Some Comments about HW2:

1. You should have used a generic node in your structure – one that expected an Object, and not some other type.
2. Main is still too long for some people
3. braces in wrong place, lines too long, meaningless variable names, poor indentation – read and following the coding standard
4. End of Block comments. I know the book uses these, but I think that they are just clutter. You don't need them if you keep your methods and blocks short.
5. Commented out code – don't include it. Remove it before submitting your program.

Midterm Regrade

I found a lot of problems with the midterm grading. I went over all of the exams that I had. Many people got 1 or 2 extra points. Some people got 4 or 5. A small number lost credit. After the regrade, the class average went up by nearly 1 point.

If you took your exam home last time, I will still regrade it if you want.

Reading

There is a new reading assignment on the course web site

Partners

There seems to be a number of people who need new partners. Please come up after class if you are one of these people.

Lecture

We'll finish up with recursion today, and then start with Trees. The Recursion notes are in a separate file.

Trees

We've seen recursion and the power of divide-and-conquer
• split a problem into smaller parts.
• solve each of the parts separately: easier to code and understand.

We want to apply these techniques to storing data so that it is
• ordered
• easy and efficient to find

List-type structures don't do both
• Sorted Lists and arrays are ordered, but lookup is slow.

We want a structure that can do both
**problem:** In this class, who has the mth best grade?
We could use a sorted linked list:
- can find mth best by following links
Have to follow links to find place to insert or remove.

We could use a sorted array:
- Easy to find mth best
- Difficult to add or remove – must search along list to find insertion point, and then need to move other data out of the way.

What if we want to find the student whose score was 77?
- we don't know the index, and we can't just count.
- Have to search the array or list to find it
- \( O(n) \)

There's a better way: Binary Search!
- With array, we just look at the middle item, and keep looking in the correct half. We can get to any element in constant time, so the search is \( O(\log n) \).
- But, insert and delete are difficult.

- With a linked list, how do we get to the middle item? Can't get there directly. Long lists --> long time to find the middle.
- But easy to insert and delete.

How could we improve on the linked list?
- Keep references to the center? and the \( \frac{1}{4} \) points? And so on?
- What if we add and delete elements? What happens to our references?
- What if list has 1 million elements? How do we know how many references to maintain?
- So, this idea, while a good one, doesn't scale well

We need a new structure that uses links, but is easy to do binary search on.

**Solution**: Trees

A tree is a linked data structure where each node may have more than one 'next' node.

- "next" of a node is its child
- "prev" of a node is its parent
- Base of the tree is called the root – this is the only node with no parent.
- a node with no children is called a leaf.
- Nodes along the path to the root are ancestors.
- Nodes along the path to a leaf are descendents
- Nodes with the same parent are siblings.
- The height of tree is the length of the longest path from the root to a leaf.
- Subtree of a node n – a tree that consists of a child of n plus all of that child's
descendents.

*Binary Tree* – a special case of a tree, where each node may have at most two children. The children are called the left and right children, and they are in turn the root nodes of the left and right subtrees.

*Binary Search Tree* – A binary tree where the value in every node \( n \) is greater than the value in every node in its left subtree, but less than the value in every node in its right subtree.

*Empty binary tree* – a binary tree with no nodes

*Full binary tree* – A binary tree of height \( h \) with no missing nodes. All leaves are at level \( h \), and all other nodes have 2 children.

*Balanced binary tree* – a binary tree whose leaves are at most \( h \) apart. For each node, the height of the left and right subtrees differ by at most 1.

*Balanced binary tree* – a binary tree that is full to level \( h-1 \) and that has level \( h \) filled in from left to right.

**Why use a Tree?**

It has the advantages of both lists and arrays:
- easy to insert and delete nodes
- can grow to any size
- easy to keep sorted
- easy to do binary search
- lookup can be done quickly if tree is kept sorted

Disadvantages of trees
- More references to use space and keep track of
- Tree can grow unbalanced, resulting in reduced performance – worst case tree is like a linked list
Comments:
- New programming assignments on web site.
- Written assignment on-line: due next Thursday.
- Browser program – your program does not have to match my program as far as its defects are concerned.

What classes do we need to build a binary tree?

As with lists, there are two classes:
- A node class: TreeNode, for each node in the tree
- A tree class: BinaryTree, for each subtree rooted at a particular TreeNode

TreeNode objects support the expected operations:
- TreeNode( Object newItem );
- TreeNode( Object newItem, TreeNode left, TreeNode right )
- Object getItem()
- void SetItem( Object item )
- TreeNode getLeft()
- void setLeft( TreeNode left )
- TreeNode getRight()
- void setRight( TreeNode right )

These operations are straightforward, and similar to those in the linked list Node.

**Methods to build BinaryTrees:**
- BinaryTree() - creates an empty tree
- BinaryTree( Object item ) - creates a tree with a root
- BinaryTree( Object item, BinaryTree leftTree, BinaryTree rightTree ) - creates a tree with a root and left and right subtree

Attach things to an already existing BinaryTree
- attachLeft/Right( Object newItem )
- attachLeft/RightSubtree( BinaryTree subtree )
  - attachLeft/Right could be done by creating a new BinaryTree and using attachLeft/RightSubtree

Exceptions for
- trying to attach something that already has a subtree there

Sometimes it is necessary to take trees apart
- to delete an item
- to rebalance the tree

BinaryTree detachLeft/RightSubtree() - returns subtree and sets the reference in the node
to null

**Informational methods:**
Object getRootItem()
void setRootItem( Object newItem )
boolean isEmpty()

**How can we use these methods to build a tree?**

```
btt = BinaryTree( "A" )
btt.attachLeft( "B" )
c = BinaryTree( "C" )
c.attachLeft("D")
btt.attachRightSubtree( c )
```

Binary Tree Examples
- Words sorted lexicographically
- Arithmetic expressions

**Recursive Definition of a Binary Tree**

A Tree T is a Binary Tree if either:
1. T has no nodes
2. T is of the form:

```
    n
   / \
  /   \
TL   TR
```

where n is a node and TL and TR are both Binary Trees.

**Recursive Definition of the Height of a Binary Tree:**
1. If T is empty, then its height is 0
2. If T is non-empty, then its height is 1 greater than the height of its root's tallest subtree. That is,

   height(T) = 1 + max( height(TL), height(TR) )

**Traversals of a Binary Tree:**

A traversal algorithm visits every node of a binary tree.

Now that we have seen the recursive definition of a binary tree, we can look at recursive traversal algorithms. The general form is:

```
traverse( binTree ) {
    if ( binTree is not empty ) {
```
traverse( left subtree of binTree )
traverse( right subtree of binTree )
}
}

What's missing from this algorithm?

It doesn't do anything with the root node

Three choices:
1. can 'visit' root r before traversing left and right subtrees.
2. can visit root after traversing left subtree but before traversing right subtree
3. can visit root after traversing left and right subtrees

These are called preorder, inorder, and postorder traversals, respectively.

Let's look at preorder traversal that displays a node's data when it visits the node.

    traverse( binTree ) {
      if ( binTree is not empty ) {
        display the data in the binTree's root
        traverse( left subtree of binTree )
        traverse( right subtree of binTree )
      }
    }

What is the order of visits with some example trees?

Look at expression trees also.

What is the Order of the traversal algorithm?
How many times is each node visited?
How many nodes are there?

Let's look at reference based implementation of a binary Tree:

(See BinaryTreeBasis.java, and BinaryTree.java – we've already looked at TreeNode)
Binary Search Trees

A binary tree where for every node N
• the value in N's left child < the value in N
• the value in N's right child > the value in N

Insert a new node by
• following links down
• attaching the new node where it 'should' go

Result – All nodes to left of root are < root, all nodes to the right are greater

The value stored in each node is called the "key" – it can be searched for. Examples: words, student names, passengers, ...

More complex objects: student records (sid, name, class level, gpa,...)

Important: can only use BST for items that have a key that has an ordering.

Recursive definition of BST:

for each node N, a BST satisfies the following 3 properties:
1. N's search key is greater than all search keys in N's left subtree TL.
2. N's search key is less than all search keys in N's right subtree TR
3. TL and TR are both BSTs

Operations on BSTs

In addition to the normal basic operations (create, isEmpty, etc):

• insert( newItem ) - Insert a new item into a BST
• delete( searchKey ) - delete an item with a given search key from a BST
• retrieve( searchKey ) - retrieve the item with a given search key from a BST
• Also want to traverse the BST in some order (pre, in, post)

Algorithms for the BST operations

search:

search( bst, searchKey )
if ( bst is empty )
    the desired record is not in the BST
else if ( searchKey = search key of the root's item )
    the root contains the desired record
else if ( searchKey < search key of the root's item )
    search( Left subtree of BST, searchKey )
else
search (Right subtree of BST, SearchKey )

This algorithm provides the basis for the insert, delete, and retrieve operations on the BST.

Assumption in book – items in tree have unique search keys – no duplicates.

**Insertion**

Uses search algorithm to find place where new item should go. Since the item is not in the tree (by assumption), search looks in place where new item belongs.

```java
insertItem( treeNode, newItem )  {     // treeNode is root
of tree
    if ( treeNode is null ) {
        TreeNode newNode = new TreeNode( newItem );
        treeNode = newNode;
    } else if (newItem.getKey() < treeNode.getItem().getKey())
    {
        treeNode.setLeft(
            insertItem( treeNode.getLeft(), newItem));
    } else {
        treeNode.setRight (  
            insertItem( treeNode.getRight(), newItem));
    }
    return treeNode;
}
```

**Retrieval** – just needs to return the item from the node found by searching:

```java
retrieveItem( treeNode, searchKey ) {  
    if ( (treeNode == null )
        treeItem = null;
    else if ( searchKey == treeNode.getItem().getKey())
        treeItem = treeNode.getItem();
    else if ( searchKey < treeNode.getItem().getKey())
        treeItem = retrieveItem( treeNode.getLeft(), searchKey
    )
    else
        treeItem = retrieveItem( treeNode.getLeft(), searchKey
    )
    return treeItem;
}
```
Delete an item from a BST

This is harder. Need to look at 3 cases:
1. Item is in a leaf
2. Item is in a node with one child
3. Item is in a node with two children

Case 1 is easiest – just set the reference in its parent to null.

Case 2: Let N be the node you want to delete. Let P be N's parent and C be N's child. It doesn't matter if C is a left or right child. Then, change the reference in P from N to C - that is, let P 'adopt' N's child.

Does this work? Let's look at some cases. For example, assume that N is left subchild of P. Then all of N's children are less than P (because they are in the left subtree of P). When P adopts C, C is now the left subchild of P. Every Node in C is less than P, so it is still a BST.

Third case: Node to be removed has 2 children.

What can you do? N's parent can't adopt 2 children.

Remember, we only want to delete the ITEM from the BST – don't have to delete the actual Node.
Solution – Don't delete the Node – replace its item with the item from another node, and then delete that node. We can't choose just any node (why not?) – Which is the right one?

What choices do we have for replacing the node?
Which one is easy to find?
How about the Inorder successor?
Which node is it? "leftmost" node in right subtree.
Note that this node will never have two children.

Move contents of inorder successor to node
Remove the inorder successor.

Look at BinarySearchTree.java

Iterators

An interator provides a general way to traverse a tree.
- Can't use internal type like TreeNode – violates 'wall' of information hiding

An iterator is a class that allows data from other structures to be read/accessed in order.
- sort of like StringTokenizer
Iterator methods:
- constructor: specifies the data structure to iterate over (traverse)
- hasNext() - true if there is another object to iterate to
- next() - returns the next object.
- other methods may be needed to define iteration order. e.g., with trees: preorder, inorder, postorder.

Java provides an Iterator interface that we can implement.

Look at TreeIterator.java and IteratorTest.java