Binary Trees

Dr. N. L. Gavankar

Binary Tree

1. Maximum Height

   $H_{max} = N$

2. Minimum Height

   $H_{min} = \lfloor \log_2 N \rfloor + 1$

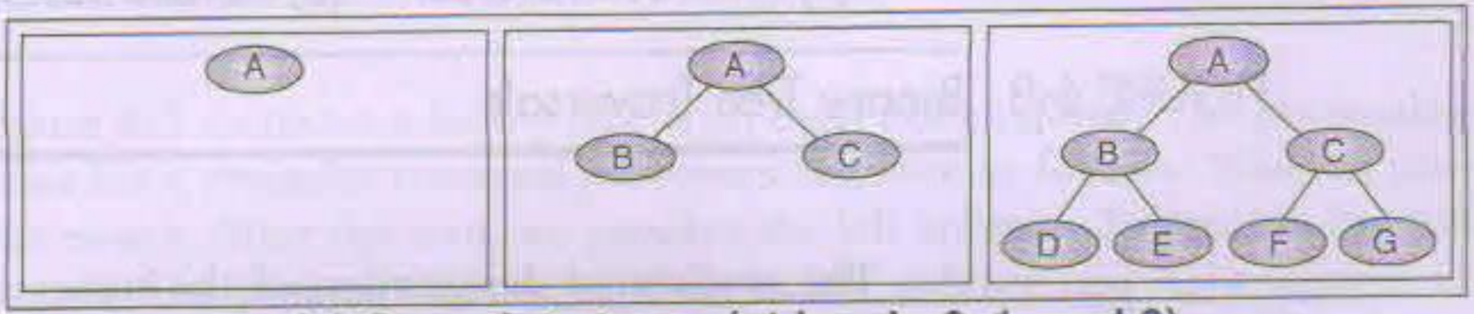
3. Minimum number of nodes of a specific height H

   $N_{min} = H$

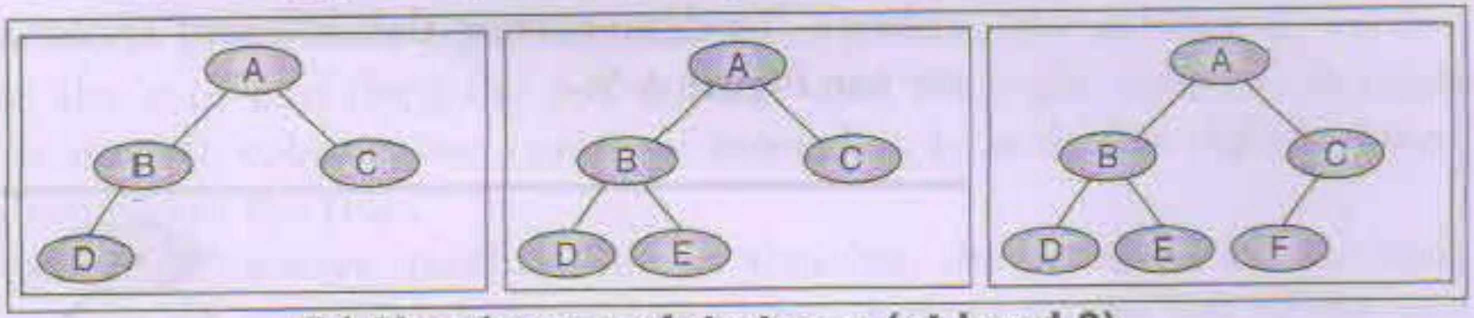
4. Maximum number of nodes of a specific height H

   $N_{max} = 2^H - 1$

Complete and Nearly Complete Binary Tree



(a) Complete trees (at levels 0, 1, and 2)

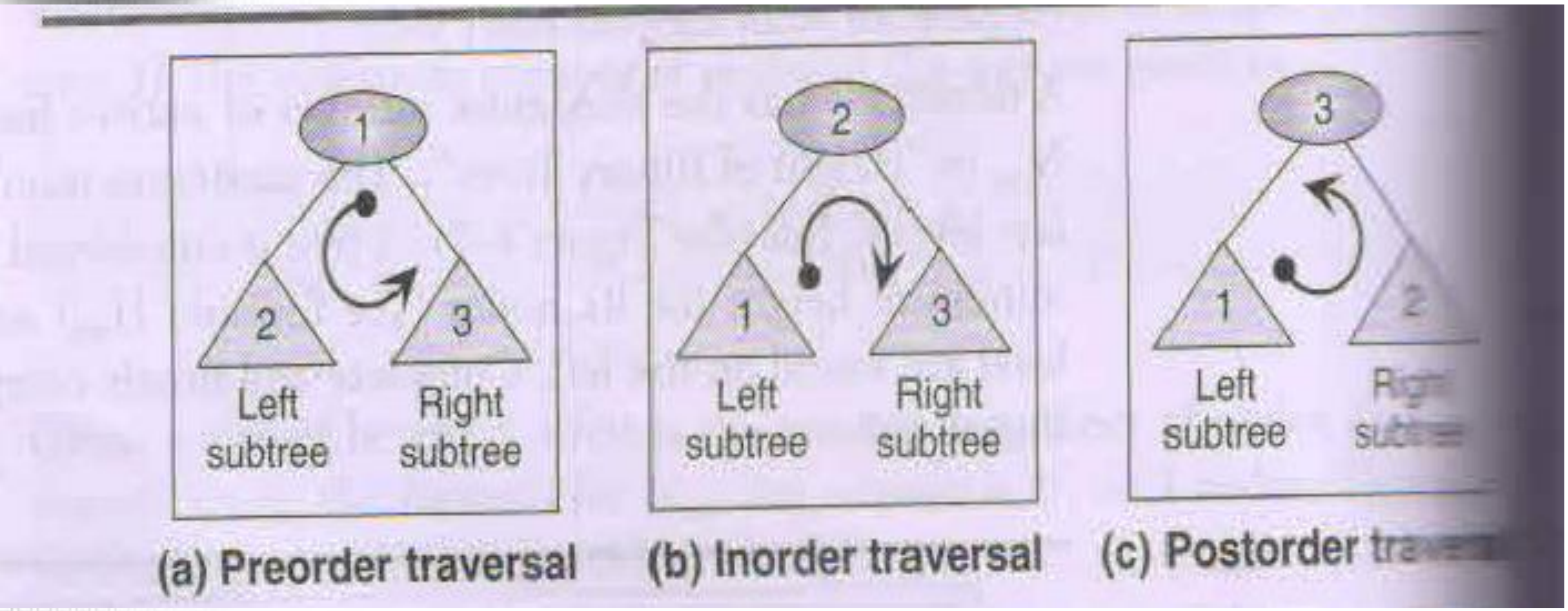


(b) Nearly complete trees (at level 2)

Binary Tree Traversal

- Depth-First-Search
- Breadth

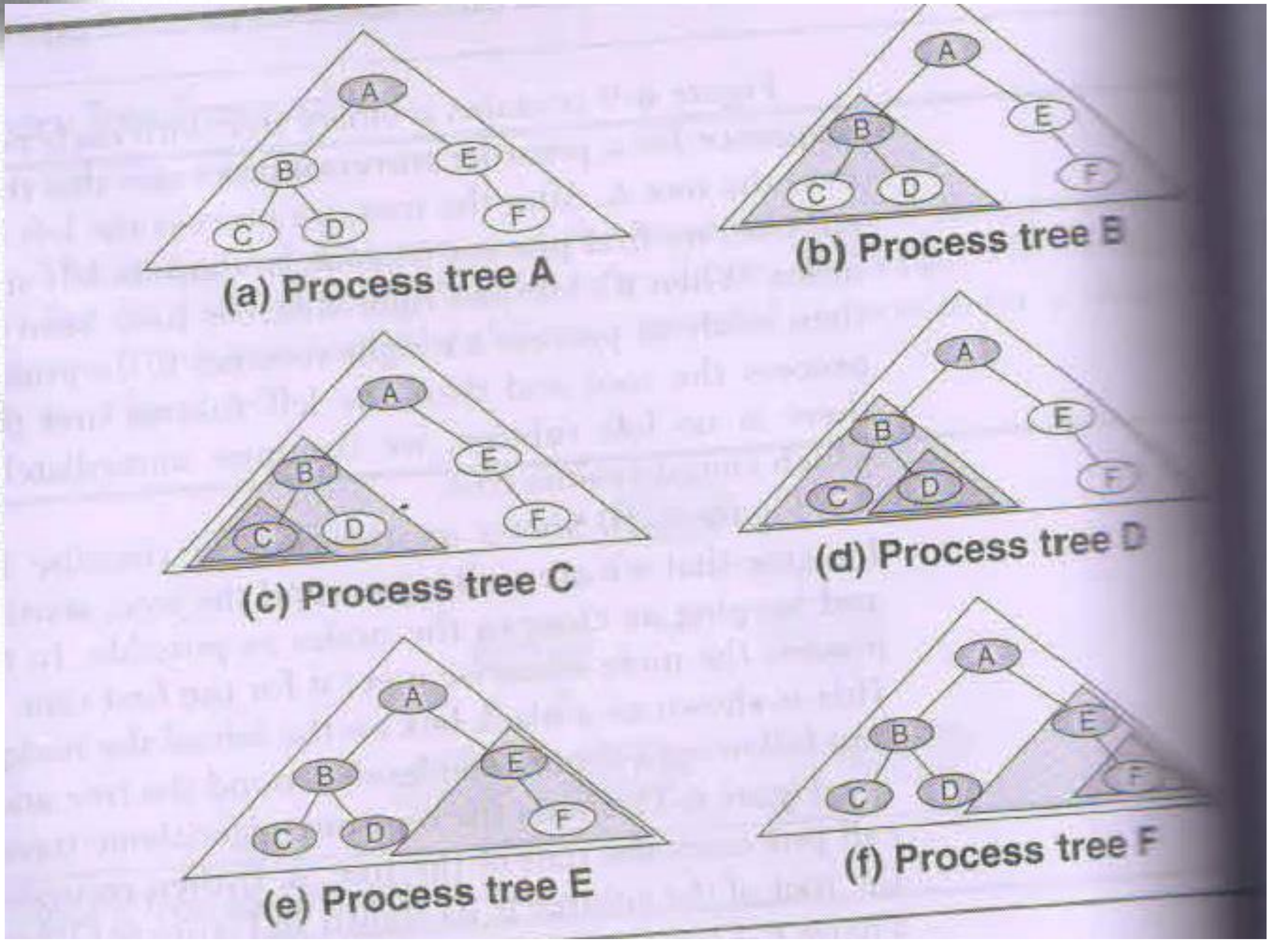
Depth-First-Search



Preorder Traversal

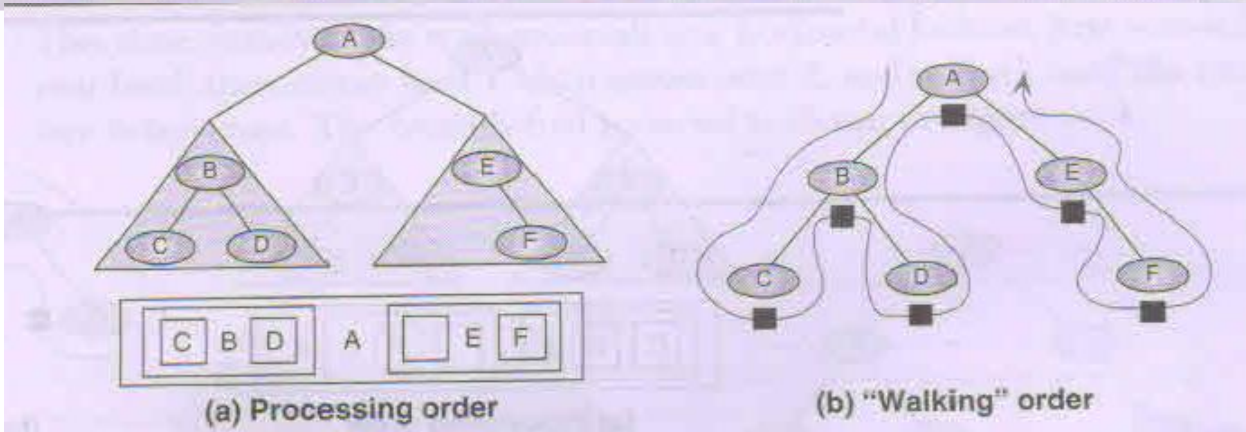
```
Algorithm preOrder (root)
Traverse a binary tree in node-left-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1 if (root is not null)
     1  process (root)
     2  preOrder (leftSubtree)
     3  preOrder (rightSubtree)
2 end if
end preOrder
```

Preorder Traversal



Inorder Traversal

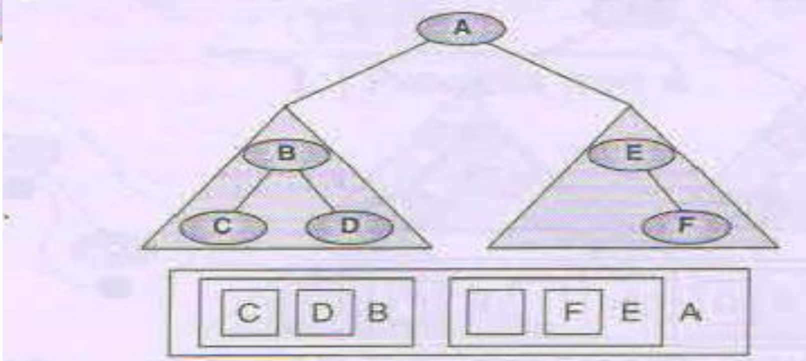
```
Algorithm inOrder (root)
Traverse a binary tree in left-node-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  inOrder (leftSubTree)
   2  process (root)
   3  inOrder (rightSubTree)
 2 end if
end inOrder
```



Inorder Traversal—C B D A E F

Inorder Traversal

```
Algorithm postOrder (root)
Traverse a binary tree in left-right-node sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  postOrder (left subtree)
   2  postOrder (right subtree)
   3  process (root)
 2 end if
end postOrder
```

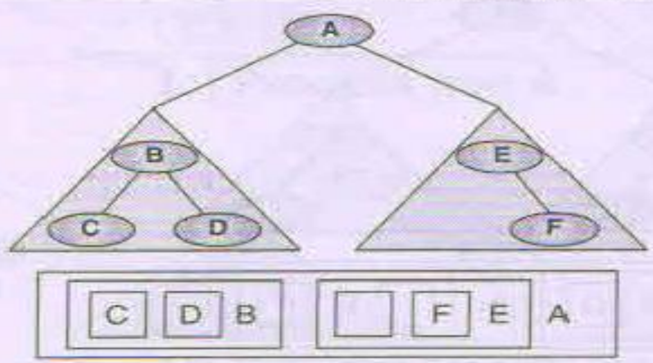


(a) Processing order

Postorder Traversal—C D B F E A

Postorder Traversal

```
Algorithm postOrder (root)
Traverse a binary tree in left-right-node sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  postOrder (left subtree)
   2  postOrder (right subtree)
   3  process (root)
 2 end if
end postOrder
```



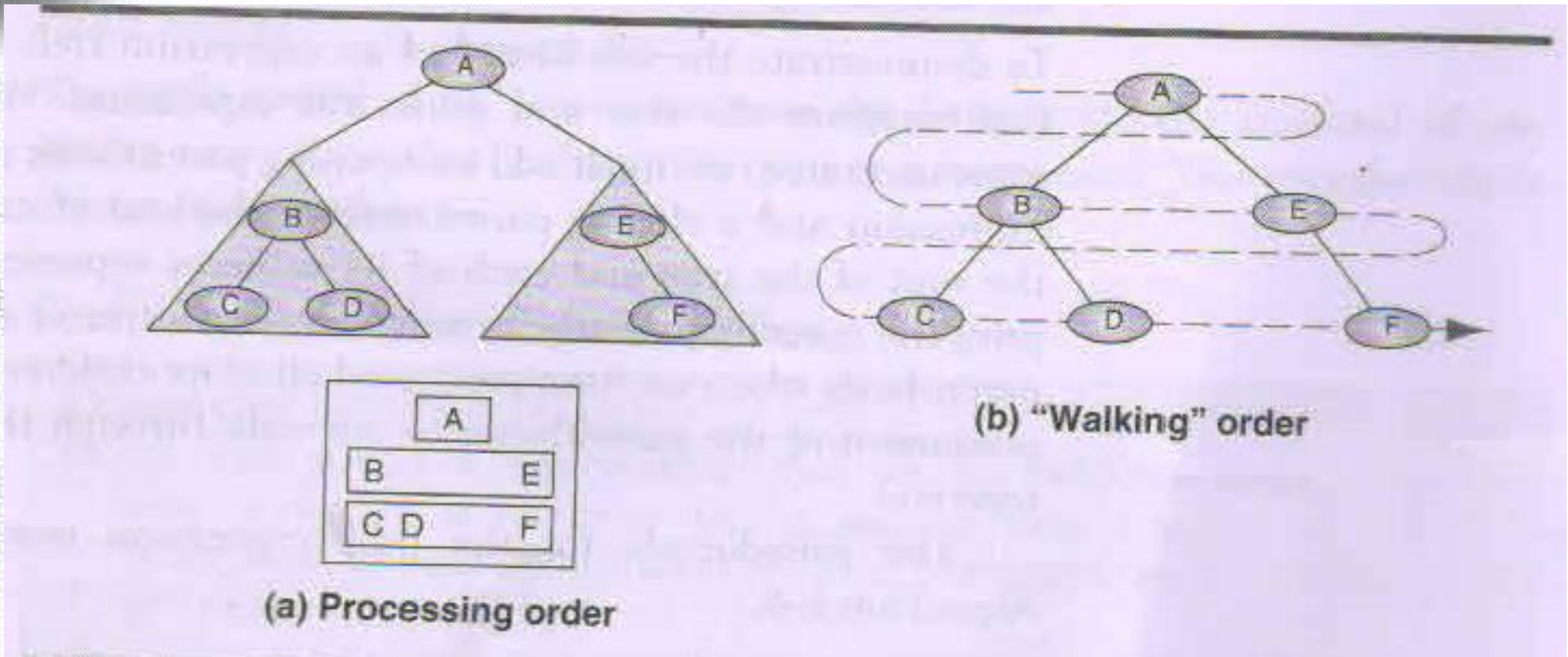
(a) Processing order

Postorder Traversal—C D B F E A

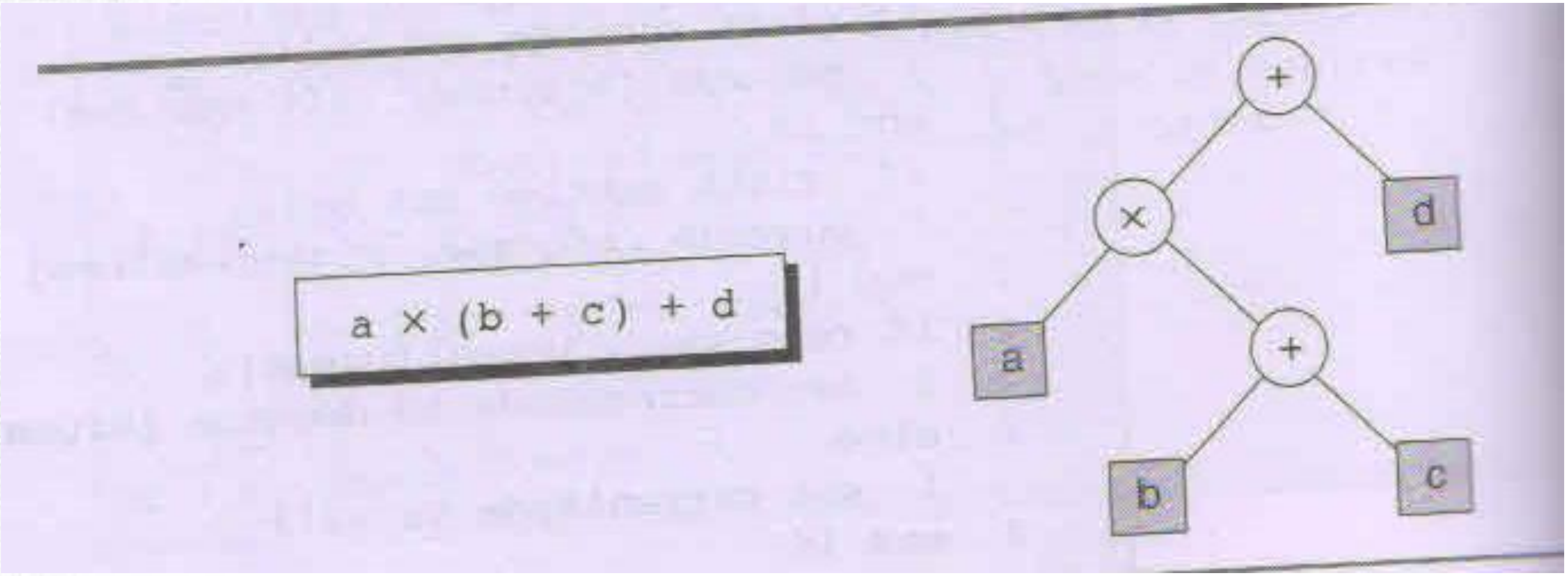
Breadth-First-Search

```
Algorithm breadthFirst (root)
Process tree using breadth-first traversal.
   Pre    root is node to be processed
   Post   tree has been processed
1 set currentNode to root
2 createQueue (bfQueue)
3 loop (currentNode not null)
   1 process (currentNode)
   2 if (left subtree not null)
      1 enqueue (bfQueue, left subtree)
   3 end if
   4 if (right subtree not null)
      1 enqueue (bfQueue, right subtree)
   5 end if
   6 if (not emptyQueue(bfQueue))
      1 set currentNode to dequeue (bfQueue)
   7 else
      1 set currentNode to null
   8 end if
4 end loop
5 destroyQueue (bfQueue)
end breadthFirst
```

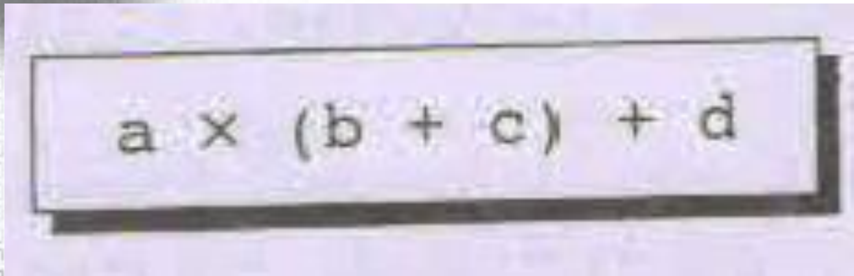
Breadth-First-Search

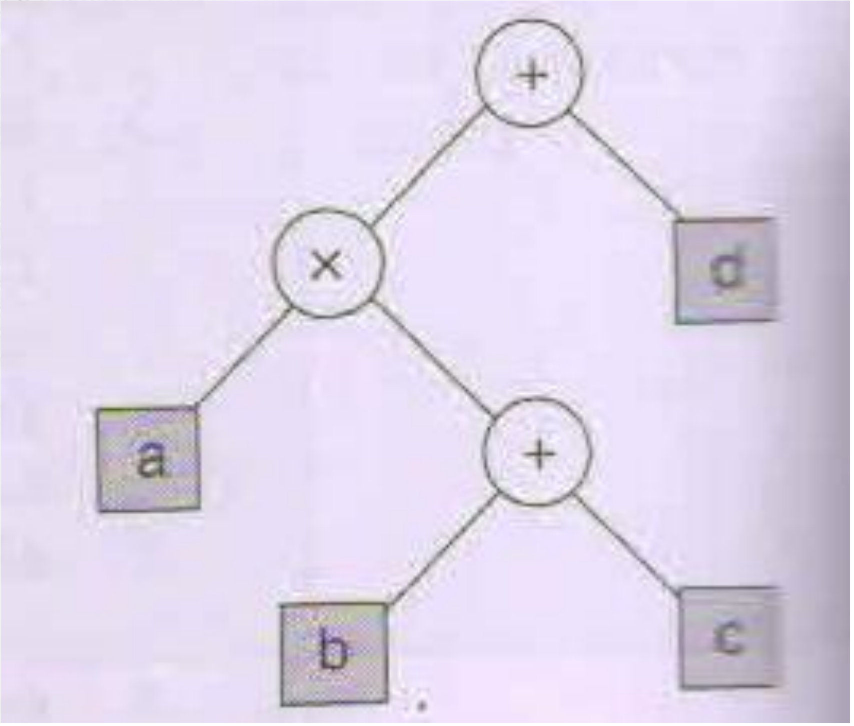


Expression Tree



Expression Tree





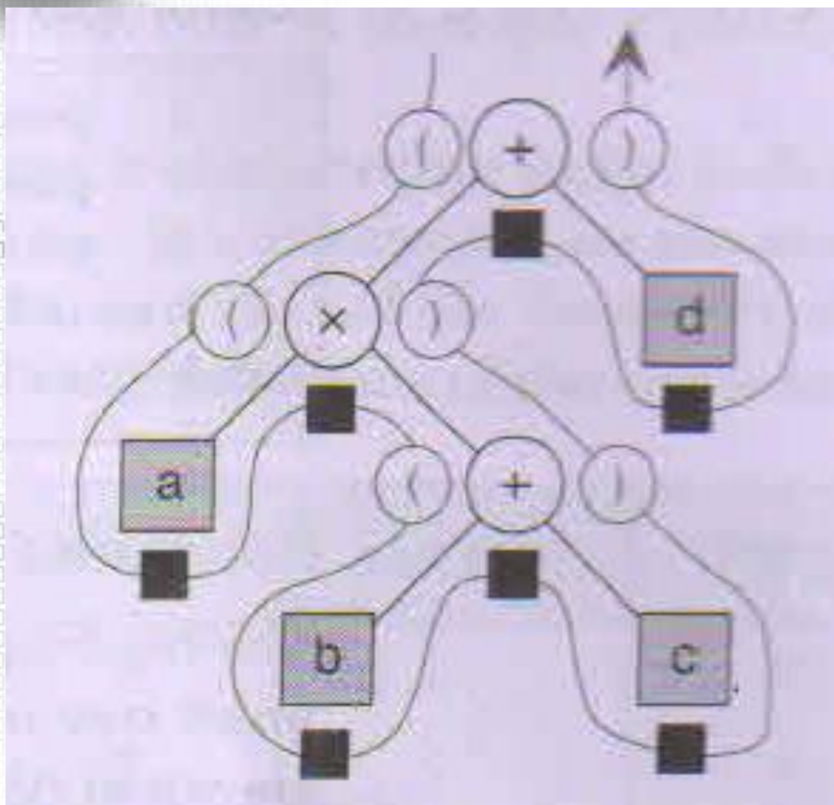
An Expression Tree is a binary tree with following properties

- Each leaf is an operand
- The root and internal nodes are operators
- Subtrees are expressions, with the root being an operator.

Infix Traversal of an Expression Tree

```
Algorithm infix (tree)
Print the infix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the infix expression has been printed
 1 if (tree not empty)
   1  if (tree token is an operand)
      1  print (tree-token)
   2  else
      1  print (open parenthesis)
      2  infix (tree left subtree)
      3  print (tree token)
      4  infix (tree right subtree)
      5  print (close parenthesis)
   3  end if
 2 end if
end infix
```

Infix Traversal of an Expression Tree



((a × (b + c)) + d)

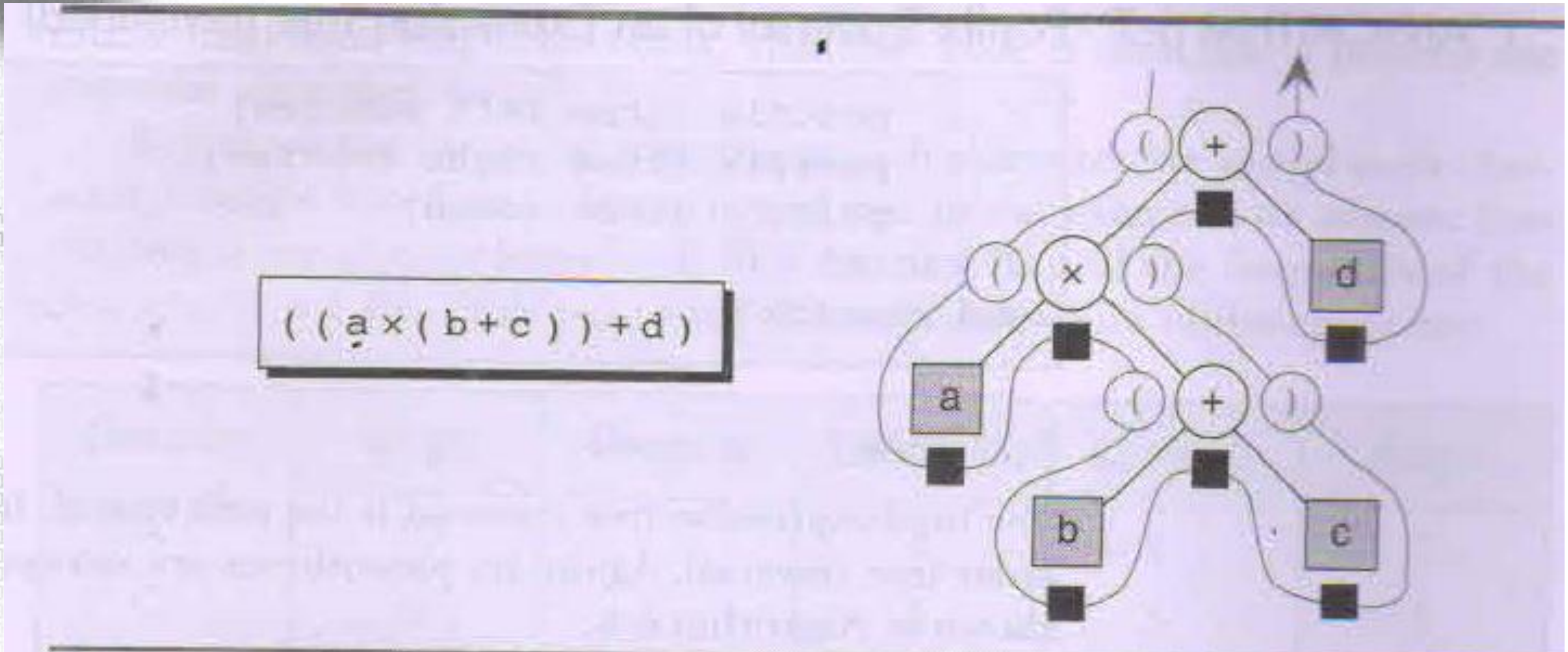
Postfix Traversal of an Expression Tree

```
Algorithm postfix (tree)
Print the postfix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the postfix expression has been printed
1 if (tree not empty)
   1 postfix (tree left subtree)
   2 postfix (tree right subtree)
   3 print   (tree token)
2 end if
end postfix
```

Prefix Traversal of an Expression Tree

```
Algorithm prefix (tree)
Print the prefix expression for an expression tree
  Pre  tree is a pointer to an expression tree
  Post the prefix expression has been printed
1 if (tree not empty)
   1  print  (tree token)
   2  prefix (tree left subtree)
   3  prefix (tree right subtree)
2 end if
end prefix
```

Expression Tree



Infix Traversal of an Expression Tree

Infix Expression Tree Traversal

```
Algorithm infix (tree)
Print the infix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the infix expression has been printed
1 if (tree not empty)
   1 if (tree token is an operand)
      1 print (tree-token)
   2 else
      1 print (open parenthesis)
      2 infix (tree left subtree)
      3 print (tree token)
      4 infix (tree right subtree)
      5 print (close parenthesis)
   3 end if
2 end if
end infix
```

Assignment: Infix & Postfix

Binary Search Tree

- All items in the left subtree are less than the root
- All items in the right subtree are greater than or equal to the root
- Each subtree is itself a BST

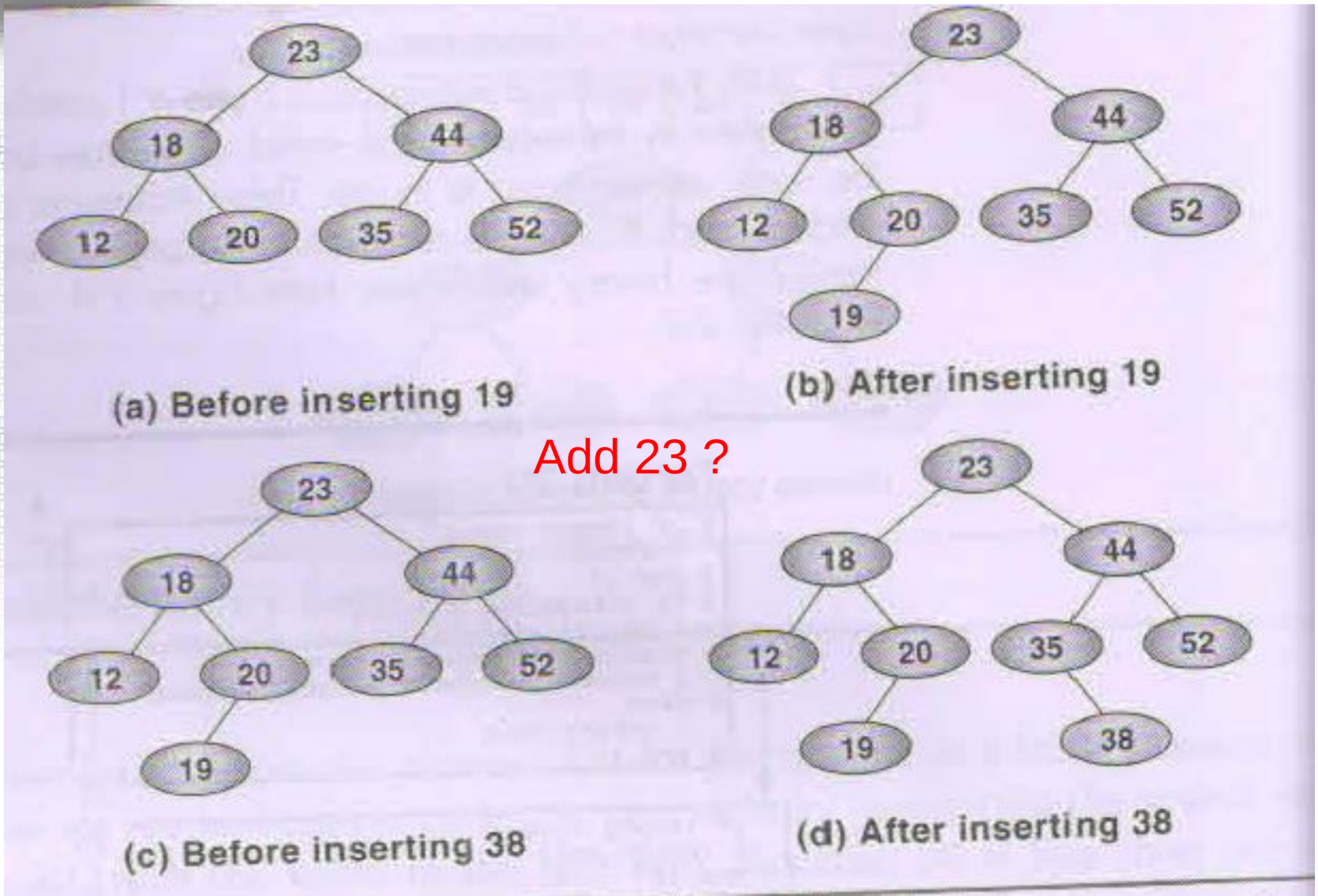
BST operations

1. Traversal
2. Find Largest Node
3. Find Smallest Node
4. Searching
5. Insertion
6. Deletion

BST- Searching

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
  Pre    root is the root to a binary tree or subtree
         targetKey is the key value requested
  Return the node address if the value is found
         null if the node is not in the tree
1 if (empty tree)
     Not found
  1  return null
2 end if
3 if (targetKey < root)
  1  return searchBST (left subtree, targetKey)
4 else if (targetKey > root)
  1  return searchBST (right subtree, targetKey)
5 else
     Found target key
  1  return root
6 end if
end searchBST
```

BST- Insertion



Add 23 ?

BST- Insertion

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
 1 if (empty tree)
   1  set root to newNode
   2  return newNode
 2 end if
   Locate null subtree for insertion
 3 if (newNode < root)
   1  return addBST (left subtree, newNode)
 4 else
   1  return addBST (right subtree, newNode)
 5 end if
end addBST
```

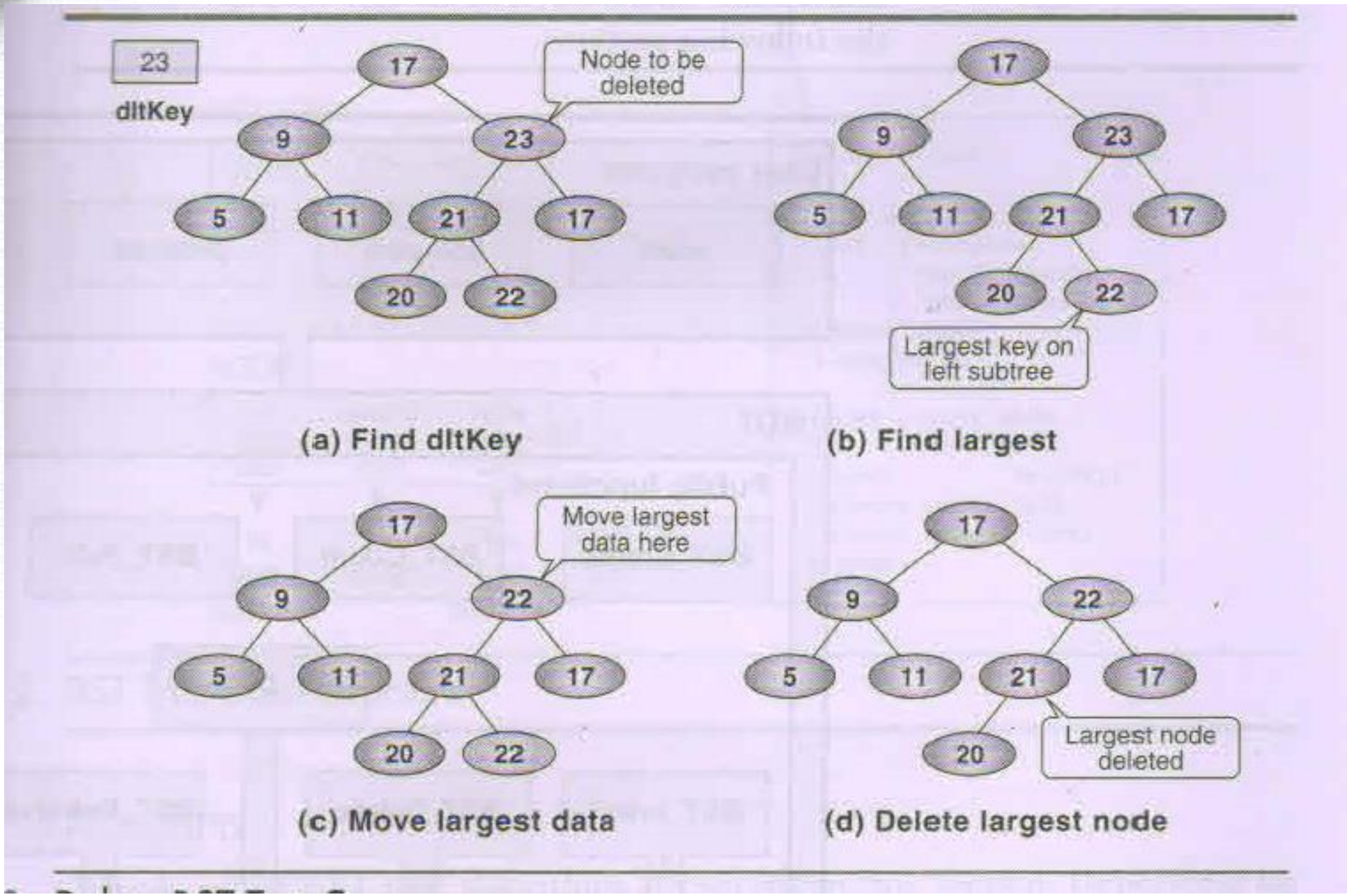
Deletion : BST

Deletion

To **delete a node** from a binary search tree, we must first locate it. There are four possible cases when we delete a node:

1. The node to be deleted has no children. In this case, all we need to do is delete the node.
2. The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
3. The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.
4. The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced

Deletion


23
dltKey
17
9
23
5
11
21
17
20
22
Node to be deleted
(a) Find dltKey
17
9
23
5
11
21
17
20
22
Largest key on left subtree
(b) Find largest
17
9
22
5
11
21
17
20
22
Move largest data here
(c) Move largest data
17
9
22
5
11
21
17
20
Largest node deleted
(d) Delete largest node

Deletion

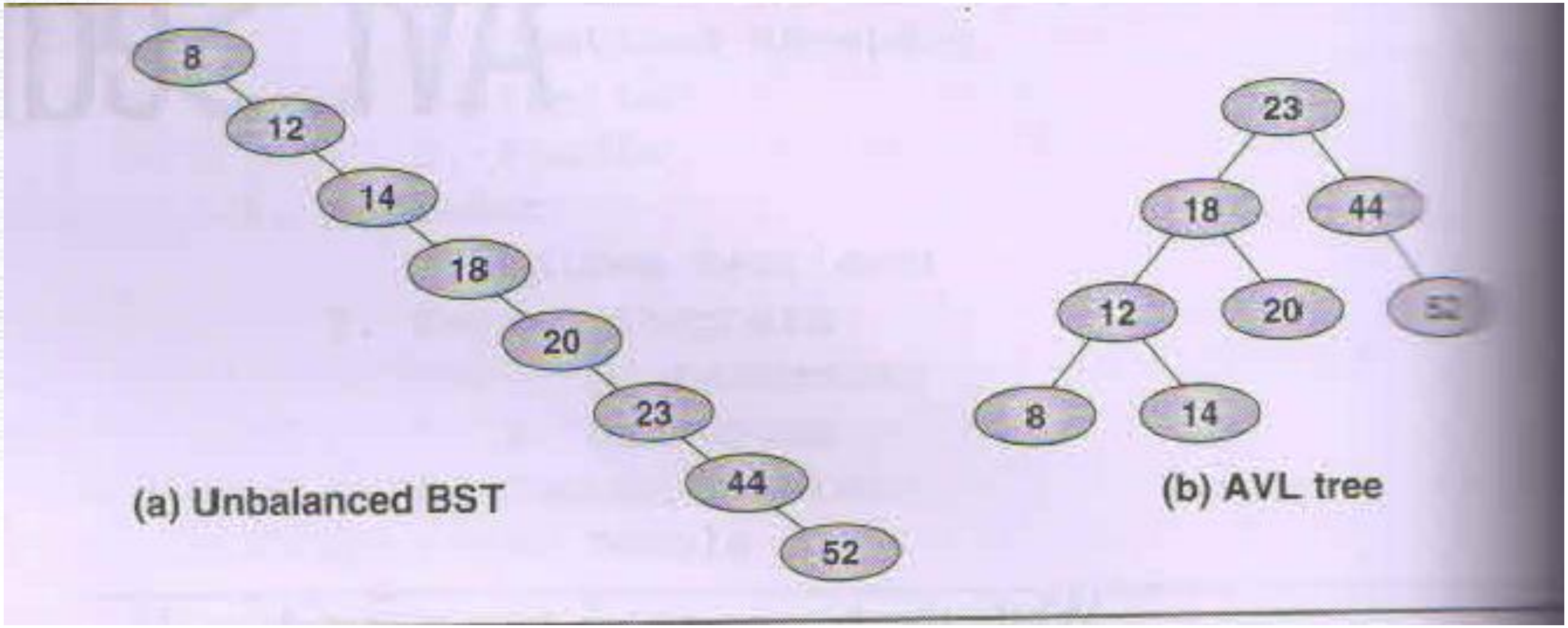
```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
  Pre     root is reference to node to be deleted
          dltKey is key of node to be deleted
  Post    node deleted
          if dltKey not found, root unchanged
  Return true if node deleted, false if not found
1 if (empty tree)
  1  return false
2 end if
3 if (dltKey < root)
  1  return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
  1  return deleteBST (right subtree, dltKey)
```

Deletion

```
5 else
     Delete node found--test for leaf node
  1  If (no left subtree)
     1  make right subtree the root
     2  return true
  2  else if (no right subtree)
     1  make left subtree the root
     2  return true
  3  else
        Node to be deleted not a leaf. Find largest
        left subtree.
     1  save root in deleteNode
     2  set largest to largestBST (left subtree)
     3  move data in largest to deleteNode
     4  return deleteBST (left subtree of deleteNode,
                                   key of largest
  4  end if
6 end if
end deleteBST
```

AVL Tree

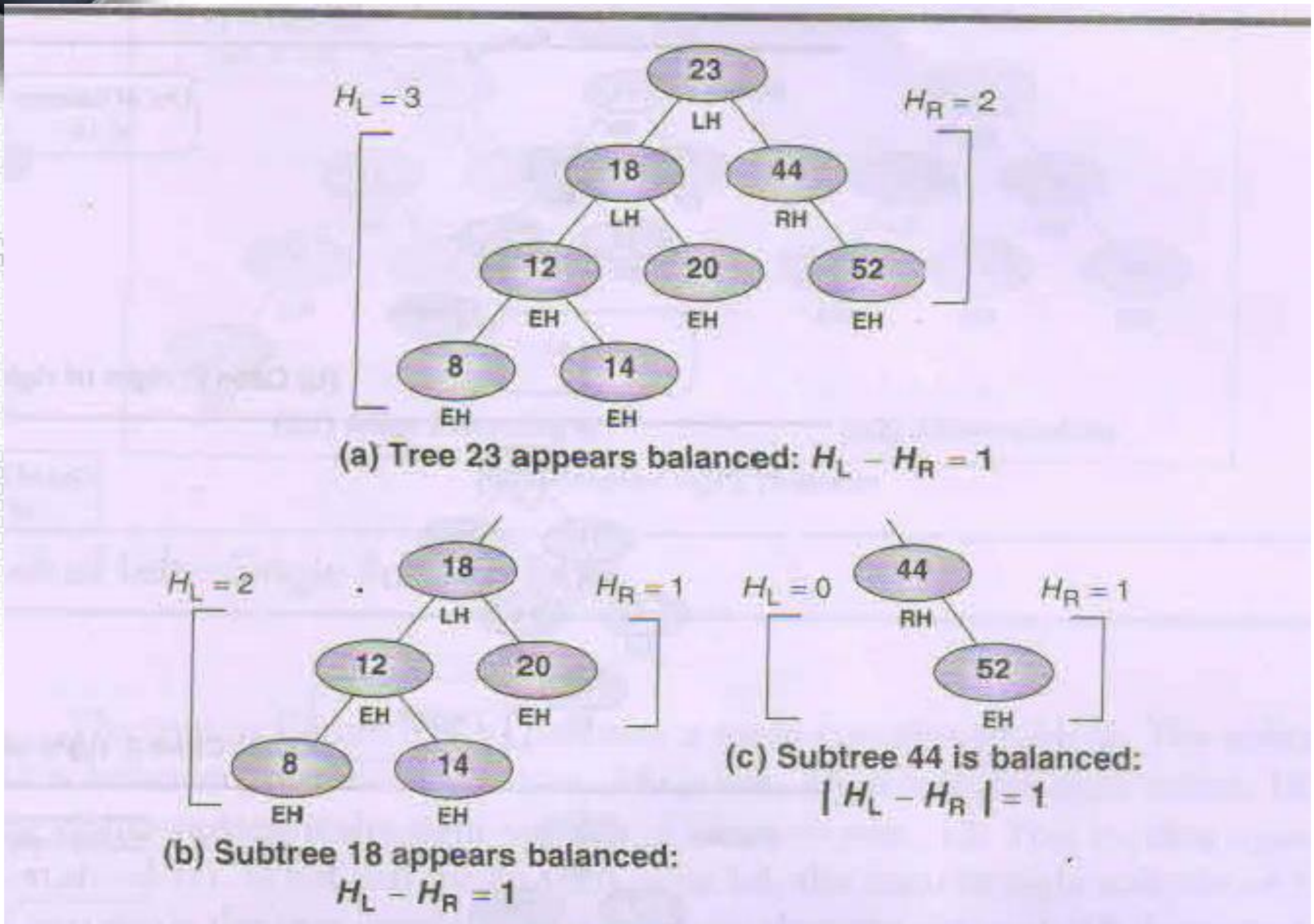
- An AVL tree is a height-balanced BST



$| H_L – H_R | <= 1$ **Balance Factor +1, 0, -1**

Time Complexity : O (log n)

AVL Tree



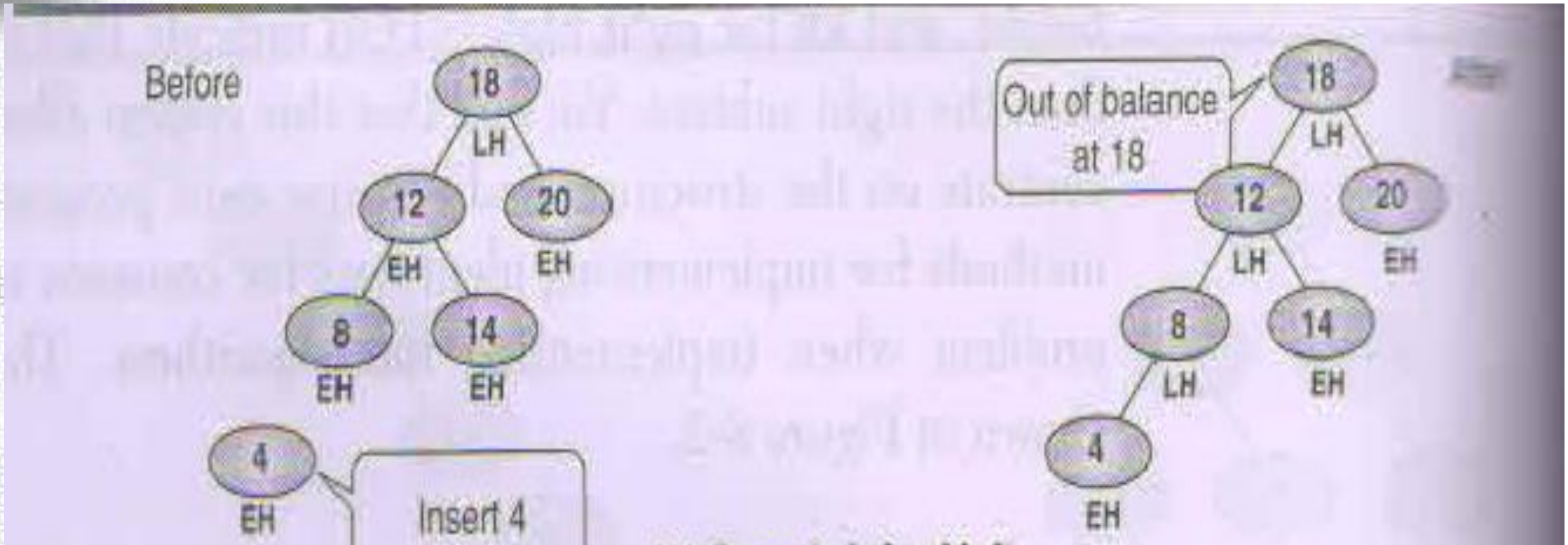
$H_L = 3$
$H_R = 2$
23
LH
18
LH
44
RH
12
EH
20
EH
52
EH
8
EH
14
EH
(a) Tree 23 appears balanced: $H_L - H_R = 1$
$H_L = 2$
$H_R = 1$
18
LH
12
EH
20
EH
8
EH
14
EH
(b) Subtree 18 appears balanced: $H_L - H_R = 1$
$H_L = 0$
$H_R = 1$
44
RH
52
EH
(c) Subtree 44 is balanced: $| H_L - H_R | = 1$

AVL Tree – Balancing Trees

We consider Four cases that require balancing

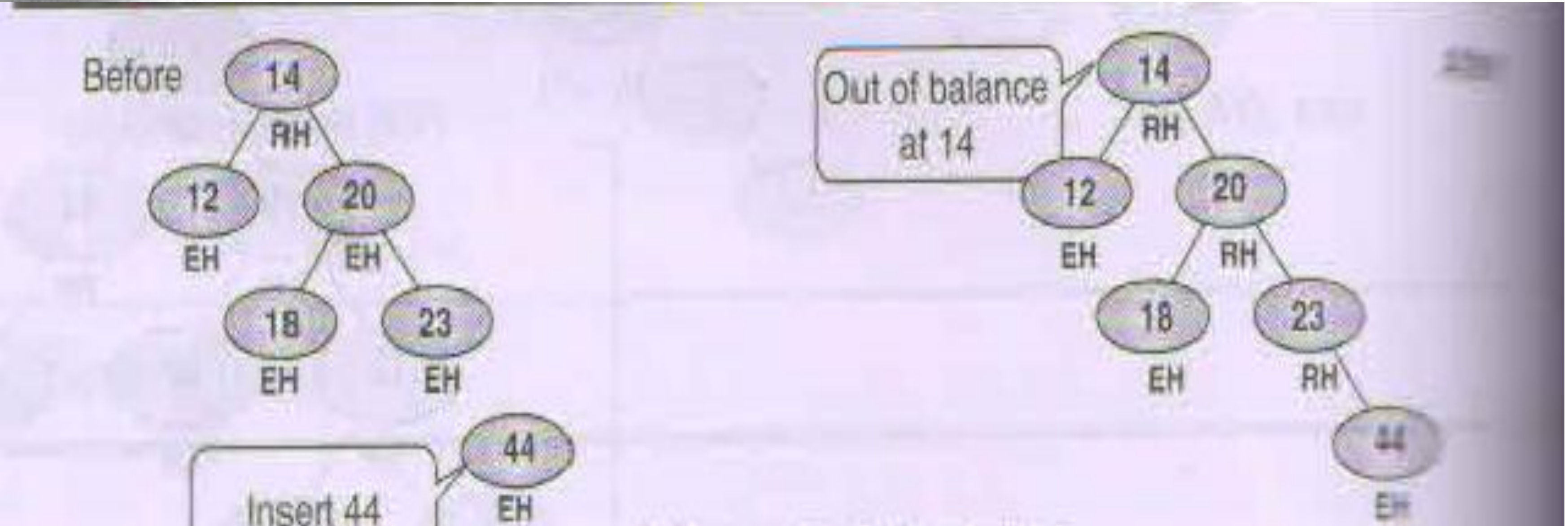
1. Left of Left : A subtree of a tree that is left high has also become left high
2. Right of Right : A subtree of a tree that is right high has also become right high
3. Right of Left : A subtree of a tree that is left high has become right high
4. Left of Right : A subtree of a tree that is right high has become left high

AVL Tree – Left of Left



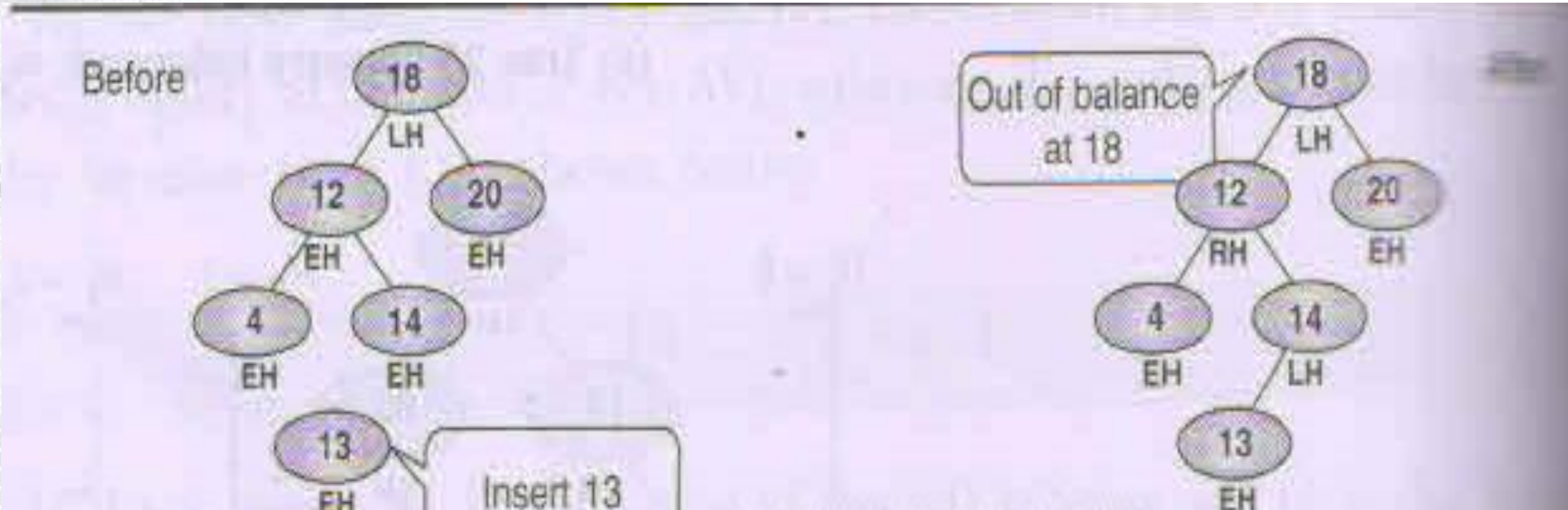
(a) Case 1: left of left

AVL Tree – Right of Right



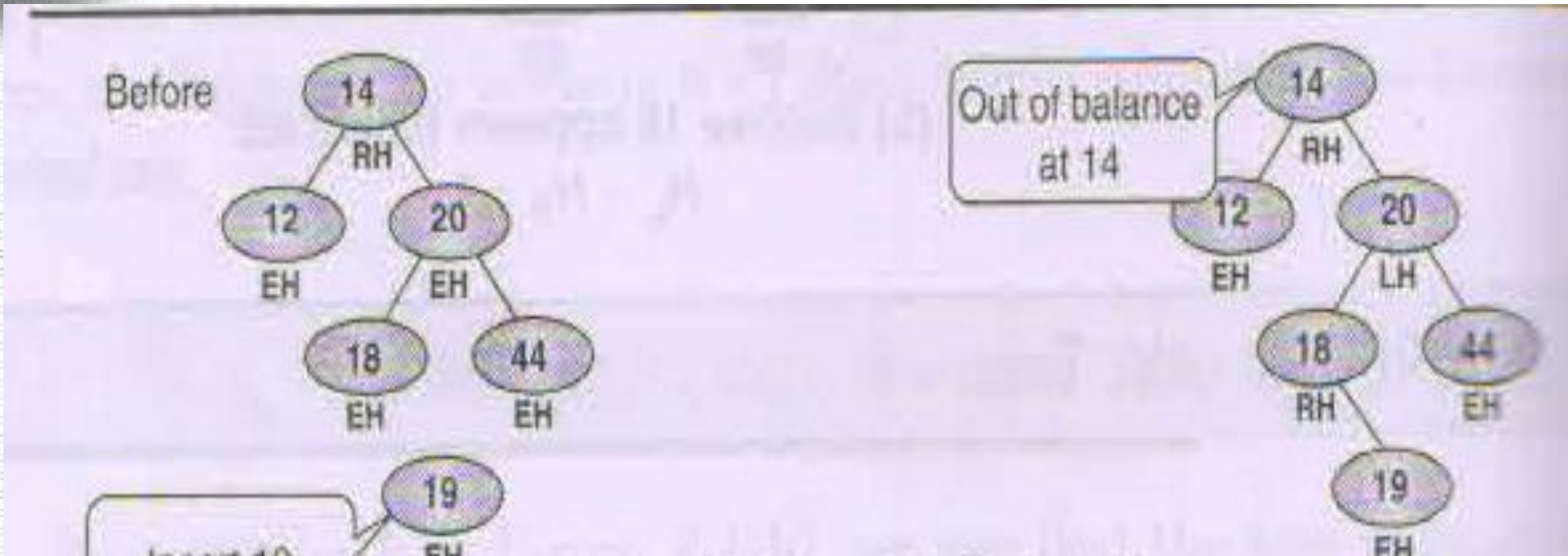
(b) Case 2: right of right

AVL Tree – Right of Left



(c) Case 3: right of left

AVL Tree – Left of Right



(d) Case 4: left of right

Assignment : Balancing by Rotation

General Trees

- Unlimited out degree for each node

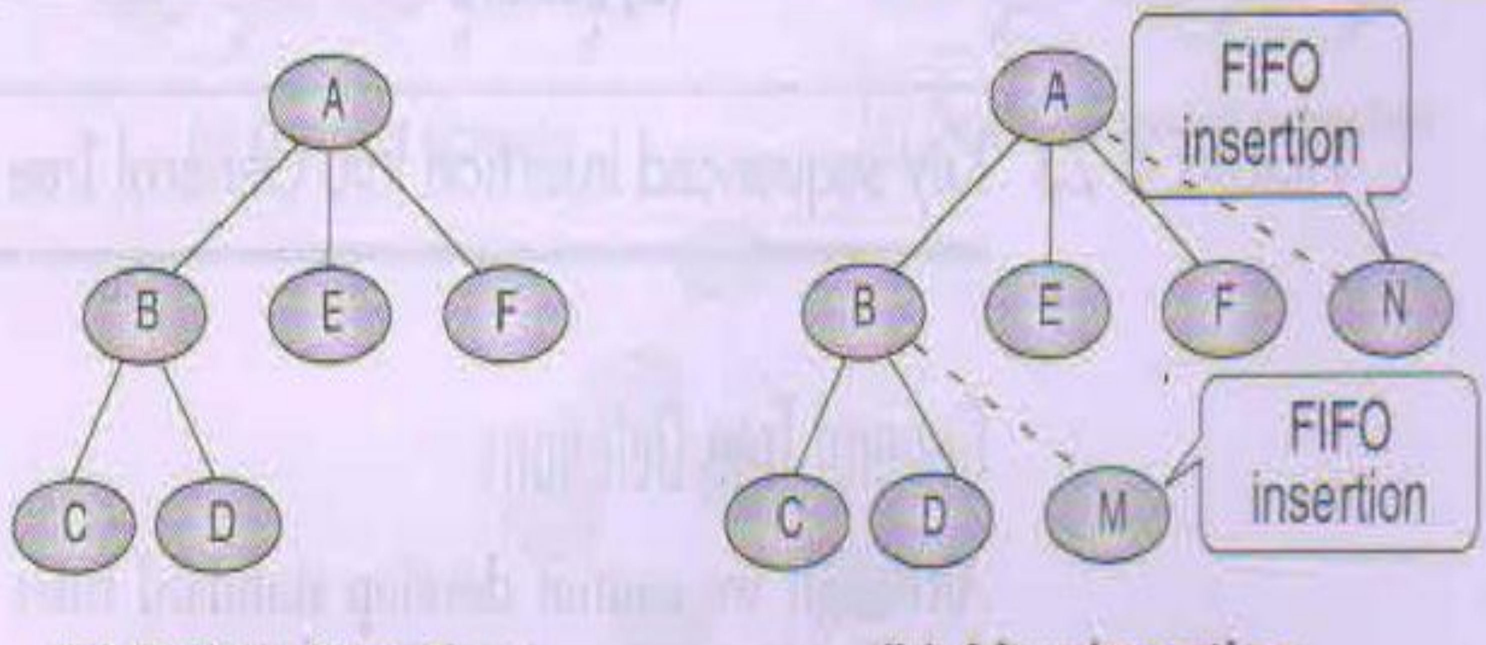
Insertion into General tree

- User must supply Parent of the node.

Three different rules may be used

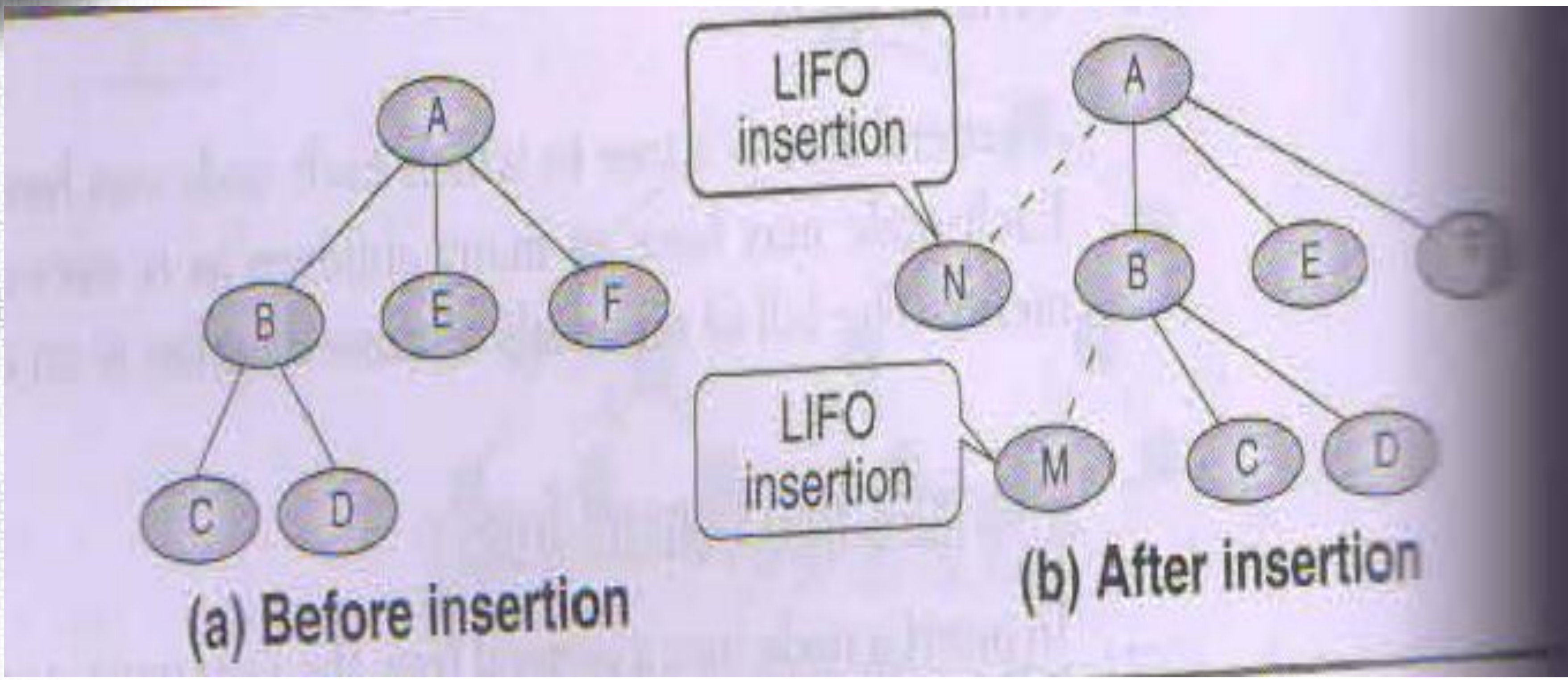
1. FIFO
2. LIFO
3. Key-sequence insertion

General Trees – FIFO Insertion

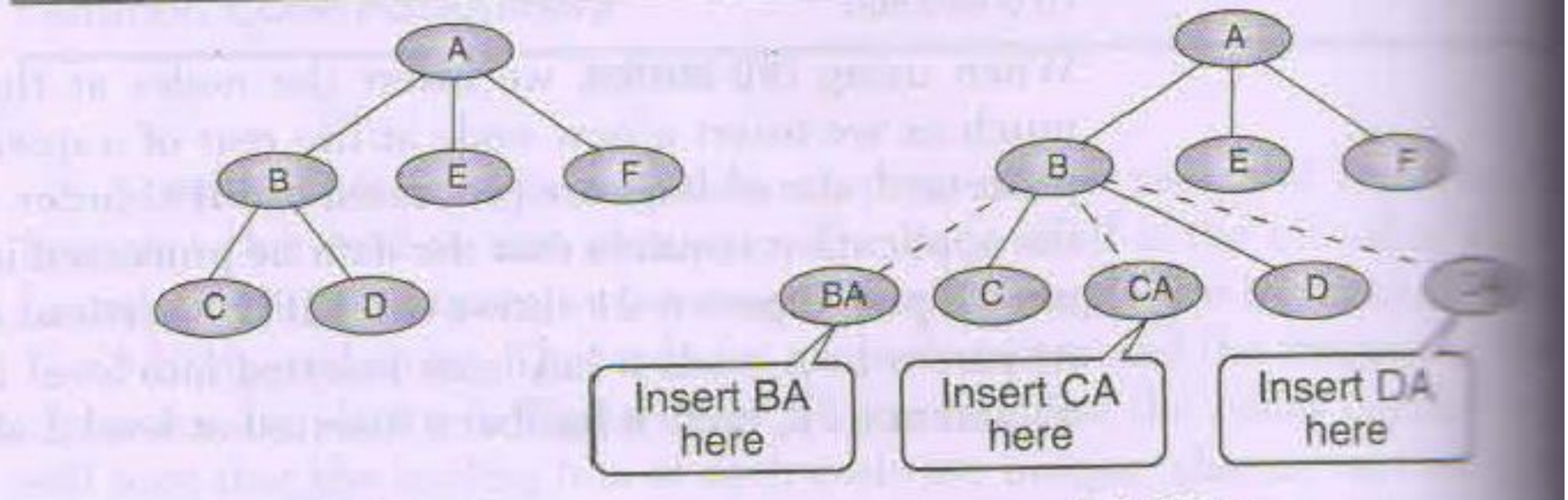


(a) Before insertion
(b) After insertion

General Trees – LIFO Insertion



General Trees – Key-sequence Insertion



(a) Before (b) After