Chapter 22

Binary Trees

Like stacks and lists, binary trees are structures that store values. Stacks and lists are linear. That is, you can visualize them as consisting of a sequential row of values, one after the other. Each data structure has operations for storing and retrieving values. With a stack, to store an item you push it onto the top of the stack, while a list allows you to insert a value at an arbitrary location.

Binary trees have more of a two-dimensional structure compared to stacks and lists. Like all data structures, however they have operations for storing and retrieving values. This chapter begins by defining the characteristics of an abstract binary tree. It concludes by showing how pointers can be used to implement the data structure.

Abstract binary trees

The definition of an abstract binary tree is recursive. It is defined in terms of itself. An abstract binary tree is

The definition of an abstract binary tree

- an empty tree

or

- a nonempty tree consisting of a root cell containing
  - a left child, which is a binary tree
  - a value
  - a right child, which is a binary tree.

You can see the recursive nature of the definition, because the root cell of a nonempty tree contains a left child, which is in turn a tree. Similarly to programming with recursion, a recursive definition must have a base case that stops the recursion. In this definition, the empty tree is the base case, because it is not defined in terms of another tree.

The definition of a leaf

Figure 22.1 illustrates this definition of an abstract binary tree for a tree of integers. The root of the tree is the node that contains 5. The leaves—6, 1, and 7—are those nodes whose left and right children are both empty. The structure in Figure 22.1(a) is a binary tree because its root, 5, has a left child that is a binary subtree, as shown in (b), and a right child that is a binary subtree, as shown in (c).

Figure 22.2 shows why the structure in Figure 22.1(b) is a binary tree. The root of this tree contains 4, its left child in Figure 22.2(b) is empty, and its right child, shown in Figure 22.2(c), consists of a single node. You can see from this line of rea-

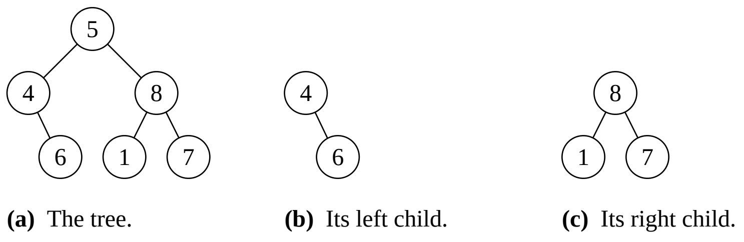


Figure 22.1
An abstract binary tree.

soning that each child of a node in the original structure of Figure 22.1(a) can be shown to be a binary tree. The basis of the definition is the fact that a binary tree can be empty.

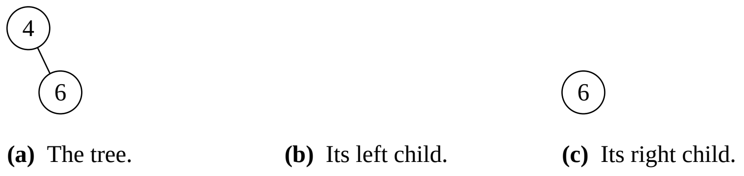


Figure 22.2
Another abstract binary tree.

Every node in a tree has a depth. The depth of the root node is always zero. The depth of any child of the root is one. In general, the depth of any node is one plus the depth of its parent node. The height of a tree is the maximum value of the depths of all its nodes. Figure 22.3 shows a binary tree having nine nodes with the depth of each node labeled. Node 42 has a depth of 2. Node 7 has the maximum depth, 4, of all the nodes. Therefore, 4 is the height of the tree

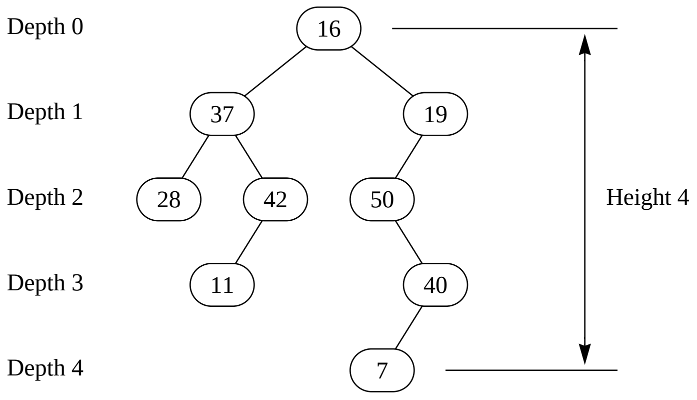


Figure 22.3
The depth of the nodes and the height of the tree.

The values in the binary tree of Figure 22.3 have no particular order associated with them. This binary tree is said to be unordered. In practice, binary trees usually are ordered in a way that makes retrieving a value efficient. Such a tree is called a *binary search tree*. A binary search tree satisfies four criteria:

The definition of a binary search tree

- every value in the left subtree of the root is less than the value of the root
- the left subtree is a search tree
- every value in the right subtree of the root is greater than the value of the root
- the right subtree is a search tree

It is possible to construct a binary tree in which the left and right children of the root are search trees, but the tree itself is not a search tree. It is also possible to construct a binary tree in which every value in the left subtree is less than the root and every value in the right subtree is greater than the root, but the tree itself is not a search tree. For a binary tree to be a search tree, all four criteria must hold.

Example 22.1 In Figure 22.1(a), the fact that 1 is in the right subtree of root 5, shows that the tree is not a search tree. Another node out of order is 6, which is in the left subtree of the root. ▮

Example 22.2 The binary tree of Figure 22.4 is a search tree. All the nodes in the left subtree of the root (20, 10, 30) are less than the root value, and all the nodes in the right subtree (60, 50) are greater than the root value. Furthermore, the subtree with 20 as a root is itself a search tree, because 10 is less than 20, and 30 is greater than 20. Similarly, the subtree with 60 as a root is itself a search tree, because 50, the value in the left child of 60, is less than 60. ▮

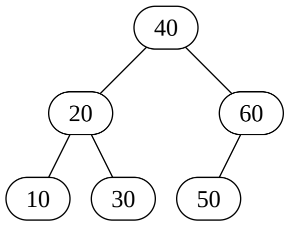


Figure 22.4
A binary search tree.

In their abstract form, binary trees are usually written on paper or displayed on the screen as two-dimensional drawings. It is frequently necessary, however, to output the values from the tree in a single list as opposed to a flat drawing. To print all the values requires a procedure that somehow travels around the tree, visiting the various nodes and outputting their values. Such a trip is called a *traversal*. Three common traversals of a binary tree are:

Binary tree traversals

- Preorder traversal
- Inorder traversal
- Postorder traversal

The definition of each traversal is recursive and is related to the recursive nature of the definition of a binary tree.

The definition of a preorder traversal is

The preorder traversal

- Visit the root.
- Make a preorder traversal of the left subtree, if any.
- Make a preorder traversal of the right subtree, if any.

This definition is recursive because the preorder traversal requires two other preorder traversals.

Figure 22.5(a) shows the preorder traversal of the binary tree of Figure 22.1. The line that enters from the upper left and exits to the upper right traces the path. The definition says to first visit the root, which the figure indicates by the solid box to the left of the root. Then do a preorder traversal of the subtree whose root is 20, followed by a preorder traversal of the subtree whose root is 60.

Now apply the preorder traversal to the tree whose root is 20. First visit 20, then

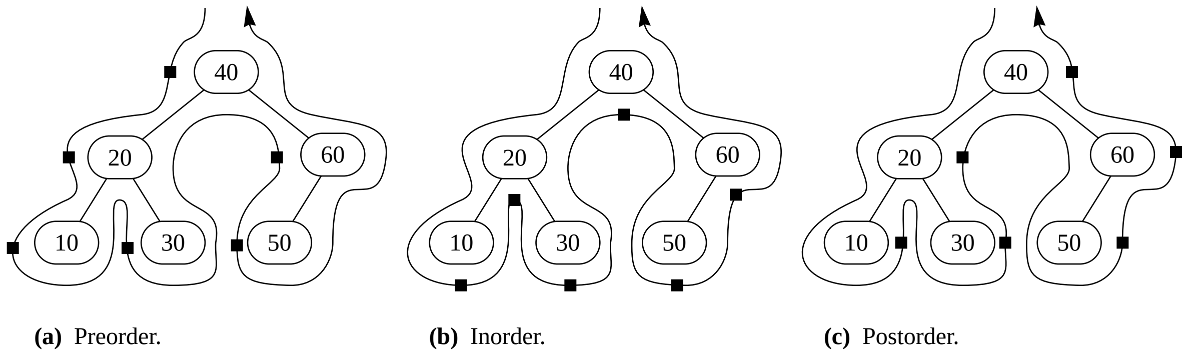


Figure 22.5
The visits to the nodes in the tree traversal algorithms.

do a preorder traversal with 10 as the root, followed by a preorder traversal with 30 as the root. Similarly, the preorder traversal of the tree whose root is 60 consists of a visit to 60, followed by a visit to 50. The net result is

40 20 10 30 60 50

The definition of an inorder traversal is

The inorder traversal

- Make an inorder traversal of the left subtree, if any.
- Visit the root.
- Make an inorder traversal of the right subtree, if any.

Figure 22.5(b) shows the corresponding inorder visitation on the same tree. This time the incoming path does not first visit the root. Instead, it waits until the left subtree has been traversed. Then the root is visited as indicated by the solid box on the path just under the root. After the root is visited, the path traverses the right subtree. The net result is

10 20 30 40 50 60

The definition of a postorder traversal is

The postorder traversal

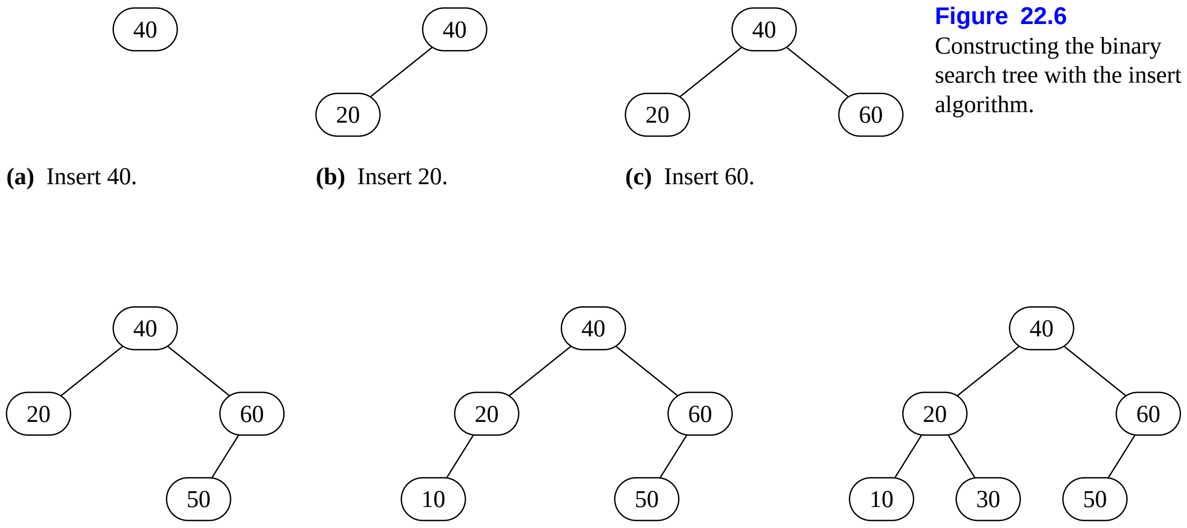
- Make a postorder traversal of the left subtree, if any.
- Make a postorder traversal of the right subtree, if any.
- Visit the root.

Figure 22.5(c) shows the postorder traversal. The path does not show a visit to the root until both the left and right subtrees have been traversed. This time the output is

10 30 20 50 60 40

Remember that the tree in Figure 22.5 is a binary search tree. The example of an inorder traversal of a binary search tree shows that it outputs the values in order, as if they were sorted. In fact, one of the primary uses of binary trees is to maintain lists of elements in sorted order.

To build a binary search tree requires an insert operation. Insert assumes that a given binary tree is a search tree and inserts a new node with some value to the tree, maintaining its ordered state. Figure 22.6 shows the structure of an abstract tree of integers that is initially empty and is constructed with the sequence of inserts 40, 20, 60, 50, 10, 30.



(a) Insert 40. **(b)** Insert 20. **(c)** Insert 60.

(d) Insert 50. **(e)** Insert 10. **(f)** Insert 30.

Figure 22.6 Constructing the binary search tree with the insert algorithm.

With each insert operation, the newly created node takes the place of an empty left child or an empty right child of some node in the tree. The node that is inserted becomes a leaf. If the node to which it is attached was previously a leaf, that node becomes an *internal node*, that is, a node that is not a leaf.

Definition of an internal node

When a given value is inserted to a given ordered binary tree, the attachment point is unique. For example, to insert 10 to the tree of Figure 10.18(d) it must take the place of the left child of 20. Placing 10 at any other available location would produce a tree that is not ordered.

A binary search tree ADT

In the same way that a linked list can be packaged as an ADT or as a class, a binary search tree can be packaged both ways. Figure 22.7 shows the interface for a binary search tree ADT.

```
DEFINITION PboxTreeADT;

   TYPE
      Tree = POINTER TO Node;
      T = ARRAY 16 OF CHAR;

   PROCEDURE Clear (OUT tr: Tree);
   PROCEDURE Contains (tr: Tree; IN val: T): BOOLEAN;
   PROCEDURE Insert (VAR tr: Tree; IN val: T);
   PROCEDURE NumItems (tr: Tree): INTEGER;
   PROCEDURE PreOrder (tr: Tree);
   PROCEDURE InOrder (tr: Tree);
   PROCEDURE PostOrder (tr: Tree)

END PboxTreeADT.
```

Figure 22.7
The interface for the binary search tree ADT.

As with the list ADT in Chapter 21, type T is the type of the values that are stored in the data structure. The tree itself is a pointer. The documentation for Tree and T is

TYPE **Tree**
The binary search tree ADT supplied by PboxTreeADT.

TYPE **T**
The type of each element in the tree, a string of at most 15 characters.

The documentation for procedure Insert shows that it has a precondition.

PROCEDURE **Insert** (VAR tr: Tree; IN val: T)
Pre
Tree tr does not already contain val. 20
Post
val is inserted in tree tr, maintaining its ordered property.

If you try to insert an element in the tree that already contains the same value, a trap will be generated with error number 20.

Procedure Contains returns true iff tr contains element val, and function NumItems returns the number of items in tr. Neither of these methods has a precondition. The next three procedures output the tree in preorder, inorder, and postorder. Because the operation of outputting a tree to the Log does not change the tree, tr in these methods is called by value. The Clear procedure clears an existing tree to the empty tree. Because this method will change the tree, tr is called by reference.

As with the linked list interface in Chapter 21, no capacity is specified, indicating that the only limit on the size of a binary search tree is the amount of available memory. It is apparent from this fact that this binary tree is implemented as a linked structure with dynamic storage allocation as opposed to being implemented with an array. While the previous examples showed binary search trees that stored integers, this interface is for an ordered binary tree that stores strings.

Figure 22.8 shows one possible data structure for the binary search tree ADT. As with the list ADT, type Tree is a pointer to Node. Node, however, contains two links to the left and right subtree rather than just one link to the next element in a list. Rather than describe the implementation of the binary search tree ADT, this chapter concludes with the corresponding implementation of the binary search tree class.

```
TYPE
    T* = ARRAY 16 OF CHAR;
    Tree* = POINTER TO Node;
    Node = RECORD
        leftChild: Tree;
        value: T;
        rightChild: Tree
    END;
```

Figure 22.8
The data structure for the binary search tree ADT.

A binary search tree class

Module PboxTreeObj implements an ordered binary tree as a class. Figure 22.9 shows its interface. The methods have the same names and operations as the corresponding procedures in the ADT of Figure 22.7. In particular, the only method that has a precondition is Insert, which does not allow the insertion of a duplicate item.

Figure 22.10 shows the dialog box for a program that uses the binary search tree class of Figure 22.9. The user does not specify a position for an insertion. Instead, the insert algorithm inserts an element into the one place that will maintain the ordered property of the tree. The result of a search is simply a statement of whether a tree contains a given element. No position is associated with the result as it is with the locate option in the dialog box of Figure 21.28 for the linked list. Completion of method Contains, which performs the search, is left as a problem for the student. Also left as a problem are methods NumItems, InOrder, and PostOrder. The numbers 999 in the dialog box are produced by a stub in method NumItems.

```
DEFINITION PboxTreeObj;

    TYPE
        T = ARRAY 16 OF CHAR;
        Tree = RECORD
            (VAR tr: Tree) Clear, NEW;
            (IN tr: Tree) Contains (IN val: T): BOOLEAN, NEW;
            (VAR tr: Tree) Insert (IN val: T), NEW;
            (IN tr: Tree) NumItems (): INTEGER, NEW;
            (IN tr: Tree) PreOrder, NEW;
            (IN tr: Tree) InOrder, NEW;
            (IN tr: Tree) PostOrder, NEW
        END;

END PboxTreeObj.
```

Figure 22.9
The interface for the binary search tree class.

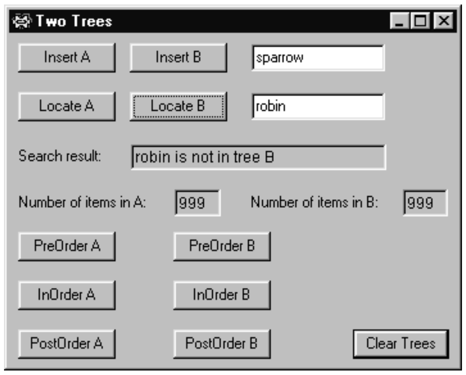


Figure 22.10
The dialog box for manipulating two binary search trees.

The program in Figure 22.11 shows how to use the ordered binary tree class. The interactor is linked to the dialog box of Figure 22.10. Corresponding to the 11 buttons in the dialog box are 11 procedures in module Pbox22A.

Figure 22.11
A program that uses the binary search tree class.

```
MODULE Pbox22A;
   IMPORT Dialog, PboxTreeObj, StdLog;

   VAR
      d*: RECORD
         insertT*: PboxTreeObj.T;
         searchT*: PboxTreeObj.T;
         resultString-: ARRAY 64 OF CHAR;
         numItemsA-, numItemsB-: INTEGER;
      END;
      treeA, treeB: PboxTreeObj.Tree;

   PROCEDURE InsertA*;
   BEGIN
      treeA.Insert(d.insertT);
      d.numItemsA := treeA.NumItems();
      Dialog.Update(d)
   END InsertA;

   PROCEDURE InsertB*;
   BEGIN
      treeB.Insert(d.insertT);
      d.numItemsB := treeB.NumItems();
      Dialog.Update(d)
   END InsertB;
```

Figure 22.11
Continued.

```
PROCEDURE SearchA*;
BEGIN
    IF treeA.Contains(d.searchT) THEN
        d.resultString := d.searchT + " is in tree A"
    ELSE
        d.resultString := d.searchT + " is not in tree A"
    END;
    Dialog.Update(d)
END SearchA;

PROCEDURE SearchB*;
BEGIN
    IF treeB.Contains(d.searchT ) THEN
        d.resultString := d.searchT + " is in tree B"
    ELSE
        d.resultString := d.searchT + " is not in tree B"
    END;
    Dialog.Update(d)
END SearchB;

PROCEDURE PreOrderA*;
BEGIN
    StdLog.Ln;
    treeA.PreOrder
END PreOrderA;

PROCEDURE PreOrderB*;
BEGIN
    StdLog.Ln;
    treeB.PreOrder
END PreOrderB;

PROCEDURE InOrderA*;
BEGIN
    StdLog.Ln;
    treeA.InOrder
END InOrderA;

PROCEDURE InOrderB*;
BEGIN
    StdLog.Ln;
    treeB.InOrder
END InOrderB;

PROCEDURE PostOrderA*;
BEGIN
    StdLog.Ln;
    treeA.PostOrder
END PostOrderA;
```

```
    PROCEDURE PostOrderB*;
    BEGIN
        StdLog.Ln;
        treeB.PostOrder
    END PostOrderB;

    PROCEDURE ClearTrees*;
    BEGIN
        treeA.Clear; treeB.Clear;
        d.insertT := "";
        d.searchT := ""; d.resultString := "";
        d.numItemsA := 0; d.numItemsB := 0;
        Dialog.Update(d)
    END ClearTrees;

BEGIN
    ClearTrees
END Pbox22A.
```

Figure 22.11
Continued.

treeA and treeB are the two global variables whose states are maintained between clicks of the buttons of the dialog box. Most of the procedures simply use the user input from the dialog box as actual parameters in a call to the corresponding method from the class. The procedures for outputting the tree traversals include a call of StdLog.Ln before calling on the class so that the output for a traversal will begin on a new line.

The implementation of a binary tree is closely related to the recursive definition of an abstract binary tree. From an abstract perspective, a binary tree is either empty or is a cell containing a value and two binary trees. In the same way that a list is implemented as a pointer to the *head* node of the list, a binary tree is implemented as a pointer to the *root* node of a tree. Each node in the tree contains a value part to hold the data and two other parts to hold its left child and its right child. The data part obviously has type T, but what is the type of the left child and right child? From the recursive definition of a binary tree, they should each have type Tree. Figure 22.12 shows the structure of a record for a node in the implementation of a binary tree.

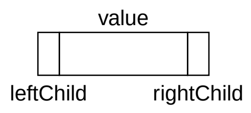
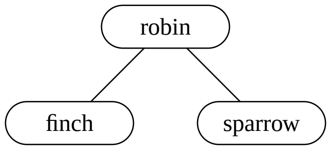


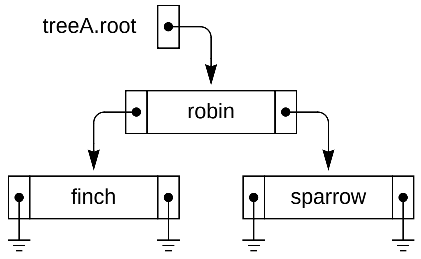
Figure 22.12
The structure of a record for a node in the implementation of a binary tree.

The abstract binary tree of strings in Figure 22.13(a) has three elements with robin as the root of the tree, finch as its left child and sparrow as its right child. The links are simply drawn as lines between the nodes. Figure 22.13(b) shows the nodes in the tree as it would be implemented with the record of Figure 22.12. treeA is a pointer to the node record that represents its root. The value part of the root node contains the string robin. The leftChild is a tree. That is, it is a pointer to the root of the left subtree, a record containing finch in its value part. Because the node containing finch has no left child, the pointer in its leftChild part is NIL. Likewise, the pointer in its rightChild part is NIL.

Figure 22.14 shows the implementation using the structure of the nodes as shown in the previous figures. The type declarations define type Tree to be a record, which contains a pointer to the tree's root node. The root node, in turn, contains a left subtree, a value to store the data at the node, and a right subtree. Subtrees leftChild and



(a) An abstract binary tree.



(b) Its linked implementation.

Figure 22.13 An abstract binary tree and its implementation as a class.

rightChild have type Tree. That is, they are records with a single pointer to the roots of the left and right subtrees.

Because a tree is a pointer to its root node, and an empty tree has no nodes, an empty tree is represented by a pointer whose value is NIL. Method Clear makes a tree empty by setting to NIL the pointer to its root node. The nodes that comprised the tree are later reclaimed by the automatic garbage collector.

Figure 22.14 Implementation of the ordered binary tree class that is used in Figure 22.11.

```
MODULE  PboxTreeObj;
   IMPORT StdLog;

   TYPE
      T* = ARRAY 16 OF CHAR;
      Tree* = RECORD
         root: POINTER TO Node
      END;
      Node = RECORD
         leftChild: Tree;
         value: T;
         rightChild: Tree
      END;

   PROCEDURE (VAR tr: Tree) Clear*, NEW;
   BEGIN
      tr.root := NIL
   END Clear;

   PROCEDURE (IN tr: Tree) Contains* (IN val: T): BOOLEAN, NEW;
   BEGIN
      (* A problem for the student *)
      RETURN FALSE
   END Contains;
```

Figure 22.14
Continued.

```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            p := p.root.leftChild
        ELSE
            p := p.root.rightChild
        END
    END;
    (* Attach new node to parent *)
    NEW(p.root);
    p.root.value := val;
    IF parent.root = NIL THEN (* tr is empty *)
        tr := p
    ELSIF val < parent.root.value THEN
        parent.root.leftChild := p
    ELSE
        parent.root.rightChild := p
    END
END Insert;

PROCEDURE (IN tr: Tree) NumItems* (): INTEGER, NEW;
BEGIN
    (* A problem for the student *)
    RETURN 999
END NumItems;

PROCEDURE (IN tr: Tree) PreOrder*, NEW;
BEGIN
    IF tr.root # NIL THEN
        StdLog.String(tr.root.value); StdLog.String("  ");
        tr.root.leftChild.PreOrder;
        tr.root.rightChild.PreOrder
    END
END PreOrder;

PROCEDURE (IN tr: Tree) InOrder*, NEW;
BEGIN
    (* A problem for the student *)
END InOrder;
```

```
    PROCEDURE (IN tr: Tree) PostOrder*, NEW;
    BEGIN
        (* A problem for the student *)
    END PostOrder;

END PboxTreeObj.
```

Figure 22.14
Continued.

PboxTreeObj implements method PreOrder directly from the definition of a preorder traversal. If a tree is empty, it has no preorder traversal and the method simply returns to the calling procedure. Otherwise, the steps in the algorithm are to first visit the root, then to do a preorder traversal of the left subtree followed by a preorder traversal of the right subtree. To indicate that the root is visited, the method outputs the data from the current node. It then recursively calls for a preorder traversal of the left subtree followed by a recursive call for a preorder traversal of the right subtree. Implementation of procedures InOrder and PostOrder are similar and are left as a problem for the student.

Figure 22.15 shows the action of procedure Insert when it is called to insert the value pat into an ordered binary tree. The procedure has two parts. First, it must find the position in the tree to attach the new node. Then, it must allocate the new node and attach it.

To find the position, the procedure maintains two local variables, p and parent, which are both trees. Variable p advances through the tree starting from the root, and makes its way to the correct insertion point. Each time p advances one level down the tree, variable parent points to the node from which p advances. When p finally gets NIL, parent will point to the node to which the new node must be attached. Figure 22.15(a) through (d) shows the sequence of events of the first part of the procedure to find the position in the tree.

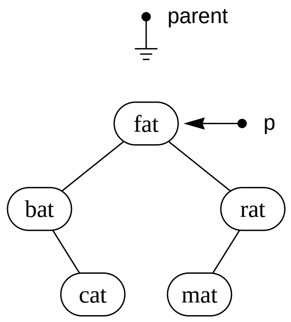
Each time through the loop the procedure must decide whether to advance p to the left or to the right. Recall from Figure 22.9 the precondition that duplicate values are not allowed. The assertion in the body of the WHILE loop in procedure Insert guarantees that if v is not less than p.value it will be greater than p.value.

Figure 22.15(e) shows the effect of the call to procedure NEW. Figure 22.15(f) shows how the new node is attached to its parent. In the figure, the original tree tr is not empty. If it were, variable parent would be NIL, and procedure Insert would simply point tr to the newly allocated node.

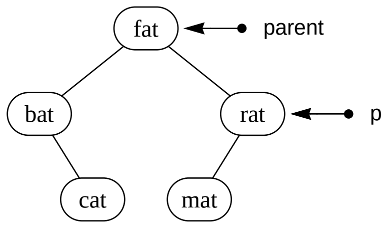
Implementation of procedure Contains is left as a problem for the student. It is best programmed recursively. If tree tr is empty it obviously does not contain v and can return false. Otherwise, tr is not empty and its root must contain some value. If that value equals v, the procedure can return true with no further recursive calls necessary. Otherwise, the procedure must determine whether v might be contained in the left subtree or the right subtree. Your implementation should not search the entire tree, but should use the fact that every value in the left subtree is less than the value in the root and every value in the right subtree is greater than the root.

Procedure NumItems is also left as a problem for the student, and is also best programmed recursively. If tree tr is empty, it obviously has zero elements. Otherwise, it contains one element, its left subtree contains some number of elements, and its right subtree contains some number of elements. The integer it must return is, therefore, one plus the number of elements in its left subtree plus the number of elements in its right subtree.

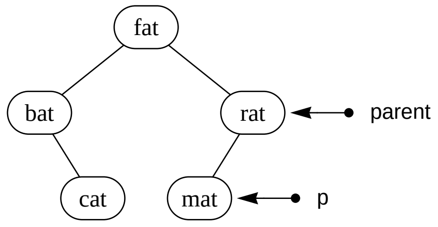
Figure 22.15
The action of procedure Insert to insert the value pat in an ordered binary tree.



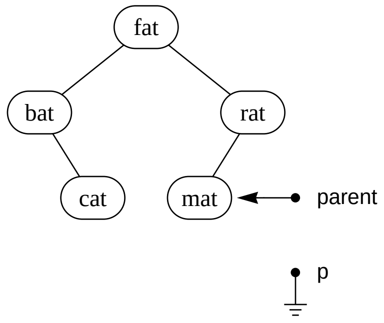
(a) parent.root := NIL; p := tr



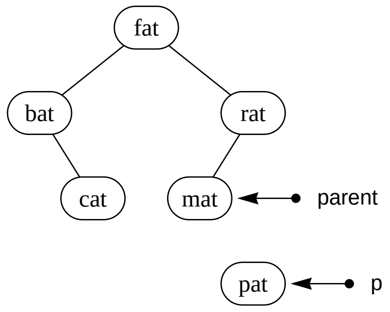
(b) parent := p; p := p.root.rightChild



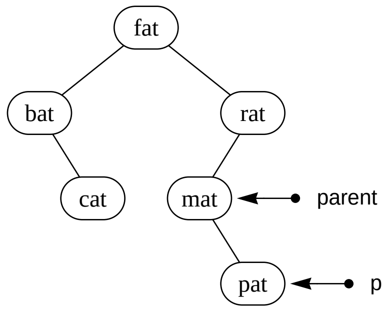
(c) parent := p; p := p.root.leftChild



(d) parent := p, p := p.root.rightChild



(e) NEW(p.root); p.root.value := val



(f) parent.root.rightChild := p

Exercises

1. Draw the final binary search tree as in Figure 22.6(f) for each of the following sequences of the insert operation.

(a) 50 30 80 60 40 20 10 **(b)** 50 30 80 60 40 10 20
(c) 50 60 70 80 10 20 30 40 **(d)** 10 20 30 40 50
(e) 50 40 30 20 10

2. For each of the binary search trees of Exercise 1, write the preorder sequence.

3. For each of the binary search trees of Exercise 1, write the postorder sequence.

4. For each binary tree in Figure 22.16, (1) state whether the tree is a search tree, (2) write the preorder traversal, (3) write the inorder traversal, and (4) write the postorder traversal.

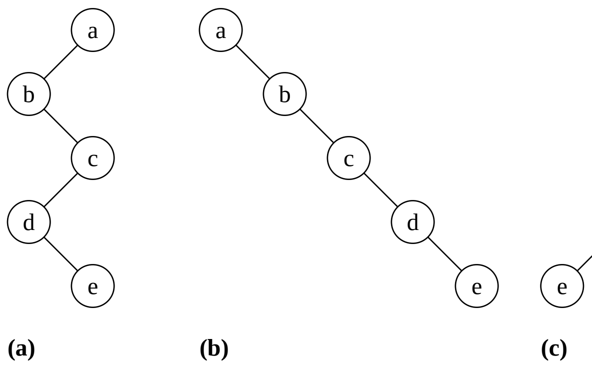


Figure 22.16 The binary trees for Exercise 4.

5. A binary search tree contains a set of integers. Assume that each of the following sequences is the preorder sequence. From the preorder sequence and the known inorder sequence, draw the ordered binary tree.

(a) 40 20 60 **(b)** 60 40 20
(c) 60 20 40 **(d)** 20 40 60 80
(e) 60 30 10 80 70 90

6. A binary search tree contains a set of integers. Assume that each of the following sequences is the postorder sequence. From the postorder sequence and the known inorder sequence, draw the ordered binary tree.

(a) 40 20 60 **(b)** 20 60 40
(c) 20 40 60 **(d)** 80 60 40 20
(e) 30 10 80 90 70 60

Problems

7. This problem is for you to complete the procedures for the binary search tree ADT in module PboxTreeADT. Test your procedures with a program similar to the one in Figure 22.11 using the dialog box of Figure 22.10. For each procedure, access the documentation and implement any preconditions with the appropriate ASSERT or HALT procedure.

(a) PROCEDURE Insert (VAR tr: Tree; IN val: T)
(b) PROCEDURE Contains (tr: Tree; IN val: T): BOOLEAN
(c) PROCEDURE NumItems (tr: Tree): INTEGER
(d) PROCEDURE PreOrder (tr: Tree)
(e) PROCEDURE InOrder (tr: Tree)
(f) PROCEDURE PostOrder (tr: Tree)
(g) PROCEDURE Clear (OUT tr: Tree)

8. Implement the following methods in module PboxTreeObj.

(a) PROCEDURE (IN tr: Tree) Contains (IN val: T): BOOLEAN, NEW
Write a nonrecursive version.

(b) PROCEDURE (IN tr: Tree) Contains (IN val: T): BOOLEAN, NEW
Write a recursive version without a loop. Use the fact that the tree is a search tree to avoid unnecessary comparisons. For example, if val is not in the tree, do not search the entire tree.

(c) PROCEDURE (IN tr: Tree) NumItems (): INTEGER, NEW
(d) PROCEDURE (IN tr: Tree) InOrder, NEW
(e) PROCEDURE (IN tr: Tree) PostOrder, NEW

9. This problem requires you to add the following methods to Figure 22.14. Test your methods by importing them into the module of Figure 22.11. Augment the dialog box of Figure 22.10 to test the procedures.

(a) PROCEDURE (IN tr: Tree) NumLeaves (): INTEGER, NEW
Return the number of leaves of tree tr. A leaf is a node that has no children.

(b) PROCEDURE (IN tr: Tree) NumInternals (): INTEGER, NEW
Return the number of internal nodes of tree tr. An internal node is a node that is not a leaf.

(c) PROCEDURE (IN tr: Tree) OutLeaves, NEW
Output the leaves of tree tr to the Log.

(d) PROCEDURE (VAR tr: Tree) StripLeaves, NEW
Remove all the leaves from binary tree tr.

(e) PROCEDURE (IN tr: Tree) ReverseOrder, NEW
Output the values from binary tree tr to the Log in the reverse of inorder.

(f) PROCEDURE (IN tr: Tree) Height (): INTEGER, NEW
Return the height of the binary tree tr. The definition of the height is recursive. If the tree is empty, its height is –1. Otherwise its height is 1 plus the larger of the height of the left and right subtrees.

(g) PROCEDURE (VAR tr: Tree) Copy (trB: Tree), NEW
Create a new tree tr, which is a copy of trB. tr must contain the same values as trB and must also have the same shape as trB. This is best done recursively. If trB is empty then tr should be made empty. Otherwise, a copy of trB's root node should be created for tr's root, and copies should be made recursively for the left and right children.

(h) PROCEDURE (VAR tr: Tree) Equal (trB: Tree): BOOLEAN, NEW
Return true iff tr is equal to trB. Two trees are equal if they have the same number of equal values and if they have the same shape. This is best done recursively. The base cases are (1) both trees empty, (2) one tree empty and the other not, and (3) both trees not empty but with unequal values in their roots.