# COMP2003
# Data Structures and Algorithms

## Lecture 12: Binary Trees, Binary Search Trees

# Trees

- Linear access time of linked lists is prohibitive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations

# Trees

- A tree is a collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *subtrees* $T_1$, $T_2$, ...., $T_k$, each of whose roots are connected by a directed *edge* from r

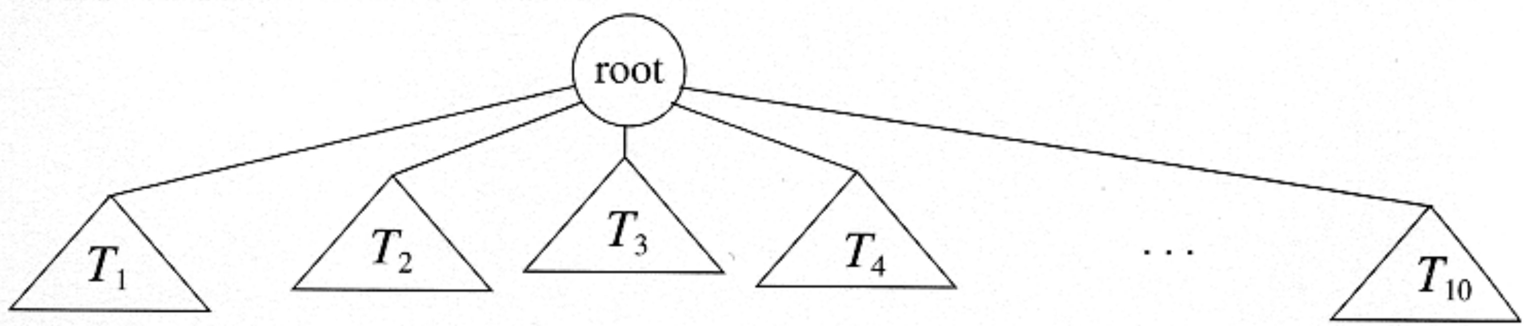



**Figure 4.1** Generic tree

# Some Terminologies

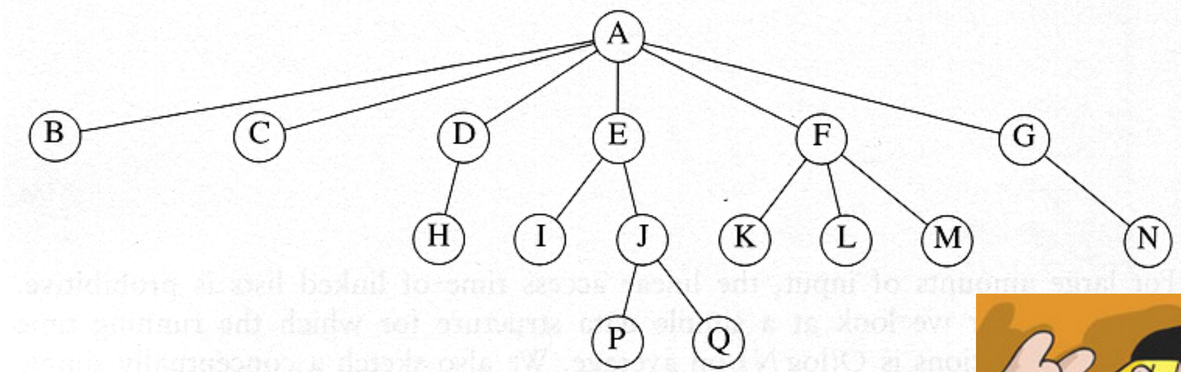



**Figure 4.2** A tree

- *Child* and *Parent*
  - Every node except the root has one parent
  - A node can have an zero or more children
- *Leaves*
  - Leaves are nodes with no children
- *Sibling*
  - nodes with same parent

# More Terminologies

- *Path*
  - A sequence of edges
- *Length of a path*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node
- *Height* of a node
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- The height of a tree = the height of the root
  = the depth of the deepest leaf
- *Ancestor* and *descendant*
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1
  - *Proper ancestor* and *proper descendant*

# Example: UNIX Directory

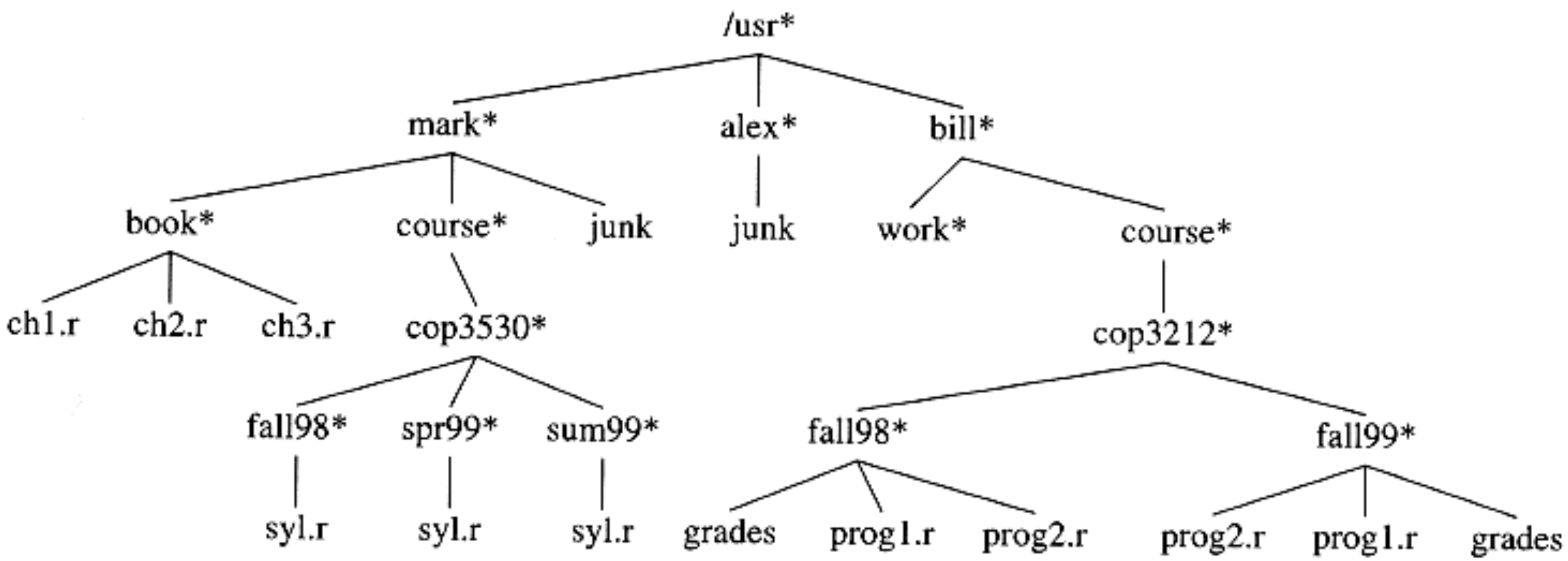



**Figure 4.5** UNIX directory

# Example: Expression Trees

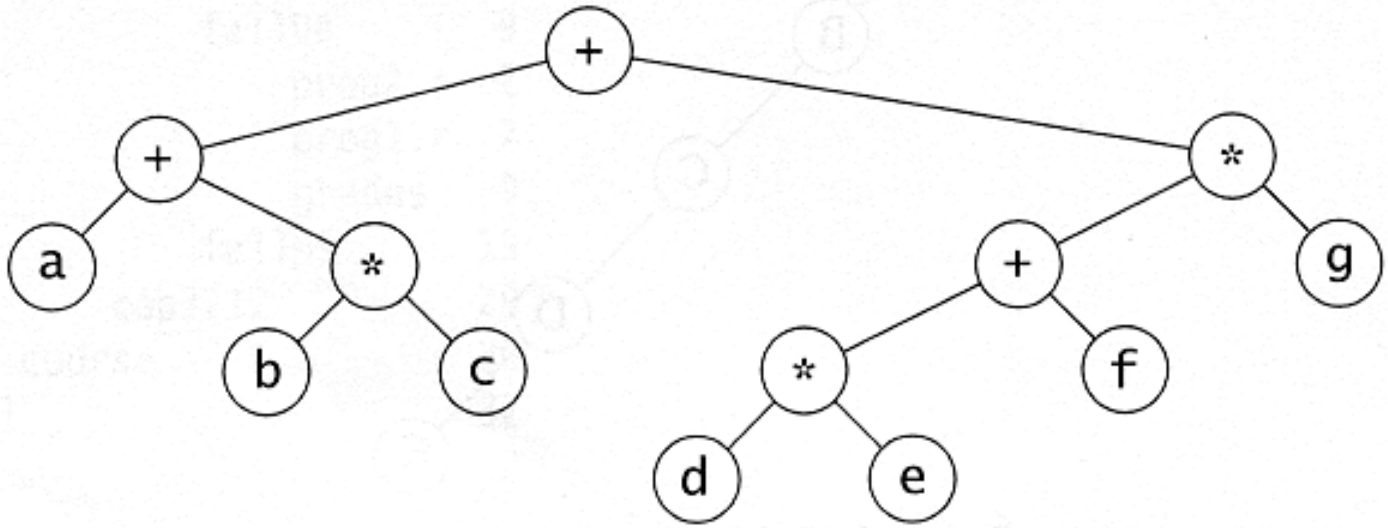
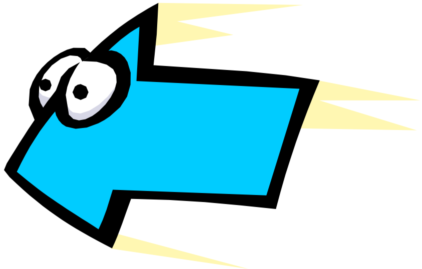
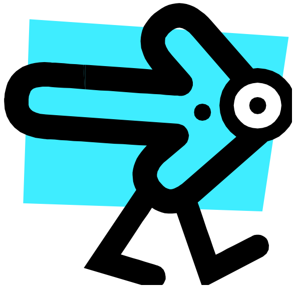



**Figure 4.14** Expression tree for (a + b * c) + ((d * e + f ) * g)

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary

# Tree Traversal

- Used to print out the data in a tree in a certain order
- Pre-order traversal
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree

# Preorder, Postorder and Inorder

- Preorder traversal
  - node, left, right
  - prefix expression
    - ++a*bc*+*defg

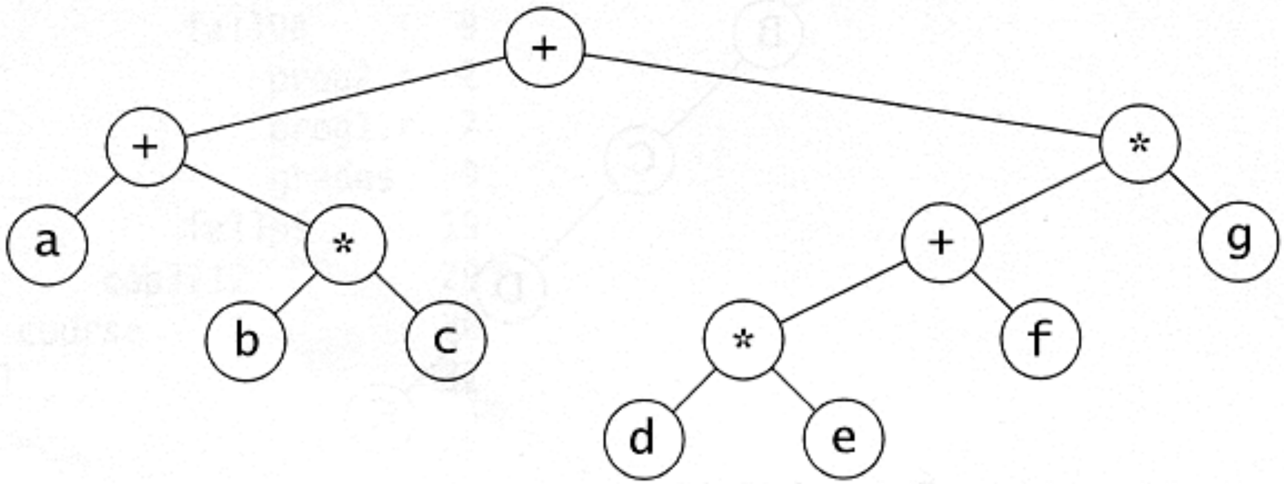



**Figure 4.14** Expression tree for (a + b * c) + ((d * e + f ) * g)

# Preorder, Postorder and Inorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+
- Inorder traversal
  - left, node, right
  - infix expression
    - a+b*c+d*e+f*g

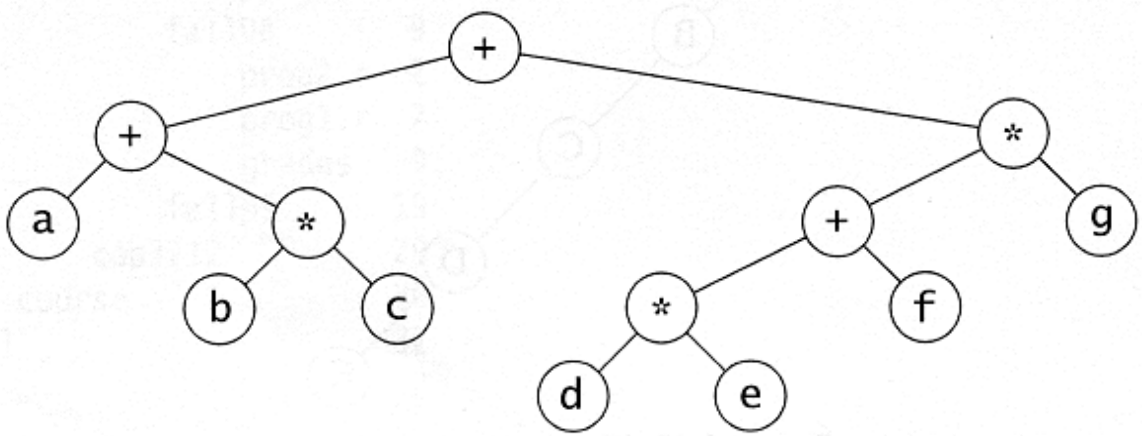



**Figure 4.14** Expression tree for (a + b * c) + ((d * e + f ) * g)

# Example: Unix Directory Traversal

## PreOrder

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall98
                    syl.r
                spr99
                    syl.r
                sum99
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall98
                    grades
                    prog1.r
                    prog2.r
                fall99
                    prog2.r
                    prog1.r
                    grades
```

## PostOrder

```
            ch1.r              3
            ch2.r              2
            ch3.r              4
        book                  10
                    syl.r      1
                fall98         2
                    syl.r      5
                spr99          6
                    syl.r      2
                sum99          3
            cop3530           12
        course                13
        junk                   6
    mark                      30
        junk                   8
    alex                       9
        work                   1
                    grades     3
                    prog1.r    4
                    prog2.r    1
                fall98         9
                    prog2.r    2
                    prog1.r    7
                    grades     9
                fall99        19
            cop3212           29
        course                30
    bill                      32
/usr                          72
```

# Preorder, Postorder and Inorder Pseudo Code

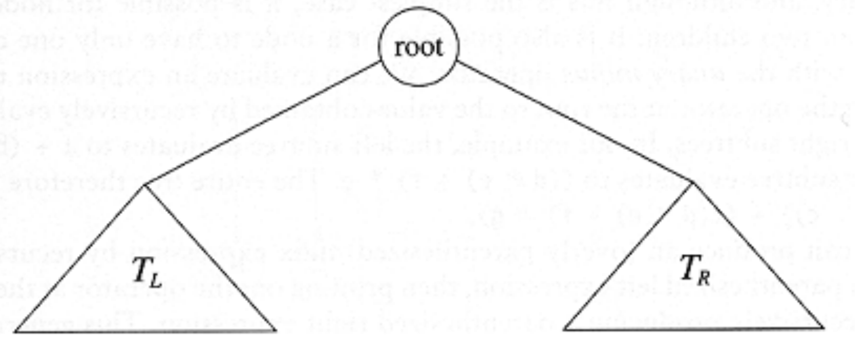
**Algorithm** *Preorder*($x$)
**Input:** $x$ is the root of a subtree.
  **if** $x \neq$ NULL
    **then** output key($x$);
        *Preorder*(left($x$));
        *Preorder*(right($x$));

**Algorithm** *Postorder*($x$)
**Input:** $x$ is the root of a subtree.
  **if** $x \neq$ NULL
    **then** *Postorder*(left($x$));
        *Postorder*(right($x$));
        output key($x$);

**Algorithm** *Inorder*($x$)
**Input:** $x$ is the root of a subtree.
  **if** $x \neq$ NULL
    **then** *Inorder*(left($x$));
        output key($x$);
        *Inorder*(right($x$));

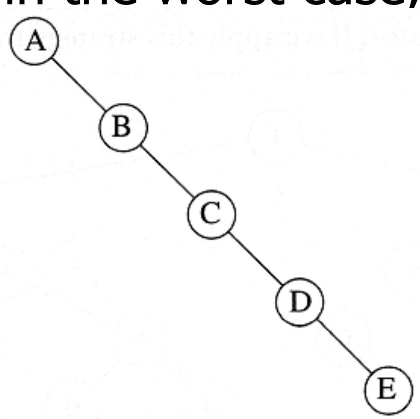
# Binary Trees

- A tree in which no node can have more than two children




**Generic binary tree**

- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.




**Worst-case binary tree**

# Node Struct of Binary Tree

- Possible operations on the Binary Tree ADT
  - Parent, left_child, right_child, sibling, root, etc
- Implementation
  - Because a binary tree has at most two children, we can keep direct pointers to them

```
struct BinaryNode
{
    Object      element;        // The data in the node
    BinaryNode *left;           // Left child
    BinaryNode *right;          // Right child
};
```

# Convert a Generic Tree to a Binary Tree

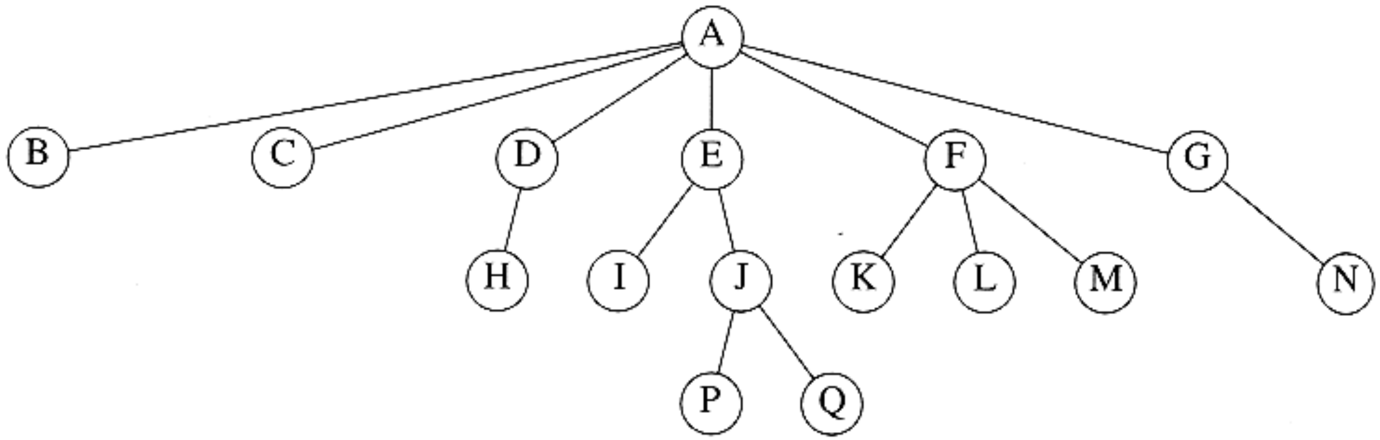



**Figure 4.2** A tree

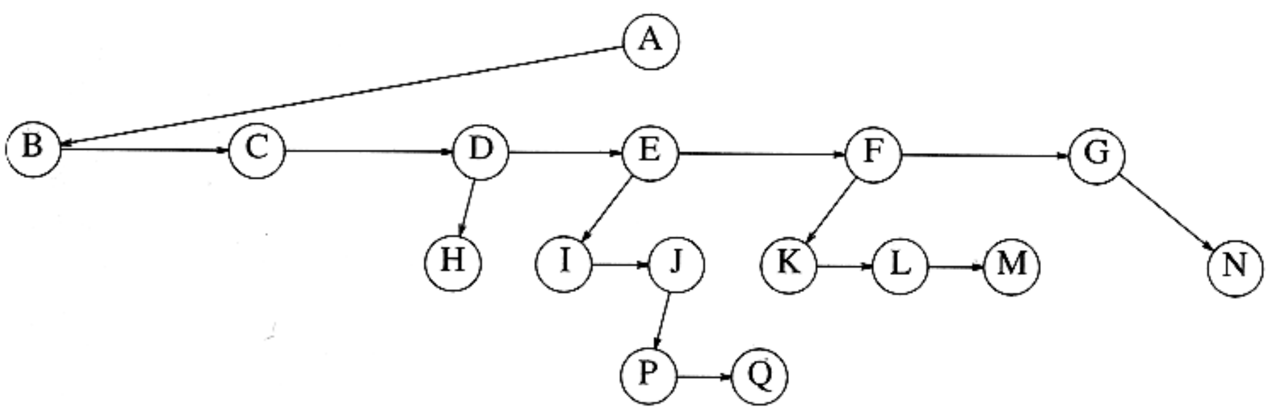
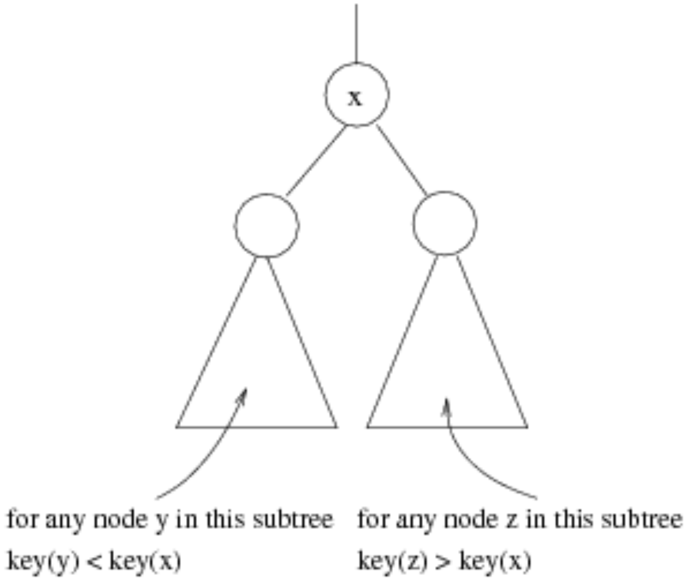


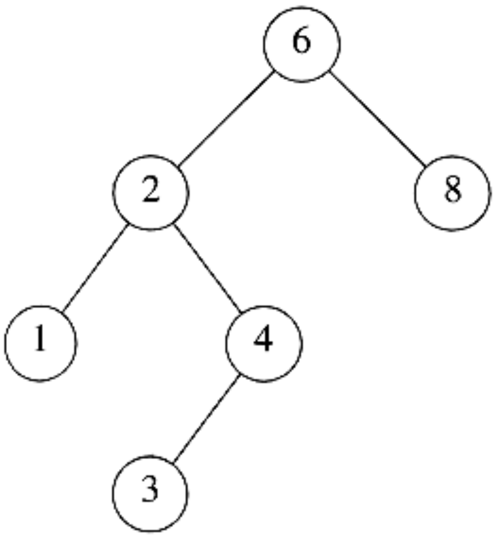

**Figure 4.4** First child/next sibling representation of the tree shown in Figure 4.2

# Binary Search Trees (BST)

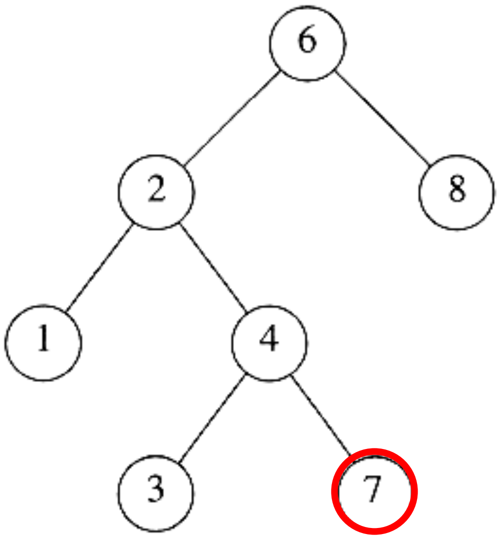
- A data structure for efficient searching, insertion and deletion
- Binary search tree property
  - For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X



# Binary Search Trees



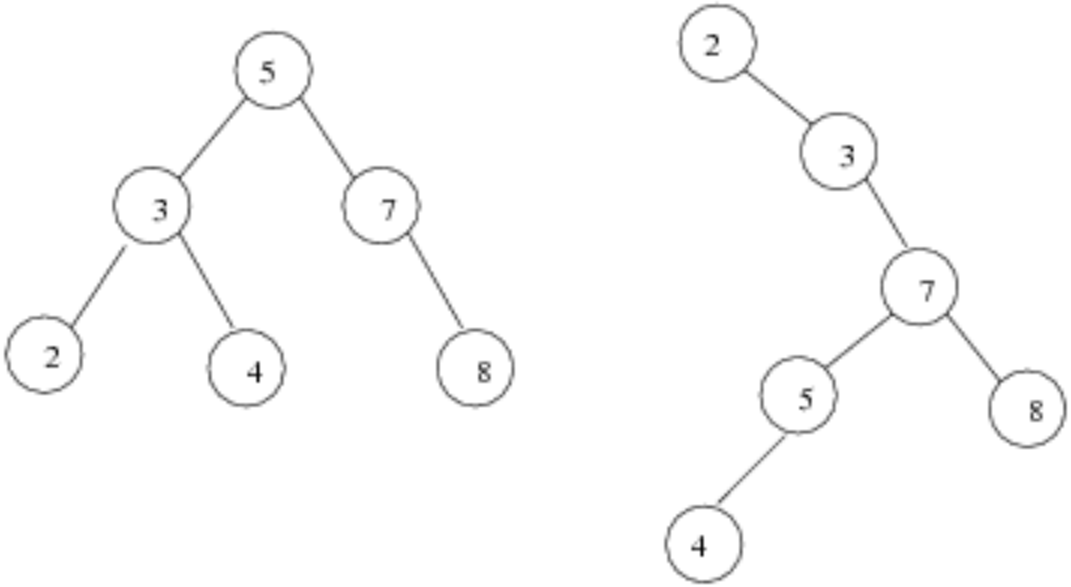

**A binary search tree**




**Not a binary search tree**

# Binary Search Trees

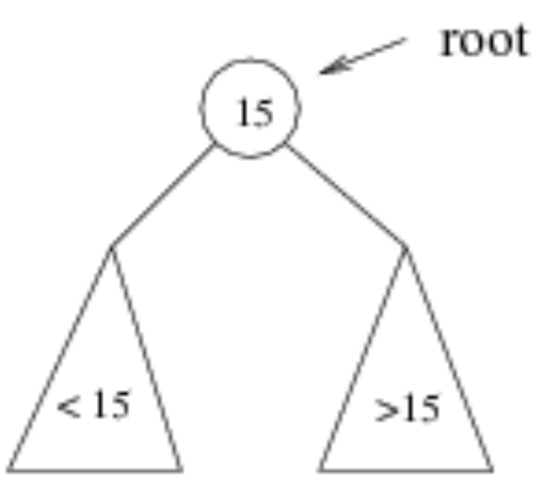
**The same set of keys may have different BSTs**



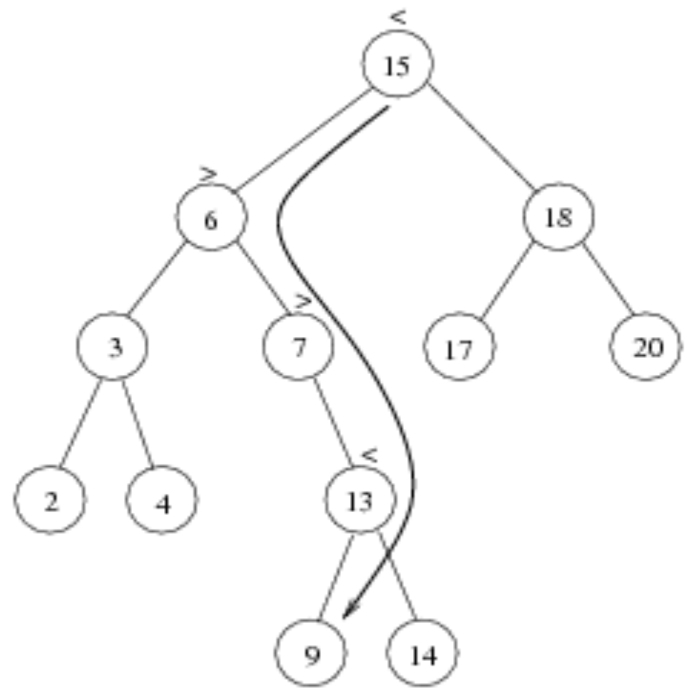

- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

# Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



*Example:* Search for 9 ...




Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

# Searching (Find)

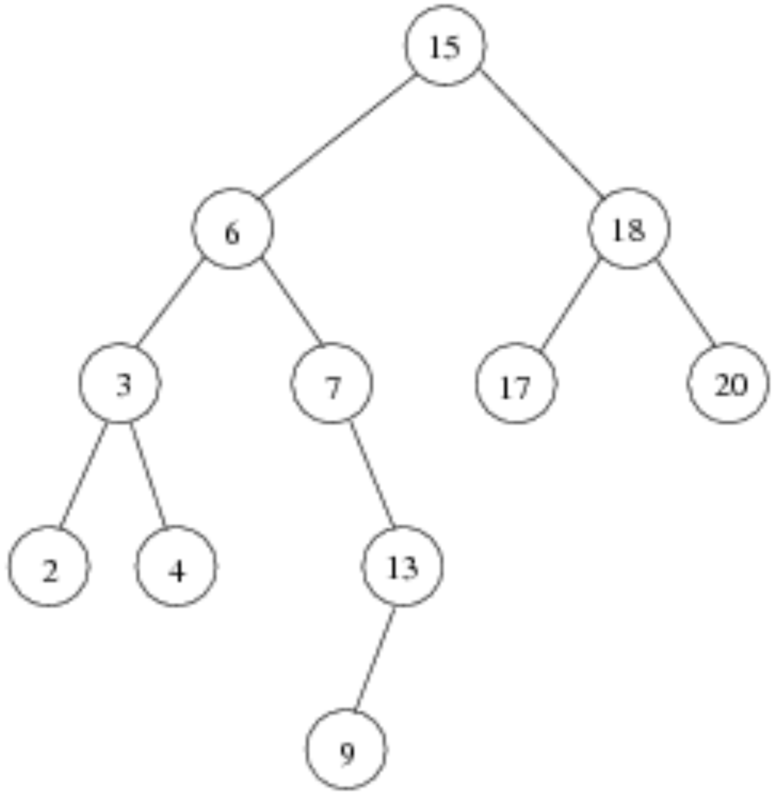
- Find X: return a pointer to the node that has key X, or NULL if there is no such node

```
BinaryNode * BinarySearchTree::Find(const float &x, BinaryNode *t) const
{
    if (t == NULL)
        return NULL;
    else if (x < t->element)
            return Find(x, t->left);
        else if (t->element < x)
                return Find(x, t->right);
            else
                return t;      // match
}
```

- Time complexity: O(height of the tree)

# Inorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order




**Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20**

# findMin/ findMax

- Goal: return the node containing the smallest (largest) key in the tree
- Algorithm: Start at the root and go left (right) as long as there is a left (right) child. The stopping point is the smallest (largest) element

```
BinaryNode * BinarySearchTree::FindMin(BinaryNode *t) const
{
    if (t == NULL)
        return NULL;
    if (t->left == NULL)
        return t;
    return FindMin(t->left);
}
```

# Binary Tree Height

- Given a binary tree, find its maximum depth.
- The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
- **Note:** A leaf is a node with no children.

- **Example:**

Given binary tree [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

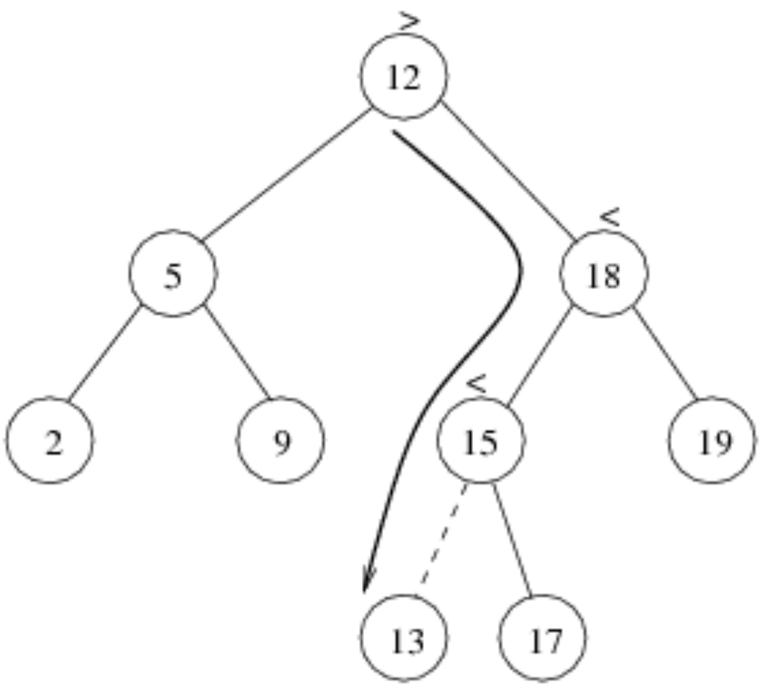
return its depth = 3.

# Binary Tree Height

```
int maxDepth(BinaryNode* root) {
    if(root == NULL) return 0;
    return max(maxDepth(root->left), maxDepth(root->right)) + 1;
}
```

# Insertion

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



# Insertion

```
node* insert(int x, node* t) {
    if(t == NULL) {
        t = new node;
        t->data = x;
        t->left = t->right = NULL;
    }
    else if(x < t->data)
        t->left = insert(x, t->left);
    else if(x > t->data)
        t->right = insert(x, t->right);
    return t;
}
```

# Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

# Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

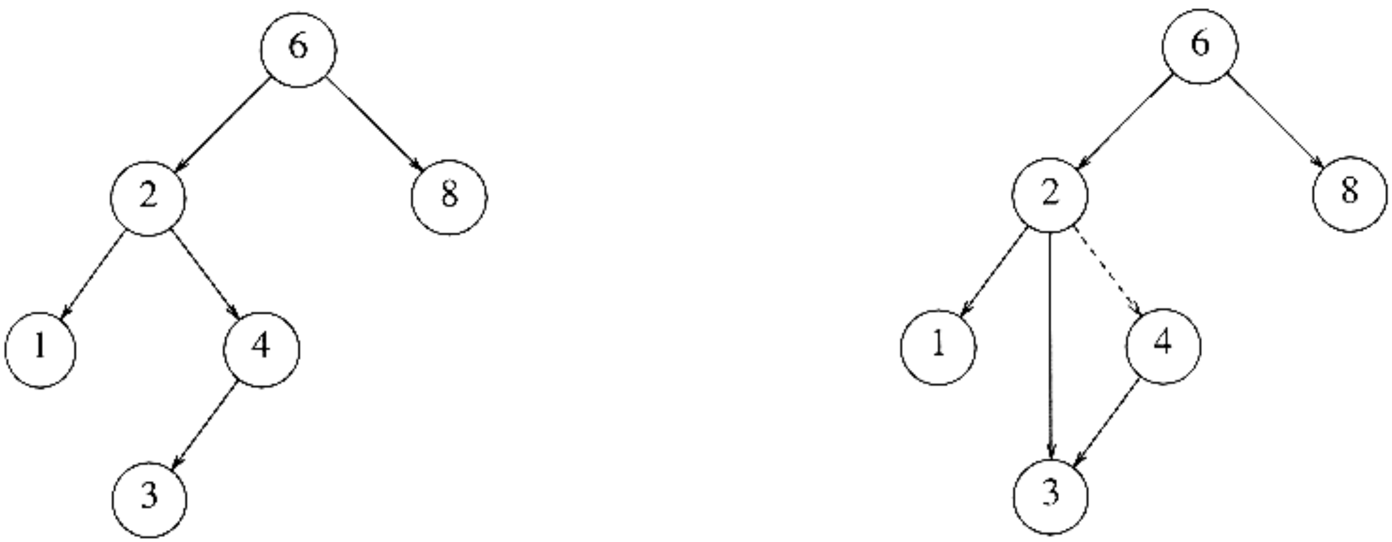



**Figure 4.24** Deletion of a node (4) with one child, before and after

# Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

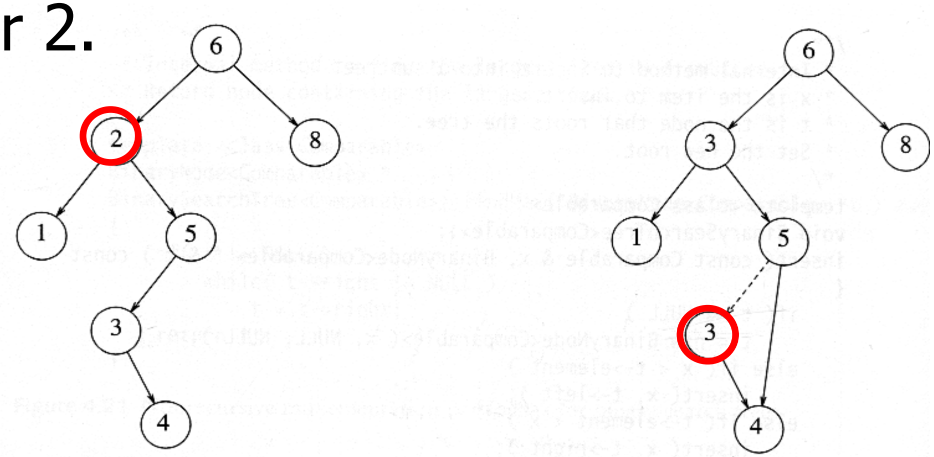



**Figure 4.25** Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)