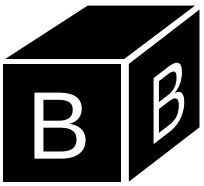


Chapter 22

Binary Trees

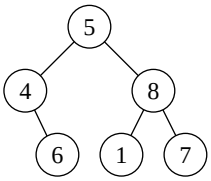
An abstract binary tree is

- an empty tree

The definition of an abstract binary tree

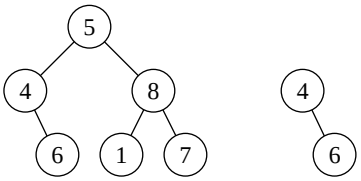
or

- 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.



(a) The tree.

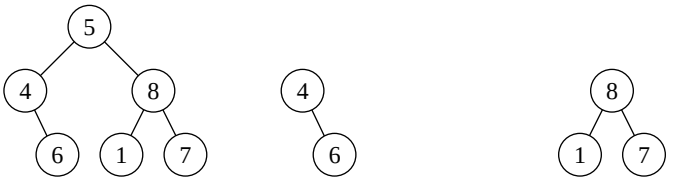
Figure 22.1
An abstract binary tree.



(a) The tree.

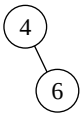
(b) Its left child.

Figure 22.1
An abstract binary tree.



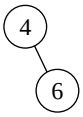
(a) The tree. **(b)** Its left child. **(c)** Its right child.

Figure 22.1
An abstract binary tree.



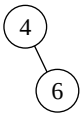
(a) The tree.

Figure 22.2
Another abstract binary tree.



(a) The tree. **(b)** Its left child.

Figure 22.2
Another abstract binary tree.



(a) The tree.

(b) Its left child.

(c) Its right child.

Figure 22.2
Another abstract binary tree.

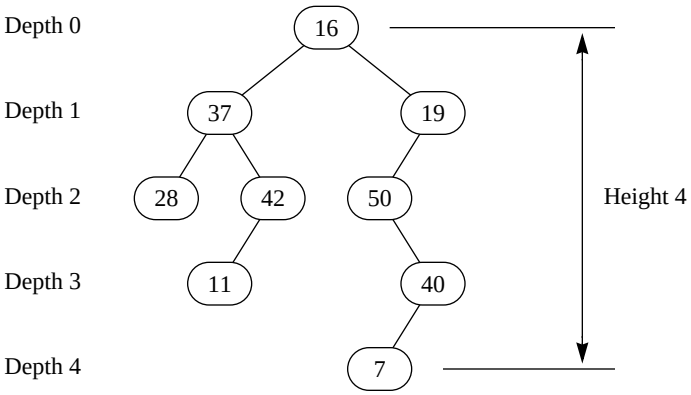


Figure 22.3
The depth of the nodes and the height of the tree.

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

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

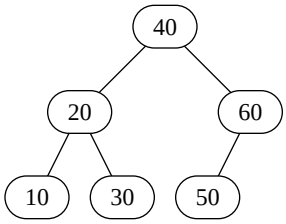


Figure 22.4
A binary search tree.

Three common traversals of a binary tree are:

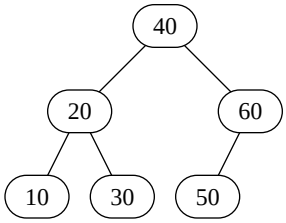
Binary tree traversals

- Preorder traversal
- Inorder traversal
- Postorder traversal

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.

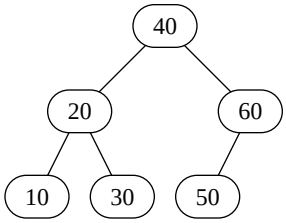


Preorder traversal:

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.



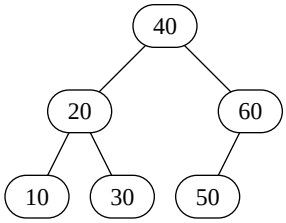
Preorder traversal:

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.

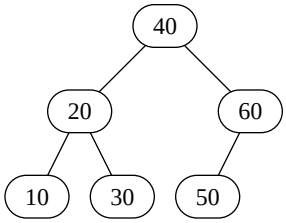


Inorder traversal:

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.



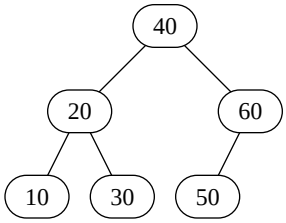
Inorder traversal:

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.

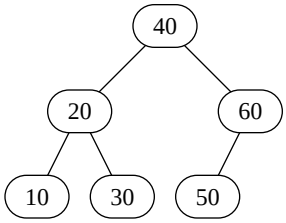


Postorder traversal:

The definition of a postorder traversal is

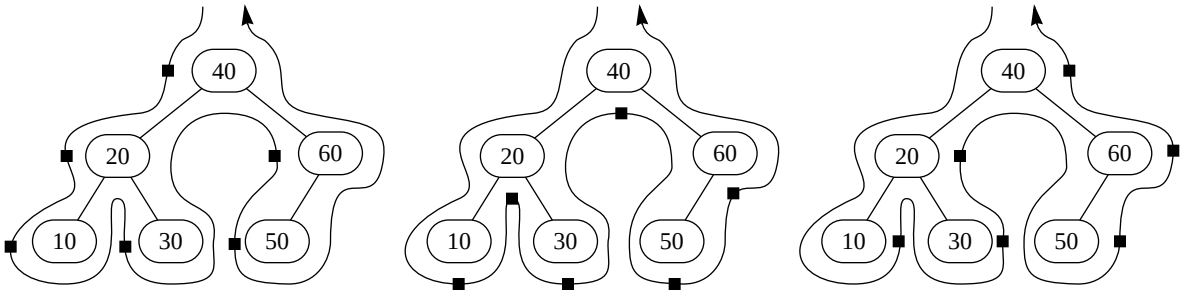
The postorder traversal

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



Postorder traversal:

10 30 20 50 60 40



(a) Preorder. **(b)** Inorder. **(c)** Postorder.

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

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

(a) Insert 40.

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

(b) Insert 20.

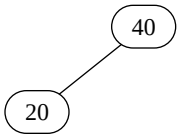


Figure 22.6
Constructing the binary search tree with the insert algorithm.

(c) Insert 60.

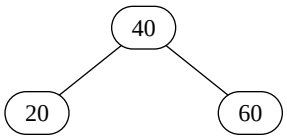
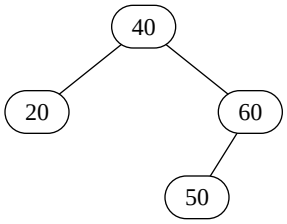


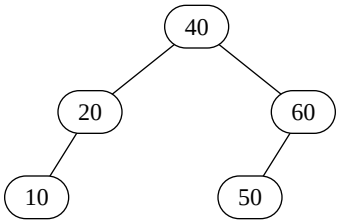
Figure 22.6
Constructing the binary search tree with the insert algorithm.

(d) Insert 50.



(e) Insert 10.

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



(f) Insert 30.

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

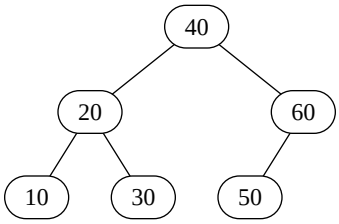


Figure 22.6
Constructing the binary search tree with the insert algorithm.

```
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.

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.

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.

```
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.

```
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.

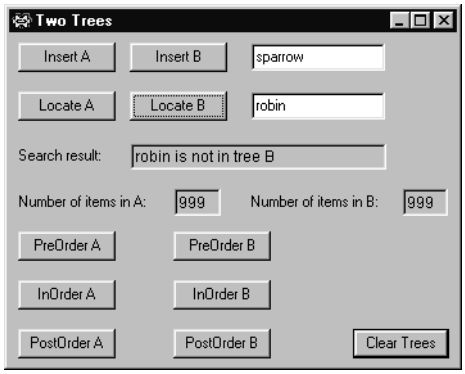


Figure 22.10
The dialog box for manipulating two binary search trees.

```
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;
```

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

```
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;
```

```
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.
```

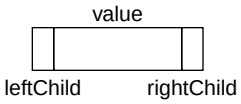
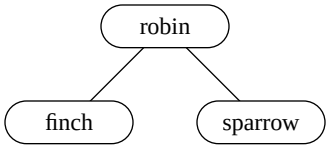
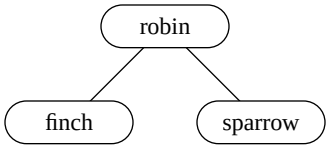


Figure 22.12
The structure of a record for a node in the implementation of a binary tree.

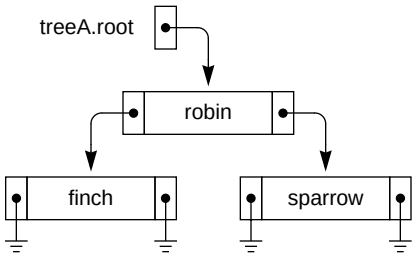


(a) An abstract binary tree.

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



(a) An abstract binary tree.



(b) Its linked implementation.

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

```
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;
```

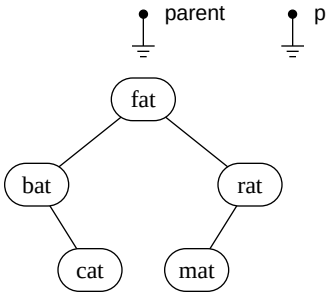
Figure 22.14
Implementation of the ordered binary tree class that is used in Figure 22.11.

```
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;
```

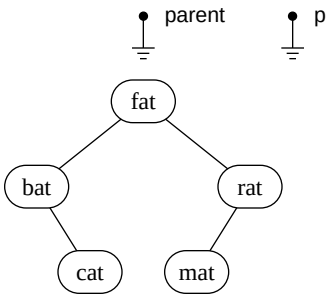
```
    PROCEDURE (IN tr: Tree) NumItems* (): INTEGER, NEW;
    BEGIN
        (* A problem for the student *)
        RETURN 999
    END NumItems;

    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.
```

```
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;
```

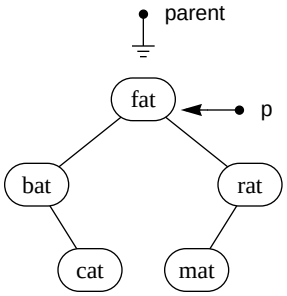
val
pat



```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
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;
```

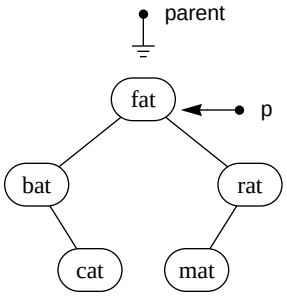


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

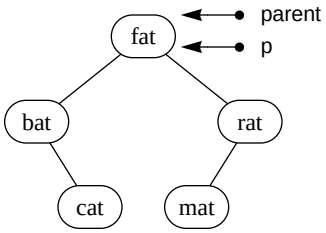


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            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;
```

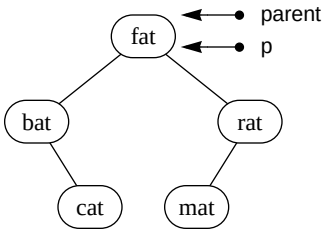
val
pat



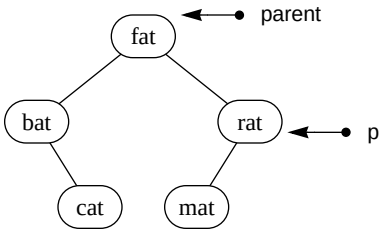
```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```



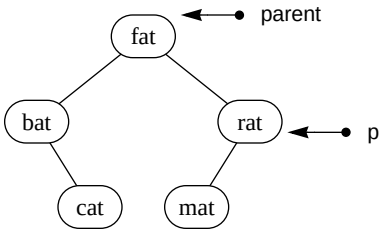
```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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 (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
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;
```

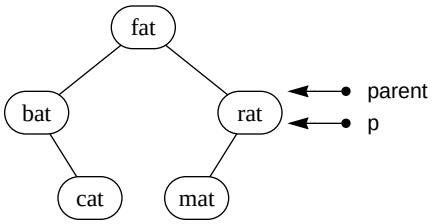


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

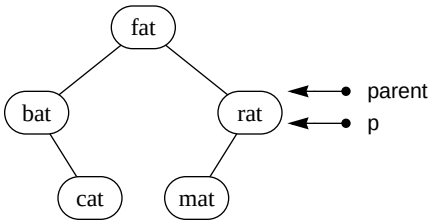


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            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;
```

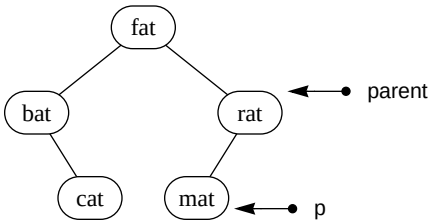
val
pat



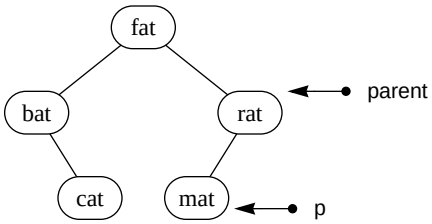
```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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 (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
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;
```

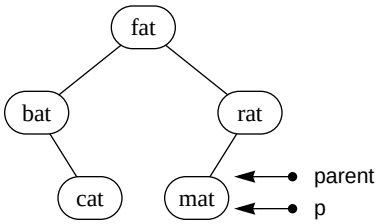


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

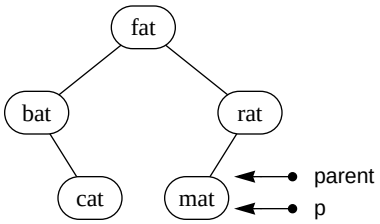


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            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;
```

val
pat

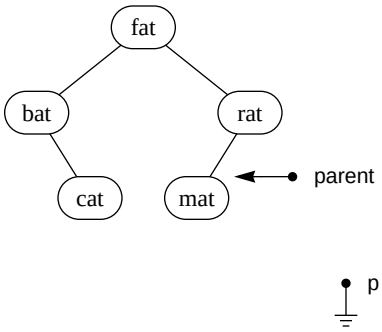


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

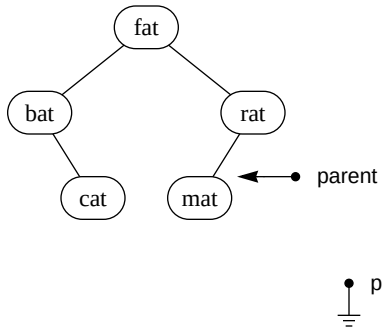


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            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;
```

val
pat

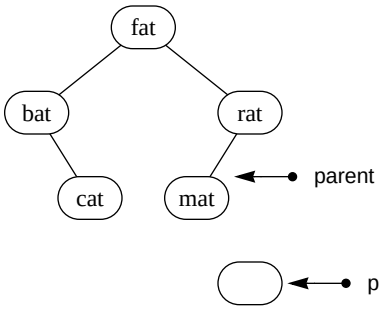


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

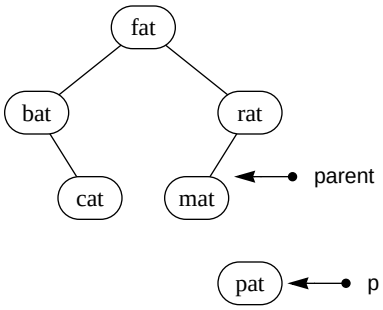


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;
        p: Tree;
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # val, 20);
        IF val < p.root.value THEN
            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;
```

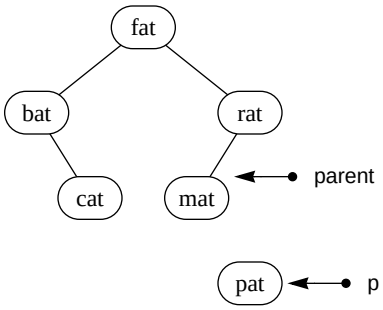
val
pat



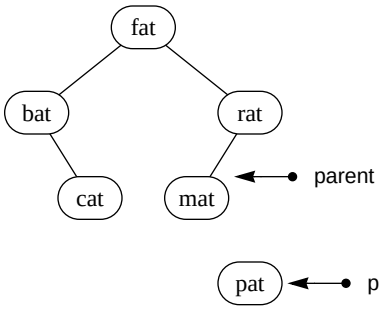
```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```



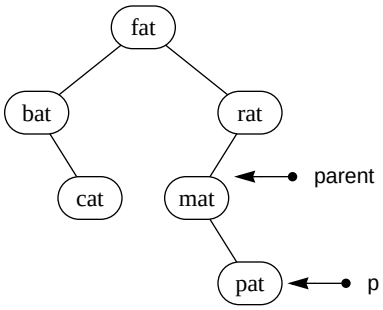
```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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;
```

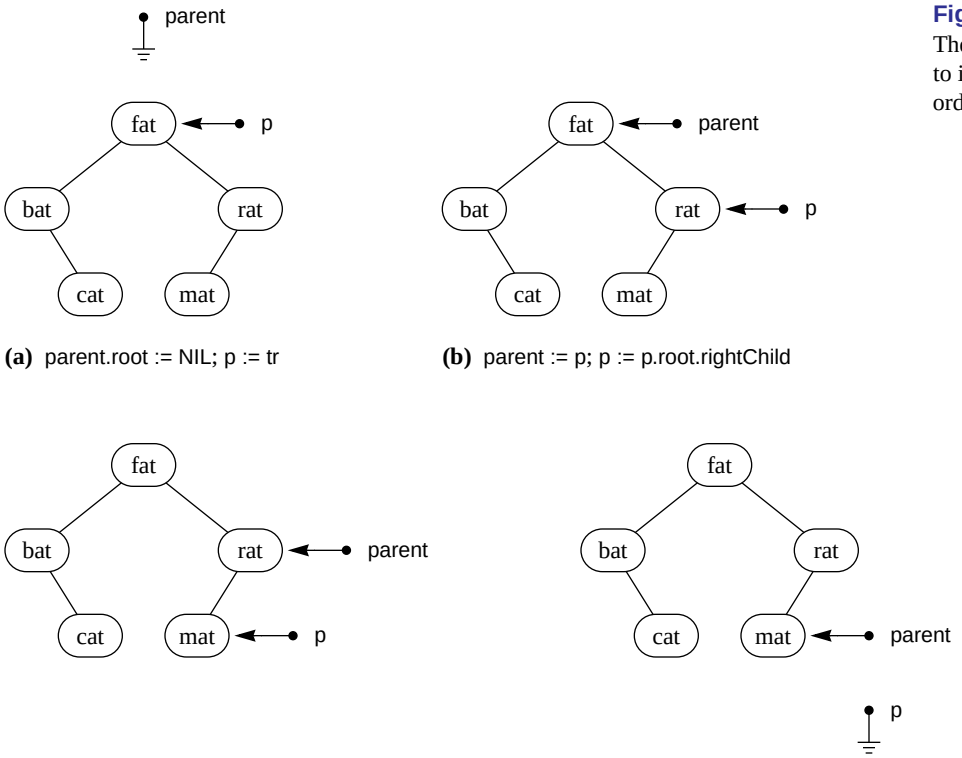


```
PROCEDURE (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
BEGIN
    (* Find insertion point *)
    parent.root := NIL;
    p := tr;
    WHILE p.root # NIL DO
        parent := p;
        ASSERT(p.root.value # 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 (VAR tr: Tree) Insert* (IN val: T), NEW;
    VAR
        parent: Tree;                          val
        p: Tree;                               pat
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;
```





(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

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