CISC 235 Topic 3

General Trees, Binary Trees,

Binary Search Trees

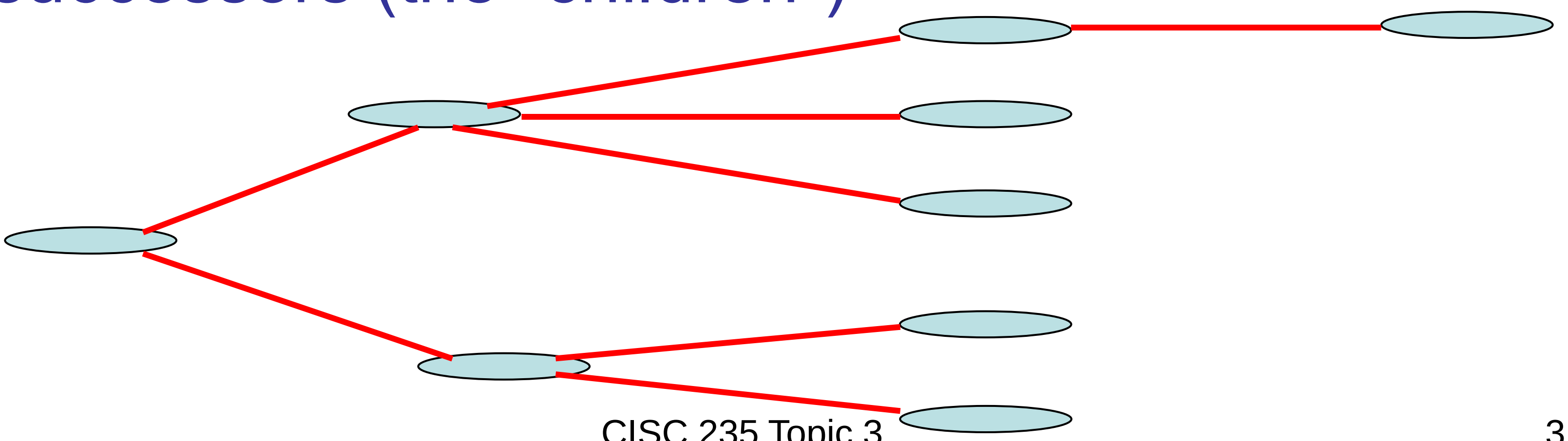
Outline

- General Trees
  - Terminology, Representation, Properties
- Binary Trees
  - Representations, Properties, Traversals
- Recursive Algorithms for Trees
- Binary Search Trees
  - Searching, Insertion, and Deletion
  - Analysis of Complexity

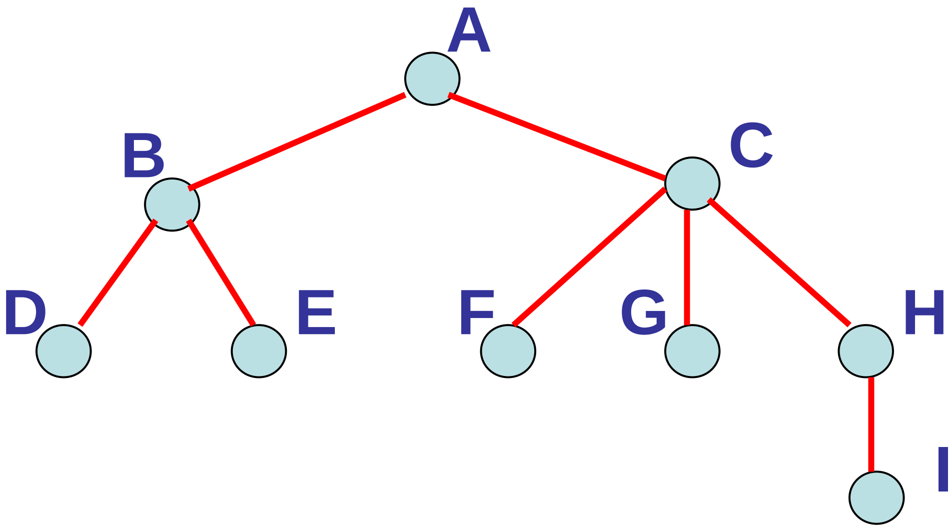
Rooted Trees

In a sequence, each element has zero or one predecessors & zero or one successors

In a rooted tree, each element has either zero or one predecessor (the “parent”) and zero or more successors (the “children”)



Tree Terminology



Parent of x

The node directly above node x in the tree

Child of x

A node directly below node x in the tree

Siblings

Nodes with common parent

Root

Only node with no parent

Leaf or External Node

A node with no children

Internal Node

Nonleaf node

Tree Terminology

Path

A sequence of connected nodes

Ancestor of x

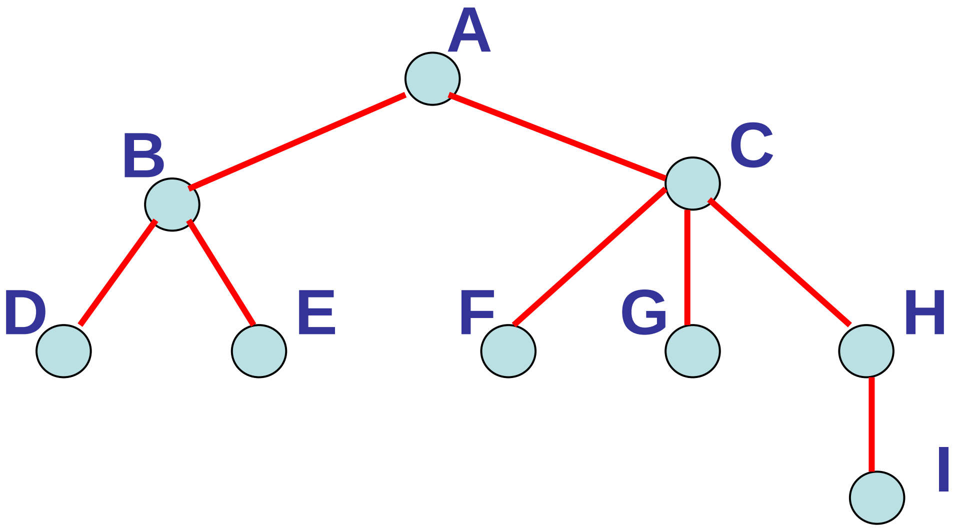
A node on the path from the root to x

Descendent of x

A node on a path from x to a leaf

Empty Tree

A tree with no nodes



Tree Terminology

Height of Tree

Number of edges on the longest path from the root to a leaf

Depth of x

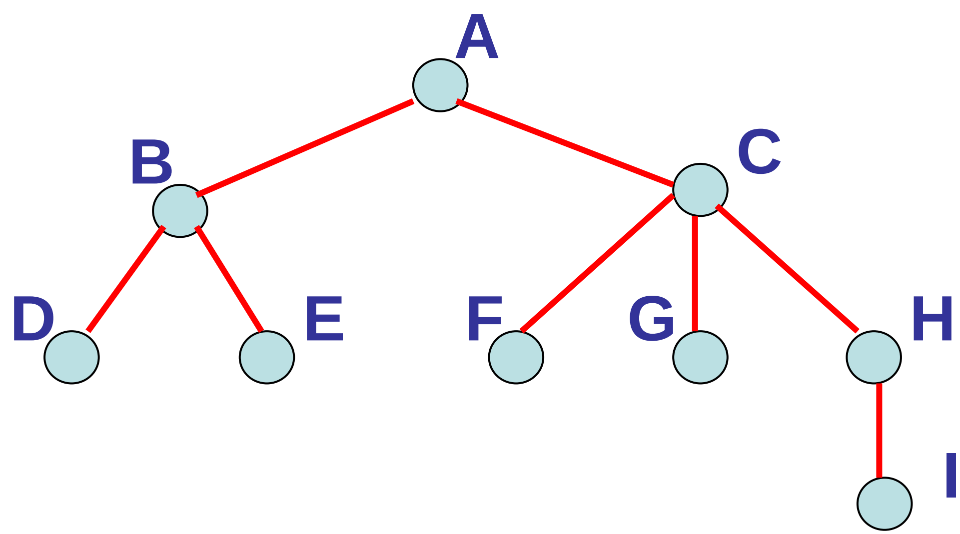
Number of ancestors x has

Level of x

Number of edges on the path from the root to x (= depth)

Degree of x

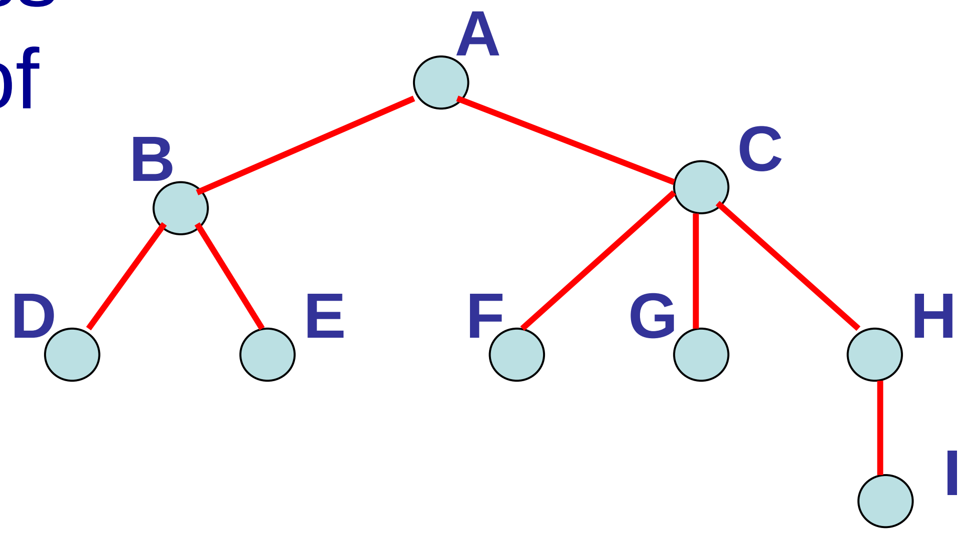
Number of children x has



Tree Properties

The number of edges in a tree is one less than the number of nodes.

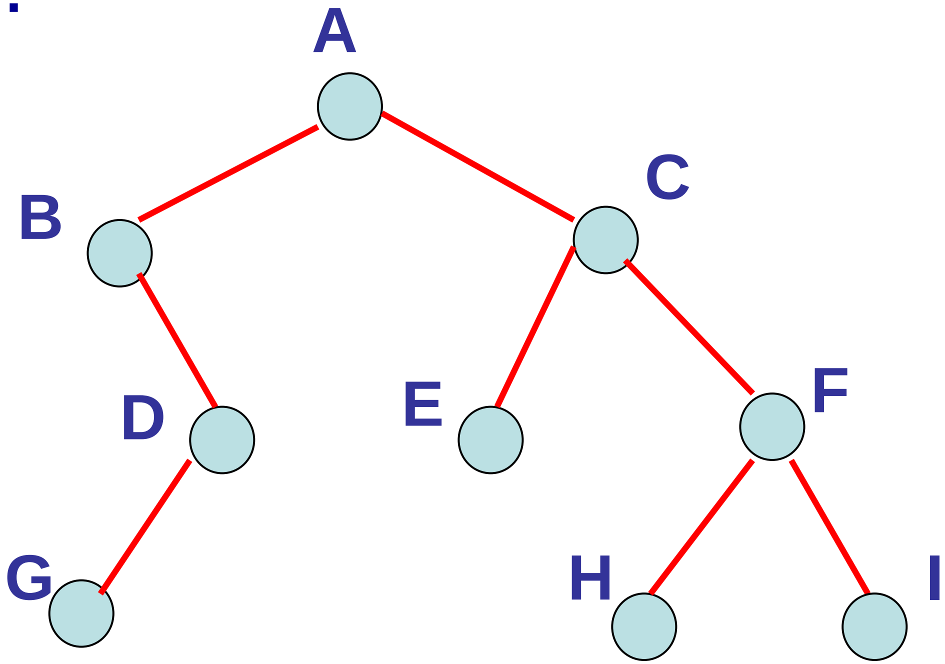
$$|E| = |V| - 1$$



Binary Trees: Recursive Definition

A *binary tree* is one of:

- An empty tree
- A root with two binary trees as children (its left and right sub-trees)



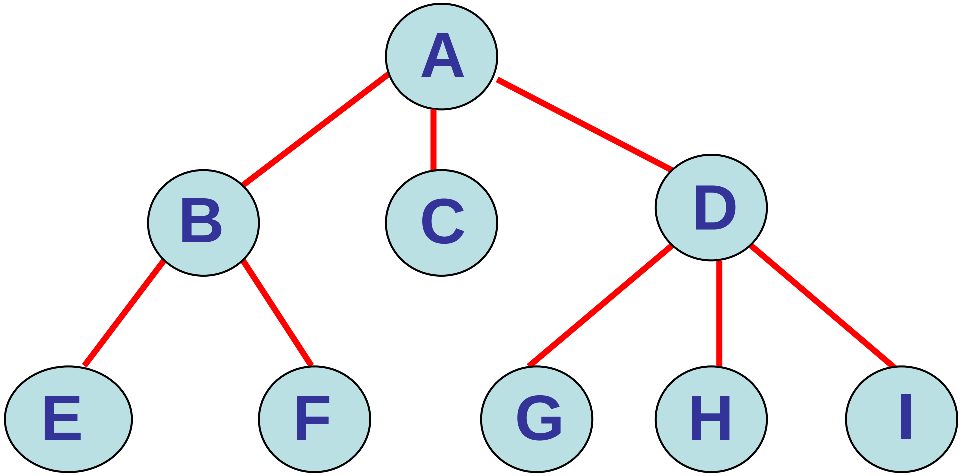
Tree Implementations

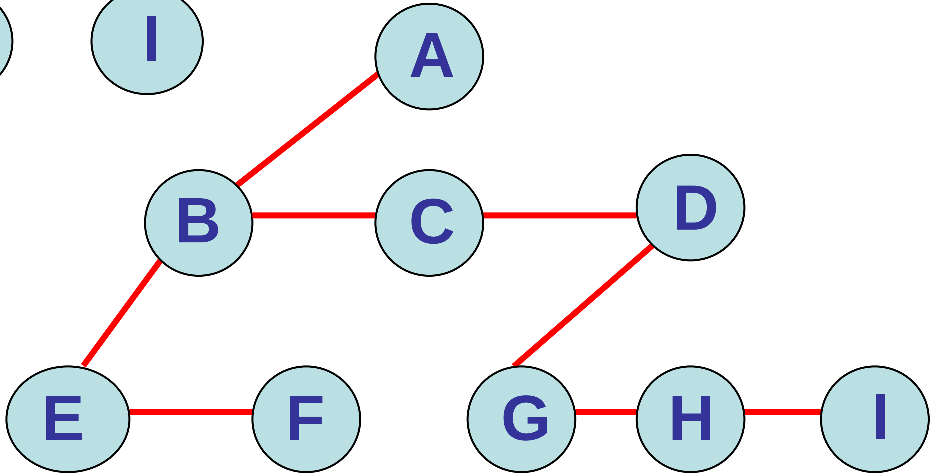
What would a linked representation of a binary tree look like?

What would a linked representation of a general tree look like?

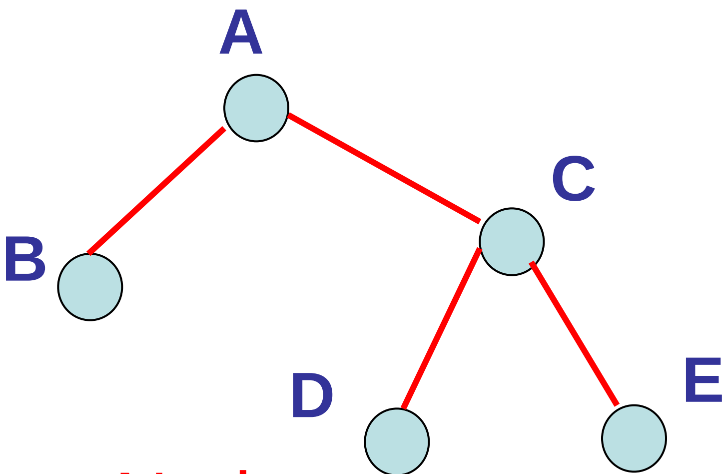
What would an array representation of a binary tree look like?

Representing a General Tree as a Binary Tree





Array Implementation



	A	B	C			D	E

Node	Location
Root	1
Left Child of x	2 * location of x
Right Child of x	(2 * location of x) + 1

Traversal Sequences of Binary Trees

- Level Order
- Preorder

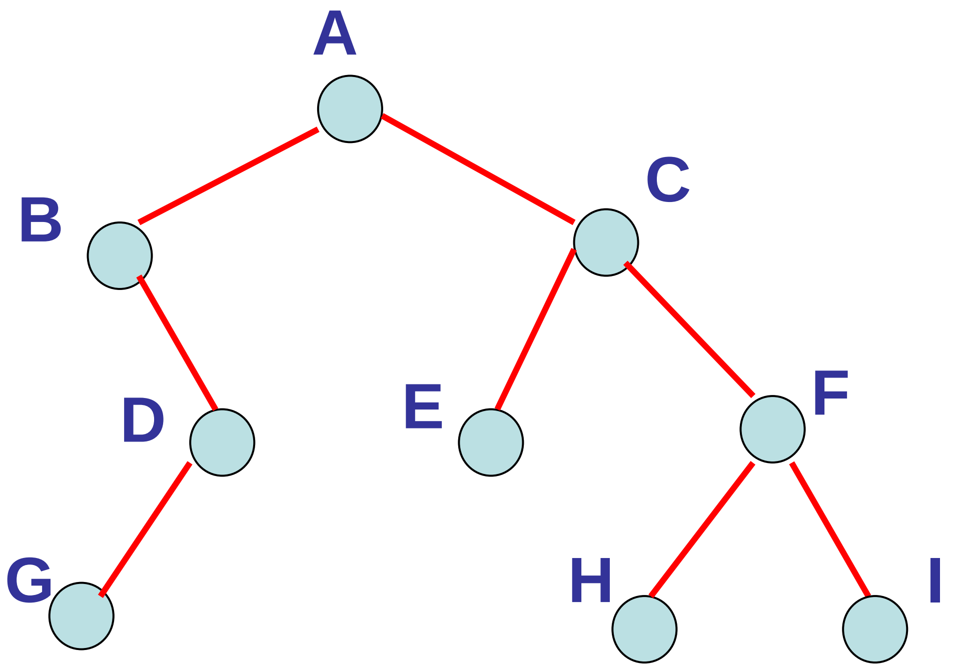
  Root | LT | RT

- Inorder

  | LT | Root | RT

- Postorder

  | LT | | RT | Root



Inorder Traversal Algorithm

Inorder-Tree-Walk (x)

if x $\neq$ NIL

then Inorder-Tree-Walk(*left*[x])

print *key*[x]

Inorder-Tree-Walk(*right*[x])

Inorder Traversal Print Method

```
void printInorder ( TreeNode root )
{
        if ( root != null )
        {       printInorder( root.left );
                System.out.println ( root.data );
                printInorder( root.right );
        }
}
```

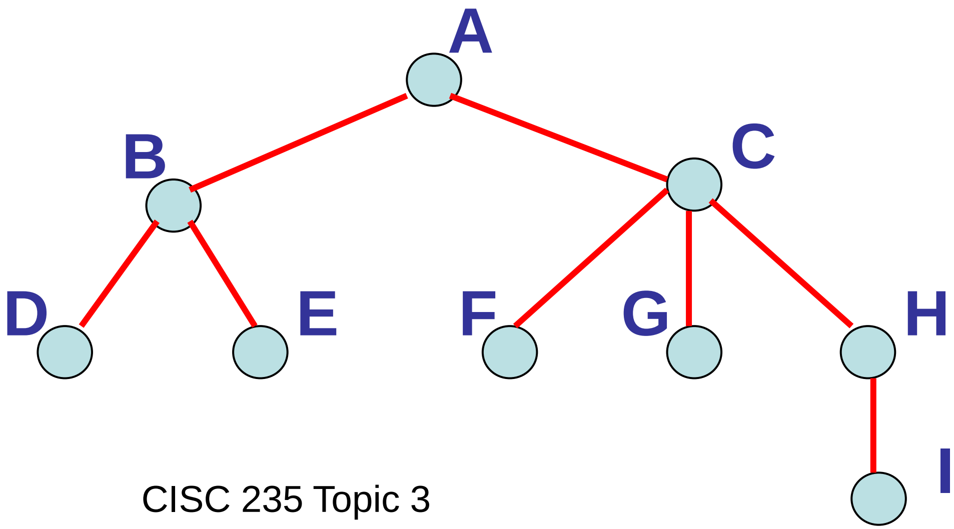
Preorder Traversal for General Tree

Preorder-Tree-Walk (x)

print *key*[x]

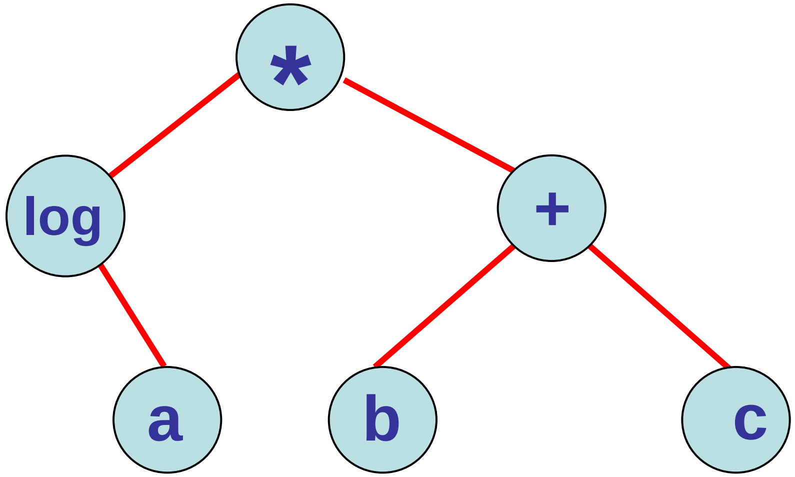
for each child w of x do

Preorder-Tree-Walk(w)



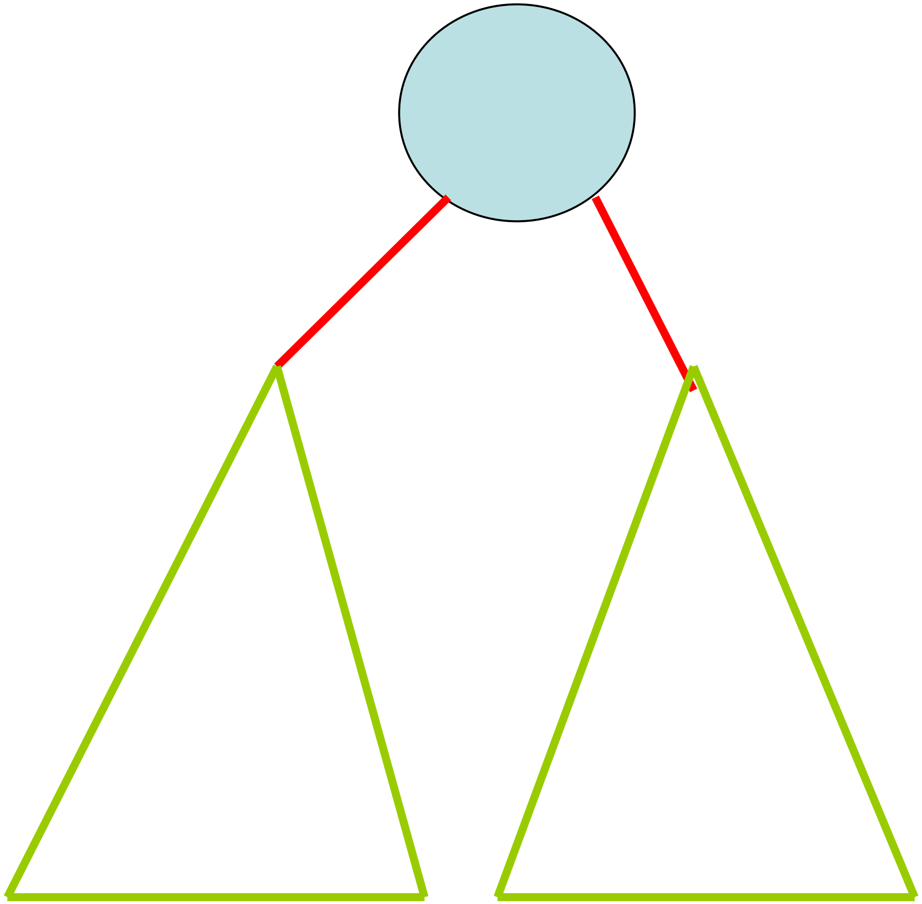
Application: Expression Trees

- Infix – customary form
  - log a * (b + c)
- Prefix – every operator before its operand(s)
  - * log a + b c
- Postfix – every operator after its operand(s)
  - a log b c + *



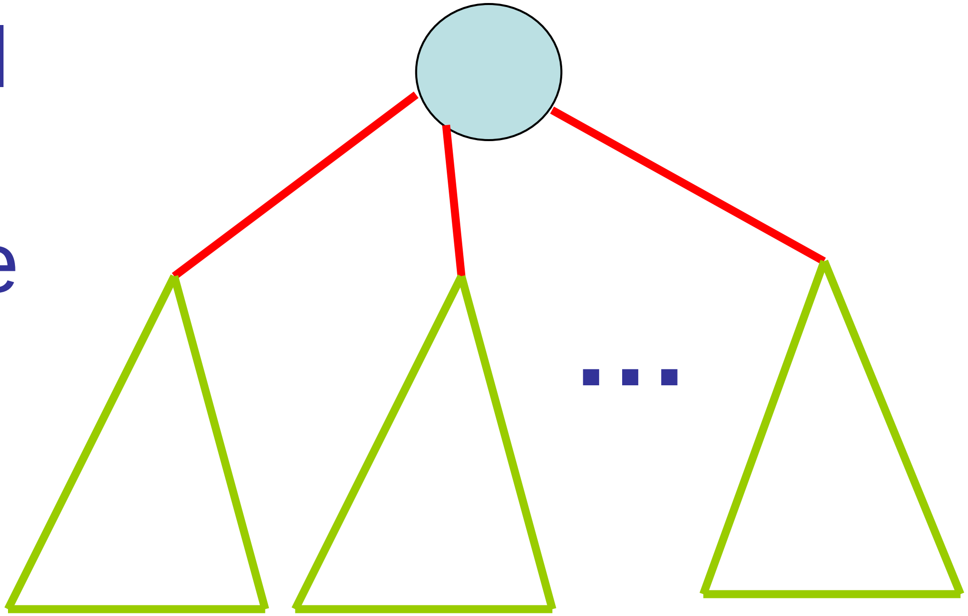
Recursive Thinking with Binary Trees

- Assume you have the solution for both the left and right sub-tree (i.e., the recursive calls).
- Then figure out the solution for the whole tree.



Recursive Thinking with General Trees

- Assume you have the solution for all of the sub-trees (i.e., the recursive calls).
- Then figure out the solution for the whole tree.



Practice Thinking Recursively

1. Write an algorithm to calculate the total number of nodes in a binary tree.

2. Write an algorithm to calculate the height of a binary tree.

Application: File Systems

1. Print a list of all the files and folders in a hierarchical directory structure

2. Calculate the total number of blocks used by all files in the directory system.

Print a Directory List (pseudocode)

```
void listAll( int depth )
{
        printName( depth );   // Print name of object
        if ( isDirectory( ) )
                for each file c in this directory (each child)
                        c.listAll( depth + 1 );
}
```

Calculate Total Size of All Files in a Directory (pseudocode)

```
int size( )
{
        int totalSize = sizeOfThisFile( );
        if ( isDirectory( ) )
                for each file c in this directory (each child)
                        totalSize += c.size( );

         return totalSize;
}
```

Binary Tree Properties

Minimum # of nodes, minN, in a tree of height h:

minN = ______________________

Maximum height, maxH, for tree of n nodes:

maxH = ______________________

Maximum # of nodes, maxN, in a tree of height h:

maxN = ______________________

Minimum height, minH, in tree of n nodes:

minH = ______________________

Searching Methods

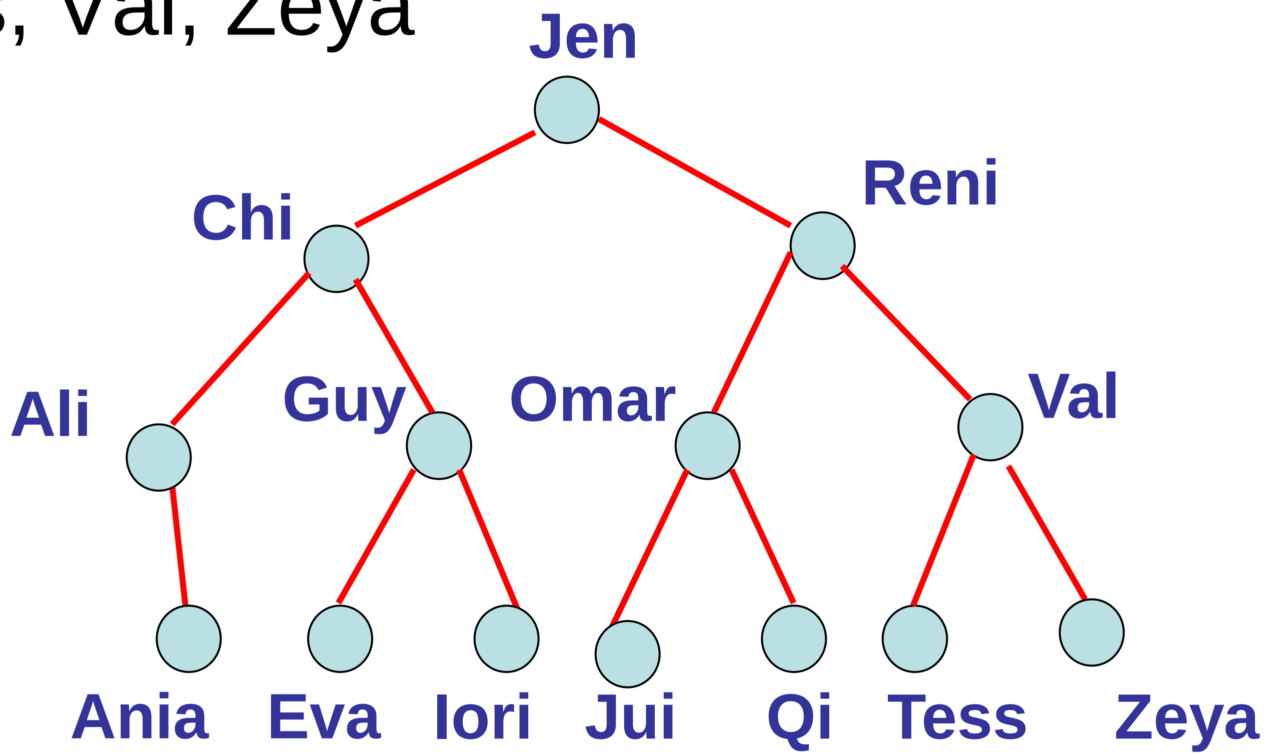
Search: Given an integer x and a list L, presorted in increasing order, return the location of x in L, or -1 if x is not in L

Algorithms: Linear Search: $O(n)$

Binary Search: $O(\log n)$

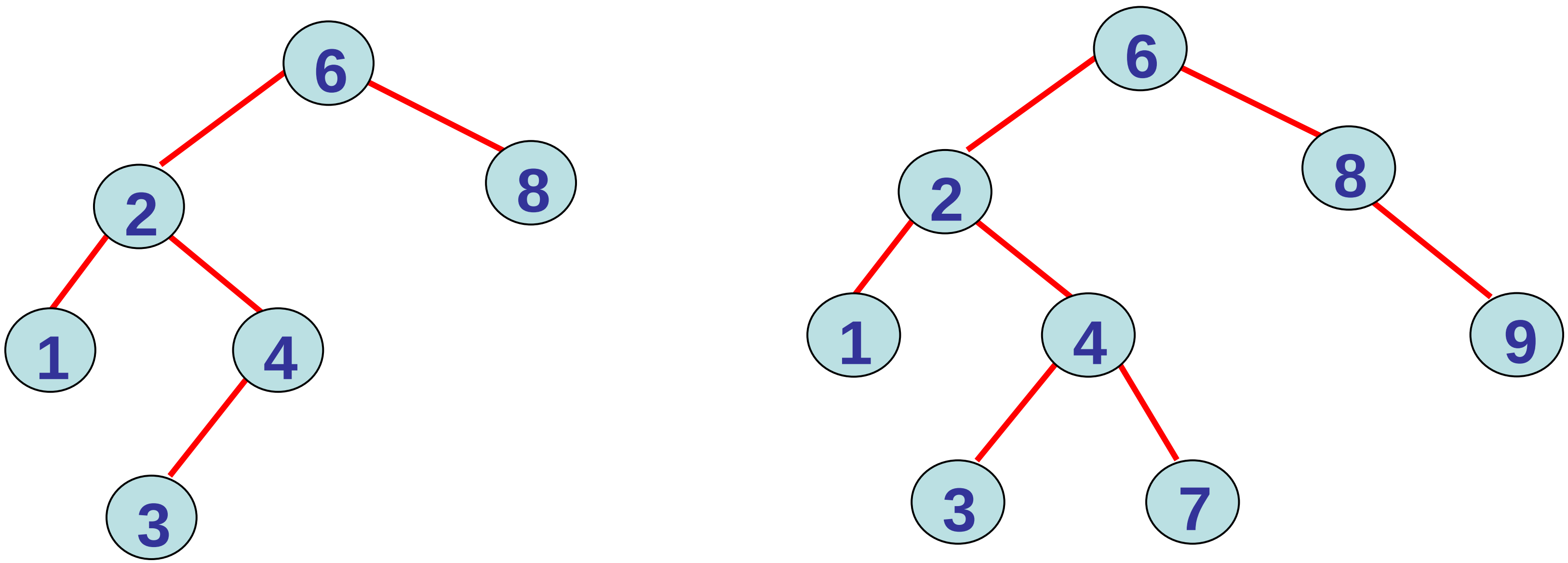
Comparison Tree

Consider a list of names: Ali, Ania, Chi, Eva, Guy, Iori, Jen, Jui, Omar, Qi, Reni, Tess, Val, Zeya



Binary Search Tree Property

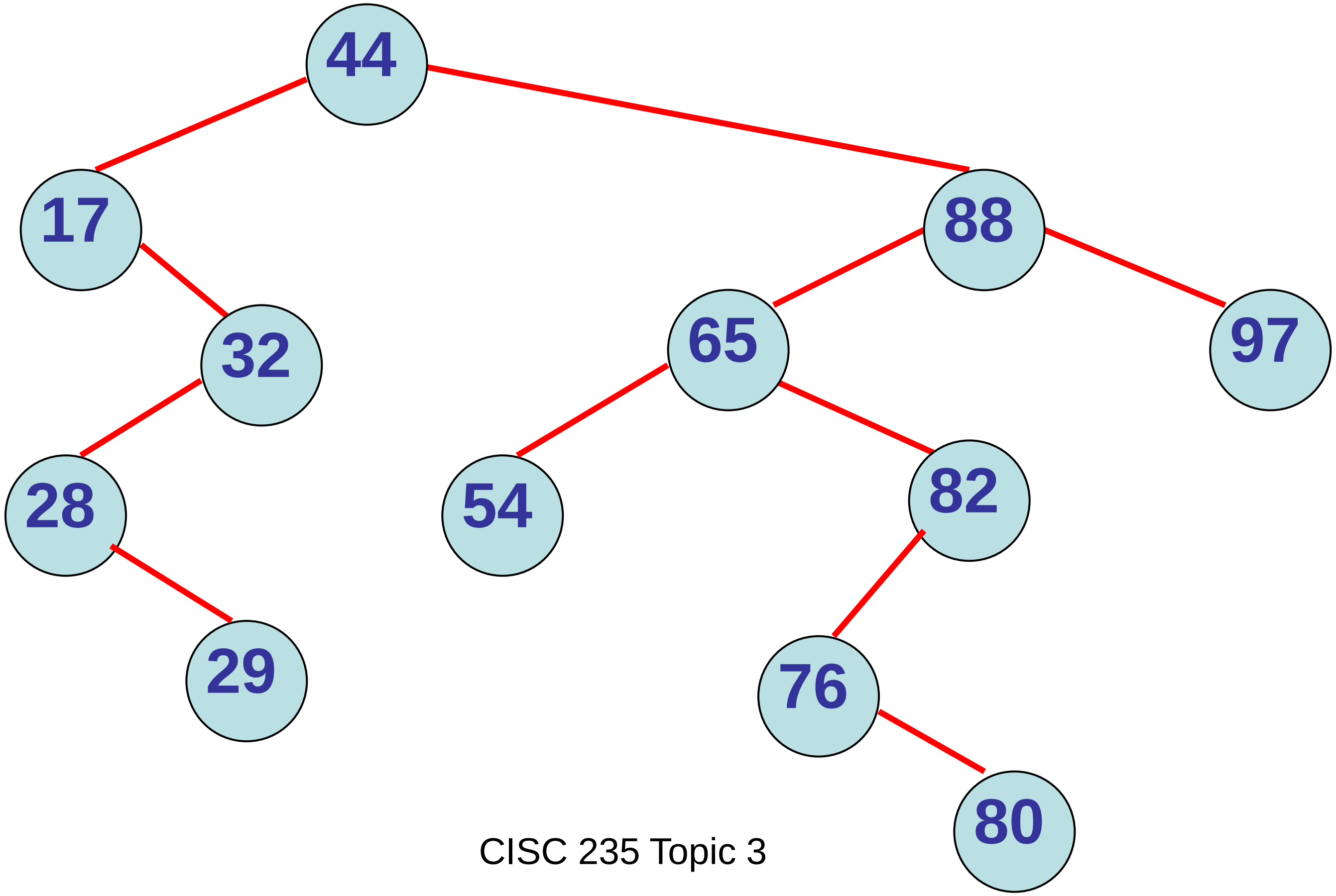
Let x be a node in a binary search tree. If y is a node in the left subtree of x, then *key*$[y] \le$ *key*$[x]$. If y is a node in the right sub-tree of x, then *key*$[x] \le$ *key*$[y]$.



Which is not a BST? Why not?

Binary Search Tree Algorithms

Find, FindMin, FindMax



Recursive Search Algorithm

```
// Returns node with key k in BST rooted
// at node x, or NIL if not found
Tree-Search( x, k )
   if x = NIL or k = key[ x ]
        then return x
   if k < key[ x ]
        then return Tree-Search( left[ x ], k )
        else return Tree-Search( right[ x ], k )
```

Iterative Search Algorithm

// Returns node with key k in BST rooted
// at node x, or NIL if not found

```
Iterative-Tree-Search( x, k )
    while x ≠ NIL and k ≠ key[ x ]
        do if k < key[ x ]
            then x ← left[ x ]
            else x ← right[ x ]
    return x
```

Algorithm to Find Minimum

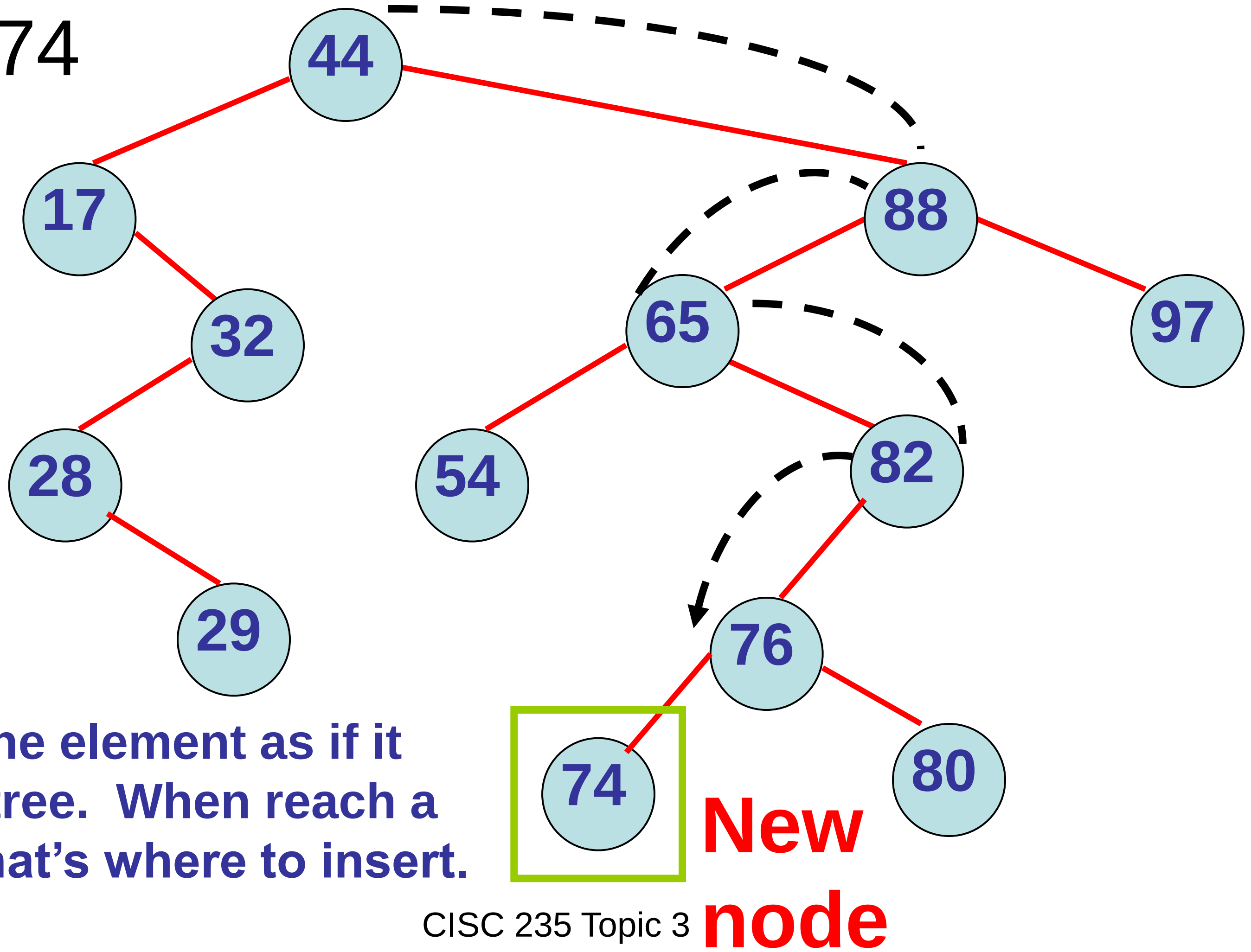
```
// Returns minimum in BST
// rooted at node x
Tree-Minimum( x )
        while left[ x ] ≠ NIL
                do x ← left[ x ]
        return x
```

Algorithm to Find Maximum

```
// Returns maximum in BST
// rooted at node x
Tree-Maximum( x )
        while right[ x ] ≠ NIL
                do x ← right[ x ]
        return x
```

Binary Search Tree: Insertion

Insert 74



Search for the element as if it were in the tree. When reach a null node, that's where to insert.

Recursive Insertion Algorithm for BST with no pointers to parents

```
// Inserts node y into BST rooted at node root
// Only for BSTs with no pointers to parents
BSTInsert ( y, root )
    if ( root = NIL )  // have found where to insert
            then root ← y
      else if key[ y ] < key[ root ]
            then left[root] = BSTInsert( y, left[ root ] )
      else if key[ y ] > key[ root ]
            then right[root] = BSTInsert( y, right[ root ] )
      else y is a duplicate; handle duplicate case
    return root
```

Recursive Insert Function for BST with no pointers to parents

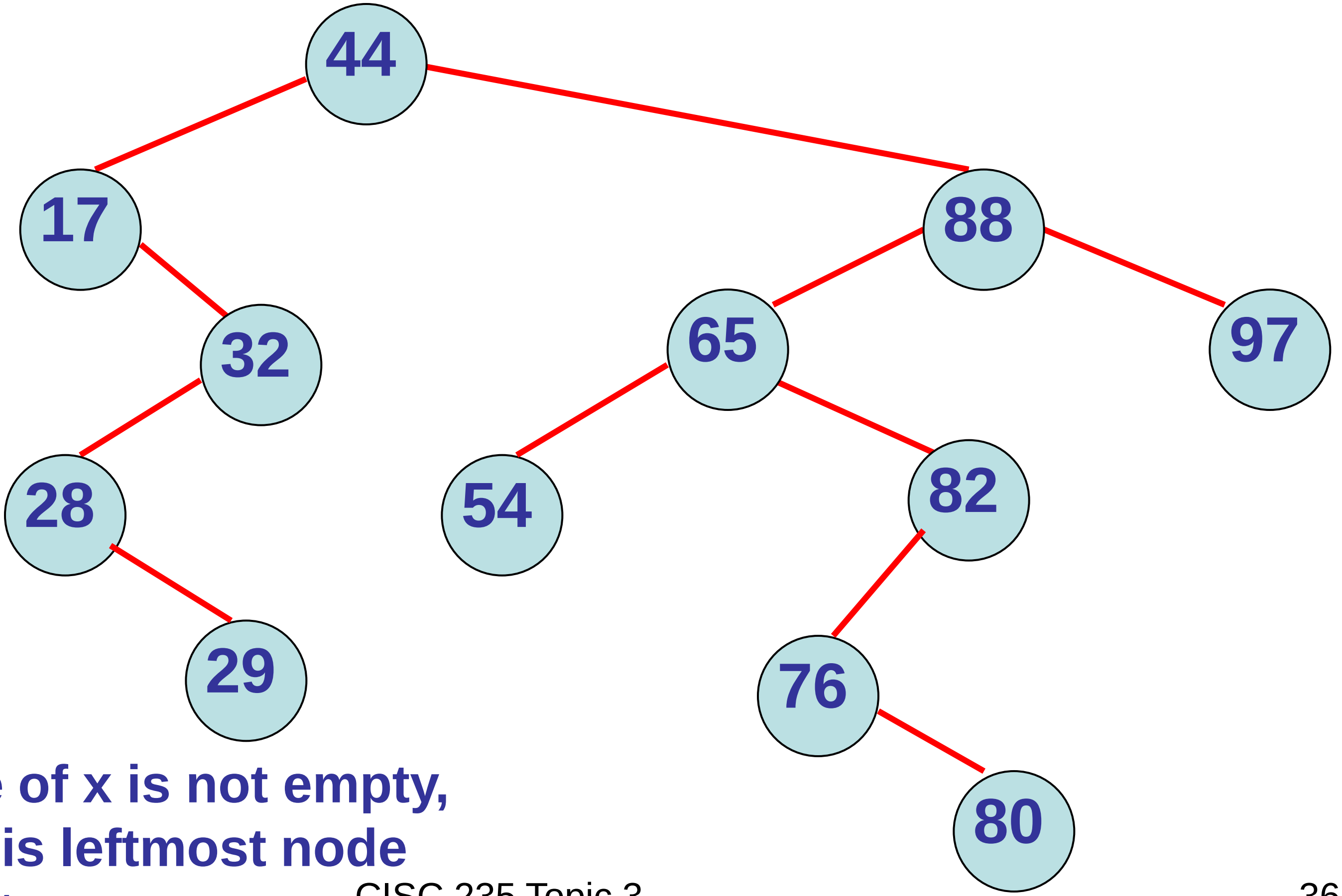
```
Node insert ( int x, Node root )
{   Node p = new Node( x );
    if( root == null )                // empty tree
        root = p;
    else if ( x < root.data )         // goes in left sub-tree
        root.left = insert( x, root.left );
    else if ( x > root.data )         // goes in right sub-tree
        root.right = insert( x, root.right );
    else
        System.out.println( x + " is already in tree" );
    return root;
}
```

Iterative Insertion Algorithm for BST with pointers to parents

```
Tree-Insert ( T, z )                    // Inserts node z into BST T
        y ← NIL
        x ← root[ T ]
        while  x ≠ NIL                  // Find position at which to insert
                do y ← x
                if key[ z ] < key[ x ]
                        then x ← left[ x ]
                        else x ← right[ x ]
        p[ z ] ← y                          // Set parent link of new node
        if y = NIL                          // If tree T was empty
                then root[ T ] ← z                // New node is root, else
                else if key[ z ] < key[ y ]       // Connect node to parent
                        then left[ y ] ← z
                        else right[ y ] ← z
```

BST: Find Successor of a Node

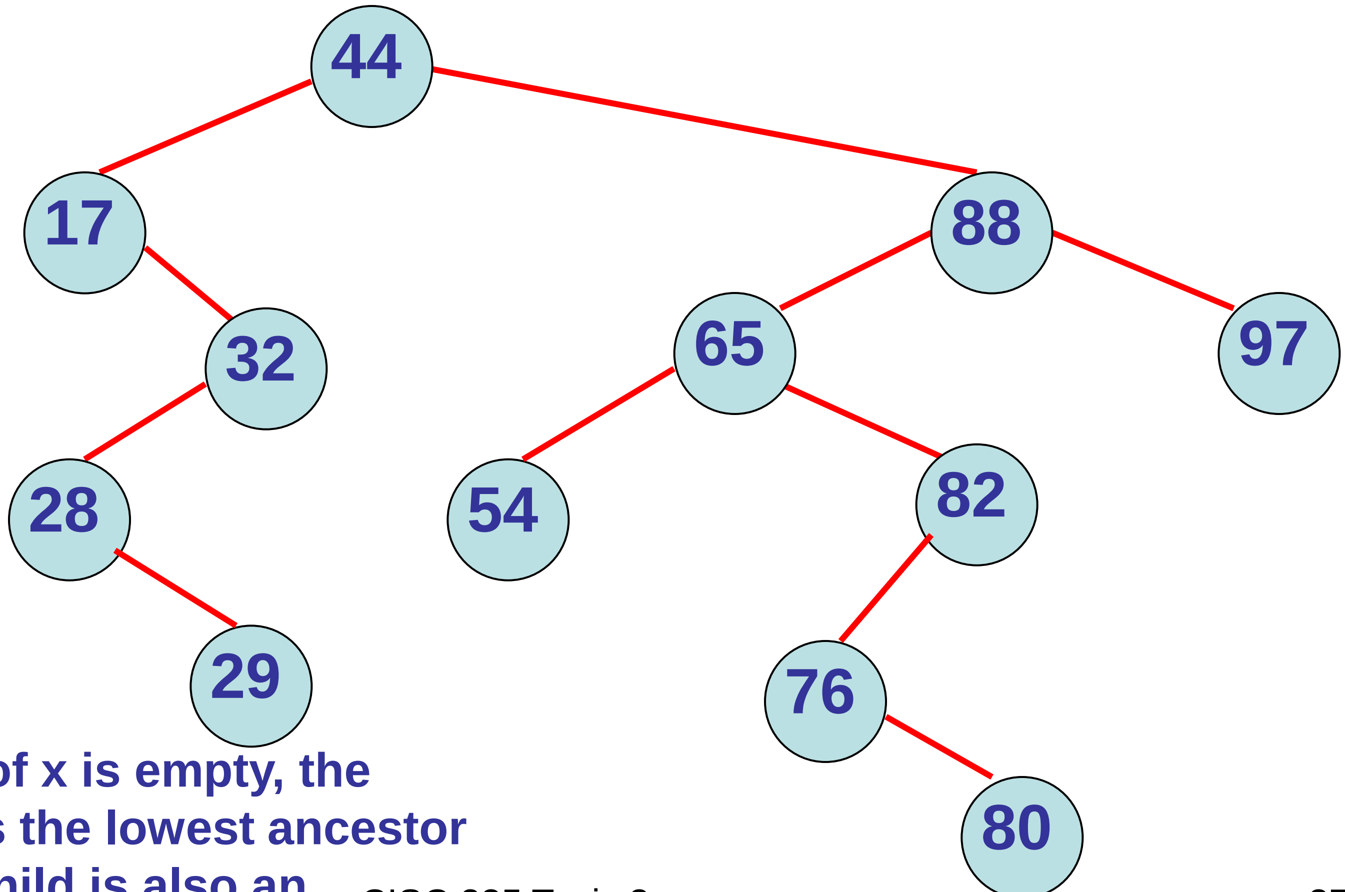
Find Successor of 65



If right sub-tree of x is not empty, successor of x is leftmost node in its right sub-tree.

BST: Find Successor of a Node

Find Successor of 32



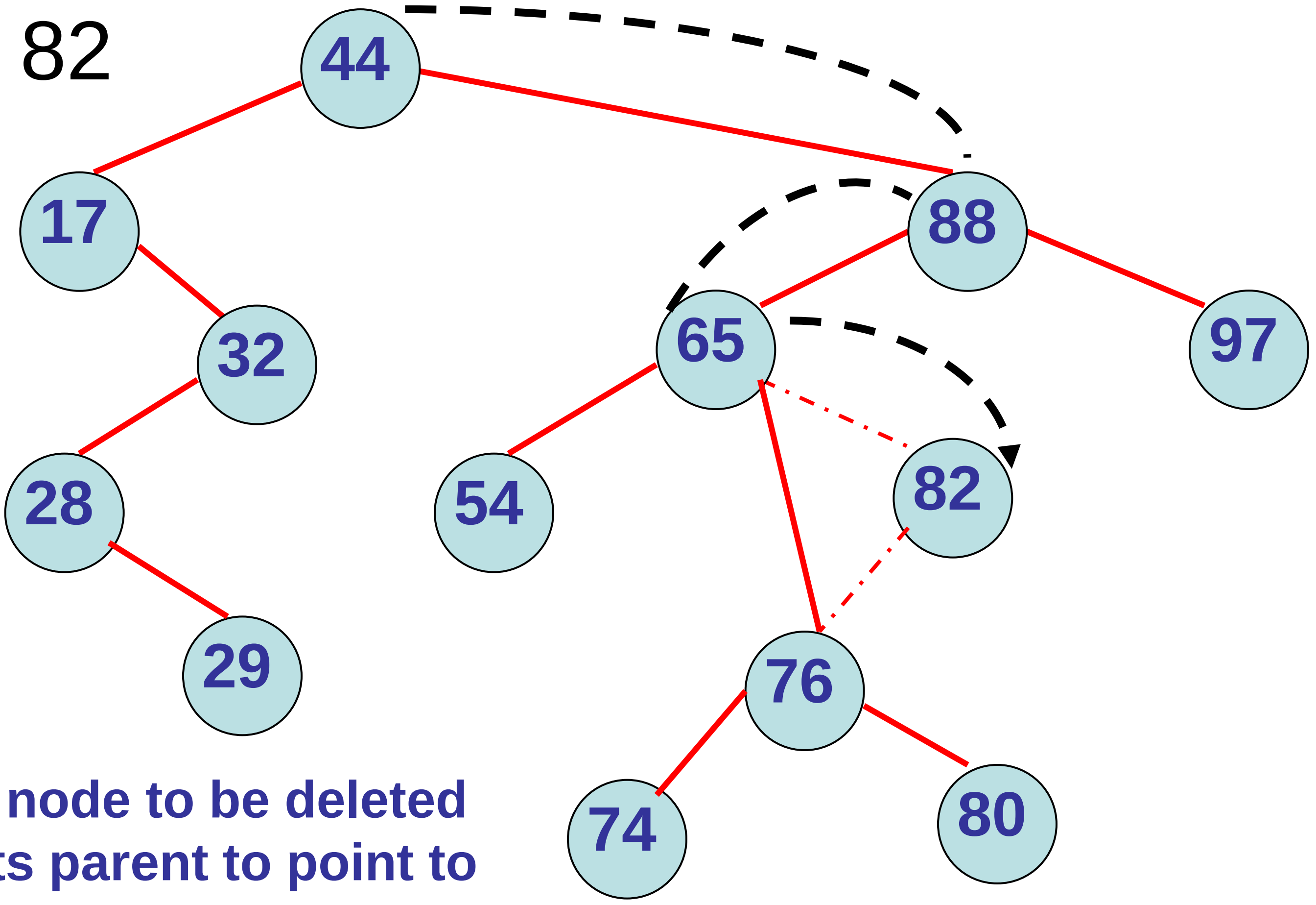
If right sub-tree of x is empty, the successor of x is the lowest ancestor of x whose left child is also an ancestor of x

Find Successor Algorithm for BST with pointers to parents

```
// Returns node in BST that is the successor
// of node x, or NIL if no successor
Tree-Successor( x )
        if right[ x ] ≠ NIL
                then return Tree-Minimum( right[ x ] )
        y ← p[ x ]
        while y ≠ NIL and x = right[ y ]
                do x ← y
                    y ← p[ y ]
        return y
```

Deletion of Node with Zero or One Child

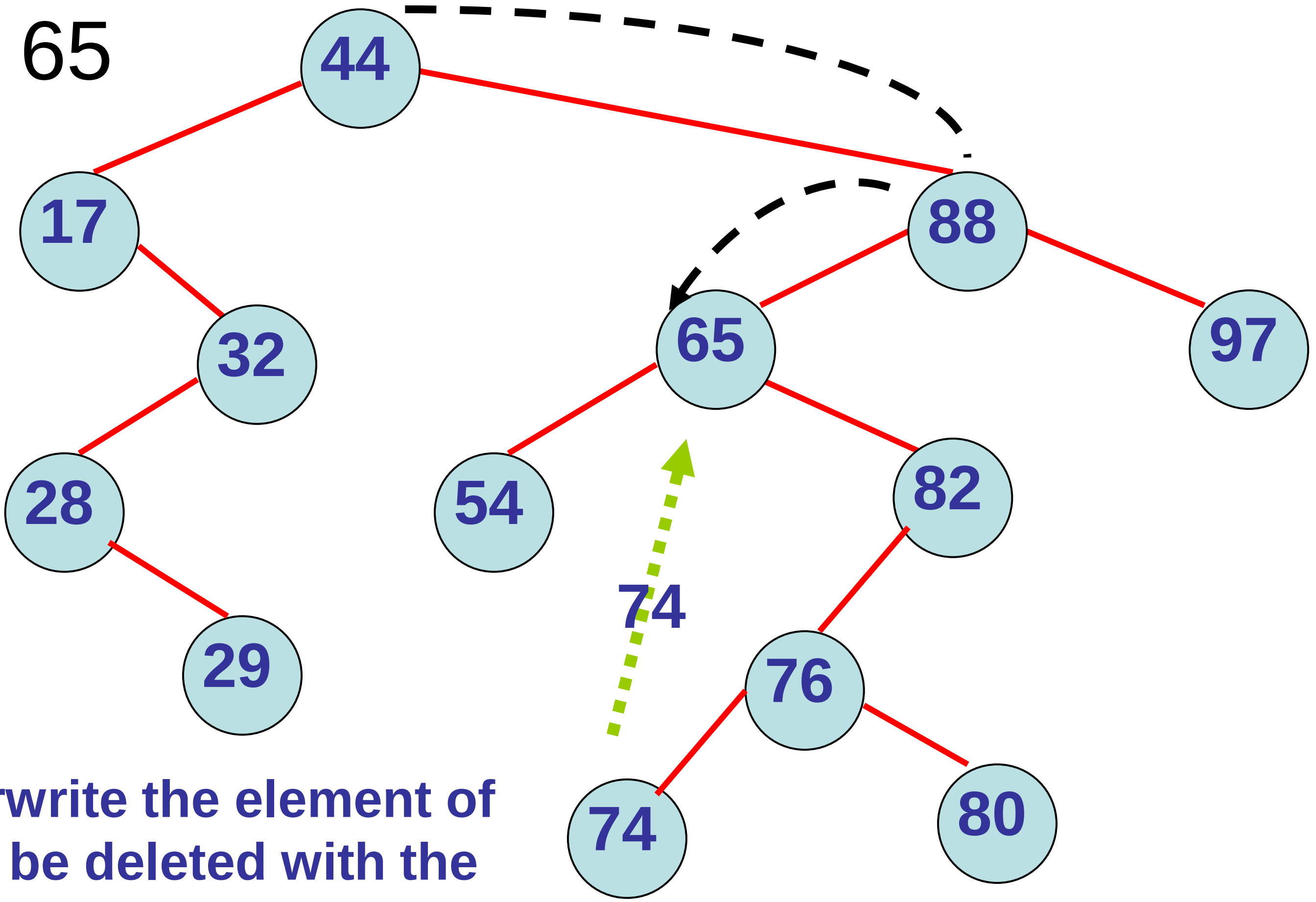
Delete 82



Bypass the node to be deleted by setting its parent to point to its child

Deletion of Node with Two Children

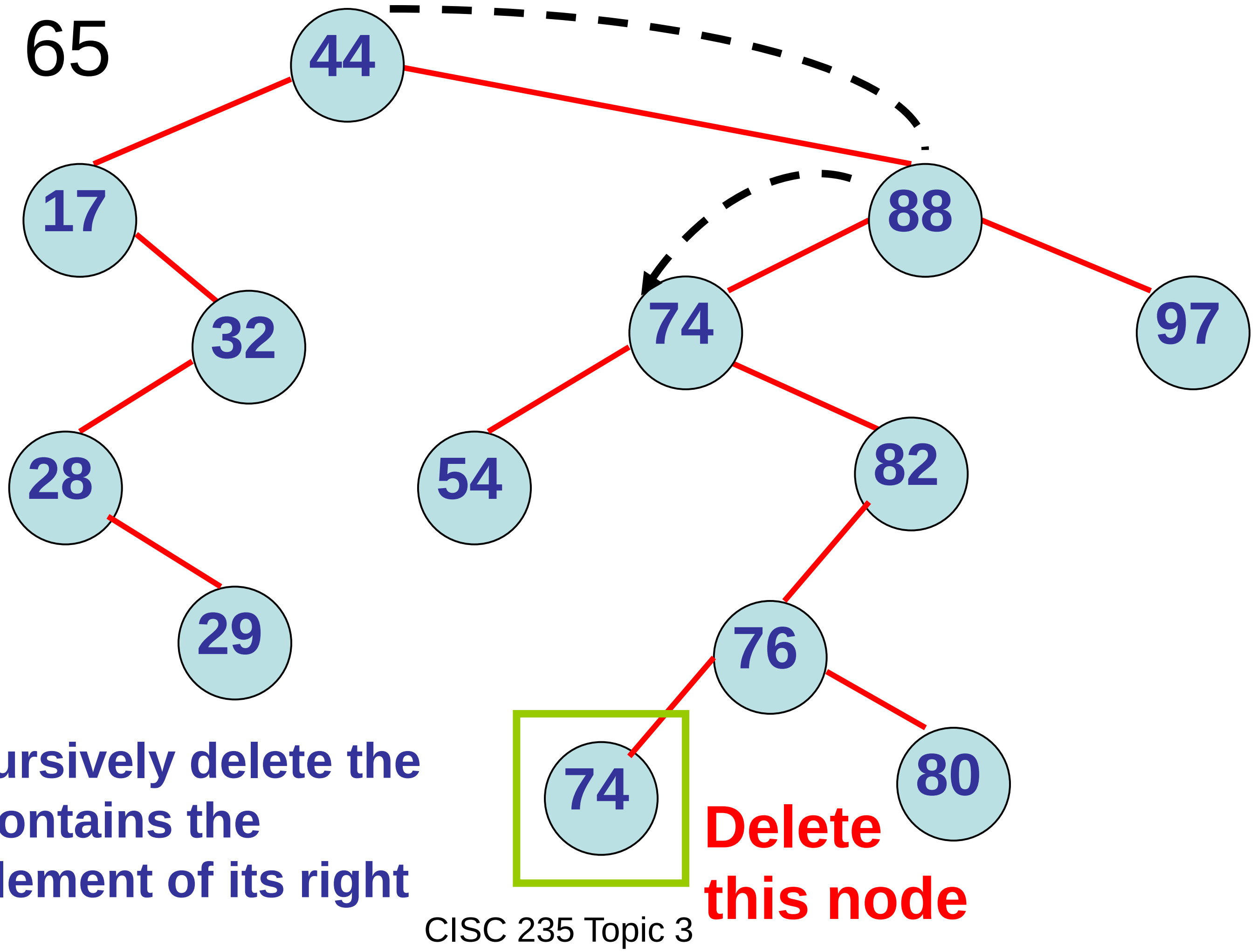
Delete 65



Step1: **Overwrite the element of the node to be deleted with the minimum element of its right sub-tree**

Deletion of Node with Two Children

Delete 65



Step2: **Recursively delete the node that contains the minimum element of its right sub-tree**

Iterative Deletion Algorithm for BST with pointers to parents

```
// Deletes node z from BST T
Tree-Delete ( T, z )
    // Set y to point to node to splice out
    if left[ z ] = NIL or right[ z ] = NIL
        then y ← z
        else y ← Tree-Successor( z )
    // Set x to non-NIL child of y,
    // or to NIL if y has no children
    if left[ y ] ≠ NIL
        then x ← left[ y ]
        else x ← right[ y ]
```

Deletion Algorithm, con.

```
// Splice out node y (next 7 lines)
    if x ≠ NIL
            then p[ x ] ← p[ y ]
    if p[ y ] = NIL
            then root[ T ] ← x
            else if y = left[ p[ y ] ]

                    then left[ p[ y ] ] ← x
                    else right[ p[ y ] ] ← x
// If successor to z was spliced out, copy y's data to z
    if y ≠ z
            then key[ z ] ← key[ y ]
    return y
```

Binary Search Tree: Complexity

Search, insertion, and deletion in a binary search tree are all $O(h)$, where h is the height of the tree.

What does this imply about the complexity in terms of n?