CS61B, Fall 2009 **HW #7** **P. N. Hilfinger**

Due: Mon., 26 October 2009

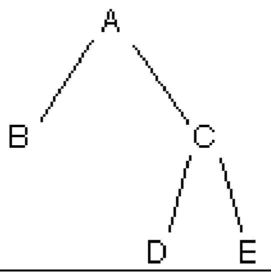
Homework Exercises. You'll find a skeleton for your answers in the `hw7` staff directory.

1. Consider an implementation of binary trees using a `BinaryTree` class with an inner `TreeNode` class, as shown below. The framework is available online in the skeleton file `hw7/BinaryTree.java`.

Fill in the blanks in the following code (part of `BinaryTree` to print a tree so as to see its structure. Empty trees (such as the children of leaf nodes) should print nothing.

```
/** Dump THIS, with indentation showing structure. */
public void print ( ) {
  if (myRoot != null) {
    print (myRoot, 0);
  }
}

/** Dump ROOT indented by INDENT indentation units. */
void print (TreeNode<?> root, int indent) {

  // REPLACE THIS

  println (root.myItem, indent);

  // REPLACE THIS

}

/** Number of spaces in one indentation unit. */
static int INDENTATION = 4;

/** Print OBJ, indented by INDENT indentation units, followed by a
 *  newline. */
static private void println (Object obj, int indent) {
  for (int k = 0; k < indent * INDENTATION; k += 1)
    System.out.print (" ");
  System.out.println (obj);
}
```

The `print` method should print the tree in such a way that if you turned it 90 degrees clockwise, you see the tree. Here's an example:

Tree	**Printed version**
	

```
            E
        C
            D
    A
        B
```

2. Compilers and interpreters convert string representations of structured data into tree data structures. For instance, they would contain a method that, given a `String` representation of an expression, returns a tree representing that expression:

```
/** The expression tree corresponding to S.  S is a legal, fully
 *  parenthesized expressions, contains no blanks, and involves
 *  only the operations + and *, and leaf labels (which can be
 *  any string of characters other than *, + and parentheses). */
public static BinaryTree<String> exprTree (String s) {
    BinaryTree<String> result = new BinaryTree<String> ( );
    result.myRoot = result.exprTreeHelper (s);
    return result;
}
```

See the example on the next page.

Complete and test the following helper method for `exprTree`. You will find this in skeleton file `hw7/BinaryTree.java`.

```
private TreeNode<String> exprTreeHelper (String expr) {
    if (expr.charAt (0) != '(') {
        return null; // REPLACE WITH MISSING CODE
    } else {
        // expr is a parenthesized expression.
        // Strip off the beginning and ending parentheses,
        // find the main operator (an occurrence of + or * not nested
        // in parentheses), and construct the two subtrees.
        int nesting = 0;
        int opPos = 0;
        for (int k=1; k<expr.length()-1; k += 1) {
            // REPLACE WITH MISSING CODE
        }
        String opnd1 = expr.substring (1, opPos);
        String opnd2 = expr.substring (opPos+1, expr.length()-1);
        String op = expr.substring (opPos, opPos+1);
        return null; // REPLACE WITH MISSING CODE.
    }
}
```

Given the expression `((a+(5*(a+b)))+(6*5))`, your method should produce a tree that, when printed using the `print` method you just designed, would look like

```
        5
    *
        6
+
                b
            +
                a
        *
            5
    +
        a
```

3. Given a tree returned by the `exprTree` method, write and test a method named `optimize` that replaces all occurrences of an expression involving only integers with the computed value. Here's the header.

```
public static void optimize (BinaryTree<String> expr)
```

It will call a helper method as did `BinaryTree` methods in earlier exercises.

For example, given the tree produced for

```
((a+(5*(9+1)))+(6*5))
```

your `optimize` method should produce the tree corresponding to the expression

```
((a+50)+30)
```

Don't create any new `TreeNodes`; merely relink those already in the tree.

4. Assume that we have a heap that is stored with the largest element at the root. To print all elements of this heap that are greater than or equal to some key X, we *could* perform the `removeFirst` operation repeatedly until we get something less than X, but this would presumably take worst-case time Theta(k lg N), where N is the number of items in the heap and k is the number of items greater than or equal to X. Furthermore, of course, it changes the heap. Show how to perform this operation in $\Theta(k)$ time *without* modifying the heap. See the skeleton file `hw7/HeapStuff.java`.

5. Getting started on the project. Your skeleton file contains enough information to implement the tracker package. You can start out using either the staff util package or using the `util.SimpleSet2D` class as a temporary substitute for `util.QuadTree`. Follow the advice in the Project 2 spec: write `User-Manual`; implement simple commands likes like comments, the empty command, `help`, and `quit`; implement `bounds` and `rad`; implement `add`. With these, you can, in principle, set up any particular set of points you want. At this point, you might want to add an extra command that simply prints all the points in your set with their positions and velocities, or you can go straight to the first form of `near`. Be sure to implement the part of the spec that says that you can abbreviate commands. Try to get through as much of the tracker as possible. Have (at least) regression tests for each of the commands you add.

Put all of your work in your `proj2` SVN directory and commit it. Submit the resulting project as `proj2-`N (yes, we will be able to keep it straight from your later real submission).