CSC 130:
Trees

Trees

- A tree is a collection of nodes:
  - One node is the root node.
- A node contains data and has pointers (possibly null) to other nodes, its children.
  - The pointers are directed edges.
  - Each child node can itself be the root of a subtree.
  - A leaf node is a node that has no children.
- Each node other than the root node has exactly one parent node.

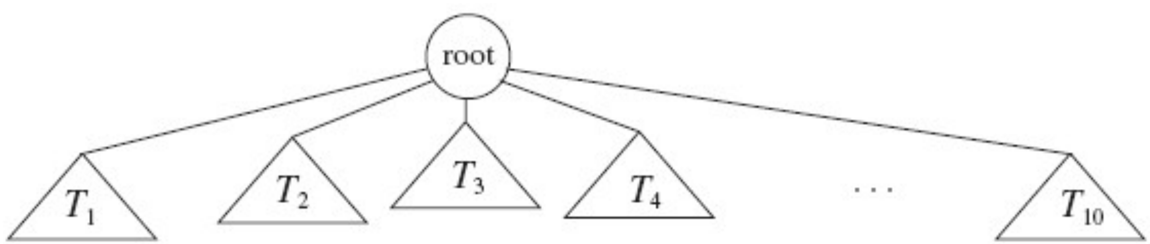


Figure 4.1 Generic tree

Trees, *cont'd*

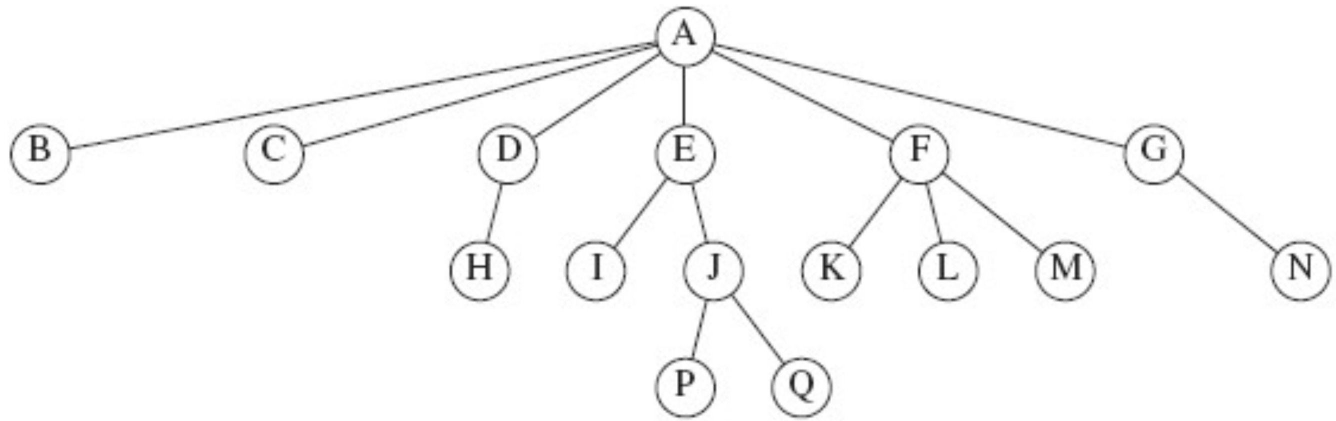


Figure 4.2 A tree

- The **path** from node n_1 to node n_k is the sequence of nodes in the tree from n_1 to n_k.
  - What is the path from A to Q? From E to P?

- The **length** of a path is the number of its edges.
  - What is the length of the path from A to Q?

Trees, *cont'd*

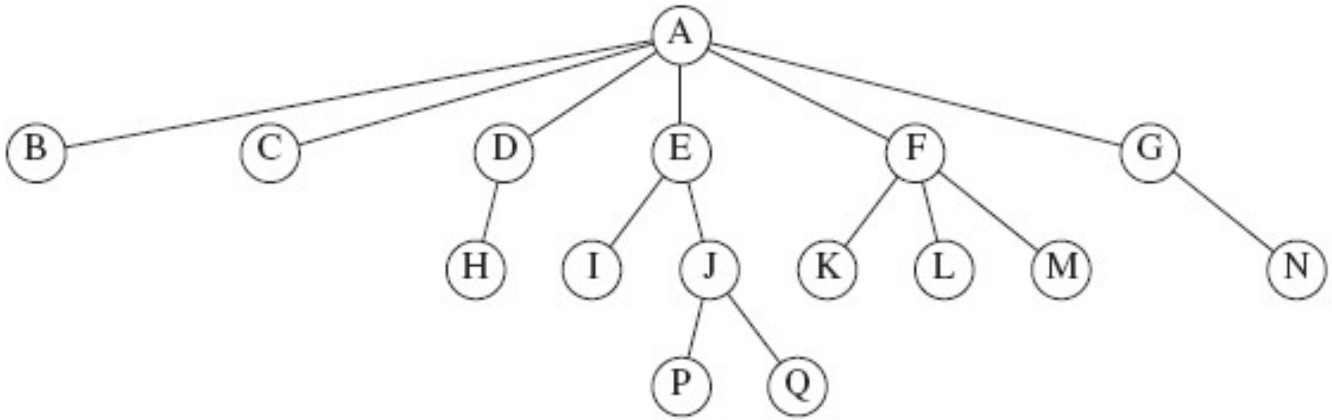


Figure 4.2 A tree

- The depth of a node is the length of the path from the root to that node.
  - What is the depth of node J? Of the root node?

Trees, *cont'd*

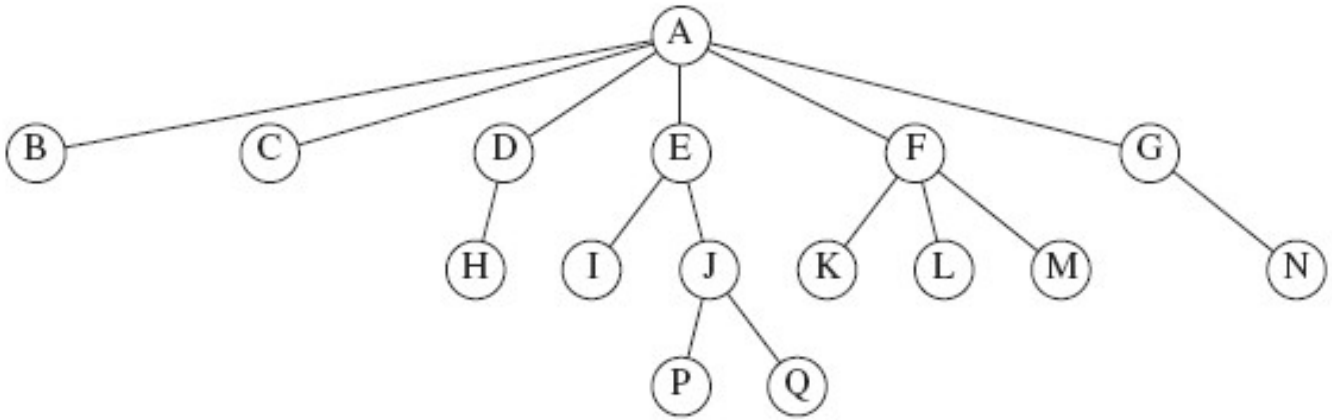


Figure 4.2 A tree

- The height of a node is the length of the longest path from the node to a leaf node.
  - What is the height of node E? Of the root node?
- Depth of a tree = depth of its deepest node = height of the tree

Tree Implementation

- In general, a tree node can have an arbitrary number of child nodes.
- Therefore, each tree node should have
  * a link to its first child, and
  * a link to its next sibling:

```
class TreeNode
{
     Object   element;
     TreeNode firstChild;
     TreeNode nextSibling;
}
```

Tree Implementation, *cont'd*

- Conceptual view of a tree:

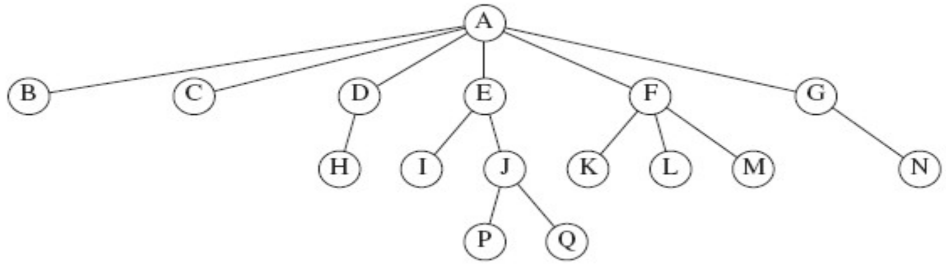


Figure 4.2 A tree

- Implementation view of the same tree:

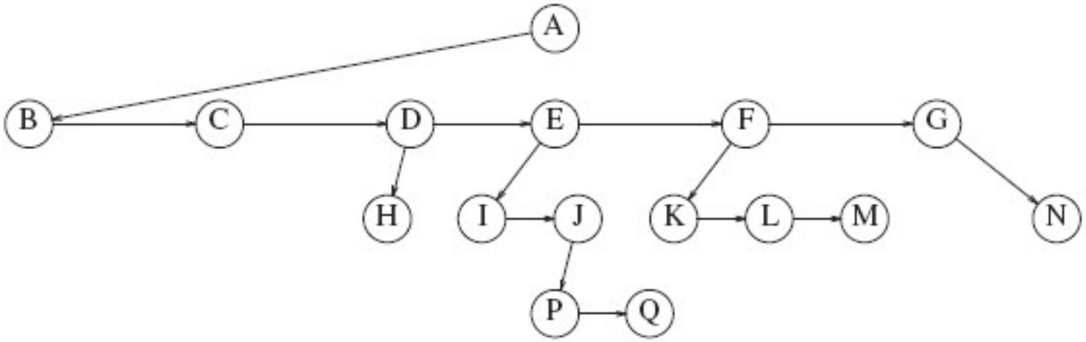


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Tree Traversals

- There are several different algorithms to "walk" or "traverse" a tree.
- Each algorithm determines a unique order that each and every node in the tree is "visited".

Preorder Tree Traversal

- First visit a node.
  - Visit the node before (pre) visiting its child nodes.
- Then recursively visit each of the node's child nodes in sibling order.

Preorder Tree Traversal, *cont'd*

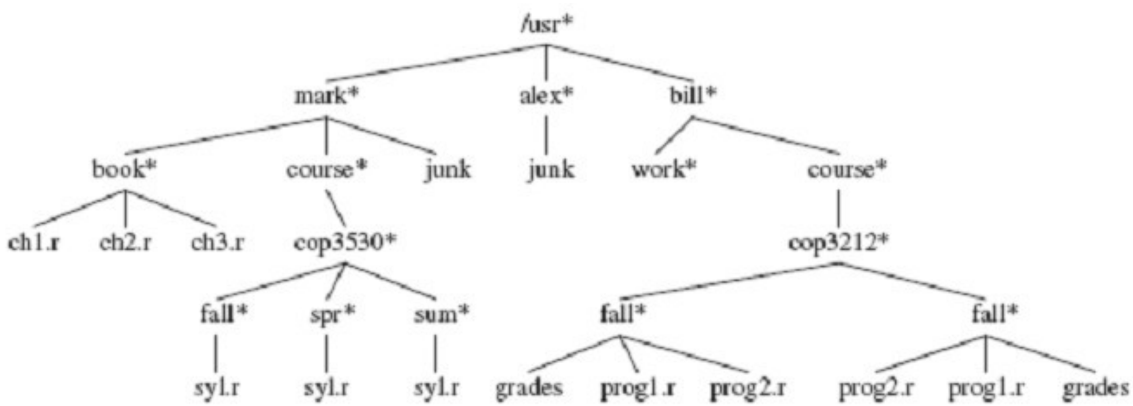


Figure 4.5 UNIX directory

```
private void listAll(int depth)
{
    printName(depth);
    if (isDirectory()) {
        for each file f in directory {
            f.listAll(depth+1);
        }
    }
}

public void listAll()
{
    listAll(0);
}
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Figure 4.7 The (preorder) directory listing

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Postorder Tree Traversal

- First recursively visit each of a node’s child nodes in sibling order.
- Then visit the node itself.

Postorder Tree Traversal, *cont'd*

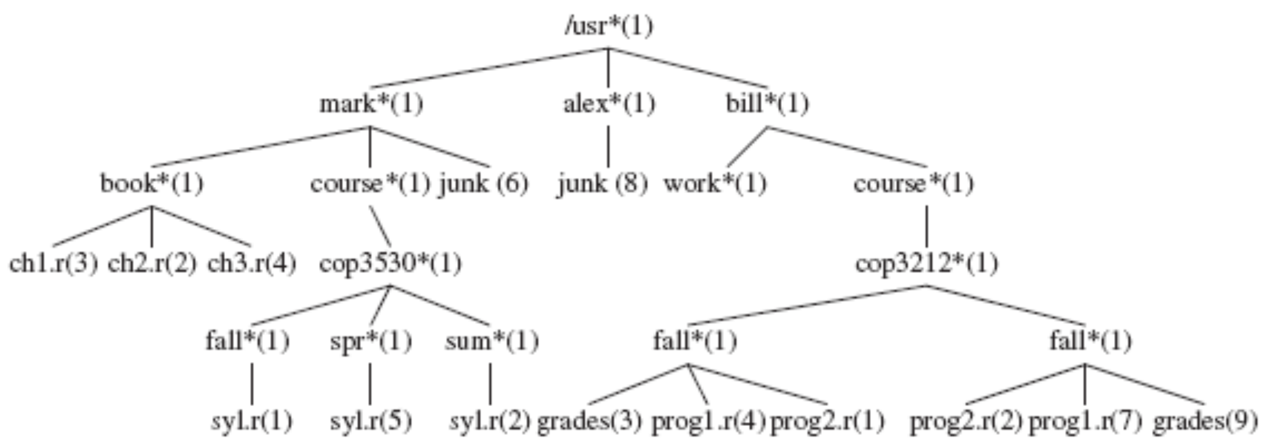


Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

```
private void size()
{
    int totalSize =
sizeOfThisFile();

    if (isDirectory()) {
        for each file f in
directory {
            totalSize += f.size();
        }
    }
```

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

Figure 4.10 Trace of the size function

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Binary Trees

- A **binary tree** is a tree where each node can have **0, 1, or 2 child nodes**.
  - The depth of an average binary tree with N nodes is much smaller than N: O

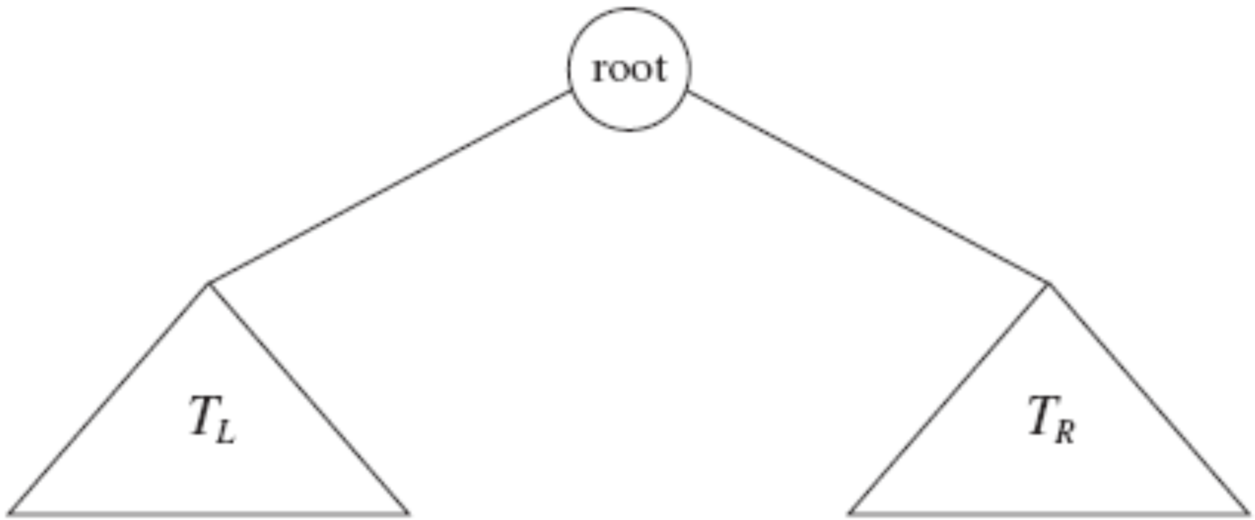


Figure 4.11 Generic binary tree

Binary Trees, *cont'd*

- An arithmetic expression tree:

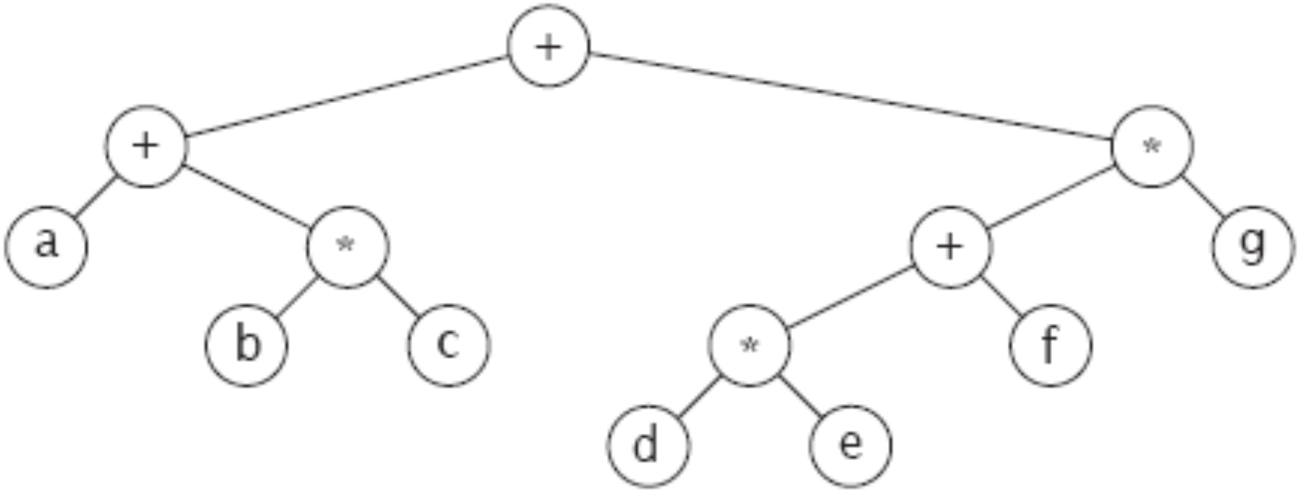


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Understand from the textbook how this tree is built.
  - Or take Compilers

Conversion from Infix to Postfix Notation

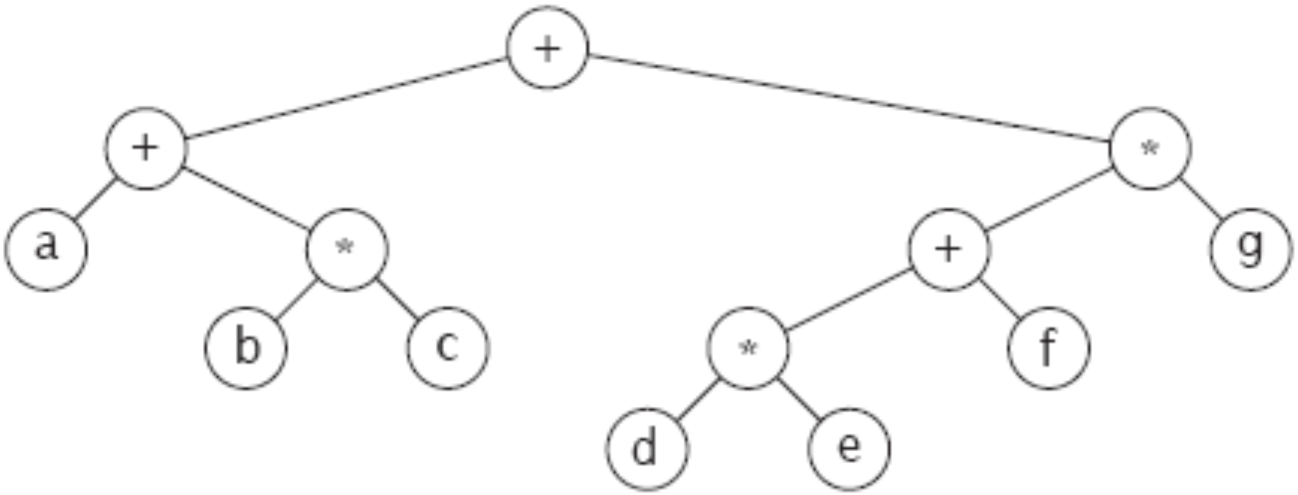


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Do a postorder walk of our expression tree to output the expression in postfix notation:

abc*+de*f+g*+

Binary Search Trees

- A binary search tree has these properties for each of its nodes:
  - All the values in the node's left subtree is less than the value of the node itself.
  - All the values in the node's right subtree is greater than the value of the node itself.

Inorder Tree Traversal

- Recursively visit a node’s left subtree.
- Visit the node itself.
- Recursively visit the node’s right subtree.

- If you do an inorder walk of a binary search tree, you will visit the nodes in sorted order.

Inorder Tree Traversal, *cont'd*

Figure 4.15 Two binary trees (only the left tree is a search tree)

- An inorder walk of the left tree visits the nodes in sorted order: 1 2 3 4 6 8

The Binary Search Tree ADT

- The node class of our binary search tree ADT.

```
private static class BinaryNode<AnyType>
{
    AnyType element;              // data in the node
    BinaryNode<AnyType> left;   // left child
    BinaryNode<AnyType> right; // right child

    BinaryNode(AnyType theElement)
    {
        this(theElement, null, null);
    }

    BinaryNode(AnyType theElement,
BinaryNode<AnyType> lt,

BinaryNode<AnyType> rt)
    {
        element = theElement;
        left    = lt;
        right   = rt;
```

The Binary Search Tree ADT

```
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
{
    private BinaryNode<AnyType> root;

    public BinarySearchTree()
    {
        root = null;
    }

    private BinaryNode<AnyType> findMin(BinaryNode<AnyType> t)
    {
        ...
    }

    ...

    private static class BinaryNode<AnyType>
    {
        ...
    }
}
```

The Binary Search Tree: Min and Max

- Finding the minimum and maximum values in a binary search tree is easy.
  - The leftmost node has the minimum value.
  - The rightmost node has the maximum value.
- You can find the minimum and maximum values recursively or (better) iteratively.

The Binary Search Tree: Min and Max, *cont'd*

- Recursive code to find the minimum value.
  - Chase down the left child links.

```
private BinaryNode<AnyType> findMin(BinaryNode<AnyType> t)
{
    if (t == null) {
        return null;
    }
    else if (t.left == null) {
        return t; // found the leftmost node
    }
    else {
        return findMin(t.left);
    }
}
```

The Binary Search Tree: Min and Max, *cont'd*

- Iterative code to find the maximum value.
  - Chase down the right child links.

```
private BinaryNode<AnyType> findMax(BinaryNode<AnyType> t)
{
    if (t != null) {
        while (t.right != null) {
            t = t.right;
        }
    }

    return t;
}
```

Checkpoint

- Design an algorithm which checks if a binary tree is a binary search tree

In-order traversal Solution

- In-order traversal, copy the elements to an array, and then check to see if the array is sorted.
  - Takes extra memory
  - Can't handle duplicate values in the tree properly... Why?

In-order traversal Solution

- Solution without Array

```
public static Integer last_printed = null;
public static boolean checkBST(TreeNode n){
    if (n == null) return true;
    // check / recurse left
    if (!checkBST(n.left)) return false;

    // check current
    if( last_printed != null && n.data <= last_printed) {
      return false;
    }

    //check / recurse right
    if (!checkBST(n.right)) return false;

    return true;
 }
```

left.data <= current.data < right.data

- ALL left nodes must be less than or equal to the current node, which must be less than all right nodes
- Let's draw out some examples

Min Max Approach

```
boolean checkBST(TreeNode n){
    return checkBST(n, null, null);
}

boolean checkBST(TreeNode n, Integer min, Integer max){
    if(n == null){
        return true;
    }

    if((min != null && n.data <= min) ||
       (max !=null && n.data > max)){
        return false;
    }

    if(!checkBST(n.left, min, n.data) ||
       !checkBST(n.right, n.data, max)){
        return false;
    }
    return true;
}
```

The Binary Search Tree: Contains

- Does a binary search tree contain a target value?
- Search recursively starting at the root node:
  - If the target value is less than the node's value, then search the node's left subtree.
  - If the target value is greater than the node's value, then search the node's right subtree.
  - If the values are equal, then yes, the target value is contained in the tree.
  - If you "run off the bottom" of the tree, then no, the target value is not contained in the tree.

The Binary Search Tree: Contains, *cont'd*

```
private boolean contains(AnyType x, BinaryNode<AnyType> t)
{
    if (t == null) return false;

    int compareResult = x.compareTo(t.element);

    if (compareResult < 0)  {
        return contains(x, t.left);
    }
    else if (compareResult > 0) {
        return contains(x, t.right);
    }
    else {
        return true; // Match
    }
}
```

The Binary Search Tree: Insert

- To insert a target value into the tree:
  - Proceed as if you are checking whether the tree contains the target value.
- As you're recursively examining left and right subtrees, if you encounter a null link (either a left link or a right link), then that's where the new value should be inserted.
  - Create a new node containing the target value and replace the null link with a link to the new node.
  - So the new node is attached to the last-visited node.

The Binary Search Tree: Insert, *cont'd*

- If the target value is already in the tree, either:
  - Insert a duplicate value into the tree.
  - Don't insert but "update" the existing node.

The Binary Search Tree: Insert

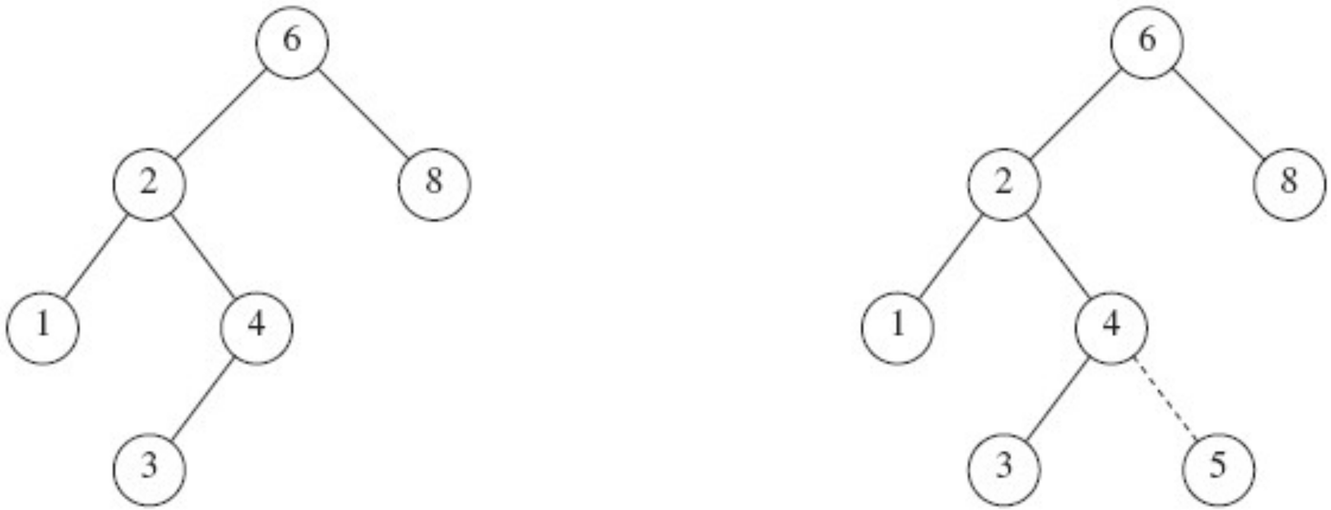


Figure 4.21 Binary search trees before and after inserting 5

The Binary Search Tree: Insert

```
private BinaryNode<AnyType> insert(AnyType x, BinaryNode<AnyType> t)
{
    // Create a new node to be attached
    // to the last-visited node.
    if (t == null) {
        return new BinaryNode<>(x, null, null);
    }

    int compareResult = x.compareTo(t.element);

    // Find the insertion point.
    if (compareResult < 0)  {
        t.left = insert(x, t.left);
    }
    else if (compareResult > 0) {
        t.right = insert(x, t.right);
    }
    else {
        // Duplicate: do nothing.
    }

    return t;
}
```

Only when a null link is encountered is a node created and returned.

The newly created node will be attached to the last-visited node.

Next class

- Binary Search Tree: Remove
- Fun with Binary Search Trees

The Binary Search Tree: Remove

- After removing a node from a binary search tree, the remaining nodes must still be in order.
- No child case: The target node to be removed is a leaf node.
  - Just remove the target node.

The Binary Search Tree: Remove, *cont'd*

- One child case: The target node to be removed has one child node.
  - Change the parent's link to the target node to point instead to the target node's child.

Figure 4.23 Deletion of a node (4) with one child, before and after

The Binary Search Tree: Remove, *cont'd*

- Two children case: The target node to be removed has two child nodes.
  - This is the complicated case.
- How do we restructure the tree so that the order of the node values is preserved?

The Binary Search Tree: Remove, *cont'd*

- Recall what happens you remove a list node.
  - Assume that the list is sorted.

    0 1 2 3 4 5 6 7 8 9

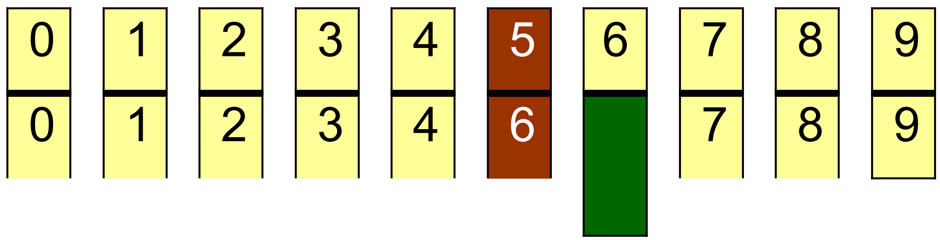
  - If we delete target node 5, which node takes its place?

    0 1 2 3 4 6 7 8 9

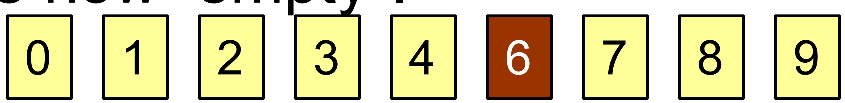
  - The replacement node is the node that is immediately after the target node in the sorted order.

The Binary Search Tree: Remove, *cont’d*

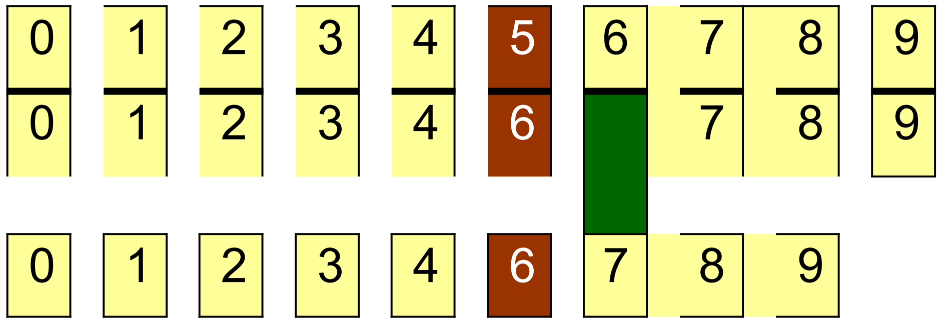
- A somewhat convoluted way to do this:
  - Replace the target node’s value with the successor node’s value.



  - Then remove the successor node, which is now “empty”.



The Binary Search Tree: Remove, *cont'd*



- The same convoluted process happens when you remove a node from a binary search tree.
  - The successor node is the node that is immediately after the deleted node in the sorted order.
  - Replace the target node's value with the successor node's value.
  - Remove the successor node, which is now "empty".

The Binary Search Tree: Remove, *cont'd*

- If you have a target node in a binary search tree, where is the node that is its immediate successor in the sort order?
  - The successor's value is ≥ than the target value.
  - It must be the minimum value in the right subtree.

- General idea:
  - Replace the value in the target node with the value of the successor node.
    - The successor node is now "empty".
  - Recursively delete the successor node.

The Binary Search Tree: Remove, *cont'd*

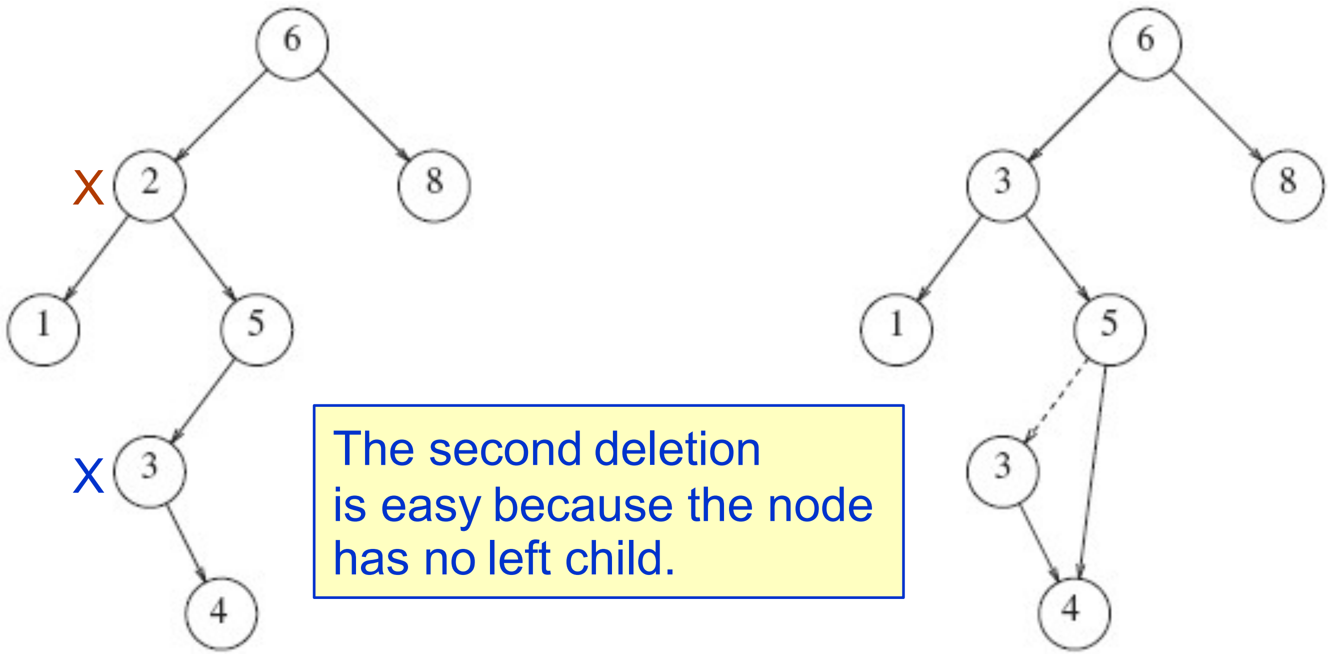


Figure 4.24 Deletion of a node (2) with two children, before and after

- Replace the value of the target node 2 with the value of the successor node 3.
- Now recursively remove node 3.

The Binary Search Tree: Remove, *cont'd*

```
private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> t)
{
    if (t == null) return t;

    int compareResult = x.compareTo(t.element);

    if (compareResult < 0) {
        t.left = remove(x, t.left);
    }
    else if (compareResult > 0) {
        t.right = remove(x, t.right);
    }

    else if (t.left != null && t.right != null) {
        t.element = findMin(t.right).element;
        t.right = remove(t.element, t.right);
    }

    else {
        t = (t.left != null) ? t.left : t.right;
    }

    return t;
}
```

Item not found: do nothing.

Two children:
Replace the target value with the successor value.
Then recursively remove the successor node.

No children or one child.

The Binary Search Tree Animations

- Download Java applets from http://www.informit.com/content/images/0672324539/downloads/ExamplePrograms.ZIP
  - These are from the book *Data Structures and Algorithms in Java, 2nd edition*, by Robert LaFlore: http://www.informit.com/store/data-structures-and-algorithms-in-java-9780672324536
- The binary search tree applet
- Run with the appletviewer application that is in your java/bin directory:

```
appletviewer Tree.html
```

Fun with Binary Search Trees

- Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one

Fun with Binary Search Trees

- Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height