International Islamic University Chittagong
Department of Computer Science & Engineering
Spring - 2021

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

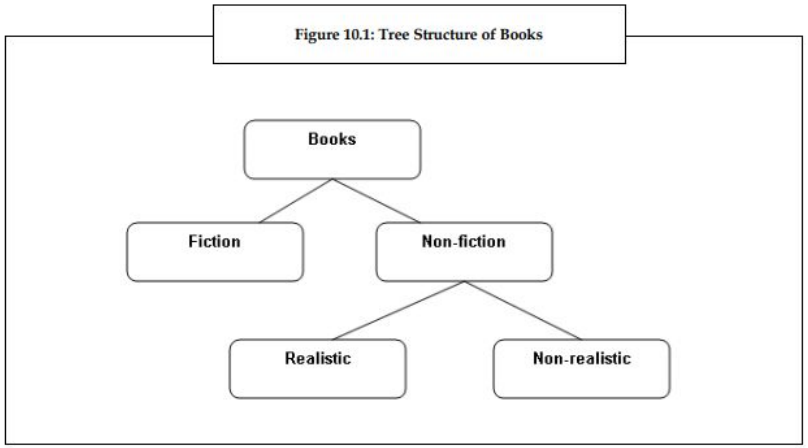
Lecture – 14

Tree

Tree

A tree is a very important **non linear data structure** as it is useful in many applications. A tree structure is a method of representing the **hierarchical** nature of a structure in a graphical form. It is termed as "tree structure" since its representation resembles a tree. However, the chart of a tree is normally upside down compared to an actual tree, with the **root** at the *top* and the **leaves** at the *bottom*.

The figure 10.1 depicts a tree structure, which represents the hierarchical organization of **books**.



In the hierarchical organization of books shown in figure 10.1, Books is the **root** of the tree. Books can be classified as Fiction and Non-fiction. Non-fiction books can be further classified as Realistic and Non-realistic, which are the **leaves** of the tree. Thus, it forms a complete tree structure.

Binary Tree

- A **binary tree** is a tree in which each node can have ***at most two children***.
- A binary tree is either *empty* or consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree*.
- A tree with no nodes is called as a *null tree*.

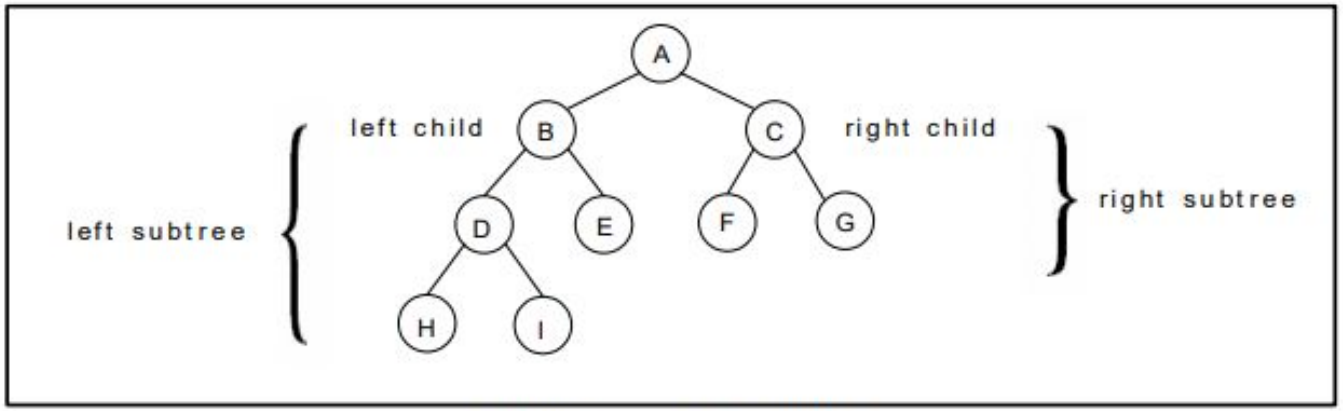


Figure . Binary Tree

- A Binary Tree T is defined as finite set of elements, called nodes, such that:

  a) T is empty (called the null tree or empty tree.)

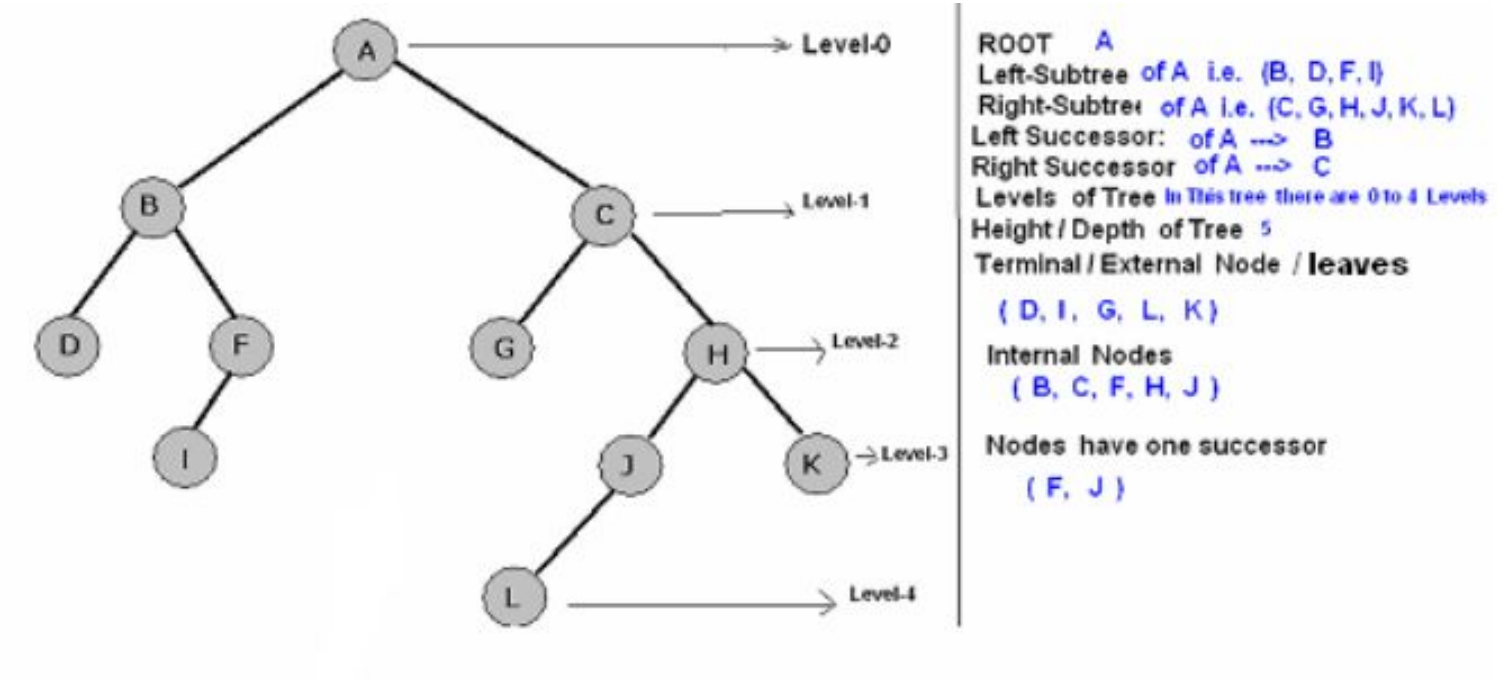
  b) T contains a distinguished node R, called the **root** of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

If T does contain a root R, then the two trees T_1 and T_2 are called, respectively, the *left sub tree* and *right sub tree* of R. If T_1 is non empty, then its root is called the *left successor* of R; similarly, if T_2 is non empty, then its root is called the *right successor* of R. The nodes with no successors are called the **terminal** nodes.

Tree Terminology

- If N is a node in T with left successor S_1 and right successor S_2, then N is called the **parent**(or father) of S_1 and S_2. Analogously, S_1 is called the **left child** (or son) of N, and S_2 is called the **right child** (or son) of N. Furthermore, S_1 and S_2 are said to **siblings** (or brothers). Every node in the binary tree T, except the root, has a *unique* parent, called the **predecessor** of N.
- The terms descendant and ancestor have their usual meaning. A node L is called a **descendant** of a node N (and N is called an **ancestor** of L) if there I a succession of children from N to L.
- The line drawn from a node N of T to a successor is called an **edge**, and a sequence of consecutive edges is called a **path**. A terminal node is called a **leaf**, and a path ending in a leaf is called a **branch**.
- Each node in binary tree T is assigned a **level number**, as follows. The root R of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent. *The maximum number of nodes at any level is* 2^n. Those nodes with the same level number are said to belong to the **same generation**.
- The **depth** (or **height**) of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T.

Tree Terminology



- Binary trees T and T' are said to be **similar** if they have the same structure or, in other words, if they have the same shape. The trees are said to be **copies** if they are similar and if they have the same contents at corresponding nodes.

Consider the four binary trees in Fig. 7.2. The three trees (a), (c) and (d) are similar. In particular, the trees (a) and (c) are copies since they also have the same data at corresponding nodes. The tree (b) is neither similar nor a copy of the tree (d) because, in a binary tree, we distinguish between a left successor and a right successor even when there is only one successor.

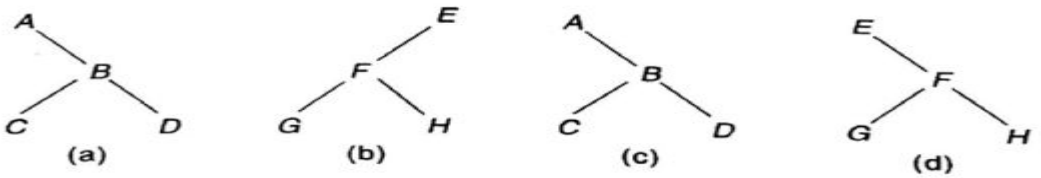
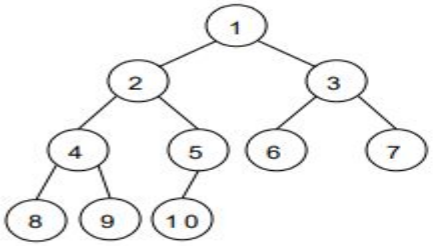


Fig. 7.2

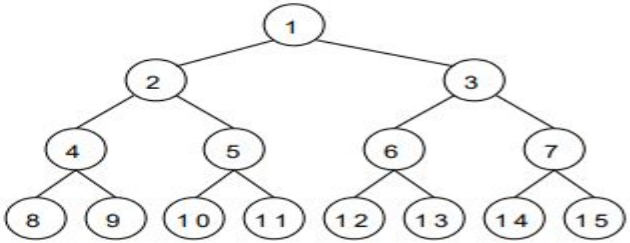
Tree Terminology

Full Binary tree

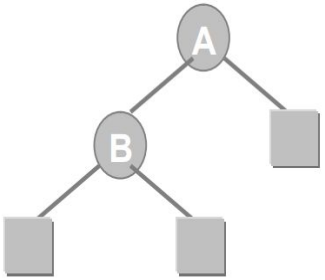
A full binary tree of height **h** has all its leaves at level **h**. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children. A full binary tree with height **h** has $\mathbf{2^h - 1}$ nodes.



Complete binary tree



Full binary tree



Extended 2-tree

Complete Binary Tree:

Consider a binary tree T. The tree T is said to be complete if all its levels, except possibly the last have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

Extended Binary Tree / 2-Tree:

A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes, with 2 children are called **internal nodes**, and the node with 0 children are called **external node**.

Representing Binary Trees in Memory

Let T be a binary tree. Here we will discuss two ways of representing T in memory-

I. Linked Representation of Binary Tree
II. Sequential Representation of Binary Tree

Linked Representation of Binary Tree

Consider a binary tree T. Unless otherwise stated or implied, T will be maintained in memory by means of a *linked representation* which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

(1) INFO[K] contains the data at the node N.
(2) LEFT[K] contains the location of the left child of node N.
(3) RIGHT[K] contains the location of the right child of node N.

Furthermore, ROOT will contain the location of the root R of T. If any subtree is empty, then the corresponding pointer will contain the null value; if the tree T itself is empty, then ROOT will contain the null value.

Representing Binary Trees in Memory

Example 7.3

Consider the binary tree T in Fig. 7.1. A schematic diagram of the linked representation of T appears in Fig. 7.6. Observe that each node is pictured with its three fields, and that the empty subtrees are pictured by using × for the null entries. Figure 7.7 shows how this linked representation may appear in memory. The choice of 20 elements for the arrays is arbitrary. Observe that the AVAIL list is maintained as a one-way list using the array LEFT.

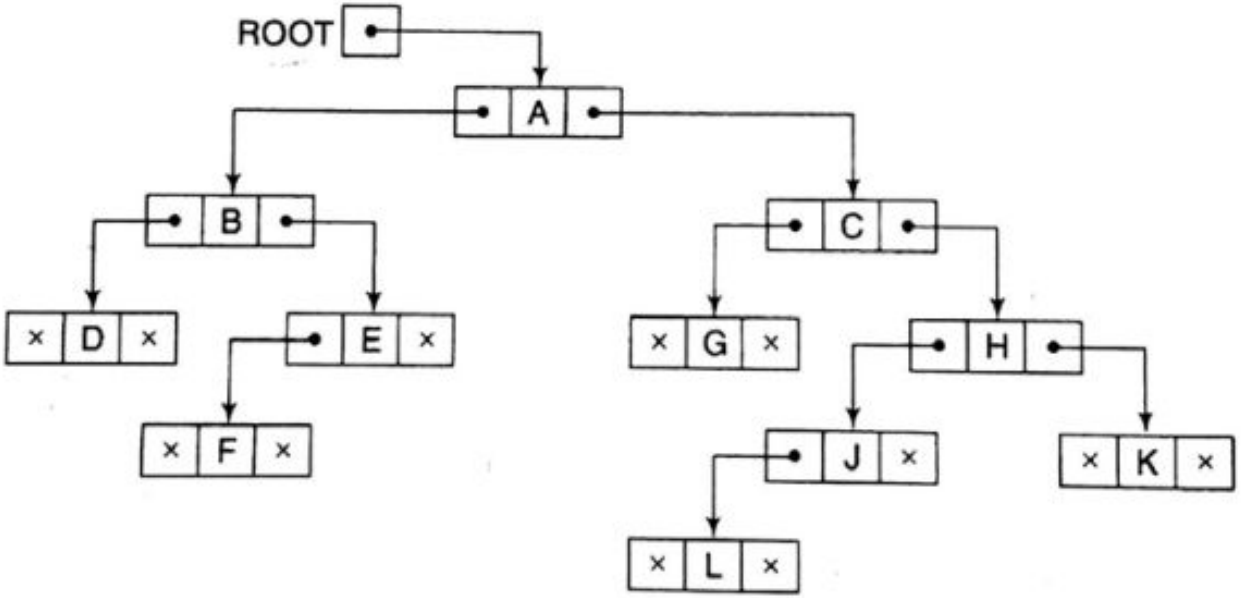


Fig. 7.6

Representing Binary Trees in Memory

Sequential Representation of Binary Tree

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining T in memory called the *sequential representation* of T. This representation uses only a single linear array TREE as follows:

(a) The root R of T is stored in TREE[1].

(b) If a node N occupies TREE[K], then its left child is stored in TREE[2 * K] and its right child is stored in TREE[2 * K + 1].

Again, NULL is used to indicate an empty subtree. In particular, TREE[1] = NULL indicates that the tree is empty.

The sequential representation of the binary tree T in Fig. 7.10(a) appears in Fig. 7.10(b).

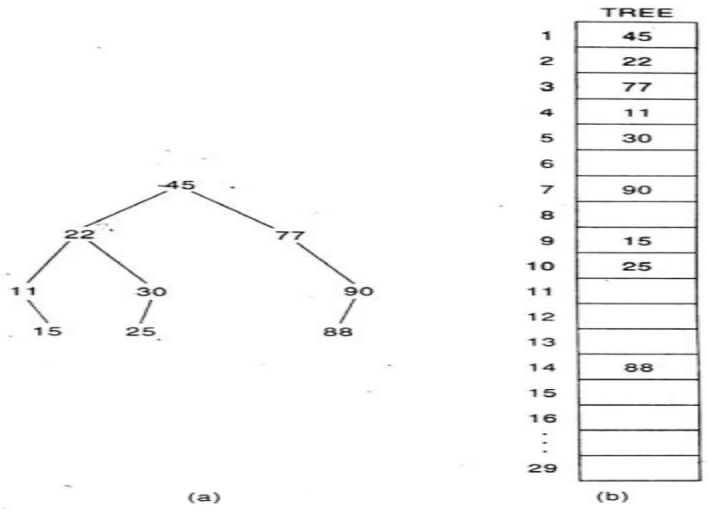


Fig. 7.10

Representing Binary Trees in Memory

- Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.
- The advantage of using linked representation of binary tree is that: Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.
- The disadvantages of linked representation of binary tree includes: Given a node structure, it is difficult to determine its parent node. Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

Traversing Binary Trees

A traversal of a tree T is a systematic way of accessing or visiting all the node of T. There are three standard ways of traversing a binary tree T with root R. these are :

1**. Preorder (N L R):**

a) Process the node/root.

b) Traverse the Left sub tree.

c) Traverse the Right sub tree.

2. **Inorder (L N R):**

a) Traverse the Left sub tree.

b) Process the node/root.

c) Traverse the Right sub tree.

3. **Postorder (L R N):**

a) Traverse the Left sub tree.

b) Traverse the Right sub tree.

c) Process the node/root.

Traversing Binary Trees

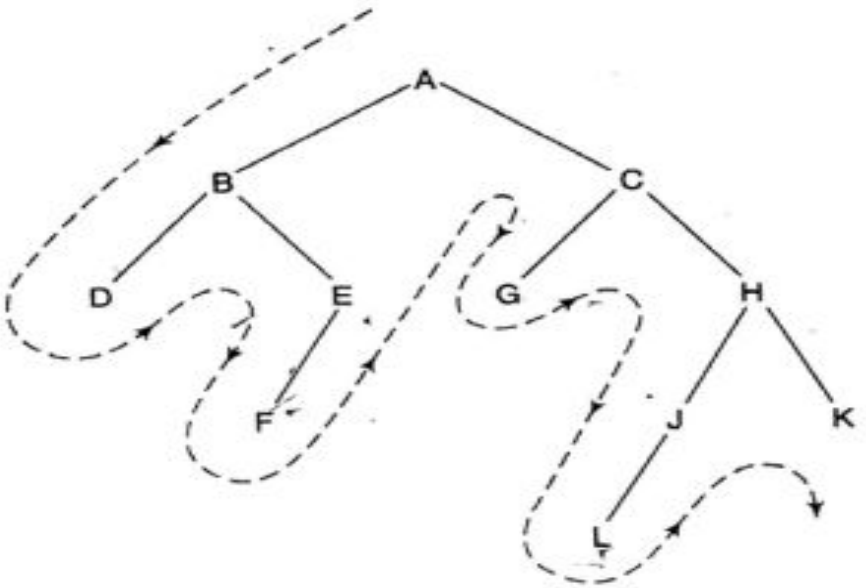
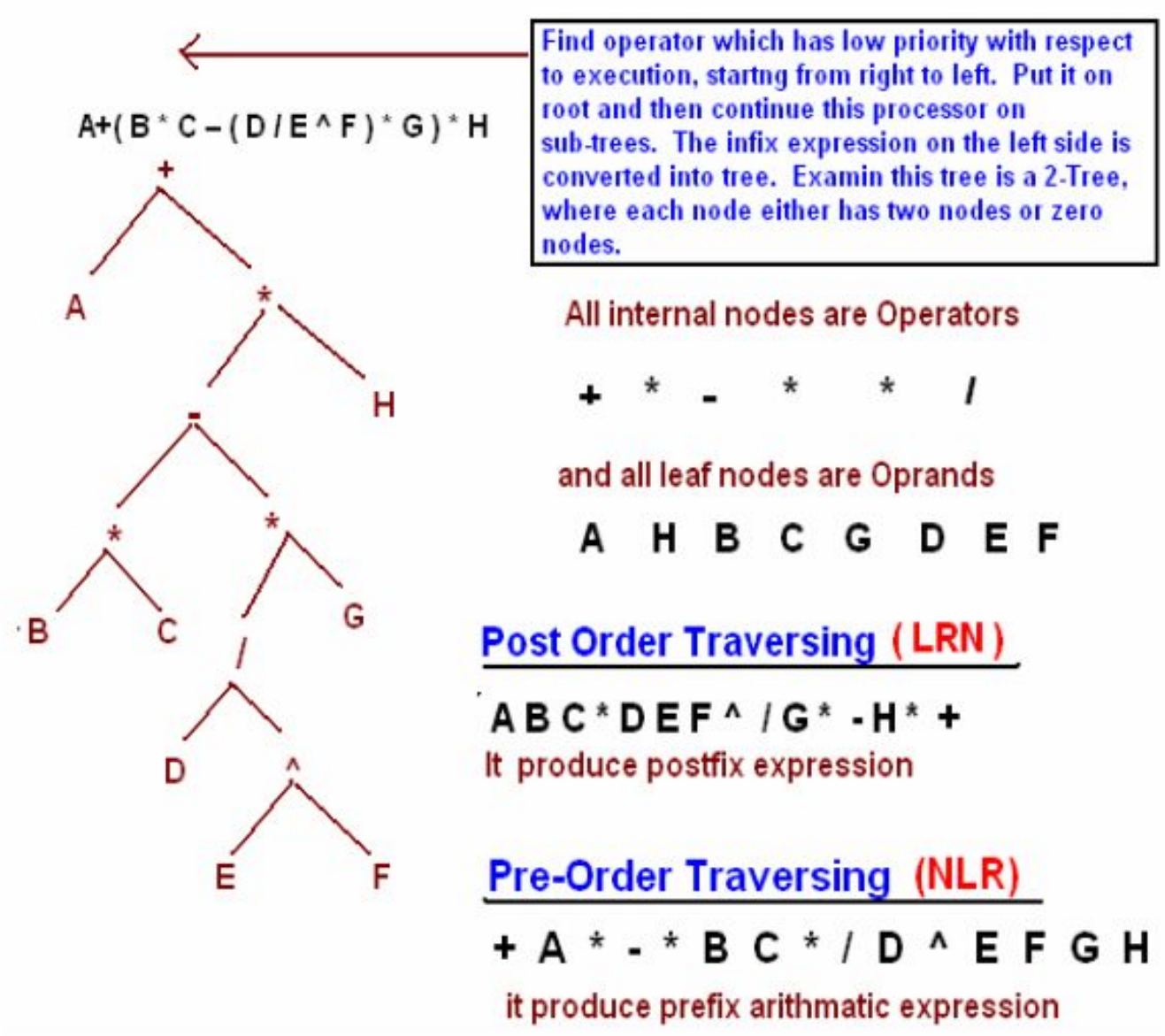


Fig. 7.12

The preorder traversal of T is ABDEFCGHJLK.

(Inorder)	D	B	F	E	A	G	C	L	J	H	K
(Postorder)	D	F	E	B	G	L	J	K	H	C	A

Preparing a Tree from an infix arithmetic expression



Formation of Binary Tree from its Traversal

- Sometimes it is required to construct a binary tree if its traversals are known. From a single travers al it is not possible to construct unique binary tree. However, if two traversals is given then corresponding tree can be drawn uniquely.
- The basic principle for formulation is as follows:

  If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub- trees of the root node can be identified. Same technique can be applied repeatedly to form sub- trees.
- It can be noted that, two traversal are essential out of which one should be **inorder traversal** and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Formation of Binary Tree from its Traversal

7.2 A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the followi sequences of nodes:

Inorder:	E	A	C	K	F	H	D	B	G
Preorder:	F	A	E	K	C	D	H	G	B

Draw the tree T.

The tree T is drawn from its root downward as follows.

(a) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(b) The left child of the node F is obtained as follows. First use the inorder of T to find the nodes in the left subtree T_1 of F. Thus T_1 consists of the nodes E, A, C and K. Then the left child of F is obtained by choosing the first node in the preorder of T_1 (which appears in the preorder of T). Thus A is the left son of F.

(c) Similarly, the right subtree T_2 of F consists of the nodes H, D, B and G, and D is the root of T_2, that is, D is the right child of F.

Repeating the above process with each new node, we finally obtain the required tree in Fig. 7.74.

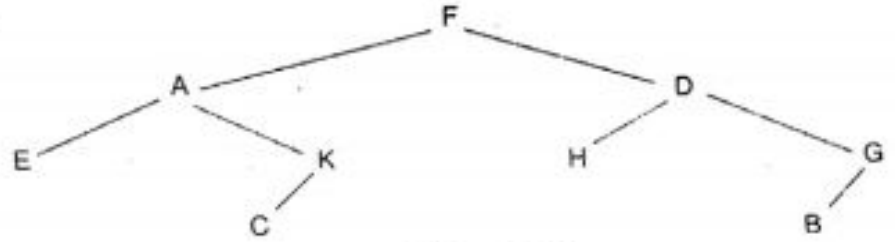


Fig. 7.74

Binary Search Tree

Suppose T is a binary tree. Then T is called a **binary search tree** or **binary sorted tree** if each node N of T has the following property:

The value of at N (node) is greater than every value in the left sub tree of N and is less than or equal to every value in the right sub tree of N.

Consider the binary tree T in Fig. 7.21. T is a binary search tree; that is, every node N in T exceeds every number in its left subtree and is less than every number in its right subtree. Suppose the 23 were replaced by 35. Then T would still be a binary search tree. On the other hand, suppose the 23 were replaced by 40. Then T would not be a binary search tree, since the 38 would not be greater than the 40 in its left subtree.

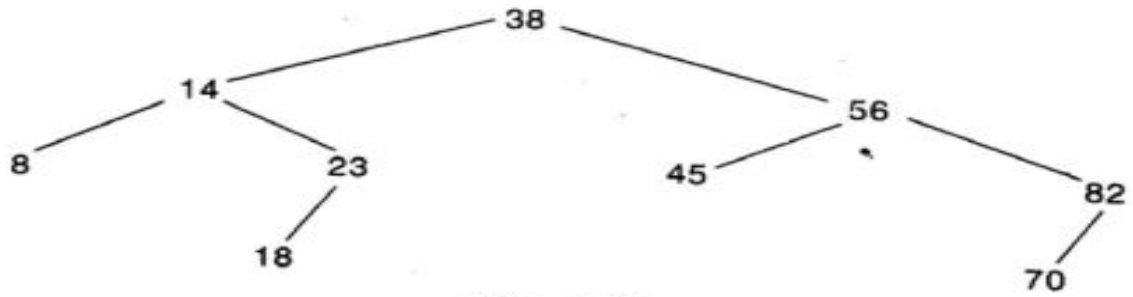


Fig. 7.21

Searching & Inserting in Binary Search Trees

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.

- **(i)** If ITEM < N, proceed to the left child of N.
- **(ii)** If ITEM > N, proceed to the right child of N.

(b) Repeat Step (a) until one of the following occurs:

- **(i)** We meet a node N such that ITEM = N. In this case the search is successful.
- **(ii)** We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

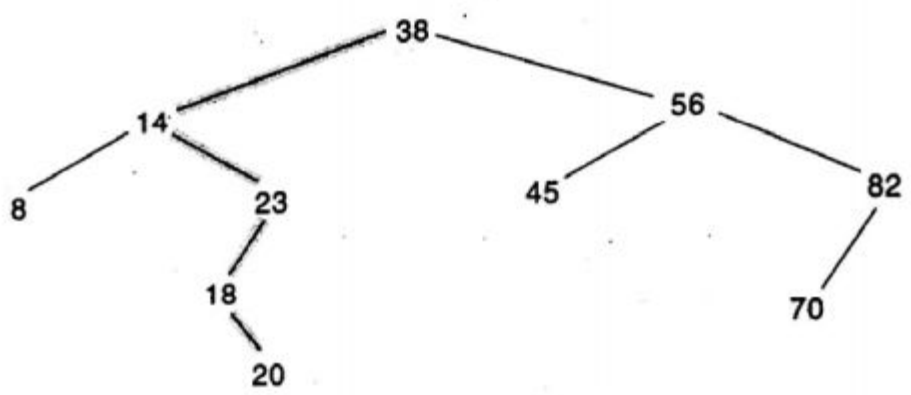


Fig. 7.22 *ITEM = 20 Inserted*

Binary Search Tree

Suppose the following six numbers are inserted in order into an empty binary search tree:

40, 60, 50, 33, 55, 11

Figure 7.23 shows the six stages of the tree. We emphasize that if the six numbers were given in a different order, then the tree might be different and we might have a different depth.

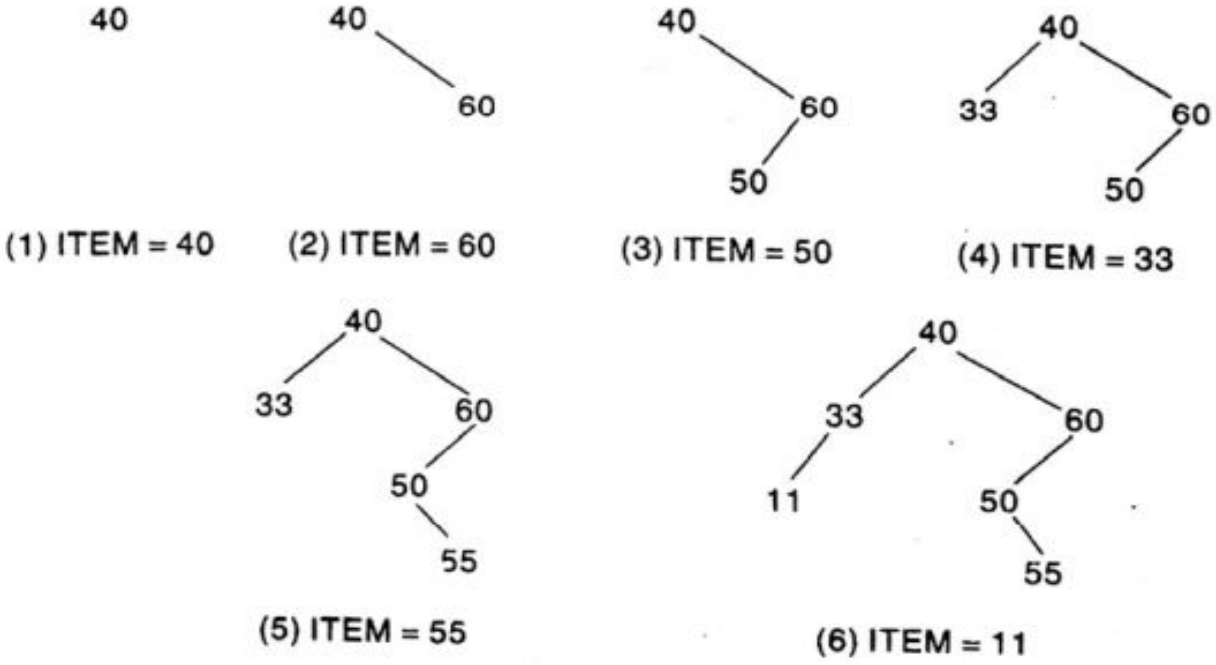


Fig. 7.23

Searching & Inserting in Binary Search Trees

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are three special cases:

(i) LOC = NULL and PAR = NULL will indicate that the tree is empty.
(ii) LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T.
(iii) LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

1. [Tree empty?]
   If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and Return.
2. [ITEM at root?]
   If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAB := NULL, and Return.
3. [Initialize pointers PTR and SAVE.]
   If ITEM < INFO[ROOT], then:
   Set PTR := LEFT[ROOT] and SAVE := ROOT.
   Else:
   Set PTR := RIGHT[ROOT] and SAVE := ROOT.
   [End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. [ITEM found?]
   If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6. If ITEM < INFO[PTR], then:
   Set SAVE := PTR and PTR := LEFT[PTR].
   Else:
   Set SAVE := PTR and PTR := RIGHT[PTR].
   [End of If structure.]
   [End of Step 4 loop.]
7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.
8. Exit.

Searching & Inserting in Binary Search Trees

INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
A binary search tree T is in memory and an ITEM of information is given. This algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T at location LOC.

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
   [Procedure 7.4.]
2. If LOC ≠ NULL, then Exit.
3. [Copy ITEM into new node in AVAIL list.]
   (a) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
   (b) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and
       INFO[NEW] := ITEM.
   (c) Set LOC := NEW, LEFT[NEW] := NULL and
       RIGHT[NEW] := NULL.
4. [Add ITEM to tree.]
   If PAR = NULL, then:
       Set ROOT := NEW.
   Else if ITEM < INFO[PAR], then:
       Set LEFT[PAR] := NEW.
   Else:
       Set RIGHT[PAR] := NEW.
   [End of If structure.]
5. Exit.

Complexity of the Searching Algorithm

Suppose we are searching for an item of information in a binary search tree T. Observe that the number of comparisons is bounded by the depth of the tree. This comes from the fact that we proceed down a single path of the tree. Accordingly, the running time of the search will be proportional to the depth of the tree.

Suppose we are given n data items, A_1 A_2, ..., A_N, and suppose the items are inserted in order into a binary search tree T. Recall that there are $n!$ permutations of the n items (Sec. 2.2). Each such permutation will give rise to a corresponding tree. It can be shown that the average depth of the $n!$ trees is approximately $c \log_2 n$, where $c = 1.4$. Accordingly, the average running time $f(n)$ to search for an item in a binary tree T with n elements is proportional to $\log_2 n$, that is, $f(n) = O(\log_2 n)$.

Deleting in a Binary Search Tree

Suppose T is a binary search tree, and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T.

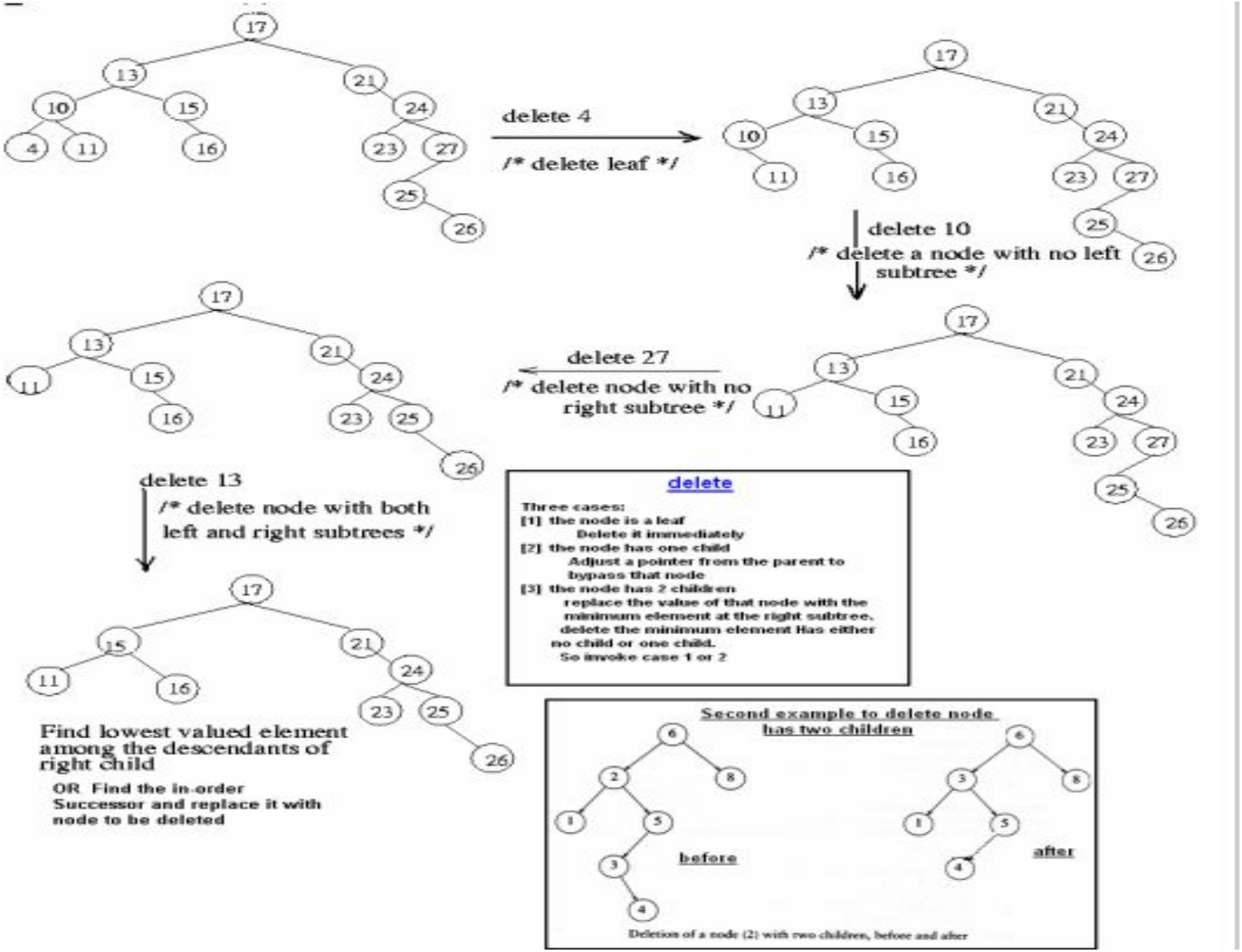
The deletion algorithm first uses Procedure 7.4 to find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases:

Case 1. N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case 2. N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

Case 3. N has two children. Let S(N) denote the inorder successor of N. (The reader can verify that S(N) does not have a left child.) Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Deleting in a Binary Search Tree



Deleting in a Binary Search Tree

Procedure 7.6: CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure deletes the node N at location LOC, where N does not have two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N, or else CHILD = NULL indicates N has no children.

```
1. [Initializes CHILD.]
   If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
       Set CHILD := NULL.
   Else if LEFT[LOC] ≠ NULL, then:
       Set CHILD := LEFT[LOC].
   Else
       Set CHILD := RIGHT[LOC].
   [End of If structure.]
2. If PAR ≠ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] := CHILD.
       Else:
           Set RIGHT[PAR] := CHILD.
       [End of If structure.]
   Else:
       Set ROOT := CHILD.
   [End of If structure.]
3. Return.
```

Deleting in a Binary Search Tree

Procedure 7.7: CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

1. [Find SUC and PARSUC.]
  (a) Set PTR := RIGHT[LOC] and SAVE := LOC.
  (b) Repeat while LEFT[PTR] ≠ NULL:
    Set SAVE := PTR and PTR := LEFT[PTR].
  [End of loop.]
  (c) Set SUC := PTR and PARSUC := SAVE.

2. [Delete inorder successor, using Procedure 7.6.]
Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).

3. [Replace node N by its inorder successor.]
  (a) If PAR ≠ NULL, then:
    If LOC = LEFT[PAR], then:
      Set LEFT[PAR] := SUC.
    Else:
      Set RIGHT[PAR] := SUC.
    [End of If structure.]
  Else:
    Set ROOT := SUC.
  [End of If structure.]
  (b) Set LEFT[SUC] := LEFT[LOC] and
  RIGHT[SUC] := RIGHT[LOC].

4. Return.

Deleting in a Binary Search Tree

Algorithm 7.8: DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

1. [Find the locations of ITEM and its parent, using Procedure 7.4.]
   Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
2. [ITEM in tree?]
   If LOC = NULL, then: Write: ITEM not in tree, and Exit.
3. [Delete node containing ITEM.]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then:
      Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   Else:
      Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LEFT[LOC] := AVAIL and AVAIL := LOC.
5. Exit.