CHAPTER 3

C++ Trees

LESSON 3.1

Trees

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. Example of binary trees shown below.

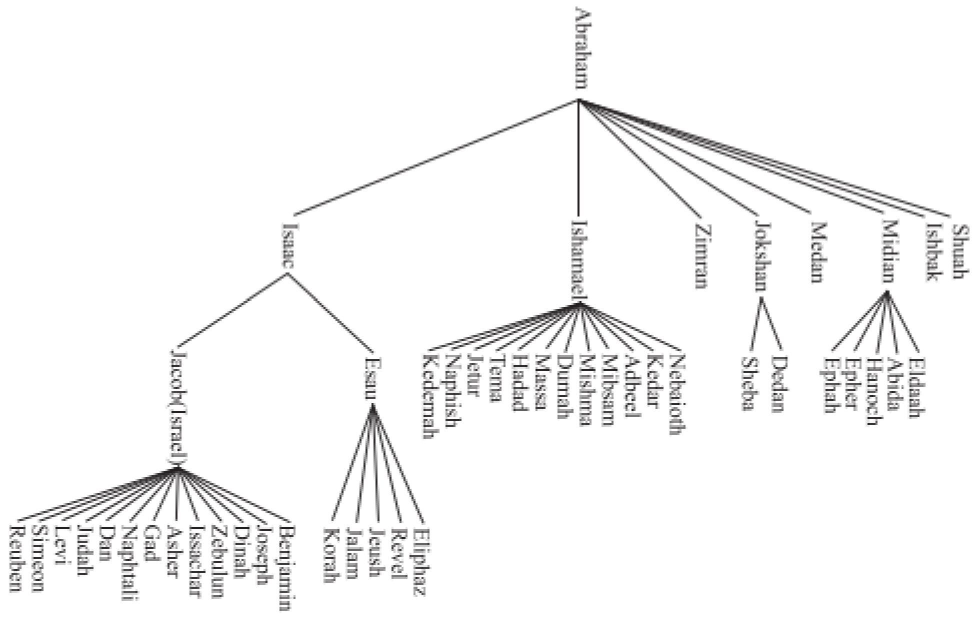

Figure 3.1.1 A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

Tree Definition 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 3.1.2) 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).

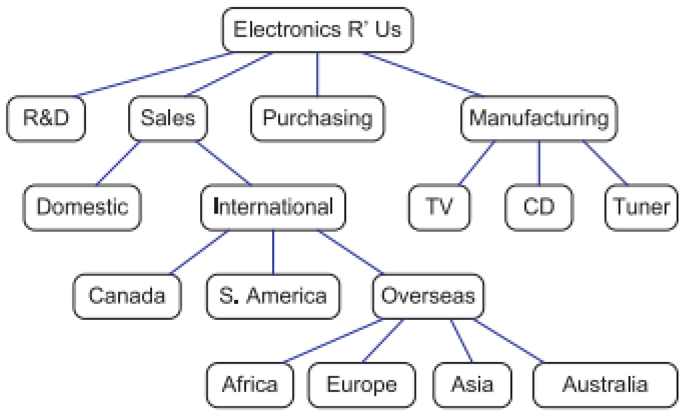

Figure 3.1.2 A tree with 17 nodes representing the organizational structure of a fictitious corporation. Electronics R'Us is stored at the root. The children of the root store R&D, Sales, Purchasing, and Manufacturing. The internal nodes store Sales, International, Overseas, Electronics R'Us, and Manufacturing.

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.

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: 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 3.1.3.) 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 "/."

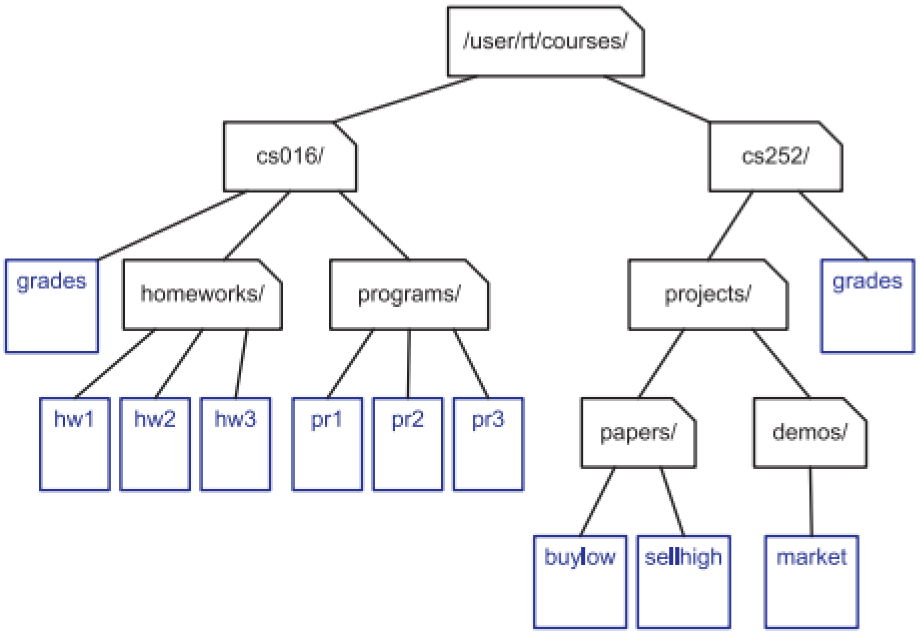

Figure 3.1.3 Tree representing a portion of a file system.

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 3.1.3, 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 3.1.3, 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 3.1.3 contains the path (cs252/, projects/, demos/, market).

> *Example*: 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*: 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.

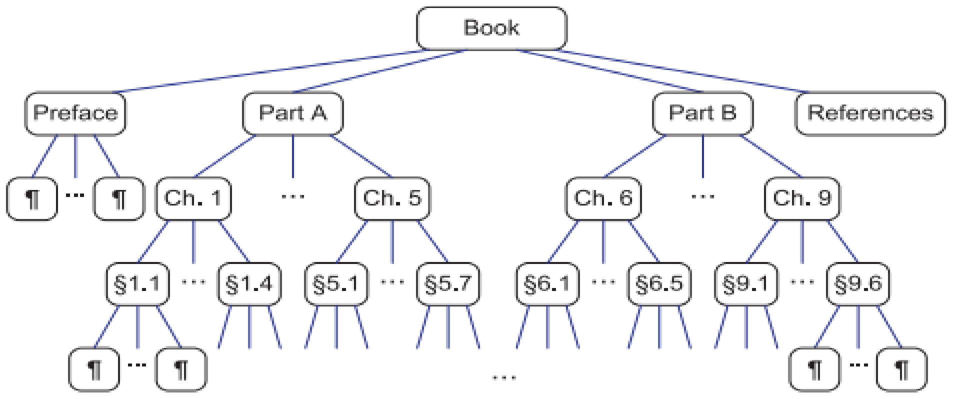

Figure 3.1.4 An ordered tree associated with a book

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.

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.

LESSON 3.2

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: 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 3.2.1 illustrates a decision tree that provides recommendations to a prospective investor

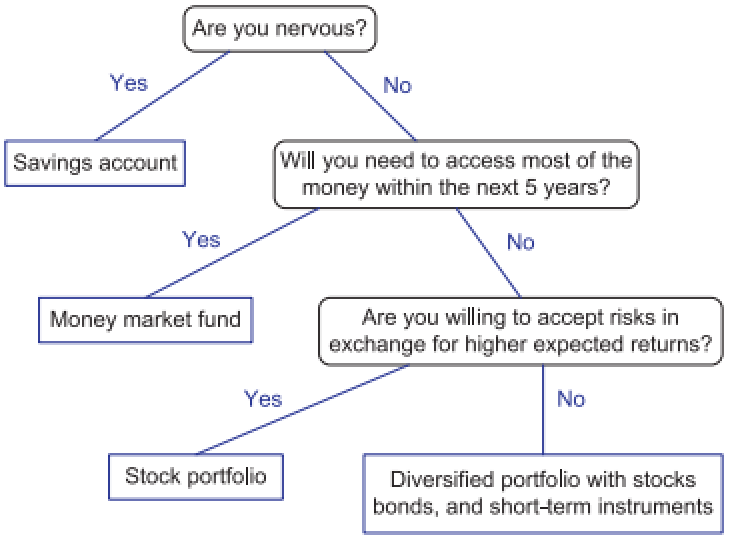
Are you nervous?
Yes
No
Savings account
Will you need to access most of the money within the next 5 years?
Yes
No
Money market fund
Are you willing to accept risks in exchange for higher expected returns?
Yes
No
Stock portfolio
Diversified portfolio with stocks bonds, and short-term instruments

Figure 3.2.1 A decision tree providing investment advice.

Example: 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 +, −, ×, and /. (See Figure 3.2.2) 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 +, −, ×, 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.

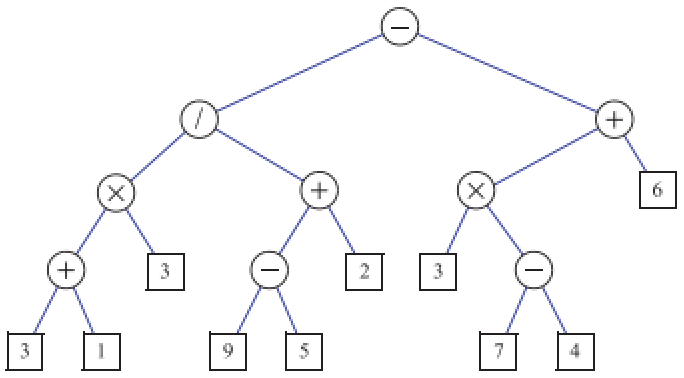

Figure 3.2.2 A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Recursive Binary Tree

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 in the next topic.

The Binary Tree ADT

In this topic, 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.

C++ Binary Tree Interface

Let us present an informal C++ interface for the binary tree ADT. We begin in the Fragment Code by presenting an informal C++ interface for the class Position, which represents a position in a tree. 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?
};
```

- 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. In our code examples, we assume that PositionList is implemented as an STL list of objects of type Position.

Implementing Binary Trees

The previous sections described various operations that can be performed on a binary tree, as well as the functions to implement these operations. This section describes binary trees as an abstract data type (ADT). Before designing the class to implement a binary tree as an ADT, let us list various operations that are typically performed on a binary tree:

- Determine whether the binary tree is empty.
- Search the binary tree for a particular item.
- Insert an item in the binary tree.

- Delete an item from the binary tree.
- Find the height of the binary tree.
- Find the number of nodes in the binary tree.
- Find the number of leaves in the binary tree.
- Traverse the binary tree.
- Copy the binary tree.

The item search, insertion, and deletion operations all require the binary tree to be traversed. However, because the nodes of a binary tree are in no particular order, these algorithms are not very efficient on arbitrary binary trees. That is, no criteria exist to guide the search on these binary trees, as we will see in the next section. Therefore, we will discuss these algorithms when we discuss special binary trees. Other than for the search, insertion, and deletion operations, the following class defines binary trees as an ADT. The definition of the node is the same as before. However, for the sake of completeness and easy reference, we give the definition of the node followed by the definition of the class.

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 3.2.3) 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).

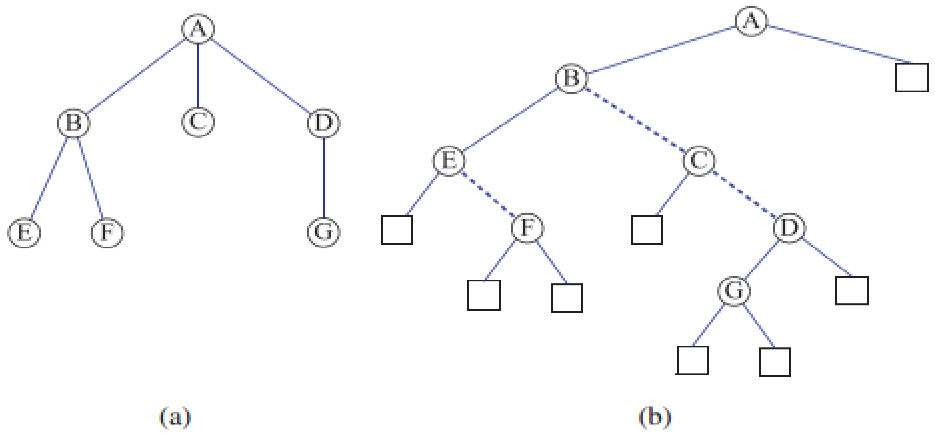

Figure 3.2.3 Representation of a tree by means of a binary tree: (a) tree T; (b) binary tree T′ associated with T. The dashed edges connect nodes of T′ associated with sibling nodes of T.

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 {v1,v2,...,vk} in T with parent v and replaces it with a chain of right children rooted at v1, which then becomes the left child of v.

References:

"Data Structures and Algorithm in C++" 2nd Edition , Michael T. Goodrich, Roberto Tamassia and David Mount

"Fundamentals of Data Structures in C++", E. Horowits, S. Sahni and D. Mehta

"Principles of Data Structures using C and C++", New Age, Vinu V Das, MES College of Engineering, Kuttipuran, Kerala India

"Data Structures Using C++", 2nd Edition, D.S Malik

"Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A Shaffer, Department of Computer Science, Virginia Tech Blacksburg, VA 24061

"Practical Introduction to Data Structures and Algorithm Analysis", C++ Edition, 2nd Edition by Cli!ord A. Sha!er. Prentice Hall, 2000

"Foundations of Multidimensional and Metric Data Structures", by Hanan Samet Morgan Kaufmann, 2006.

"Introduction to Data Structures", Tim French, CITS2200

"Handouts on Fundamenta Data Structures", IS103 Computational Thinking, SMU Singapore Management University

"C++ Language Tutorial", Juan Soulie, http://www.cplusplus.com/doc/tutorial

"Open Data Structures in C++", Edition 0.1G, Pat Morin