2.3 Trees

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 2.15.) 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).

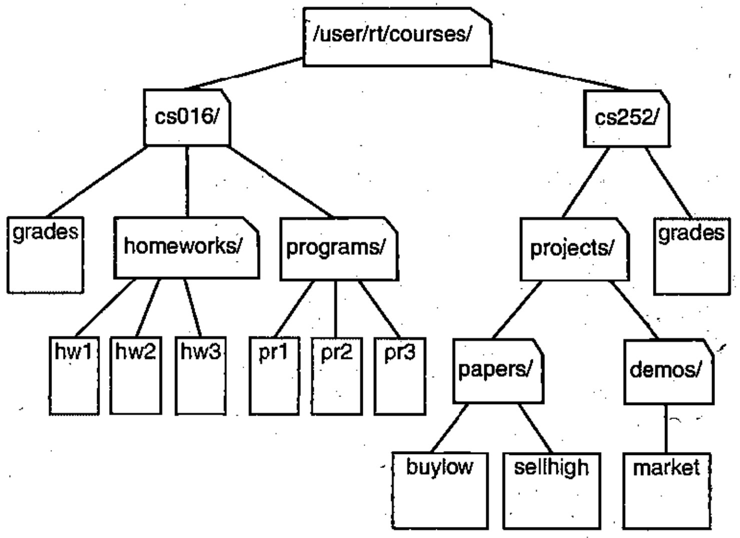

Figure 2.15: A tree representing a portion of a file system.

A ***tree*** T is a set of ***nodes*** storing elements in a ***parent-child*** relationship with the following properties:

- T has a special node r, called the ***root*** of T.
- Each node v of T different from r has a ***parent*** node u.

Note that according to the above definition, a tree cannot be empty, since it must have at least one node, the root. One could also allow the definition to include empty trees, but we adopt the convention that a tree always has a root so as to keep our presentation simple and to avoid having to always deal with the special case of an empty tree in our algorithms.

If node u is the parent of node v, then we say that v is a ***child*** of u. Two nodes that are children of the same parent are ***siblings***. A node is ***external*** if it has no children and it is ***internal*** if it has one or more children. External nodes are also known as ***leaves***. The ***subtree*** of T ***rooted*** at a node v is the tree consisting of all the descendents of v in T (including v itself). An ***ancestor*** of a node is either the node itself or an ancestor of the parent of the node. Conversely, we say that a node v is a ***descendent*** of a node u if u is an ancestor of v.

Example 2.2: *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 2.15.) More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX/Linux operating system, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/." It is the ancestor of all directories and files in a UNIX/Linux file system.*

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. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in a sequence or iterator in the correct order.

Example 2.3: *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 2.16.) 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.*

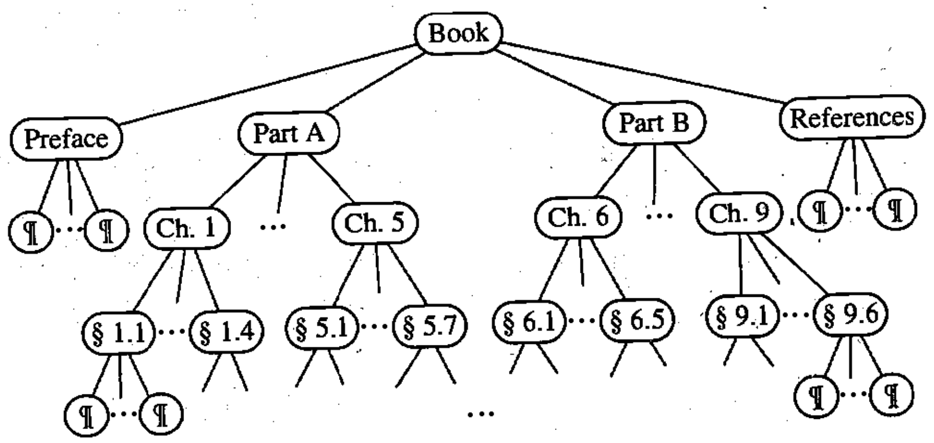

Figure 2.16: A tree associated with a book.

A ***binary tree*** is an ordered tree in which every node has at most two children. A binary tree is ***proper*** if each internal node has two children. For each internal node in a binary tree, we label each child as either being a ***left child*** or a ***right child***. These children are ordered so that a left child comes before a right child. The subtree rooted at a left or right child of an internal node v is called a ***left subtree*** or ***right subtree***, respectively, of v. We make the convention in this book that, unless otherwise stated, every binary tree is a proper binary tree. Of course, even an improper binary tree is still a general tree, with the property that each internal node has at most two children. Binary trees have a number of useful applications, including the following.

Example 2.4: *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 2.17.) 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.*

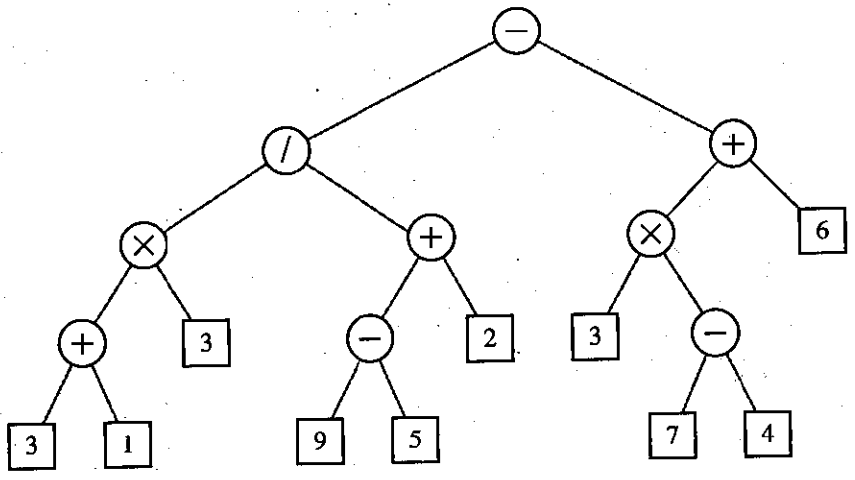

Figure 2.17: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

2.3.1 The Tree Abstract Data Type

The tree ADT stores elements at positions, which, as with positions in a list, are defined relative to neighboring positions. The ***positions*** in a tree are its ***nodes***, and neighboring positions satisfy the parent-child relationships that define a valid tree. Therefore we use the terms "position" and "node" interchangeably for trees. As with a list position, a position object for a tree supports the element() method, which returns the object at this position. The real power of node positions in a tree, however, comes from the following ***accessor methods*** of the tree ADT:

root():	Return the root of the tree.
parent(v):	Return the parent of node v; an error occurs if v is root.
children(v):	Return an iterator of the children of node v.

If a tree T is ordered, then the iterator children(v) provides access to the children of v in order. If v is an external node, then children(v) is an empty iterator.

In addition, we also include the following ***query methods***:

isInternal(v):	Test whether node v is internal.
isExternal(v):	Test whether node v is external.
isRoot(v):	Test whether node v is the root.

There are also a number of methods a tree should support that are not necessarily related to its tree structure. Such ***generic methods*** include the following:

size():	Return the number of nodes in the tree.
elements():	Return an iterator of all the elements stored at nodes of the tree.
positions():	Return an iterator of all the nodes of the tree.
swapElements(v, w):	Swap the elements stored at the nodes v and w.
replaceElement(v, e):	Replace with e and return the element stored at node v.

We do not define any specialized update methods for a tree here. Instead, let us reserve the potential to define different tree update methods in conjunction with specific tree applications.

2.3.2 Tree Traversal

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

Assumptions

In order to analyze the running time of tree-based algorithms, we make the following assumptions on the running times of the methods of the tree ADT.

- The accessor methods root() and parent(v) take $O(1)$ time.
- The query methods isInternal(v), isExternal(v), and isRoot(v) take $O(1)$ time, as well.
- The accessor method children(v) takes $O(c_v)$ time, where c_v is the number of children of v.
- The generic methods swapElements(v, w) and replaceElement(v, e) take $O(1)$ time.
- The generic methods elements() and positions(), which return iterators, take $O(n)$ time, where n is the number of nodes in the tree.
- For the iterators returned by methods elements(), positions(), and children(v), the methods hasNext(), nextObject() or nextPosition() take $O(1)$ time.

In Section 2.3.4, we will present data structures for trees that satisfy the above assumptions. Before we describe how to implement the tree ADT using a concrete data structure, however, let us describe how we can use the methods of the tree ADT to solve some interesting problems for trees.

Depth and Height

Let v be a node of a tree T. The ***depth*** of v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also be recursively defined as follows:

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

Based on the above definition, the recursive algorithm depth, shown in Algorithm 2.18, computes the depth of a node v of T by calling itself recursively on the parent of v, and adding 1 to the value returned.

```
Algorithm depth($T,v$):
  if $T$.isRoot($v$) then
    return 0
  else
    return $1+$depth($T,T$.parent($v$))
```

Algorithm 2.18: Algorithm depth for computing the depth of a node v in a tree T.

The running time of algorithm depth(T,v) is $O(1+d_v)$, where d_v denotes the depth of the node v in the tree T, because the algorithm performs a constant-time recursive step for each ancestor of v. 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 nearly 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_v, since this will often be much smaller than n.

The ***height*** of a tree T is equal to the maximum depth of an external node of T While this definition is correct, it does not lead to an efficient algorithm. Indeed, if we were to apply the above depth-finding algorithm to each node in the tree T, we would derive an $O(n^2)$-time algorithm to compute the height of T. We can do much better, however, using the following recursive definition of the ***height*** of a node v in a tree T:

- If v is an external node, then the height of v is 0.
- Otherwise, the height of v is one plus the maximum height of a ehild of v.

The ***height*** of a tree T is the height of the root of T.

Algorithm height, shown in Algorithm 2.19 computes the height of tree T in an efficient manner by using the above recursive definition of height. The algorithm is expressed by a recursive method height(T,v) that computes the height of the subtree of T rooted at a node v. The height of tree T is obtained by calling height$(T,T.\text{root}())$.

```
Algorithm height(T, v):
  if T.isExternal(v) then
    return 0
  else
    h = 0
    for each w ∈ T.children(v) do
      h = max(h, height(T, w))
    return 1 + h
```

Algorithm 2.19: Algorithm height for computing the height of the subtree of tree T rooted at a node v.

The height algorithm is recursive, and if it is initially called on the root of T, it will eventually be called once on each node of T. Thus, we can determine the running time of this method by an amortization argument where we first determine the amount of time spent at each node (on the nonrecursive part), and then sum this time bound over all the nodes. The computation of an iterator children(v) takes $O(c_v)$ time, where c_v denotes the number of children of node v. Also, the **while** loop has c_v iterations, and each iteration of the loop takes $O(1)$ time plus the time for the recursive call on a child of v. Thus, algorithm height spends $O(1+c_v)$ time at each node v, and its running time is $O(\sum_{v\in T}(1+c_v))$. In order to complete the analysis, we make use of the following property.

Theorem 2.5: *Let T be a tree with n nodes, and let c_v denote the number of children of a node v of T. Then*

$$\sum_{v\in T} c_v = n-1.$$

Proof: Each node of T, with the exception of the root, is a child of another node, and thus contributes one unit to the summation $\sum_{v\in T} c_v$. ■

By Theorem 2.5, the running time of Algorithm height when called on the root of T is $O(n)$, where n is the number of nodes of T.

A ***traversal*** of a tree T is a systematic way of accessing, or "visiting," all the nodes of T. We next present basic traversal schemes for trees, called preorder and postorder traversals.

Preorder 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 v depends on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for v. The pseudo-code for the preorder traversal of the subtree rooted at a node v is shown in Algorithm 2.20. We initially call this routine as preorder(T,T.root()).

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

Algorithm 2.20: Algorithm preorder.

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.

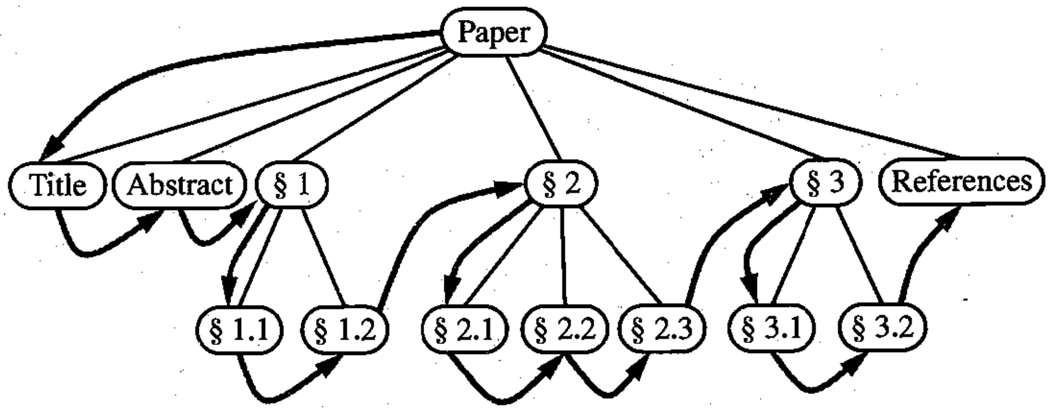

Figure 2.21: Preorder traversal of an ordered tree.

Example 2.6: *The preorder traversal of the tree associated with a document, as in Example 2.3, 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 2.21.)*

The analysis of preorder traversal is actually similar to that of algorithm height given above. At each node v, the nonrecursive part of the preorder traversal algorithm requires time $O(1+c_v)$, where c_v is the number of children of v. Thus, by Theorem 2.5, the overall running time of the preorder traversal of T is $O(n)$.

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 v. Still, as with the preorder traversal, if the tree is ordered, we make recursive calls for the children of a node v according to their specified order. Pseudo-code for the postorder traversal is given in Algorithm 2.22.

```
Algorithm postorder($T, v$):
  for each child $w$ of $v$ do
    recursively traverse the subtree rooted at $w$ by calling postorder($T, w$)
  perform the "visit" action for node $v$
```

Algorithm 2.22: Method postorder.

The name of the postorder traversal comes from the fact that this traversal method will visit a node v after it has visited all the other nodes in the subtree rooted at v. (See Figure 2.23.)

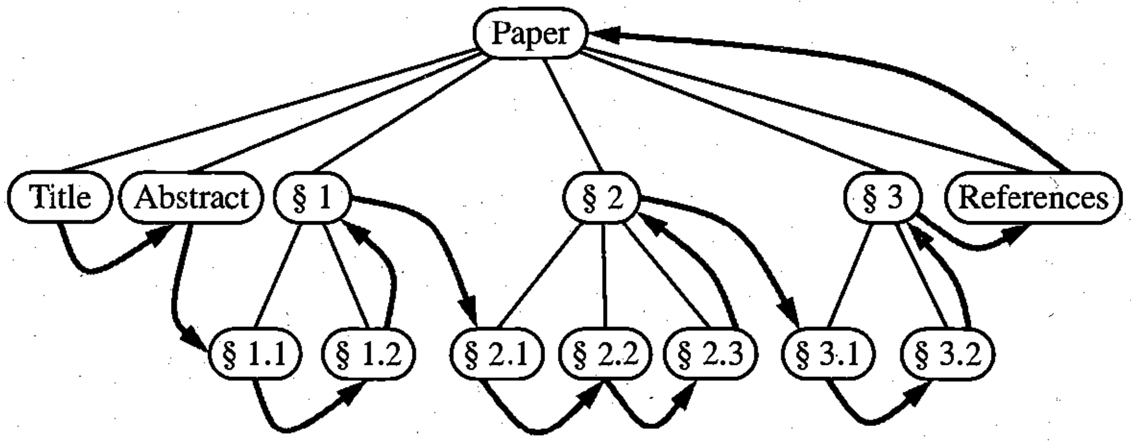

Figure 2.23: Postorder traversal of the ordered tree of Figure 2.21.

The analysis of the running time of a postorder traversal is analogous to that of a preorder traversal. 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.

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

Example 2.7: *Consider a file system tree T, where external nodes represent files and internal nodes represent directories (Example 2.2). Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of:*

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

(See Figure 2.24.) This computation can be done with a postorder traversal of tree T. After the subtrees of an internal node v have been traversed, we compute the space used by v by adding the sizes of the directory v itself and of the files contained in v, to the space used by each internal child of v, which was computed by the recursive postorder traversals of the children of v.

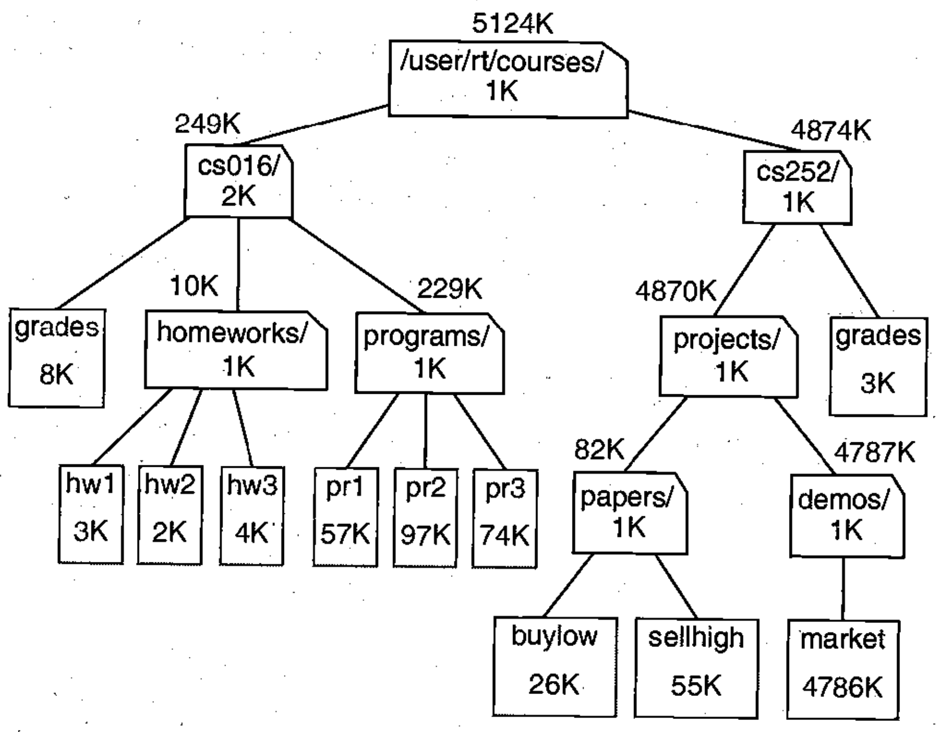

Figure 2.24: The tree of Figure 2.15 representing a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.

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 could be implemented, for example, using a queue, whereas the preorder and postorder traversals use a stack (this stack is implicit in our use of recursion to describe these methods, 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.

2.3.3 Binary Trees

One kind of tree that is of particular interest is the binary tree. As we mentioned in Section 2.3.1, a proper ***binary tree*** is an ordered tree in which each internal node has exactly two children. We make the convention that, unless otherwise stated, binary trees are assumed to be proper. Note that our convention for binary trees is made without loss of generality, for we can easily convert any improper binary tree into a proper one, as we explore in Exercise C-2.14. Even without such a conversion, we can consider an improper binary tree as proper, simply by viewing missing external nodes as "null nodes" or place holders that still count as nodes.

The Binary Tree Abstract Data Type

As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:

leftChild(v):	Return the left child of v; an error condition occurs if v is an external node.
rightChild(v):	Return the right child of v; an error condition occurs if v is an external node.
sibling(v):	Return the sibling of node v; an error condition occurs if v is the root.

Note that these methods must have additional error conditions if we are dealing with improper binary trees. For example, in an improper binary tree, an internal node may not have the left child or right child. We do not include here any methods for updating a binary tree, for such methods can be created as required in the context of specific needs.

Properties of Binary Trees

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 2.25.) In general, level d has at most 2^d nodes, which implies the following theorem (whose proof is left to Exercise R-2.4).

Theorem 2.8: *Let T be a (proper) binary tree with n nodes, and let h denote the height of T. Then T has the following properties:*

1. *The number of external nodes in T is at least $h+1$ and at most 2^h.*
2. *The number of internal nodes in T is at least h and at most 2^h-1.*
3. *The total number of nodes in T is at least $2h+1$ and at most $2^{h+1}-1$.*
4. *The height of T is at least $\log(n+1)-1$ and at most $(n-1)/2$, that is $\log(n+1)-1 \leq h \leq (n-1)/2$.*

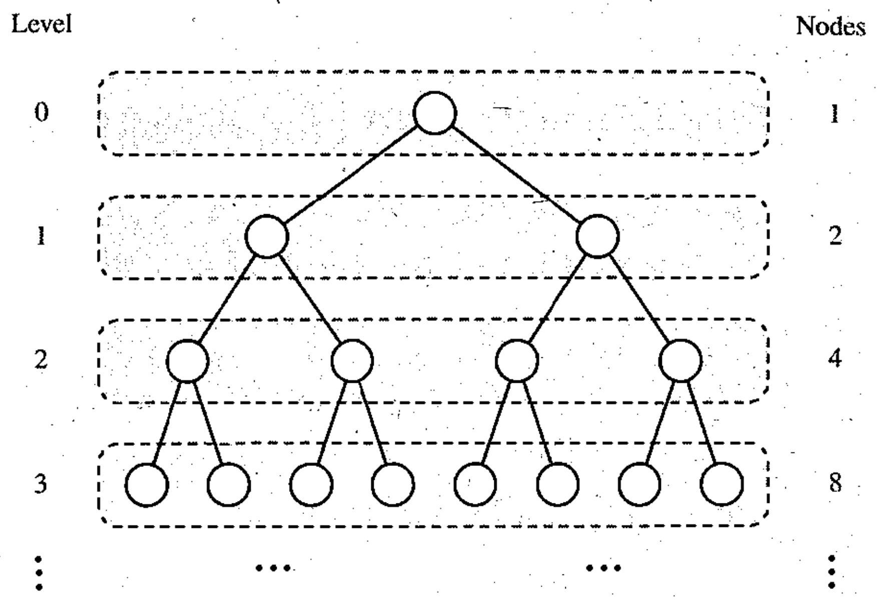

Figure 2.25: Maximum number of nodes in the levels of a binary tree.

In addition, we also have the following.

Theorem 2.9: *In a proper) binary tree T, the number of external nodes is* 1 *more than the number of internal nodes.*

Proof: The proof is by induction. If T itself has only one node v, then v is external, and the proposition clearly holds. Otherwise, we remove from T an (arbitrary) external node w and its parent v, which is an internal node. If v has a parent u, then we reconnect u with the former sibling z of w, as shown in Figure 2.26. This operation, which we call removeAboveExternal(w), removes one internal node and one external node, and it leaves the tree being a proper binary tree. Thus, by the inductive hypothesis, the number of external nodes in this tree is one more than the number of internal nodes. Since we removed one internal and one external node to reduce T to this smaller tree, this same property must hold for T. ■

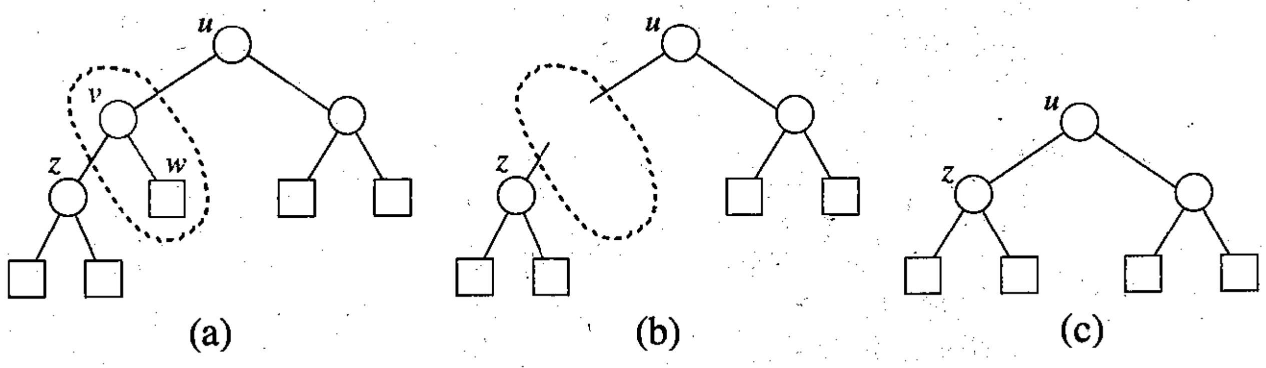

Figure 2.26: Operation removeAboveExternal(w), which removes an external node and its parent node, used in the justification of Theorem 2.9.

Note that the above relationship does not hold, in general, for nonbinary trees.

In subsequent chapters, we explore some important applications of the above facts. Before we can discuss such applications, however, we should first understand more about how binary trees are traversed and represented.

Traversals of a Binary Tree

As with general trees, computations performed on binary trees often involve tree traversals. In this section, we present binary tree traversal algorithms expressed using the binary tree ADT methods. As for running times, in addition to the assumptions on the running time of the tree ADT methods made in Section 2.3.2, we assume that, for a binary tree, method children(v) takes $O(1)$ time, because each node has either zero or two children. Likewise, we assume that methods leftChild(v), rightChild(v), and sibling(v) each take $O(1)$ time.

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 2.20) can be applied to any binary tree. We can simplify the pseudo-code in the case of a binary tree traversal, however, as we show in Algorithm 2.27.

```
Algorithm binaryPreorder($T,v$):
  perform the "visit" action for node $v$
  if $v$ is an internal node then
    binaryPreorder($T,T$.leftChild($v$))      {recursively traverse left subtree}
    binaryPreorder($T,T$.rightChild($v$))     {recursively traverse right subtree}
```

Algorithm 2.27: Algorithm binaryPreorder that performs the preorder traversal of the subtree of a binary tree T rooted at node v.

Postorder Traversal of a Binary Tree

Analogously, the postorder traversal for general trees (Algorithm 2.22) can be specialized for binary trees, as shown in Algorithm 2.28.

```
Algorithm binaryPostorder($T,v$):
  if $v$ is an internal node then
    binaryPostorder($T,T$.leftChild($v$))      {recursively traverse left subtree}
    binaryPostorder($T,T$.rightChild($v$))     {recursively traverse right subtree}
  perform the "visit" action for the node $v$
```

Algorithm 2.28: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node v.

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.

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, as shown in in Algorithm 2.29.

```
Algorithm inorder($T,v$):
  if $v$ is an internal node then
    inorder($T,T$.leftChild($v$))          {recursively traverse left subtree}
  perform the "visit" action for node $v$
  if $v$ is an internal node then
    inorder($T,T$.rightChild($v$))         {recursively traverse right subtree}
```

Algorithm 2.29: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node v.

The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right." Indeed, for every node v, the inorder traversal visits v after all the nodes in the left subtree of v and before all the nodes in the right subtree of v. (See Figure 2.30.)

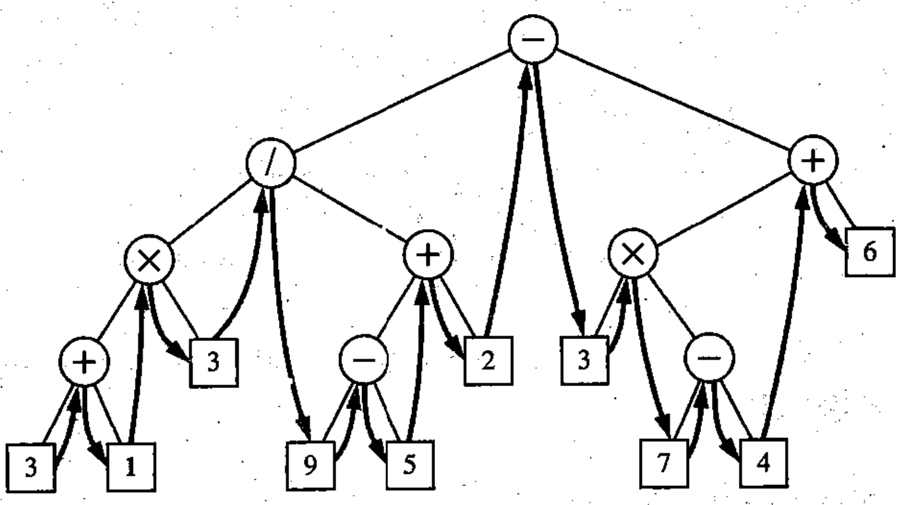

Figure 2.30: Inorder traversal of a binary tree.

A Unified Tree Traversal Framework

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 design pattern, 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

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 2.31.) Each node v of T is encountered three times by the Euler tour:

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

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

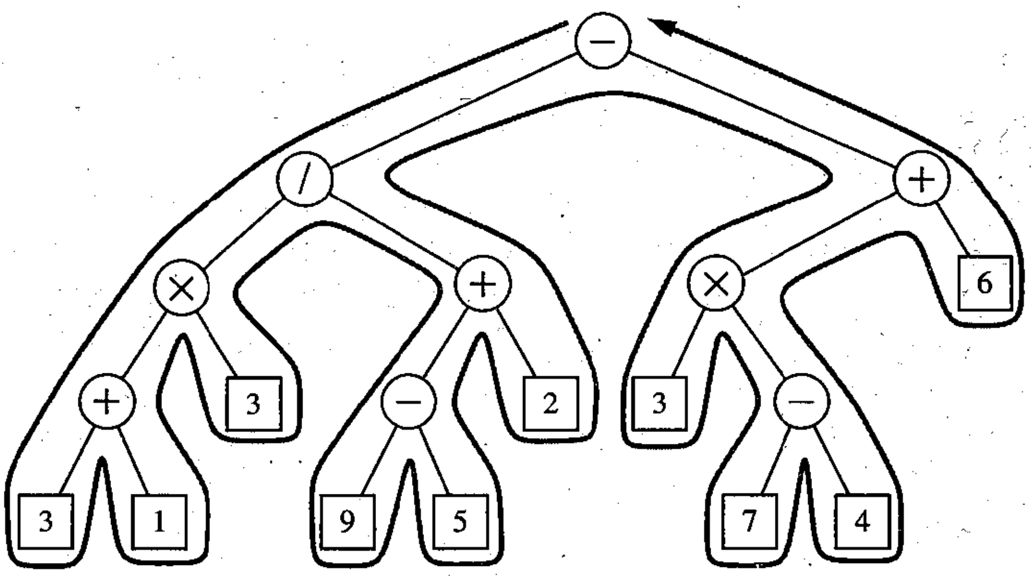

Figure 2.31: Euler tour traversal of a binary tree.

We give pseudo-code for the Euler tour of the subtree rooted at a node v in Algorithm 2.32.

```
Algorithm eulerTour($T, v$):
  perform the action for visiting node $v$ on the left
  if $v$ is an internal node then
    recursively tour the left subtree of $v$ by calling eulerTour($T, T$.leftChild($v$))
  perform the action for visiting node $v$ from below
  if $v$ is an internal node then
    recursively tour the right subtree of $v$ by calling eulerTour($T, T$.rightChild($v$))
  perform the action for visiting node $v$ on the right
```

Algorithm 2.32: Algorithm eulerTour for computing the Euler tour traversal of the subtree of a binary tree T rooted at a node v.

The preorder traversal of a binary tree is equivalent to an Euler tour traversal such that 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 such that 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 descendents of each node v 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 descendents of a node v, we compute the difference between the values of the counter when v is visited on the left and when it is visited on the right, and add 1. This simple rule gives us the number of descendents of v, because each node in the subtree rooted at v is counted between v's visit on the left and v's visit on the right. Therefore, we have an $O(n)$-time method for computing the number of descendents of each node in T.

The running time of the Euler tour traversal is easy to analyze, assuming 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 2.4). The method printExpression, shown in Algorithm 2.33, 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($T, v$):
  if $T$.isExternal($v$) then
    print the value stored at $v$
  else
    print "("
    printExpression($T, T$.leftChild($v$))
    print the operator stored at $v$
    printExpression($T, T$.rightChild($v$))
    print ")"
```

Algorithm 2.33: An algorithm for printing the arithmetic expression associated with the subtree of an arithmetic expression tree T rooted at v.

Having presented these pseudo-code examples, we now describe a number of efficient ways of realizing the tree abstract data type by concrete data structures, such as sequences and linked structures.

2.3.4 Data Structures for Representing Trees

In this section, we describe concrete data structures for representing trees.

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 $p(v)$ be the integer defined as follows.

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

The numbering function p is known as a ***level numbering*** of the nodes in a binary tree T, 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 2.34.)

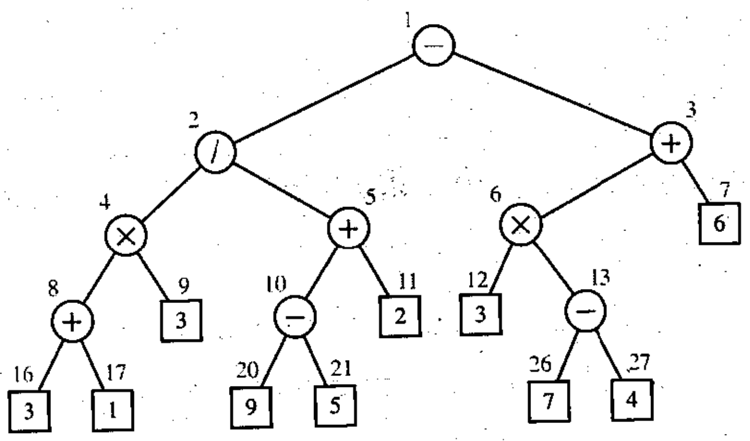

Figure 2.34: An example binary tree level numbering.

The level numbering function p 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 $p(v)$. (See Figure 2.35.) Typically, we realize the vector S by means of an extendable array. (See Section 1.5.2.) Such an implementation is simple and fast, for we can use it to easily perform the methods root, parent, leftChild, rightChild, sibling, isInternal, isExternal, and isRoot by using simple arithmetic operations on the numbers $p(v)$ associated with each node v involved in the operation. That is, each position object v is simply a "wrapper" for the index $p(v)$ into the vector S. We leave the details of such implementations as a simple exercise (R-2.7).

Let n be the number of nodes of T, and let p_M be the maximum value of $p(v)$ over all the nodes of T. Vector S has size $N = p_M + 1$ since the element of S at rank 0 is not associated with any node of T. Also, vector S will have, in general, a number of empty elements that do not refer to existing nodes of T. These empty slots could, for example, correspond to empty external nodes or even slots where descendents of such nodes would go. In fact, in the worst case, $N = 2^{(n+1)/2}$,

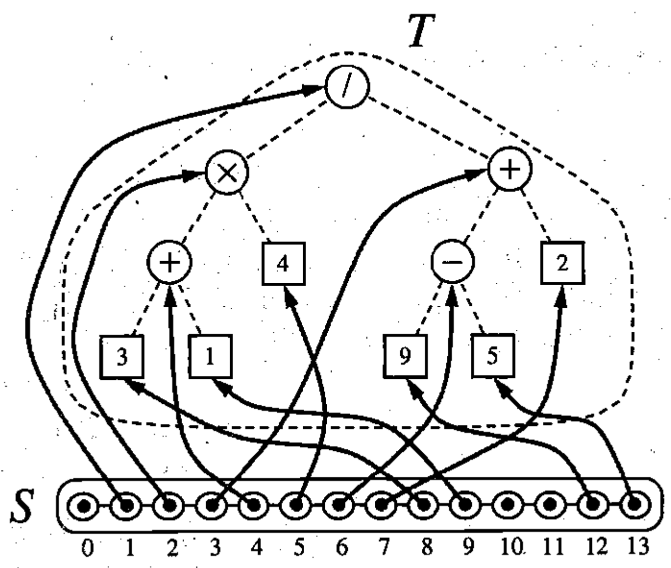

Figure 2.35: Representation of a binary tree T by means of a vector S.

the justification of which is left as an exercise (R-2.6). In Section 2.4.3, we will see a class of binary trees, called "heaps" for which $N = n+1$. Moreover, if all external nodes are empty, as will be the case in our heap implementation, then we can save additional space by not even extending the size of the vector S to include external nodes whose index is past that of the last internal node in the tree. Thus, in spite of the worst-case space usage, there are applications for which the vector 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 2.36 summarizes the running times of the methods of a binary tree implemented with a vector. In this table, we do not include any methods for updating a binary tree.

Operation	Time
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent, children	$O(1)$
leftChild, rightChild, sibling	$O(1)$
isInternal, isExternal, isRoot	$O(1)$

Table 2.36: Running times of the methods of a binary tree T implemented with a vector S, where S is realized by means of an array. We denote with n the number of nodes of T, and N denotes the size of S. Methods hasNext(), nextObject(), and nextPosition() of the iterators elements(), positions(), and children(v) take $O(1)$ time. The space usage is $O(N)$, which is $O(2^{(n+1)/2})$ in the worst case.

The vector implementation of a binary tree is fast and simple, but it can be very space inefficient if the height of the tree is large. The next data structure we discuss for representing binary trees does not have this drawback.

A Linked Structure for Binary Trees

A natural way to realize a binary tree T is to use a ***linked structure***. In this approach we represent each node v of T by an object with references to the element stored at v and the positions associated with the children and parent of v. We show a linked structure representation of a binary tree in Figure 2.37.

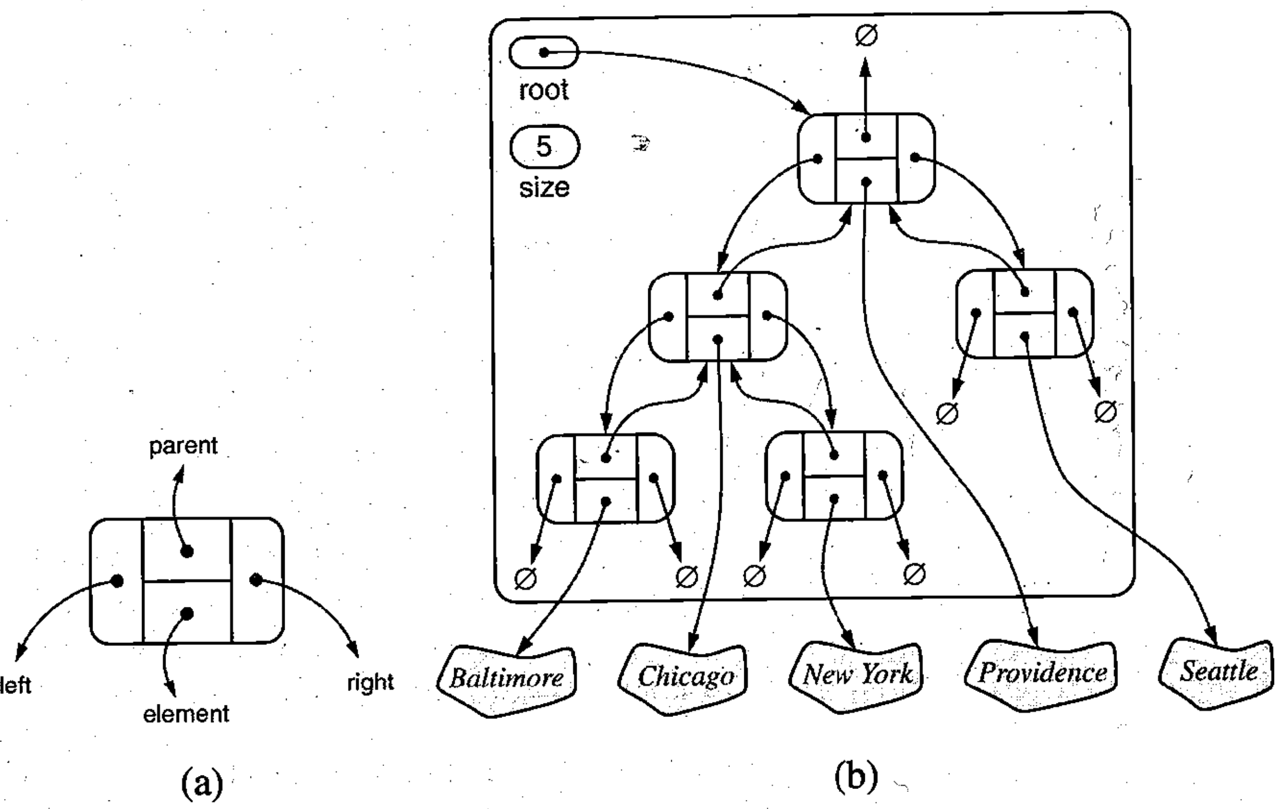

Figure 2.37: An example linked data structure for representing a binary tree: (a) object associated with a node; (b) complete data structure for a binary tree with five nodes.

If v is the root of T, then the reference to the parent node is null, and if v is an external node, then the references to the children of v are null.

If we wish to save space for cases when external nodes are empty, then we can have references to empty external nodes be null. That is, we can allow a reference from an internal node to an external node child to be null.

In addition, it is fairly straightforward to implement each of the methods size(), isEmpty(), swapElements(v, w), and replaceElement(v, e) in $O(1)$ time. Moreover, the method positions() can be implemented by performing an inorder traversal, and implementing the method elements() is similar. Thus, methods positions() and elements() take $O(n)$ time each.

Considering the space required by this data structure, note that there is a constant-sizer object for every node of tree T. Thus, the overall space requirement is $O(n)$.

A Linked Structure for General Trees

We can extend the linked structure for binary trees to represent general trees. Since there is no limit on the number of children that a node v in a general tree can have, we use a container (for example, a list or vector) to store the children of v, instead of using instance variables. This structure is schematically illustrated in Figure 2.38, assuming we implement the container for a node as a sequence.

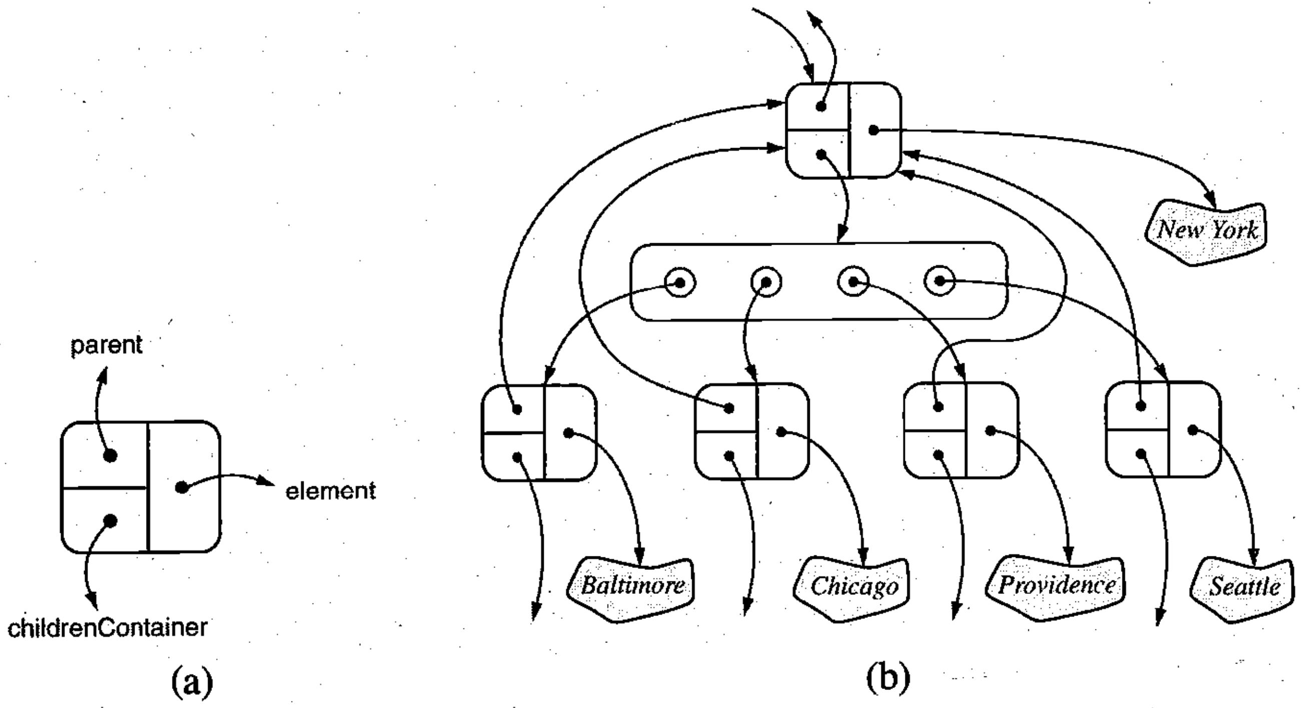

Figure 2.38: The linked structure for a tree: (a) the object associated with a node; (b) the portion of the data structure associated with a node and its children.

We note that the performance of a linked implementation of the tree ADT, shown in Table 2.39, is similar to that of the linked implementation of a binary tree. The main difference is that in the implementation of the tree ADT we use an efficient container, such as a list or vector, to store the children of each node v, instead of direct links to exactly two children.

Operation	Time
size, isEmpty	$O(1)$
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

Table 2.39: Running times of the methods of an n-node tree implemented with a linked structure. We let c_v denote the number of children of a node v.