Lists, Trees,and Recursion

CS 211, Week of 4/8/02. TA: Kiri Wagstaff, wkiri@cs.cornell.edu

1 An API for Problem 1, Assignment 5:

See the newsgroup for an example of a useful API for this problem (Sabina's PERMUTATIONS HINTS post). It includes an example of a main method you can use to test your code. Seeing how the methods are used should make it easier to figure out how to implement them.

2 Doubly-linked Lists

Note: the linked list implementation shown in lecture is not the same as the LinkedList class in Java. The lecture version shows you how a LinkedList class could be implemented. The Java LinkedList class is already implemented for you. It has different methods, but supports the same functionality.

Doubly-linked lists (DLLs) include references to the cell *before* as well as the cell after the current one. They make list traversal a bit easier. Let's see how insert() and delete() work for a DLL.

We define a DLLcell (inner) class inside the DLL class itself. This indicates what data is stored in each cell of the list. The DLL class contains a reference to the first such cell, which is the head of the list.

```
public class DLL {
  DLLcell head; // reference to the first cell in the list
  // default constructor - nothing in the list
  public DLL() { head = null; }

  // Insert obj at the beginning of the list.
  public void insert(Comparable obj) {
    DLLcell d = new DLLcell(obj);
    // Hook d's next reference to the beginning of the list.
    d.next = head;
    // d.prev is initialized to null by the DLLcell constructor.
    // Update the old head to point back to d.
    head.prev = d;
    // Now reset head to point to d, the new head of the list.
    head = d;
  }

  public Comparable delete(Comparable obj) {
    // First, find 'obj' in the list.
    // Use a 'finger' to walk through the list until we find it.
    // We'll do it iteratively.
    DLLcell finger = head;
    // While we aren't at the end of the list, and the current item
    // isn't the same as obj, keep looping.
    while (finger != null && finger.datum.compareTo(obj) != 0) {
      finger = finger.next;
    }
```

```
    // Now either finger is null (didn't find obj)
    // or it points to the cell that contains obj.
    if (finger == null) return null; // no obj in list

    // Remove obj by:
    // 1. Set the 'next' of the cell before it to point to the cell after
    finger.prev.next = finger.next;
    // 2. Set the 'prev' of the cell after it to point to the cell before
    finger.next.prev = finger.prev;

    // Done!  Now return the data in finger.
    return finger.datum;
  }

  // Inner class!
  private class DLLcell {
    Comparable datum;
    DLLcell prev; // reference to the previous cell in list
    DLLcell next; // reference to next cell in list

    // constructor - stores obj in the DLLcell.
    // prev and next aren't connected to anything.
    DLLcell(Comparable obj) {
      datum = obj;
      prev = null; next = null;
   }
}
```

Some questions:

1. Exercise: write the delete() method recursively (yes, it can be done).
2. Why did we implement delete() iteratively?
3. How would you implement an insertAt(Comparable obj, index i) method?

In general:

Iteration is useful when there is only one "next" item to check each time.

Recursion is useful when there is more than one "next" item to check each time.

3 Parent Links in a Tree

```
/** Binary Tree program to show the pain of iterative tree solutions
 * @author by Jeff Hoy for section week 10
 * @date April 8, 2002
*/
class Node {
  private Node left;
  private Node right;
```

```
private Node parent;
private char value;

public Node(char value)
{ left = null;  right = null;  parent = null;  this.value = value; }

public Node(char value, Node left, Node right) {
  this.left = left;  this.right = right;  this.value = value;
  if (left != null)   left.setParent(this);
  if (right != null)  right.setParent(this);
  parent = null;
}
public void setParent(Node parent) { this.parent = parent; }
public String toString() { return value + ""; }
public Node getLeft()   { return left; }
public Node getRight()  { return right; }
public Node getParent() { return parent; }

// Returns iterative in-order walk of the tree
public String iterativeInOrder() {
  Node current = this, last = null;
  String ret = "";
  while (current.getLeft() != null) {
    ret += '(';
    current = current.getLeft();
  }
  while (current != null) {
    if (current.getLeft() != null || current.getRight() != null)  ret += ' ';
    ret += current.toString();
    if (current.getLeft() != null || current.getRight() != null)  ret += ' ';
    if (current.getRight() == null) {
      // advance to next unprinted parent node
      last = null;
      while (current != null && last == current.getRight()) {
        if (current.getRight() != null)  ret+= ')';
        last = current;
        current = current.getParent();
      }
    } else {
      // advance to the leftmost Node on the right side
      current = current.getRight();
      while (current.getLeft() != null) {
        ret += '(';
        current = current.getLeft();
      }
    }
  }
  return ret;
}
```

```
static void main(String[] args) {
   // Build expression (((1 + 2) * (3 - 4)) % 5)
   Node one = new Node('1');
   Node two = new Node('2');
   Node three = new Node('3');
   Node four = new Node('4');
   Node five = new Node('5');

   Node plus = new Node('+', one, two);
   Node minus = new Node('-', three, four);
   Node times = new Node('*', plus, minus);
   Node mod = new Node('%', times, five);

   Node root = mod;
   // Should print "(((1 + 2) * (3 - 4)) % 5)"
   System.out.println(root.iterativeInOrder());
  }
}
```

4 Binary Search Trees

We'd like to write a method which, given a tree, determines whether or not the tree is a binary search tree (BST). Let's do it *recursively.*

First, what are the properties of a BST?

1. It is a binary tree: each node has 0 or 2 children.
2. Recursive definition of a BST:
   (a) Base case: Leaf nodes are BSTs.
   (b) Recursive case:
   The left sub-tree is a BST **and**
   The right sub-tree is a BST **and**
   The value at this node is ¿= the max value in the left sub-tree**and**
   The value at this node is ¡= the min value in the right sub-tree.

We will solve this problem by writing a method minMaxBST() that takes in a tree and returns a Triplet. A Triplet is an object that contains three values:

1. isBST: a boolean (is this tree a BST?)
2. min: an int (the min value in the tree)
3. max: an int (the max value in the tree)

```
// This method checks if a tree is a BST
public static boolean isBST(TreeCell t) {
   return minMaxBST(t).isBST;
}

// returns a triplet containing <boolean,low,hi>
// boolean is true if parameter is a BST
// low and hi are the lowest and highest objects in BST
public static Triplet minMaxBST(TreeCell t) {
  Triplet left, right;
  // Sanity check: handle null trees. Say they're empty BSTs.
  if (t == null)  return new Triplet(true, null, null);

  // BASE CASE: Leaf node.
  // Leaf nodes are BSTs, by default.
  if (t.getLeft() == null && t.getRight() == null)
    return new Triplet(true, (Comparable)nodeValue, (Comparable)nodeValue);

  // RECURSIVE CASE: Internal node.  Check left and right sub-trees,
  // then see if the BST properties are satisfied.
  Object nodeValue = t.getDatum();
  // get left triplet
  left = minMaxBST(t.getLeft());
  // get right triplet
  right = minMaxBST(t.getRight());

  // if leftsubtree is a BST and rightsubtree is a BST
  // and max value in left subtree is <= value in node
  // and min value in right subtree is >= value in node
  // we have a BST!
  boolean isBST = left.isBST && right.isBST
                  && (left.max.compareTo(nodeValue) <= 0)
                  && (right.min.compareTo(nodeValue) >= 0);
  return new Triplet(isBST, left.min, right.max);
}

// Inner class!
private class DLLcell {
  boolean isBST;
  int min, max;

  Triplet(boolean isBST, int min, int max) {
    this.isBST = isBST;
    this.min = min;  this.max = max;
  }
}
```

Note that we assume that t is at least a binary tree.

1. What will happen if it isn't? (I.e. if there is a node with only a left child or only a right child?)

2. What would you need to change to handle non-binary input trees?