## 5.4 Balanced Tree Implementations of Sets

In Sections 5.1 and 5.2 we saw how sets could be implemented by binary search trees, and we saw that operations like INSERT could be performed in time proportional to the average depth of nodes in that tree. Further, we discovered that this average depth is $O(\log n)$ for a "random" tree of $n$ nodes. However, some sequences of insertions and deletions can produce binary search trees whose average depth is proportional to $n$. This suggests that we might try to rearrange the tree after each insertion and deletion so that it is always complete, then the time for INSERT and similar operations would always be $O(\log n)$.

In Fig. 5.12(a) we see a tree of six nodes that becomes the complete tree of seven nodes in Fig. 5.12(b), when element 1 is inserted. Every element in Fig. 5.12(a), however, has a different parent in Fig. 5.12(b), so we must take $n$ steps to insert 1 into a tree like Fig. 5.12(a), if we wish to keep the tree as balanced as possible. It is thus unlikely that simply insisting that the binary search tree be complete will lead to an implementation of a dictionary, priority queue, or other ADT that includes INSERT among its operations, in $O(\log n)$ time.

There are several other approaches that do yield worst case $O(\log n)$ time per operation for dictionaries and priority queues, and we shall consider one of them, called a "2-3 tree," in detail. A *2-3 tree* is a tree with the following two properties.

1. Each interior node has two or three children.
2. Each path from the root to a leaf has the same length.

We shall also consider a tree with zero nodes or one node as special cases of a 2-3 tree.

We represent sets of elements that are ordered by some linear order $<$, as follows. Elements are placed at the leaves; if element $a$ is to the left of

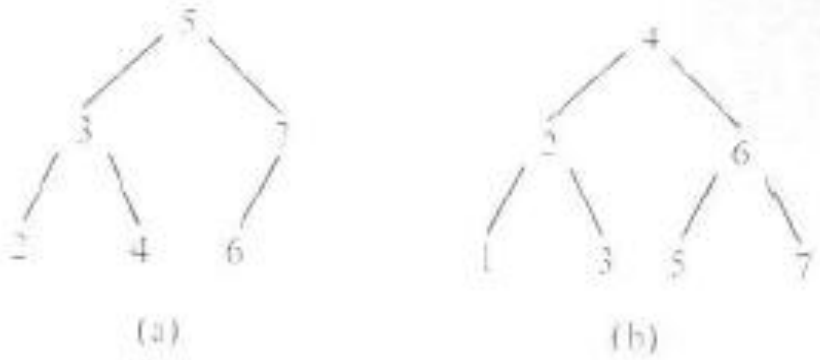



**Fig. 5.12.** Complete trees.

ment $b$, then $a < b$ must hold. We shall assume that the "<" ordering of ments is based on one field of a record that forms the element type; this ld is called the *key*. For example, elements might represent people and cer- n information about people, and in that case, the key field might be "social urity number."

At each interior node we record the key of the smallest element that is a scendant of the second child and, if there is a third child, we record the key the smallest element descending from that child as well.† Figure 5.13 is an mple of a 2-3 tree. In that and subsequent examples, we shall identify an ment with its key field, so the order of elements becomes obvious.

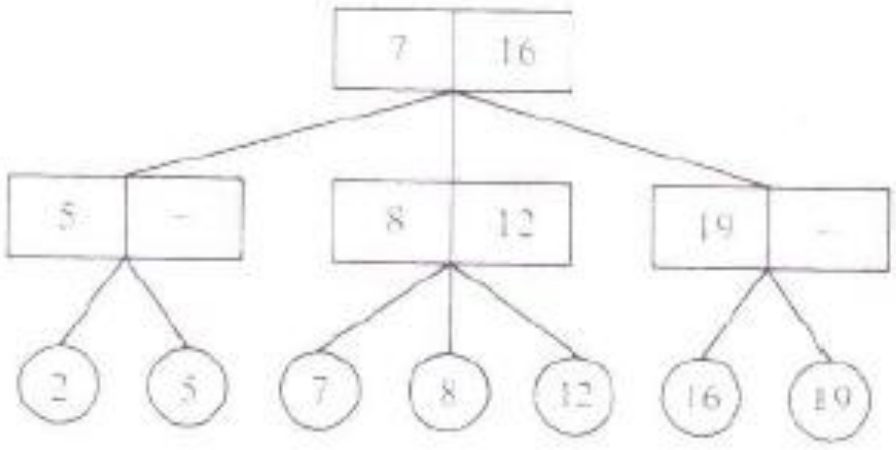



**Fig. 5.13.** A 2-3 tree.

ere is another version of 2-3 trees that places whole records at interior nodes, as a y search tree does.

Observe that a 2-3 tree of $k$ levels has between $2^{k-1}$ and $3^{k-1}$ leaves. Put another way, a 2-3 tree representing a set of $n$ elements requires at least $1 + \log_3 n$ levels and no more than $1 + \log_2 n$ levels. Thus, path lengths in the tree are $O(\log n)$.

We can test membership of a record with key $x$ in a set represented by a 2-3 tree in $O(\log n)$ time by simply moving down the tree, using the values of the elements recorded at the interior nodes to guide our path. At a node *node*, compare $x$ with the value $y$ that represents the smallest element descending from the second child of *node*. (Recall we are treating elements as if they consisted solely of a key field.) If $x < y$, move to the first child of *node*. If $x \geq y$, and *node* has only two children, move to the second child of *node*. If *node* has three children and $x \geq y$, compare $x$ with $z$, the second value recorded at *node*, the value that indicates the smallest descendant of the third child of *node*. If $x < z$, go to the second child, and if $x \geq z$, go to the third child. In this manner, we find ourselves at a leaf eventually, and $x$ is in the represented set if and only if $x$ is at the leaf. Evidently, if during this process we find $x = y$ or $x = z$, we can stop immediately. However, we stated the algorithm as we did because in some cases we shall wish to find the leaf with $x$ as well as to verify its existence.

### Insertion into a 2-3 Tree

To insert a new element $x$ into a 2-3 tree, we proceed at first as if we were testing membership of $x$ in the set. However, at the level just above the leaves, we shall be at a node *node* whose children, we discover, do not include $x$. If *node* has only two children, we simply make $x$ the third child of *node*, placing the children in the proper order. We then adjust the two numbers at *node* to reflect the new situation.

For example, if we insert 18 into Fig. 5.13, we wind up with *node* equal to the rightmost node at the middle level. We place 18 among the children of *node*, whose proper order is 16, 18, 19. The two values recorded at *node* become 18 and 19, the elements at the second and third children. The result is shown in Fig. 5.14.

Suppose, however, that $x$ is the fourth, rather than the third child of *node*. We cannot have a node with four children in a 2-3 tree, so we split *node* into two nodes, which we call *node* and *node'*. The two smallest elements among the four children of *node* stay with *node*, while the two larger elements become children of *node'*. Now, we must insert *node'* among the children of $p$, the parent of *node*. This part of the insertion is analogous to the insertion of a leaf as a child of *node*. That is, if $p$ had two children, we make *node'* the third and place it immediately to the right of *node*. If $p$ had three children before *node'* was created, we split $p$ into $p$ and $p'$, giving $p$ the two leftmost children and $p'$ the remaining two, and then we insert $p'$ among the children of $p$'s parent, recursively.

One special case occurs when we wind up splitting the root. In that case we create a new root, whose two children are the two nodes into which the

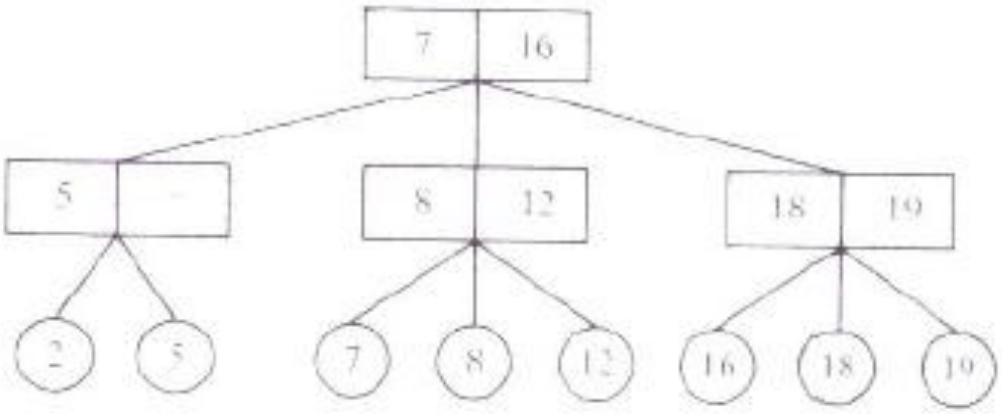



**Fig. 5.14.** 2-3 tree with 18 inserted.

old root was split. This is how the number of levels in a 2-3 tree increases.

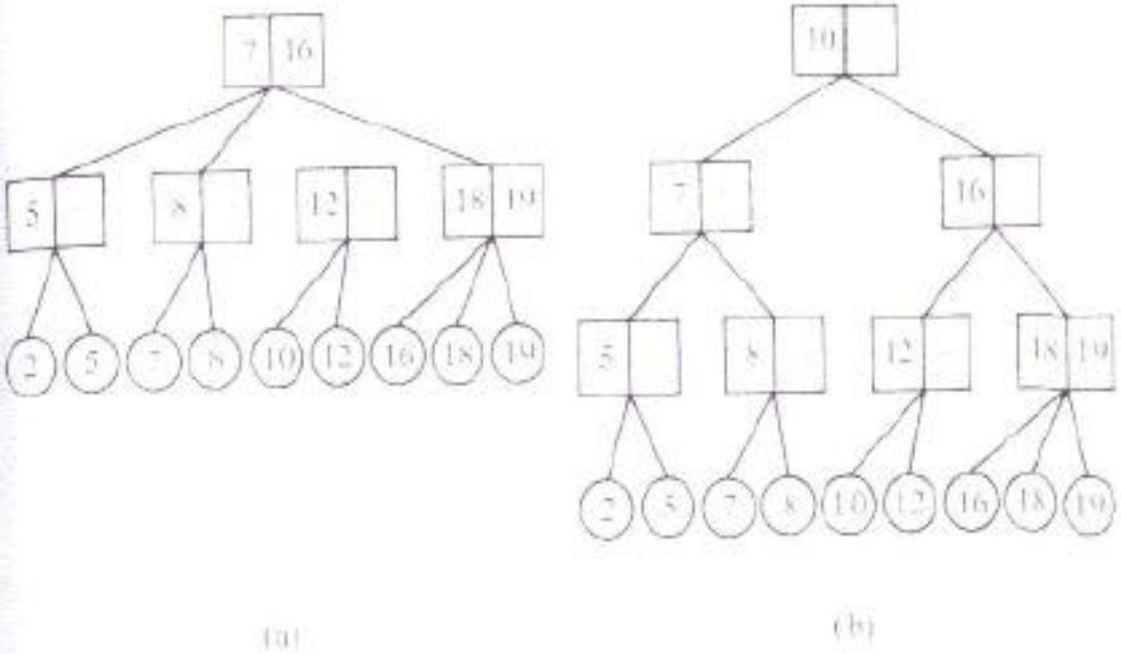
**Example 5.4.** Suppose we insert 10 into the tree of Fig. 5.14. The intended parent of 10 already has children 7, 8, and 12, so we split it into two nodes. The first of these has children 7 and 8; the second has 10 and 12. The result is shown in Fig. 5.15(a). We must now insert the new node with children 10 and 12 in its proper place as a child of the root of Fig. 5.15(a). Doing so gives the root four children, so we split it, and create a new root, as shown in Fig. 5.15(b). The details of how information regarding smallest elements of subtrees is carried up the tree will be given when we develop the program for the command INSERT. □

### Deletion in a 2-3 tree

When we delete a leaf, we may leave its parent *node* with only one child. If *node* is the root, delete *node* and let its lone child be the new root. Otherwise, let $p$ be the parent of *node*. If $p$ has another child, adjacent to *node* on either the right or the left, and that child of $p$ has three children, we can transfer the proper one of those three to *node*. Then *node* has two children, and we are done.

If the children of $p$ adjacent to *node* have only two children, transfer the lone child of *node* to an adjacent sibling of *node*, and delete *node*. Should $p$ now have only one child, repeat all the above, recursively, with $p$ in place of *node*.

**Example 5.5.** Let us begin with the tree of Fig. 5.15(b). If 10 is deleted, its parent has only one child. But the grandparent has another child that has three children, 16, 18, and 19. This node is to the right of the deficient node, so we pass the deficient node the smallest element, 16, leaving the 2-3 tree in Fig. 5.16(a).




(a) (b)

**Fig. 5.15.** Insertion of 10 into the tree of Fig. 5.14.

Next suppose we delete 7 from the tree of Fig. 5.16(a). Its parent now has only one child, 8, and the grandparent has no child with three children. We therefore make 8 be a sibling of 2 and 5, leaving the tree of Fig. 5.16(b). Now the node starred in Fig. 5.16(b) has only one child, and its parent has no other child with three children. Thus we delete the starred node, making its child be a child of the sibling of the starred node. Now the root has only one child, and we delete it, leaving the tree of Fig. 5.16(c).

Observe in the above examples, the frequent manipulation of the values at interior nodes. While we can always calculate these values by walking the tree, it can be done as we manipulate the tree itself, provided we remember the smallest value among the descendants of each node along the path from the root to the deleted leaf. This information can be computed by a recursive deletion algorithm, with the call at each node being passed, from above, the correct quantity (or the value "minus infinity" if we are along the leftmost path). The details require careful case analysis and will be sketched later when we consider the program for the DELETE operation. □

### Data Types for 2-3 Trees

Let us restrict ourselves to representing by 2-3 trees sets of elements whose keys are real numbers. The nature of other fields that go with the field key, to make up a record of type elementtype, we shall leave unspecified, as it has no bearing on what follows.

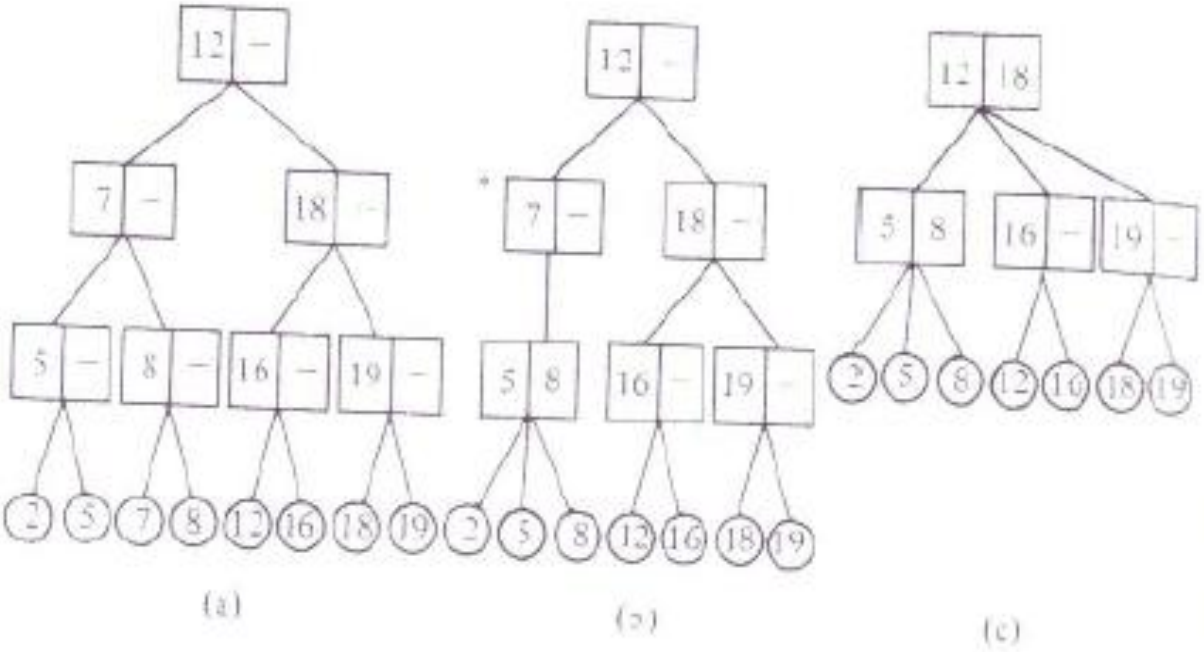



**Fig. 5.16.** Deletion in a 2-3 tree.

In Pascal, the parents of leaves must be records consisting of two reals (the keys of the smallest elements in the second and third subtrees) and of three pointers to elements. The parents of these nodes are records consisting of two reals and of three pointers to parents of leaves. This progression continues indefinitely; each level in a 2-3 tree is of a different type from all other levels. Such a situation would make programming 2-3 tree operations in Pascal impossible, but fortunately, Pascal provides a mechanism, the variant record structure, that enables us to regard all 2-3 tree nodes as having the same type, even though some are elements, and some are records with pointers and reals.† We can define nodes as in Fig. 5.17. Then we declare a set, represented by a 2-3 tree, to be a pointer to the root as in Fig. 5.17.

## Implementation of INSERT

The details of operations on 2-3 trees are quite involved, although the principles are simple. We shall therefore describe only one operation, insertion, in detail; the others, deletion and membership testing, are similar in spirit, and finding the minimum is a trivial search down the leftmost path. We shall write the insertion routine as main procedure, INSERT, which we call at the root, and a procedure *insert*1, which gets called recursively down the tree.

† All nodes, however, take the largest amount of space needed for any of the variant types, so Pascal is not really the best language for implementing 2-3 trees in practice.

```
type
    elementtype = record
        key: real;
        {other fields as warranted}
    end;
    nodetypes = (leaf, interior);
    twothreenode = record
        case kind: nodetypes of
            leaf: (element: elementtype);
            interior: (firstchild, secondchild, thirdchild: ↑twothreenode;
                lowofsecond, lowofthird: real)
    end;
    SET = ↑twothreenode;
```

**Fig. 5.17.** Definition of a node in a 2-3 tree.

For convenience, we assume that a 2-3 tree is not a single node or empty. These two cases require a straightforward sequence of steps which the reader is invited to provide as an exercise.

```
procedure insert1 ( node: ↑twothreenode;
    x: elementtype; { x is to be inserted into the subtree of node }
    var pnew: ↑twothreenode; { pointer to new node created to right of node }
    var low: real ); { smallest element in the subtree pointed to by pnew }

    begin
        pnew := nil;
        if node is a leaf then begin
            if x is not the element at node then begin
                create new node pointed to by pnew;
                put x at the new node;
                low := x.key
            end
        end
        else begin { node is an interior node }
            let w be the child of node to whose subtree x belongs;
            insert1(w, x, pback, lowback);
            if pback <> nil then begin
                insert pointer pback among the children of node just
                    to the right of w;
                if node has four children then begin
                    create new node pointed to by pnew;
                    give the new node the third and fourth children of node;
                    adjust lowofsecond and lowofthird in node and the new node;
```

```
                set low to be the lowest key among the
                    children of the new node
            end
        end
    end
end; { insert1 }
```

**Fig. 5.18.** Sketch of 2-3 tree insertion program.

We would like *insert*1 to return both a pointer to a new node, if it must create one, and the key of the smallest element descended from that new node. As the mechanism in Pascal for creating such a function is awkward, we shall instead declare *insert*1 to be a procedure that assigns values to parameters *pnew* and *low* in the case that it must "return" a new node. We sketch *insert*1 in Fig. 5.18. The complete procedure is shown in Fig. 5.19, some comments in Fig. 5.18 have been omitted from Fig. 5.19 to save space.

```
procedure insert1 ( node : ↑ twothreenode; x : elementtype;
    var pnew : ↑ twothreenode; var low : real );

    var
        pback : ↑ twothreenode;
        lowback : real;
        child : 1..3; { indicates which child of node is followed
            in recursive call (cf. w in Fig. 5.18) }
        w : ↑ twothreenode; { pointer to the child }

    begin
        pnew := nil;
        if node↑.kind = leaf then begin
            if node↑.element.key <> x.key then begin
                { create new leaf holding x and "return" this node }
                new(pnew, leaf);
                if (node↑.element.key < x.key) then
                    { place x in new node to right of current node }
                    begin pnew↑.element := x; low := x.key end
                else begin { x belongs to left of element at current node }
                    pnew↑.element := node↑.element;
                    node↑.element := x;
                    low := pnew↑.element.key
                end
            end
        end
        else begin { node is an interior node }
            { select the child of node that we must follow }
            if x.key < node↑.lowofsecond then
                begin child := 1; w := node↑.firstchild end
```

```
else if (node↑.thirdchild = nil) or (x.key < node↑.lowofthird) then begin
    { x is in second subtree }
    child := 2;
    w := node↑.secondchild
end
else begin { x is in third subtree }
    child := 3;
    w := node↑.thirdchild
end;
insert1(w, x, pback, lowback);
if pback <> nil then
    { a new child of node must be inserted }
    if node↑.thirdchild = nil then
        { node had only two children, so insert new node in proper place }
        if child = 2 then begin
            node↑.thirdchild := pback;
            node↑.lowofthird := lowback
        end
        else begin { child = 1 }
            node↑.thirdchild := node↑.secondchild;
            node↑.lowofthird := node↑.lowofsecond;
            node↑.secondchild := pback;
            node↑.lowofsecond := lowback
        end
    else begin { node already had three children }
        new(pnew, interior);
        if child = 3 then begin
            { pback and third child become children of new node }
            pnew↑.firstchild := node↑.thirdchild;
            pnew↑.secondchild := pback;
            pnew↑.thirdchild := nil;
            pnew↑.lowofsecond := lowback;
                { lowofthird is undefined for pnew }
            low := node↑.lowofthird;
            node↑.thirdchild := nil
        end
        else begin { child ≤ 2; move third child of node to pnew }
            pnew↑.secondchild := node↑.thirdchild;
            pnew↑.lowofsecond := node↑.lowofthird;
            pnew↑.thirdchild := nil;
            node↑.thirdchild := nil
        end;
        if child = 2 then begin
            { pback becomes first child of pnew }
            pnew↑.firstchild := pback;
```

```
                low := lowback
            end;
            if child = 1 then begin
                { second child of node is moved to pnew;
                    pback becomes second child of node }
                pnew↑.firstchild := node↑.secondchild;
                low := node↑.lowofsecond;
                node↑.secondchild := pback;
                node↑.lowofsecond := lowback
            end
        end
    end
end; { insert1 }
```

**Fig. 5.19.** The procedure *insert*1.

Now we can write the procedure INSERT, which calls *insert*1. If *insert*1 "returns" a new node, then INSERT must create a new root. The code is shown in Fig. 5.20 on the assumption that the type SET is ↑twothreenode, i.e., a pointer to the root of a 2-3 tree whose leaves contain the members of the set.

```
procedure INSERT ( x: elementtype; var S: SET );
    var
        pback: ↑twothreenode; { pointer to new node returned by insert1 }
        lowback: real; { low value in subtree of pback }
        saveS: SET; { place to store a temporary copy of the pointer S }
    begin
        { checks for S being empty or a single node should occur here,
            and an appropriate insertion procedure should be included }
        insert1(S, x, pback, lowback);
        if pback <> nil then begin
            { create new root; its children are now pointed to by S and pback }
            saveS := S;
            new(S);
            S↑.firstchild := saveS;
            S↑.secondchild := pback;
            S↑.lowofsecond := lowback;
            S↑.thirdchild := nil;
        end
    end; { INSERT }
```

**Fig. 5.20.** INSERT for sets represented by 2-3 trees.

### Implementation of DELETE

We shall sketch a function *delete*1 that takes a pointer to a node *node* and an element $x$, and deletes a leaf descended from *node* having value $x$, if there is one.† Function *delete*1 returns true if after deletion *node* has only one child, and it returns false if *node* still has two or three children. A sketch of the code for *delete*1 is shown in Fig. 5.21.

We leave the detailed code for function *delete*1 for the reader. Another exercise is to write a procedure DELETE($S$, $x$) that checks for the special cases that the set $S$ consists only of a single leaf or is empty, and otherwise calls *delete*1($S$, $x$); if *delete*1 returns true, the procedure removes the root (the node pointed to by $S$) and makes $S$ point to its lone child.

```
function delete1 ( node: ↑ twothreenode; x: elementtype ) : boolean;
    var
        onlyone: boolean; { to hold the value returned by a call to delete1 }
    begin
        delete1 := false;
        if the children of node are leaves then begin
            if x is among those leaves then begin
                remove x;
                shift children of node to the right of x one position left;
                if node now has one child then
                    delete1 := true
            end
        end
        else begin { node is at level two or higher }
            determine which child of node could have x as a descendant;
            onlyone := delete1(w, x); { w stands for node↑.firstchild,
                node↑.secondchild, or node↑.thirdchild, as appropriate }
            if onlyone then begin { fix children of node }
                if w is the first child of node then
                    if y, the second child of node, has three children then
                        make the first child of y be the second child of w
                    else begin { y has two children }
                        make the child of w be the first child of y;
                        remove w from among the children of node;
                        if node now has one child then
                            delete1 := true
                    end;
                if w is the second child of node then
                    if y, the first child of node, has three children then
                        make the third child of y be the first child of w
```

† A useful variant would take only a key value and delete any element with that key.

```
            else { y has two children }
                if z, the third child of node, exists
                    and has three children then
                        make first child of z be the second child of w
                else begin { no other child of node has three children }
                    make the child of w be the third child of y;
                    remove w from among the children of node;
                    if node now has one child then
                        delete1 := true
                end;
            if w is the third child of node then
                if y, the second child of node, has three children then
                    make the third child of y be the second child of w
                else begin { y has two children }
                    make the child of w be the third child of y;
                    remove w from among the children of node
                end { note node surely has two children left in this case }
        end
    end
end; { delete1 }
```

Fig. 5.21. Recursive deletion procedure.