CS 133 – Data Structures and File Organization

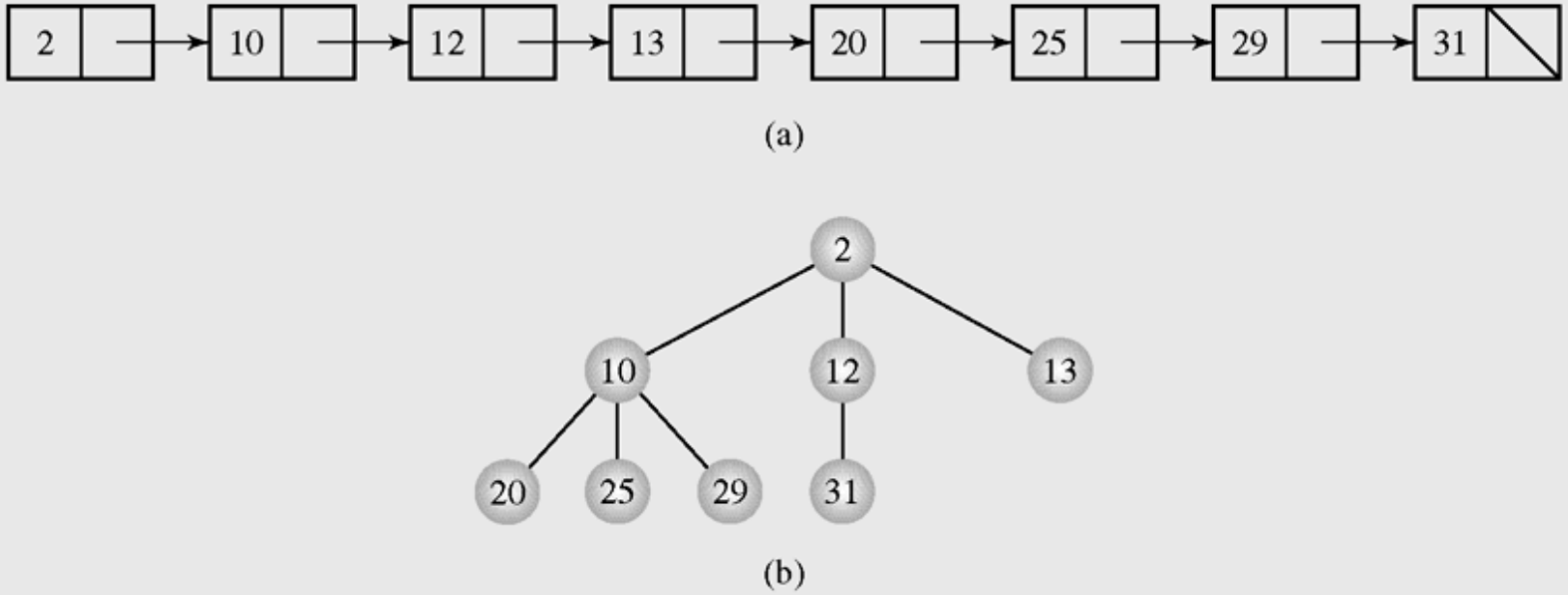
Chapter 6

Binary Tree

Linked Lists

- Over previous weeks we have investigated Linked Lists
  - A ***linear*** data structure, used to implement queues or stacks etc.
- Stacks and Queues do represent a hierarchy, it is however a one dimensional hierarchy.
- Many fields contain hierarchical data structures, so this week we begin to investigate a new data structure called Trees.

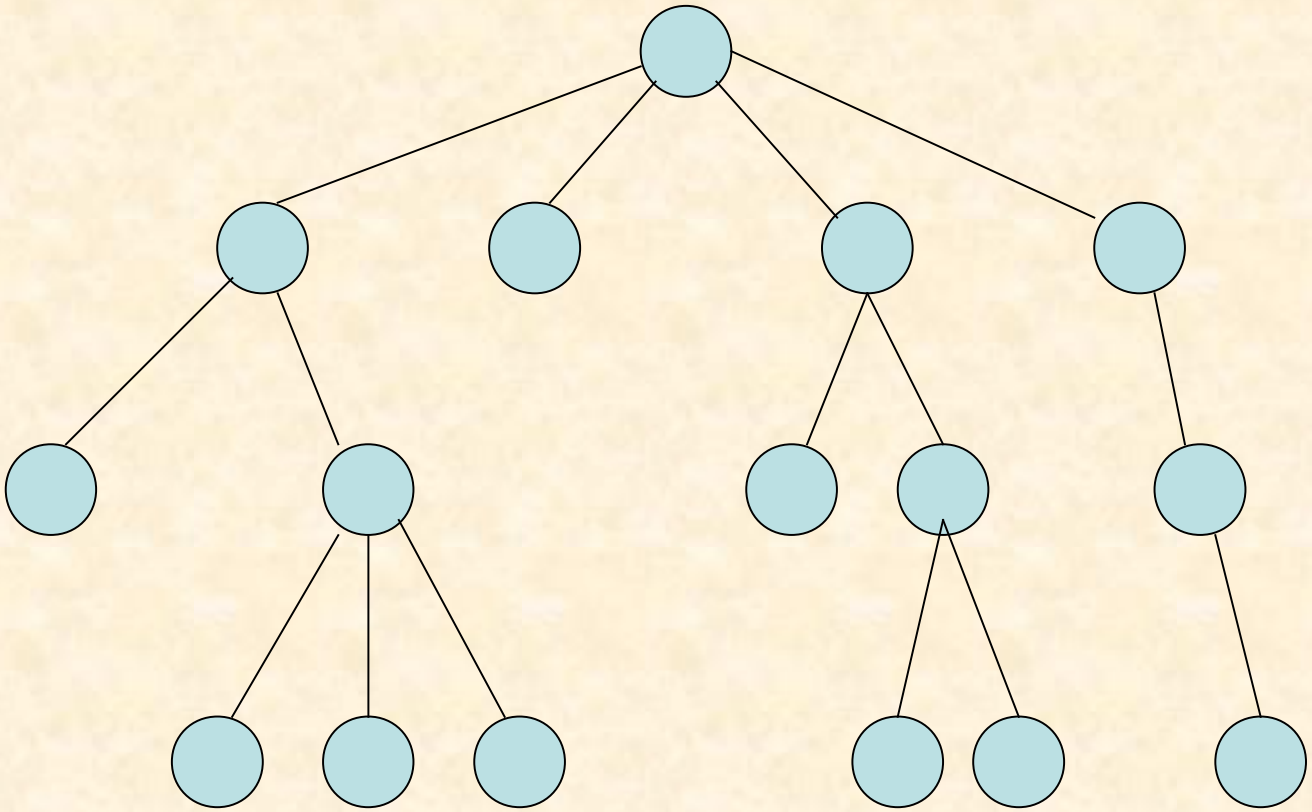
FIGURE 6.3 Transforming (a) a linked list into (b) a tree.



Trees

- Trees can be pictured like upside down trees;
  - With a root at the top
  - Branches pointing downwards
  - And leaves at the bottom.
- A tree consists of nodes and connecting arcs
  - For each node there is a unique path from the root.
  - The number of arcs in this sequence is known as the length of the path
  - The level of the node is the length of the path + 1.
  - The height of the tree is the same as the highest height.

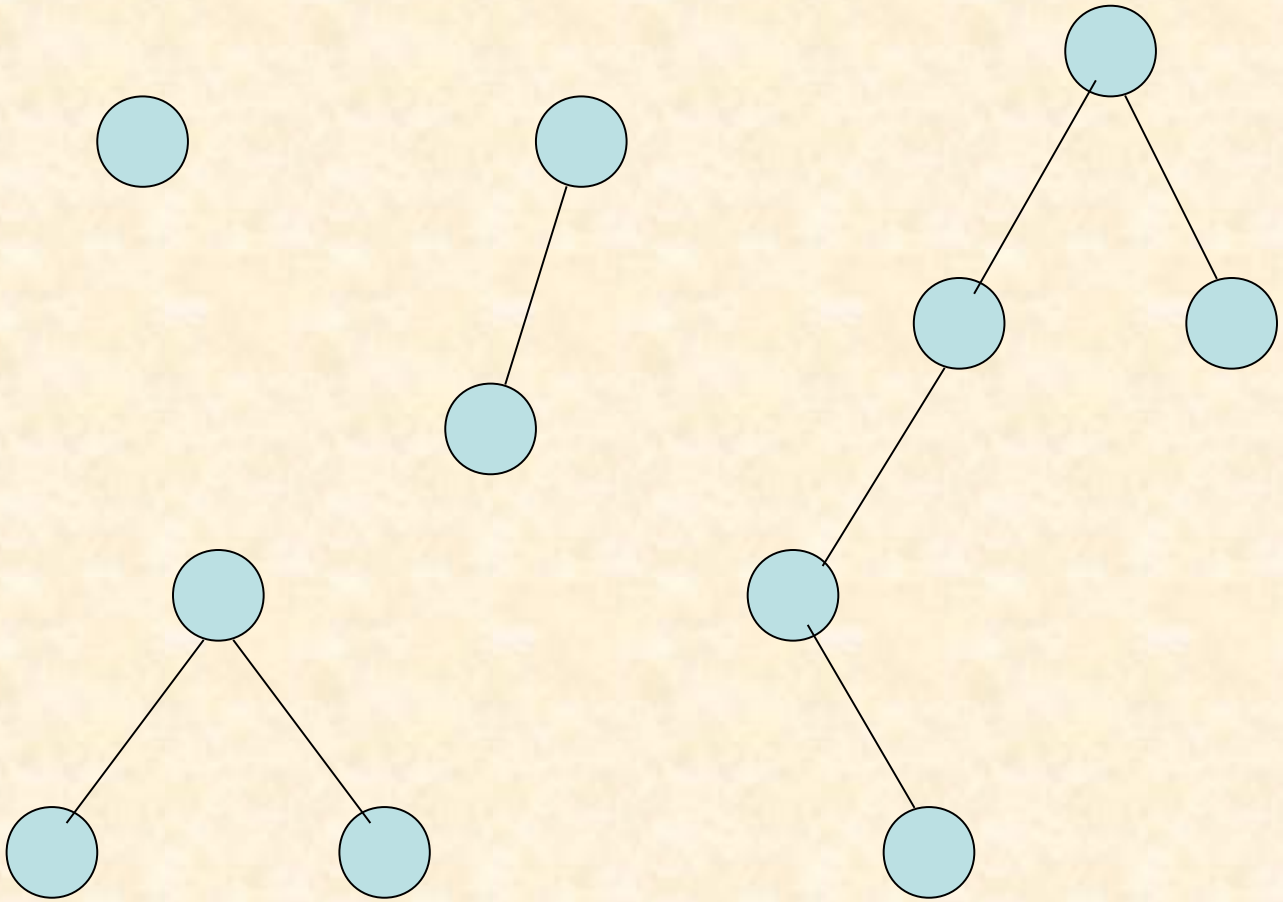
A Tree



Binary Trees

- A binary tree is a tree whose nodes have a maximum of 2 children – i.e. 0, 1 or 2 branches coming from any node.
  - The tree on the previous slide is not a binary tree, the following slide gives some binary tree examples.
- When dealing with binary trees it is worth considering how we have used ‘If-Else’ statements in the past.

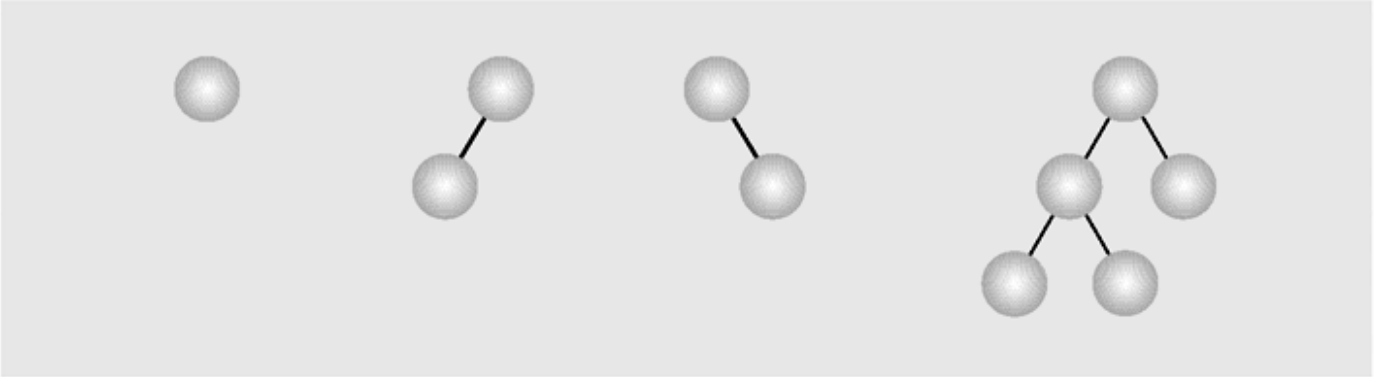
Binary Trees



Binary Trees

- The final example on the previous slide was an unbalanced binary tree.
  - The left hand side is longer (higher) than the right hand side.
- A *complete binary tree* is a tree where every non-terminal node has 2 branches.
  - *For all nonempty binary trees whose nonterminal nodes have exactly 2 nonempty children, the number of leaves (m) is greater than the number of nonterminal nodes (k), and m=k+1.*

FIGURE 6.4 Examples of binary trees.



Binary Search Tree

Binary Search Tree

- sometimes also be called an ordered or sorted binary tree
- a binary tree which has the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
  - Both the left and right subtrees must also be binary search trees.

Binary Search Tree

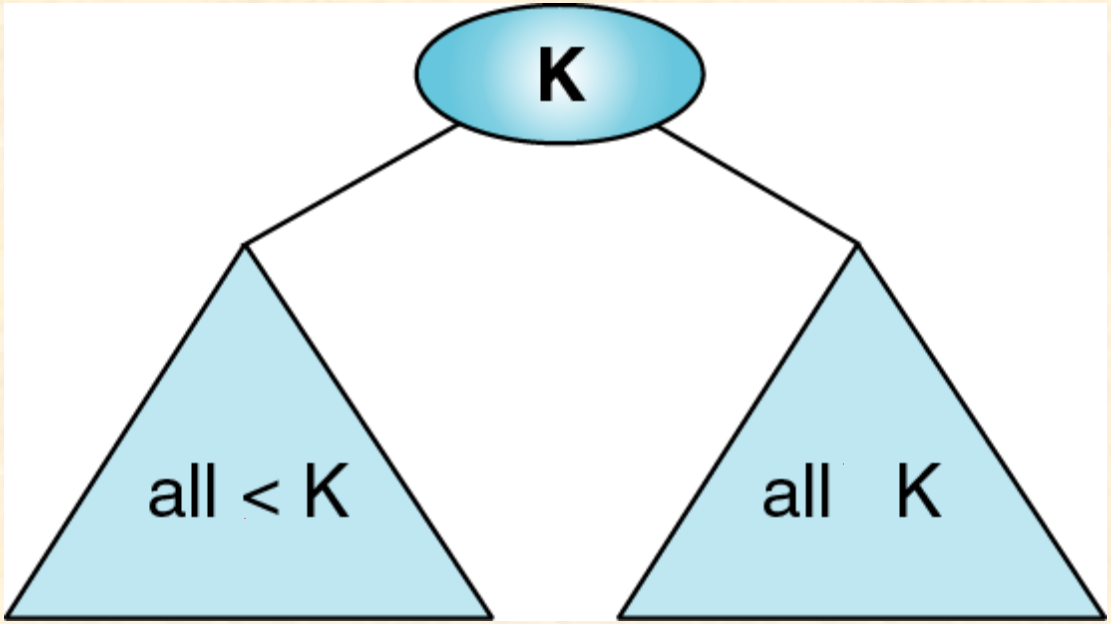
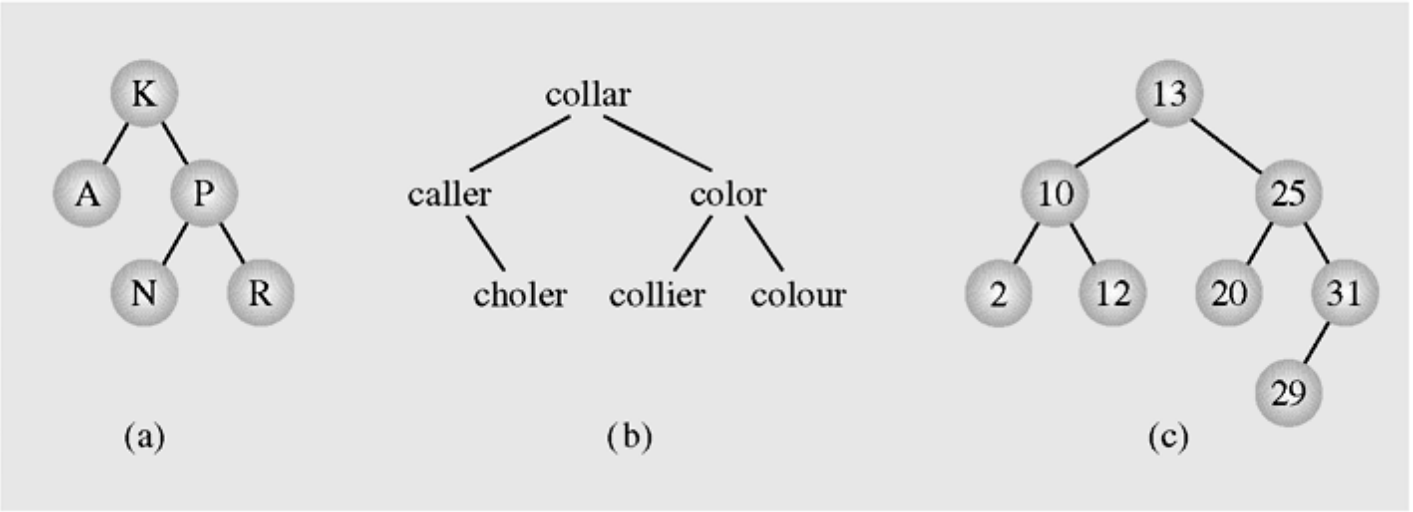
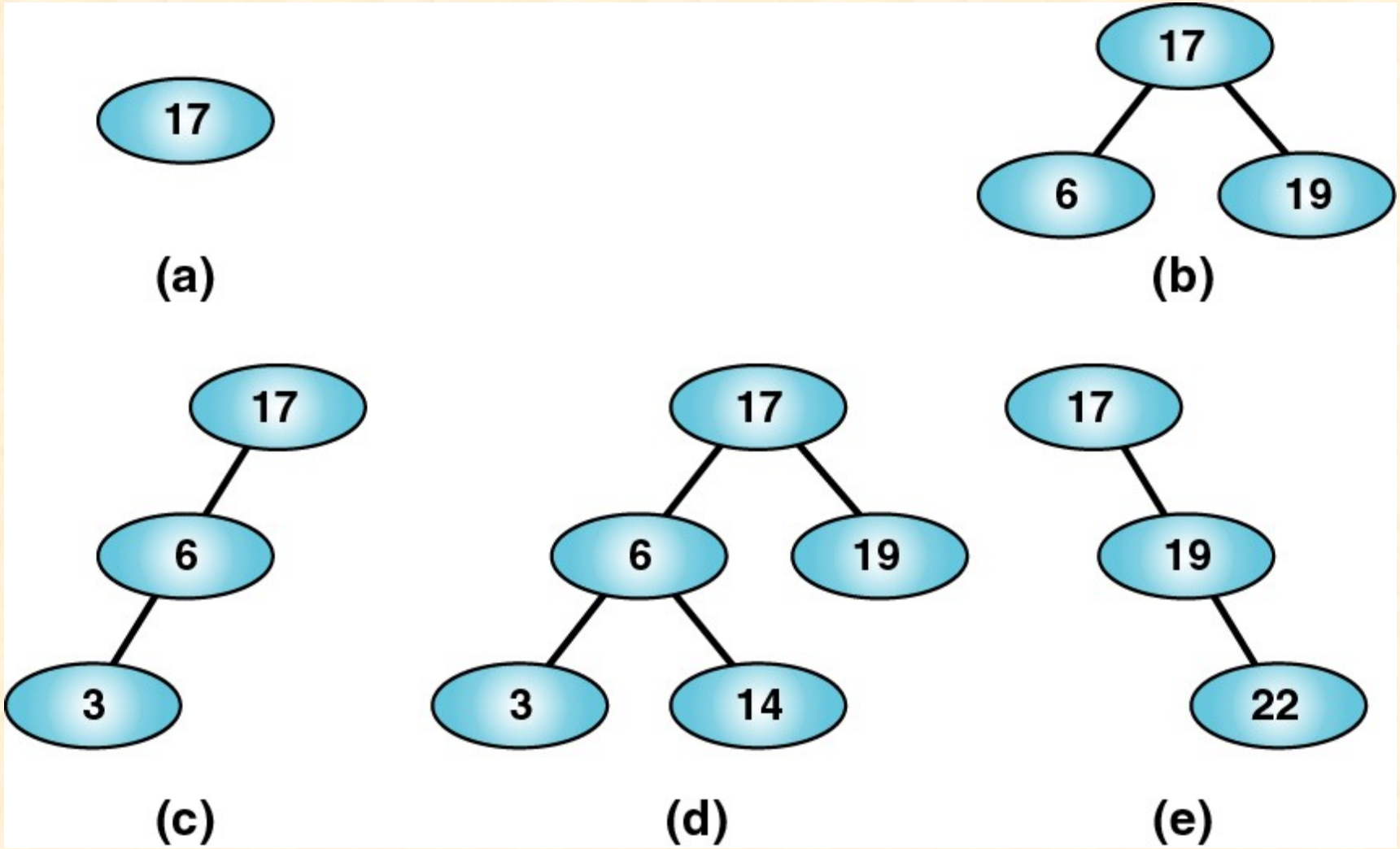


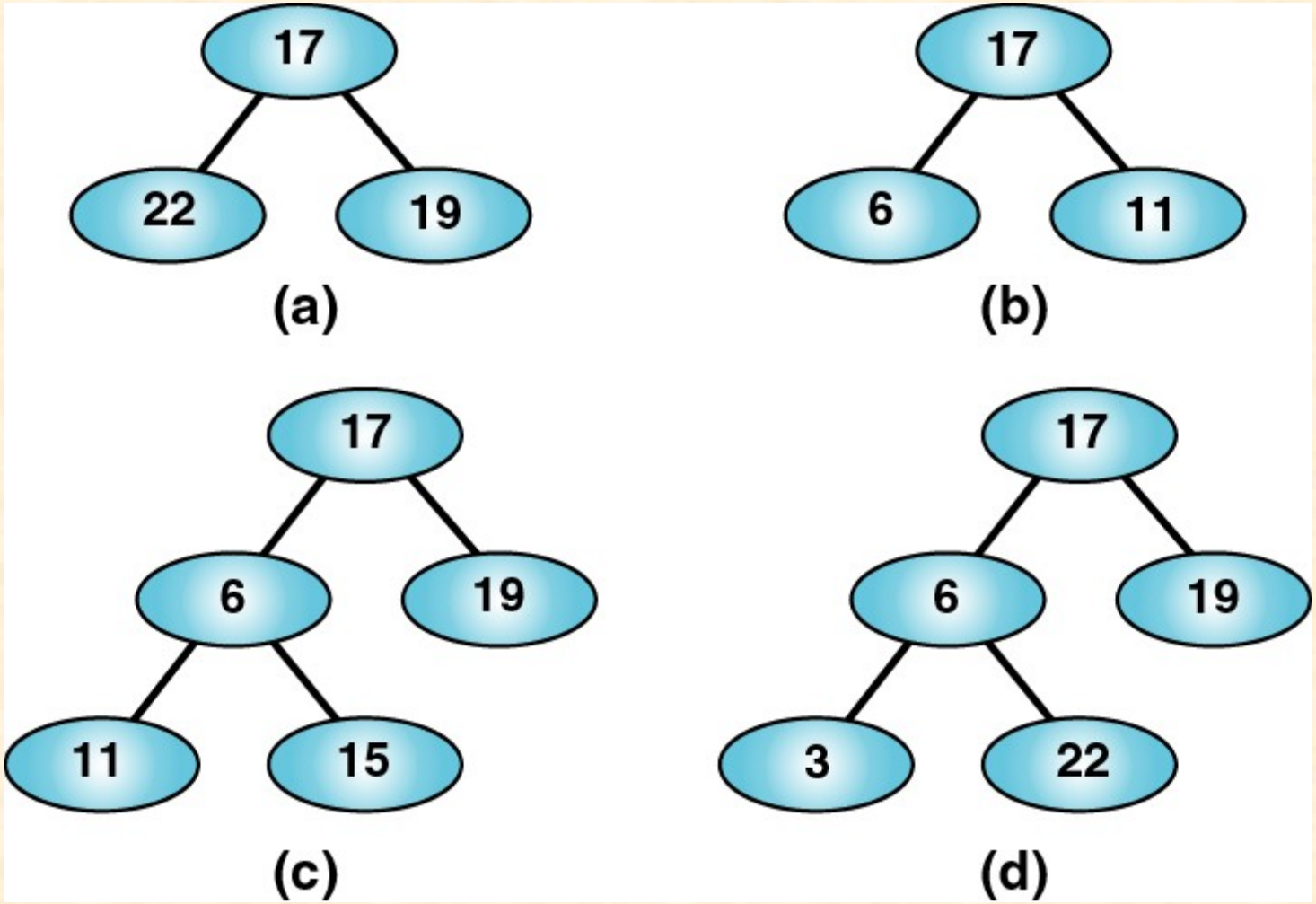
FIGURE 6.6 Examples of binary search trees.



More Examples of Binary Search Trees



Not Binary Search Trees



Array Implementation

- Implementing as an array is inconvenient as adding and removing nodes from a tree would leave a poorly structured array.
  - Deleting a node would leave empty cells
  - Inserting a node could mean nearby nodes are stored distinctly.

Searching a Binary Tree

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
        while (p != 0)
                if (el == p->key)
                        return &p->key;
                else if (el < p->key)
                        p = p->left;
                else
                        p = p->right;
        return 0;
}
```

Efficient searching

- Searching for a particular node is more efficient when using a complete tree
  - We will investigate creating a complete tree further later.
- Maintaining a complete tree
  - Involves investigating efficient insertion and deletion algorithms
  - We investigate insertion and deletion later
  - As well as self adjusting trees
- But first, another application of binary trees – traversal.

Tree Traversal

- Tree traversal is the process of visiting each node in the tree exactly once.
- Nodes can be visited in any order, which means for a tree with n nodes, there are n! different traversal patterns
  - Most of these are not practical
- Here we investigate different traversal strategies.

Tree Traversal Strategies

- Breadth First
  - Starting at either the highest or lowest node and visiting each level until the other end is reached.
    - Lowest to Highest
    - Highest to Lowest
- Depth First
  - Proceeding as far as possible to the right (or left), and then returning one level, stepping over and stepping down.
    - Left to Right
    - Right to Left

Breadth First Traversal

- Consider Top Down, left to right traversal.
  - Here we start with the root node and work downwards
    - The node's children are placed in a queue.
    - Then the next node is removed from the front of the queue

Depth First Traversal

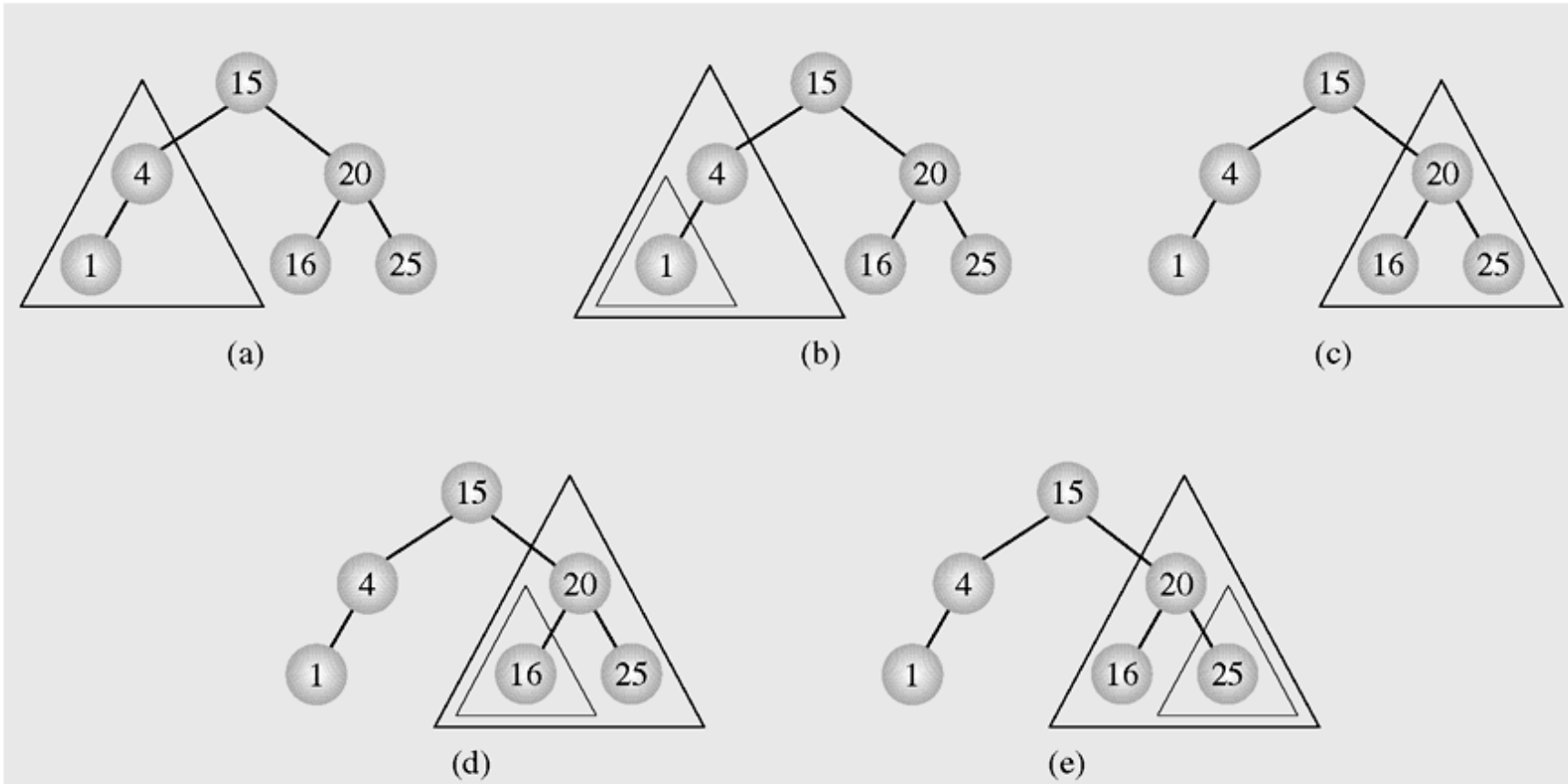
- V = visiting a node
- L = traversing the left subtree
- R = traversing the right subtree

- Options
  - **VLR (Pre Order Tree Traversal)**
  - VRL
  - **LVR (In order Tree Traversal)**
  - RVL
  - **LRV (Post order Tree Traversal)**
  - RLV

Depth First Traversal

- These Depth first traversals can easily be implemented using recursion;
  - In fact Double Recursion!

FIGURE 6.12 Inorder tree traversal.



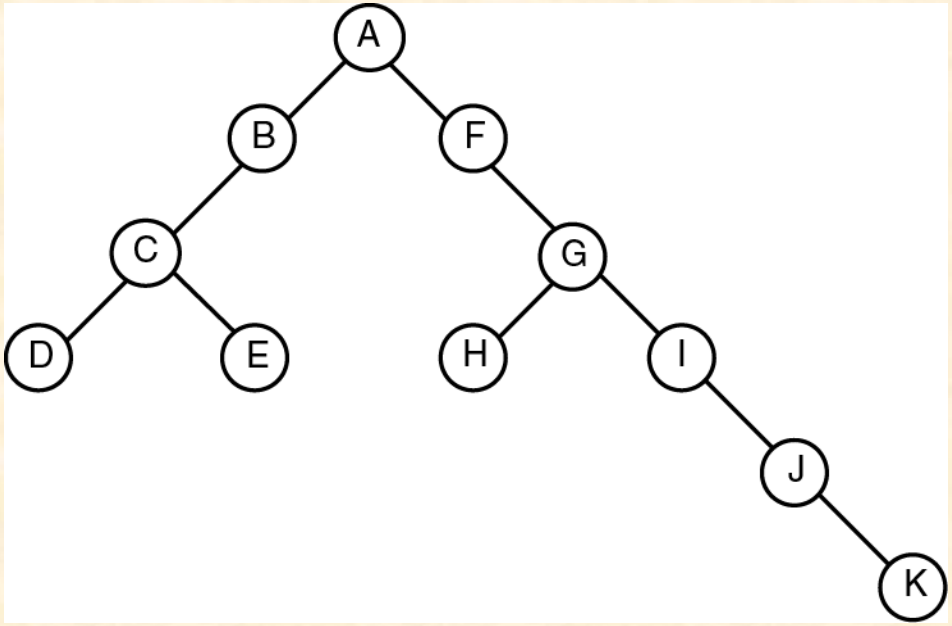
Pre Order

```
template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
   if (p!=0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
   }
}
```

Post Order

```
template<class T>
void BST<T>::postorder(BSTNode<T> *p) {
   if (p!=0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
   }
}
```

Example 1. Show the depth-first traversals (preorder, inorder, and postorder) and breadth-first traversal (level order) of the binary tree.



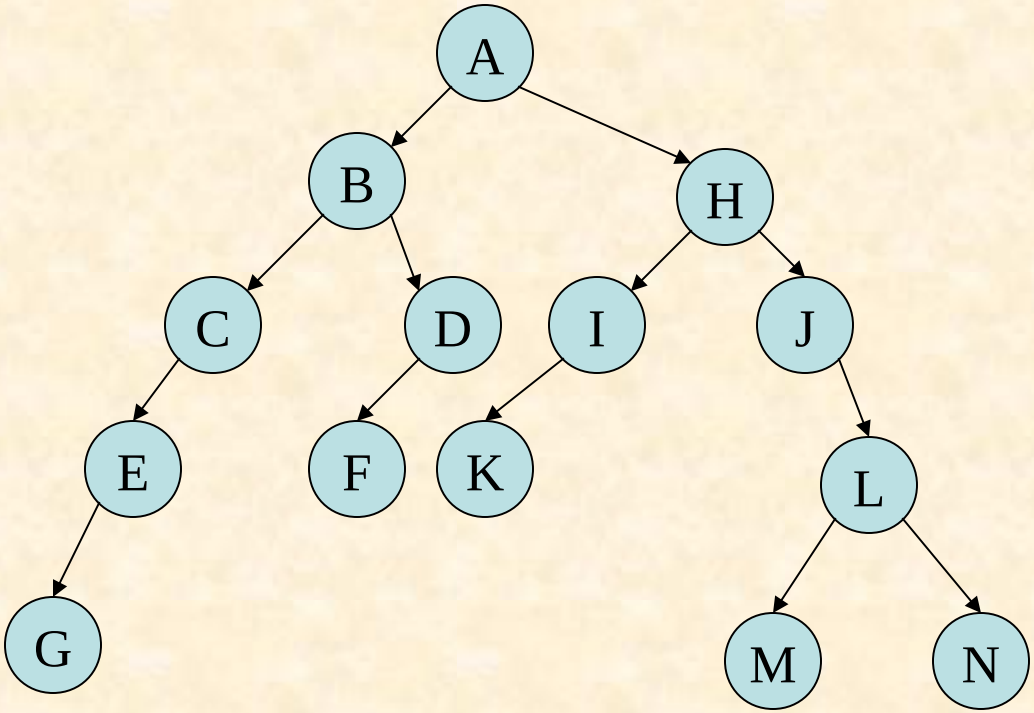
Preorder: A B C D E F G H I J K

Inorder: D C E B A F H G I J K

Postorder: D E C B H K J I G F A

Level order: A B F C G D E H I J K

Example 2. Show the depth-first and breadth-first traversals of the binary tree shown below.



Inorder:

GECBFDAKIHJMLN

Preorder:

ABCEGDFHIKJLMN

Postorder:

GECFDBKIMNLJHA

Level order:

ABHCDIJEFKLGMN

Example 3. Find the root of each of the following binary trees.

a. Tree with postorder traversal: FCBDG

Answer: G

b. Tree with preorder traversal: IBCDFEN

Answer: I

c. Tree with inorder traversal: CBIDFGE

Answer: Cannot be determined from the information given since the tree can be organized in several different ways.

Example 4. A binary tree has ten nodes. The inorder and preorder traversals of the tree are shown below. Draw the tree.

Preorder: JCBADEFIGH

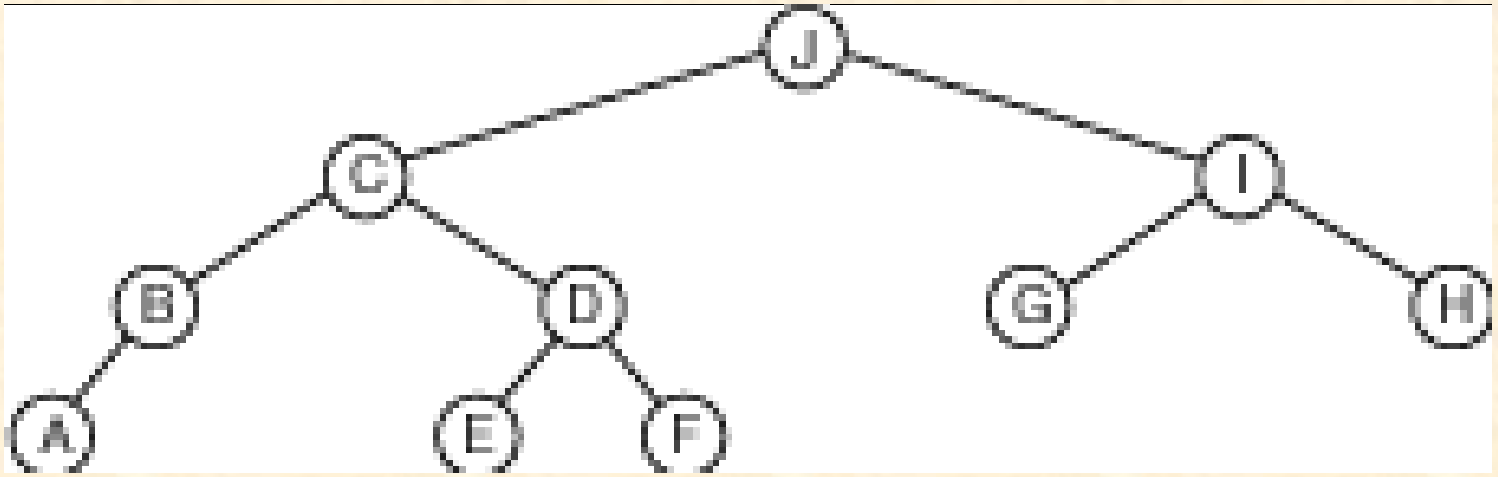
Inorder: ABCEDFJGIH

Solution: This problem is solved by setting the root of the tree to the first node in the preorder traversal and then, from the inorder traversal, identifying its left subtrees as A B C E D F and the right subtree as G I H. The process is repeated with each subtree until the complete tree is developed.

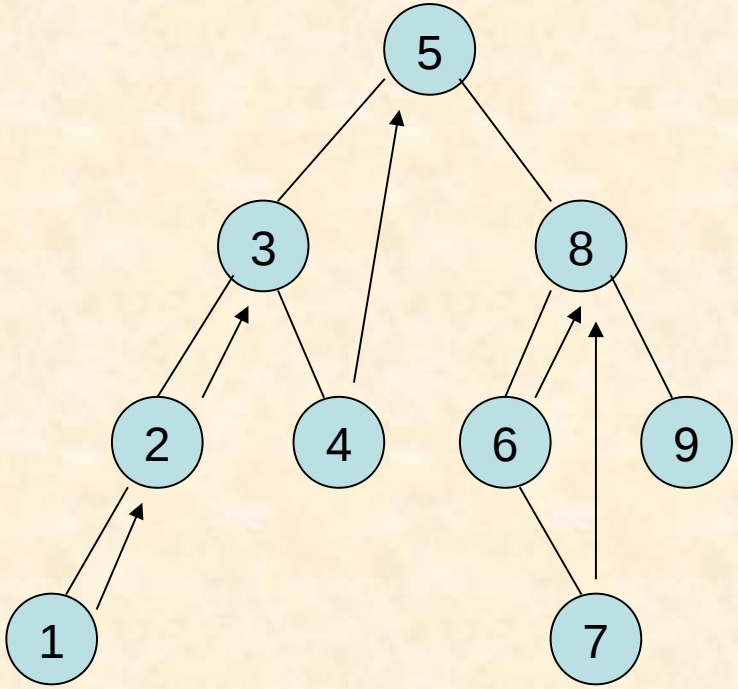
Binary Tree for Example 4.

Preorder: JCBADEFIGH

Inorder: ABCEDFJGIH



Threaded Trees



In this example, threads are the pointers.

Left nodes point to their predecessors, right nodes point to their successors.

Efficiency

- Creating threaded tree is an alternative to the recursive, or iterative use of the stack.
  - However the stack still exists, though now incorporated into the tree
- So how efficient are the different approaches?
  - All are O(n) for time.
  - But when the recursive version is so much more intuitively simple, why not use it?
- The issue with stack usage concerns space, rather than time.
  - But can we find a solution which uses less space?

Node Insertion

- Inserting a node into a tree means finding a node with a ‘dead-end’ or empty child node.
- To find the appropriate node, we can following the searching algorithm.
  - If element to be inserted (el) is greater than the root, move right, if less than the root move left.
  - If the node is empty, insert el.
- Obviously over time this could lead to a very unbalanced tree.

Deletion

- The complexity of deletion can vary depending on the node to be deleted.
- Starting simply;
  - A node with no children (i.e. a leaf).
    - The node can be deleted, and the parent will point to null.
  - A node with one child.
    - The node can be deleted, and the parent will point to the former grandchild.

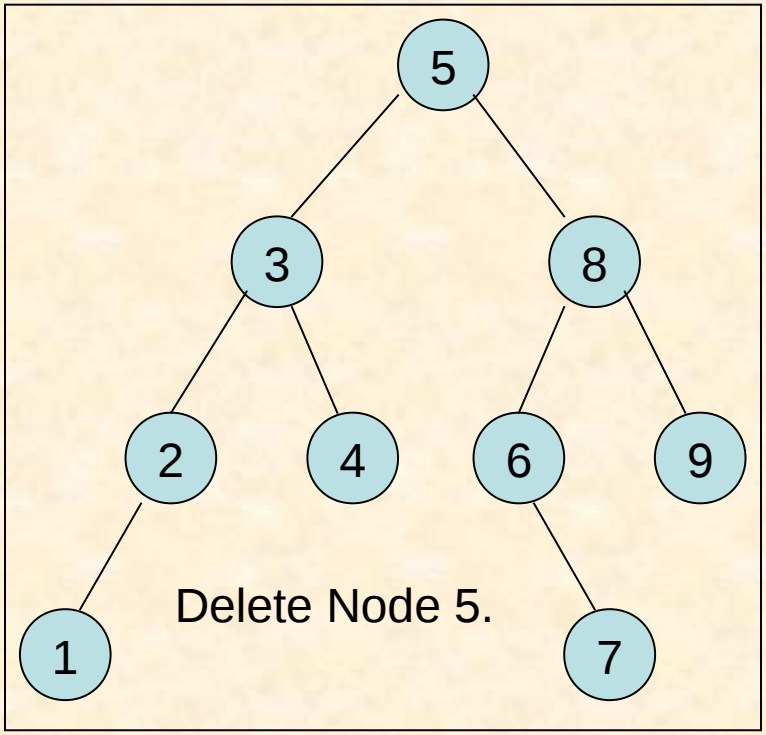
Deleting a node with 2 children

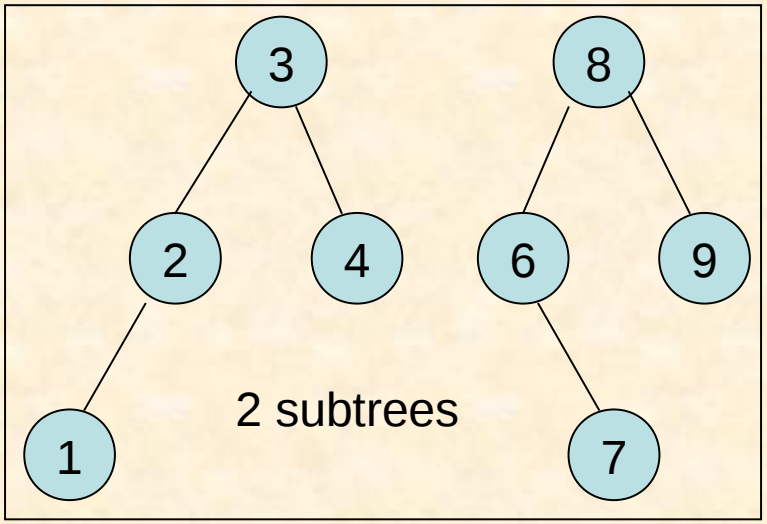
- In this case there are 2 alternatives to deletion
  - Deletion by Merging
    - The two child trees are considered as separate trees which require merging.
  - Deletion by Copying

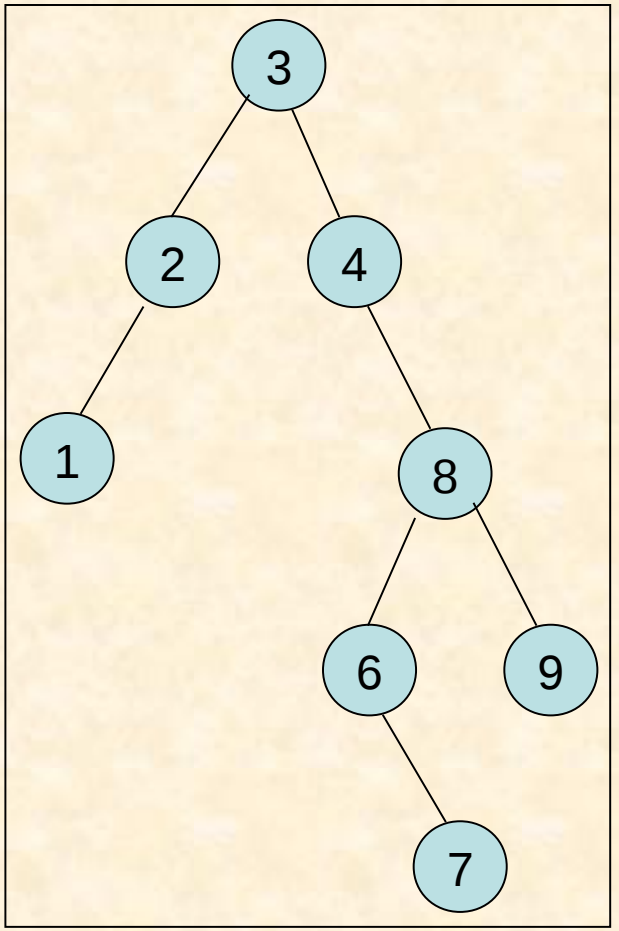
Deletion by Merging

- When a node with two children is deleted, 2 subtrees are created, with the 2 children each becoming roots.
  - All values in the left subtree are lower than all values in the right subtree.
  - In deletion by merging, the root of the left tree replaces the node being deleted.
  - The rightmost child of this tree then becomes the parent of the right tree.

Deletion by Merging







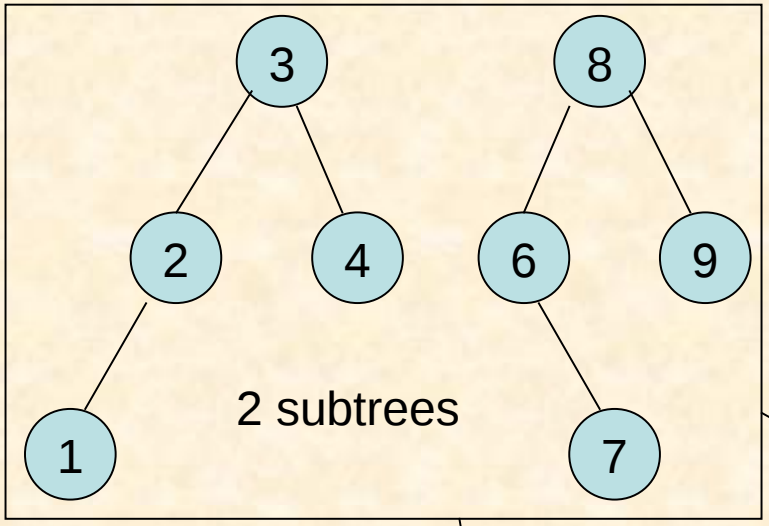
Deletion By Merging

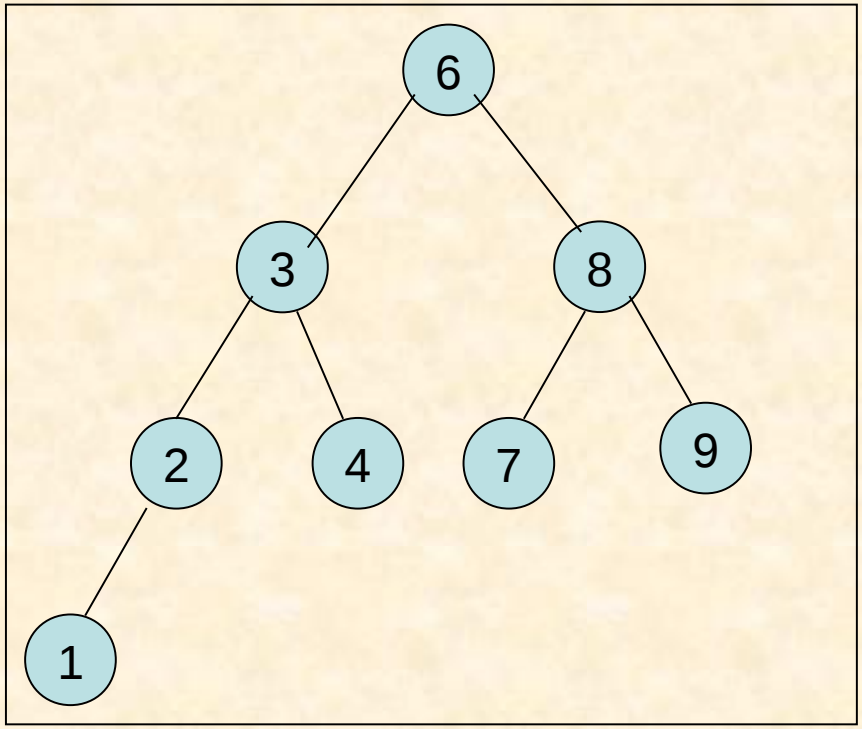
- When deleting by merging, the resulting tree may gain height, as in the previous example.
- Or it may lose height.
- The algorithm may produce a highly unbalanced tree, so while it isn’t inefficient, it isn’t perfect.

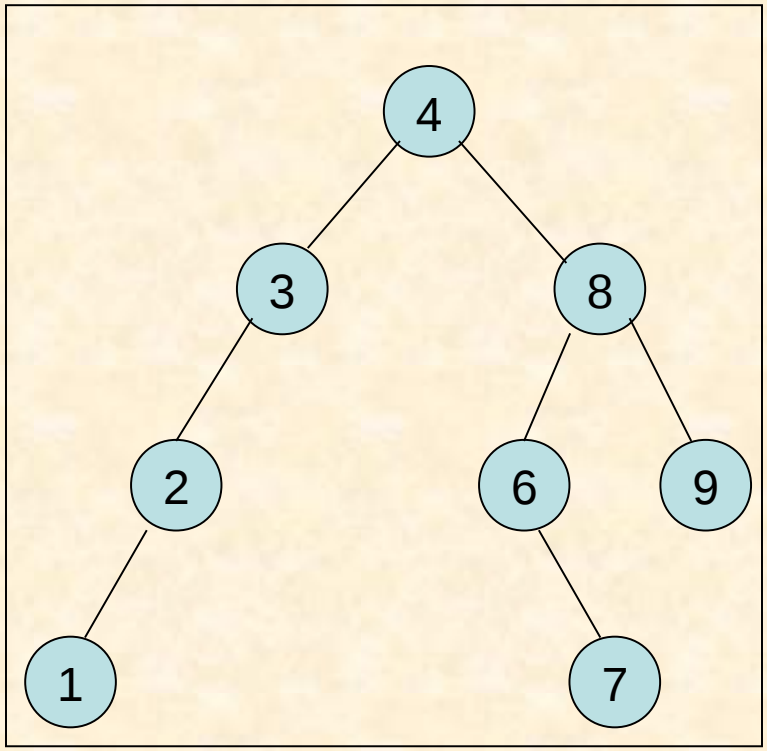
Deletion By Copying

- Again 2 subtrees are created.
  - This time, we notice that the leftmost node of the right subtree is the immediate successor of the rightmost leaf of the left subtree (and vice versa).
  - Ergo, we could replace the root of the new tree, with the rightmost leaf of the left subtree (or with the leftmost leaf of the right subtree).
  - This will not extend the height of the tree, but may lead to a bushier left side.
  - To avoid this, we can alternate between making the root the predecessor or sucessor of the deleted node.

Deletion by Copying







Creating a balanced tree

- Assume all the data is in a sorted array.
  - The middle element becomes the root.
  - The middle element of one half becomes a child.
  - The middle element of the other half becomes the other child.
  - Recurse…

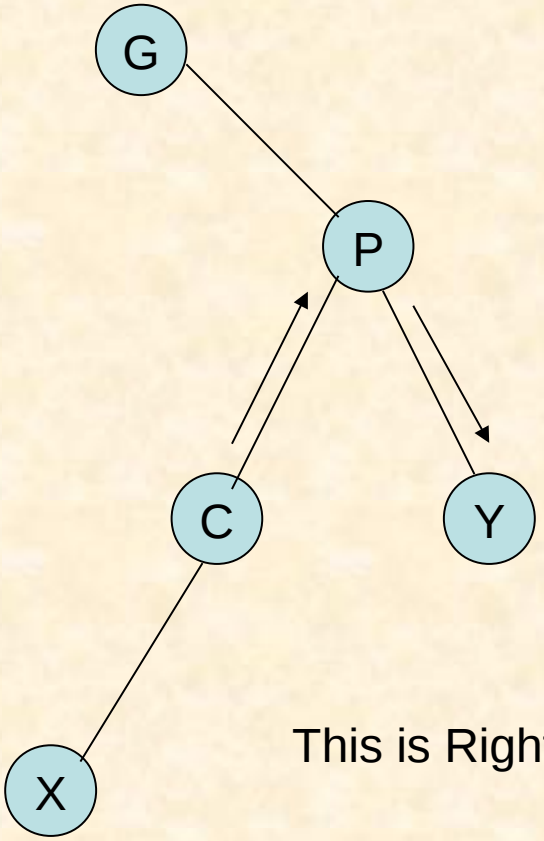
Weakness

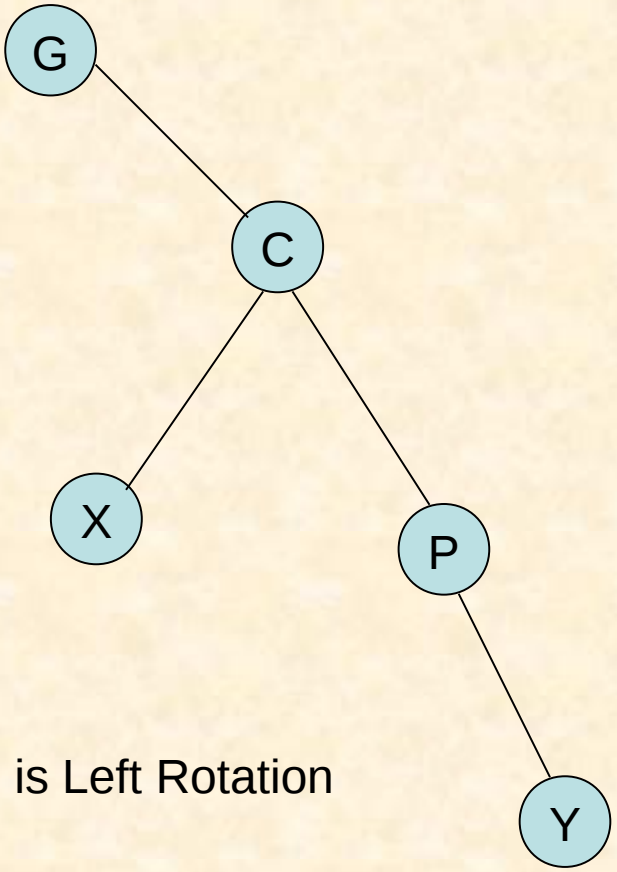
- This algorithm is quite inefficient as it relies on using extra space to store an array of values, and all values must be in this array (perhaps by an inorder traversal).
- Lets look at some alternatives.

The DSW Algorithm

- The DSW algorithm was devised by Day, and improved by Stout and Warren.
- In this algorithm first the tree is stretched into a linked list like tree.
- Then it is balanced.
- The key to the operation comes from a rotation function, where a child is rotated around its parent.

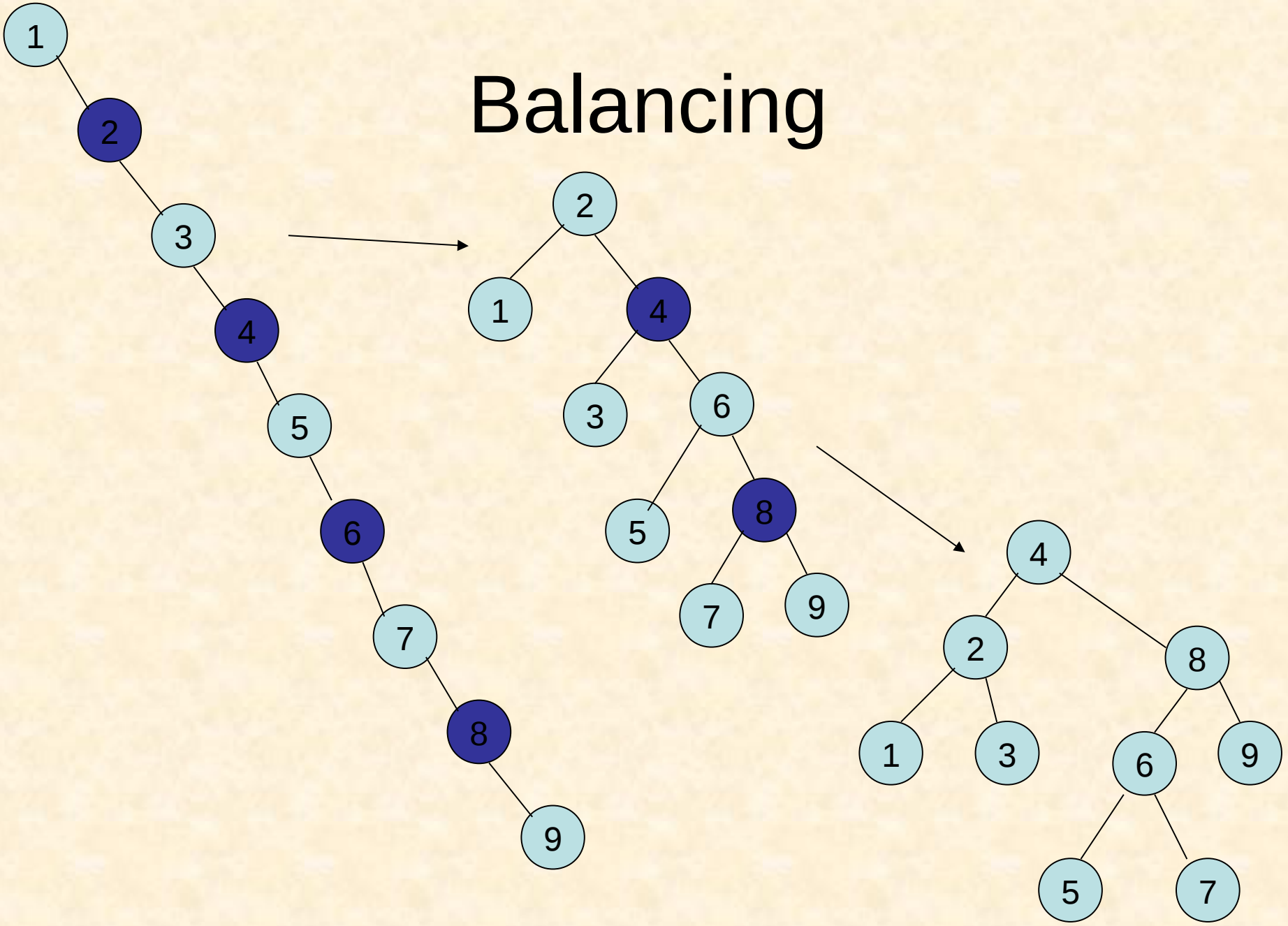
Rotation





This is Right Rotation, the reverse is Left Rotation

Balancing



DSW Algorithm

- The DSW Algorithm is effective at balancing an entire tree. (Actually O(n))
- Sometimes trees need only be balanced periodically, in which case this cost can be amortised.
- Alternatively the tree may only become unstable after a series of insertions and deletions, in which case a DSW balancing may be appropriate.

AVL Trees

Adelson-Velskii and Landis

- An alternative approach is to ensure the tree remains balanced by incorporating balancing into any insertion / deletion algorithms.
  - In an AVL insertion and deletion considers the structure of the tree.
    - All nodes have a balance factor – i.e. a difference between the number of nodes in the right subtree and the left subtree. This difference must remain at +1, 0 or -1.
    - An AVL tree may not appear completely balanced as after the DSW algorithm.

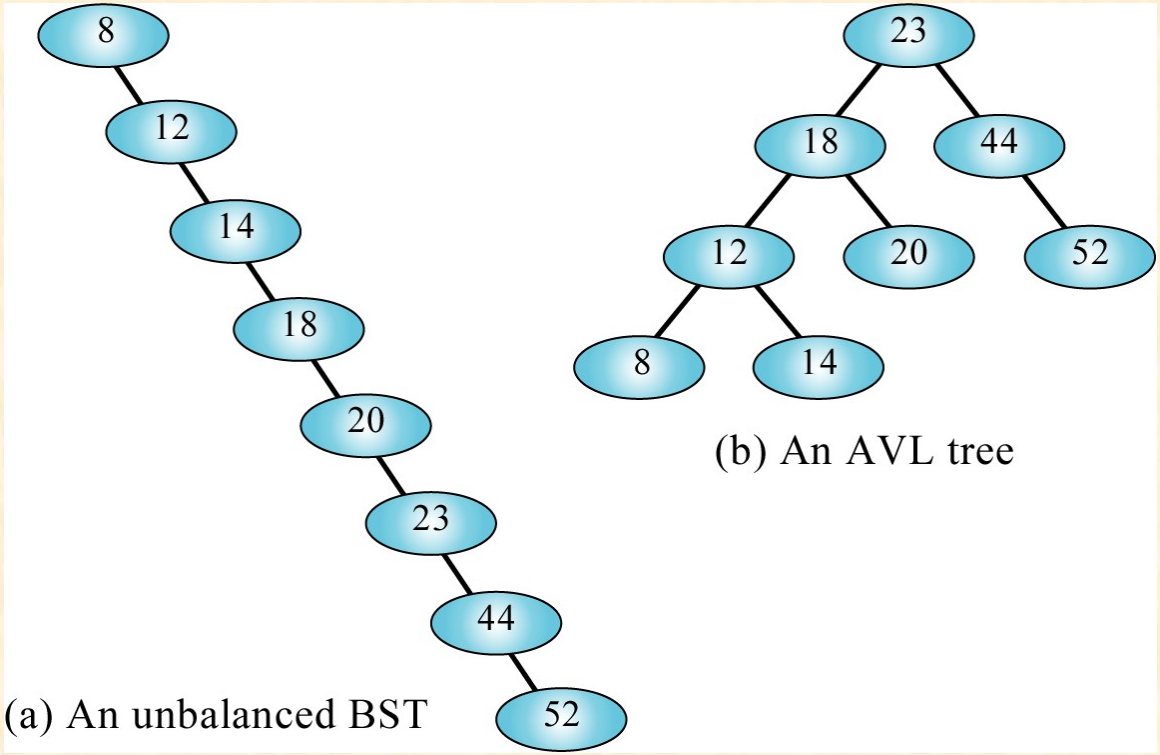
AVL Tree

- method that can be used to balance binary search tree proposed by Adel'son-Vel'skii and Landis
- AVL tree was originally called *admissible tree*
  - one in which the height of the left and right subtrees of every node differ by at most one
  - balanced factor should be +1, 0, or -1

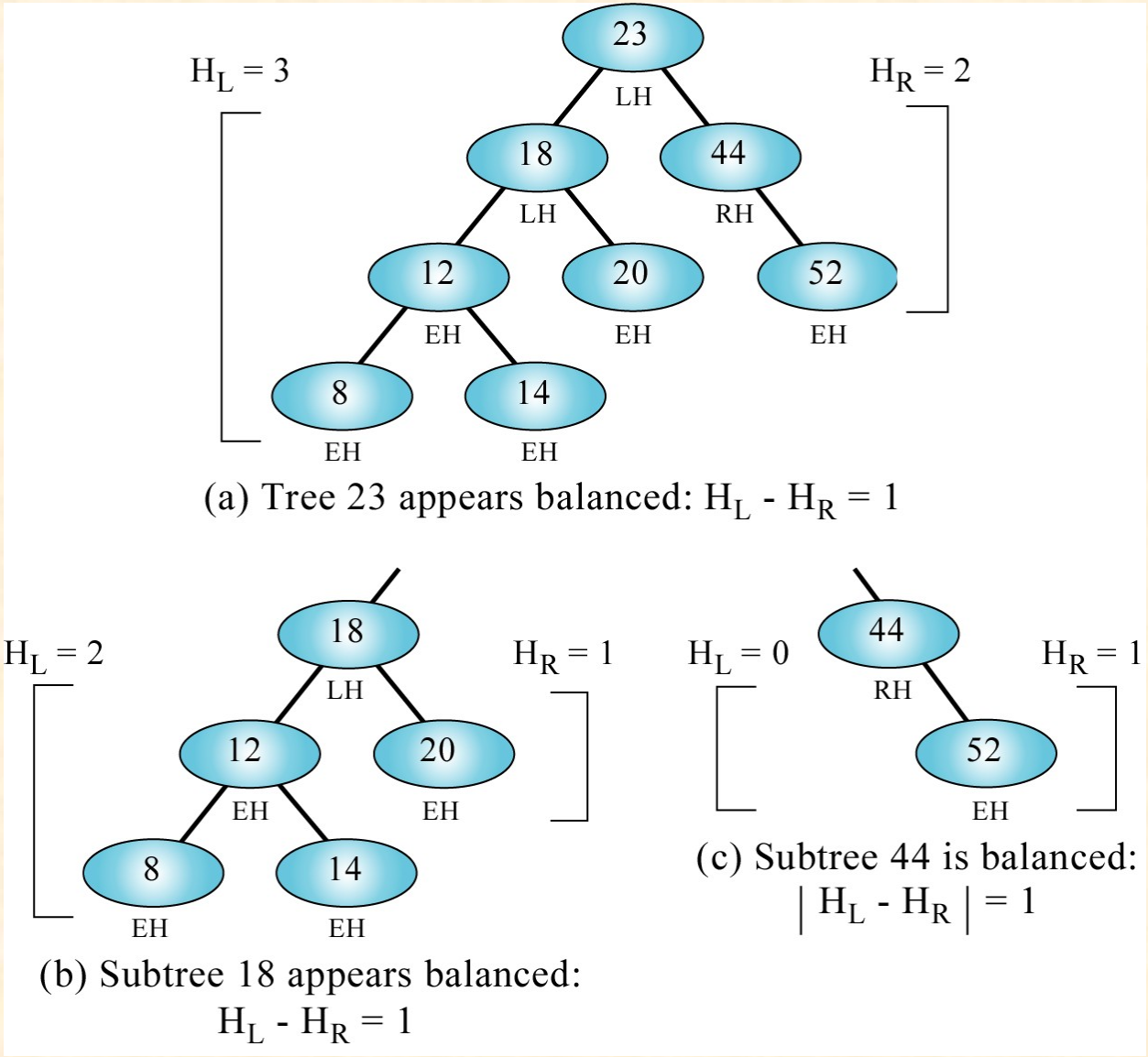
$$B = H_L - H_R$$

  - balanced factor can be 0 or 1

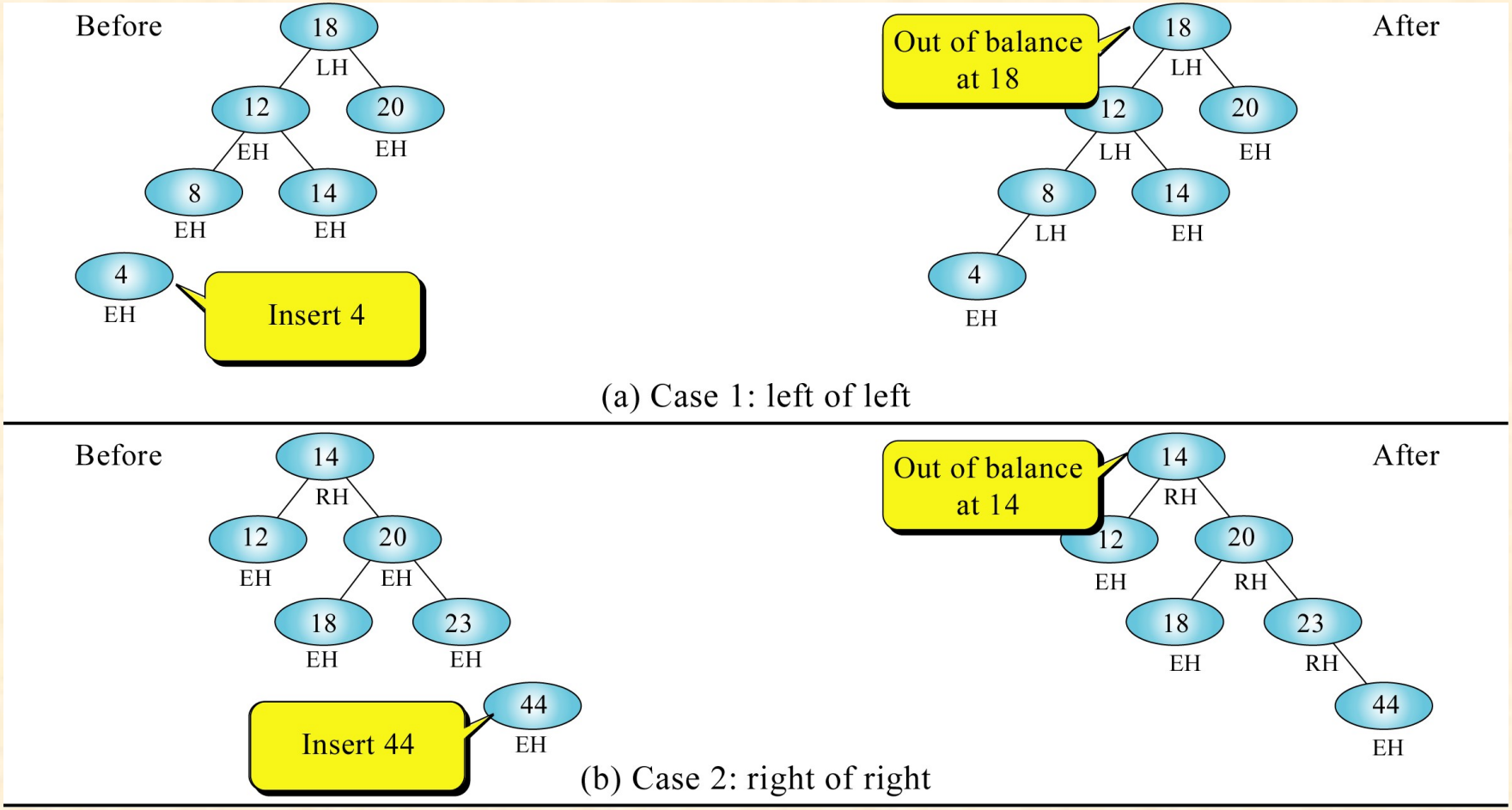
$$B = |H_L - H_R|$$


8
12
14
18
20
23
44
52
(a) An unbalanced BST
23
18
44
12
20
52
8
14
(b) An AVL tree

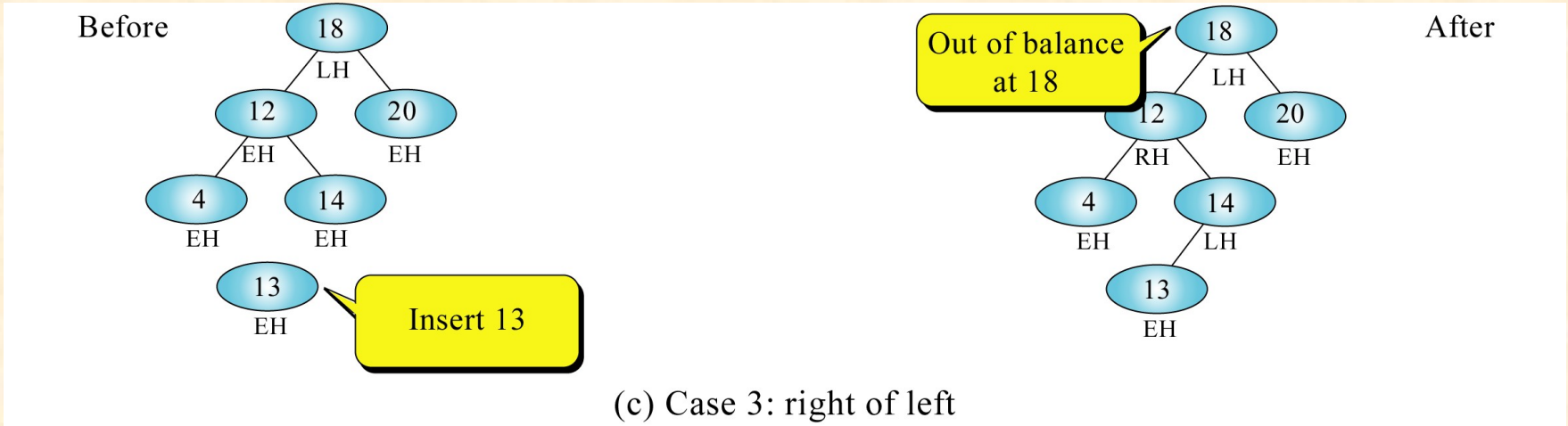
A Balanced BST



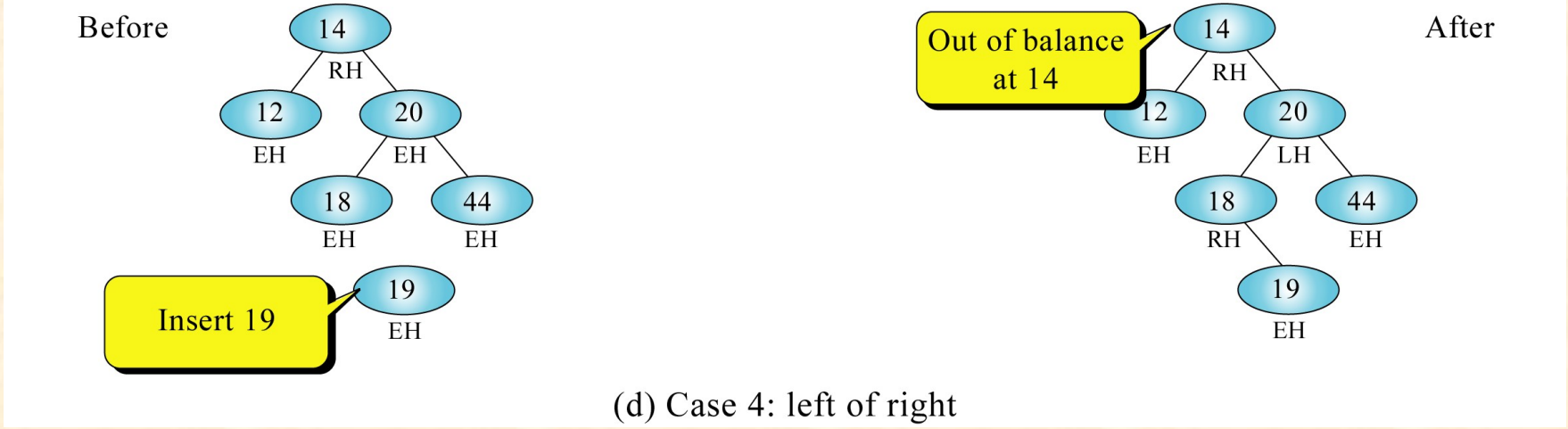
Unbalanced Cases



Unbalanced Cases

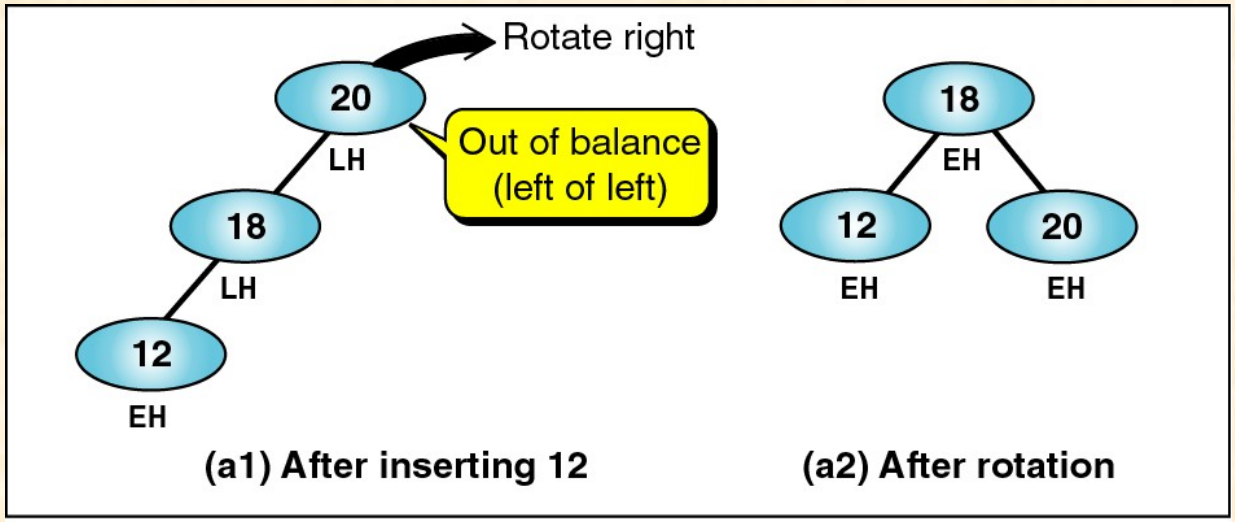


(c) Case 3: right of left

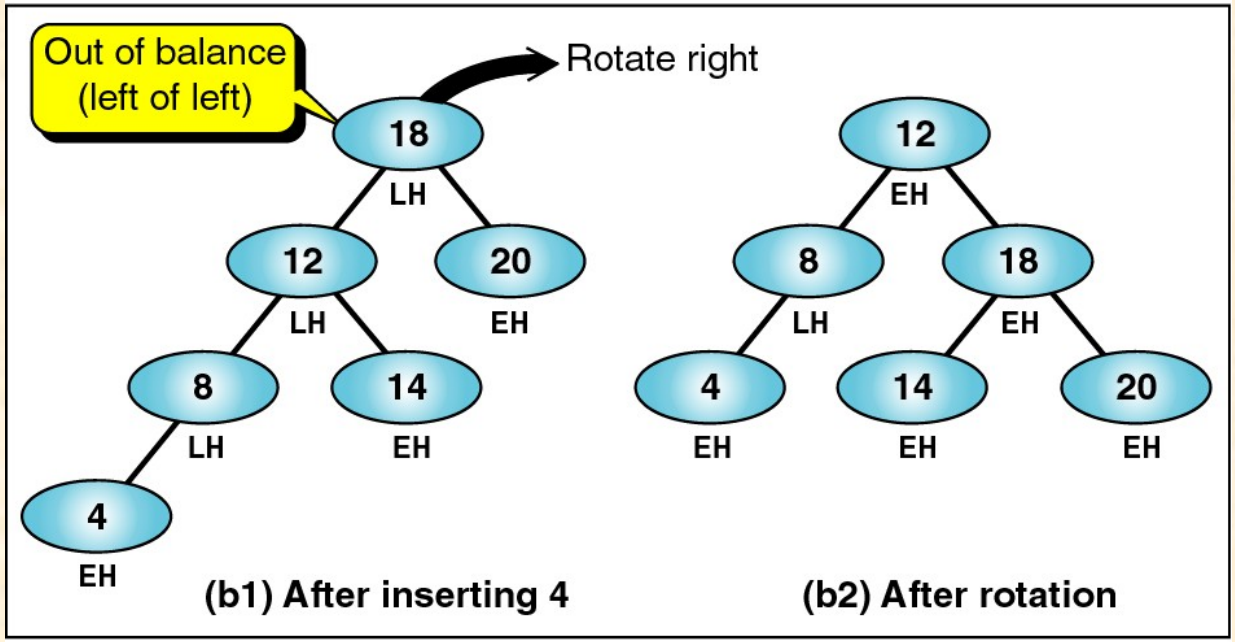


(d) Case 4: left of right

Balancing by Single Rotation

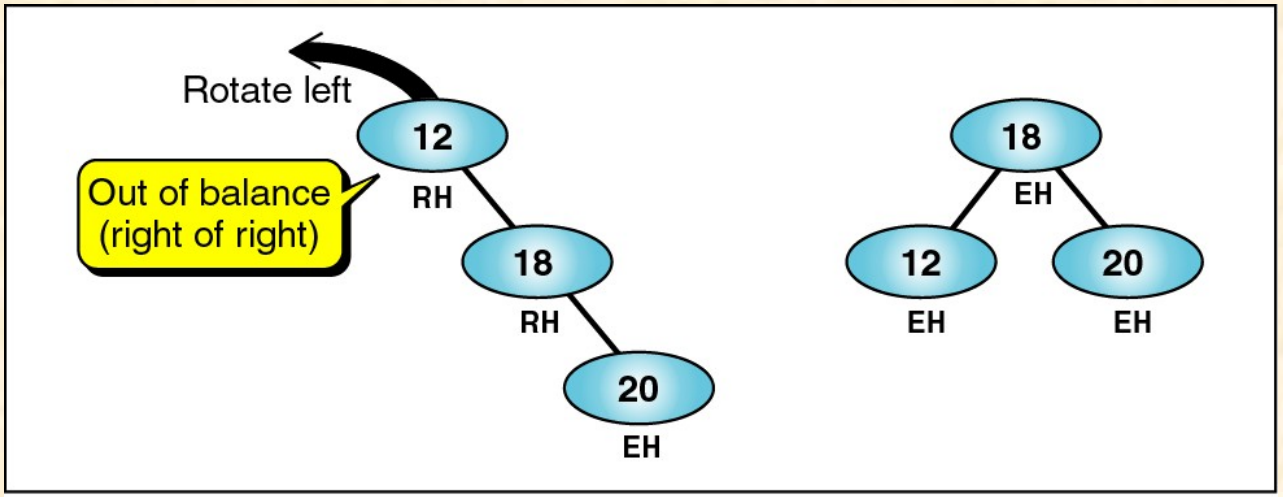


(a) Simple right rotation

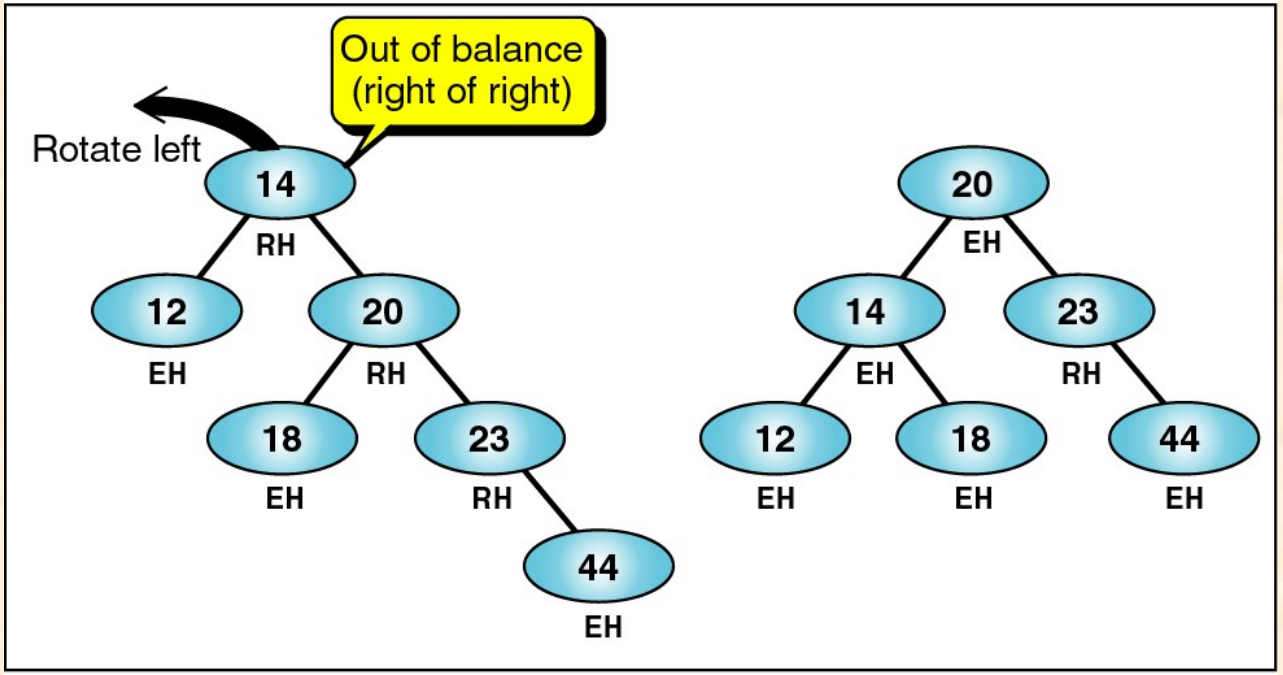


(b) Complex right rotation

Balancing by Single Rotation

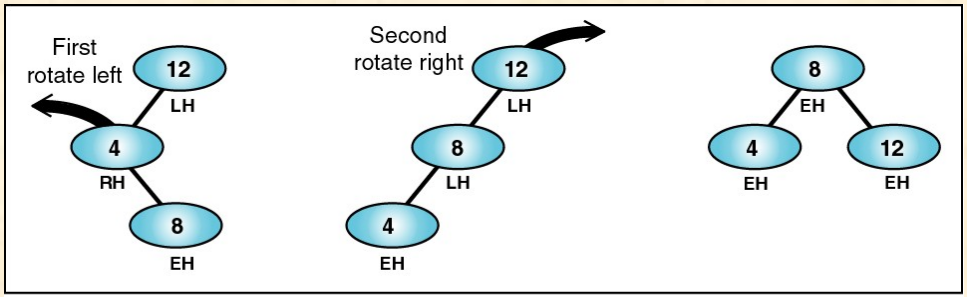


(a) Simple left rotation

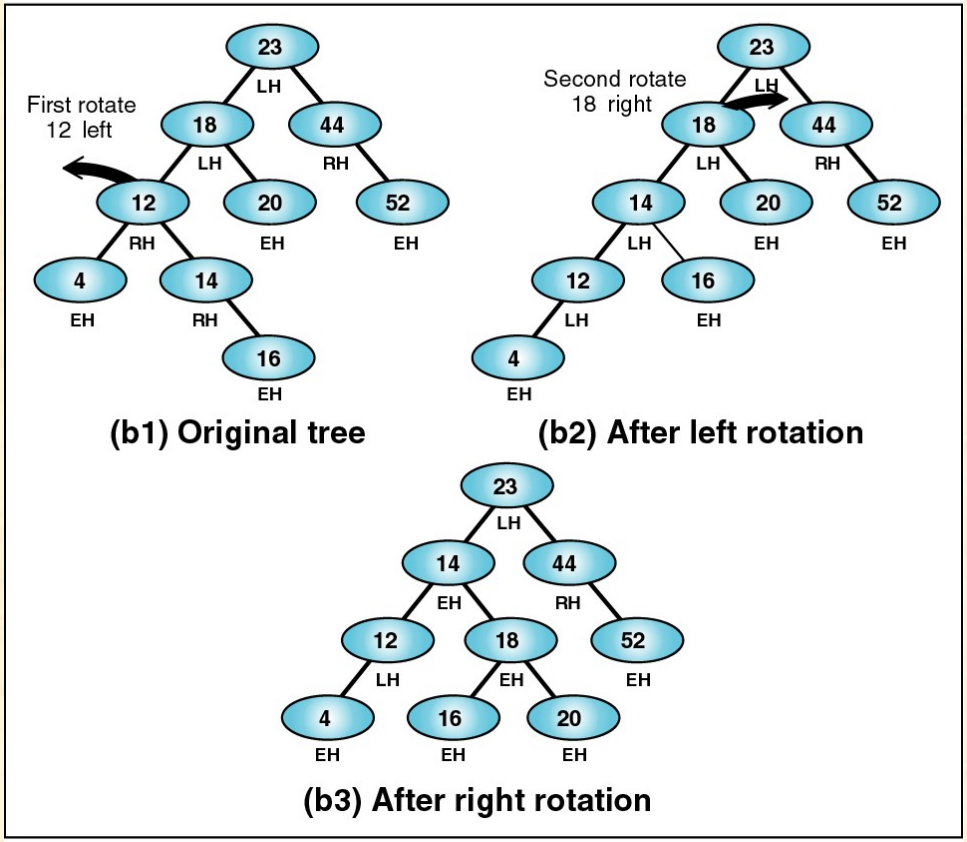


(b) Complex left rotation

Balancing by Double Rotation



(a) Simple double rotation right



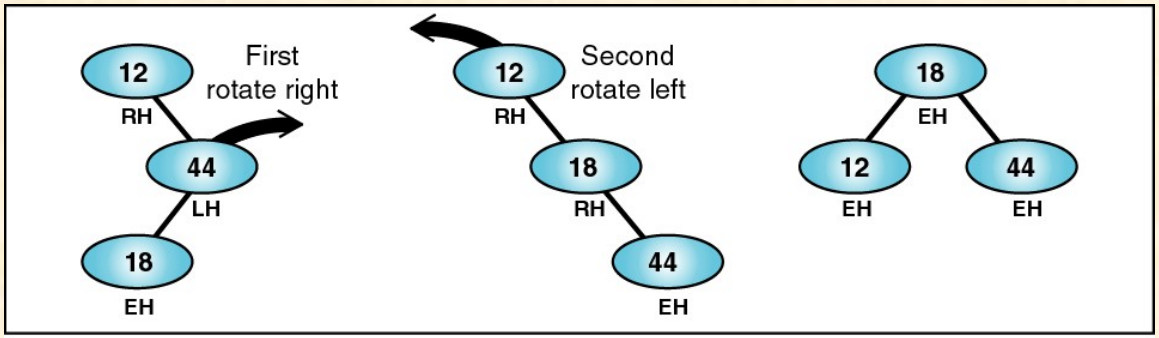
(b1) Original tree

(b2) After left rotation

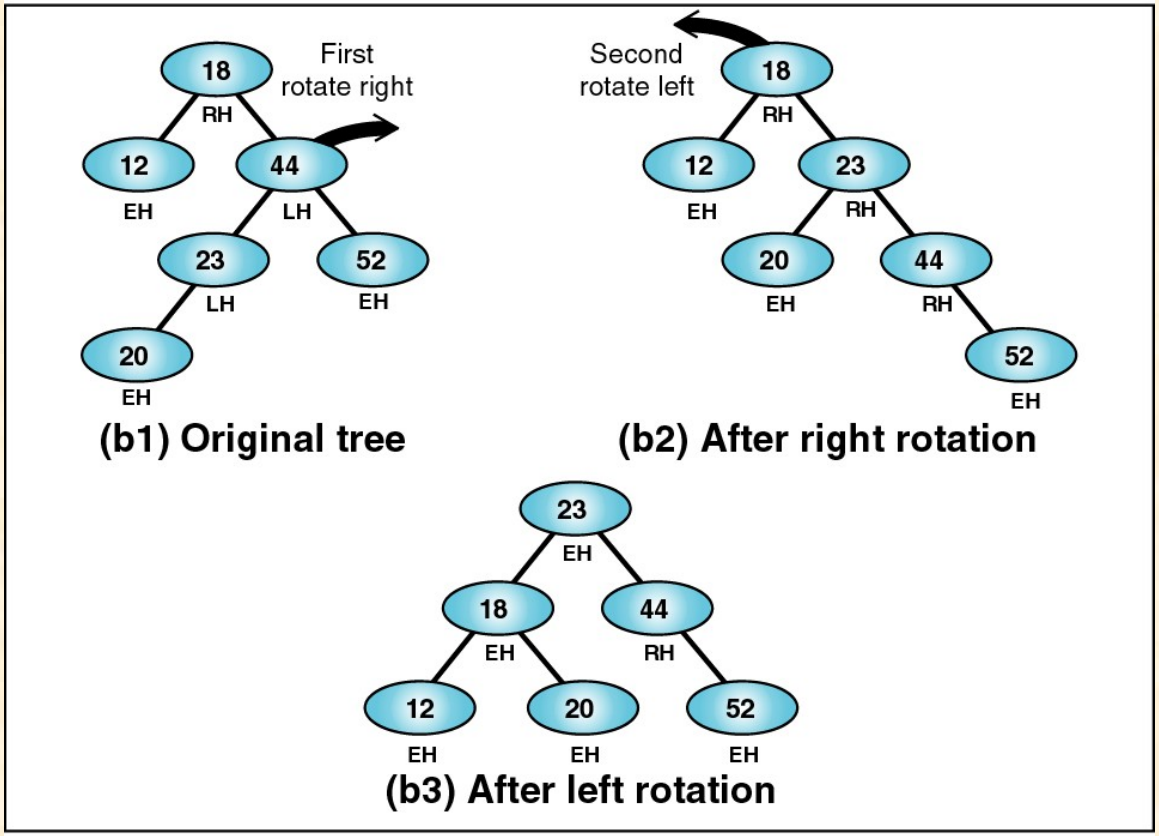
(b3) After right rotation

(b) Complex double rotation right

Balancing by Double Rotation



(a) Simple double rotation right



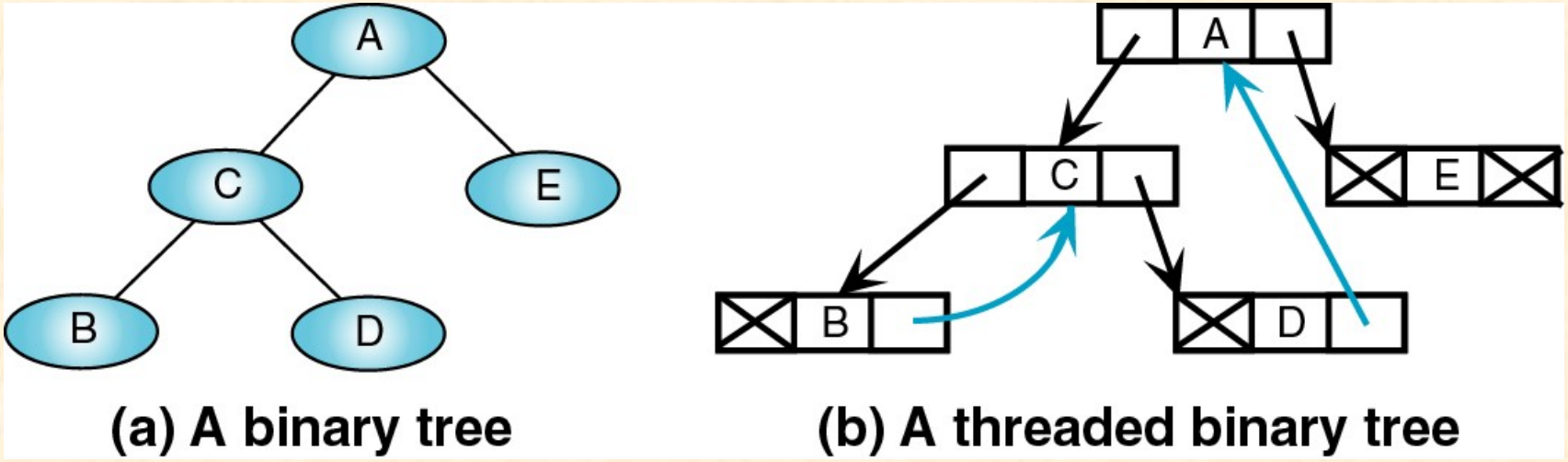
(b1) Original tree

(b2) After right rotation

(b3) After left rotation

(b) Complex double rotation right

Converting a BST to a threaded tree



Self Adjusting Trees

- We have encountered some self adjustment strategies already, when dealing with linked lists. For Binary Trees self adjustment stems from an observation similar to one we encountered there.
- Balancing a Binary Tree has focused on building a complete binary tree, with a view that this is the most efficient when searching for any node in it.
- However, some nodes may be searched more frequently than others – so if these nodes appear near the top of a binary tree, searches become more efficient.
- Strategies for self restructuring trees include; *Single Rotation* (rotating the node around its parent), or *Move to Root*.

Expression Trees

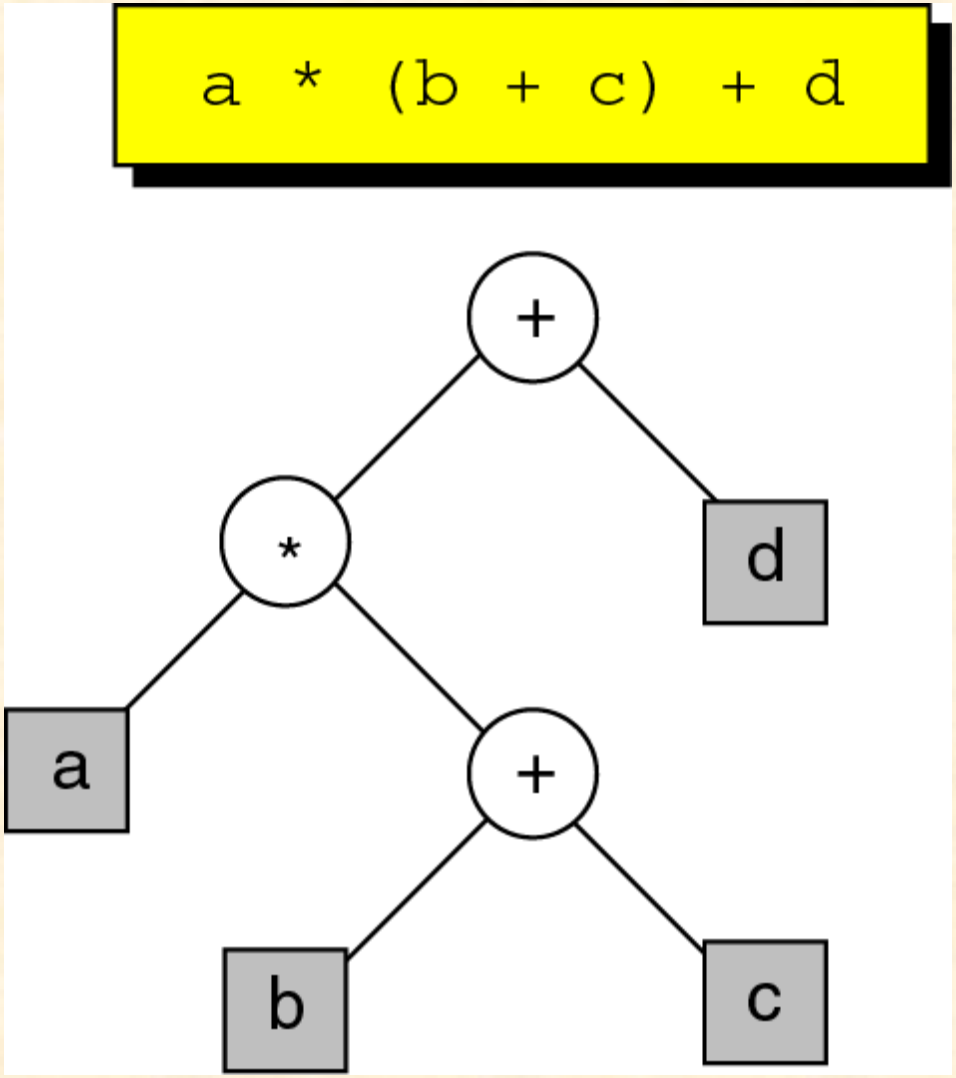
Expression tree – is a binary tree with the following properties:

1. Each leaf is an operand
2. The root and internal nodes are operators
3. The subtrees are subexpressions, with the root being an operand

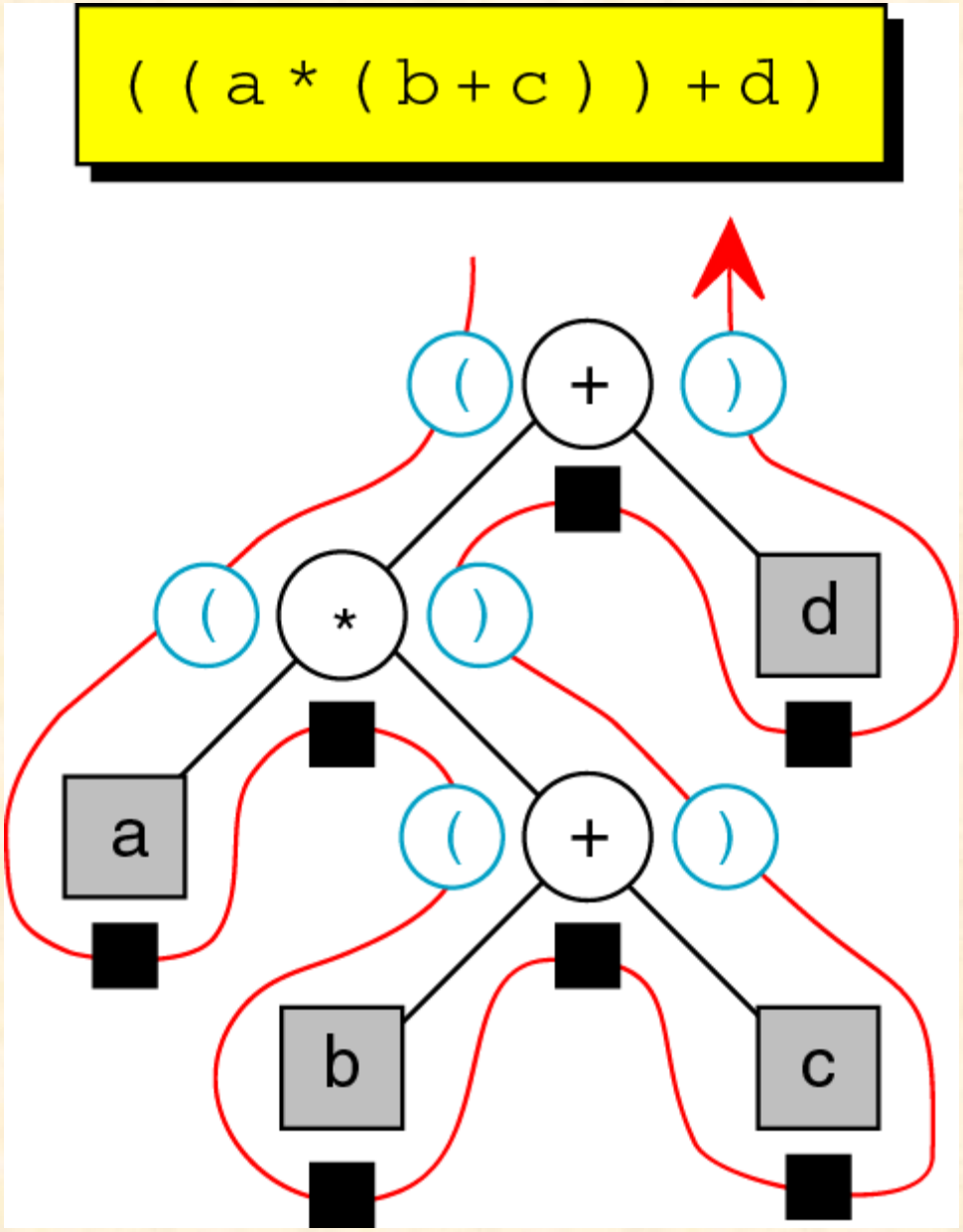
Expression Tree Traversals

1. Infix Traversal – add an opening parenthesis at the left of the operator node and a closing parenthesis at the right, then traverse the tree by inorder traversal

An infix expression and its expression tree



Infix Traversal



Example 1. Find the infix, prefix, and postfix expressions in the expression tree.

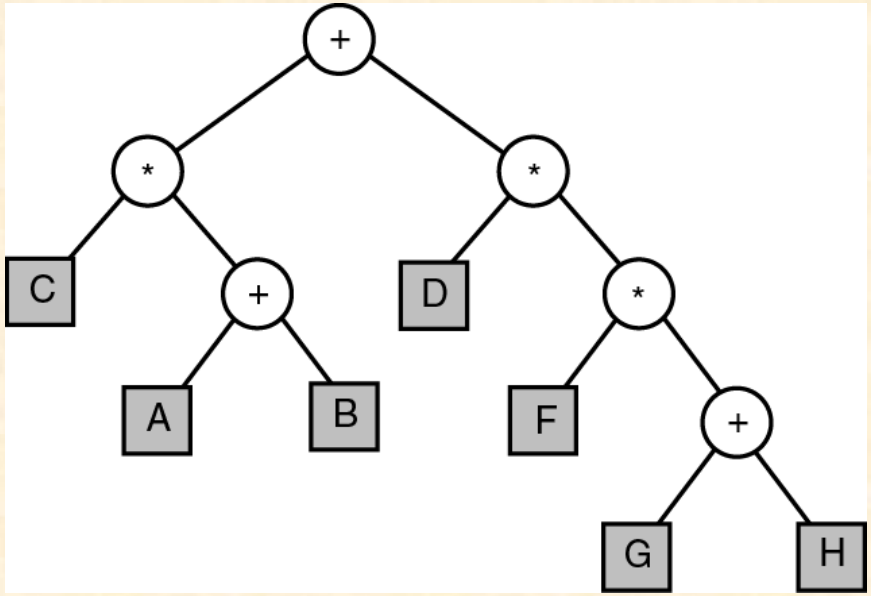
Answers

Infix: ((C * (A + B)) + (D * (F * (G + H))))

C * (A + B) + D * (F * (G + H))

Prefix: + * C + A B * D * F + G H

Postfix: C A B + * D F G H + * * +



Expression Tree Traversals

2. Postfix Traversal – of an expression uses the basic postorder traversal of any binary tree

   - it does not require the parentheses

3. Prefix Traversal – of an expression uses the standard preorder tree traversal

   - it does not require the parenthees

Example 2. Draw the expression tree and find the prefix and postfix expressions for the following infix expression:

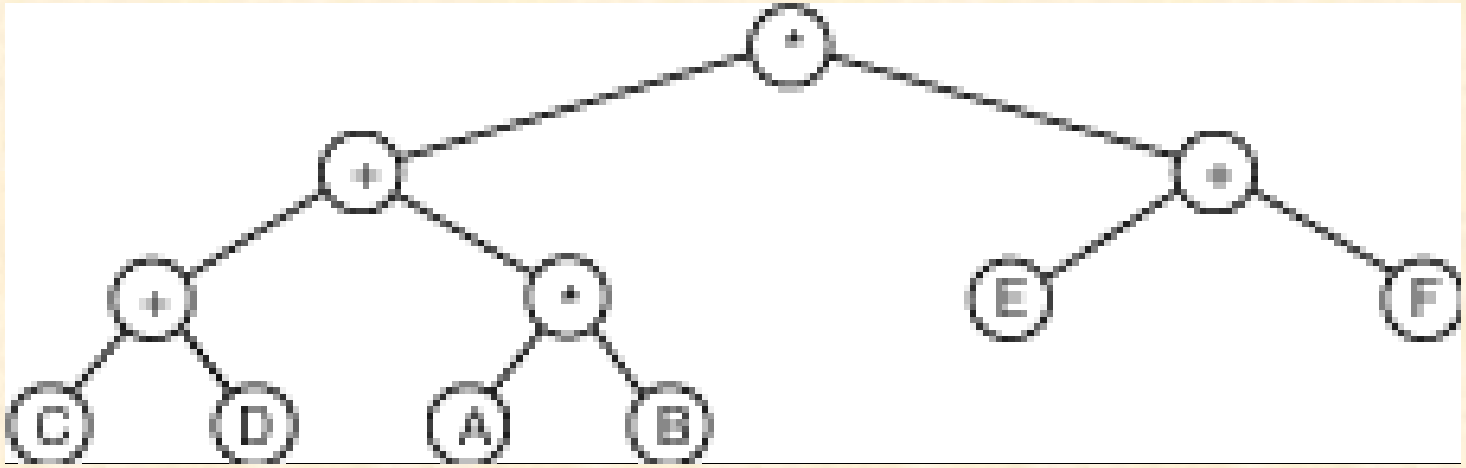
(C + D + A * B) * (E + F)

Prefix: *** + + C D * A B + E F**

Postfix: **C D + A B * + E F + ***

Example 2. Draw the expression tree and find the prefix and postfix expressions for the following infix expression:

(C + D + A * B) * (E + F)



Prefix: *** + + C D * A B + E F**

Postfix: **C D + A B * + E F + ***