More Fun With Trees

Iterators

- Provide a general way of traversing a tree
  - Can’t use internal types like TreeNode!
  - Instead, use an *iterator*

- An *iterator* is a class whose purpose is to allow other structures to be “read” in order
  - Example (sort of): Tokenizer

- An *iterator* supports a set of methods
  - Constructor: specifies the data structure to iterate over
  - hasNext(): **true** if there is another object to iterate to
  - next(): returns the next object in the traversal
Traversing a binary tree

- Trees can be traversed in three orders
  - Pre-order: root, L, R
    - A, B, C, D, F, E
  - In-order: L, root, R
    - B, A, F, D, C, E
  - Post-order: L, R, root
    - B, F, D, E, C, A
- Order chosen depends on
  - What the tree is being used for
  - What the traversal is supposed to accomplish
- Traversal is done recursively!
  - Treat L, R as trees in their own right
  - Recursively visit them

Pre-order traversal

- Visit nodes in this order
  - Root node
  - Left subtree
  - Right subtree
- Recursive visit
- Perform operation at the leaf
- Printing in this example

```java
void preorder () {
    System.out.println (root.value);
    if (left != null) {
        left.preorder ();
    }
    if (right != null) {
        right.preorder ();
    }
}
```
In-order traversal

- Visit nodes in this order
  - Left subtree
  - Root node
  - Right subtree
- Recursive visit
- Perform operation at the leaf
- Printing in this example

```java
t void inorder () {
    if (left != null) {
        left.inorder ();
    }
    System.out.println (root.value);
    if (right != null) {
        right.inorder ();
    }
}
```

Post-order traversal

- Visit nodes in this order
  - Left subtree
  - Right subtree
  - Root node
- Recursive visit
- Perform operation at the leaf
- Printing in this example

```java
t void postorder () {
    if (left != null) {
        left.postorder ();
    }
    if (right != null) {
        right.postorder ();
    }
    System.out.println (root.value);
}
### Binary search tree

- Rule 1: left node is less than root
- Rule 2: right node is greater than root
- Insert a new node by
  - Following the links down
  - Attaching the new node where it “should” go
- Result: all nodes in the left subtree are less than the root!

```java
void bininsert (Object item, String key) {
    if (key.compareTo(root.key)<0) {
        if (left != null) {
            left.bininsert (item, key);
        } else {
            insertLeft (item, key);
        }
    // same code for right...
}
```

### Searching in a binary search tree

- Searching similar to insertion
  - Go left if less
  - Go right if more
  - Can be done recursively (code below)
  - Can be done non-recursively
    - Loop, setting appropriate subtree to root each time

```java
Object binsearch (String key) {
    if(key.compareTo(root.key)==0) {
        return (item);
    } else if (item.compareTo(root.key)<0) {
        if (left != null) {
            return (left.binsearch (key));
        } else {
            return (null);
        }
    // same code for right...
}
```
Accessing a BST in sorted order

- How can we print this tree in sorted order?
- Starting at root, we know
  - All nodes in left subtree are “less” than root
  - All nodes in right subtree are “greater” than root
  - Left & right subtrees can both be printed in sorted order
- Solution: in-order traversal!
  - Print all nodes less than root in sorted order
  - Print root
  - Print all nodes greater than root in sorted order

Balancing trees (overview)

- Binary trees are most effective when they are balanced
  - Maximum depth of any node is no more than 1 greater than minimum depth of any node
- How can we balance a tree?
  - Preserve ordering rules!
  - Rearrange tree to fix heights
- Ensure tree is balanced after each insertion
  - Prevent it from getting too far out of balance
- Use a special kind of binary tree: 2-3 tree
  - General idea: keep the tree balanced at every step
  - Details beyond the scope of this class
Saving and restoring binary trees

- What if we want to save a tree?
  - Print it out in such a way that we can restore it later
  - Provide sufficient information to reconstruct an exact copy
- Use pre-order traversal
  - Print information about each node saying whether it has left, right, neither or both subtrees
  - Print the contents of the node
- Reconstruct recursively

```java
Tree buildTree () {
  read value, status
  t = new Tree (value);
  if (status == 2) { // A
    status == L)
      t.attachL(buildTree());
  }
  if (status == 2) { // B
    status == R)
      t.attachR (buildTree());
  }
  return (t);
}
```

Why save and restore trees?

- Helpful when one program creates a tree that another wants to use
  - Placement within the tree could be important!
  - Might not simply be sorted…
- Assignment 4: write a program to decompress a file
  - Read in compression “dictionary”: build a tree
  - Traverse the tree for each code to find the letter it corresponds to
  - This can save space by representing common letters with short codes
  - Extra credit: implement a program to compress a file
  - More details on Friday!