CMPE 180-92

Data Structures and Algorithms in C++

November 16 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017

Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Assignment #12: Solution

- `Element`
- `Node`
- `InsertionSort::run_sort_algorithm()`
- `ShellSortSuboptimal::run_sort_algorithm()`
- `ShellSortOptimal::run_sort_algorithm()`
- `QuickSorter::quicksort()`
- `QuickSorter::partition()`
- `QuickSortSuboptimal::choose_pivot_strategy()`
- `QuickSortOptimal::choose_pivot_strategy()`
- `MergeSort::mergesort()`
- `MergeSort::merge()`
- `LinkedList::split()`
- `LinkedList::concatenate()`

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.

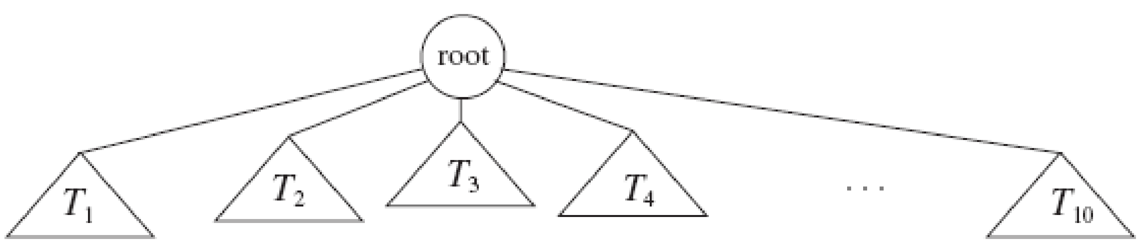


Figure 4.1 Generic tree

Trees, *cont'd*

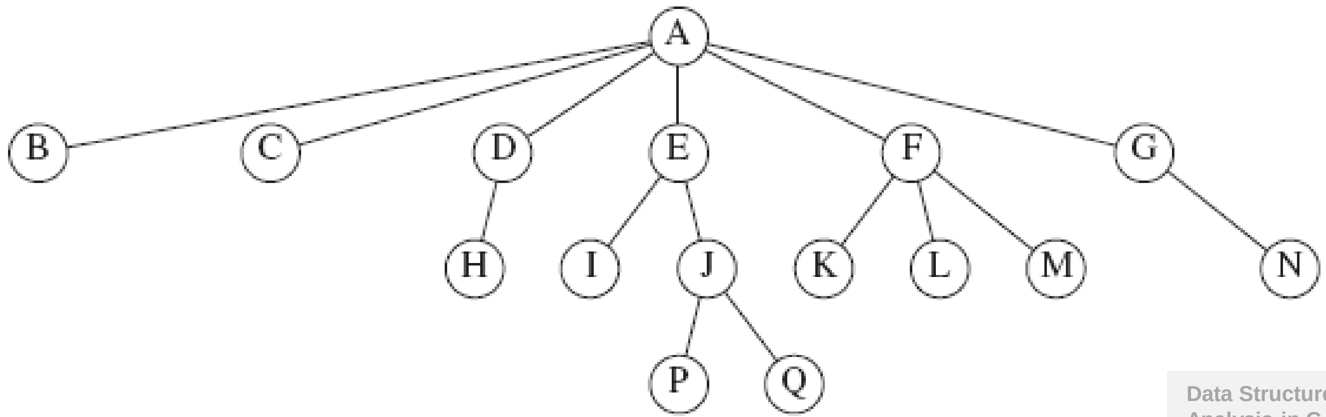


Figure 4.2 A tree

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

- 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*

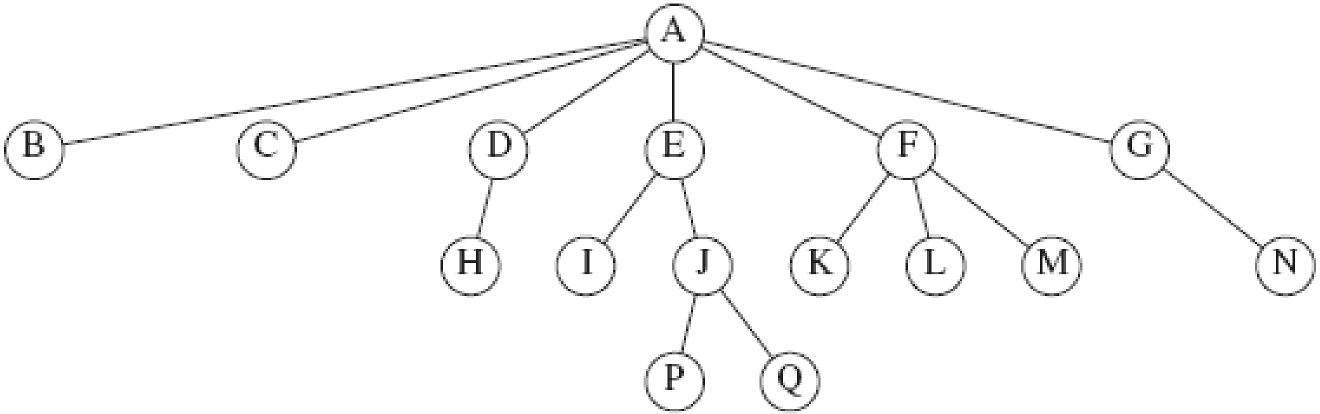


Figure 4.2 A tree

- 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?

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Trees, *cont'd*

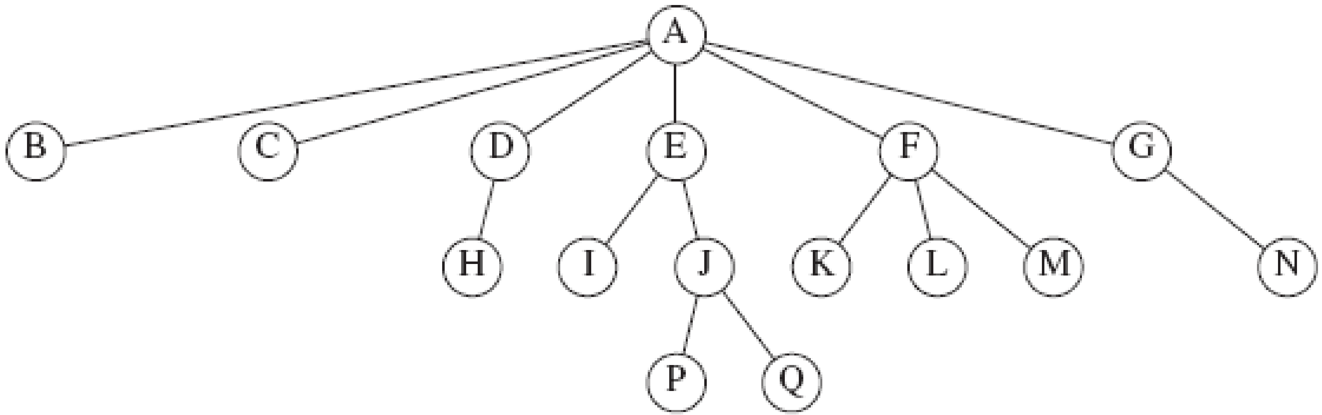


Figure 4.2 A tree

- 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:

```
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
}
```

Tree Implementation, *cont'd*

- Conceptual view of a tree:

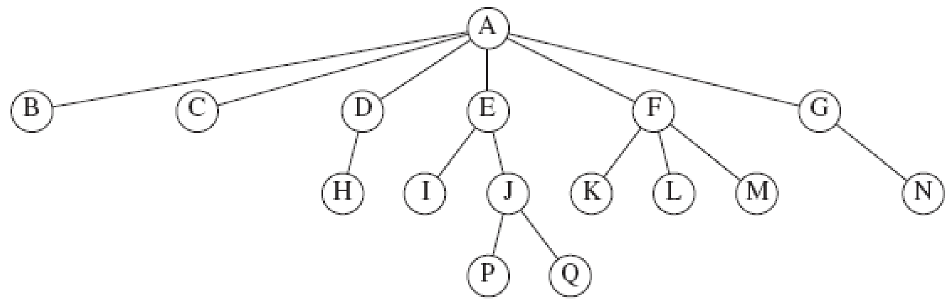


Figure 4.2 A tree

- Implementation view of the same tree:

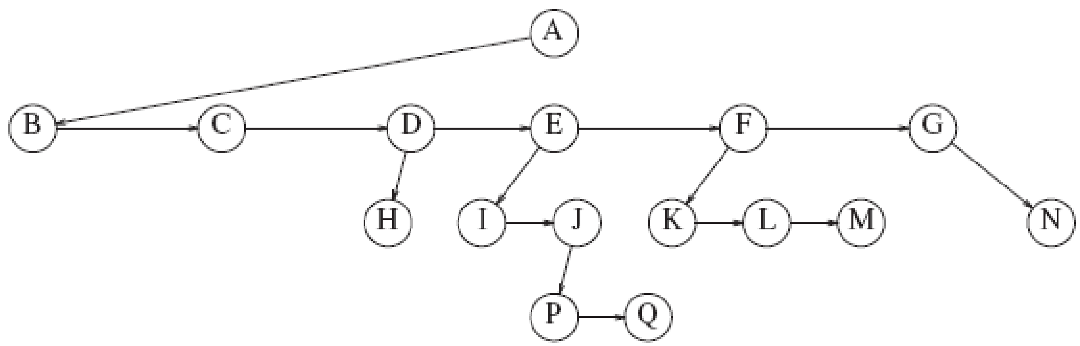


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

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*

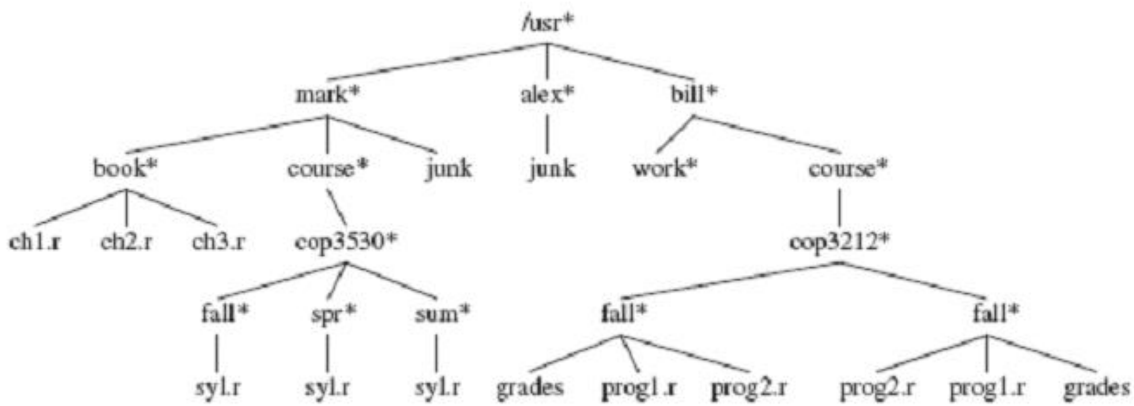


Figure 4.5 UNIX directory

```
void FileSystem::listAll(int depth = 0) const
{
    printName(depth);

    if (isDirectory())
    {
        for each file f in this directory
        {
            f.listAll(depth + 1);
        }
    }
}
```

```
/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 Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

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*

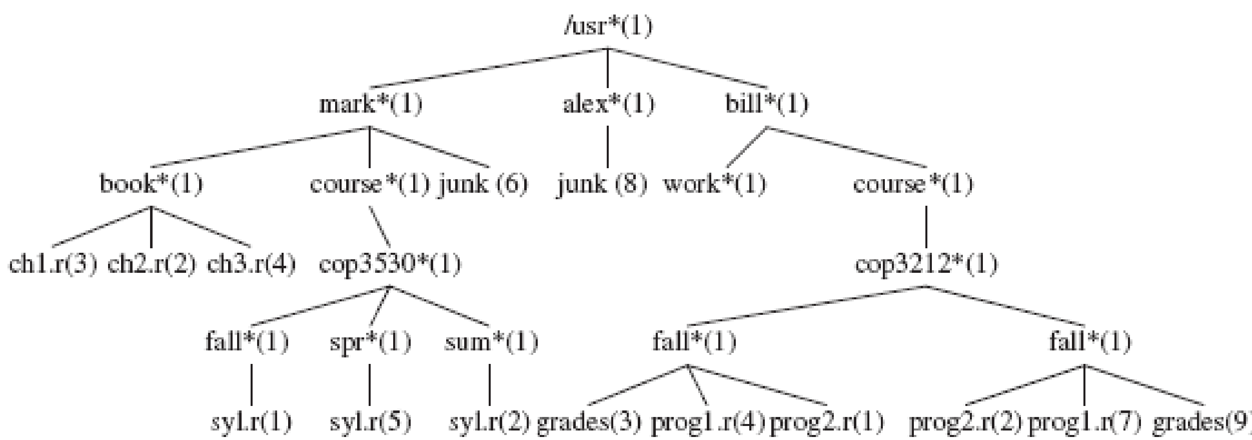


Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

```
int FileSystem::size() const
{
    int totalSize = sizeOfThisFile();

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

    return totalSize;
}
```

```
            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 Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Binary Trees

- A <u>binary tree</u> is a tree where each node can have <u>0, 1, or 2 child nodes</u>.

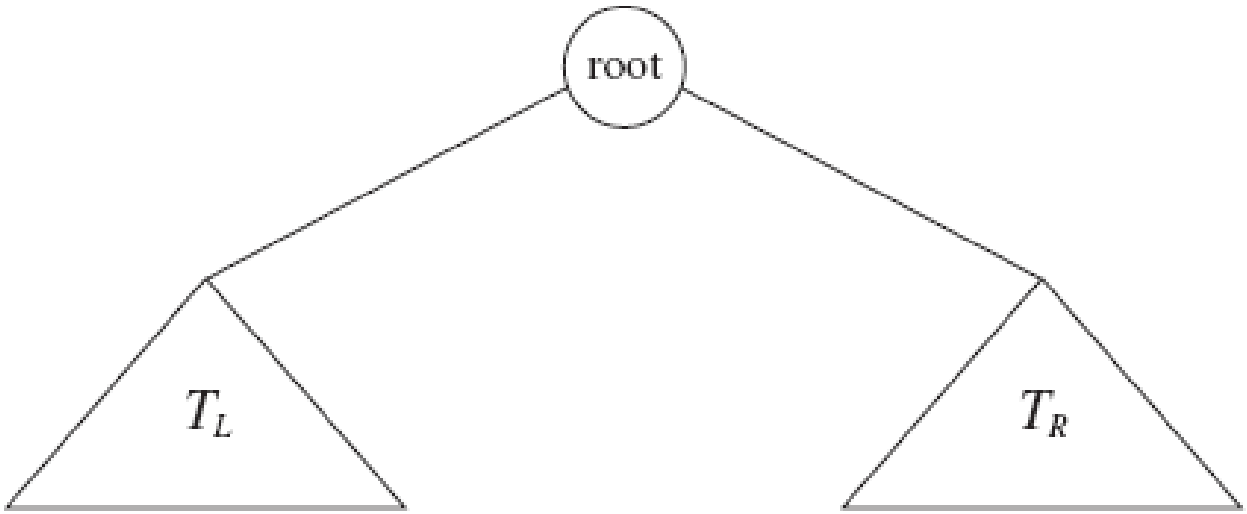


Figure 4.11 Generic binary tree

Binary Trees, *cont'd*

- An arithmetic expression tree:

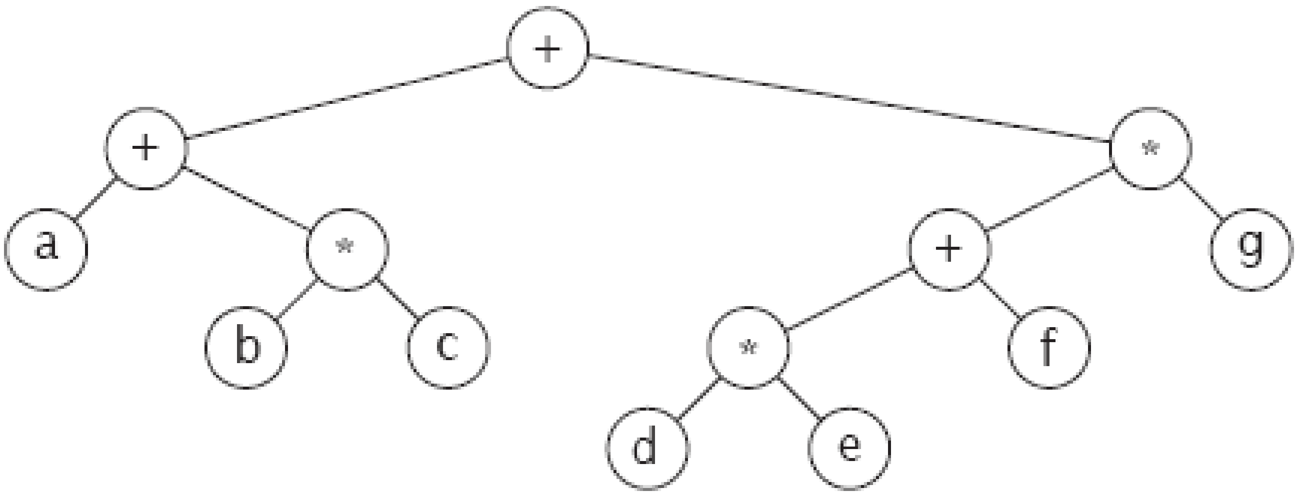


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

Conversion from Infix to Postfix Notation

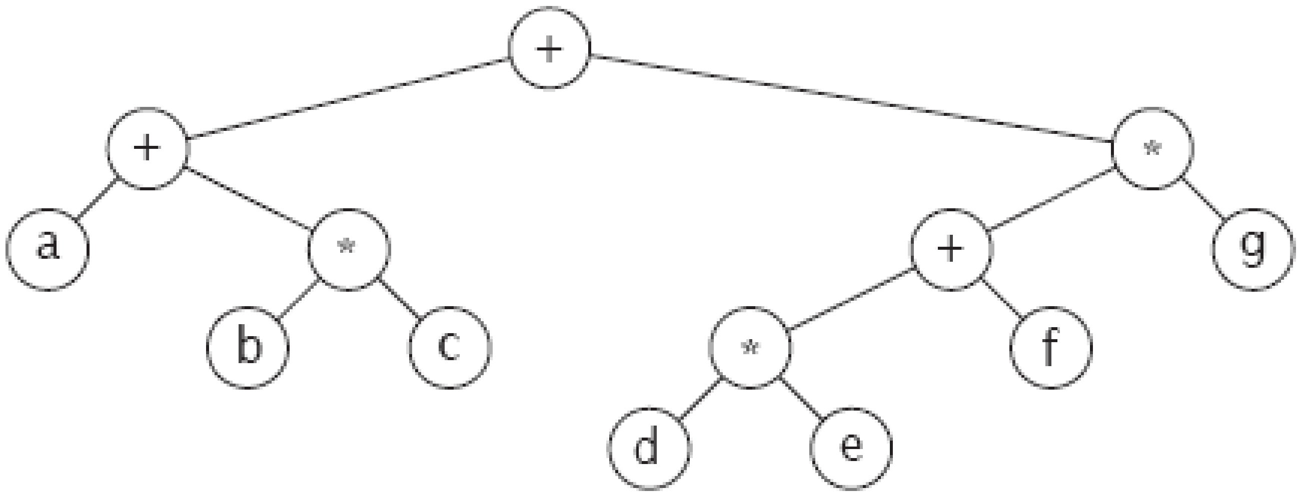


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- 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 (BST) has these properties for each of its nodes:
  - All the values in the node's left subtree are less than the value of the node itself.
  - All the values in the node's right subtree are 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.

```
template <class Comparable>
class BinaryNode
{
public:
    BinaryNode(Comparable data);
    BinaryNode(const Comparable& data, BinaryNode *left, BinaryNode *right);
    virtual ~BinaryNode();

    Comparable data;
    BinaryNode *left;
    BinaryNode *right;
};
```

The Binary Search Tree ADT, *cont'd*

```
template <typename Comparable>
class BinarySearchTree
{
public:
    BinarySearchTree();
    BinarySearchTree(const BinarySearchTree& rhs);
    virtual ~BinarySearchTree();

    BinarySearchTree& operator=(const BinarySearchTree& rhs);

    BinaryNode<Comparable> *getRoot() const;
    int height();
    const Comparable &findMin() const;
    const Comparable &findMax() const;

    void clear();
    bool isEmpty() const;
    bool contains(const Comparable& data) const;
    void insert(const Comparable data);
    void remove(const Comparable& data);

...
```

The Binary Search Tree ADT, *cont'd*

```
...

protected:
    virtual int height(BinaryNode<Comparable> *ptr);
    virtual void insert(const Comparable& data, BinaryNode<Comparable>* &ptr);
    virtual void remove(const Comparable& data, BinaryNode<Comparable>* &ptr);

private:
    BinaryNode<Comparable> *root;

    BinaryNode<Comparable> *findMin(BinaryNode<Comparable> *ptr) const;
    BinaryNode<Comparable> *findMax(BinaryNode<Comparable> *ptr) const;
    void clear(BinaryNode<Comparable>* &ptr);
    bool contains(const Comparable& data, BinaryNode<Comparable> *ptr) const;
};
```

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.
  - The minimum is the leftmost child.

```
template <typename Comparable>
BinaryNode<Comparable>
    *BinarySearchTree<Comparable>::findMin(BinaryNode<Comparable> *ptr) const
{
    if (ptr == nullptr)        return nullptr;
    if (ptr->left == nullptr) return node;

    return findMin(ptr->left);
}
```

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

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

```
template < template <typename Comparable>
BinaryNode<Comparable>
    *BinarySearchTree<Comparable>::findMax(BinaryNode<Comparable> *ptr) const
{
    if (ptr != nullptr)
    {
        while(ptr->right != nullptr) ptr = ptr->right;
    }

    return ptr;
}
```

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*

```
template <typename Comparable>
bool BinarySearchTree<Comparable>::contains(const Comparable& data,
                                            BinaryNode<Comparable> *ptr) const
{
    while (ptr != nullptr)
    {
        if (data < ptr->data)
        {
            ptr = ptr->left;
        }
        else if (data > ptr->data)
        {
            ptr = ptr->right;
        }
        else
        {
            return true;  // found
        }
    }

    return false;         // not found
}
```

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()`

```
template <typename Comparable>
void BinarySearchTree<Comparable>::insert(const Comparable& data,
                                          BinaryNode<Comparable>* &ptr)
{
    if (ptr == nullptr)
    {
        ptr = new BinaryNode<Comparable>(data);
    }
    else if (data < ptr->data)
    {
        insert(data, ptr->left);
    }
    else if (data > ptr->data)
    {
        insert(data, ptr->right);
    }
}
```

`ptr` passed by reference

Create a new node only when a null link is encountered.

Attach the newly created node to the last-visited node (pass the pointers by reference).

The Binary Search Tree: Remove

- After removing a node from a binary search tree, <u>the remaining nodes must still be in order</u>.

- <u>No child case</u>: 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.

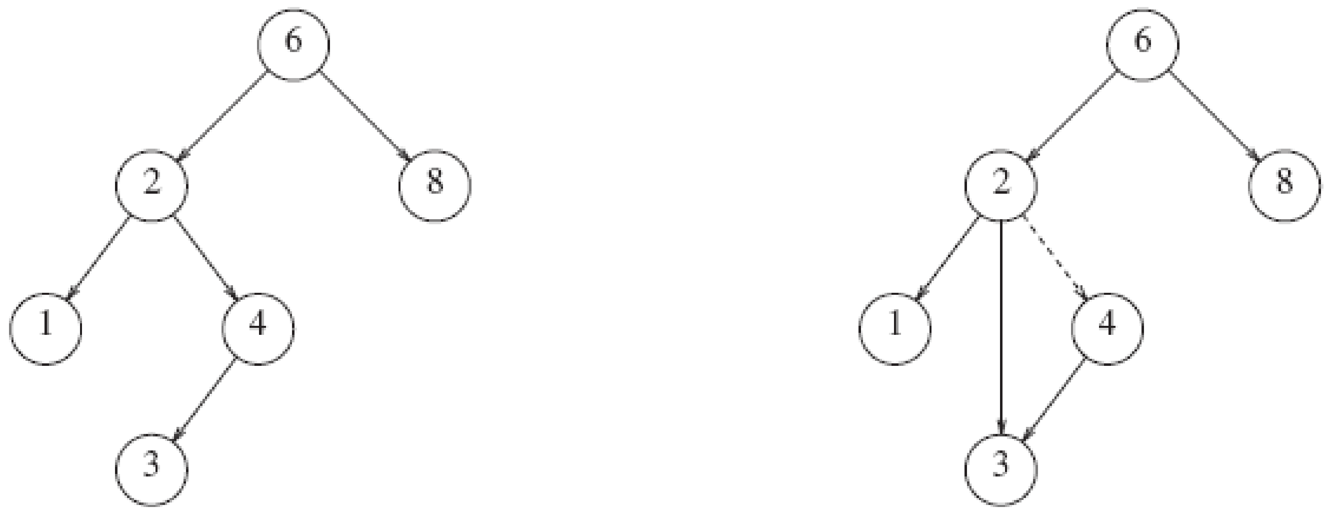


Figure 4.23 Deletion of a node (4) with one child, before and after

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.

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	6		7	8	9

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

0	1	2	3	4	6	7	8	9

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

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	6		7	8	9
0	1	2	3	4	6	7	8	9	

- 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 <u>immediate successor</u> in the sort order?
  - The successor's value is ≥ than the target value.
  - It must be the <u>minimum value in the right subtree</u>.

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

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

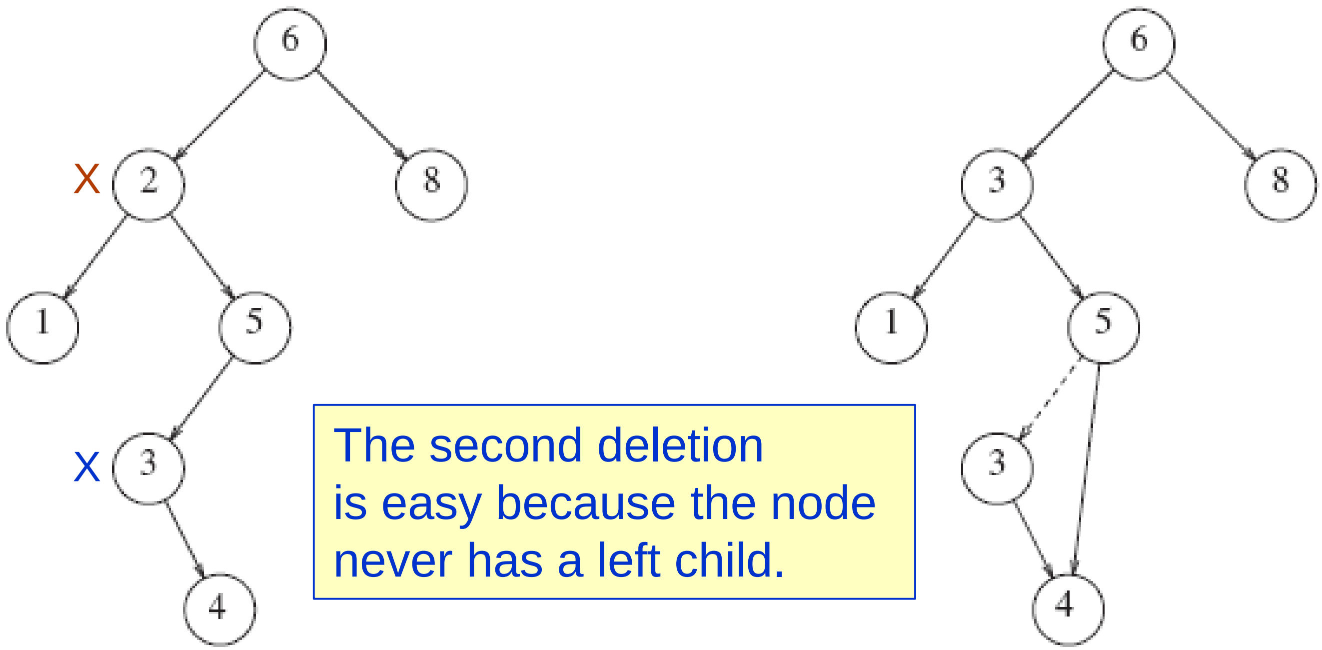


Figure 4.24 Deletion of a node (2) with two children, before and after

- 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*

```
template <typename Comparable>
void BinarySearchTree<Comparable>::remove(const Comparable& data,
                                          BinaryNode<Comparable>* &ptr)
{
    if (ptr == nullptr) return;

    if (data < ptr->data)
    {
        remove(data, ptr->left);
    }
    else if (data > ptr->data)
    {
        remove(data, ptr->right);
    }
    else if (   (ptr->left  != nullptr)
             && (ptr->right != nullptr))
    {
        ptr->data = findMin(ptr->right)->data;
        remove(ptr->data, ptr->right);
    }
    else
    {
        BinaryNode<Comparable> *oldNode = ptr;
        ptr = (ptr->left != nullptr) ? ptr->left
                                     : ptr->right;
        delete oldNode;
    }
}
```

`ptr` passed by reference

Item not found: do nothing.

Search left.

Search right.

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 is in Chap08/Tree
- Run with the appletviewer application that is in your java/bin directory:

```
appletviewer Tree.html
```

Break

AVL Trees

- An AVL tree is a binary search tree (BST) with a balance condition.
  - Named after its inventors, Adelson-Velskii and Landis.
- For each node of the BST, the heights of its left and right subtrees can <u>differ by at most 1</u>.
  - Remember that the height of a tree is the length of the longest path from the root to a leaf.
  - The height of the root = the height of the tree.
  - <u>The height of an empty tree is -1.</u>

AVL Trees, *cont'd*

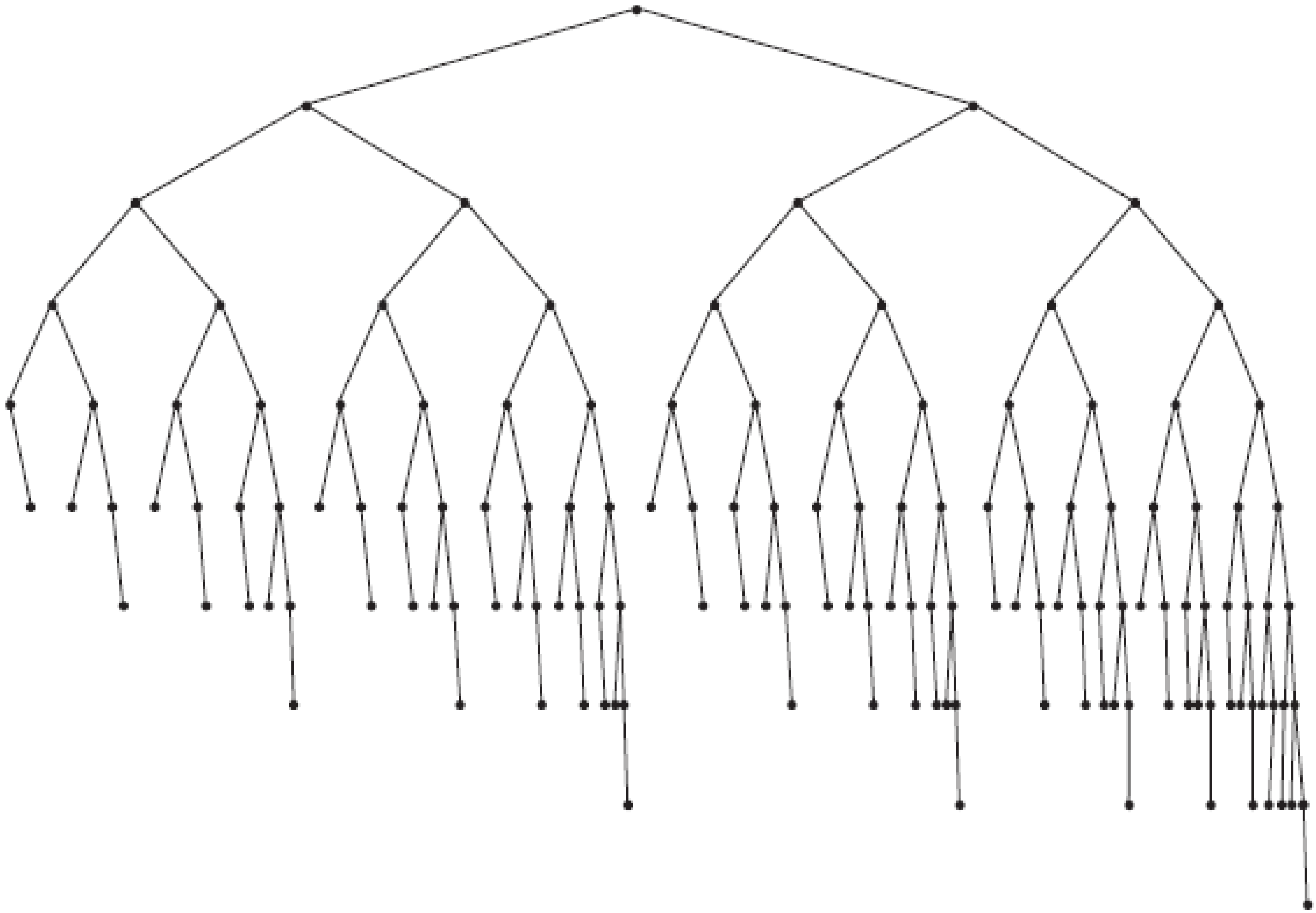


Figure 4.30 Smallest AVL tree of height 9

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Balancing AVL Trees

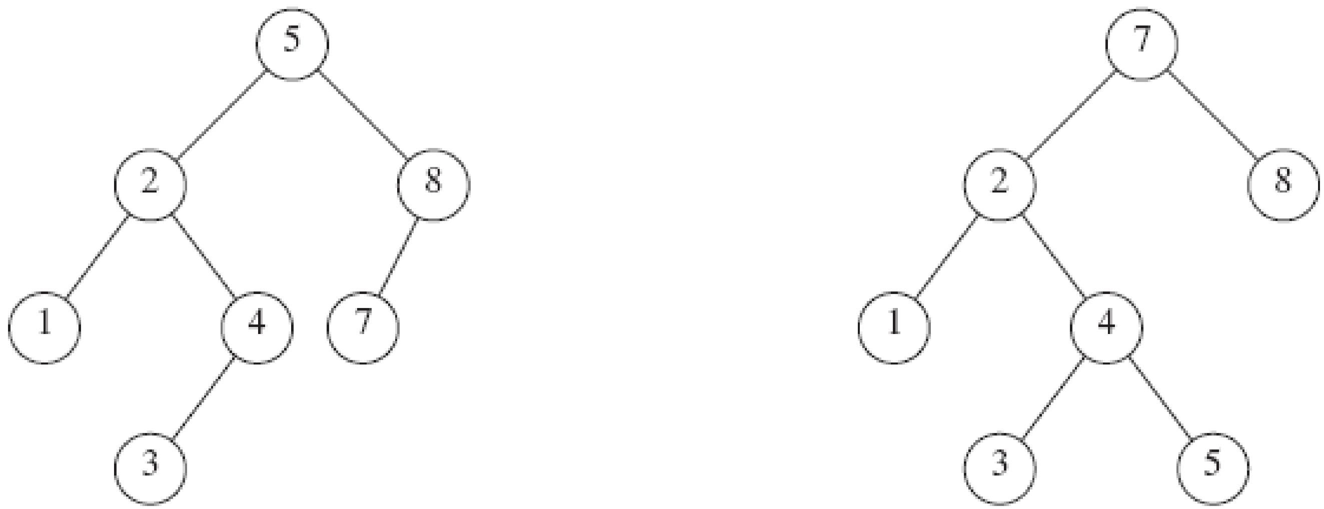


Figure 4.29 Two binary search trees. Only the left tree is AVL

- We need to <u>rebalance the tree</u> whenever the balance condition is violated.
  - Check after every insertion and deletion.

Balancing AVL Trees, *cont'd*

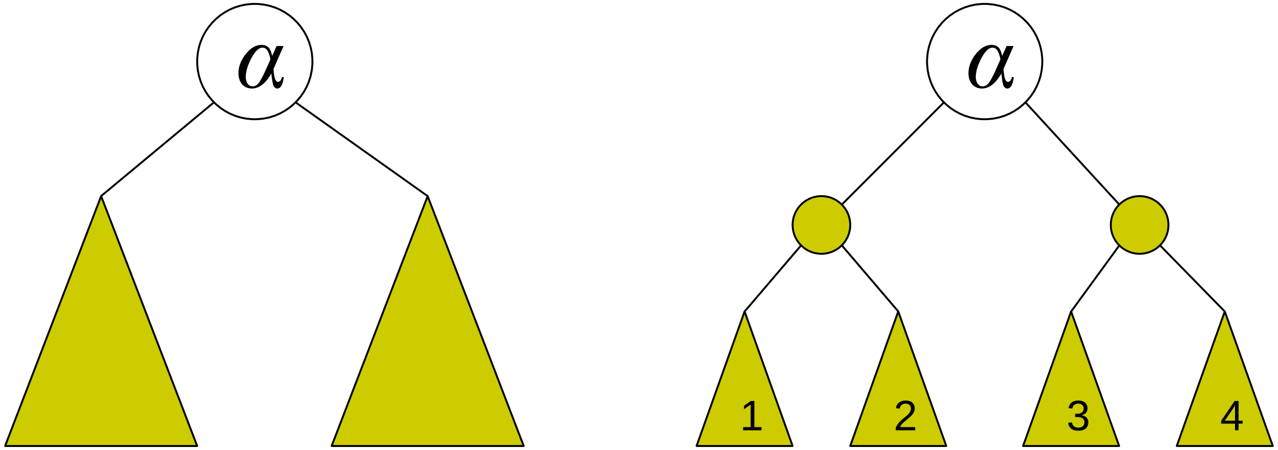
- Assume the tree was <u>balanced before</u> an insertion.

- If it became unbalanced due to the insertion, then the inserted node must have caused some nodes between itself and the root to be unbalanced.

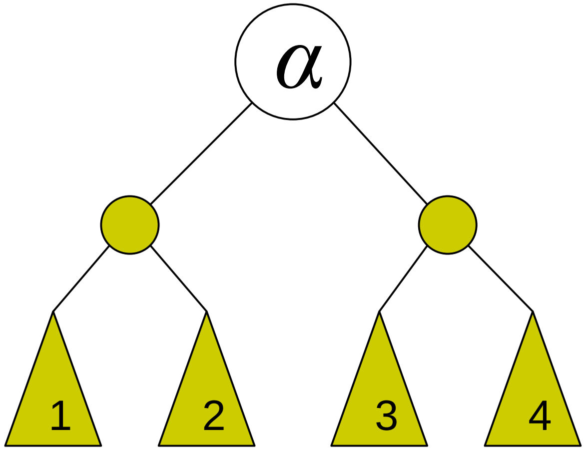
- An unbalanced node must have the height of one of its subtrees <u>exactly 2 greater</u> than the height its other subtree.

Balancing AVL Trees, *cont'd*

- Let the deepest unbalanced node be α.
- Any node has at most two children.
- A new height imbalance means that the heights of α's two subtrees now differ by 2.



Balancing AVL Trees, *cont'd*



- Therefore, one of the following had to occur:
  - Case 1 (outside left-left): The insertion was into the left subtree of the left child of α.
  - Case 2 (inside left-right): The insertion was into the right subtree of the left child of α.
  - Case 3 (inside right-left): The insertion was into the left subtree of the right child of α.
  - Case 4 (outside right-right): The insertion was into the right subtree of the right child of α.

Cases 1 and 4 are mirrors of each other, and cases 2 and 3 are mirrors of each other.

Balancing AVL Trees: Case 1

- Case 1 (outside left-left):
  Rebalance with a single right rotation.

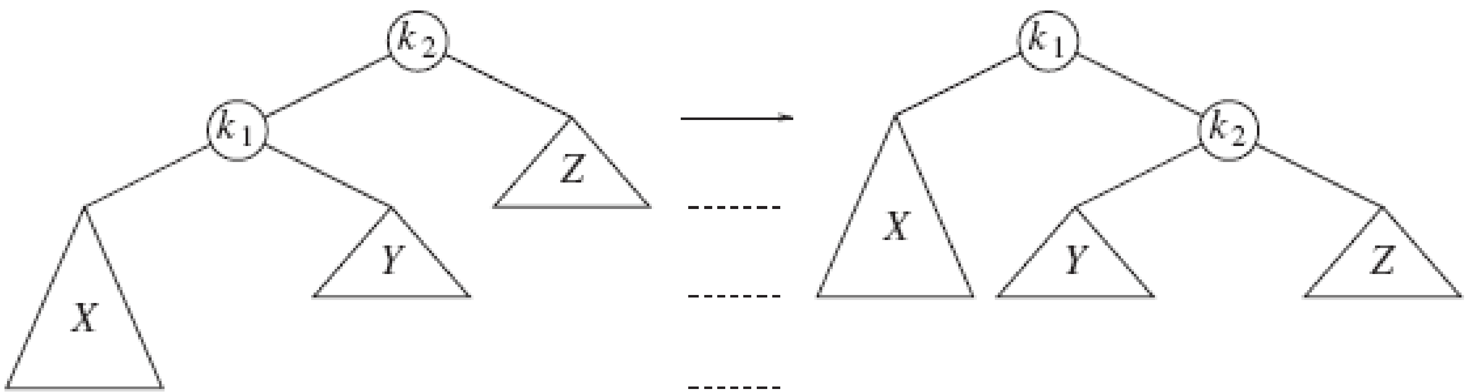
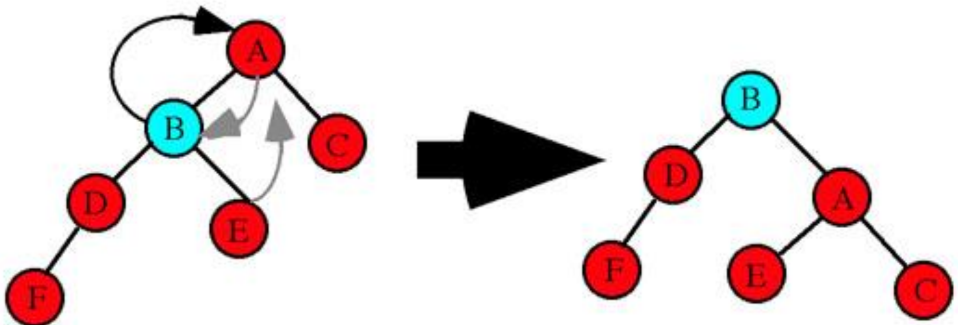


Figure 4.31 Single rotation to fix case 1

Balancing AVL Trees: Case 1, *cont'd*

- Case 1 (outside left-left):
  Rebalance with a single right rotation.



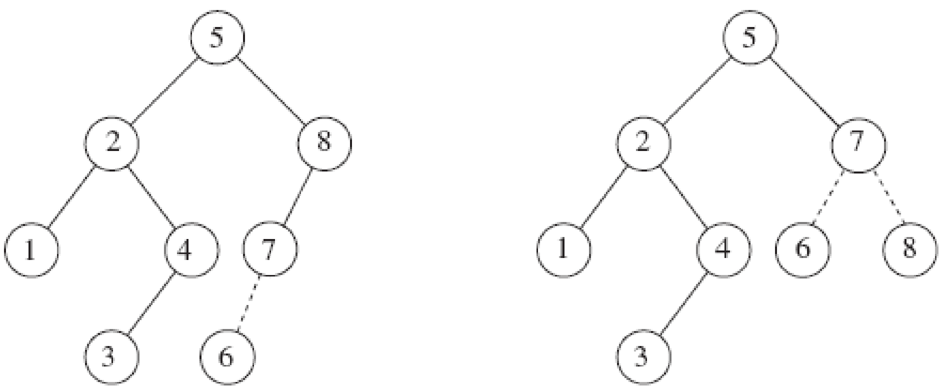


Figure 4.32 AVL property destroyed by insertion of 6, then fixed by a single rotation

Node A is unbalanced.
Single right rotation: A's left child B becomes the new root of the subtree.
Node A becomes the right child and adopts B's right child as its new left child.

Node 8 is unbalanced.
Single right rotation: 8's left child 7 becomes the new root of the subtree.
Node 8 is the right child.

http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.html

Balancing AVL Trees: Case 4

- <u>Case 4</u> (outside right-right):
  Rebalance with a <u>single left rotation</u>.

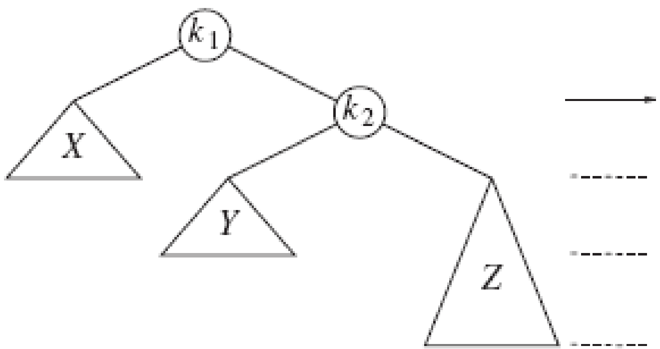
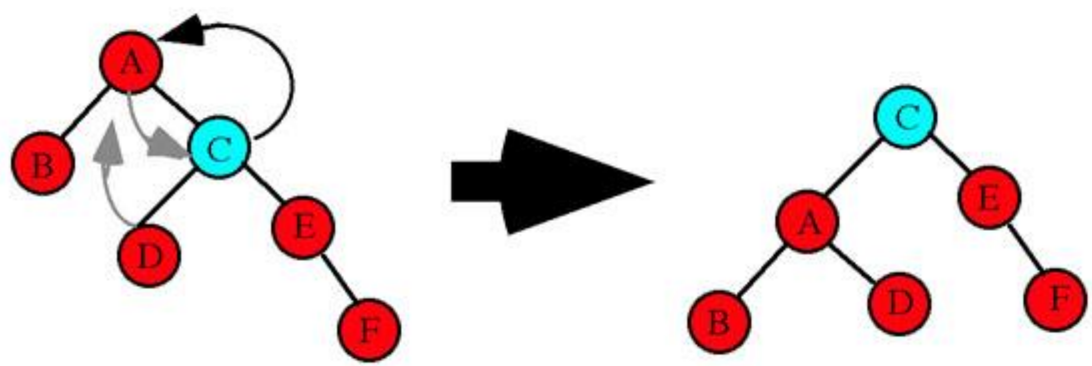


Figure 4.33 Single rotation fixes case 4

Data Structures and Algorithm Analysis in C++, 4th ed.
by Mark Allen Weiss
Pearson Education, Inc., 2014

Node A is unbalanced.
Single left rotation: A's right child C becomes the new root of the subtree.
Node A becomes the left child and adopts C's left child as its new right child.



http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.html

Balancing AVL Trees: Case 2

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.

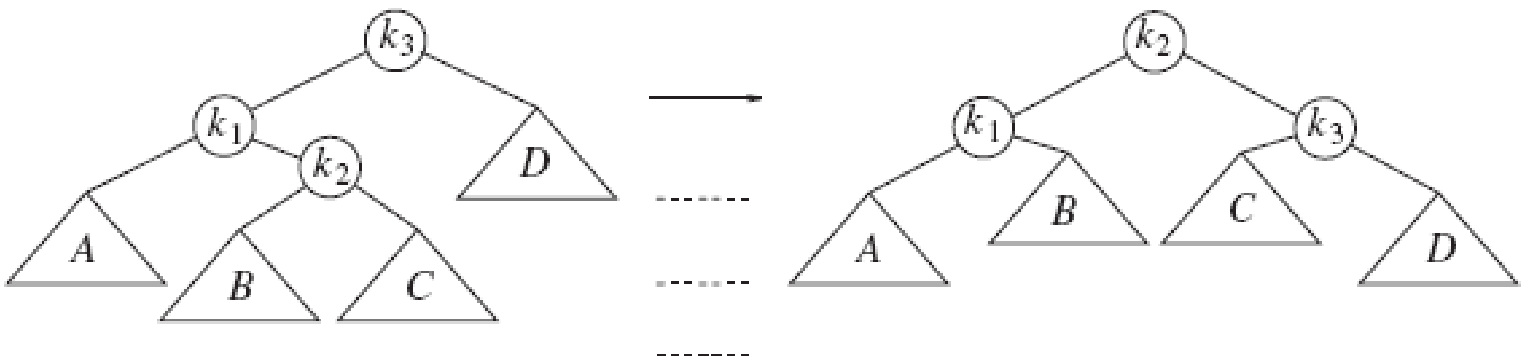
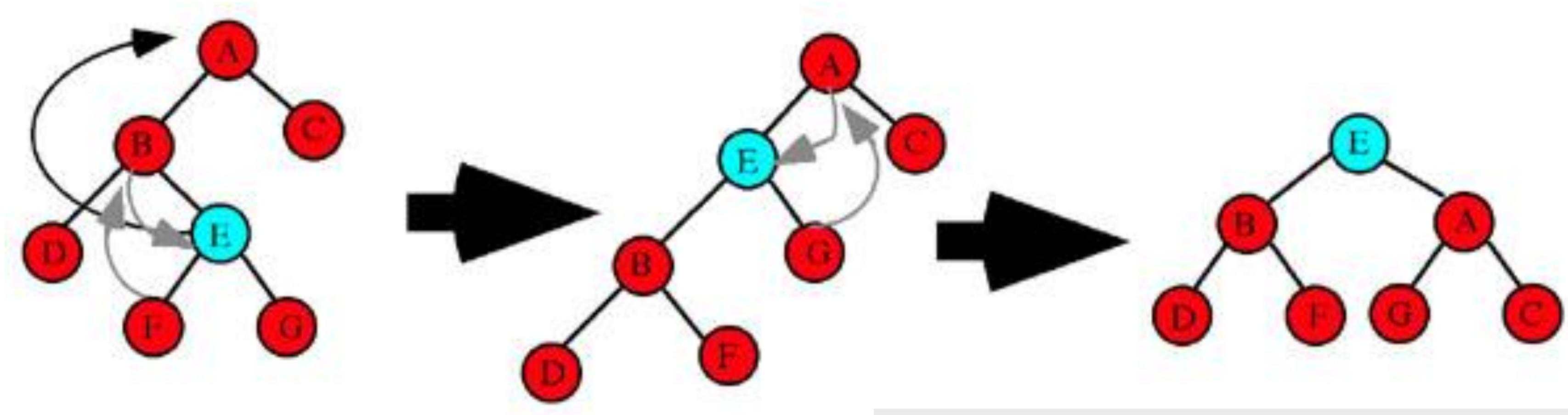


Figure 4.35 Left–right double rotation to fix case 2

Balancing AVL Trees: Case 2, *cont'd*

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.



http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.html

Node A is unbalanced.
Double left-right rotation: E becomes the new root of the subtree after two rotations. Step 1 is a single left rotation between B and E. E replaces B as the subtree root. B becomes E's left child and B adopts E's left child F as its new right child. Step 2 is a single right rotation between E and A. E replaces A is the subtree root. A becomes E's right child and A adopts E's right child G as its new left child.

Balancing AVL Trees: Case 3

- <u>Case 3</u> (inside right-left):
  Rebalance with a <u>double right-left rotation</u>.

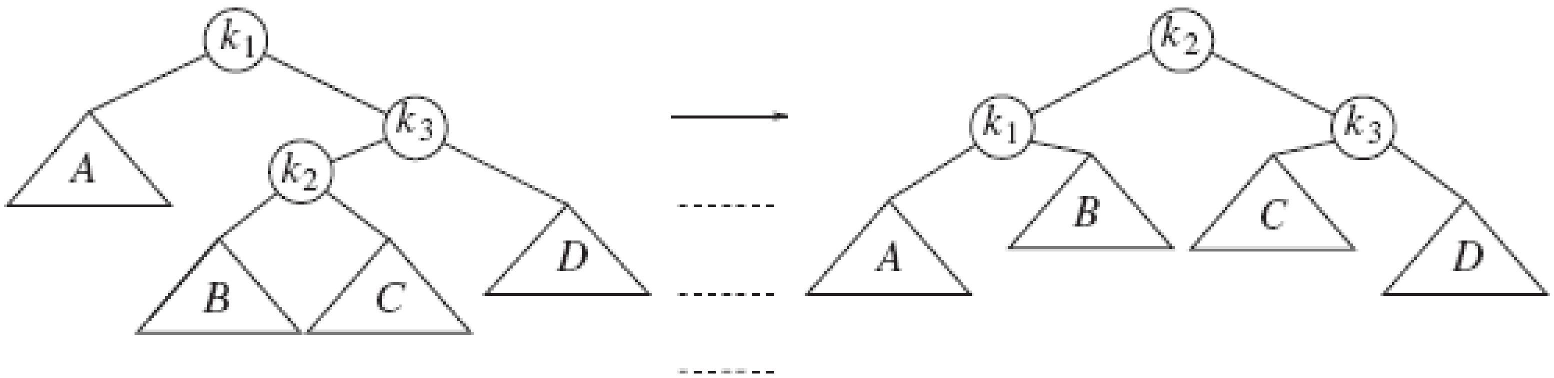
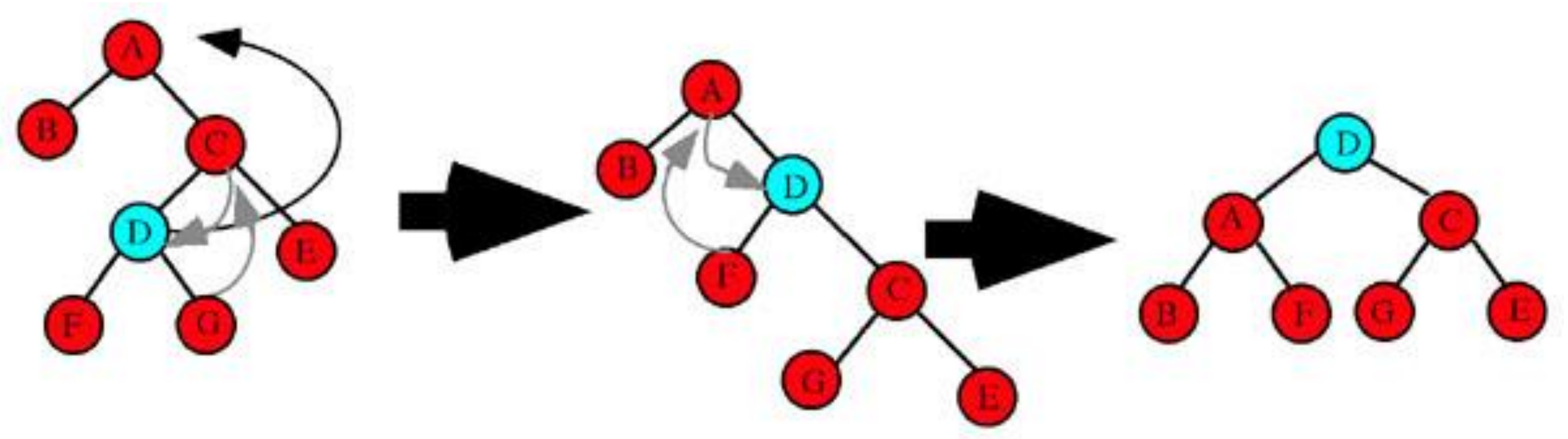


Figure 4.36 Right–left double rotation to fix case 3

Balancing AVL Trees: Case 3, *cont'd*

http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.html

- Case 3 (inside right-left):
  Rebalance with a double right-left rotation.



Node A is unbalanced.
Double right-left rotation: D becomes the new root of the subtree after two rotations. Step 1 is a single right rotation between C and C. D replaces C as the subtree root. C becomes D's right child and C adopts D's right child G as its new left child. Step 2 is a single left rotation between D and A. D replaces A is the subtree root. A becomes D's left child and A adopts D's left child F as its new right child.

AVL Tree Implementation

- Since an AVL tree is just a BST with a balance condition, it makes sense to make the AVL tree class a <u>subclass of the BST class</u>.

```
template <class Comparable>
class AvlTree : public BinarySearchTree<Comparable>
```

- Both classes can share the same **`BinaryNode`** class.

The AVL Tree Node

- With so many height calculations, it makes sense to store each node's height in the node itself.

```
template <class Comparable>
class BinaryNode
{
public:
    BinaryNode(Comparable data);
    BinaryNode(const Comparable& data, BinaryNode *left, BinaryNode *right);
    virtual ~BinaryNode();

    Comparable data;
    int        height;  // node height

    BinaryNode *left;
    BinaryNode *right;
};
```

AVL Tree Implementation, *cont'd*

- Class **`AVLTree`** overrides the **`insert()`** and **`remove()`** methods of class **`BinarySearchTree`**.
  - Each method calls the superclass's method and then passes the node to the **`balance()`** method.

```
template <class Comparable>
void AvlTree<Comparable>::insert(const Comparable& data, BinaryNode<Comparable>* &ptr)
{
    BinarySearchTree<Comparable>::insert(data, ptr);
    balance(ptr);
}
```

```
template <class Comparable>
void AvlTree<Comparable>::remove(const Comparable& data, BinaryNode<Comparable>* & ptr)
{
    BinarySearchTree<Comparable>::remove(data, ptr);
    balance(ptr);
}
```

AVL Tree Implementation, *cont'd*

- The private `AVLTree` method `balance()` checks whether the balance condition still holds, and <u>rebalances the tree</u> with rotations whenever necessary.

AVL Tree Implementation, *cont'd*

```
template <class Comparable>
BinaryNode<Comparable> *AvlTree<Comparable>::balance(BinaryNode<Comparable>* &ptr)
{
    if (ptr == nullptr) return ptr;

    // Left side too high.
    if (height(ptr->left) - height(ptr->right) > 1)
    {
        if (height(ptr->left->left)
                >= height(ptr->left->right))
        {
            ptr = singleRightRotation(ptr);
            cout << "    --- Single right rotation at "
                 << ptr->data << endl;
        }
        else
        {
            ptr = doubleLeftRightRotation(ptr);
            cout << "    --- Double left-right rotation at "
                 << ptr->data << endl;
        }
    }

    ...
```

Case 1

Case 2

AVL Tree Implementation, *cont'd*

```
    ...

    // Right side too high.
    else if (height(ptr->right) - height(ptr->left) > 1)
    {
        if (height(ptr->right->right)
                >= height(ptr->right->left))
        {
            ptr = singleLeftRotation(ptr);
            cout << "    --- Single left rotation at "
                 << ptr->data << endl;
        }
        else
        {
            ptr = doubleRightLeftRotation(ptr);
            cout << "    --- Double right-left rotation at "
                 << ptr->data << endl;
        }
    }
    // Recompute the node's height.
    node->height = (max(height(node->left),
                        height(node->right)) + 1);
    return node;
}
```

Case 4

Case 3

Assignment #13

- This assignment will give you practice with binary search trees (BST) and AVL trees.
- You are provided a **`TreePrinter`** class that has a **`print()`** method that will print any arbitrary binary tree.
  - A template for how it prints a tree:

```
                                                              xx
                                                              /\
                                ------------------------------  ------------------------------
                               /                                                              \
                              xx                                                              xx
                              /\                                                              /\
                ------------  --------------                                  --------------  --------------
               /                              \                                /                              \
              xx                              xx                              xx                              xx
              /\                              /\                              /\                              /\
        ------  ------                  ------  ------                  ------  ------                  ------  ------
       /              \                /              \                /              \                /              \
      xx              xx              xx              xx              xx              xx              xx              xx
      /\              /\              /\              /\              /\              /\              /\              /\
    --  --          --  --          --  --          --  --          --  --          --  --          --  --          --  --
   /      \        /      \        /      \        /      \        /      \        /      \        /      \        /      \
  xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx      xx
  /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\      /\
 /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \    /  \
xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx
```

Assignment #13, *cont’d*

- **`TreePrinter`** is able to print trees with height up to 5, *i.e.*, 32 node values on the bottom row.
  - An example of an actual printed tree:

```
                            24
                            /\
              --------------  --------------
             /                              \
            18                              73
            /\                              /\
      ------  ------                  ------  ------
     /              \                /              \
    12              19              38              87
    /                               /\               \
  --                              --  --              --
 /                               /      \               \
10                              37      41              90
                                         \
                                          \
                                          64
```

Assignment #13: Part 1

- The first part of the assignment makes sure that you can successfully insert nodes into, and delete nodes from, a binary search tree (BST) and an AVL tree.

Assignment #13: Part 1, *cont'd*

- First create a BST, node by node.
  - You will be provided the sequence of values to insert into the tree.
  - Print the tree after each insertion.
  - The tree will be unbalanced.
- Now repeatedly delete the root of the tree.
  - Print the tree after each deletion.
  - Stop when the tree becomes empty.

Assignment #13: Part 1, *cont'd*

- Second, create an AVL tree, node by node.
  - Insert the same given sequence of values.
  - Print the tree after each insertion to verify that you are keeping it balanced.
  - Each time you do a rebalancing, print a message indicating which rotation operation(s) at which node.
    - Example:

```
Inserted node 10:
    --- Single right rotation at node 21
```

- As you did with the BST, repeatedly delete the root of your AVL tree.
  - Print the tree after each deletion to verify that you are keeping it balanced.

Assignment #13: Part 1, *cont'd*

- A handy AVL tree balance checker:

```
template <class Comparable>
int AvlTree<Comparable>::checkBalance(BinaryNode<Comparable> *ptr)
{
    if (ptr == nullptr) return -1;

    int leftHeight  = checkBalance(ptr->left);
    int rightHeight = checkBalance(ptr->right);

    if ((abs(height(ptr->left) - height(ptr->right)) > 1)
        || (height(ptr->left)  != leftHeight)
        || (height(ptr->right) != rightHeight))
    {
        return -2;       // unbalanced
    }

    return height(ptr);  // balanced
}
```

Assignment #13: Part 2

- The second part of the assignment compares the performance of a BST vs. an AVL tree.

- First, generate n random integers.
  - n is some large number, to be explained.

- Insert the random integers one at a time into the BST and AVL trees.

Assignment #13: Part 2, *cont'd*

- For each tree, collect the following statistics:
  - Probe counts
    - A probe is whenever you visit a tree node, even if you don't do anything with the node other than use its left or right link to go to a child node.
  - Comparison counts
    - A comparison is a probe where you also check the node's value.
  - Elapsed time in milliseconds
- Do <u>not</u> print the tree after each insertion.
- Be sure to count probes and comparisons during AVL tree rotations.

Assignment #13: Part 2, *cont'd*

- Second, generate another *n* random integer values.
- Search the BST and AVL trees for the values, one at a time.
  - Count probes and comparisons and compute elapsed times.
  - It doesn't matter whether or not a search succeeds

Assignment #13: Part 2, *cont'd*

- Choose values of n large enough to give you consistent timings that you can compare.
  - Try values of n = 10,000 to 100,000 in increments of 10,000
  - Slower machines can use a different range of values for n.

Assignment #13: Part 2, *cont'd*

- Print tables of these statistics for insertion and search with BST and AVL trees as comma-separated values.
- Use Excel to create the following graphs, each one containing plots for BST and AVL:
  - insertion probe counts
  - insertion compare counts
  - insertion elapsed time
  - search probe counts
  - search compare counts
  - search elapsed time

Assignment #13, *cont'd*

- Do Part 1 in CodeCheck.
  - CodeCheck will check your output.
- Do Part 2 outside of CodeCheck.
- You can use any code from the lectures or from the textbook or from the Web.
- Be sure to give <u>proper citations</u> if you use code that you didn't write yourself.
  - Names of books, URLs, etc.
  - Put the citations in your program comments.