Algorithms

Lecture13

13.1: What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field, each node contains fields *left, right,* and *p* that point to the nodes corresponding to its left child, its right child, and its parent. If a child or the parent is missing, the appropriate field contains the value NIL.

The root node is the only node in the tree whose parent field is NIL.

13.1: What is a binary search tree?

A ***rooted tree*** is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the ***root*** of the tree.

Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an ***ancestor*** of x. If y is a/n (proper) ancestor of x, then x is a (proper) descendant of y. (Every node is both a/n (non-proper) ancestor and a (non-proper) descendant of itself.) The ***subtree rooted at x*** is the tree induced by the descendants of x, rooted at x.

13.1: What is a binary search tree?

If the last edge on the path from the root r of the tree T to a node x is (y,x), then y is a parent of x, and x is a ***child*** of y. The root is the only node in T with no parent. If two nodes have the same parent, they are ***siblings.*** A node with no children is an ***external node*** or ***leaf.*** A nonleaf node is an ***internal node.***

The number of children of a node x in a rooted tree T is called the ***degree*** of x. The length of the path from the root r to a node x is the ***depth*** of x in T. The largest depth of any node in T is the ***height*** of T.

13.1: What is a binary search tree?

Binary trees are best described recursively. A ***binary tree*** *T* is a structure defined on a finite set of nodes that either

• contains no nodes, or

• is comprised of three disjoint sets of nodes: a ***root*** node, a binary tree called its ***left subtree***, and a binary tree called its ***right subtree.***

The binary tree that contains no nodes is called the ***empty tree,*** sometimes denoted NIL. If the left (right) subtree is nonempty, its root is called the ***left (right) child*** of the root of the entire tree. If a subtree is the empty tree, we say that the child is ***absent*** or ***missing***.

13.1: What is a binary search tree?

In a ***full binary tree*** each node is either a leaf or has degree exactly 2. Any binary tree can be transformed into a full binary tree: we replace each missing child in the binary tree with a node having no children. These new leaf nodes are usually drawn as squares in the figures (and are implemented as NILs).

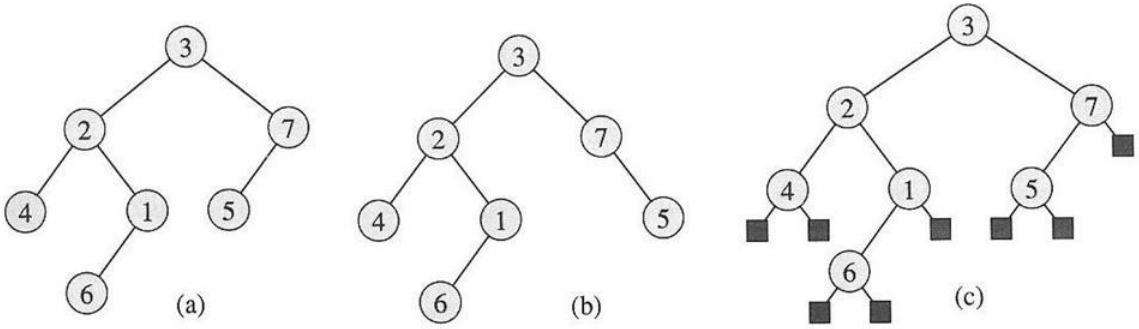

Figure 5.7 Binary trees. **(a)** A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. The right child is drawn beneath and to the right. **(b)** A binary tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct. **(c)** The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 2. The leaves in the tree are shown as squares.

13.1: What is a binary search tree?

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree-property:***

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[y] \geq key[x]$.

13.1: What is a binary search tree?

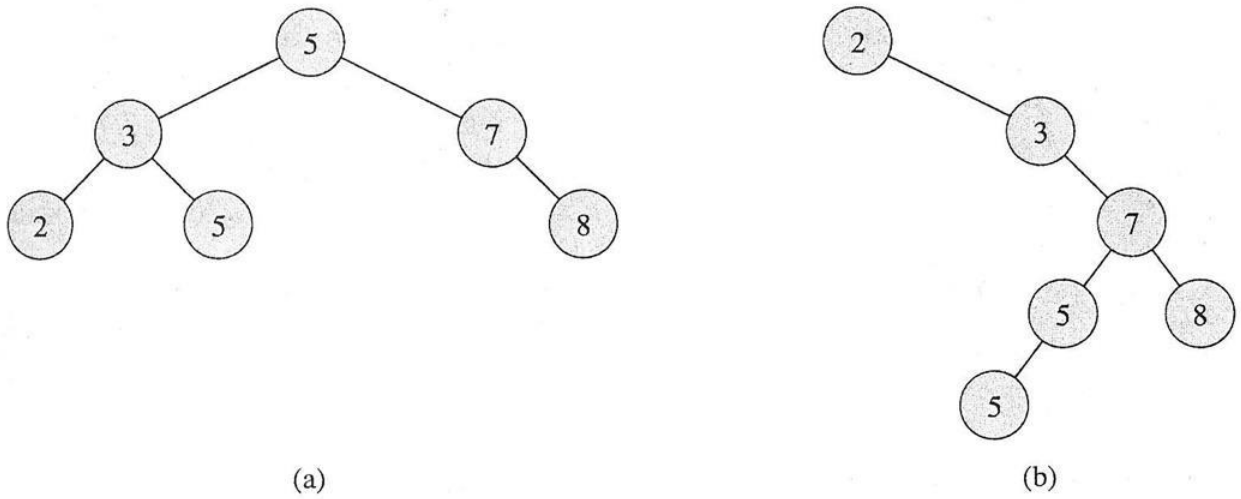

Figure 13.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (**a**) A binary search tree on 6 nodes with height 2. (**b**) A less efficient binary search tree with height 4 that contains the same keys.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called the ***inorder tree walk.*** This algorithm derives its name from the fact that the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree.

To use the following procedure on a given tree *T,* we call

Inorder-Tree-Walk(*root*[*T*]).

13.1: What is a binary search tree?

```
INORDER-TREE-WALK (x)
    if x ≠ NIL
      then INORDER-TREE-WALK (left[x])
           print key[x]
           INORDER-TREE-WALK (right[x])
```

13.1: What is a binary search tree?

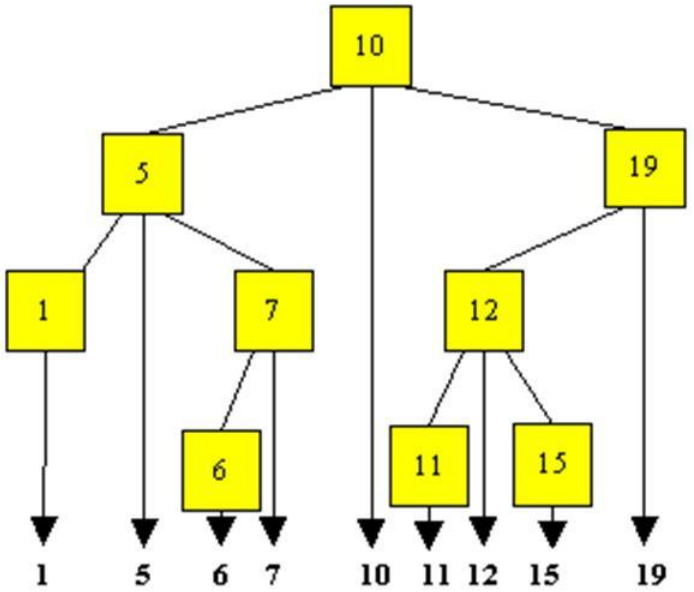

Fig. 2

Similar definitions exist for the ***preorder tree walk*** and the ***postorder tree walk.***

13.2: Querying a binary search tree

The most common operation performed on a binary search tree is (as expected) searching for a key stored in the tree. Beside the SEARCH operation, binary search trees can support queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

We will show: each of these may be supported in time $O(h)$ on a binary search tree of height h.

13.2: Querying a binary search tree - Searching

The following procedure searches for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k*, TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

```
TREE-SEARCH (x,k)
    if x = NIL or k = key[x]
      then return x
    if k < key[x]
      then return TREE-SEARCH (left[x],k)
      else return TREE-SEARCH (right[x],k)
```

13.2: Querying a binary search tree - Searching

It may also be done by the following iterative procedure:

ITERATIVE-TREE-SEARCH (x,k)

```
while x ≠ NIL and k ≠ key[x]
do if k < key[x]
    then x ← left [x]
    else x ← right[x]
return x
```

13.2: Querying a binary search tree - Searching

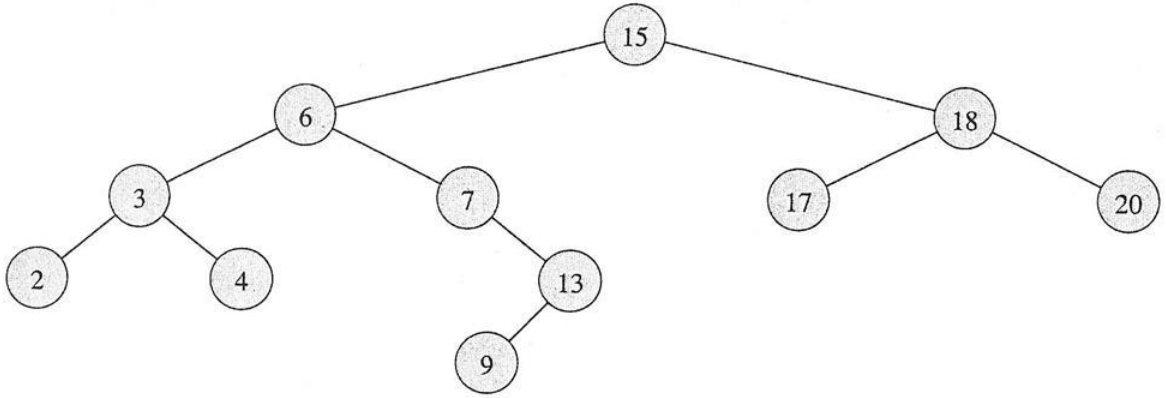

Figure 13.2 Queries on a binary search tree. To search for the key 13 in the tree, the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ is followed from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

13.2: Querying a binary search tree – Minimum & Maximum

An element in a binary search tree whose key is a minimum can always be found following *left* from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node *x*.

TREE-MINIMUM *(x)*

while *left* [*x*] ≠ NIL

do *x* ← *left* [*x*]

return *x*

… and similarly for the maximal element (with *right* [*x*]).

13.2: Querying a binary search tree – Successor and Predecessor

Given a node in a binary search tree, it is important to find its successor in the sorted order. If all keys are distinct, the successor of a node x is the node with the smallest key greater then $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.

The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

13.2: Querying a binary search tree – Successor and Predecessor

```
TREE-SUCCESSOR (x)
    if right [x] ≠ NIL
       then return TREE-MINIMUM (right[x])
    y ← p[x]
    while y ≠ NIL and x = right [y]
         do x ← y
            y ← p[y]
    return y
```

Predecessor – symmetric to Successor.

13.2: Querying a binary search tree

Theorem:

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in $O(h)$ time on a binary search tree of height h.

13.3: Insertion and Deletion

It is quite easy to insert a new value into a binary search tree. It is always added as a new leaf, in an appropriate position, as described by the following procedure:

```
TREE-INSERT (T,z)
    y ← NIL
    x ← root[T]
    while x ≠ NIL
        do y ← x
            if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
    p[z] ← y
```

The first thing we do is to trace a path downwards from the root. y is always the parent of x.

$p[z] \leftarrow y$

if y = NIL

  then $root[T] \leftarrow z$

  else if $key[z] < key[y]$

    then $left[y] \leftarrow z$

    else $right[y] \leftarrow z$

z is placed in the right position by setting the pointers correctly.

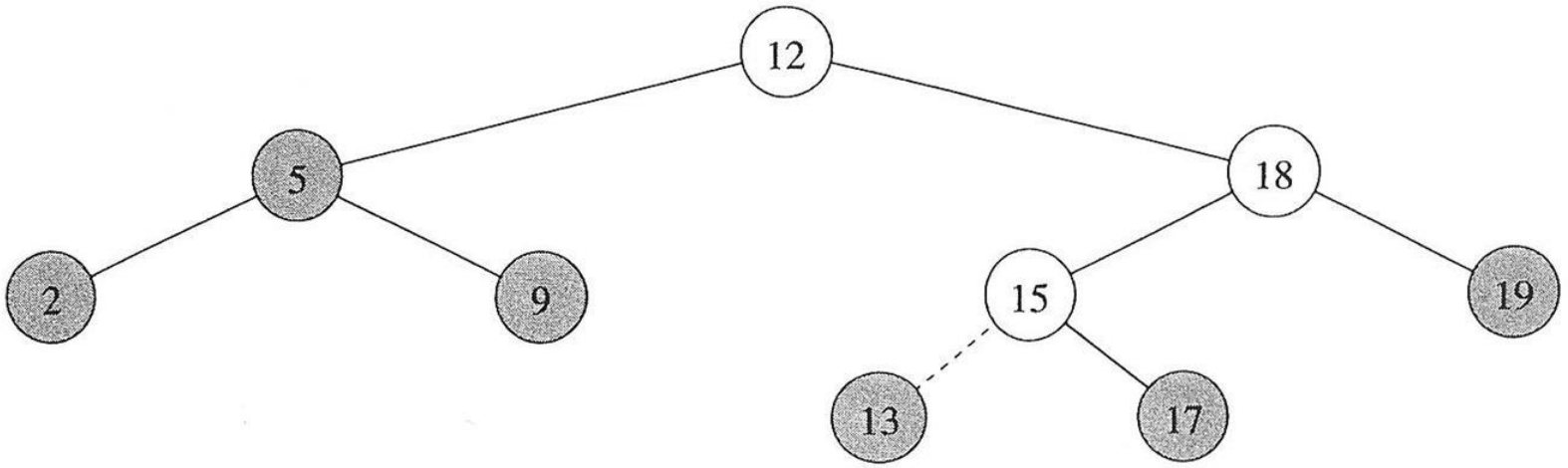

Figure 13.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

13.3: Insertion and Deletion

The procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

13.3: Insertion and Deletion

The procedure TREE-DELETE also runs in $O(h)$ time on a tree of height h. However, its description is somewhat more complicated.

The procedure's input is a pointer to a given node z to be deleted. There are three cases to consider, depending on the number of children of z:

0. If z is a leaf then it is simply deleted.
1. If z has only one child, then z is spliced out and its only child becomes the child of its parent.
2. If z has two children, z's successor, y, which has no left child (why??) is spliced out, and the contents of y replaces the contents of z.

```
TREE-DELETE (T,z)
    if left[z] = NIL or right[z] = NIL
        then y ← z
      else y ← TREE-SUCCESSOR (z)
    if left[y] ≠ NIL
        then x ← left[y]
      else x ← right[y]
    if x ≠ NIL
        then p[x] ← p[y]
```

```
if p[y] = NIL
    then root[T] ← x
  else if y = left[p[y]]
          then left[p[y]] ← x
          else right[p[y]] ← x
if y ≠ z
  then key[z] ← key[y]
return y
```

If *y* has other fields, copy them too.

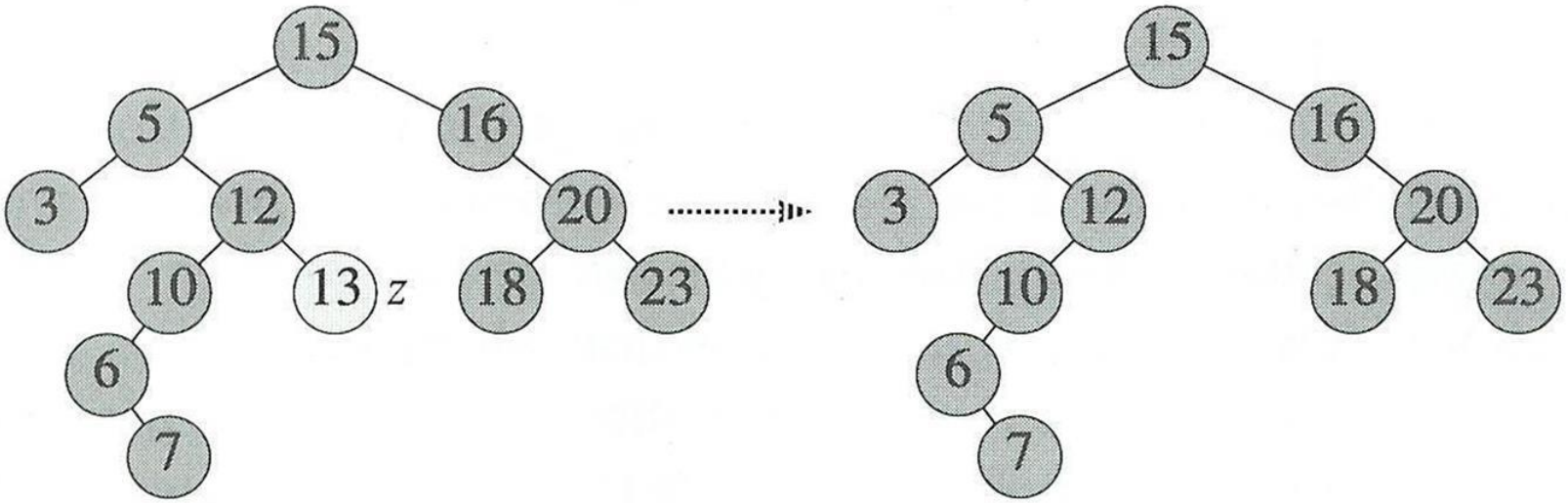

Figure 13.4 Deleting a node z from a binary search tree. **(a)** If z has no children, we just remove it.

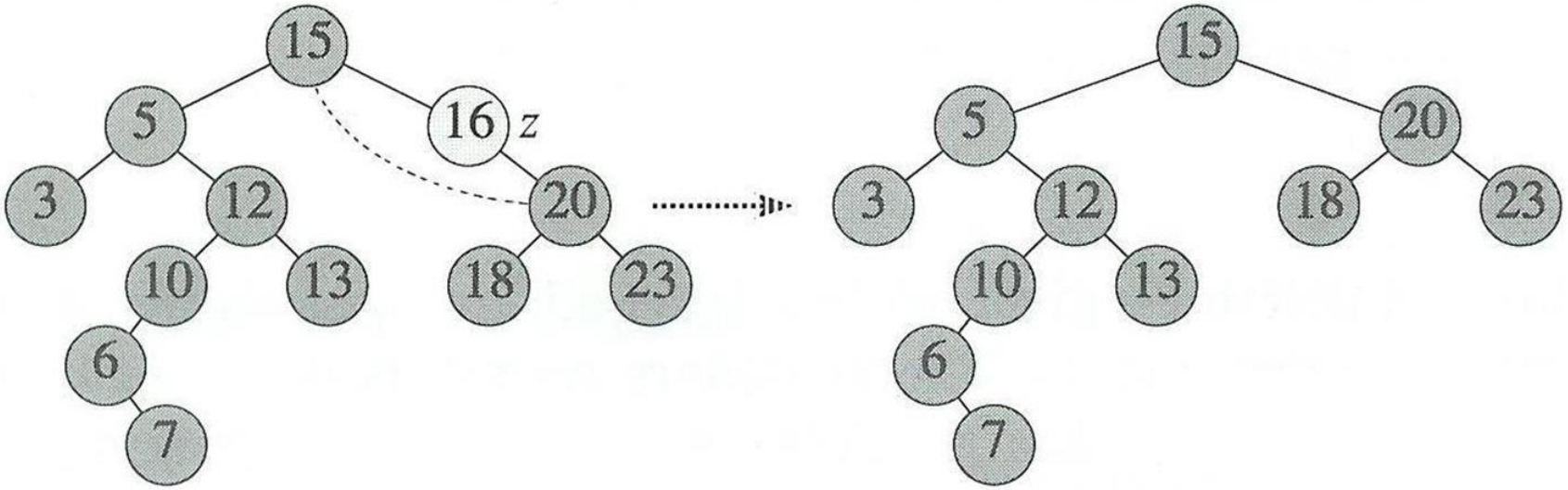

Figure 13.4 Deleting a node z from a binary search tree. (b) If z has only one child, we splice out z.

13.3: Insertion and Deletion

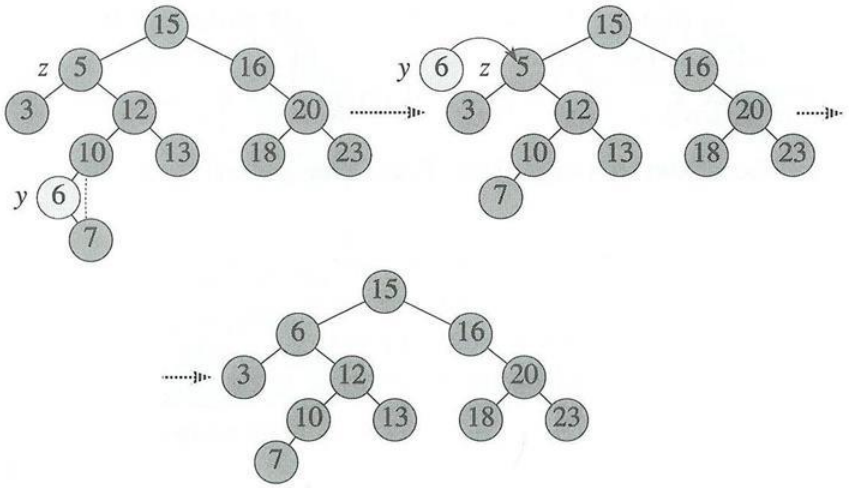

Figure 13.4 Deleting a node z from a binary search tree.
(c) If z has two children, we splice out its successor y, which has at most one child, and then replace the contents of z with the contents of y.

The procedure TREE-DELETE runs in *O(h)* time on a tree of height *h*.