7.25 Draw the binary tree representation of the following arithmetic expression: "$(((5+2)*(2-1))/((2+9)+((7-2)-1))*8)$."

R-7.26 Let T be a binary tree with n nodes that is realized with a vector, S, and let f be the level numbering of the nodes in T as given in Section 7.3.5. Give pseudo-code descriptions of each of the functions root, parent, leftChild, rightChild, isExternal, and isRoot.

R-7.27 Show how to use the Euler tour traversal to compute the level number, defined in Section 7.3.5, of each node in a binary tree T.

Creativity

C-7.1 Show that there are more than 2^n different potentially improper binary trees with n internal nodes, where two trees are considered different if they can be drawn as different looking trees.

C-7.2 Describe an efficient algorithm for converting a fully balanced string of parentheses into an equivalent tree. The tree associated with such a string is defined recursively. The outer-most pair of balanced parentheses is associated with the root and each substring inside this pair, defined by the substring between two balanced parentheses, is associated with a subtree of this root.

C-7.3 For each node v in a tree T, let $pre(v)$ be the rank of v in a preorder traversal of T, let $post(v)$ be the rank of v in a postorder traversal of T, let $depth(v)$ be the depth of v, and let $desc(v)$ be the number of descendents of v, not counting v itself. Derive a formula defining $post(v)$ in terms of $desc(v)$, $depth(v)$, and $pre(v)$, for each node v in T.

C-7.4 Let T be a tree whose nodes store strings. Give an algorithm that computes and prints, for every internal node v of T, the string stored at v and the height of the subtree rooted at v.

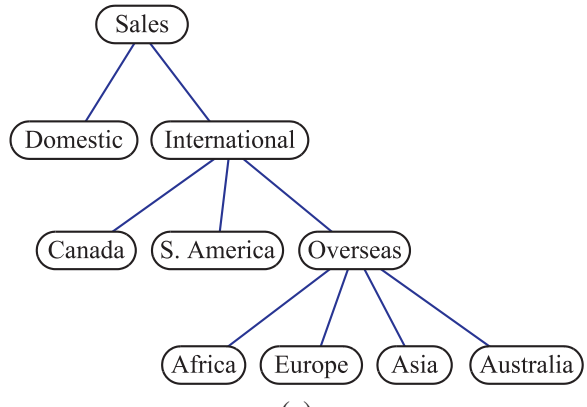
C-7.5 Design algorithms for the following operations for a binary tree T.

- preorderNext(v): return the node visited after node v in a preorder traversal of T.
- inorderNext(v): return the node visited after node v in an inorder traversal of T.
- postorderNext(v): return the node visited after node v in a postorder traversal of T.

What are the worst-case running times of your algorithms?

C-7.6 Give an $O(n)$-time algorithm for computing the depth of all the nodes of a tree T, where n is the number of nodes of T.

C-7.7 The ***indented parenthetic representation*** of a tree T is a variation of the parenthetic representation of T (see Figure 7.7) that uses indentation and line breaks as illustrated in Figure 7.23. Give an algorithm that prints this representation of a tree.



(a)

```
Sales (
  Domestic
  International (
    Canada
    S. America
    Overseas (
      Africa
      Europe
      Asia
      Australia
    )
  )
)
```

(b)

Figure 7.23: (a) Tree T; (b) indented parenthetic representation of T.

C-7.8 Let T be a (possibly improper) binary tree with n nodes, and let D be the sum of the depths of all the external nodes of T. Describe a configuration for T such that D is $\Omega(n^2)$. Such a tree would be the worst case for the asymptotic running time of Algorithm height1 (Code Fragment 7.6).

C-7.9 For a tree T, let n_I denote the number of its internal nodes, and let n_E denote the number of its external nodes. Show that if every internal node in T has exactly 3 children, then $n_E = 2n_I + 1$.

C-7.10 The update operations expandExternal and removeAboveExternal do not permit the creation of an improper binary tree. Give pseudo-code descriptions for alternate update operations suitable for improper binary trees. You may need to define new query operations as well.

C-7.11 The ***balance factor*** of an internal node v of a binary tree is the difference between the heights of the right and left subtrees of v. Show how to specialize the Euler tour traversal of Section 7.3.7 to print the balance factors of all the nodes of a binary tree.

C-7.12 Two ordered trees T' and T'' are said to be ***isomorphic*** if one of the following holds:

- Both T' and T'' consist of a single node
- Both T' and T'' have the same number k of subtrees, and the ith subtree of T' is isomorphic to the ith subtree of T'', for $i = 1, \ldots, k$.

Design an algorithm that tests whether two given ordered trees are isomorphic. What is the running time of your algorithm?

C-7.13 Extend the concept of an Euler tour to an ordered tree that is not necessarily a binary tree.

C-7.14 As mentioned in Exercise C-5.8, ***postfix notation*** is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if "$(exp_1) \circ (exp_2)$" is a normal (infix) fully parenthesized expression with operation "$\circ$," then its postfix equivalent is "$pexp_1\ pexp_2 \circ$," where $pexp_1$ is the postfix version of exp_1 and $pexp_2$ is the postfix version of exp_2. The postfix version of a single number of variables is just that number or variable. So, for example, the postfix version of the infix expression "$((5+2)*(8-3))/4$" is "$5\ 2 + 8\ 3 - * 4\ /$." Give an efficient algorithm, that when given an expression tree, outputs the expression in postfix notation.

C-7.15 Given a proper binary tree T, define the ***reflection*** of T to be the binary tree T' such that each node v in T is also in T', but the left child of v in T is v's right child in T' and the right child of v in T is v's left child in T'. Show that a preorder traversal of a proper binary tree T is the same as the postorder traversal of T's reflection, but in reverse order.

C-7.16 Algorithm preorderDraw draws a binary tree T by assigning x- and y-coordinates to each node v such that $x(v)$ is the number of nodes preceding v in the preorder traversal of T and $y(v)$ is the depth of v in T. Algorithm postorderDraw is similar to preorderDraw but assigns x-coordinates using a postorder traversal.

a. Show that the drawing of T produced by preorderDraw has no pairs of crossing edges.
b. Redraw the binary tree of Figure 7.20 using preorderDraw.
c. Show that the drawing of T produced by postorderDraw has no pairs of crossing edges.
d. Redraw the binary tree of Figure 7.20 using postorderDraw.

C-7.17 Let a visit action in the Euler tour traversal be denoted by a pair (v,a), where v is the visited node and a is one of ***left***, ***below***, or ***right***. Design an algorithm for performing operation tourNext(v,a), which returns the visit action (w,b) following (v,a). What is the worst-case running time of your algorithm?

C-7.18 Algorithm preorderDraw draws a binary tree T by assigning x- and y-coordinates to each node v as follows:

- Set $x(v)$ equal to the number of nodes preceding v in the preorder traversal of T.
- Set $y(v)$ equal to the depth of v in T.

a. Show that the drawing of T produced by algorithm preorderDraw has no pairs of crossing edges.
b. Use algorithm preorderDraw to redraw the binary tree shown in Figure 7.20.
c. Use algorithm postorderDraw, which is similar to preorderDraw but assigns x-coordinates using a postorder traversal, to redraw the binary tree of Figure 7.20.

C-7.19 Design an algorithm for drawing general trees that generalizes the inorder traversal approach for drawing binary trees.

C-7.20 Consider a variation of the linked data structure for binary trees where each node object has pointers to the node objects of the children but not to the node object of the parent. Describe an implementation of the functions of a binary tree with this data structure and analyze the time complexity for these functions.

C-7.21 Design an alternative implementation of the linked data structure for binary trees using a class for nodes that specializes into subclasses for an internal node, an external node, and the root node.

C-7.22 Provide the missing housekeeping functions (destructor, copy constructor, and assignment operator) for the class LinkedBinaryTree given in Code Fragment 7.19.

C-7.23 Our linked binary tree implementation given in Code Fragment 7.19 assumes that the tree is proper. Design an alternative implementation of the linked data structure for a general (possibly improper) binary tree.

C-7.24 Let T be a tree with n nodes. Define the ***lowest common ancestor*** (LCA) between two nodes v and w as the lowest node in T that has both v and w as descendents (where we allow a node to be a descendent of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your method?

C-7.25 Let T be a tree with n nodes, and, for any node v in T, let d_v denote the depth of v in T. The ***distance*** between two nodes v and w in T is $d_v + d_w - 2d_u$, where u is the LCA u of v and w (as defined in the previous exercise). The ***diameter*** of T is the maximum distance between two nodes in T. Describe an efficient algorithm for finding the diameter of T. What is the running time of your method?

C-7.26 Suppose each node v of a binary tree T is labeled with its value $f(v)$ in a level numbering of T. Design a fast method for determining $f(u)$ for the lowest common ancestor (LCA), u, of two nodes v and w in T, given $f(v)$ and $f(w)$. You do not need to find node u, just compute its level-numbering label.

C-7.27 Justify Table 7.1, summarizing the running time of the functions of a tree represented with a linked structure, by providing, for each function, a description of its implementation, and an analysis of its running time.

C-7.28 Describe efficient implementations of the expandExternal and removeAboveExternal binary tree update functions, described in Section 7.3.4, for the case when the binary tree is implemented using a vector S, where S is realized using an expandable array. Your functions should work even for null external nodes, assuming we represent such a node as a wrapper object storing an index to an empty or nonexistent cell in S. What are the worst-case running times of these functions? What is the running time of removeAboveExternal if the internal node removed has only external node children?

C-7.29 Describe a nonrecursive method for evaluating a binary tree representing an arithmetic expression.

C-7.30 Let T be a binary tree with n nodes. Define a ***Roman node*** to be a node v in T, such that the number of descendents in v's left subtree differ from the number of descendants in v's right subtree by at most 5. Describe a linear-time method for finding each node v of T, such that v is not a Roman node, but all of v descendants are Roman nodes.

C-7.31 Let T' be the binary tree representing a tree T. (See Section 7.3.8.)

a. Is a preorder traversal of T' equivalent to a preorder traversal of T?
b. Is a postorder traversal of T' equivalent to a postorder traversal of T?
c. Is an inorder traversal of T' equivalent to some well-structured traversal of T?

C-7.32 Describe a nonrecursive method for performing an Euler tour traversal of a binary tree that runs in linear time and does not use a stack.

(Hint: You can tell which visit action to perform at a node by taking note of where you are coming from.)

C-7.33 Describe, in pseudo-code, a nonrecursive method for performing an in-order traversal of a binary tree in linear time.

(Hint: Use a stack.)

C-7.34 Let T be a binary tree with n nodes (T may or may not be realized with a vector). Give a linear-time method that uses the functions of the BinaryTree interface to traverse the nodes of T by increasing values of the level numbering function f given in Section 7.3.5. This traversal is known as the ***level order traversal***.

(Hint: Use a queue.)

C-7.35 The ***path length*** of a tree T is the sum of the depths of all the nodes in T. Describe a linear-time method for computing the path length of a tree T (which is not necessarily binary).

C-7.36 Define the ***internal path length***, $I(T)$, of a tree T, to be the sum of the depths of all the internal nodes in T. Likewise, define the ***external path length***, $E(T)$, of a tree T, to be the sum of the depths of all the external nodes in T. Show that if T is a binary tree with n internal nodes, then $E(T) = I(T) + 2n$.

(Hint: Use the fact that we can build T from a single root node via a series of n expandExternal operations.)

Projects

P-7.1 Write a program that takes as input a rooted tree T and a node v of T and converts T to another tree with the same set of node adjacencies but now rooted at v.

P-7.2 Give a fully generic implementation of the class LinkedBinaryTree using class templates and taking into account error conditions.

P-7.3 Implement the binary tree ADT using a vector.

P-7.4 Implement the binary tree ADT using a linked structure.

P-7.5 Write a program that draws a binary tree.

P-7.6 Write a program that draws a general tree.

P-7.7 Write a program that can input and display a person's family tree.

P-7.8 Implement the binary tree representation of the tree ADT. You may reuse the LinkedBinaryTree implementation of a binary tree.

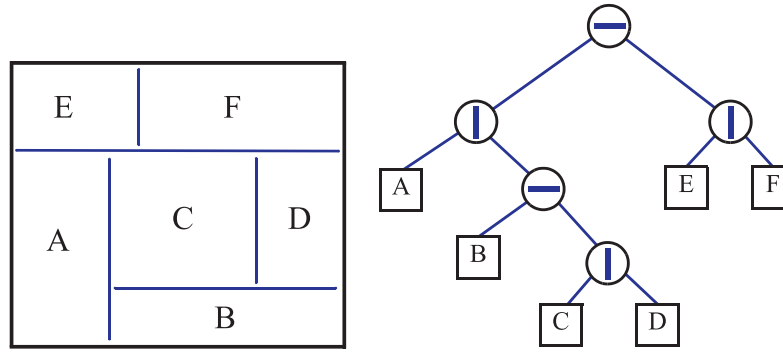
P-7.9 A ***slicing floorplan*** is a decomposition of a rectangle with horizontal and vertical sides using horizontal and vertical ***cuts*** (see Figure 7.24(a)). A slicing floorplan can be represented by a binary tree, called a ***slicing tree***, whose internal nodes represent the cuts, and whose external nodes represent the ***basic rectangles*** into which the floorplan is decomposed by the cuts (see Figure 7.24(b)). The ***compaction problem*** is defined as follows. Assume that each basic rectangle of a slicing floorplan is assigned a minimum width w and a minimum height h. The compaction problem is to find the smallest possible height and width for each rectangle of the slicing floorplan that is compatible with the minimum dimensions of the basic rectangles. Namely, this problem requires the assignment of values $h(v)$ and $w(v)$ to each node v of the slicing tree, such that

$$\mathsf{w}(v) = \begin{cases} w & \text{if } v \text{ is an external node whose basic rectangle has minimum width } w \\ \max(\mathsf{w}(w), \mathsf{w}(z)) & \text{if } v \text{ is an internal node associated with a horizontal cut with left child } w \text{ and right child } z \\ \mathsf{w}(w) + \mathsf{w}(z) & \text{if } v \text{ is an internal node associated with a vertical cut with left child } w \text{ and right child } z \end{cases}$$

$$\mathsf{h}(v) = \begin{cases} h & \text{if } v \text{ is an external node whose basic rectangle has minimum height } h \\ \mathsf{h}(w) + \mathsf{h}(z) & \text{if } v \text{ is an internal node associated with a horizontal cut with left child } w \text{ and right child } z \\ \max(\mathsf{h}(w), \mathsf{h}(z)) & \text{if } v \text{ is an internal node associated with a vertical cut with left child } w \text{ and right child } z \end{cases}$$

Design a data structure for slicing floorplans that supports the following operations:

- Create a floorplan consisting of a single basic rectangle
- Decompose a basic rectangle by means of a horizontal cut
- Decompose a basic rectangle by means of a vertical cut
- Assign minimum height and width to a basic rectangle
- Draw the slicing tree associated with the floorplan
- Compact the floorplan
- Draw the compacted floorplan



(a) (b)

Figure 7.24: (a) Slicing floorplan; (b) slicing tree associated with the floorplan.

P-7.10 Write a program that takes, as input, a fully parenthesized, arithmetic expression and converts it to a binary expression tree. Your program should display the tree in some way and also print the value associated with the root. For an additional challenge, allow for the leaves to store variables of the form x_1, x_2, x_3, and so on, which are initially 0 and which can be updated interactively by your program, with the corresponding update in the printed value of the root of the expression tree.

P-7.11 Write a program that can play Tic-Tac-Toe effectively. (See Section 3.1.3.) To do this, you will need to create a ***game tree*** T, which is a tree where each node corresponds to a ***game configuration***, which, in this case, is a representation of the tic-tac-toe board. The root node corresponds to the initial configuration. For each internal node v in T, the children of v correspond to the game states we can reach from v's game state in a single legal move for the appropriate player, A (the first player) or B (the second player). Nodes at even depths correspond to moves for A and nodes at odd depths correspond to moves for B. External nodes are either final game states or are at a depth beyond which we don't want to explore. We score each external node with a value that indicates how good this state is for player A. In large games, like chess, we have to use a heuristic scoring function, but for small games, like tic-tac-toe, we can construct the entire game tree and score external nodes as $+1$, 0, -1, indicating whether player A has a win, draw, or lose in that configuration. A good algorithm for choosing moves is ***minimax***. In this algorithm, we assign a score to each internal node v in T, such that if v represents A's turn, we compute v's score as the maximum of the scores of v's children (which corresponds to A's optimal play from v). If an internal node v represents B's turn, then we compute v's score as the minimum of the scores of v's children (which corresponds to B's optimal play from v).

Chapter Notes

Our use of the position abstraction derives from the ***position*** and ***node*** abstractions introduced by Aho, Hopcroft, and Ullman [5]. Discussions of the classic preorder, inorder, and postorder tree traversal methods can be found in Knuth's *Fundamental Algorithms* book [56]. The Euler tour traversal technique comes from the parallel algorithms community, as it is introduced by Tarjan and Vishkin [93] and is discussed by JáJá [49] and by Karp and Ramachandran [53]. The algorithm for drawing a tree is generally considered to be a part of the "folklore" of graph drawing algorithms. The reader interested in graph drawing is referred to the book by Di Battista, Eades, Tamassia and Tollis [28] and the survey by Tamassia and Liotta [92]. The puzzler in Exercise R-7.11 was communicated by Micha Sharir.