George Mason University
INFS 519 – Data Structures
Fall 2018
Programming Assignment 2 – Queues, Stacks and Hashtables

In this project, you will write a calculator program to implement a *postfix machine* for evaluating arithmetic expressions as follows: a simple example of a *postfix expression* 1 2 3 * + will evaluate to 7, another example 1 2 – 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / - will evaluate to -8. Your calculator program should also be capable of evaluating expressions which include variables: x 3 2 + = y 4 = x y + which will evaluate to 9. Your program will read postfix expressions from an input file, one expression at a time, and will then display that postfix expression back on the program console's window along with the result of evaluating that expression (refer to section 11.2.1 in your text book for more about postfix machines).

Your program will be reading postfix expressions from an input file (one expression at a time) and will store the read expression in a queue (call it *input queue*) to prepare it for processing by the postfix machine. To evaluate an arithmetic expression, a stack is used (call it *program stack*) as follows: - when an operand is seen, it is pushed on a stack, - when an operator is seen, the appropriate number of operands are popped from the stack, - the operator is then evaluated and the result is pushed back onto the stack. For binary operators, two operands are popped. A variable (which is a String type) can be assigned a value by an assignment operator of low precedence. So the program will push a variable onto the stack and then perform a pop of the variable when an assignment operator is encountered. The program will maintain a hash table structure (call it *symbol table*) for storing variables and their values. This hash table specifically maps Strings to some other data type, such that the keys always have a predictable hash code (use the String class' .hashCode() method). When the complete postfix expression is evaluated, the result should be a single item on the stack that represents the answer.

The postfix machine should support the following binary operators: +, -, *, /, ^ (exponentiation) along with the following assignment operators: =, +=, -=, *=, /=. You can assume the following with the postfix program you need to process:

- All inputs are well-structured postfix programs with no syntax errors; however, if traversing an input arithmetic expression leads to a stack situation with an error, the program displays an INVALID input and moves to the next arithmetic expression.
- There will not be any division by zero;
- There is always a single space between different numbers, operators, and names in an input file;
- The postfix program will *only* include integer numbers;
- You only need to support the following arithmetic operations +, -, *, /, and ^ (exponentiation); all arithmetic operations are binary. You will need to support additionally =, +=, -=, *=, /= as assignment operators.

A sample run of the program may look like:

```
Input expression: 1 2 3 * +
Value: 7

Input expression: * 1 2 3 * +
Value: INVALID expression format


Input expression: 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -
Value: -8

Input expression: x 3 2 + = y 4 = x y +
Value: 9

Input expression: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -
Value: -749

Input expression: 6 2 + 5 * 8 4 / -
Value: 38

Input expression: 1 2 + 3 4 - 5 6 + 4 2 / * * +
Value: -19

Input expression: x 3 2 + = y 4 = x y +
Value: 9
Symbol table entries: x=5, y=4

Input expression: x 1 = y 6 = y 1 1 + /= y
Value: 3
Symbol table entries: x=1, y=3

Input expression: x 1 = y 6 = y x x + /= y
Value: INVALID format
Symbol table entries: x=1, y=6

Input expression: 2 num1 = y 17
Value: INVALID format

Input expression: number1 15 = number1 3 /= number1 20 + var2 100 =
Value: 25
Symbol table entries: number1=5, var2=100

Input expression: somevalue1 1 1 * 2 3 4 + somevalue2 2 = 1 + / * =
somevalue1 /= 2
Value: 4
Symbol table entries: symbolvalue1=4, symbolvalue2=2
```

Internals

You may **not** use any classes from java.util (unless for program I/O).
You need to write the following classes:

- **ExpressionEvaluator.java**
  This is the main program, it reads input from an input file (one expression at a time) and adds elements of an expression to an internal **input queue** object - which supports enqueue and dequeue methods. Once an expression is read and stored into the input queue, this class will start processing the expression by retrieving the elements from the input queue and pushing them onto the **program stack** for processing. This class will also add elements to the symbol table as needed. Once done, it displays the postfix expression and the result of evaluation on the console screen. After displaying an expression and its value, this class will clear both the program stack and the hash table entries and the hash table is resized to initial size of 2. The program will then read the next input arithmetic expression from file and process it .. Program ends when there is no more expressions in the input file.

- **ProgramStack.java**
  This is a **generic** class used to implement the program stack, includes *only* the following public methods:
  - `public ProgramStack(/* add arguments as needed*/)` : this is a constructor used to initialize the instance variables of a ProgramStack object.
  - `public void push(T item)` : pushes an item on the stack in O(1) time, you may assume item is never null.
  - `public T pop()` : pops an item off the stack in O(1) time, returns null if stack is empty.
  - `public T peek()` : returns the item at top of the stack in O(1) time, returns null if stack is empty.
  - `public void clear()` : removes everything from the stack in O(1) time.

- **SymbolTable.java**
  A **generic** class to implement a simple probing hash table to store variables along with their values. The hash table is implemented as a dynamic array (with initial size of 2)and rehashing must be implemented if the array is doubled/shrinked. This class defines the symbol table used in the program as a private instance variable:

```
private TableEntry<String,T>[] hashtable;  // a dynamic array!
```

  It also includes the `TableEntry` class defined as follows:

```
class TableEntry<K,V> {
      private K key;
      private V value;

      public TableEntry(K key, V value) {
            this.key = key; this.value = value; }
      public K getKey() { return key; }
      public V getValue() { return value; }
      public String toString() {
            return key.toString()+":"+value.toString(); }
}
```

Furthermore, this class includes *only* the following public methods:

- `public SymbolTable()` : creates a hash table where the initial storage is 2 and string keys can be mapped to T values.
- `public int getCapacity()` : returns how bid the hash table is in O(1) time.
- `public int size()` : returns the number of elements in the hash table in O(1) time.
- `public void put(String k, T v)` : places value v at the location of key k. This method uses linear probing if a location is in use. If the key already exists in the table, replace its value with v. You may assume both k and v are not null. If the key isn't in the table and the table is >= 80% full, expand the table to twice the size and rehash. Worst case: O(n), Average case: O(1).
- `public T remove(String k)` : Removes the given key (and associated value) from the table and returns the value removed or null if value is not found. Must to use *lazy delete*s! Worst case: O(n), average case: O(1).
- `public T get(String k)` : Given a key, returns the value from the table, returns null if not found. Worst case: O(n), average case: O(1).
- `public void clear()` : removes everything from the hash tabl: worst case: O(n), average case: O(1).
- `public boolean rehash(int size)` : increases or decreases the size of the hash table, rehashing all values. If the new size won't fit all the elements, returns false and does not rehash. Returns true if able to rehash.

To turn in

You will turn in a (well-commented) source listing via Blackboard. A rubric will be posted for the next class.