46-935 Homework Assignment #1

Short Answer (25 points total):

1. Java technology spans several levels. Java source code is the high level closest to the developer; at the lowest level machine level code is executed by the processor on the machine that is running a Java program. Give a description of how Java source code is transformed into machine level instructions. Which intermediate levels are *platform independent*, and which are tied to specific platform choices?

2. What is an *exception*? Describe Java's approach to exception handling. What does it mean to **catch** an exception? to **throw** an exception? Why might you want to define your own exception class?

3. What is an *interface*? Why might you use an interface? How does an abstract class differ from an interface?

4. What is the difference between a Java object reference and a C++ pointer to an object? Why do you think the architects of Java chose not to include pointers in the Java language?

Programming (75 points total):

Download the source files from the Assign1 handout directory via FTP (/afs/andrew/course/46/935/handout/Assign1) or download the zip file from the 46-935 web page. Make sure to download all of the files.

The source files are a Java implementation of the extensible program for pricing options on a probabilistic directed acyclic graph that was studied in the previous course (46-927). You should begin by examining the Java source code. The Java classes should be very similar to the C++ classes you worked with during the last mini. Familiarize yourself with the code and think about any important differences that you notice in the Java implementation.

Note: This course will build upon the material covered in 46-927. If you did not take 46-927, please examine Sections 2 and 3 of the 46-927 course notes as well as the code distributed with 46-927 Assignments #2 and #3. This material is available off of the 46-927 course web page: http://www.andrew.cmu.edu/course/46-927. If you have further questions, please contact the TA or course instructor.

Part 1 (50 points):

Your first programming task will be to carry over some of the extensions you made to the C++ implementation last mini to the new Java implementation. This familiar assignment should help you get up to speed working in the Java environment.

Specifically, you must implement the **LookBackOption**, **GeometricCallOption**, and **AsianCallOption** classes. In addition, you must fill in the **calc_american_option()** method in the *AbstractOption* class.

Recall that the *LookBackOption*, *GeometricCallOption*, and *AsianCallOption* classes extend the *AbstractOption* class to implement their particular kind of option as defined in Section 3.1 of the 46-927 course notes.

Recall that *calc_american_option()* will calculate an option on a PDAG using the American method (where payoff can be computed at any node). Your function should be very similar to the existing *calc_european_option()*. Refer to Section 3.3 of the 46-927 course notes for information on calculating an American option.

Part 2 (25 points):

The second programming task will be to improve the efficiency of the Option Calculator program. Recall that we represent an option lattice as an array of linked list objects (where nodes[i] contains all the nodes in the PDAG for time value i). The option lattices we represent are often recombining (two different nodes at time t may share a successor at time t+1). To take advantage of this recombining, each linked list only contains the nodes with unique data elements (nodes with duplicate data are not inserted).

Examine the Insert(KeyInterface k) method of the *LinkList* class and underlying InsertElement(KeyInterface k) method of the *Node* class. These methods implement unique insertion into a linked list (where duplicate keys are not inserted in the list). Notice that the algorithm employed by these methods is inefficient. A search for a node with equivalent data is done through all elements of the linked list with each insertion. Assuming that such a node is already in the list, this algorithm will have an average running time of (n/2) where n is the number of elements in the list. If the node is not already in the list, the running time will be simply n.

This algorithm could be improved by using a binary search tree. In a binary search tree, each node in a tree has references to left and right subtrees. In its most common form, the left subtree is filled with nodes that are “less than” the parent node, while the right subtree is filled with nodes that are “greater than” the parent node. Assuming a reasonably balanced tree, this invariant makes searching the tree for an item take an average of log(n) comparisons. Think about why!

Your task will be reduce the average running time of the unique insertion algorithm to log(n) using a binary search tree **ON TOP** of the existing linked list. In addition to keeping a linked list of nodes, we will maintain a binary search tree. You will fill in the methods in the *BSTNode* class. The *BSTNode* class is an extension of the *Node* class and inherits its data members and methods. You will override the InsertElement() method to make it more efficient (by using the left and right subtree pointers to search through the nodes). Remember that if a node is unique, in addition to adding it to the linked list (using InsertAfter() or equivalent) you must update the binary tree.

The *BSTList* class has already been filled in for you. When you are confident in your *BSTNode* class, change the PDAG representation in the *AbstractOption* class to use *BSTList* objects instead of *LinkList* objects.

Notice how little code we had to write to change the underlying representation of the list to use a binary search tree. This is the power of inheritance and good program design!!

Handin Procedure:

Pittsburgh/NY: FTP the modified source files and a document (txt, rtf, or doc) containing your answers to the written questions to the course handin directory (where *userid* is your andrew user id):

/afs/andrew/course/46/935/handin/*userid*/

London: Email the modified source files and a document (txt, rtf, or doc) containing your answers to jeffreys@andrew.cmu.edu **AND** pavani@andrew.cmu.edu.