CS4100/5100 COMPILER PROJECT

Part 1, Foundations: Assignment 2, Symbol and Quad Tables

Continuing the purpose of this assignment, which is to give the student a practical introduction to the structure and usage of the foundational data elements of the compiler project, Assignment 2 is the construction of two more table ADTs. The code developed here will be used throughout the rest of the semester. **Creating these Abstract Data Types (ADTs), exactly as described, is crucial to the success of the project!** While there is room for individuality in the actual *implementation*, it is necessary that each structure utilize exactly the ***interfaces*** and ***storage capabilities*** described, in order for them to be used effectively and efficiently in the development of the rest of the project. **The language required to operate with the automated testing system for all submissions is Java. Read all requirements below carefully!**

Program Objectives- 2: Implement classes for the Symbol and Quad Tables according to the interface requirements provided below.

Main Program:

The student should do initial testing by creating their own main program for this part of the project. A main test program and expected output .txt file will be provided by the instructor to test the ADT methods and verify that they work according to specifications. Code will be submitted via Canvas.

The ADTs to Implement

The following ADTs must be established as Java classes, according to the exacting requirements below:

1) Class SymbolTable

The symbol table conceptually is a ***fixed index list*** (i.e., once a row of data has been added, its index **must not change** during the run of the program) consisting of an indexed list of entries, with each entry (a single row of data) containing all of the following: a *name* field (String type); a *kind* field (char to hold a ***label***, ***variable***, or ***constant*** indicator, represented by the characters **'L', 'V', or 'C'**); a *data_type* field specifying the data as an integer, float, or string (char, represented **'I', 'F', or 'S'**); and a set of *3 value* fields of the appropriate type (*integerValue, floatValue, stringValue*) to hold one of these: an integer, a double, or a String, as selected by the contents of the *data_type* field. (Later, it will be clear why labels will always store their data in the *integerValue* field, while variables and constants may use any one of the *integerValue, floatValue, stringValue* storage areas based on their *data_type* field). The interface to the SymbolTable must implement the following methods (in addition to a constructor, as shown)**:**

SymbolTable(int maxSize)

Initializes the SymbolTable to hold *maxSize* rows of data.

int AddSymbol(String symbol, char kind, int value)
int AddSymbol(String symbol, char kind, double value)
int AddSymbol(String symbol, char kind, String value)

Three overloaded methods to add *symbol* with given *kind* and *value* to the end of the symbol table, automatically setting the correct *data_type,* and returns the index where the symbol was located. **If the symbol is already in the table according to a non-case-sensitive comparison** ["Total" matches "total" as well as "ToTaL"]**with all the existing strings in the table, no change or verification is made, and this just returns the row index where the symbol was found.** These methods only FAIL, and return -1, when the table already contains *maxSize* rows, and adding a new row would exceed this size limit. This should not happen.

int LookupSymbol(String symbol)

Returns the index where symbol is found, or -1 if not in the table. **Uses a non-case-sensitive comparison**.

String GetSymbol(int index)
char GetKind(int index)
char GetDataType(int index)
String GetString(int index)
int GetInteger(int index)
double GetFloat(int index)

Return the various values currently stored at *index.*

UpdateSymbol(int index, char kind, int value)
UpdateSymbol(int index, char kind, double value)
UpdateSymbol(int index, char kind, String value)

Overloaded methods, these set the kind and value fields at the slot indicated by index.

PrintSymbolTable(String filename)

Prints to a plain text file all the data in **only the occupied rows** of the symbol table. Must be in neat tabular format, 1 text line per row, selectively showing only the used value field (stringValue, integerValue, or floatValue) which is active for that row based on the dataType for that row.

2) Class QuadTable

The QuadTable is different from the SymbolTable in its access and contents. Each fixed indexed entry row consits of four int values representing an opcode and three operands. This means it can be implemented simply with a 2-dimensional array of integers, maxSize x 4 in dimension. The methods needed are:

QuadTable(maxSize)

Constructor creates a new, empty QuadTable ready for data to be added, with the specified maxumum number of rows (maxS*ize*). An array is suggested as the cleanest implementation of this, along with a private int *nextAvailable* counter to keep track of which rows have been used so far.

int NextQuad()

Returns the int index of the **next open slot in the QuadTable.** Very important during code generation, this must be implemented exactly as described, and must be the index where the next *AddQuad* call will put its data.

void AddQuad(int opcode, op1, op2, op3)

Expands the active length of the quad table by adding a new row at the *NextQuad* slot, with the parameters sent as the new contents, **and increments the NextQuad counter to the next available (empty) index when done.**

int GetQuad(int index, int column)

Returns the int data for the row and column specified at *index, column.*

void UpdateQuad(int index, opcode, op1, op2, op3)

Changes the contents of the existing quad at *index.* Used only when backfilling jump addresses later, during code generation, and very important.

PrintQuadTable(String filename)

Prints to the named file only the **currently used contents** of the Quad table in neat tabular format, one row per output text line.

Turn-In Requirements

The turn-in consists of **five files:** the .java class files for SymbolTable and QuadTable, the console output from the instructor-provided **main.java** program (saved as a .txt file), and the Symbol.txt and Quad.txt files created by the calls to ***PrintSymbolTable(...Symbol.txt)*** and ***PrintQuadTable(...Quad.txt).***