Data Structure

Tree

Instructors:

Md Nazrul Islam Mondal &

Rizoan Toufiq

Department of Computer Science & Engineering

Rajshahi University of Engineering &

Technology Rajshahi-6204

Outline

- Binary Tree
- Representing Binary Trees in Memory
- Traversing Binary Trees
- Traversal Algorithm using Stacks
- **Header Nodes: Threads**
- **Binary Search Trees**
- **Searching and Inserting in Binary Search Trees**
- **Deleting in Binary Search Tree**
- AVL Search Trees
- Insertion in an AVL Search Tree
- Deletion in an AVL Search Tree
- m-way Search Trees
- Searching, Insertion and Deletion in an m-way Search Tree
- B Trees
- Searching, Insertion and Deletion in a B-tree

Header Nodes: Threads

Header Nodes: Threads

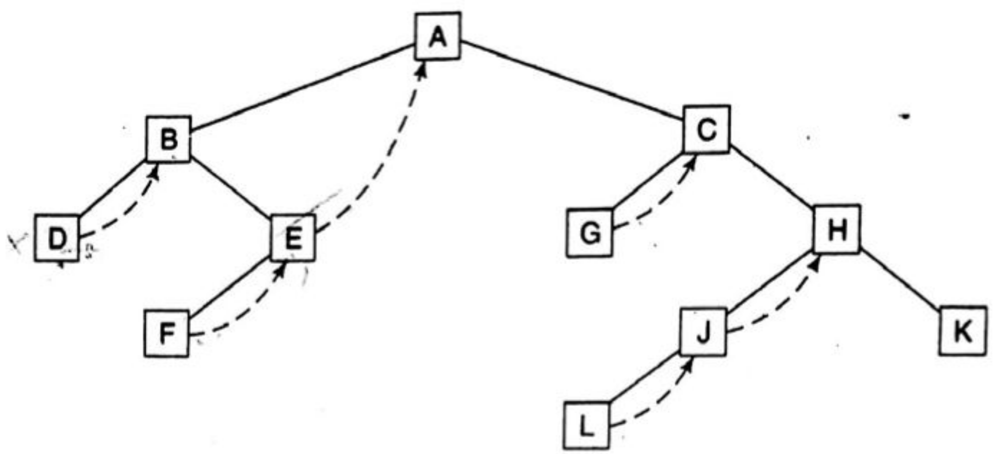
- Approximately half of the entries in the pointer fields LEFT and RIGHT will contain null element.
- We will replace certain null entries by special pointers which point to nodes higher in the tree.
- These special pointers are called **threads**.
- Binary trees with such pointers are called **threaded trees**.

Header Nodes: Threads

- There are many ways to thread a binary tree T.
- We will discuss about –
  - One-way threading
  - Two-way threading

Header Nodes: Threads

- **One-way Threading** –
  - A thread will appear in the **right field** of a node and
  - Will point to the next node in the **inorder** traversal of T



(a) One-way inorder threading

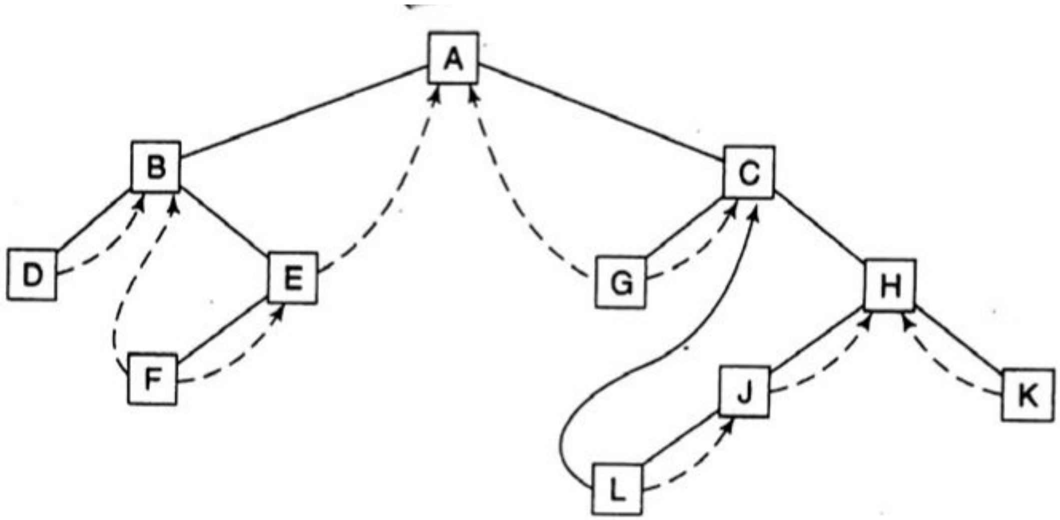
- Inorder traverse:
  - DBFEAGCLJHK
- D uses right pointer to point B
- Right pointer of F points to E
- Right Pointer of E points to A
- Similarly for G J and L

Header Nodes: Threads

- Two-way Threading –
  - A thread will appear in the right field of a node and
  - Will point to the next node in the inorder traversal of T
  - A thread also appear in the **LEFT field** of a node and
  - Will point to the preceding node in the inorder traversal of T

Header Nodes: Threads

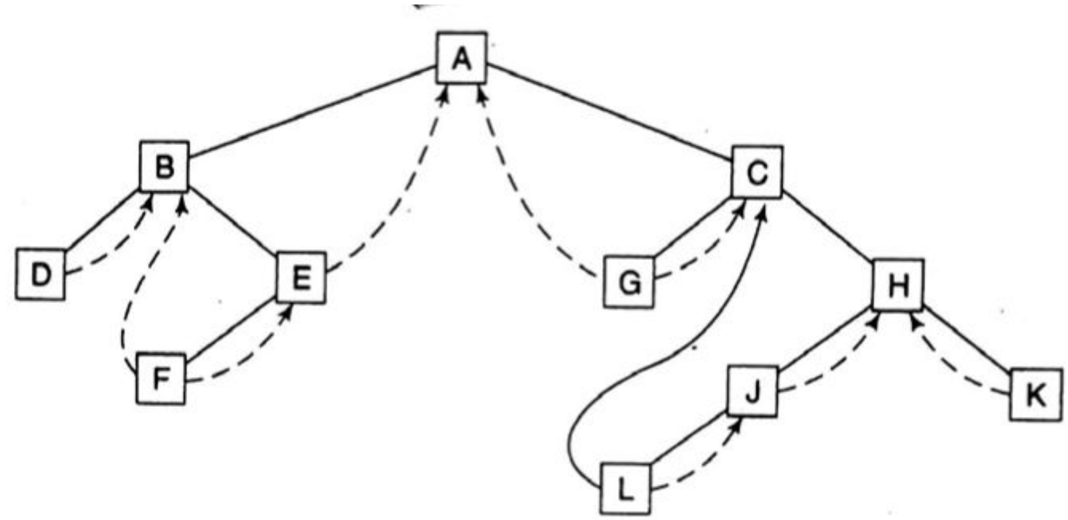
- Inorder Traversal: **D**BFEAGCLJHK
- Example:
- The left pointer of D cant use to point any node (No Preceding Node of D)
- The right pointer of D point to B node.
-



(b) Two-way inorder threading

Header Nodes: Threads

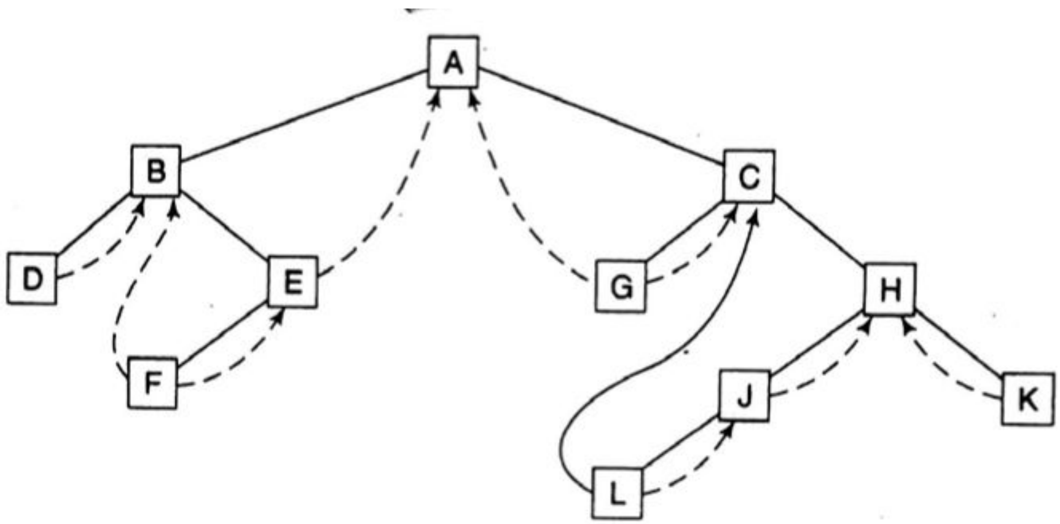
- Inorder Traversal: DBFEAGC**L**JHK
- Example:
– The **left pointer** of L point to C node (Preceding Node of D)
– The right pointer of L point to J node.
-



(b) Two-way inorder threading

Header Nodes: Threads

- Inorder Traversal: DBFEAGCLJHK
- Example:
– The **left pointer** of G point to A node (Preceding Node of D)
– The right pointer of G point to C node.
-



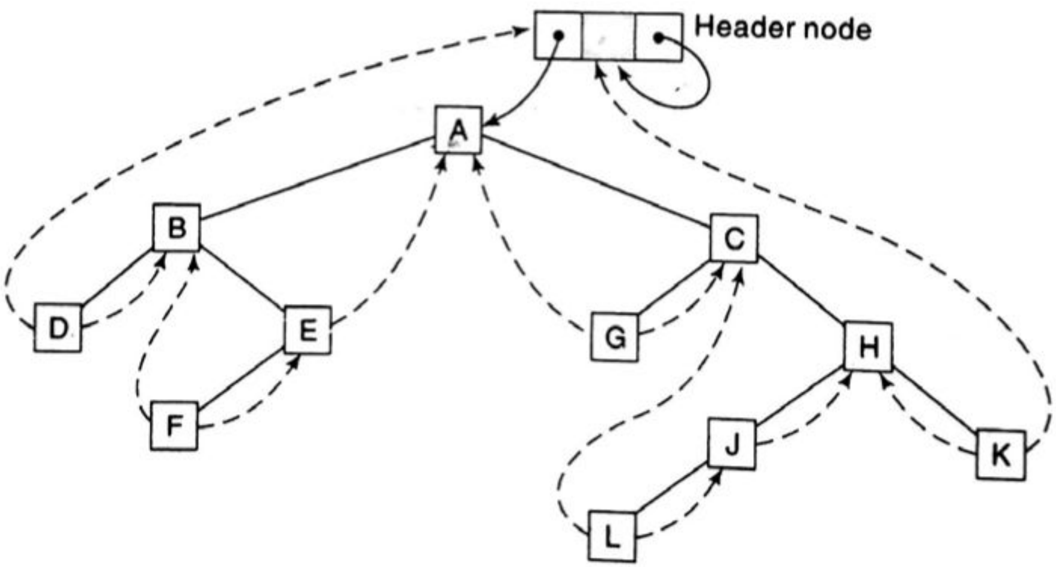
(b) Two-way inorder threading

Header Nodes: Threads

- Two-way Threading –
  - A thread will appear in the right field of a node and
  - Will point to the next node in the inorder traversal of T
  - A thread also appear in the **LEFT field** of a node and
  - Will point to the preceding node in the inorder traversal of T
  - **The left pointer of first node and the right pointer of last node** (inorder traversal of T) will contain the null value when T does not have a header node,
  - But **The left pointer of first node and the right pointer of last node** (inorder traversal of T) will pointer to the header node when T does have a header node.

Header Nodes: Threads

- The left pointer of first node D is point to header node
- The right pointer of last node K is point to header node.



(c) Two-way threading with header node

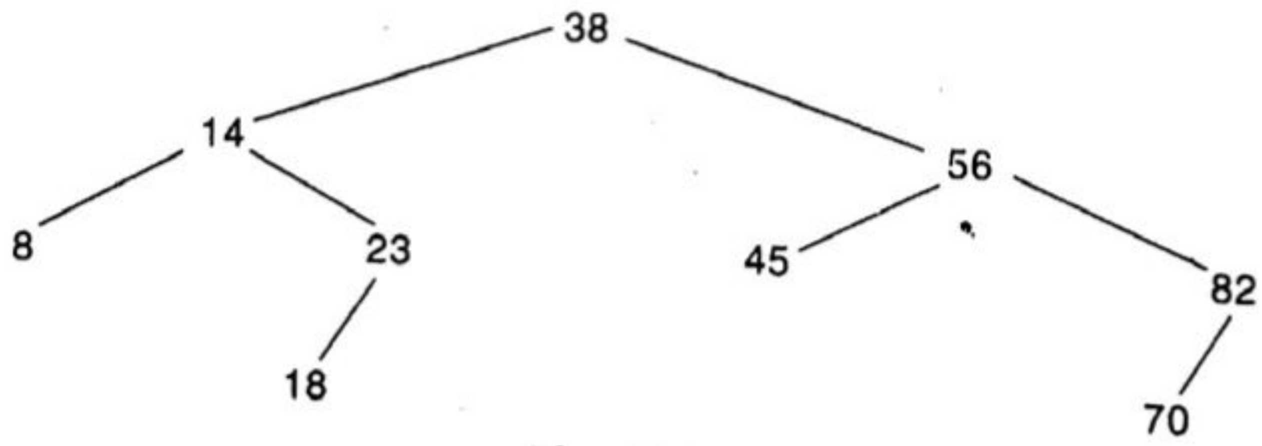
Binary Search Trees

Binary Search Trees

- A tree T is called a binary search tree (or binary sorted tree) if each node N of T has the following property:
  - The value at N is **greater** than every value in the **left** subtree of N
  - The value at N is **less** than or **equal** to every value in the right subtree of N

- **Traverse inorder** to find a sorted list of the elements of T.

Binary Search Trees



A binary search Tree

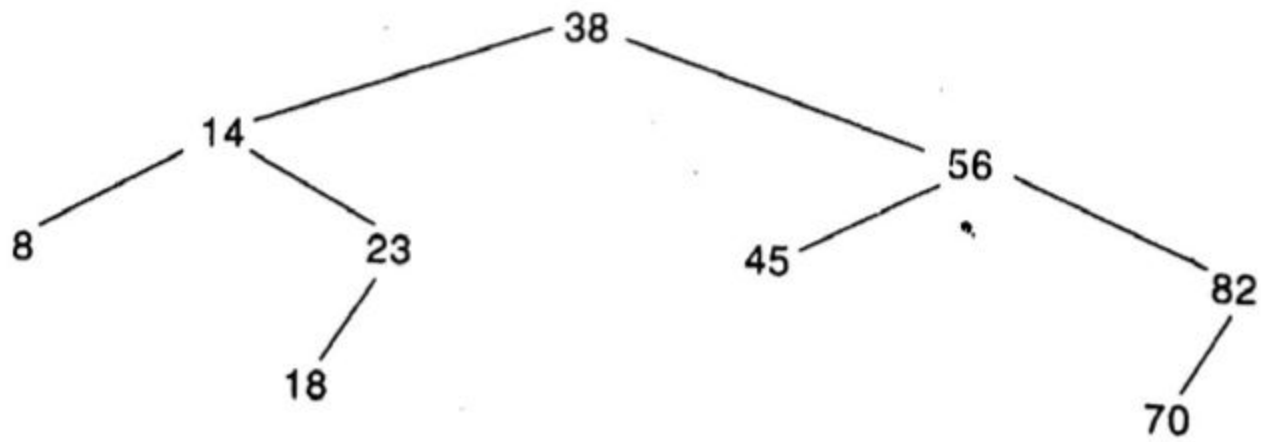
- **Inorder Traversal**: 8 14 18 23 38 45 56 70 82

Searching and Inserting in Binary Search Trees

Searching and Inserting in Binary Search Trees

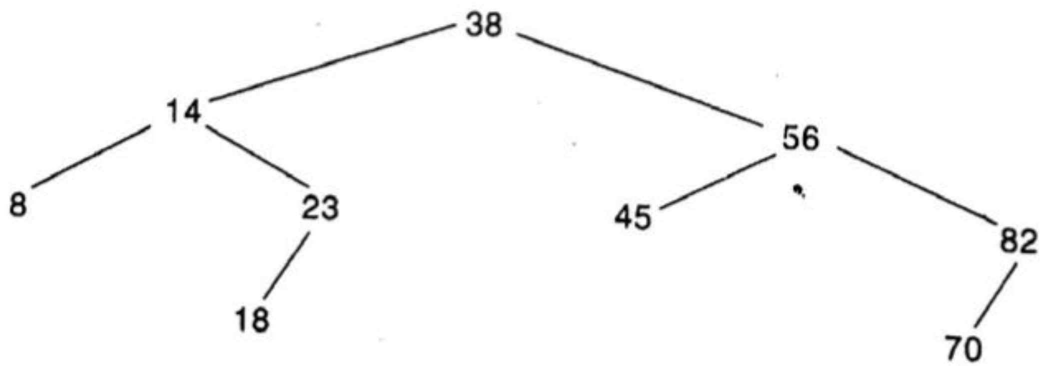
- **Objective**:
  - An ITEM of information is given.
  - Find the location of ITEM in the binary search tree T
  - If ITEM is not found, insert ITEM as a new node in its appropriate place in the tree.

Searching and Inserting in Binary Search Trees



- Given ITEM = 20.
- LOC Location of ITEM
- PAR Parent node of the ITEM

Searching and Inserting in Binary Search Trees



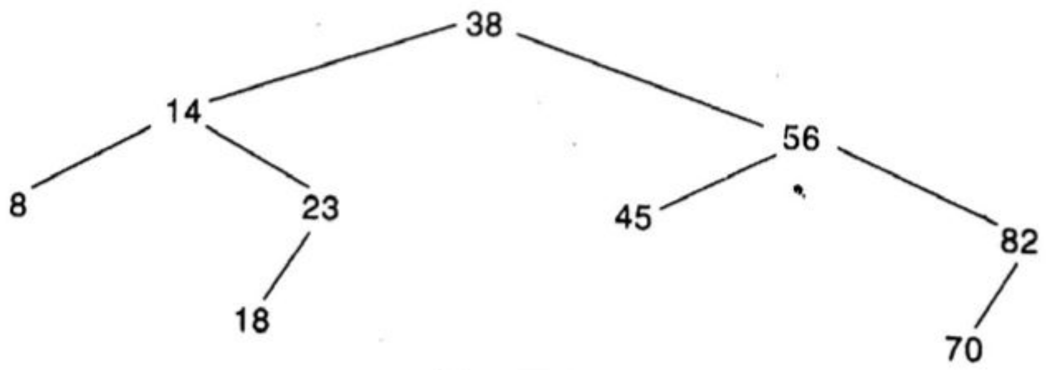
Condition -1 :

- If ROOT ← NULL
  - LOC ← NULL
  - PAR ← NULL

Condition -2 :

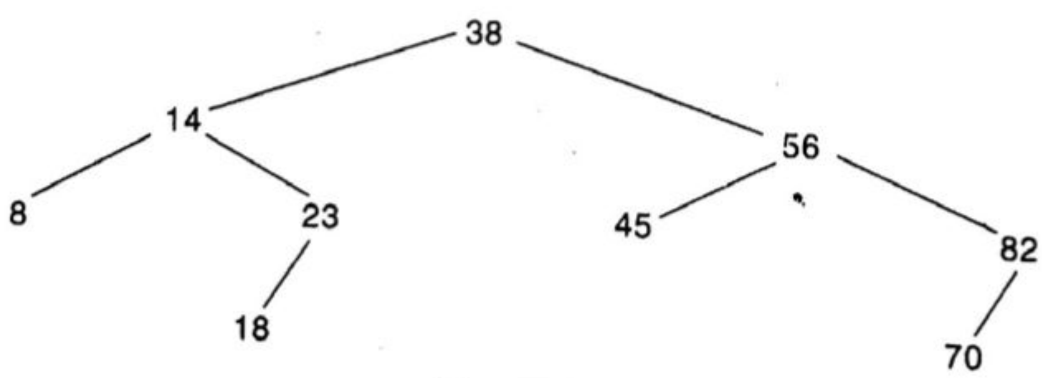
- If INFO[ROOT]=ITEM
  - LOC ← ROOT
  - PAR ← NULL

Searching and Inserting in Binary Search Trees



- **Traversing Pointer:**
  - PTR point current node
  - SAVE Parent of current node

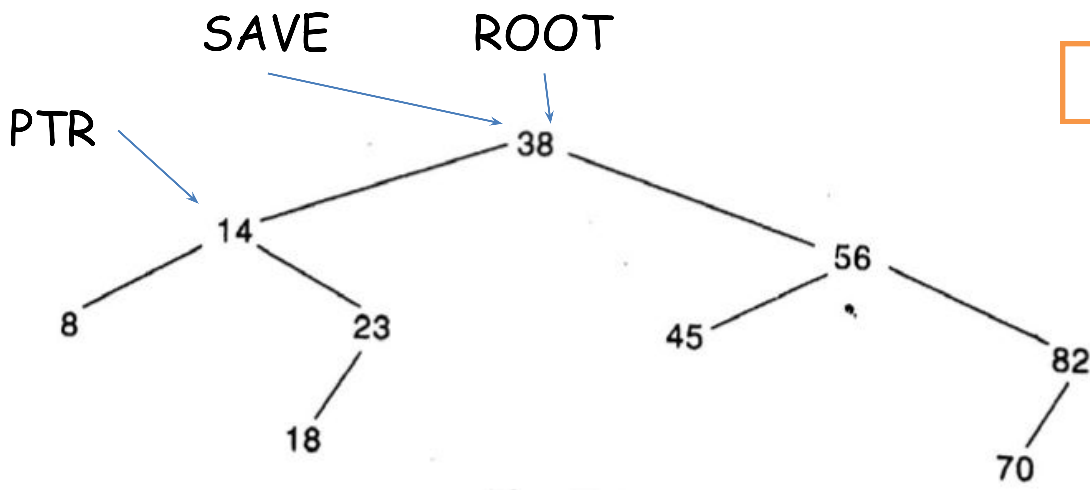
Searching and Inserting in Binary Search Trees



- ITEM←20

- ***Condition – 3***
  - If ITEM<INFO[ROOT]
    - [search Left subtree] PTR←LEFT[ROOT], SAVE←ROOT
  - Else
    - [search Right subtree] PTR←RIGHT[ROOT], SAVE←ROOT

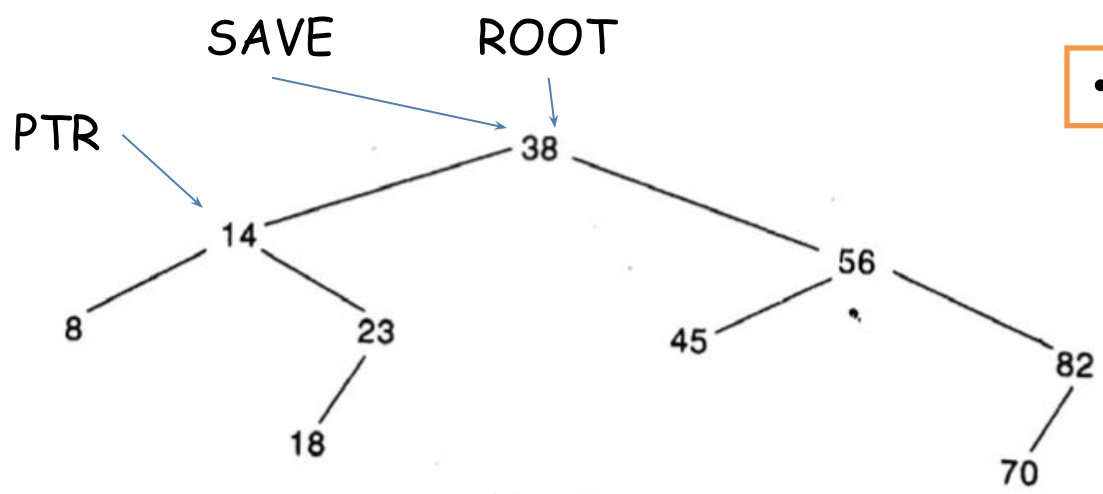
Searching and Inserting in Binary Search Trees



- ITEM20

- **Condition – 3**
  - ITEM = 20 <ROOT so PTR LEFT[ROOT], SAVE ROOT

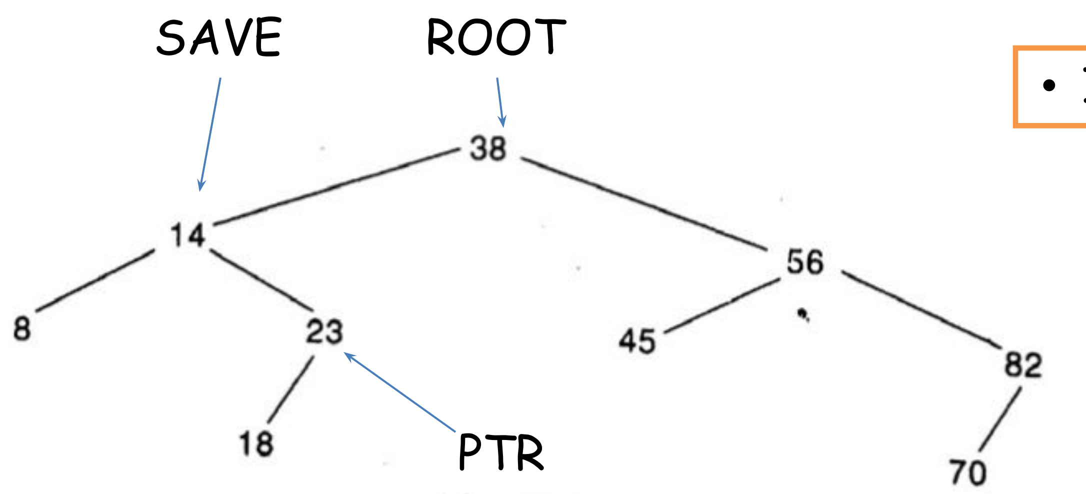
Searching and Inserting in Binary Search Trees



- ITEM←20

- **Loop -1: Until PTR ≠NULL**
  - IF INFO[PTR] = ITEM, LOC←PTR, PAR←SAVE, **BREAK**
  - IF ITEM<INFO[PTR], [Search Left subtree], PTR←LEFT[PTR]
  - Else [Search right subtree], PTR←RIGHT[PTR]
  - SAVE←PTR

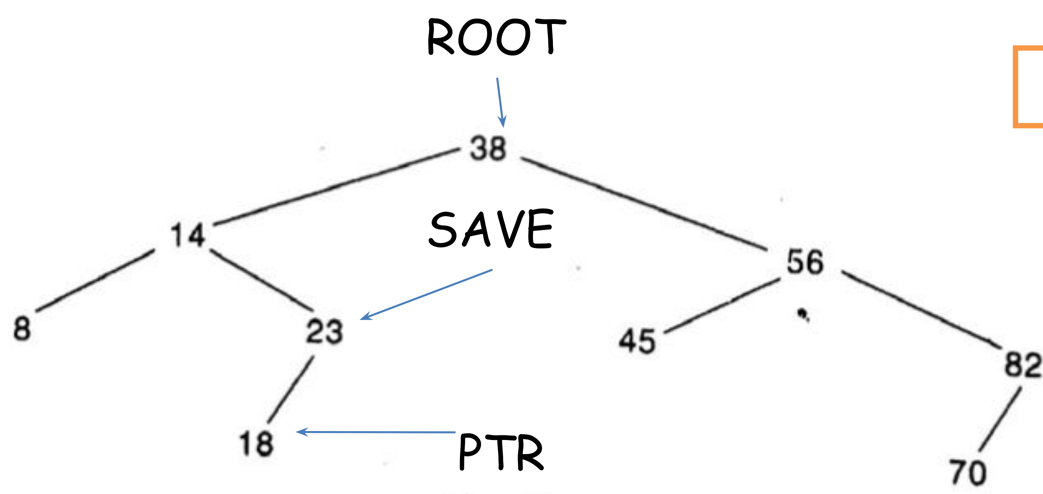
Searching and Inserting in Binary Search Trees



- ITEM←20

- **Loop -1: Until PTR ≠NULL**
  - ITEM>INFO[PTR], PTR←RIGHT[PTR]
  - SAVE←PTR

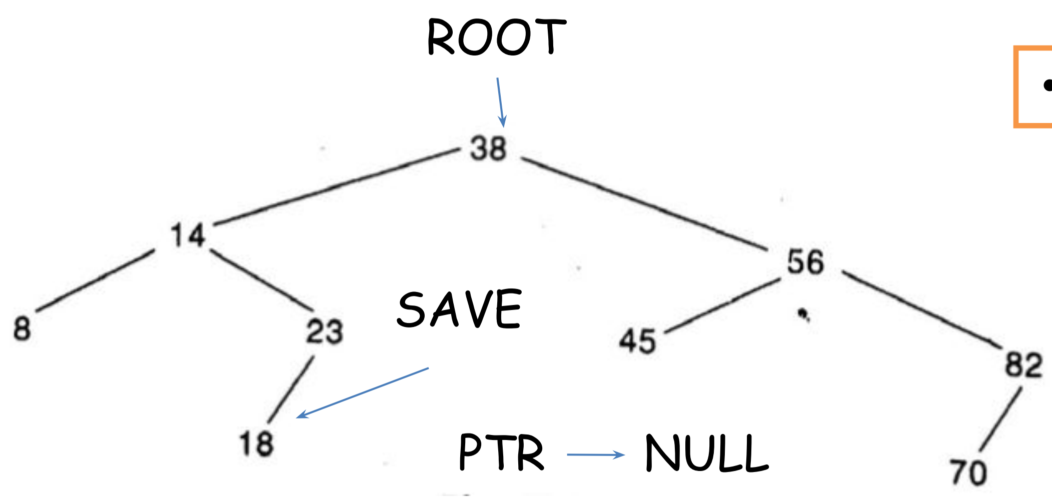
Searching and Inserting in Binary Search Trees



- ITEM←20

- **Loop -1: Until PTR ≠NULL**
    - ITEM<INFO[PTR], PTR←LEFT[PTR]
    - SAVE←PTR

Searching and Inserting in Binary Search Trees

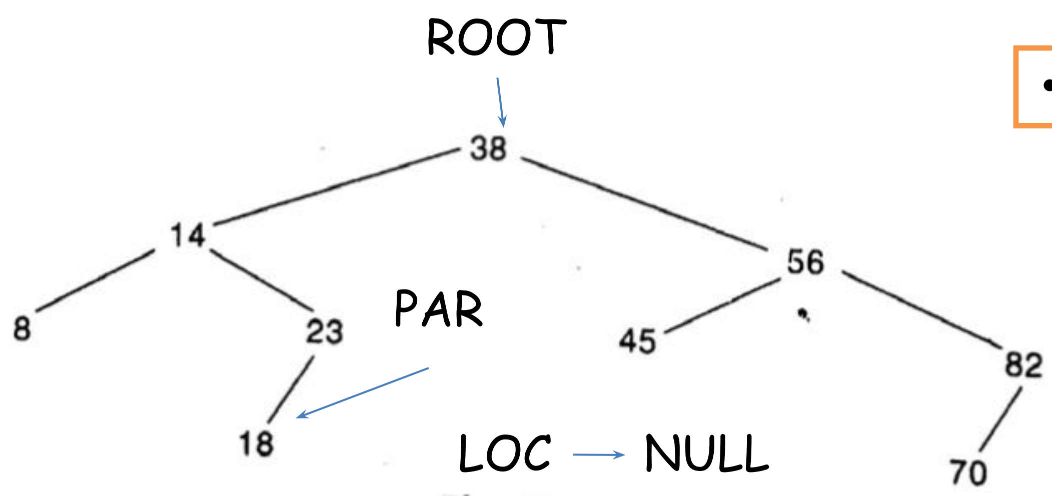


- ITEM←20

- **Loop -1: Until PTR ≠NULL**
  - ITEM>INFO[PTR], PTR←RIGHT[PTR]
  - SAVE←PTR
  - **PTR=NULL, BREAK**

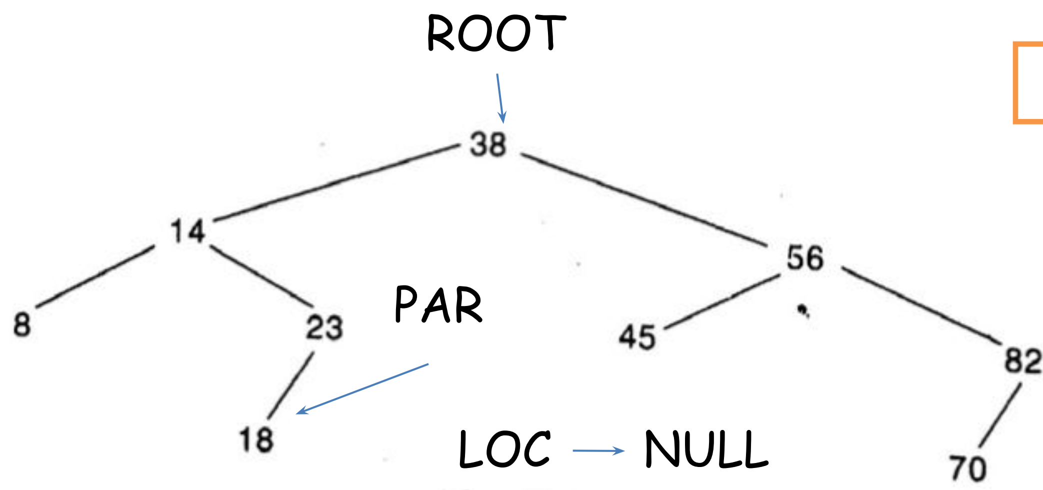
Searching and Inserting in Binary Search Trees

- ITEM←20



- Since [Search Unsuccessful] PTR = NULL,
    - LOC←NULL, PAR←SAVE

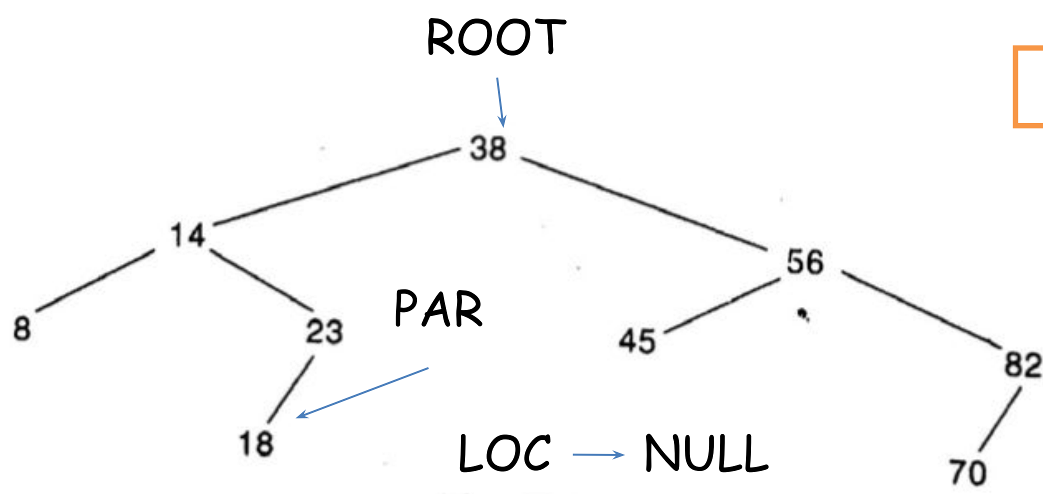
Searching and Inserting in Binary Search Trees



- ITEM□20

- **Insertion Step:**
  - **If [Successful, No insertion] LOC ≠ NULL, Exit**

Searching and Inserting in Binary Search Trees



- ITEM←20

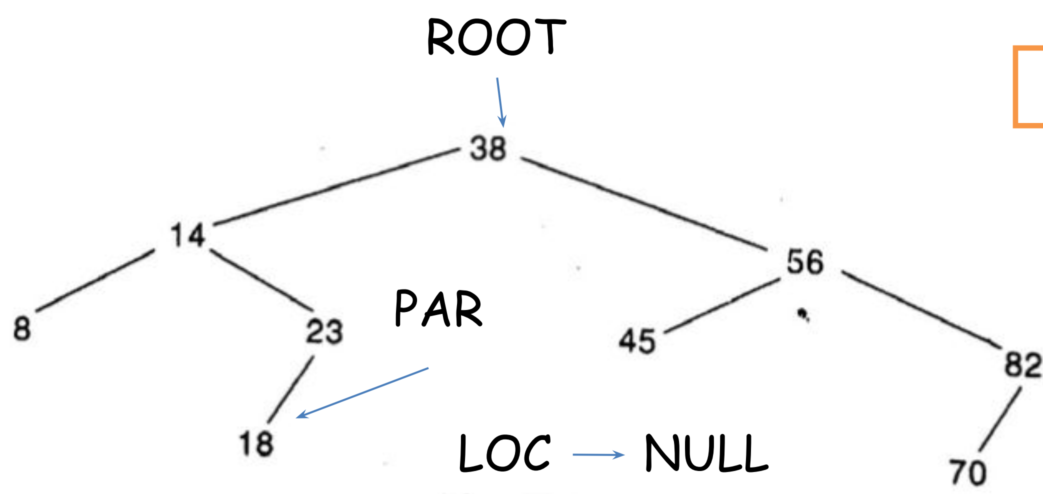
- **Insertion Step:**
  - If AVAIL = NULL, Print Overflow
  - NEW← AVAIL
  - AVAIL← LEFT[AVAIL]
  - INFO[NEW]←ITEM, LEFT[NEW]←NULL, RIGHT[NEW]←NULL

New

\0	ITEM	\0

Searching and Inserting in Binary Search Trees

- ITEM←20

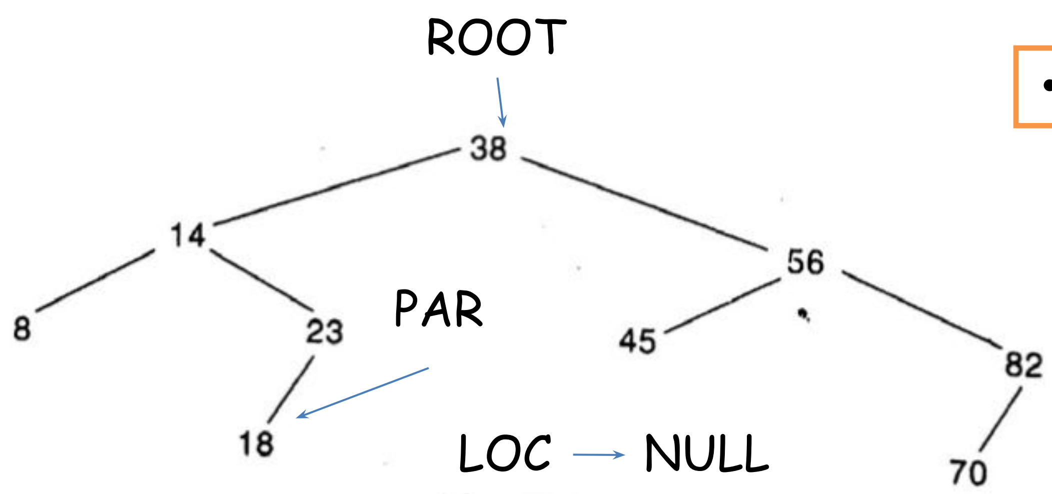


New

\0	ITEM	\0

- **Insertion Step:**
  - If PAR=NULL
    - [Empty Tree] ROOT←NEW

Searching and Inserting in Binary Search Trees



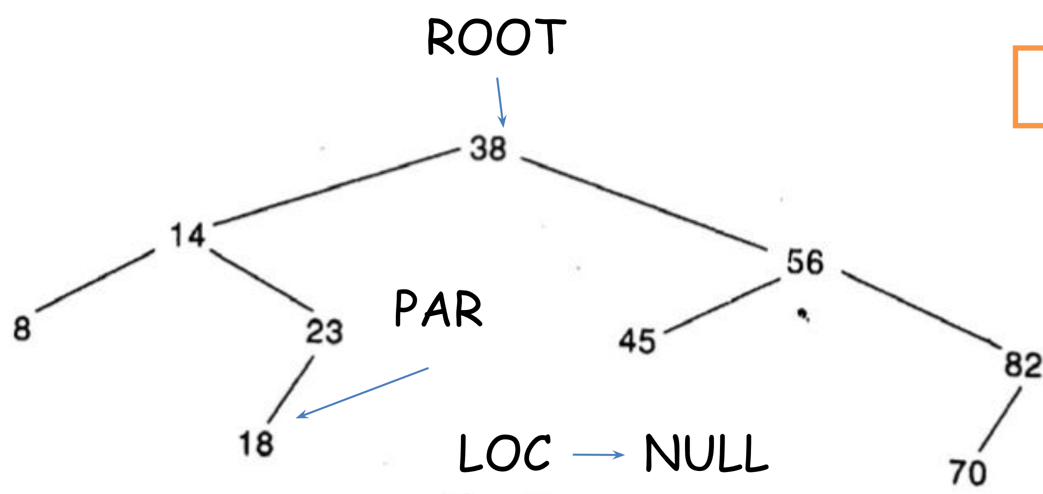
- ITEM←20

New

\0	ITEM	\0

- **Insertion Step:**
  - If ITEM<INFO[PAR]
    - [Left Child of PAR] LEFT[PAR]←NEW

Searching and Inserting in Binary Search Trees



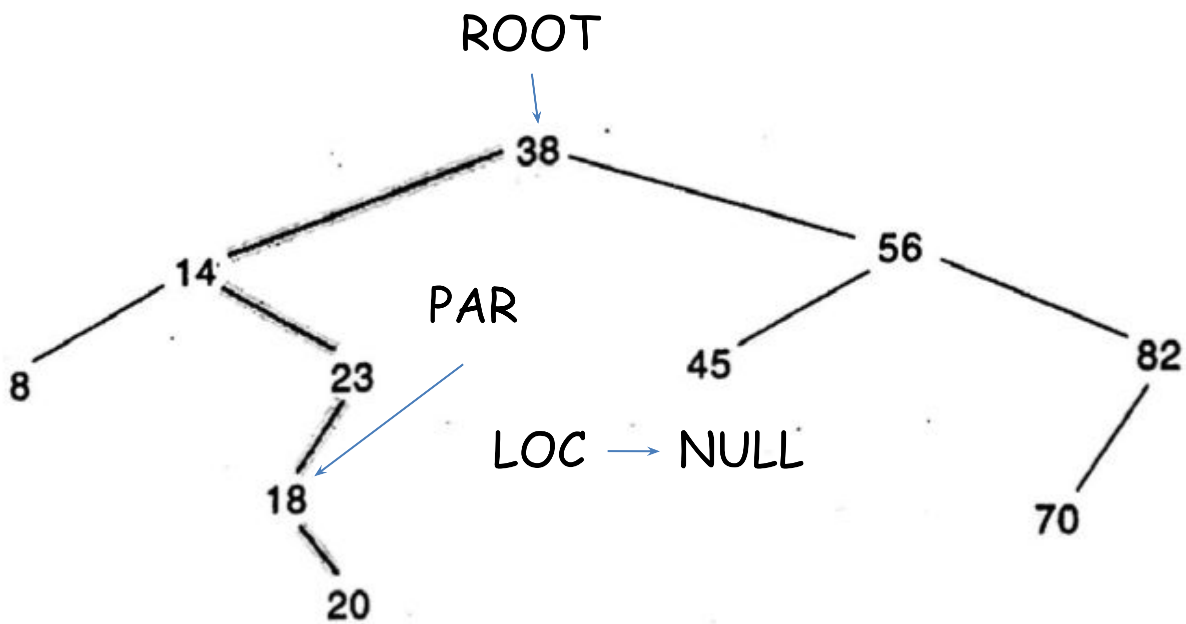
- ITEM←20

New

\0	ITEM	\0

- **Insertion Step:**
  - If ITEM≥INFO[PAR]
    - [Right Child of PAR] RIGHT[PAR]←NEW
  - 20>18, So add new node as right node of PAR

Searching and Inserting in Binary Search Trees



Searching and Inserting in Binary Search Trees

Procedure 7.4: 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.]

Searching and Inserting in Binary Search Trees

Data 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.

the right child according to whether

Searching and Inserting in Binary Search Trees

Algorithm 7.5: 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.

Searching and Inserting in Binary Search Trees

- Complexity of searching: $O(\log_2 n)$

Searching and Inserting in Binary Search Trees

- Application:
  - Want to find and delete all duplicates in the collection
  - Suppose we have

    14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

  - Observe that the first four numbers (14, 10, 17, 12) are not deleted (check: 0+1+2+3).
  - 10 is deleted (check: 2 times)
  - 11 is not deleted (Check: 4)
  - 20 is not deleted (Check: 5) and To be continue this process
  - Final result: 14 10 17 12 11 20 18 25 8 22 23
  - **Complexity**: $O(n^2)$

Searching and Inserting in Binary Search Trees

- Application:
  - We can solve this problem using binary search tree effeciently.
  - **Complexity**: $O(n\log_2 n)$

Searching and Inserting in Binary Search Trees

Consider again the following list of 15 numbers:

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

Applying Algorithm B to this list of numbers, we obtain the tree in Fig. 7.24. The exact number of comparisons is

$$0 + 1 + 1 + 2 + 2 + 3 + 2 + 3 + 3 + 3 + 3 + 2 + 4 + 4 + 5 = 38$$

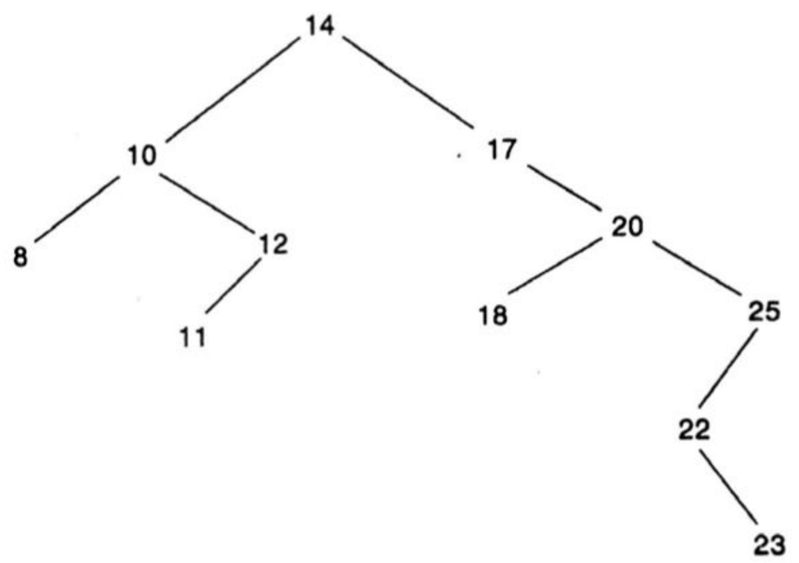


Fig. 7.24

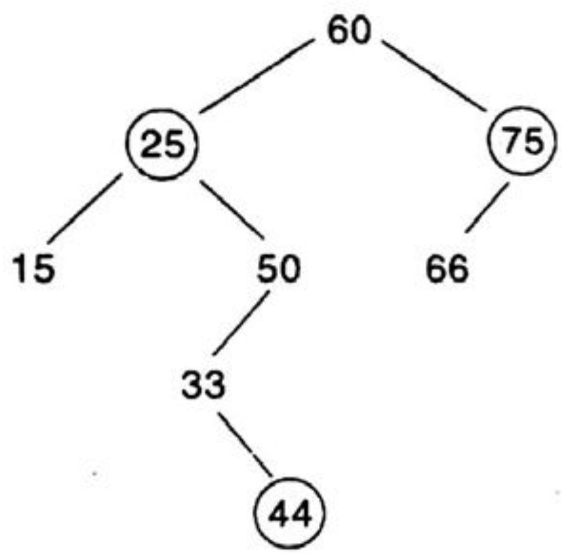
Deleting in Binary Search Tree

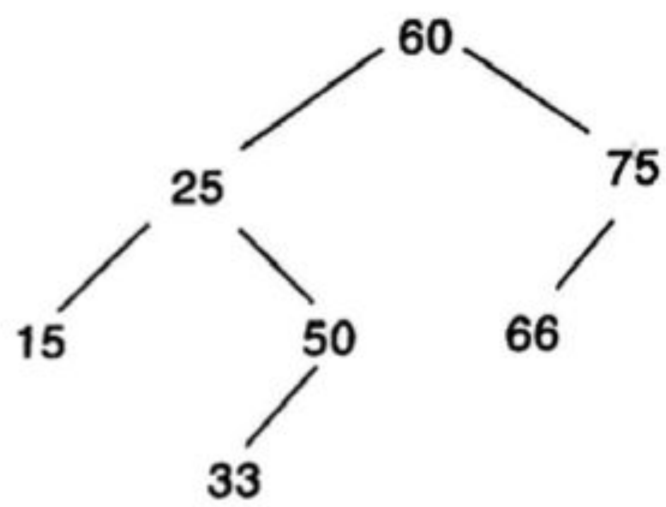
Deleting in Binary Search Tree

- Find the location of node N which contains ITEM (Discuss Previous slid)
- The location of the parent node P(N)
- The way N is deleted from the tree depends on the number of children of node N.
- These are three cases:
  a) 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

Deleting in Binary Search Tree

- These are three cases:
  a) 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

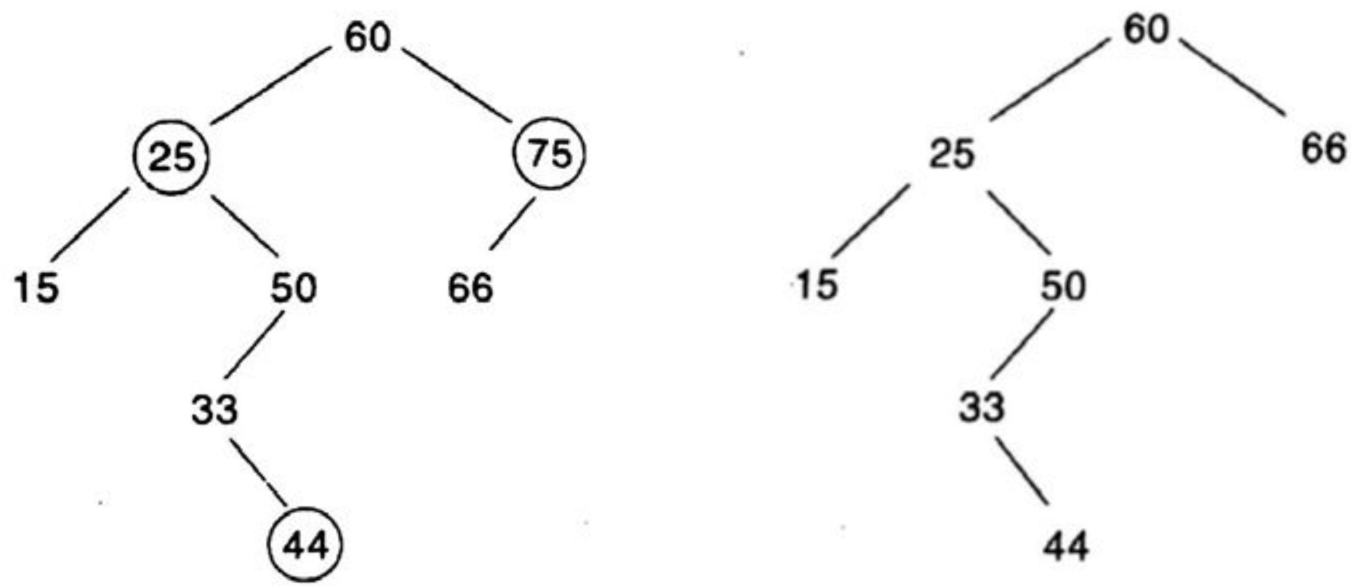




Deleting in Binary Search Tree

- These are three cases:
  a) 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
  b) N has exactly one child.
     - The N is deleted from T by simply replacing the location N in P(N) by the location of the only child of N.

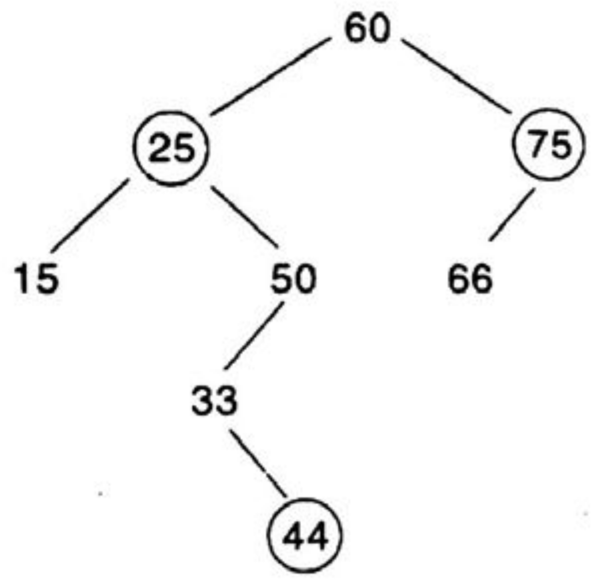
Deleting in Binary Search Tree



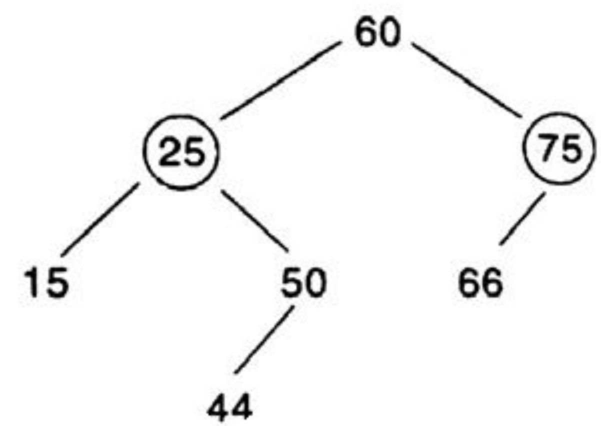
Deleting in Binary Search Tree

- These are three cases:
  a) 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
  b) N has exactly one child. The N is deleted from T by simply replacing the location N in P(N) by the location of the only child of N.
  c) N has two children.
     - Let S(N) denote the inorder successor of N
     - Delete S(N) using Case (a) or Case (b)
     - Replacing node N in T by the node S(N),

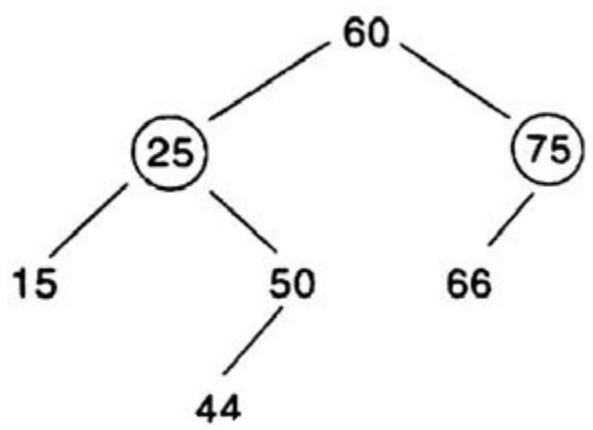
Deleting in Binary Search Tree



- 15 and 50 are two children of 25
- **Inorder Successor** of 25: 33
  - 15 **25** 33 44 50 60 66 75
- Delete 33 as Case (b)

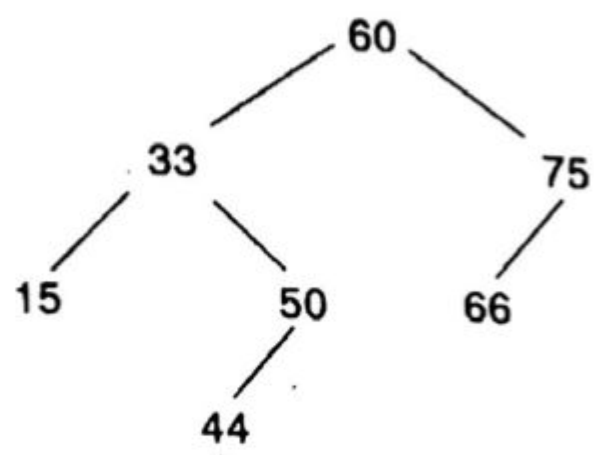


Deleting in Binary Search Tree



- 15 and 50 are two children of 25
- **Inorder Successor** of 25: 33
  - 15 **25** 33 44 50 60 66 75

- Delete 33 as Case (b)
- Replacing 25 by 33



Deleting in 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.

CASE A and CASE B

Deleting in 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).

CASE C

Deleting in Binary Search Tree

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.

CASE C

Deleting in 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.

Any Query?