CHAPTER 5

The Tree Data Model

There are many situations in which information has a hierarchical or nested structure like that found in family trees or organization charts. The abstraction that models hierarchical structure is called a *tree* and this data model is among the most fundamental in computer science. It is the model that underlies several programming languages, including Lisp.

Trees of various types appear in many of the chapters of this book. For instance, in Section 1.3 we saw how directories and files in some computer systems are organized into a tree structure. In Section 2.8 we used trees to show how lists are split recursively and then recombined in the merge sort algorithm. In Section 3.7 we used trees to illustrate how simple statements in a program can be combined to form progressively more complex statements.

5.1 What This Chapter Is About

The following themes form the major topics of this chapter:

The terms and concepts related to trees (Section 5.2).

The basic data structures used to represent trees in programs (Section 5.3).

Recursive algorithms that operate on the nodes of a tree (Section 5.4).

A method for making inductive proofs about trees, called structural induction, where we proceed from small trees to progressively larger ones (Section 5.5).

The binary tree, which is a variant of a tree in which nodes have two "slots" for children (Section 5.6).

The binary search tree, a data structure for maintaining a set of elements from which insertions and deletions are made (Sections 5.7 and 5.8).

The priority queue, which is a set to which elements can be added, but from which only the maximum element can be deleted at any one time. An efficient data structure, called a partially ordered tree, is introduced for implementing priority queues, and an $O(n \log n)$ algorithm, called heapsort, for sorting n elements is derived using a balanced partially ordered tree data structure, called a heap (Sections 5.9 and 5.10).

5.2 Basic Terminology

Nodes and edges

Trees are sets of points, called *nodes*, and lines, called *edges*. An edge connects two distinct nodes. To be a tree, a collection of nodes and edges must satisfy certain properties; Fig. 5.1 is an example of a tree.

Root

1. In a tree, one node is distinguished and called the *root*. The root of a tree is generally drawn at the top. In Fig. 5.1, the root is n_1.

Parent and child

2. Every node c other than the root is connected by an edge to some one other node p called the *parent* of c. We also call c a *child* of p. We draw the parent of a node above that node. For example, in Fig. 5.1, n_1 is the parent of n_2, n_3, and n_4, while n_2 is the parent of n_5 and n_6. Said another way, n_2, n_3, and n_4 are children of n_1, while n_5 and n_6 are children of n_2.

All nodes are connected to the root

3. A tree is *connected* in the sense that if we start at any node n other than the root, move to the parent of n, to the parent of the parent of n, and so on, we eventually reach the root of the tree. For instance, starting at n_7, we move to its parent, n_4, and from there to n_4's parent, which is the root, n_1.

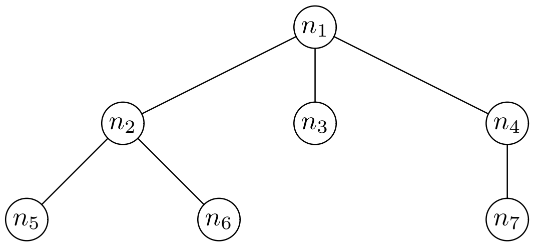


Fig. 5.1. Tree with seven nodes.

An Equivalent Recursive Definition of Trees

It is also possible to define trees recursively with an inductive definition that constructs larger trees out of smaller ones.

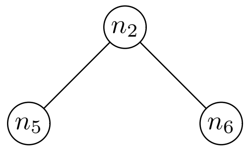
BASIS. A single node n is a tree. We say that n is the root of this one-node tree.

INDUCTION. Let r be a new node and let $T_1, T_2, \ldots, T_k$ be one or more trees with roots $c_1, c_2, \ldots, c_k$, respectively. We require that no node appear more than once in the T_i's; and of course r, being a "new" node, cannot appear in any of these trees. We form a new tree T from r and $T_1, T_2, \ldots, T_k$ as follows:

a) Make r the root of tree T.

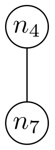
b) Add an edge from r to each of c_1, $c_2, \ldots, c_k$, thereby making each of these nodes a child of the root r. Another way to view this step is that we have made r the parent of each of the roots of the trees T_1, $T_2, \ldots, T_k$.

Example 5.1. We can use this recursive definition to construct the tree in Fig. 5.1. This construction also verifies that the structure in Fig. 5.1 is a tree. The nodes n_5 and n_6 are each trees themselves by the basis rule, which says that a single node can be considered a tree. Then we can apply the inductive rule to create a new tree with n_2 as the root r, and the tree T_1, consisting of n_5 alone, and the tree T_2, consisting of n_6 alone, as children of this new root. The nodes c_1 and c_2 are n_5 and n_6, respectively, since these are the roots of the trees T_1 and T_2. As a result, we can conclude that the structure



is a tree; its root is n_2.

Similarly, n_7 alone is a tree by the basis, and by the inductive rule, the structure



is a tree; its root is n_4.

Node n_3 by itself is a tree. Finally, if we take the node n_1 as r, and n_2, n_3, and n_4 as the roots of the three trees just mentioned, we create the structure in Fig. 5.1, verifying that it indeed is a tree.

Paths, Ancestors, and Descendants

The parent-child relationship can be extended naturally to ancestors and descendants. Informally, the ancestors of a node are found by following the unique path from the node to its parent, to its parent's parent, and so on. Strictly speaking, a node is also its own ancestor. The descendant relationship is the inverse of the ancestor relationship, just as the parent and child relationships are inverses of each other. That is, node d is a descendant of node a if and only if a is an ancestor of d.

More formally, suppose $m_1, m_2, \ldots, m_k$ is a sequence of nodes in a tree such that m_1 is the parent of m_2, which is the parent of m_3, and so on, down to m_{k-1}, which is the parent of m_k. Then $m_1, m_2, \ldots, m_k$ is called a *path* from m_1 to m_k in the tree. The *length* of the path is $k-1$, one less than the number of nodes on the path. Note that a path may consist of a single node (if $k = 1$), in which case the length of the path is 0.

Path length

Example 5.2. In Fig. 5.1, n_1, n_2, n_6 is a path of length 2 from the root n_1 to the node n_6; n_1 is a path of length zero from n_1 to itself.

Proper ancestor and descendant

If $m_1, m_2, \ldots, m_k$ is a path in a tree, node m_1 is called an *ancestor* of m_k and node m_k a *descendant* of m_1. If the path is of length 1 or more, then m_1 is called a *proper* ancestor of m_k and m_k a *proper* descendant of m_1. Again, remember that the case of a path of length 0 is possible, in which case the path lets us conclude that m_1 is an ancestor of itself and a descendant of itself, although not a proper ancestor or descendant. The root is an ancestor of every node in a tree and every node is a descendant of the root.

Example 5.3. In Fig. 5.1, all seven nodes are descendants of n_1, and n_1 is an ancestor of all nodes. Also, all nodes but n_1 itself are proper descendants of n_1, and n_1 is a proper ancestor of all nodes in the tree but itself. The ancestors of n_5 are n_5, n_2, and n_1. The descendants of n_4 are n_4 and n_7.

Sibling

Nodes that have the same parent are sometimes called *siblings.* For example, in Fig. 5.1, nodes n_2, n_3, and n_4 are siblings, and n_5 and n_6 are siblings.

Subtrees

In a tree T, a node n, together with all of its proper descendants, if any, is called a *subtree* of T. Node n is the root of this subtree. Notice that a subtree satisfies the three conditions for being a tree: it has a root, all other nodes in the subtree have a unique parent in the subtree, and by following parents from any node in the subtree, we eventually reach the root of the subtree.

Example 5.4. Referring again to Fig. 5.1, node n_3 by itself is a subtree, since n_3 has no descendants other than itself. As another example, nodes n_2, n_5, and n_6 form a subtree, with root n_2, since these nodes are exactly the descendants of n_2. However, the two nodes n_2 and n_6 by themselves do not form a subtree without node n_5. Finally, the entire tree of Fig. 5.1 is a subtree of itself, with root n_1.

Leaves and Interior Nodes

A *leaf* is a node of a tree that has no children. An *interior node* is a node that has one or more children. Thus, every node of a tree is either a leaf or an interior node, but not both. The root of a tree is normally an interior node, but if the tree consists of only one node, then that node is both the root and a leaf.

Example 5.5. In Fig. 5.1, the leaves are n_5, n_6, n_3, and n_7. The nodes n_1, n_2, and n_4 are interior.

Height and Depth

Level

In a tree, the *height* of a node n is the length of a longest path from n to a leaf. The *height of the tree* is the height of the root. The *depth*, or *level*, of a node n is the length of the path from the root to n.

Example 5.6. In Fig. 5.1, node n_1 has height 2, n_2 has height 1, and leaf n_3 has height 0. In fact, any leaf has height 0. The tree in Fig. 5.1 has height 2. The depth of n_1 is 0, the depth of n_2 is 1, and the depth of n_5 is 2.

Ordered Trees

Optionally, we can assign a left-to-right order to the children of any node. For example, the order of the children of n_1 in Fig. 5.1 is n_2 leftmost, then n_3, then n_4. This left-to-right ordering can be extended to order all the nodes in a tree. If m and n are siblings and m is to the left of n, then all of m's descendants are to the left of all of n's descendants.

Example 5.7. In Fig. 5.1, the nodes of the subtree rooted at n_2 — that is, n_2, n_5, and n_6 — are all to the left of the nodes of the subtrees rooted at n_3 and n_4. Thus, n_2, n_5, and n_6 are all to the left of n_3, n_4, and n_7.

In a tree, take any two nodes x and y neither of which is an ancestor of the other. As a consequence of the definition of "to the left," one of x and y will be to the left of the other. To tell which, follow the paths from x and y toward the root. At some point, perhaps at the root, perhaps lower, the paths will meet at some node z as suggested by Fig. 5.2. The paths from x and y reach z from two different nodes m and n, respectively; it is possible that $m = x$ and/or $n = y$, but it must be that $m \neq n$, or else the paths would have converged somewhere below z.

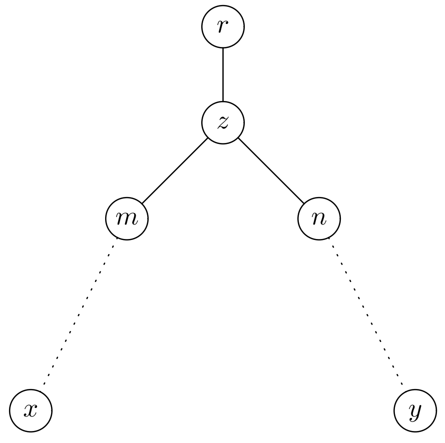


Fig. 5.2. Node x is to the left of node y.

Suppose m is to the left of n. Then since x is in the subtree rooted at m and y is in the subtree rooted at n, it follows that x is to the left of y. Similarly, if m were to the right of n, then x would be to the right of y.

Example 5.8. Since no leaf can be an ancestor of another leaf, it follows that all leaves can be ordered "from the left." For instance, the order of the leaves in Fig. 5.1 is n_5, n_6, n_3, n_7.

Labeled Trees

A *labeled* tree is a tree in which a label or value is associated with each node of the tree. We can think of the label as the information associated with a given node. The label can be something as simple, such as a single integer, or complex, such as the text of an entire document. We can change the label of a node, but we cannot change the name of a node.

If the name of a node is not important, we can represent a node by its label. However, the label does not always provide a unique name for a node, since several nodes may have the same label. Thus, many times we shall draw a node with both its label and its name. The following paragraphs illustrate the concept of a labeled tree and offer some samples.

Expression Trees — An Important Class of Trees

Arithmetic expressions are representable by labeled trees, and it is often quite helpful to visualize expressions as trees. In fact, *expression trees*, as they are sometimes called, specify the association of an expression's operands and its operators in a uniform way, regardless of whether the association is required by the placement of parentheses in the expression or by the precedence and associativity rules for the operators involved.

Let us recall the discussion of expressions in Section 2.6, especially Example 2.17, where we gave a recursive definition of expressions involving the usual arithmetic operators. By analogy with the recursive definition of expressions, we can recursively define the corresponding labeled tree. The general idea is that each time we form a larger expression by applying an operator to smaller expressions, we create a new node, labeled by that operator. The new node becomes the root of the tree for the large expression, and its children are the roots of the trees for the smaller expressions.

For instance, we can define the labeled trees for arithmetic expressions with the binary operators $+$, $-$, $\times$, and $/$, and the unary operator $-$, as follows.

BASIS. A single atomic operand (e.g., a variable, an integer, or a real, as in Section 2.6) is an expression, and its tree is a single node, labeled by that operand.

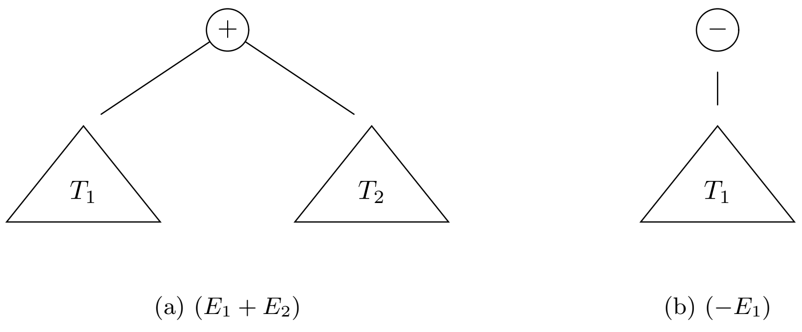


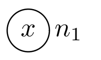
Fig. 5.3. Expression trees for $(E_1 + E_2)$ and $(-E_1)$.

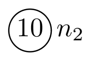
INDUCTION. If E_1 and E_2 are expressions represented by trees T_1 and T_2, respectively, then the expression $(E_1 + E_2)$ is represented by the tree of Fig. 5.3(a), whose root is labeled $+$. This root has two children, which are the roots of T_1 and T_2, respectively, in that order. Similarly, the expressions $(E_1 - E_2)$, $(E_1 \times E_2)$, and (E_1/E_2) have expression trees with roots labeled $-$, $\times$, and $/$, respectively, and subtrees T_1 and T_2. Finally, we may apply the unary minus operator to one expression, E_1. We introduce a root labeled $-$, and its one child is the root of T_1; the tree for $(-E_1)$ is shown in Fig. 5.3(b).

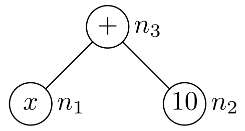
Example 5.9. In Example 2.17 we discussed the recursive construction of a sequence of six expressions from the basis and inductive rules. These expressions, listed in Fig. 2.16, were

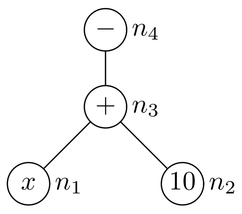
i)	x	*iv)*	$\big(-(x+10)\big)$
ii)	10	*v)*	y
iii)	$(x+10)$	*vi)*	$\Big(y \times \big(-(x+10)\big)\Big)$

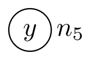
Expressions (*i*), (*ii*), and (*v*) are single operands, and so the basis rule tells us that the trees of Fig. 5.4(a), (b), and (e), respectively, represent these expressions. Note that each of these trees consists of a single node to which we have given a name — n_1, n_2, and n_5, respectively — and a label, which is the operand in the circle.


(a) For x.


(b) For 10.


(c) For $(x + 10)$.


(d) For $(-(x + 10))$.


(e) For y.

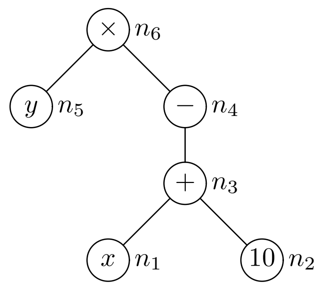

(f) For $(y \times (-(x + 10)))$.

Fig. 5.4. Construction of expression trees.

Expression (*iii*) is formed by applying the operator $+$ to the operands x and 10, and so we see in Fig. 5.4(c) the tree for this expression, with root labeled $+$, and the roots of the trees in Fig. 5.4(a) and (b) as its children. Expression (*iv*) is formed by applying unary $-$ to expression (*iii*), so that the tree for $\big(-(x+10)\big)$, shown in Fig. 5.4(d), has root labeled $-$ above the tree for $(x+10)$. Finally, the tree for the expression $\Big(y \times \big(-(x+10)\big)\Big)$, shown in Fig. 5.4(f), has a root labeled $\times$, whose children are the roots of the trees of Fig. 5.4(e) and (d), in that order.

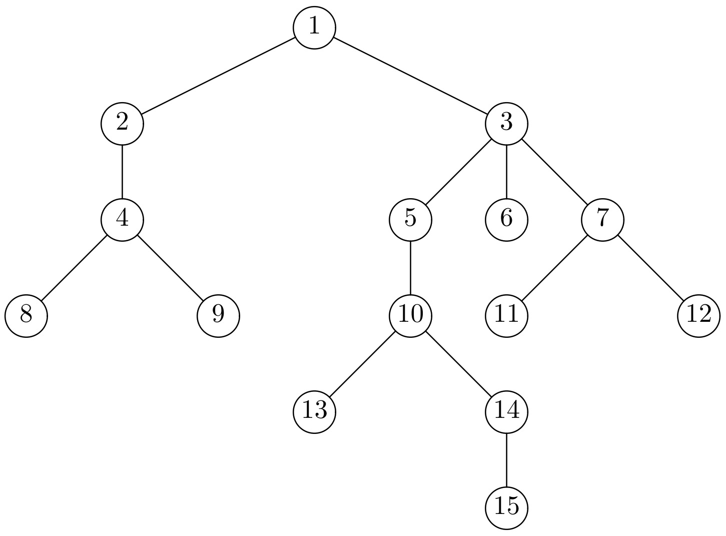


Fig. 5.5. Tree for Exercise 5.2.1.

EXERCISES

5.2.1: In Fig. 5.5 we see a tree. Tell what is described by each of the following phrases:

a) The root of the tree
b) The leaves of the tree
c) The interior nodes of the tree
d) The siblings of node 6
e) The subtree with root 5
f) The ancestors of node 10
g) The descendants of node 10
h) The nodes to the left of node 10
i) The nodes to the right of node 10
j) The longest path in the tree
k) The height of node 3
l) The depth of node 13
m) The height of the tree

5.2.2: Can a leaf in a tree ever have any (a) descendants? (b) proper descendants?

5.2.3: Prove that in a tree no leaf can be an ancestor of another leaf.

5.2.4*: Prove that the two definitions of trees in this section are equivalent. *Hint*: To show that a tree according the nonrecursive definition is a tree according the recursive definition, use induction on the number of nodes in the tree. In the opposite direction, use induction on the number of rounds used in the recursive definition.

5.2.5: Suppose we have a graph consisting of four nodes, r, a, b, and c. Node r is an isolated node and has no edges connecting it. The remaining three nodes form a cycle; that is, we have an edge connecting a and b, an edge connecting b and c, and an edge connecting c and a. Why is this graph not a tree?

5.2.6: In many kinds of trees, there is a significant distinction between the interior nodes and the leaves (or rather the labels of these two kinds of nodes). For example, in an expression tree, the interior nodes represent operators, and the leaves represent atomic operands. Give the distinction between interior nodes and leaves for each of the following kinds of trees:

a) Trees representing directory structures, as in Section 1.3
b) Trees representing the splitting and merging of lists for merge sort, as in Section 2.8
c) Trees representing the structure of a function, as in Section 3.7

5.2.7: Give expression trees for the following expressions. Note that, as is customary with expressions, we have omitted redundant parentheses. You must first restore the proper pairs of parentheses, using the customary rules for precedence and associativity of operators.

a) $(x + 1) \times (x - y + 4)$
b) $1 + 2 + 3 + 4 + 5 + 6$
c) $9 \times 8 + 7 \times 6 + 5$

5.2.8: Show that if x and y are two distinct nodes in an ordered tree, then exactly one of the following conditions must hold:

a) x is a proper ancestor of y
b) x is a proper descendant of y
c) x is to the left of y
d) x is to the right of y

5.3 Data Structures for Trees

Many data structures can be used to represent trees. Which one we should use depends on the particular operations we want to perform. As a simple example, if all we ever want to do is to locate the parents of nodes, then we can represent each node by a structure consisting of a label plus a pointer to the structure representing the parent of that node.

As a general rule, the nodes of a tree can be represented by structures in which the fields link the nodes together in a manner similar to the way in which the nodes are connected in the abstract tree; the tree itself can be represented by a pointer to the root's structure. Thus, when we talk about representing trees, we are primarily interested in how the nodes are represented.

One distinction in representations concerns where the structures for the nodes "live" in the memory of the computer. In C, we can create the space for structures for nodes by using the function `malloc` from the standard library `stdlib.h`, in which case nodes "float" in memory and are accessible only through pointers. Alternatively, we can create an array of structures and use elements of the array to represent nodes. Again nodes can be linked according to their position in the tree, but it is also possible to visit nodes by walking down the array. We can thus access nodes without following a path through the tree. The disadvantage of an array-based representation is that we cannot create more nodes than there are elements in the array. In what follows, we shall assume that nodes are created by `malloc`, although in situations where there is a limit on how large trees can grow, an array of structures of the same type is a viable, and possibly preferred, alternative.

Array-of-Pointers Representation of Trees

One of the simplest ways of representing a tree is to use for each node a structure consisting of a field or fields for the label of the node, followed by an array of pointers to the children of that node. Such a structure is suggested by Fig. 5.6. The constant *bf* is the size of the array of pointers. It represents the maximum number of children a node can have, a quantity known as the *branching factor.* The ith component of the array at a node contains a pointer to the ith child of that node. A missing child can be represented by a `NULL` pointer.

Branching factor

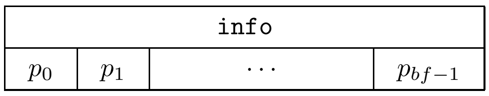


Fig. 5.6. Node represented by an array of pointers.

In C this data structure can be represented by the type declaration

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE children[BF];
};
```

Here, the field `info` represents the information that constitutes the label of a node and `BF` is the constant defined to be the branching factor. We shall see many variants of this declaration throughout this chapter.

In this and most other data structures for trees, we represent a tree by a pointer to the root node. Thus, `pNODE` also serves as the type of a tree. We could, in fact, use the type `TREE` in place of `pNODE`, and we shall adopt that convention when we talk about binary trees starting in Section 5.6. However, for the moment, we shall use the name `pNODE` for the type "pointer to node," since in some data structures, pointers to nodes are used for other purposes besides representing trees.

The array-of-pointers representation allows us to access the ith child of any node in $O(1)$ time. This representation, however, is very wasteful of space when only a few nodes in the tree have many children. In this case, most of the pointers in the arrays will be `NULL`.

Try to Remember Trie

The term “trie” comes from the middle of the word “retrieval.” It was originally intended to be pronounced “tree.” Fortunately, common parlance has switched to the distinguishing pronunciation “try.”

Example 5.10. A tree can be used to represent a collection of words in a way that makes it quite efficient to check whether a given sequence of characters is a valid word. In this type of tree, called a *trie,* each node except the root has an associated letter. The string of characters represented by a node n is the sequence of letters along the path from the root to n. Given a set of words, the trie consists of nodes for exactly those strings of characters that are prefixes of some word in the set. The label of a node consists of the letter represented by the node and also a Boolean telling whether or not the string from the root to that node forms a complete word; we shall use for the Boolean the integer 1 if so and 0 if not.[1]

Trie

For instance, suppose our “dictionary” consists of the four words `he`, `hers`, `his`, `she`. A trie for these words is shown in Fig. 5.7. To determine whether the word `he` is in the set, we start at the root n_1, move to the child n_2 labeled `h`, and then from that node move to its child n_4 labeled `e`. Since these nodes all exist in the tree, and n_4 has 1 as part of its label, we conclude that `he` is in the set.

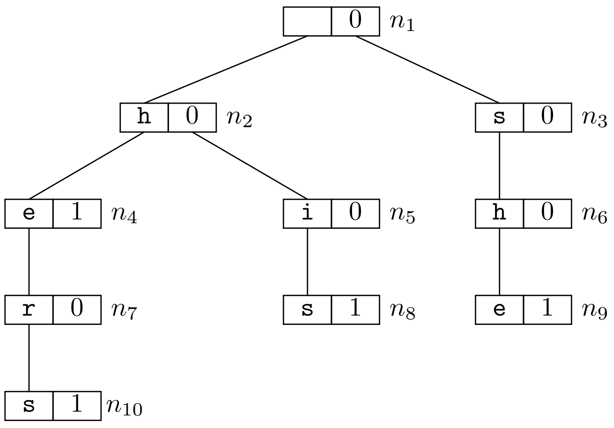


Fig. 5.7. Trie for words `he, hers, his, and she`.

As another example, suppose we want to determine whether `him` is in the set. We follow the path from the root to n_2 to n_5, which represents the prefix `hi`; but at n_5 we find no child corresponding to the letter `m`. We conclude that `him` is not in the set. Finally, if we search for the word `her`, we find our way from the root to node n_7. That node exists but does not have a 1. We therefore conclude that `her` is not in the set, although it is a proper prefix of a word, `hers`, in the set.

Nodes in a trie have a branching factor equal to the number of different characters in the alphabet from which the words are formed. For example, if we do not

[1] In the previous section we acted as if the label was a single value. However, values can be of any type, and labels can be structures consisting of two or more fields. In this case, the label has one field that is a letter and a second that is an integer that is either 0 or 1.

distinguish between upper- and lower-case, and words contain no special characters such as apostrophes, then we can take the branching factor to be 26. The type of a node, including the two label fields, can be defined as in Fig. 5.8. In the array `children`, we assume that the letter `a` is represented by index 0, the letter `b` by index 1, and so on.

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE children[BF];
};
```

Fig. 5.8. Definition of an alphabetic trie.

The abstract trie of Fig. 5.7 can be represented by the data structure of Fig. 5.9. We represent nodes by showing the first two fields, `letter` and `isWord`, along with those elements of the array `children` that have non-`NULL` pointers. In the `children` array, for each non-`NULL` element, the letter indexing the array is shown in the entry above the pointer to the child, but that letter is not actually present in the structure. Note that the `letter` field of the root is irrelevant.

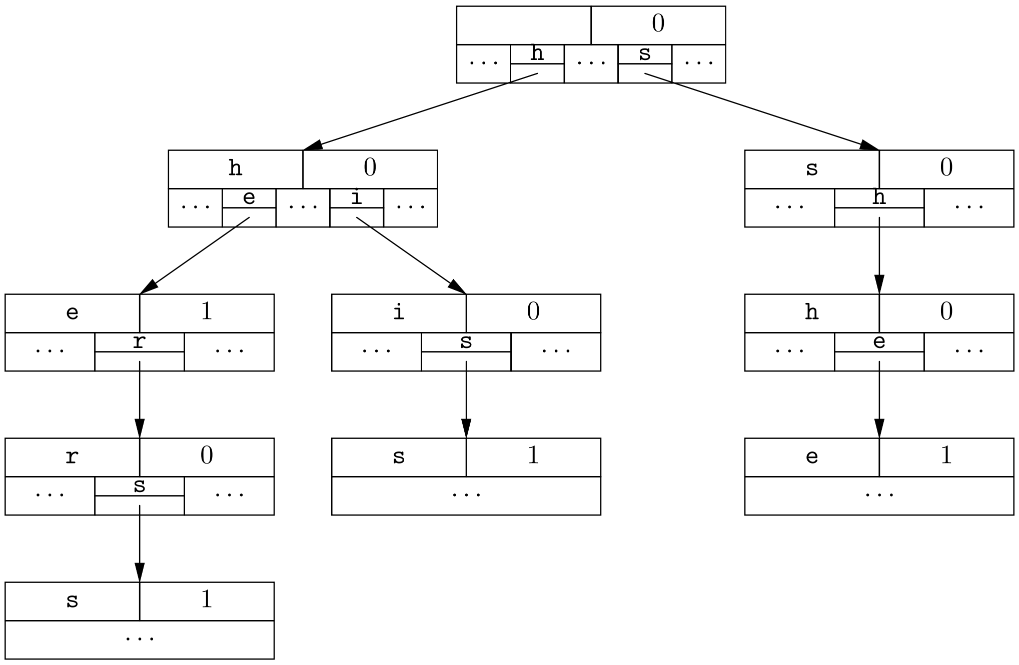


Fig. 5.9. Data structure for the trie of Fig. 5.7.

Leftmost-Child–Right-Sibling Representation of Trees

Using arrays of pointers for nodes is not necessarily space-efficient, because in typical cases, the great majority of pointers will be `NULL`. That is certainly the case in Fig. 5.9, where no node has more than two non-`NULL` pointers. In fact, if we think about it, we see that the number of pointers in any trie based on a 26-letter alphabet will have 26 times as many spaces for pointers as there are nodes. Since no node can have two parents and the root has no parent at all, it follows that among N nodes there are only $N-1$ non-`NULL` pointers; that is, less than one out of 26 pointers is useful.

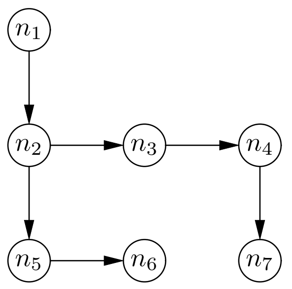
One way to overcome the space inefficiency of the array-of-pointers representation of a tree is to use linked lists to represent the children of nodes. The space occupied by a linked list for a node is proportional to the number of children of that node. There is, however, a time penalty with this representation; accessing the ith child takes $O(i)$ time, because we must traverse a list of length $i-1$ to get to the ith node. In comparison, we can get to the ith child in $O(1)$ time, independent of i, using an array of pointers to the children.

Right sibling

In the representation of trees called *leftmost-child–right-sibling,* we put into each node a pointer only to its leftmost child; a node does not have pointers to any of its other children. To find the second and subsequent children of a node n, we create a linked list of those children, with each child c pointing to the child of n immediately to the right of c. That node is called the *right sibling* of c.

Example 5.11. In Fig. 5.1, n_3 is the right sibling of n_2, n_4 is the right sibling of n_3, and n_4 has no right sibling. We would find the children of n_1 by following its leftmost-child pointer to n_2, then the right-sibling pointer to n_3, and then the right-sibling pointer of n_3 to n_4. There, we would find a `NULL` right-sibling pointer and know that n_1 has no more children.

Figure 5.10 contains a sketch of the leftmost-child–right-sibling representation for the tree in Fig. 5.1. The downward arrows are the leftmost-child links; the sideways arrows are the right-sibling links.



$$\begin{tikzcd}
n_1 \arrow[d] & & \\
n_2 \arrow[r] \arrow[d] & n_3 \arrow[r] & n_4 \arrow[d] \\
n_5 \arrow[r] & n_6 & n_7
\end{tikzcd}$$

Fig. 5.10. Leftmost-child–right-sibling representation for the tree in Fig. 5.1.

In a leftmost-child–right-sibling representation of a tree, nodes can be defined as follows:

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE leftmostChild, rightSibling;
};
```

The field `info` holds the label associated with the node and it can have any type. The fields `leftmostChild` and `rightSibling` point to the leftmost child and right sibling of the node in question. Note that while `leftmostChild` gives information about the node itself, the field `rightSibling` at a node is really part of the linked list of children of that node's parent.

Example 5.12. Let us represent the trie of Fig. 5.7 in the leftmost-child–right-sibling form. First, the type of nodes is

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling;
};
```

The first two fields represent information, according to the scheme described in Example 5.10. The trie of Fig. 5.7 is represented by the data structure shown in Fig. 5.11. Notice that each leaf has a `NULL` leftmost-child pointer, and each rightmost child has a `NULL` right-sibling pointer.

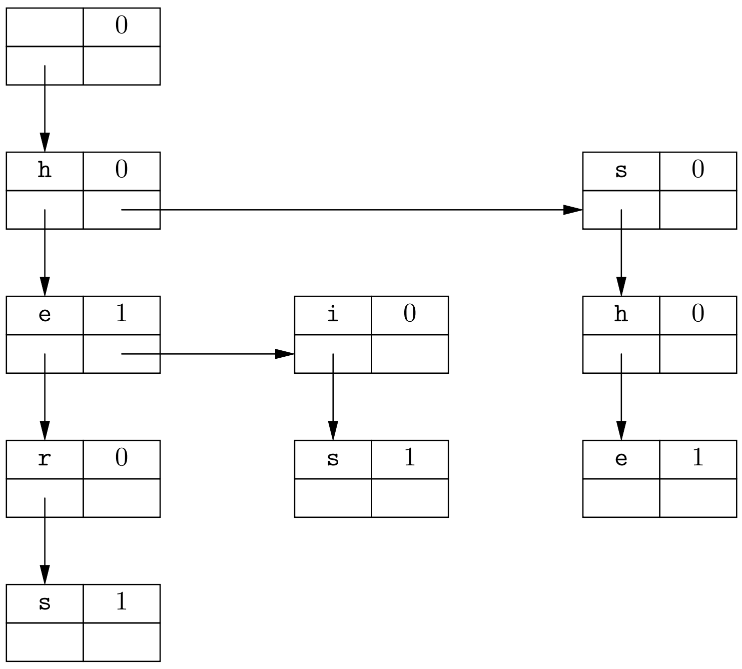


Fig. 5.11. Leftmost-child–right-sibling representation for the trie of Fig. 5.7.

As an example of how one uses the leftmost-child–right-sibling representation, we see in Fig. 5.12 a function `seek(let, n)` that takes a letter *let* and a pointer to a node n as arguments. It returns a pointer to the child of n that has *let* in its `letter` field, and it returns `NULL` if there is no such node. In the while loop of Fig. 5.12, each child of n is examined in turn. We reach line (6) if either *let* is found or we have examined all the children and thus have fallen out of the loop. In either case, `c` holds the correct value, a pointer to the child holding *let* if there is one, and `NULL` if not.

Notice that `seek` takes time proportional to the number of children that must be examined until we find the child we are looking for, and if we never find it, then the time is proportional to the number of children of node n. In comparison, using the array-of-pointers representation of trees, `seek` could simply return the value of the array element for letter *let*, taking $O(1)$ time.

```
        pNODE seek(char let, pNODE n)
        {
(1)         c = n->leftmostChild;
(2)         while (c != NULL)
(3)             if (c->letter == let)
(4)                 break;
                else
(5)                 c = c->rightSibling;
(6)         return c;
        }
```

Fig. 5.12. Finding the child for a desired letter.

Parent Pointers

Sometimes, it is useful to include in the structure for each node a pointer to the parent. The root has a `NULL` parent pointer. For example, the structure of Example 5.12 could become

```
typdef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling, parent;
};
```

With this structure, it becomes possible to determine what word a given node represents. We repeatedly follow parent pointers until we come to the root, which we can identify because it alone has the value of `parent` equal to `NULL`. The `letter` fields encountered along the way spell the word, backward.

EXERCISES

5.3.1: For each node in the tree of Fig. 5.5, indicate the leftmost child and right sibling.

Comparison of Tree Representations

We summarize the relative merits of the array-of-pointers (trie) and the leftmost-child–right-sibling representations for trees:

The array-of-pointers representation offers faster access to children, requiring $O(1)$ time to reach any child, no matter how many children there are.

The leftmost-child–right-sibling representation uses less space. For instance, in our running example of the trie of Fig. 5.7, each node contains 26 pointers in the array representation and two pointers in the leftmost-child–right-sibling representation.

The leftmost-child–right-sibling representation does not require that there be a limit on the branching factor of nodes. We can represent trees with any branching factor, without changing the data structure. However, if we use the array-of-pointers representation, once we choose the size of the array, we cannot represent a tree with a larger branching factor.

5.3.2: Represent the tree of Fig. 5.5

a) As a trie with branching factor 3
b) By leftmost-child and right-sibling pointers

How many bytes of memory are required by each representation?

5.3.3: Consider the following set of singular personal pronouns in English: I, my, mine, me, you, your, yours, he, his, him, she, her, hers. Augment the trie of Fig. 5.7 to include all thirteen of these words.

5.3.4: Suppose that a complete dictionary of English contains 2,000,000 words and that the number of prefixes of words — that is, strings of letters that can be extended at the end by zero or more additional letters to form a word — is 10,000,000.

a) How many nodes would a trie for this dictionary have?

b) Suppose that we use the structure in Example 5.10 to represent nodes. Let pointers require four bytes, and suppose that the information fields `letter` and `isWord` each take one byte. How many bytes would the trie require?

c) Of the space calculated in part (b), how much is taken up by `NULL` pointers?

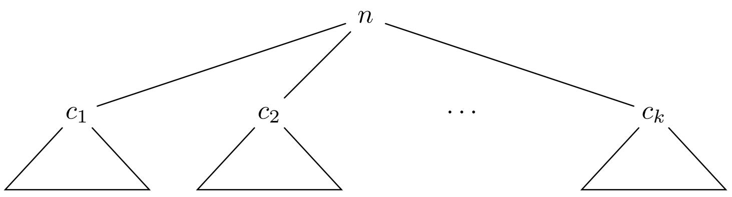
5.3.5: Suppose we represent the dictionary described in Exercise 5.3.4 by using the structure of Example 5.12 (a leftmost-child–right-sibling representation). Under the same assumptions about space required by pointers and information fields as in Exercise 5.3.4(b), how much space does the tree for the dictionary require? What portion of that space is `NULL` pointers?

Lowest common ancestor

5.3.6: In a tree, a node c is the *lowest common ancestor* of nodes x and y if c is an ancestor of both x and y, and no proper descendant of c is an ancestor of x and y. Write a program that will find the lowest common ancestor of any pair of nodes in a given tree. What is a good data structure for trees in such a program?

5.4 Recursions on Trees

The usefulness of trees is highlighted by the number of recursive operations on trees that can be written naturally and cleanly. Figure 5.13 suggests the general form of a recursive function $F(n)$ that takes a node n of a tree as argument. F first performs some steps (perhaps none), which we represent by action A_0. Then F calls itself on the first child, c_1, of n. During this recursive call, F will "explore" the subtree rooted at c_1, doing whatever it is F does to a tree. When that call returns to the call at node n, some other action — say A_1 — is performed. Then F is called on the second child of n, resulting in exploration of the second subtree, and so on, with actions at n alternating with calls to F on the children of n.



(a) General form of a tree.

```
$F(n)$
{
    action $A_0$;
    $F(c_1)$;
    action $A_1$;
    $F(c_2)$;
    action $A_2$;
    ···
    $F(c_k)$;
    action $A_k$;
}
```

(b) General form of recursive function $F(n)$ on a tree.

Fig. 5.13. A recursive function on a tree.

Preorder

Example 5.13. A simple recursion on a tree produces what is known as the *preorder listing* of the node labels of the tree. Here, action A_0 prints the label of the node, and the other actions do nothing other than some "bookkeeping" operations that enable us to visit each child of a given node. The effect is to print the labels as we would first meet them if we started at the root and circumnavigated the tree, visiting all the nodes in a counterclockwise tour. Note that we print the label of a node only the first time we visit that node. The circumnavigation is suggested by the arrow in Fig. 5.14, and the order in which the nodes are visited is $+a+*-b-c-*d*+$. The preorder listing is the sequence of node labels $+a*-bcd$.

Let us suppose that we use a leftmost-child–right-sibling representation of nodes in an expression tree, with labels consisting of a single character. The label of an interior node is the arithmetic operator at that node, and the label of a leaf is

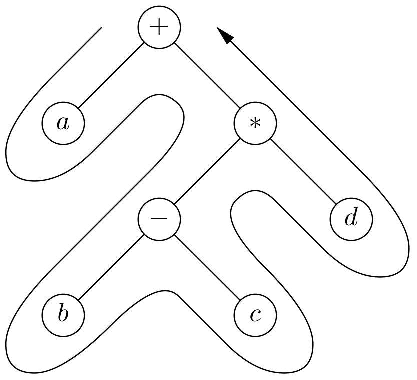


Fig. 5.14. An expression tree and its circumnavigation.

a letter standing for an operand. Nodes and pointers to nodes can be defined as follows:

```
typedef struct NODE *pNODE;
struct NODE {
    char nodeLabel;
    pNODE leftmostChild, rightSibling;
};
```

The function `preorder` is shown in Fig. 5.15. In the explanation that follows, it is convenient to think of pointers to nodes as if they were the nodes themselves.

```
                void preorder(pNODE n)
                {
                    pNODE c; /* a child of node n */

(1)                 printf("%c\n", n->nodeLabel);
(2)                 c = n->leftmostChild;
(3)                 while (c != NULL) {
(4)                     preorder(c);
(5)                     c = c->rightSibling;
                    }
                }
```

Fig. 5.15. Preorder traversal function.

Action "A_0" consists of the following parts of the program in Fig. 5.15:

1. Printing the label of node n, at line (1),
2. Initializing c to be the leftmost child of n, at line (2), and
3. Performing the first test for `c != NULL`, at line (3).

Line (2) initializes a loop in which c becomes each child of n, in turn. Note that if n is a leaf, then c is assigned the value `NULL` at line (2).

We go around the while-loop of lines (3) to (5) until we run out of children of n. For each child, we call the function `preorder` recursively on that child, at line (4), and then advance to the next child, at line (5). Each of the actions A_i, for $i \geq 1$, consists of line (5), which moves c through the children of n, and the test at line (3) to see whether we have exhausted the children. These actions are for bookkeeping only; in comparison, line (1) in action A_0 does the significant step, printing the label.

The sequence of events for calling `preorder` on the root of the tree in Fig. 5.14 is summarized in Fig. 5.16. The character at the left of each line is the label of the node n at which the call of `preorder(n)` is currently being executed. Because no two nodes have the same label, it is convenient here to use the label of a node as its name. Notice that the characters printed are $+a*-bcd$, in that order, which is the same as the order of circumnavigation.

```
        call preorder(+)
(+)         print +
(+)         call preorder(a)
(a)             print a
(+)         call preorder(*)
(*)             print *
(*)             call preorder(-)
(-)                 print -
(-)                 call preorder(b)
(b)                     print b
(-)                 call preorder(c)
(c)                     print c
(*)             call preorder(d)
(d)                 print d
```

Fig. 5.16. Action of recursive function `preorder` on tree of Fig. 5.14.

Postorder

Example 5.14. Another common way to order the nodes of the tree, called *postorder*, corresponds to circumnavigating the tree as in Fig. 5.14 but listing a node the last time it is visited, rather than the first. For instance, in Fig. 5.14, the postorder listing is $abc - d * +$.

To produce a postorder listing of the nodes, the last action does the printing, and so a node's label is printed after the postorder listing function is called on all of its children, in order from the left. The other actions initialize the loop through the children or move to the next child. Note that if a node is a leaf, all we do is list the label; there are no recursive calls.

If we use the representation of Example 5.13 for nodes, we can create postorder listings by the recursive function `postorder` of Fig. 5.17. The action of this function when called on the root of the tree in Fig. 5.14 is shown in Fig. 5.18. The same convention regarding node names is used here as in Fig. 5.16.

Example 5.15. Our next example requires us to perform significant actions among all of the recursive calls on subtrees. Suppose we are given an expression tree with integers as operands, and with binary operators, and we wish to produce

```
void postorder(pNODE n)
{
    pNODE c; /* a child of node n */

    c = n->leftmostChild;
    while (c != NULL) {
        postorder(c);
        c = c->rightSibling;
    }
    printf("%c\n", n->nodeLabel);
}
```

Fig. 5.17. Recursive postorder function.

```
       call postorder(+)
(+)        call postorder(a)
(a)            print a
(+)        call postorder(*)
(*)            call postorder(-)
(-)                call postorder(b)
(b)                    print b
(-)                call postorder(c)
(c)                    print c
(-)                print -
(*)            call postorder(d)
(d)                print d
(*)            print *
(+)        print +
```

Fig. 5.18. Action of recursive function `postorder` on tree of Fig. 5.14.

the numerical value of the expression represented by the tree. We can do so by executing the following recursive algorithm on the expression tree.

Evaluating an expression tree

BASIS. For a leaf we produce the integer value of the node as the value of the tree.

INDUCTION. Suppose we wish to compute the value of the expression formed by the subtree rooted at some node n. We evaluate the subexpressions for the two subtrees rooted at the children of n; these are the values of the operands for the operator at n. We then apply the operator labeling n to the values of these two subtrees, and we have the value of the entire subtree rooted at n.

We define a pointer to a node and a node as follows:

Prefix and Postfix Expressions

When we list the labels of an expression tree in preorder, we get the *prefix expression* equivalent to the given expression. Similarly, the list of the labels of an expression tree in postorder yields the equivalent *postfix expression.* Expressions in the ordinary notation, where binary operators appear between their operands, are called *infix expressions.* For instance, the expression tree of Fig. 5.14 has the infix expression $a + (b - c) * d$. As we saw in Examples 5.13 and 5.14, the equivalent prefix expression is $+a * -bcd$, and the equivalent postfix expression is $abc - d * +$.

Infix expression

An interesting fact about prefix and postfix notations is that, as long as each operator has a unique number of arguments (e.g., we cannot use the same symbol for binary and unary minus), then no parentheses are ever needed, yet we can still unambiguously group operators with their operands.

We can construct an infix expression from a prefix expression as follows. In the prefix expression, we find an operator that is followed by the required number of operands, with no embedded operators. In the prefix expression $+a * -bcd$, for example, the subexpression $-bc$ is such a string, since the minus sign, like all operators in our running example, takes two operands. We replace this subexpression by a new symbol, say $x = -bc$, and repeat the process of identifying an operator followed by its operands. In our example, we now work with $+a * xd$. At this point we identify the subexpression $y = *xd$ and reduce the remaining string to $+ay$. Now the remaining string is just an instance of an operator and its operands, and so we convert it to the infix expression $a + y$.

We may now reconstruct the remainder of the infix expression by retracing these steps. We observe that the subexpression $y = *xd$ in infix is $x * d$, and so we may substitute for y in $a + y$ to get $a + (x * d)$. Note that in general, parentheses are needed in infix expressions, although in this case, we can omit them because of the convention that $*$ takes precedence over $+$ when grouping operands. Then we substitute for $x = -bc$ the infix expression $b - c$, and so our final expression is $a + \big((b - c) * d\big)$, which is the same as that represented by the tree of Fig. 5.14.

For a postfix expression, we can use a similar algorithm. The only difference is that we look for an operator preceded by the requisite number of operands in order to decompose a postfix expression.

```
typedef struct NODE *pNODE;
struct NODE {
    char op;
    int value;
    pNODE leftmostChild, rightSibling;
};
```

The field `op` will hold either the character for an arithmetic operator, or the character `i`, which stands for "integer" and identifies a node as a leaf. If the node is a leaf, then the `value` field holds the integer represented; `value` is not used at interior nodes.

This notation allows operators with any number of arguments, although we shall write code on the simplifying assumption that all operators are binary. The code appears in Fig. 5.19.

```
         int eval(pNODE n)
         {
             int val1, val2; /* values of first and second subtrees */

 (1)         if (n->op) == 'i') /* n points to a leaf */
 (2)             return n->value;
             else {/* n points to an interior node */
 (3)             val1 = eval(n->leftmostChild);
 (4)             val2 = eval(n->leftmostChild->rightSibling);
 (5)             switch (n->op) {
 (6)                 case '+': return val1 + val2;
 (7)                 case '-': return val1 - val2;
 (8)                 case '*': return val1 * val2;
 (9)                 case '/': return val1 / val2;
                 }
             }
         }
```

Fig. 5.19. Evaluating an arithmetic expression.

If the node n is a leaf, then the test of line (1) succeeds and we return the integer label of that leaf at line (2). If the node is not a leaf, then we evaluate its left operand at line (3) and its right operand at line (4), storing the results in `val1` and `val2`, respectively. Note in connection with line (4) that the second child of a node n is the right sibling of the leftmost child of the node n. Lines (5) through (9) form a switch statement, in which we decide what the operator at n is and apply the appropriate operation to the values of the left and right operands.

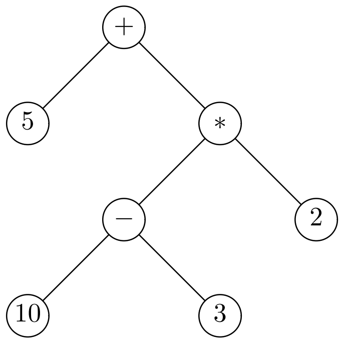


Fig. 5.20. An expression tree with integer operands.

For instance, consider the expression tree of Fig. 5.20. We see in Fig. 5.21 the sequence of calls and returns that are made at each node during the evaluation of this expression. As before, we have taken advantage of the fact that labels are unique and have named nodes by their labels.

Example 5.16. Sometimes we need to determine the height of each node in a tree. The height of a node can be defined recursively by the following function:

```
        call eval(+)
 (+)        call eval(5)
 (5)            return 5
 (+)        call eval(*)
 (*)            call eval(−)
 (−)                call eval(10)
(10)                    return 10
 (−)                call eval(3)
 (3)                    return 3
 (−)                return 7
 (*)            call eval(2)
 (2)                return 2
 (*)            return 14
 (+)        return 19
```

Fig. 5.21. Actions of function `eval` at each node on tree of Fig. 5.20.

Computing the height of a tree

BASIS. The height of a leaf is 0.

INDUCTION. The height of an interior node is 1 greater than the largest of the heights of its children.

We can translate this definition into a recursive program that computes the height of each node into a field `height`:

BASIS. At a leaf, set the height to 0.

INDUCTION. At an interior node, recursively compute the heights of the children, find the maximum, add 1, and store the result into the `height` field.

This program is shown in Fig. 5.22. We assume that nodes are structures of the form

```
typedef struct NODE *pNODE;
struct NODE {
    int height;
    pNODE leftmostChild, rightSibling;
};
```

The function `computeHt` takes a pointer to a node as argument and computes the height of that node in the field `height`. If we call this function on the root of a tree, it will compute the heights of all the nodes of that tree.

At line (1) we initialize the height of n to 0. If n is a leaf, we are done, because the test of line (3) will fail immediately, and so the height of any leaf is computed to be 0. Line (2) sets c to be (a pointer to) the leftmost child of n. As we go around the loop of lines (3) through (7), c becomes each child of n in turn. We recursively compute the height of c at line (4). As we proceed, the value in `n->height` will be 1 greater than the height of the highest child seen so far, but 0 if we have not seen any children. Thus, lines (5) and (6) allow us to increase the height of n if we

Still More Defensive Programming

Several aspects of the program in Fig. 5.19 exhibit a careless programming style that we would avoid were it not our aim to illustrate some points concisely. Specifically, we are following pointers without checking first whether they are `NULL`. Thus, in line (1), n could be `NULL`. We really should begin the program by saying

```
if (n != NULL) /* then do lines (1) to (9) */
else /* print an error message */
```

Even if n is not `NULL`, in line (3) we might find that its `leftmostChild` field is `NULL`, and so we should check whether `n->leftmostChild` is `NULL`, and if so, print an error message and not call `eval`. Similarly, even if the leftmost child of n exists, that node might not have a right sibling, and so before line (4) we need to check that

```
n->leftmostChild->rightSibling != NULL
```

It is also tempting to rely on the assumption that the information contained in the nodes of the tree is correct. For example, if a node is an interior node, it is labeled by a binary operator, and we have assumed it has two children and the pointers followed in lines (3) and (4) cannot possibly be `NULL`. However, it may be possible that the operator label is incorrect. To handle this situation properly, we should add a default case to the switch statement to detect unanticipated operator labels.

As a general rule, relying on the assumption that inputs to programs will always be correct is simplistic at best; in reality, "whatever can go wrong, will go wrong." A program, if it is used more than once, is bound to see data that is not of the form the programmer envisioned. One cannot be too careful in practice – blindly following `NULL` pointers or assuming that input data is always correct are common programming errors.

```
            void computeHt(pNODE n)
            {
                pNODE c;

(1)             n->height = 0;
(2)             c = n->leftmostChild;
(3)             while (c != NULL) {
(4)                 computeHt(c);
(5)                 if (c->height >= n->height)
(6)                     n->height = 1+c->height;
(7)                 c = c->rightSibling;
                }
            }
```

Fig. 5.22. Procedure to compute the height of all the nodes of a tree.

find a new child that is higher than any previous child. Also, for the first child, the test of line (5) will surely be satisfied, and we set `n->height` to 1 more than the height of the first child. When we fall out of the loop because we have seen all the children, `n->height` has been set to 1 more than the maximum height of any of n's children.

EXERCISES

5.4.1: Write a recursive program to count the number of nodes in a tree that is represented by leftmost-child and right-sibling pointers.

5.4.2: Write a recursive program to find the maximum label of the nodes of a tree. Assume that the tree has integer labels, and that it is represented by leftmost-child and right-sibling pointers.

5.4.3: Modify the program in Fig. 5.19 to handle trees containing unary minus nodes.

5.4.4*: Write a recursive program that computes for a tree, represented by leftmost-child and right-sibling pointers, the number of *left-right pairs,* that is, pairs of nodes n and m such that n is to the left of node m. For example, in Fig. 5.20, node 5 is to the left of the nodes labeled $*$, $-$, 10, 3, and 2; node 10 is to the left of nodes 3 and 2; 3 is to the left of 2, and node $-$ is to the left of node 2. Thus, the answer for this tree is nine pairs. *Hint*: Let your recursive function return two pieces of information when called on a node n: the number of left-right pairs in the subtree rooted at n, and also the number of nodes in the subtree rooted at n.

5.4.5: List the nodes of the tree in Fig. 5.5 (see the Exercises for Section 5.2) in (a) preorder and (b) postorder.

5.4.6: For each of the expressions

i) $(x + y) * (x + z)$

ii) $\big((x - y) * z + (y - w)\big) * x$

iii) $\Bigg(\bigg(\Big(\big((a * x + b) * x + c\big) * x + d\Big) * x + e\bigg) * x + f$

do the following:

a) Construct the expression tree.
b) Find the equivalent prefix expression.
c) Find the equivalent postfix expression.

5.4.7: Convert the expression $ab + c * de - /f+$ from postfix to (a) infix and (b) prefix.

5.4.8: Write a function that "circumnavigates" a tree, printing the name of a node each time it is passed.

5.4.9: What are the actions A_0, A_1, and so forth, for the postorder function in Fig. 5.17? ("Actions" are as indicated in Fig. 5.13.)

5.5 Structural Induction

In Chapters 2 and 3 we saw a number of inductive proofs of properties of integers. We would assume that some statement is true about n, or about all integers less than or equal to n, and use this inductive hypothesis to prove the same statement is true about $n + 1$. A similar but not identical form of proof, called "structural induction," is useful for proving properties about trees. Structural induction is analogous to recursive algorithms on trees, and this form of induction is generally the easiest to use when we wish to prove something about trees.

Suppose we want to prove that a statement $S(T)$ is true for all trees T. For a basis, we show that $S(T)$ is true when T consists of a single node. For the induction, we suppose that T is a tree with root r and children $c_1, c_2, \ldots, c_k$, for some $k \geq 1$. Let $T_1, T_2, \ldots, T_k$ be the subtrees of T whose roots are $c_1, c_2, \ldots, c_k$, respectively, as suggested by Fig. 5.23. Then the inductive step is to assume that $S(T_1), S(T_2), \ldots, S(T_k)$ are all true and prove $S(T)$. If we do so, then we can conclude that $S(T)$ is true for all trees T. This form of argument is called *structural induction.* Notice that a structural induction does not make reference to the exact number of nodes in a tree, except to distinguish the basis (one node) from the inductive step (more than one node).

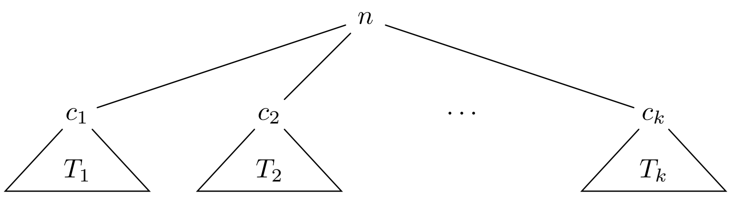


Fig. 5.23. A tree and its subtrees.

```
if (n->op) == 'i') /* n points to a leaf */
    return n->value;
else {/* n points to an interior node */
    val1 = eval(n->leftmostChild);
    val2 = eval(n->leftmostChild->rightSibling);
    switch (n->op) {
        case '+': return val1 + val2;
        case '-': return val1 - val2;
        case '*': return val1 * val2;
        case '/': return val1 / val2;
    }
}
```

Fig. 5.24. The body of the function `eval(n)` from Fig. 5.19.

Example 5.17. A structural induction is generally needed to prove the correctness of a recursive program that acts on trees. As an example, let us reconsider the function `eval` of Fig. 5.19, the body of which we reproduce as Fig. 5.24. This function is applied to a tree T by being given a pointer to the root of T as the value of its argument n. It then computes the value of the expression represented by T. We shall prove by structural induction the following statement:

STATEMENT $S(T)$: The value returned by `eval` when called on the root of T equals the value of the arithmetic expression represented by T.

BASIS. For the basis, T consists of a single node. That is, the argument n is a (pointer to a) leaf. Since the `op` field has the value `'i'` when the node represents an operand, the test of line (1) in Fig. 5.24 succeeds, and the value of that operand is returned at line (2).

INDUCTION. Suppose the node n is not a (pointer to a) leaf. The inductive hypothesis is that $S(T')$ is true for each tree T' rooted at one of the children of n. We must use this reasoning to prove $S(T)$ for the tree T rooted at n.

Since our operators are assumed to be binary, n has two subtrees. By the inductive hypothesis, the values of `val1` and `val2` computed at lines (3) and (4) respectively, are the values of the left and right subtrees. Figure 5.25 suggests these two subtrees; `val1` holds the value of T_1 and `val2` holds the value of T_2.

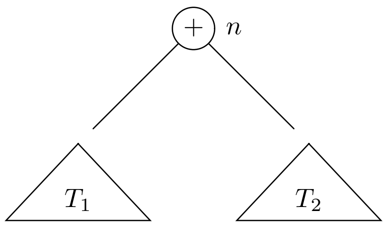


Fig. 5.25. The call $eval(n)$ returns the sum of the values of T_1 and T_2.

If we examine the switch statement of lines (5) through (9), we see that whatever operator appears at the root n is applied to the two values $val1$ and $val2$. For example, if the root holds `+`, as in Fig. 5.25, then at line (5) the value returned is $val1 + val2$, as it should be for an expression that is the sum of the expressions of trees T_1 and T_2. We have now completed the inductive step.

We conclude that $S(T)$ holds for all expression trees T, and, therefore, the function `eval` correctly evaluates trees that represent expressions.

Example 5.18. Now let us consider the function `computeHt` of Fig. 5.22, the body of which we reproduce as Fig. 5.26. This function takes as argument a (pointer to a) node n and computes the height of n. We shall prove the following statement by structural induction:

```
(1)            n->height = 0;
(2)            c = n->leftmostChild;
(3)            while (c != NULL) {
(4)                computeHt(c);
(5)                if (c->height >= n->height)
(6)                    n->height = 1+c->height;
(7)                c = c->rightSibling;
               }
```

Fig. 5.26. The body of the function `computeHt(n)` from Fig. 5.22.

STATEMENT $S(T)$: When `computeHt` is called on a pointer to the root of tree T, the correct height of each node in T is stored in the `height` field of that node.

BASIS. If the tree T is a single node n, then at line (2) of Fig. 5.26, c will be given the value `NULL`, since n has no children. Thus, the test of line (3) fails immediately, and the body of the while-loop is never executed. Since line (1) sets `n->height` to 0, which is the correct value for a leaf, we conclude that $S(T)$ holds when T has a single node.

INDUCTION. Now suppose n is the root of a tree T that is not a single node. Then n has at least one child. We may assume by the inductive hypothesis that when `computeHt(c)` is called at line (4), the correct height is installed in the `height` field of each node in the subtree rooted at c, including c itself. We need to show that the while-loop of lines (3) through (7) correctly sets `n->height` to 1 more than the maximum of the heights of the children of n. To do so, we need to perform another induction, which is nested "inside" the structural induction, just as one loop might be nested within another loop in a program. This induction is an "ordinary" induction, not a structural induction, and its statement is

STATEMENT $S'(i)$: After the loop of lines (3) to (7) has been executed i times, the value of `n->height` is 1 more than the largest of the heights of the first i children of n.

BASIS. The basis is $i = 1$. Since `n->height` is set to 0 outside the loop — at line (1) — and surely no height can be less than 0, the test of line (5) will be satisfied. Line (6) sets `n->height` to 1 more than the height of its first child.

INDUCTION. Assume that $S'(i)$ is true. That is, after i iterations of the loop, `n->height` is 1 larger than the largest height among the first i children. If there is an $(i + 1)$st child, then the test of line (3) will succeed and we execute the body an $(i + 1)$st time. The test of line (5) compares the new height with the largest of the previous heights. If the new height, `c->height`, is less than 1 plus the largest of the first i heights, no change to `n->height` will be made. That is correct, since the maximum height of the first $i+1$ children is the same as the maximum height of the first i children. However, if the new height is greater than the previous maximum,

A Template for Structural Induction

The following is an outline for building correct structural inductions.

1. Specify the statement $S(T)$ to be proved, where T is a tree.
2. Prove the basis, that $S(T)$ is true whenever T is a tree with a single node.
3. Set up the inductive step by letting T be a tree with root r and $k \geq 1$ subtrees, $T_1, T_2, \ldots, T_k$. State that you assume the inductive hypothesis: that $S(T_i)$ is true for each of the subtrees T_i, $i = 1, 2, \ldots, k$.
4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

then the test of line (5) will succeed, and `n->height` is set to 1 more than the height of the $(i+1)$st child, which is correct.

We can now return to the structural induction. When the test of line (3) fails, we have considered all the children of n. The inner induction, $S'(i)$, tells us that when i is the total number of children, `n->height` is 1 more than the largest height of any child of n. That is the correct height for n. The inductive hypothesis S applied to each of the children of n tells us that the correct height has been stored in each of their `height` fields. Since we just saw that n's height has also been correctly computed, we conclude that all the nodes in T have been assigned their correct height.

We have now completed the inductive step of the structural induction, and we conclude that `computeHt` correctly computes the height of each node of every tree on which it is called.

Why Structural Induction Works

The explanation for why structural induction is a valid proof method is similar to the reason ordinary inductions work: if the conclusion were false, there would be a smallest counterexample, and that counterexample would violate either the basis or the induction. That is, suppose there is a statement $S(T)$ for which we have proved the basis and the structural induction step, yet there are one or more trees for which S is false. Let T_0 be a tree such that $S(T_0)$ is false, but let T_0 have as few nodes as any tree for which S is false.

There are two cases. First, suppose that T_0 consists of a single node. Then $S(T_0)$ is true by the basis, and so this case cannot occur.

The only other possibility is that T_0 has more than one node — say, m nodes — and therefore T_0 consists of a root r with one or more children. Let the trees rooted at these children be $T_1, T_2, \ldots, T_k$. We claim that none of $T_1, T_2, \ldots, T_k$ can have more than $m-1$ nodes. For if one — say T_i — did have m or more nodes, then T_0, which consists of T_i and the root node r, possibly along with other subtrees, would have at least $m+1$ nodes. That contradicts our assumption that T_0 has exactly m nodes.

Now since each of the subtrees $T_1, T_2, \ldots, T_k$ has $m-1$ or fewer nodes, we know that these trees cannot violate S, because we chose T_0 to be as small as any

A Relationship between Structural and Ordinary Induction

There is a sense in which structural induction really offers nothing new. Suppose we have a statement $S(T)$ about trees that we want to prove by structural induction. We could instead prove

STATEMENT $S'(i)$: For all trees T of i nodes, $S(T)$ is true.

$S'(i)$ has the form of an ordinary induction on the integer i, with basis $i = 1$. It can be proved by complete induction, where we assume $S'(j)$ for all $j \leq i$, and prove $S'(i+1)$. This proof, however, would look exactly like the proof of $S(T)$, if we let T stand for an arbitrary tree of $i+1$ nodes.

tree making S false. Thus, we know that $S(T_1)$, $S(T_2), \ldots, S(T_k)$ are all true. The inductive step, which we assume proved, tells us that $S(T_0)$ is also true. Again we contradict the assumption that T_0 violates S.

We have considered the two possible cases, a tree of one node or a tree with more than one node, and have found that in either case, T_0 cannot be a violation of S. Therefore, S has no violations, and $S(T)$ must be true for all trees T.

EXERCISES

5.5.1: Prove by structural induction that

a) The preorder traversal function of Fig. 5.15 prints the labels of the tree in preorder.
b) The postorder function in Fig. 5.17 lists the labels in postorder.

5.5.2*: Suppose that a trie with branching factor b is represented by nodes in the format of Fig. 5.6. Prove by structural induction that if a tree T has n nodes, then there are $1 + (b-1)n$ `NULL` pointers among its nodes. How many non-`NULL` pointers are there?

Degree of a node

5.5.3*: The *degree* of a node is the number of children that node has.[2] Prove by structural induction that in any tree T, the number of nodes is 1 more than the sum of the degrees of the nodes.

5.5.4*: Prove by structural induction that in any tree T, the number of leaves is 1 more than the number of nodes that have right siblings.

5.5.5*: Prove by structural induction that in any tree T represented by the leftmost-child–right-sibling data structure, the number of `NULL` pointers is 1 more than the number of nodes.

5.5.6*: At the beginning of Section 5.2 we gave recursive and nonrecursive definitions of trees. Use a structural induction to show that every tree in the recursive sense is a tree in the nonrecursive sense.

[2] The branching factor and the degree are related concepts, but not the same. The branching factor is the maximum degree of any node in the tree.

A Fallacious Form of Tree Induction

It often is tempting to perform inductions on the number of nodes of the tree, where we assume a statement for n-node trees and prove it for $(n + 1)$-node trees. This proof will be fallacious if we are not very careful.

When doing inductions on integers in Chapter 2, we suggested the proper methodology, in which we try to prove statement $S(n + 1)$ by using $S(n)$; call this approach "leaning back." Sometimes one might be tempted to view this process as starting with $S(n)$ and proving $S(n + 1)$; call this approach "pushing out." In the integer case, these are essentially the same idea. However, with trees, we cannot start by assuming the statement for an n-node tree, add a node somewhere, and claim that the result is proved for all $(n + 1)$-node trees.

Danger: erroneous argument

For example, consider the claim $S(n)$: "all n-node trees have a path of length $n - 1$." It is surely true for the basis, $n = 1$. In a false "induction," we might argue: "Assume an n-node tree T has a path of length $n - 1$, say to node v. Add a child u to v. We now have an $(n + 1)$-node tree with a path of length n, proving the inductive step."

This argument is, of course, fallacious because it does not prove the result for all $(n+1)$-node trees, just some selected trees. A correct proof does not "push out" from n to $n + 1$ nodes, because we do not thus reach all possible trees. Rather, we must "lean back" by starting with an arbitrary $(n + 1)$-node tree and carefully selecting a node to remove to get an n-node tree.

5.5.7**: Show the converse of Exercise 5.5.6: every tree in the nonrecursive sense is a tree in the recursive sense.

5.6 Binary Trees

Left and right children

This section presents another kind of tree, called a *binary tree*, which is different from the "ordinary" tree introduced in Section 5.2. In a binary tree, a node can have at most two children, and rather than counting children from the left, there are two "slots," one for a *left child* and the other for a *right child.* Either or both slots may be empty.

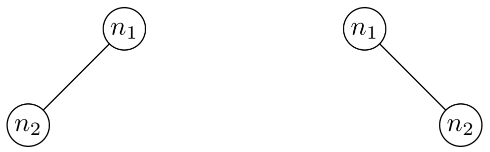


Fig. 5.27. The two binary trees with two nodes.

Example 5.19. Figure 5.27 shows two binary trees. Each has node n_1 as root. The first has n_2 as the left child of the root and no right child. The second has no left child, and has n_2 as the right child of the root. In both trees, n_2 has neither a left nor a right child. These are the only binary trees with two nodes.

We shall define binary trees recursively, as follows.

BASIS. The empty tree is a binary tree.

INDUCTION. If r is a node, and T_1 and T_2 are binary trees, then there is a binary tree with root r, left subtree T_1, and right subtree T_2, as suggested in Fig. 5.28. That is, the root of T_1 is the left child of r, unless T_1 is the empty tree, in which case r has no left child. Similarly, the root of T_2 is the right child of r, unless T_2 is empty, in which case r has no right child.

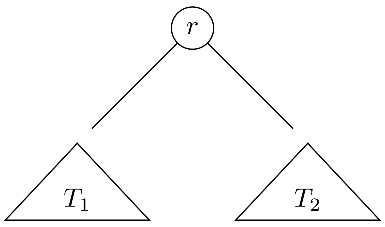


Fig. 5.28. Recursive construction of a binary tree.

Binary Tree Terminology

The notions of paths, ancestors, and descendants introduced in Section 5.2 also apply to binary trees. That is, left and right children are both regarded as "children." A path is still a sequence of nodes $m_1, m_2, \ldots, m_k$ such that m_{i+1} is a (left or right) child of m_i, for $i = 1, 2, \ldots, k-1$. This path is said to be from m_1 to m_k. The case $k = 1$ is permitted, where the path is just a single node.

The two children of a node, if they exist, are siblings. A leaf is a node with neither a left nor a right child; equivalently, a leaf is a node whose left and right subtrees are both empty. An interior node is a node that is not a leaf.

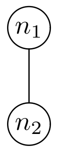
Path length, height, and depth are defined exactly as for ordinary trees. The length of a path in a binary tree is 1 less than the number of nodes; that is, the length is the number of parent-child steps along the path. The height of a node n is the length of the longest path from n to a descendant leaf. The height of a binary tree is the height of its root. The depth of a node n is the length of the path from the root to n.

Example 5.20. Figure 5.29 shows the five shapes that a binary tree of three nodes can have. In each binary tree in Fig. 5.29, n_3 is a descendant of n_1, and there is a path from n_1 to n_3. Node n_3 is a leaf in each tree, while n_2 is a leaf in the middle tree and an interior node in the other four trees.

The height of n_3 is 0 in each tree, while the height of n_1 is 2 in all but the middle tree, where the height of n_1 is 1. The height of each tree is the same as the height of n_1 in that tree. Node n_3 is of depth 2 in all but the middle tree, where it is of depth 1.

The Difference Between (Ordinary) Trees and Binary Trees

It is important to understand that while binary trees require us to distinguish whether a child is either a left child or a right child, ordinary trees require no such distinction. That is, binary trees are not just trees all of whose nodes have two or fewer children. Not only are the two trees in Fig. 5.27 different from each other, but they have no relation to the ordinary tree consisting of a root and a single child of the root:



Empty tree

There is another technical difference. While trees are defined to have at least one node, it is convenient to include the *empty tree*, the tree with no nodes, among the binary trees.

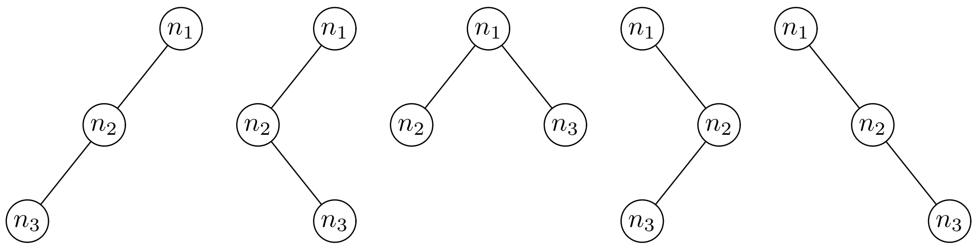


Fig. 5.29. The five binary trees with three nodes.

Data Structures for Binary Trees

There is one natural way to represent binary trees. Nodes are represented by records with two fields, `leftChild` and `rightChild`, pointing to the left and right children of the node, respectively. A `NULL` pointer in either of these fields indicates that the corresponding left or right subtree is missing — that is, that there is no left or right child, respectively.

A binary tree can be represented by a pointer to its root. The empty binary tree is represented naturally by `NULL`. Thus, the following type declarations represent binary trees:

```
typedef struct NODE *TREE;
struct NODE {
    TREE leftChild, rightChild;
};
```

Here, we call the type "pointer to node" by the name `TREE`, since the most common use for this type will be to represent trees and subtrees. We can interpret the `leftChild` and `rightChild` fields either as pointers to the children or as the left and right subtrees themselves.

Optionally, we can add to the structure for `NODE` a label field or fields, and/or we can add a pointer to the parent. Note that the type of the parent pointer is `*NODE`, or equivalently `TREE`.

Recursions on Binary Trees

There are many natural algorithms on binary trees that can be described recursively. The scheme for recursions is more limited than was the scheme of Fig. 5.13 for ordinary trees, since actions can only occur either before the left subtree is explored, between the exploration of the subtrees, or after both have been explored. The scheme for recursions on binary trees is suggested by Fig. 5.30.

```
{
    action $A_0$;
    recursive call on left subtree;
    action $A_1$;
    recursive call on right subtree;
    action $A_2$;
}
```

Fig. 5.30. Template of a recursive algorithm on a binary tree.

Example 5.21. Expression trees with binary operators can be represented by binary trees. These binary trees are special, because nodes have either two children or none. (Binary trees in general can have nodes with one child.) For instance, the expression tree of Fig. 5.14, reproduced here as Fig. 5.31, can be thought of as a binary tree.

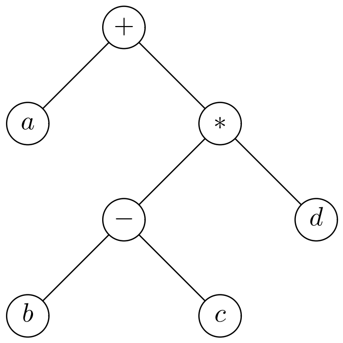


Fig. 5.31. The expression $a + (b - c) * d$ represented by a binary tree.

Suppose we use the type

```
typedef struct NODE *TREE;
struct NODE {
    char nodeLabel;
    TREE leftChild, rightChild;
};
```

```
        void preorder(TREE t)
        {
(1)         if (t != NULL) {
(2)             printf("%c\n", t->nodeLabel);
(3)             preorder(t->leftChild);
(4)             preorder(t->rightChild);
            }
        }
```

Fig. 5.32. Preorder listing of binary trees.

for nodes and trees. Then Fig. 5.32 shows a recursive function that lists the labels of the nodes of a binary tree T in preorder.

The behavior of this function is similar to that of the function of the same name in Fig. 5.15 that was designed to work on ordinary trees. The significant difference is that when the function of Fig. 5.32 comes to a leaf, it calls itself on the (missing) left and right children. These calls return immediately, because when t is `NULL`, none of the body of the function except the test of line (1) is executed. We could save the extra calls if we replaced lines (3) and (4) of Fig. 5.32 by

```
(3)  if (t->leftChild != NULL) preorder(t->leftChild);
(4)  if (t->rightChild != NULL) preorder(t->rightChild);
```

However, that would not protect us against a call to `preorder` from another function, with `NULL` as the argument. Thus, we would have to leave the test of line (1) in place for safety.

EXERCISES

5.6.1: Write a function that prints an inorder listing of the (labels of the) nodes of a binary tree. Assume that the nodes are represented by records with left-child and right-child pointers, as described in this section.

5.6.2: Write a function that takes a binary expression tree and prints a fully parenthesized version of the represented expression. Assume the same data structure as in Exercise 5.6.1.

5.6.3*: Repeat Exercise 5.6.2 but print only the needed parentheses, assuming the usual precedence and associativity of arithmetic operators.

5.6.4: Write a function that produces the height of a binary tree.

Full binary tree

5.6.5: Define a node of a binary tree to be a *full* if it has both a left and a right child. Prove by structural induction that the number of full nodes in a binary tree is 1 fewer than the number of leaves.

5.6.6: Suppose we represent a binary tree by the left-child, right-child record type. Prove by structural induction that the number of `NULL` pointers is 1 greater than the number of nodes.

Inorder Traversals

In addition to preorder and postorder listings of binary trees, there is another ordering of nodes that makes sense for binary trees only. An *inorder* listing of the nodes of a binary tree is formed by listing each node after exploring the left subtree, but before exploring the right subtree (i.e., in the position for action A_1 of Fig. 5.30). For example, on the tree of Fig. 5.31, the inorder listing would be $a + b - c * d$.

A preorder traversal of a binary tree that represents an expression produces the prefix form of that expression, and a postorder traversal of the same tree produces the postfix form of the expression. The inorder traversal almost produces the ordinary, or infix, form of an expression, but the parentheses are missing. That is, the tree of Fig. 5.31 represents the expression $a + (b - c) * d$, which is not the same as the inorder listing, $a + b - c * d$, but only because the necessary parentheses are missing from the latter.

To be sure that needed parentheses are present, we could parenthesize all operators. In this modified inorder traversal, action A_0, the step performed before exploring the left subtree, checks whether the label of the node is an operator and, if so, prints `'('`, a left parenthesis. Similarly, action A_2, performed after exploring both subtrees, prints a right parenthesis, `')'`, if the label is an operator. The result, applied to the binary tree of Fig. 5.31, would be $\Bigl(a + \bigl((b - c) * d\bigr)\Bigr)$, which has the needed pair of parentheses around $b - c$, along with two pairs of parentheses that are redundant.

5.6.7**: Trees can be used to represent recursive calls. Each node represents a recursive call of some function F, and its children represent the calls made by F. In this exercise, we shall consider the recursion for $\binom{n}{m}$ given in Section 4.5, based on the recursion $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$. Each call can be represented by a binary tree. If a node corresponds to the computation of $\binom{n}{m}$, and the basis cases ($m = 0$ and $m = n$) do not apply, then the left child represents $\binom{n-1}{m}$ and the left child represents $\binom{n-1}{m-1}$. If the node represents a basis case, then it has neither left nor right child.

a) Prove by structural induction that a binary tree with root corresponding to $\binom{n}{m}$ has exactly $2\binom{n}{m} - 1$ nodes.

b) Use (a) to show that the running time of the recursive algorithm for $\binom{n}{m}$ is $O\Bigl(\binom{n}{m}\Bigr)$. Note that this running time is therefore also $O(2^n)$, but the latter is a smooth-but-not-tight bound.

5.7 Binary Search Trees

A common activity found in a variety of computer programs is the maintenance of a set of values from which we wish to

Structural Inductions on Binary Trees

A structural induction can be applied to a binary tree as well as to an ordinary tree. There is, in fact, a somewhat simpler scheme to use, in which the basis is an empty tree. Here is a summary of the technique.

1. Specify the statement $S(T)$ to be proved, where T is a binary tree.
2. Prove the basis, that $S(T)$ is true if T is the empty tree.
3. Set up the inductive step by letting T be a tree with root r and subtrees T_L and T_R. State that you assume the inductive hypothesis: that $S(T_L)$ and $S(T_R)$ are true.
4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

1. Insert elements into the set,
2. Delete elements from the set, and
3. Look up an element to see whether it is currently in the set.

One example is a dictionary of English words, where from time to time we insert a new word, such as `fax`, delete a word that has fallen into disuse, such as `aegilops`, or look up a string of letters to see whether it is a word (as part of a spelling-checker program, for instance).

Dictionary

Because this example is so familiar, a set upon which we can execute the operations *insert*, *delete*, and *lookup*, as defined above, is called a *dictionary,* no matter what the set is used for. As another example of a dictionary, a professor might keep a roll of the students in a class. Occasionally, a student will be added to the class (an insert), or will drop the class (a delete), or it will be necessary to tell whether a certain student is registered for the class (a lookup).

One good way to implement a dictionary is with a binary search tree, which is a kind of labeled binary tree. We assume that the labels of nodes are chosen from a set with a "less than" order, which we shall write as $<$. Examples include the reals or integers with the usual less than order; or character strings, with the lexicographic or alphabetic order represented by $<$.

Binary search tree property

A *binary search tree* (*BST)* is a labeled binary tree in which the following property holds at every node x in the tree: all nodes in the left subtree of x have labels less than the label of x, and all nodes in the right subtree have labels greater than the label of x. This property is called the *binary search tree property.*

Example 5.22. Figure 5.33 shows a binary search tree for the set

{`Hairy, Bashful, Grumpy, Sleepy, Sleazy, Happy`}

where the $<$ order is lexicographic. Note that the names in the left subtree of the root are all lexicographically less than `Hairy`, while those in the right subtree are all lexicographically greater. This property holds at every node of the tree.

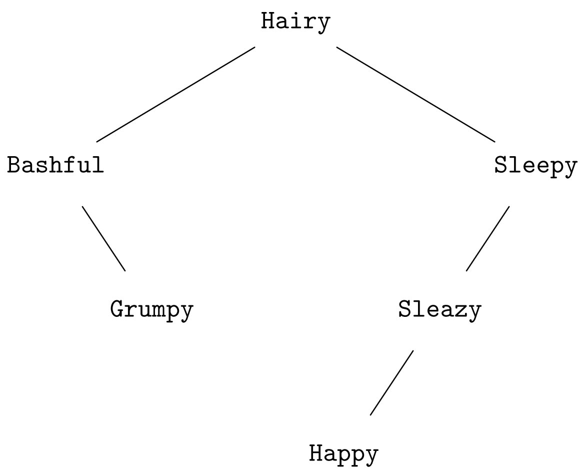


Fig. 5.33. Binary search tree with six nodes labeled by strings.

Implementation of a Dictionary as a Binary Search Tree

We can represent a binary search tree as any labeled binary tree. For example, we might define the type `NODE` by

```
typedef struct NODE *TREE;
struct NODE {
    ETYPE element;
    TREE leftChild, rightChild;
};
```

A binary search tree is represented by a pointer to the root node of the binary search tree. The type of an element, `ETYPE`, should be set appropriately. Throughout the programs of this section, we shall assume `ETYPE` is `int` so that comparisons between elements can be done simply using the arithmetic comparison operators `<`, `==` and `>`. In the examples involving lexicographic comparisons, we assume the comparisons in the programs will be done by the appropriate comparison functions *lt*, *eq*, and *gt* as discussed in Section 2.2.

Looking Up an Element in a Binary Search Tree

Suppose we want to look for an element x that may be in a dictionary represented by a binary search tree T. If we compare x with the element at the root of T, we can take advantage of the BST property to locate x quickly or determine that x is not present. If x is at the root, we are done. Otherwise, if x is less than the element at the root, x could be found only in the left subtree (by the BST property); and if x is greater, then it could be only in the right subtree (again, because of the BST property). That is, we can express the *lookup* operation by the following recursive algorithm.

BASIS. If the tree T is empty, then x is not present. If T is not empty, and x appears at the root, then x is present.

Abstract Data Types

A collection of operations, such as *insert*, *delete*, and *lookup*, that may be performed on a set of objects or a certain kind is sometimes called an *abstract data type* or ADT. The concept is also variously called a *class,* or a *module.* We shall study several abstract data types in Chapter 7, and in this chapter, we shall see one more, the priority queue.

An ADT can have more than one abstract implementation. For example, we shall see in this section that the binary search tree is a good way to implement the dictionary ADT. Lists are another plausible, though usually less efficient, way to implement the dictionary ADT. Section 7.6 covers hashing, another good implementation of the dictionary.

Each abstract implementation can, in turn, be implemented concretely by several different data structures. As an example, we shall use the left-child–right-child implementation of binary trees as a data structure implementing a binary search tree. This data structure, along with the appropriate functions for *insert*, *delete*, and *lookup*, becomes an implementation of the dictionary ADT.

An important reason for using ADT's in programs is that the data underlying the ADT is accessible only through the operations of the ADT, such as *insert*. This restriction is a form of defensive programming, protecting against accidental alteration of data by functions that manipulate the data in unexpected ways. A second important reason for using ADT's is that they allow us to redesign the data structures and functions implementing their operations, perhaps to improve the efficiency of operations, without having to worry about introducing errors into the rest of the program. There can be no new errors if the only interface to the ADT is through correctly rewritten functions for its operations.

INDUCTION. If T is not empty but x is not at the root, let y be the element at the root of T. If $x < y$ look up x only in the left subtree of the root, and if $x > y$ look up x only in the right subtree of y. The BST property guarantees that x cannot be in the subtree we do not search.

Example 5.23. Suppose we want to look up `Grumpy` in the binary search tree of Fig. 5.33. We compare `Grumpy` with `Hairy` at the root and find that `Grumpy` precedes `Hairy` in lexicographic order. We thus call `lookup` on the left subtree.

The root of the left subtree is `Bashful`, and we compare this label with `Grumpy`, finding that the former precedes the latter. We thus call `lookup` recursively on the right subtree of `Bashful`. Now we find `Grumpy` at the root of this subtree and return `TRUE`. These steps would be carried out by a function modeled after Fig. 5.34 that dealt with lexicographic comparisons.

More concretely, the recursive function `lookup(x,T)` in Fig. 5.34 implements this algorithm, using the left-child–right-child data structure. Note that `lookup` returns a value of type `BOOLEAN`, which is a defined type synonymous with `int`, but with the intent that only defined values `TRUE` and `FALSE`, defined to be 1 and 0, respectively, will be used. Type `BOOLEAN` was introduced in Section 1.6. Also, note that `lookup` is written only for types that can be compared by `=`, `<`, and so on. It would require rewriting for data like the character strings used in Example 5.23.

At line (1), `lookup` determines whether T is empty. If not, then at line (3) `lookup` determines whether x is stored at the current node. If x is not there, then `lookup` recursively searches the left subtree or right subtree depending on whether x is less than or greater than the element stored at the current node.

```
        BOOLEAN lookup(ETYPE x, TREE T)
        {
(1)         if (T == NULL)
(2)             return FALSE;
(3)         else if (x == T->element)
(4)             return TRUE;
(5)         else if (x < T->element)
(6)             return lookup(x, T->leftChild);
            else /* x must be > T->element */
(7)             return lookup(x, T->rightChild);
        }
```

Fig. 5.34. Function `lookup(x,T)` returns `TRUE` if x is in T, `FALSE` otherwise.

Inserting an Element into a Binary Search Tree

Adding a new element x to a binary search tree T is straightforward. The following recursive algorithm sketches the idea:

BASIS. If T is an empty tree, replace T by a tree consisting of a single node and place x at that node. If T is not empty and its root has element x, then x is already in the dictionary, and we do nothing.

INDUCTION. If T is not empty and does not have x at its root, then insert x into the left subtree if x is less than the element at the root, or insert x into the right subtree if x is greater than the element at the root.

The function `insert(x,T)` shown in Fig. 5.35 implements this algorithm for the left-child–right-child data structure. When we find that the value of T is `NULL` at line (1), we create a new node, which becomes the tree T. This tree is created by lines (2) through (5) and returned at line (10).

If x is not found at the root of T, then, at lines (6) through (9), `insert` is called on the left or right subtree, whichever is appropriate. The subtree, modified by the insertion, becomes the new value of the left or right subtree of the root of T at lines (7) or (9), respectively. Line (10) returns the augmented tree.

Notice that if x is at the root of T, then none of the tests of lines (1), (6), and (8) succeed. In this case, `insert` returns T without doing anything, which is correct, since x is already in the tree.

Example 5.24. Let us continue with Example 5.23, understanding that technically, the comparison of character strings requires slightly different code from that of Fig. 5.35, in which arithmetic comparisons like `<` are replaced by calls to suitably defined functions like *lt*. Figure 5.36 shows the binary search tree of Fig. 5.33

```
         TREE insert(ETYPE x, TREE T)
         {
 (1)         if (T == NULL) {
 (2)             T = (TREE) malloc(sizeof(struct NODE));
 (3)             T->element = x;
 (4)             T->leftChild = NULL;
 (5)             T->rightChild = NULL;
             }
 (6)         else if (x < T->element)
 (7)             T->leftChild = insert(x, T->leftChild);
 (8)         else if (x > T->element)
 (9)             T->rightChild = insert(x, T->rightChild);
(10)         return T;
         }
```

Fig. 5.35. Function `insert(x,T)` adds x to T.

after we insert `Filthy`. We begin by calling `insert` at the root, and we find that `Filthy` < `Hairy`. Thus, we call `insert` on the left child, at line (7) of Fig. 5.35. The result is that we find `Filthy` > `Bashful`, and so we call `insert` on the right child, at line (9). That takes us to `Grumpy`, which follows `Filthy` in lexicographic order, and we call `insert` on the left child of `Grumpy`.

The pointer to the left child of `Grumpy` is `NULL`, so at line (1) we discover that we must create a new node. This one-node tree is returned to the call of `insert` at the node for `Grumpy`, and the tree is installed as the value of the left child of `Grumpy` at line (7). The modified tree with `Grumpy` and `Filthy` is returned to the call of `insert` at the node labeled `Bashful`, and this modified tree becomes the right child of `Bashful`. Then, continuing up the tree, the new tree rooted at `Bashful` becomes the left child of the root of the entire tree. The final tree is shown in Fig. 5.36.

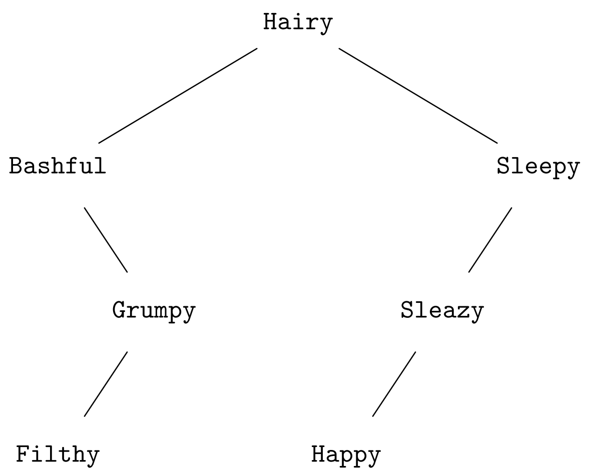


Fig. 5.36. Binary search tree after inserting `Filthy`.

Deleting an Element from a Binary Search Tree

Deleting an element x from a binary search tree is a little more complicated than *lookup* or *insert*. To begin, we may locate the node containing x; if there is no such node, we are done, since x is not in the tree to begin with. If x is at a leaf, we can simply delete the leaf. If x is at an interior node n, however, we cannot delete that node, because to do so would disconnect the tree.

We must rearrange the tree in some way so that the BST property is maintained and yet x is no longer present. There are two cases. First, if n has only one child, we can replace n by that child, and the BST property will be maintained.

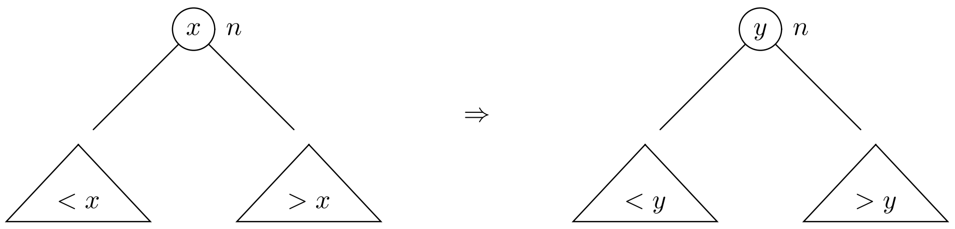


Fig. 5.37. To delete x, remove the node containing y, the smallest element in the right subtree, and then replace the label x by y at node n.

Second, suppose n has both children present. One strategy is to find the node m with label y, the smallest element in the right subtree of n, and replace x by y in node n, as suggested by Fig. 5.37. We can then remove node m from the right subtree.

The BST property continues to hold. The reason is that x is greater than everything in the left subtree of n, and so y, being greater than x (because y is in the right subtree of n), is also greater than everything in the left subtree of n. Thus, as far as the left subtree of n is concerned, y is a suitable element at n. As far as the right subtree of n is concerned, y is also suitable as the root, because y was chosen to be the smallest element in the right subtree.

```
ETYPE deletemin(TREE *pT)
{
    ETYPE min;

    if ((*pT)->leftChild == NULL) {
        min = (*pT)->element;
        (*pT) = (*pT)->rightChild;
        return min;
    }
    else
        return deletemin(&((*pT)->leftChild));
}
```

Fig. 5.38. Function `deletemin(pT)` removes and returns the smallest element from T.

It is convenient to define a function `deletemin(pT)`, shown in Fig. 5.38, to remove the node containing the smallest element from a nonempty binary search tree and to return the value of that smallest element. We pass to the function an argument that is the address of the pointer to the tree `T`. All references to `T` in the function are done indirectly through this pointer.

Call by reference

This style of tree manipulation, where we pass the function an argument that is a pointer to a place where a pointer to a node (i.e., a tree) is found, is called *call by reference*. It is essential in Fig. 5.38 because at line (3), where we have found a pointer to a node m whose left child is `NULL`, we wish to replace this pointer by another pointer — the pointer in the `rightChild` field of m. If the argument of *deletemin* were a pointer to a node, then the change would take place locally to the call to deletemin, and there would not actually be a change to the pointers in the tree itself. Incidentally, we could use the call-by-reference style to implement *insert* as well. In that case, we could modify the tree directly and not have to return a revised tree as we did in Fig. 5.35. We leave such a revised *insert* function as an exercise.

Now, let us see how Fig. 5.38 works. We locate the smallest element by following left children until we find a node whose left child is `NULL` at line (1) of Fig. 5.38. The element y at this node m must be the smallest in the subtree. To see why, first observe that y is smaller than the element at any ancestor of m in the subtree, because we have followed only left children. The only other nodes in the subtree are either in the right subtree of m, in which case their elements are surely larger than y by the BST property, or in the right subtree of one of m's ancestors. But elements in the right subtrees are greater than the element at some ancestor of m, and therefore greater than y, as suggested by Fig. 5.39.

Having found the smallest element in the subtree, we record this value at line (2), and at line (3) we replace the node of the smallest element by its right subtree. Note that when we delete the smallest element from the subtree, we always have the easy case of deletion, because there is no left subtree.

The only remaining point regarding `deletemin` is that when the test of line (1) fails, meaning that we are not yet at the smallest element, we proceed to the left child. That step is accomplished by the recursive call at line (5).

The function `delete(x,pT)` is shown in Fig. 5.40. If `pT` points to an empty tree T, there is nothing to do, and the test of line (1) makes sure that nothing is done. Otherwise, the tests of lines (2) and (4) handle the cases where x is not at the root, and we are directed to the left or right subtree, as appropriate. If we reach line (6), then x must be at the root of T, and we must replace the root node. Line (6) tests for the possibility that the left child is `NULL`, in which case we simply replace T by its right subtree at line (7). Similarly, if at line (8) we find that the right child is `NULL` then at line (9) we replace T by its left subtree. Note that if both children of the root are `NULL`, then we replace T by `NULL` at line (7).

The remaining case, where neither child is `NULL`, is handled at line (10). We call `deletemin`, which returns the smallest element, y, of the right subtree and also deletes y from that subtree. The assignment of line (10) replaces x by y at the root of T.

Example 5.25. Figure 5.41 shows what would happen if we used a function similar to `delete` (but able to compare character strings) to remove `Hairy` from the

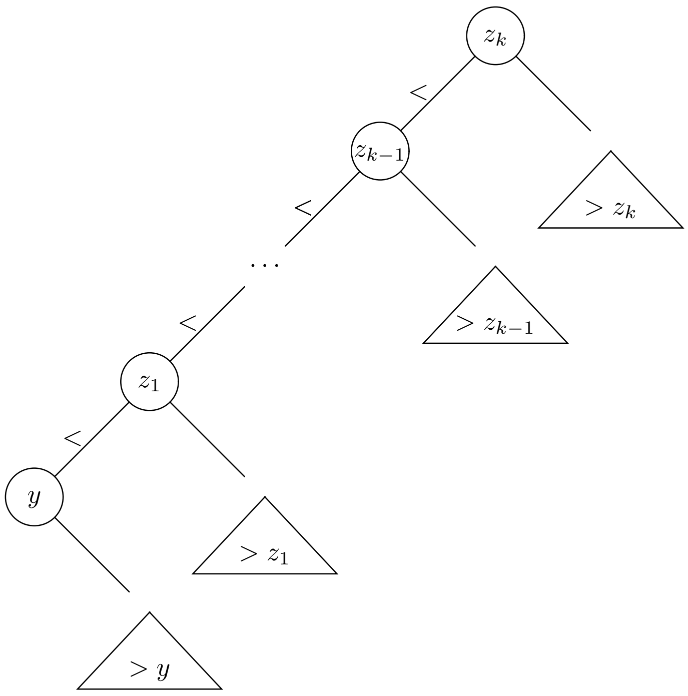


Fig. 5.39. All the other elements in the right subtree are greater than y.

```
void delete(ETYPE x, TREE *pT)
{
    if ((*pT) != NULL)
        if (x < (*pT)->element)
            delete(x, &((*pT)->leftChild));
        else if (x > (*pT)->element)
            delete(x, &((*pT)->rightChild));
        else /* here, x is at the root of (*pT) */
            if ((*pT)->leftChild == NULL)
                (*pT) = (*pT)->rightChild;
            else if ((*pT)->rightChild == NULL)
                (*pT) = (*pT)->leftChild;
            else /* here, neither child is NULL */
                (*pT)->element =
                        deletemin(&((*pT)->rightChild));
}
```

Fig. 5.40. Function `delete(x,pT)` removes the element x from T.

binary search tree of Fig. 5.36. Since `Hairy` is at a node with two children, `delete` calls the function `deletemin`, which removes and returns the smallest element, `Happy`, from the right subtree of the root. `Happy` then becomes the label of the root of the tree, the node at which `Hairy` was stored.

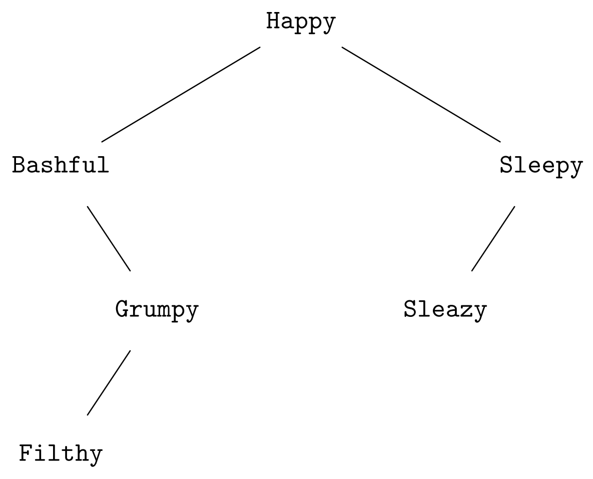


Fig. 5.41. Binary search tree after deleting Hairy.

EXERCISES

5.7.1: Suppose that we use a leftmost-child–right-sibling implementation for binary search trees. Rewrite the functions that implement the dictionary operations *insert*, *delete*, and *lookup* to work for this data structure.

5.7.2: Show what happens to the binary search tree of Fig. 5.33 if we insert the following dwarfs in order: `Doc`, `Dopey`, `Inky`, `Blinky`, `Pinky`, and `Sue`. Then show what happens when we delete in order: `Doc`, `Sleazy`, and `Hairy`.

5.7.3: Rewrite the functions `lookup`, `insert`, and `delete` to use lexicographic comparisons on strings instead of arithmetic comparisons on integers.

5.7.4*: Rewrite the function `insert` so that the tree argument is passed by reference.

5.7.5*: We wrote `delete` in the "call by reference" style. However, it is also possible to write it in a style like that of our `insert` function, where it takes a tree as argument (rather than a pointer to a tree) and returns the tree missing the deleted element. Write this version of the dictionary operation *delete*. *Note*: It is not really possible to have *deletemin* return a revised tree, since it must also return the minimum element. We could rewrite *deletemin* to return a structure with both the new tree and the minimum element, but that approach is not recommended.

5.7.6: Instead of handling the deletion of a node with two children by finding the least element in the right subtree, we could also find the greatest element in the left subtree and use that to replace the deleted element. Rewrite the functions `delete` and `deletemin` from Figs. 5.38 and 5.40 to incorporate this modification.

5.7.7*: Another way to handle *delete* when we need to remove the element at a node n that has parent p, (nonempty) left child l, and (nonempty) right child r is to find the node m holding the least element in the right subtree of n. Then, make r a left or right child of p, whichever n was, and make l the left child of m (note that

m cannot previously have had a left child). Show why this set of changes preserves the BST property. Would you prefer this strategy to the one described in Section 5.7? *Hint*: For both methods, consider their effect on the lengths of paths. As we shall see in the next section, short paths make the operations run fast.

5.7.8*: In this exercise, refer to the binary search tree represented in Fig. 5.39. Show by induction on i that if $1 \leq i \leq k$, then $y < z_i$. Then, show that y is the least element in the tree rooted at z_k.

5.7.9: Write a complete C program to implement a dictionary that stores integers. Accept commands of the form `x` i, where `x` is one of the letters `i` (insert), `d` (delete), and `l` (lookup). Integer i is the argument of the command, the integer to be inserted, deleted, or searched for.

5.8 Efficiency of Binary Search Tree Operations

The binary search tree provides a reasonably fast implementation of a dictionary. First, notice that each of the operations *insert*, *delete*, and *lookup* makes a number of recursive calls equal to the length of the path followed (but this path must include the route to the smallest element of the right subtree, in case `deletemin` is called). Also, a simple analysis of the functions `lookup`, `insert`, `delete`, and `deletemin` tells us that each operation takes $O(1)$ time, plus the time for one recursive call. Moreover, since this recursive call is always made at a child of the current node, the height of the node in each successive call decreases by at least 1.

Thus, if $T(h)$ is the time taken by any of these functions when called with a pointer to a node of height h, we have the following recurrence relation upper-bounding $T(h)$:

BASIS. $T(0) = O(1)$. That is, when called on a leaf, the call either terminates without further calls or makes a recursive call with a `NULL` argument and then returns without further calls. All of this work takes $O(1)$ time.

INDUCTION. $T(h) \leq T(h-1) + O(1)$ for $h \geq 1$. That is, the time taken by a call on any interior node is $O(1)$ plus the time for a recursive call, which is on a node of height at most $h-1$. If we make the reasonable assumption that $T(h)$ increases with increasing h, then the time for the recursive call is no greater than $T(h-1)$.

The solution to the recurrence for $T(h)$ is $O(h)$, as discussed in Section 3.9. Thus, the running time of each dictionary operation on a binary search tree of n nodes is at most proportional to the height of the tree. But what is the height of a typical binary search tree of n nodes?

The Worst Case

In the worst case, all the nodes in the binary tree will be arranged in a single path, like the tree of Fig. 5.42. That tree would result, for example, from taking a list of k elements in sorted order and inserting them one at a time into an initially empty tree. There are also trees in which the single path does not consist of right children only but is a mixture of right and left children, with the path taking a turn either left or right at each interior node.

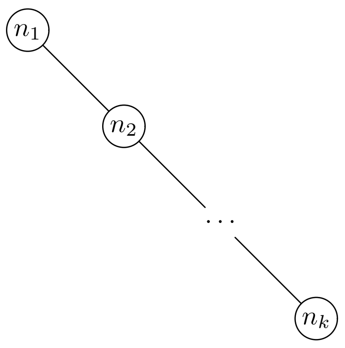


Fig. 5.42. A degenerate binary tree.

The height of a k-node tree like Fig. 5.42 is clearly $k - 1$. We thus expect that *lookup*, *insert*, and *delete* will take $O(k)$ time on a dictionary of k elements, if the representation of that dictionary happens to be one of these unfortunate trees. Intuitively, if we need to look for element x, on the average we shall find it halfway down the path, requiring us to look at $k/2$ nodes. If we fail to find x, we shall likewise have to search down the tree until we come to the place where x would be found, which will also be halfway down, on the average. Since each of the operations *lookup*, *insert*, and *delete* requires searching for the element involved, we know that these operations each take $O(k)$ time on the average, given one of the bad trees of the form of Fig. 5.42.

The Best Case

However, a binary tree need not grow long and thin like Fig. 5.42; it could be short and bushy like Fig. 5.43. A tree like the latter, where every interior node down to some level has both children present and the next level has all the leaves, is called a *full* or *complete* tree.

Complete tree

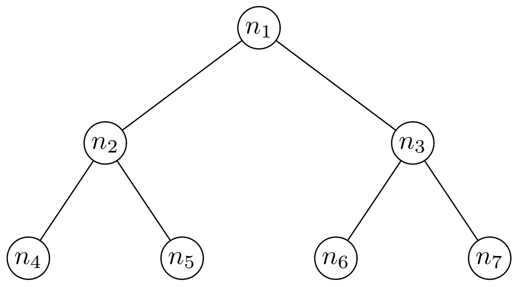


Fig. 5.43. Full binary tree with seven nodes.

A complete binary tree of height h has $2^{h+1} - 1$ nodes. We can prove this claim by induction on the height h.

BASIS. If $h = 0$, the tree consists of a single node. Since $2^{0+1} - 1 = 1$, the basis case holds.

INDUCTION. Suppose that a complete binary tree of height h has $2^{h+1}-1$ nodes, and consider a complete binary tree of height $h+1$. This tree consists of one node at the root and left and right subtrees that are complete binary trees of height h. For example, the height-2 complete binary tree of Fig. 5.43 consists of the root, n_1; a left subtree containing n_2, n_4, and n_5, which is a complete binary tree of height 1; and a right subtree consisting of the remaining three nodes, which is another complete binary tree of height 1. Now the number of nodes in two complete binary trees of height h is $2(2^{h+1}-1)$, by the inductive hypothesis. When we add the root node, we find that a complete binary tree of height $h+1$ has $2(2^{h+1}-1)+1 = 2^{h+2}-1$ nodes, which proves the inductive step.

Now we can invert this relationship and say that a complete binary tree of $k = 2^{h+1}-1$ nodes has height h. Equivalently, $k+1 = 2^{h+1}$. If we take logarithms, then $\log_2(k+1) = h+1$, or approximately, h is $O(\log k)$. Since the running time of *lookup*, *insert*, and *delete* is proportional to the height of the tree, we can conclude that on a complete binary tree, these operations take time that is logarithmic in the number of nodes. That performance is much better than the linear time taken for pessimal trees like Fig. 5.42. As the dictionary size becomes large, the running time of the dictionary operations grows much more slowly than the number of elements in the set.

The Typical Case

Is Fig. 5.42 or Fig. 5.43 the more common case? Actually, neither is common in practice, but the complete tree of Fig. 5.43 offers efficiency of the dictionary operations that is closer to what the typical case provides. That is, *lookup*, *insert*, and *delete* take logarithmic time, on the average.

A proof that the typical binary tree offers logarithmic time for the dictionary operations is difficult. The essential point of the proof is that the expected value of the length of a path from the root of such a tree to a random node is $O(\log n)$. A recurrence equation for this expected value is given in the exercises.

However, we can see intuitively why that should be the correct running time, as follows. The root of a binary tree divides the nodes, other than itself, into two subtrees. In the most even division, a k-node tree will have two subtrees of about $k/2$ nodes each. That case occurs if the root element happens to be exactly in the middle of the sorted list of elements. In the worst division, the root element is first or last among the elements of the dictionary and the division is into one subtree that is empty and another subtree of $k-1$ nodes.

On the average, we could expect the root element to be halfway between the middle and the extremes in the sorted list; and we might expect that, on the average, about $k/4$ nodes go into the smaller subtree and $3k/4$ nodes into the larger. Let us assume that as we move down the tree we always move to the root of the larger subtree at each recursive call and that similar assumptions apply to the distribution of elements at each level. At the first level, the larger subtree will be divided in the 1:3 ratio, leaving a largest subtree of $(3/4)(3k/4)$, or $9k/16$, nodes at the second level. Thus, at the dth-level, we would expect the largest subtree to have about $(3/4)^d k$ nodes.

When d becomes sufficiently large, the quantity $(3/4)^d k$ will be close to 1, and we can expect that, at this level, the largest subtree will consist of a single leaf.

Thus, we ask, For what value of d is $(3/4)^d k \leq 1$? If we take logarithms to the base 2, we get

$$d \log_2(3/4) + \log_2 k \leq \log_2 1 \tag{5.1}$$

Now $\log_2 1 = 0$, and the quantity $\log_2(3/4)$ is a negative constant, about -0.4. Thus we can rewrite (5.1) as $\log_2 k \leq 0.4d$, or $d \geq (\log_2 k)/0.4 = 2.5 \log_2 k$.

Put another way, at a depth of about two and a half times the logarithm to the base 2 of the number of nodes, we expect to find only leaves (or to have found the leaves at higher levels). This argument justifies, but does not prove, the statement that the typical binary search tree will have a height that is proportional to the logarithm of the number of nodes in the tree.

EXERCISES

5.8.1: If tree T has height h and branching factor b, what are the largest and smallest numbers of nodes that T can have?

5.8.2**: Perform an experiment in which we choose one of the $n!$ orders for n different values and insert the values in this order into an initially empty binary search tree. Let $P(n)$ be the expected value of the depth of the node at which a particular value v among the n values is found after this experiment.

a) Show that, for $n \geq 2$,

$$P(n) = 1 + \frac{2}{n^2} \sum_{k=1}^{n-1} kP(k)$$

b) Prove that $P(n)$ is $O(\log n)$.

5.9 Priority Queues and Partially Ordered Trees

So far, we have seen only one abstract data type, the dictionary, and one implementation for it, the binary search tree. In this section we shall study another abstract data type and one of its most efficient implementations. This ADT, called a *priority queue,* is a set of elements each of which has an associated *priority.* For example, the elements could be records and the priority could be the value of one field of the record. The two operations associated with the priority queue ADT are the following:

1. Inserting an element into the set (*insert*).
2. Finding and deleting from the set an element of highest priority (this combined operation is called *deletemax*). The deleted element is returned by this function.

Example 5.26. A time-shared operating system accepts requests for service from various sources, and these jobs may not all have the same priority. For example, at highest priority may be the system processes; these would include the "daemons" that watch for incoming data, such as the signal generated by a keystroke at a terminal or the arrival of a packet of bits over a local area network. Then may come user processes, the commands issued by ordinary users. Below these we may have certain background jobs such as backup of data to tape or long calculations that the user has designated to run with a low priority.

Jobs can be represented by records consisting of an integer ID for the job and an integer for the job's priority. That is, we might use the structure

```
struct ETYPE {
    int jobID;
    int priority;
};
```

for elements of a priority queue. When a new job is initiated, it gets an ID and a priority. We then execute the *insert* operation for this element on the priority queue of jobs waiting for service. When a processor becomes available, the system goes to the priority queue and executes the *deletemax* operation. The element returned by this operation is a waiting job of highest priority, and that is the one executed next.

Example 5.27. We can implement a sorting algorithm using the priority queue ADT. Suppose we are given the sequence of integers $a_1, a_2, \ldots, a_n$ to sort. We insert each into a priority queue, using the element's value as its priority. If we then execute *deletemax* n times, the integers will be selected highest first, or in the reverse of their sorted (lowest-first) order. We shall discuss this algorithm in more detail in the next section; it is known as heapsort.

Partially Ordered Trees

An efficient way to implement a priority queue is by a *partially ordered tree* (*POT*), which is a labeled binary tree with the following properties:

1. The labels of the nodes are elements with a "priority"; that priority may be the value of an element or the value of some component of an element.

2. The element stored at a node has at least as large a priority as the elements stored at the children of that node.

POT property

Property 2 implies that the element at the root of any subtree is always a largest element of that subtree. We call property 2 the *partially ordered tree property,* or *POT property.*

Example 5.28. Figure 5.44 shows a partially ordered tree with 10 elements. Here, as elsewhere in this section, we shall represent elements by their priorities, as if the element and the priority were the same thing. Note that equal elements can appear on different levels in the tree. To see that the POT property is satisfied at the root, note that 18, the element there, is no less than the elements 18 and 16 found at its children. Similarly, we can check that the POT property holds at every interior node. Thus, Fig. 5.44 is a partially ordered tree.

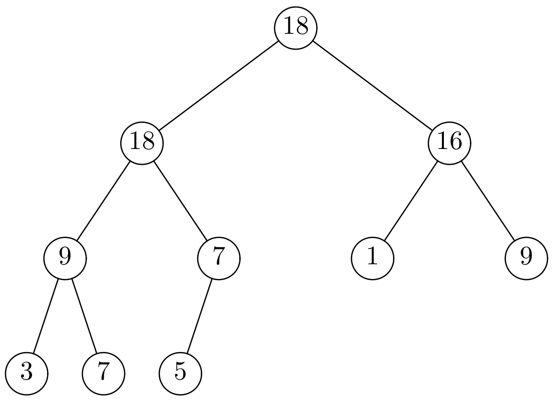


Fig. 5.44. Partially ordered tree with 10 nodes.

Partially ordered trees provide a useful abstract implementation for priority queues. Briefly, to execute *deletemax* we find the node at the root, which must be the maximum, and replace it by the rightmost node on the bottom level. However, when we do so, the POT property may be violated, and so we must restore that property by "bubbling down" the element newly placed at the root until it finds a suitable level where it is smaller than its parent but at least as large as any of its children. To execute *insert*, we can add a new leaf at the bottom level, as far left as possible, or at the left end of a new level if the bottom level is full. Again there may be a violation of the POT property, and if so, we "bubble up" the new element until it finds its rightful place.

Balanced POTs and Heaps

We say that a partially ordered tree is *balanced* if all possible nodes exist at all levels except the bottommost, and the leaves at the bottommost level are as far to the left as possible. This condition implies that if the tree has n nodes, then no path to a node from the root is longer than $\log_2 n$. The tree in Fig. 5.44 is a balanced POT.

Balanced POTs can be implemented using an array data structure called a *heap*, which provides a fast, compact implementation of the priority queue ADT. A heap is simply an array A with a special interpretation for the element indices. We start with the root in $A[1]$; $A[0]$ is not used. Following the root, the levels appear in order. Within a level, the nodes are ordered from left to right.

Thus, the left child of the root is in $A[2]$, and the right child of the root is in $A[3]$. In general, the left child of the node in $A[i]$ is in $A[2i]$ and the right child is in $A[2i + 1]$, if these children exist in the partially ordered tree. The balanced nature of the tree allows this representation. The POT property of the elements implies that if $A[i]$ has two children, then $A[i]$ is at least as large as $A[2i]$ and $A[2i + 1]$, and if $A[i]$ has one child, then $A[i]$ is at least as large as $A[2i]$.

1	2	3	4	5	6	7	8	9	10
18	18	16	9	7	1	9	3	7	5

Fig. 5.45. Heap for Fig. 5.44.

Layers of Implementation

It is useful to compare our two ADT's, the dictionary and the priority queue, and to notice that, in each case, we have given one abstract implementation and one data structure for that implementation. There are other abstract implementations for each, and other data structures for each abstract implementation. We promised to discuss other abstract implementations for the dictionary, such as the hash table, and in the exercises of Section 5.9 we suggest that the binary search tree may be a suitable abstract implementation for the priority queue. The table below summarizes what we already know about abstract implementations and data structures for the dictionary and the priority queue.

ADT	Abstract Implementation	Data Structure
dictionary	binary search tree	left-child–right-child structure
priority queue	balanced partially ordered tree	heap

Example 5.29. The heap for the balanced partially ordered tree in Fig. 5.44 is shown in Fig. 5.45. For instance, $A[4]$ holds the value 9; this array element represents the left child of the left child of the root in Fig. 5.44. The children of this node are found in $A[8]$ and $A[9]$. Their elements, 3 and 7, are each no greater than 9, as is required by the POT property. Array element $A[5]$, which corresponds to the right child of the left child of the root, has a left child in $A[10]$. It would have a right child in $A[11]$, but the partially ordered tree has only 10 elements at the moment, and so $A[11]$ is not part of the heap.

While we have shown tree nodes and array elements as if they were the priorities themselves, in principle an entire record appears at the node or in the array. As we shall see, we shall have to do much swapping of elements between children and parents in a partially ordered tree or its heap representation. Thus, it is considerably more efficient if the array elements themselves are pointers to the records representing the objects in the priority queue and these records are stored in another array "outside" the heap. Then we can simply swap pointers, leaving the records in place.

Performing Priority Queue Operations on a Heap

Throughout this section and the next, we shall represent a heap by a global array `A[1..MAX]` of integers. We assume that elements are integers and are equal to their priorities. When elements are records, we can store pointers to the records in the array and determine the priority of an element from a field in its record.

Suppose that we have a heap of $n-1$ elements satisfying the POT property, and we add an nth element in $A[n]$. The POT property continues to hold everywhere, except perhaps between $A[n]$ and its parent. Thus, if $A[n]$ is larger than $A[n/2]$, the element at the parent, we must swap these elements. Now there may be a violation of the POT property between $A[n/2]$ and its parent. If so, we recursively "bubble up" the new element until it either reaches a position where the parent has a larger element or reaches the root.

Bubbling up

The C function `bubbleUp` to perform this operation is shown in Fig. 5.46. It makes use of a function `swap(A,i,j)` that exchanges the elements in $A[i]$ and $A[j]$; this function is also defined in Fig. 5.46. The operation of `bubbleUp` is simple. Given argument i indicating the node that, with its parent, possibly violates the POT property, we test whether $i = 1$ (that is, whether we are already at the root, so that no POT violation can occur), and if not, whether the element $A[i]$ is greater than the element at its parent. If so, we swap $A[i]$ with its parent and recursively call `bubbleUp` at the parent.

```
void swap(int A[], int i, int j)
{
    int temp;

    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

void bubbleUp(int A[], int i)
{
    if (i > 1 && A[i] > A[i/2]) {
        swap(A, i, i/2);
        bubbleUp(A, i/2);
    }
}
```

Fig. 5.46. The function `swap` exchanges array elements, and the function `bubbleUp` pushes a new element of a heap into its rightful place.

Example 5.30. Suppose we start with the heap of Fig. 5.45 and we add an eleventh element, with priority 13. This element goes in $A[11]$, giving us the array

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	7	1	9	3	7	5	13

We now call `bubbleUp(A,11)`, which compares $A[11]$ with $A[5]$ and finds that we must swap these elements because $A[11]$ is larger. That is, $A[5]$ and $A[11]$ violate the POT property. Thus, the array becomes

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	13	1	9	3	7	5	7

Now we call `bubbleUp(A,5)`. This results in comparison of $A[2]$ and $A[5]$. Since $A[2]$ is larger, there is no POT violation, and `bubbleUp(A,5)` does nothing. We have now restored the POT property to the array.

Implementation of insert

We now show how to implement the priority queue operation *insert*. Let n be the current number of elements in the priority queue, and assume `A[1..n]` already satisfies the POT property. We increment n and then store the element to be inserted into the new `A[n]`. Finally, we call `bubbleUp(A,n)`. The code for *insert* is shown in Fig. 5.47. The argument x is the element to be inserted, and the argument *pn* is a pointer to the current size of the priority queue. Note that n must be passed by reference — that is, by a pointer to n — so that when n is incremented the change has an affect that is not local only to *insert*. A check that $n < MAX$ is omitted.

```
void insert(int A[], int x, int *pn)
{
    (*pn)++;
    A[*pn] = x;
    bubbleUp(A, *pn);
}
```

Fig. 5.47. Priority queue operation *insert* implemented on a heap.

To implement the priority queue operation *deletemax*, we need another operation on heaps or partially ordered trees, this time to bubble down an element at the root that may violate the POT property. Suppose that $A[i]$ is a potential violator of the POT property, in that it may be smaller than one or both of its children, $A[2i]$ and $A[2i+1]$. We can swap $A[i]$ with one of its children, but we must be careful which one. If we swap with the larger of the children, then we are sure not to introduce a POT violation between the two former children of $A[i]$, one of which has now become the parent of the other.

Bubbling down

The function `bubbleDown` of Fig. 5.48 implements this operation. After selecting a child with which to swap $A[i]$, it calls itself recursively to eliminate a possible POT violation between the element $A[i]$ in its new position — which is now `A[2i]` or `A[2i+1]` — and one of its new children. The argument n is the number of elements in the heap, or, equivalently, the index of the last element.

```
void bubbleDown(int A[], int i, int n)
{
    int child;

    child = 2*i;
    if (child < n && A[child+1] > A[child])
        ++child;
    if (child <= n && A[i] < A[child]) {
        swap(A, i, child);
        bubbleDown(A, child, n);
    }
}
```

Fig. 5.48. `bubbleDown` pushes a POT violator down to its proper position.

This function is a bit tricky. We have to decide which child of $A[i]$ to swap with, if any, and the first thing we do is assume that the larger child is $A[2i]$, at line (1) of Fig. 5.48. If the right child exists (i.e., $child < n$) and the right child is the larger, then the tests of line (2) are met and at line (3) we make *child* be the right child of $A[i]$.

Now at line (4) we test for two things. First, it is possible that $A[i]$ really has no children in the heap. We therefore check whether $A[i]$ is an interior node by asking whether $child \leq n$. The second test of line (4) is whether $A[i]$ is less than $A[child]$. If both these conditions are met, then at line (5) we swap $A[i]$ with its larger child, and at line (6) we recursively call `bubbleDown`, to push the offending element further down, if necessary.

Implementation of deletemax

We can use `bubbleDown` to implement the priority queue operation *deletemax* as shown in Fig. 5.49. The function `deletemax` takes as arguments an array A and a pointer *pn* to the number n that is the number of elements currently in the heap. We omit a test that $n > 0$.

In line (1), we swap the element at the root, which is to be deleted, with the last element, in `A[n]`. Technically, we should return the deleted element, but, as we shall see, it is convenient to put it in `A[n]`, which will no longer be part of the heap.

At line (2), we decrement n by 1, effectively deleting the largest element, now residing in the old `A[n]`. Since the root may now violate the POT property, we call `bubbleDown(A,1,n)` at line (3), which will recursively push the offending element down until it either reaches a point where it is no less than either of its children, or becomes a leaf; either way, there is no violation of the POT property.

```
     void deletemax(int A[], int *pn)
     {
(1)      swap(A, 1, *pn);
(2)      --(*pn);
(3)      bubbleDown(A, 1, *pn);
     }
```

Fig. 5.49. Priority queue operation *deletemax* implemented by a heap.

Example 5.31. Suppose we start with the heap of Fig. 5.45 and execute *deletemax*. After swapping $A[1]$ and $A[10]$, we set n to 9. The heap then becomes

1	2	3	4	5	6	7	8	9
5	18	16	9	7	1	9	3	7

When we execute `bubbleDown(A,1,9)`, we set *child* to 2. Since $A[2] \geq A[3]$, we do not increment *child* at line (3) of Fig. 5.48. Then since $child \leq n$ and $A[1] < A[2]$, we swap these elements, to obtain the array

1	2	3	4	5	6	7	8	9
18	5	16	9	7	1	9	3	7

We then call `bubbleDown(A,2,9)`. That requires us to compare $A[4]$ with $A[5]$ at line (2), and we find that the former is larger. Thus, $\mathit{child} = 4$ at line (4) of Fig. 5.48. When we find that $A[2] < A[4]$, we swap these elements and call `bubbleDown(A,4,9)` on the array

1	2	3	4	5	6	7	8	9
18	9	16	5	7	1	9	3	7

Next, we compare $A[8]$ and $A[9]$, finding that the latter is larger, so that $\mathit{child} = 9$ at line (4) of `bubbleDown(A,4,9)`. We again perform the swap, since $A[4] < A[9]$, resulting in the array

1	2	3	4	5	6	7	8	9
18	9	16	7	7	1	9	3	5

Next, we call `bubbleDown(A,9,9)`. We set *child* to 18 at line (1), and the first test of line (2) fails, because $\mathit{child} < n$ is false. Similarly, the test of line (4) fails, and we make no swap or recursive call. The array is now a heap with the POT property restored.

Running Time of Priority Queue Operations

The heap implementation of priority queues offers $O(\log n)$ running time per *insert* or *deletemax* operation. To see why, let us first consider the *insert* program of Fig. 5.47. This program evidently takes $O(1)$ time for the first two steps, plus whatever the call to `bubbleUp` takes. Thus, we need to determine the running time of `bubbleUp`.

Informally, we notice that each time `bubbleUp` calls itself recursively, we are at a node one position closer to the root. Since a balanced partially ordered tree has height approximately $\log_2 n$, the number of recursive calls is $O(\log_2 n)$. Since each call to `bubbleUp` takes time $O(1)$ plus the time of the recursive call, if any, the total time should be $O(\log n)$.

More formally, let $T(i)$ be the running time of `bubbleUp(A,i)`. Then we can create a recurrence relation for $T(i)$ as follows.

BASIS. If $i = 1$, then $T(i)$ is $O(1)$, since it is easy to check that the `bubbleUp` program of Fig. 5.46 does not make any recursive calls and only the test of the if-statement is executed.

INDUCTION. If $i > 1$, then the if-statement test may fail anyway, because $A[i]$ does not need to rise further. If the test succeeds, then we execute *swap*, which takes $O(1)$ time, plus a recursive call to `bubbleUp` with an argument $i/2$ (or slightly less if i is odd). Thus $T(i) \leq T(i/2) + O(1)$.

We thus have, for some constants a and b, the recurrence

$$T(1) = a$$
$$T(i) = T(i/2) + b \text{ for } i > 1$$

as an upper bound on the running time of `bubbleUp`. If we expand $T(i/2)$ we get

$$T(i) = T(i/2^j) + bj \tag{5.2}$$

for each j. As in Section 3.10, we choose the value of j that makes $T(i/2^j)$ simplest. In this case, we make j equal to $\log_2 i$, so that $i/2^j = 1$. Thus, (5.2) becomes $T(i) = a + b\log_2 i$; that is, $T(i)$ is $O(\log i)$. Since `bubbleUp` is $O(\log i)$, so is *insert*.

Now consider *deletemax*. We can see from Fig. 5.49 that the running time of *deletemax* is $O(1)$ plus the running time of `bubbleDown`. The analysis of `bubbleDown`, in Fig. 5.48, is essentially the same as that of `bubbleUp`. We omit it and conclude that `bubbleDown` and *deletemax* also take $O(\log n)$ time.

EXERCISES

5.9.1: Starting with the heap of Fig. 5.45, show what happens when we

a) Insert 3
b) Insert 20
c) Delete the maximum element
d) Again delete the maximum element

5.9.2: Prove Equation (5.2) by induction on i.

5.9.3: Prove by induction on the depth of the POT-property violation that the function `bubbleUp` of Fig. 5.46 correctly restores a tree with one violation to a tree that has the POT property.

5.9.4: Prove that the function `insert(A,x,n)` makes `A` into a heap of size n, if `A` was previously a heap of size $n - 1$. You may use Exercise 5.9.3. What happens if `A` was not previously a heap?

5.9.5: Prove by induction on the height of the POT-property violation that the function `bubbleDown` of Fig. 5.48 correctly restores a tree with one violation to a tree that has the POT property.

5.9.6: Prove that `deletemax(A,n)` makes a heap of size n into one of size $n - 1$. What happens if `A` was not previously a heap?

5.9.7: Prove that `bubbleDown(A,1,n)` takes $O(\log n)$ time on a heap of length n.

5.9.8**: What is the probability that an n-element heap, with distinct element priorities chosen at random, is a partially ordered tree? If you cannot derive the general rule, write a recursive function to compute the probability as a function of n.

5.9.9: We do not need to use a heap to implement a partially ordered tree. Suppose we use the conventional left-child–right-child data structure for binary trees. Show how to implement the functions `bubbleDown`, `insert`, and `deletemax` using this structure instead of the heap structure.

5.9.10*: A binary search tree can be used as an abstract implementation of a priority queue. Show how the operations *insert* and *deletemax* can be implemented using a binary search tree with the left-child–right-child data structure. What is the running time of these operations (a) in the worst case and (b) on the average?

5.10 Heapsort: Sorting with Balanced POTs

We shall now describe the algorithm known as *heapsort.* It sorts an array `A[1..n]` in two phases. In the first phase, heapsort gives A the POT property. The second phase of heapsort repeatedly selects the largest remaining element from the heap until the heap consists of only the smallest element, whereupon the array A is sorted.

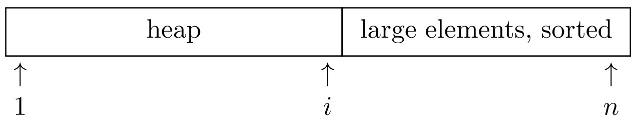


Fig. 5.50. Condition of array A during heapsort.

Figure 5.50 shows the array A during the second phase. The initial part of the array has the POT property, and the remaining part has its elements sorted in nondecreasing order. Furthermore, the elements in the sorted part are the largest $n-i$ elements in the array. During the second phase, i is allowed to run from n down to 1, so that the heap, initially the entire array A, eventually shrinks until it is only the smallest element, located in `A[1]`. In more detail, the second phase consists of the following steps.

1. $A[1]$, the largest element in `A[1..i]`, is exchanged with $A[i]$. Since all elements in `A[i+1..n]` are as large as or larger than any of `A[1..i]`, and since we just moved the largest of the latter group of elements to position i, we know that `A[i..n]` are the largest $n-i+1$ elements and are in sorted order.

2. The value i is decremented, reducing the size of the heap by 1.

3. The POT property is restored to the initial part of the array by bubbling down the element at the root, which we just moved to $A[1]$.

Example 5.32. Consider the array in Fig. 5.45, which has the POT property. Let us go through the first iteration of the second phase. In the first step, we exchange $A[1]$ and $A[10]$ to get:

1	2	3	4	5	6	7	8	9	10
5	18	16	9	7	1	9	3	7	18

The second step reduces the heap size to 9, and the third step restores the POT property to the first nine elements by calling `bubbleDown(1)`. In this call, $A[1]$ and $A[2]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	5	16	9	7	1	9	3	7	18

Then, $A[2]$ and $A[4]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	9	16	5	7	1	9	3	7	18

Finally, $A[4]$ and $A[9]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	9	16	7	7	1	9	3	5	18

At this point, `A[1..9]` has the POT property.

The second iteration of phase 2 begins by swapping the element 18 in `A[1]` with the element 5 in `A[9]`. After bubbling 5 down, the array becomes

1	2	3	4	5	6	7	8	9	10
16	9	9	7	7	1	5	3	18	18

At this stage, the last two elements of the array are the two largest elements, in sorted order.

Phase 2 continues until the array is completely sorted:

1	2	3	4	5	6	7	8	9	10
1	3	5	7	7	9	9	16	18	18

Heapifying an Array

We could describe heapsort informally as follows:

```
for (i = 1; i <= n; i++)
    insert($a_i$);
for (i = 1; i <= n; i++)
    deletemax
```

To implement this algorithm, we insert the n elements $a_1, a_2, \ldots, a_n$ to be sorted into a heap that is initially empty. We then perform *deletemax* n times, getting the elements in largest-first order. The arrangement of Fig. 5.50 allows us to store the deleted elements in the tail of the array, as we shrink the heap portion of that array.

Since we just argued in the last section that *insert* and *deletemax* take $O(\log n)$ time each, and since we evidently execute each operation n times, we have an $O(n \log n)$ sorting algorithm, which is comparable to merge sort. In fact, heapsort can be superior to merge sort in a situation in which we only need a few of the largest elements, rather than the entire sorted list. The reason is that we can make the array be a heap in $O(n)$ time, rather than $O(n \log n)$ time, if we use the function `heapify` of Fig. 5.51.

```
void heapify(int A[], int n)
{
    int i;

    for (i = n/2; i >= 1; i--)
        bubbleDown(A, i, n);
}
```

Fig. 5.51. Heapifying an array.

Running Time of Heapify

At first, it might appear that the $n/2$ calls to `bubbleDown` in Fig. 5.51 should take $O(n \log n)$ time in total, because $\log n$ is the only upper bound we know on the running time of `bubbleDown`. However, we can get a tighter bound, $O(n)$, if we exploit the fact that most of the sequences that bubble down elements are very short.

To begin, we did not even have to call `bubbleDown` on the second half of the array, because all the elements there are leaves. On the second quarter of the array, that is,

```
A[(n/4)+1..n/2]
```

we may call `bubbleDown` once, if the element is smaller than either of its children; but those children are in the second half, and therefore are leaves. Thus, in the second quarter of A, we call `bubbleDown` at most once. Similarly, in the second eighth of the array, we call `bubbleDown` at most twice, and so on. The number of calls to `bubbleDown` in the various regions of the array is indicated in Fig. 5.52.

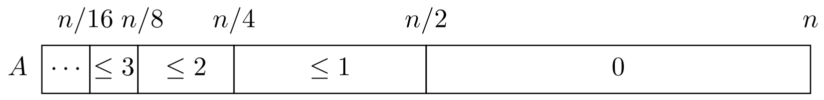


Fig. 5.52. The number of calls to `bubbleDown` decreases rapidly as we go through the array from low to high indices.

Let us count the number of calls to `bubbleDown` made by `heapify`, including recursive calls. From Fig. 5.52 we see that it is possible to divide `A` into *zones,* where the ith zone consists of `A[j]` for j greater than $n/2^{i+1}$ but no greater than $n/2^i$. The number of elements in zone i is thus $n/2^{i+1}$, and there are at most i calls to `bubbleDown` for each element in zone i. Further, the zones $i > \log_2 n$ are empty, since they contain at most $n/2^{1+\log_2 n} = 1/2$ element. The element `A[1]` is the sole occupant of zone $\log_2 n$. We thus need to compute the sum

$$\sum_{i=1}^{\log_2 n} in/2^{i+1} \tag{5.3}$$

We can provide an upper bound on the finite sum (5.3) by extending it to an infinite sum and then pulling out the factor $n/2$:

$$\frac{n}{2}\sum_{i=1}^{\infty} i/2^i \qquad (5.4)$$

We must now get an upper bound on the sum in (5.4). This sum, $\sum_{i=1}^{\infty} i/2^i$, can be written as

$$(1/2) + (1/4 + 1/4) + (1/8 + 1/8 + 1/8) + (1/16 + 1/16 + 1/16 + 1/16) + \cdots$$

We can write these inverse powers of 2 as the triangle shown in Fig. 5.53. Each row is an infinite geometric series with ratio 1/2, which sums to twice the first term in the series, as indicated at the right edge of Fig. 5.53. The row sums form another geometric series, which sums to 2.

$$\begin{array}{ccccccccccc}
1/2 & + & 1/4 & + & 1/8 & + & 1/16 & + & \cdots & = & 1 \\
 & & 1/4 & + & 1/8 & + & 1/16 & + & \cdots & = & 1/2 \\
 & & & & 1/8 & + & 1/16 & + & \cdots & = & 1/4 \\
 & & & & & & 1/16 & + & \cdots & = & 1/8 \\
 & & & & & & & & \cdots & = & \cdots
\end{array}$$

Fig. 5.53. Arranging $\sum_{i=1}^{\infty} i/2^i$ as a triangular sum.

It follows that (5.4) is upper-bounded by $(n/2) \times 2 = n$. That is, the number of calls to `bubbleDown` in the function `heapify` is no greater than n. Since we have already established that each call takes $O(1)$ time, exclusive of any recursive calls, we conclude that the total time taken by `heapify` is $O(n)$.

The Complete Heapsort Algorithm

The C program for heapsort is shown in Fig. 5.54. It uses an array of integers `A[1..MAX]` for the heap. The elements to be sorted are inserted in `A[1..n]`. The definitions of the function declarations in Fig. 5.54 are contained in Sections 5.9 and 5.10.

Line (1) calls `heapify`, which turns the n elements to be sorted into a heap; and line (2) initializes `i`, which marks the end of the heap, to n. The loop of lines (3) and (4) applies `deletemax` $n-1$ times. We should examine the code of Fig. 5.49 again to observe that `deletemax(A,i)` swaps the maximum element of the remaining heap — which is always in `A[1]` — with $A[i]$. As a side effect, i is decremented by 1, so that the size of the heap shrinks by 1. The element "deleted" by `deletemax` at line (4) is now part of the sorted tail of the array. It is less than or equal to any element in the previous tail, `A[i+1..n]`, but greater than or equal to any element still in the heap. Thus, the claimed property is maintained; all the heap elements precede all the elements of the tail.

Running Time of Heapsort

We have just established that `heapify` in line (1) takes time proportional to n. Line (2) clearly takes $O(1)$ time. Since i decreases by 1 each time around the loop of lines (3) and (4), the number of times around the loop is $n-1$. The call to `deletemax` at line (4) takes $O(\log n)$ time. Thus, the total time for the loop is $O(n \log n)$. That time dominates lines (1) and (2), and so the running time of `heapsort` is $O(n \log n)$ on n elements.

```
    #include <stdio.h>

    #define MAX 100

    int A[MAX+1];

    void bubbleDown(int A[], int i, int n);
    void deletemax(int A[], int *pn);
    void heapify(int A[], int n);
    void heapsort(int A[], int n);
    void swap(int A[], int i, int j);

    main()
    {
        int i, n, x;

        n = 0;
        while (n < MAX && scanf("%d", &x) != EOF)
            A[++n] = x;
        heapsort(A, n);
        for (i = 1; i <= n; i++)
            printf("%d\n", A[i]);
    }

    void heapsort(int A[], int n)
    {
        int i;

(1)     heapify(A, n);
(2)     i = n;
(3)     while (i > 1)
(4)         deletemax(A, &i);
    }
```

Fig. 5.54. Heapsorting an array.

EXERCISES

5.10.1: Apply heapsort to the list of elements 3, 1, 4, 1, 5, 9, 2, 6, 5.

5.10.2*: Give an $O(n)$ running time algorithm that finds the $\sqrt{n}$ largest elements in a list of n elements.

5.11 Summary of Chapter 5

The reader should take away the following points from Chapter 5:

Trees are an important data model for representing hierarchical information.

Many data structures involving combinations of arrays and pointers can be used to implement trees, and the data structure of choice depends on the operations performed on the tree.

Two of the most important representations for tree nodes are the leftmost-child–right-sibling representation and the trie (array of pointers to children).

Recursive algorithms and proofs are well suited for trees. A variant of our basic induction scheme, called structural induction, is effectively a complete induction on the number of nodes in a tree.

The binary tree is a variant of the tree model in which each node has (optional) left and right children.

A binary search tree is a labeled binary tree with the "binary search tree property" that all the labels in the left subtree precede the label at a node, and all labels in the right subtree follow the label at the node.

The dictionary abstract data type is a set upon which we can perform the operations *insert*, *delete*, and *lookup*. The binary search tree efficiently implements dictionaries.

A priority queue is another abstract data type, a set upon which we can perform the operations *insert* and *deletemax*.

A partially ordered tree is a labeled binary tree with the property that the label at any node is at least as great as the label at its children.

Balanced partially ordered trees, where the nodes fully occupy levels from the root to the lowest level, where only the leftmost positions are occupied, can be implemented by an array structure called a heap. This structure provides an $O(\log n)$ implementation of a priority queue and leads to an $O(n \log n)$ sorting algorithm called heapsort.

5.12 Bibliographic Notes for Chapter 5

The trie representation of trees is from Fredkin [1960]. The binary search tree was invented independently by a number of people, and the reader is referred to Knuth [1973] for a history as well as a great deal more information on various kinds of search trees. For more advanced applications of trees, see Tarjan [1983].

Williams [1964] devised the heap implementation of balanced partially ordered trees. Floyd [1964] describes an efficient version of heapsort.

Floyd, R. W. [1964]. "Algorithm 245: Treesort 3," *Comm. ACM* **7**:12, pp. 701.

Fredkin, E. [1960]. "Trie memory," *Comm. ACM* **3**:4, pp. 490–500.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM Press, Philadelphia.

Williams, J. W. J. [1964]. "Algorithm 232: Heapsort," *Comm. ACM* **7**:6, pp. 347–348.