Computer Science E214

Tutorial 5

1 Bracket matching (Book 4.3.6)

Write a `Stack` client `Parentheses` that reads in a text stream from standard input and uses a stack to determine whether its parentheses are properly balanced. For example, your program should print `true` for `[()]{}{[()()]()}` and `false` for `[(])`.

2 Linked Stack of Strings (Book 4.3.23)

Augment the class `LinkedStackOfStrings` in PROGRAM 4.3.2 in the following ways:

- Write an iterative method `delete()` that takes an integer parameter k and deletes the kth element (assuming it exists).
- Write a method `find()` that takes an instance of `LinkedStackOfStrings` and a string `key` as arguments and returns `true` if some node in the list has `key` as its item field, and `false` otherwise.
- Write a method `removeAfter()` that takes a `Node` as an argument and removes the node following it (and does nothing if the argument or the next field in the argument node is null).
- Write a method `insertAfter()` that takes two `Node` arguments and inserts the second after the first in the list (and does nothing if either argument is null).

Add suitable test code to the `main` method of the class to verify your code. `LinkedStackOfStrings`.

3 Array Queue or Strings (Book 4.3.18)

Develop a class `ArrayQueueOfStrings` that implements the queue abstraction for strings using a fixed-size array. Allow the array size to be specified as a parameter to the constructor. Ensure that both the `enqueue` and `dequeue` methods run in constant time. *Hint*: if you want to add something before the front of the array, wrap around to the end of the array — but be careful of the array getting full!

Alternatively, make use of doubling and halving when the array becomes too full or too empty to remove the size parameter to the constructor. This approach to managing the underlying array is the basis of Java's `ArrayList` class.

Optional extension: Modify the above code to create a queue for a parametrized data type, rather than strings. Note the Q+A in the textbook at the end of Section 4.3.

4 Min-Max (Book 4.4.20)

Modify `BST` to add methods `min()` and `max()` that return the smallest (or largest) key in the table (or null if no such key exists).

Use recursion for `min()`, but do not use recursion for `max()`.

5 More `SET` operations

Write a new class `SetOps`, which implements the following additional operations for the `SET` class. Do not add the methods to the `SET` class. You only need to support sets of strings.

- set minus: the relative complement of B in A, written $A \setminus B$, is the set of elements of A not in B. Implement this using the enhanced for-loop provided by the `SET` class implementing `Iterable`. Your method should have signature:
  `public static SET<String> setminus(SET<String> a, SET<String> b)`
- symmetric difference: the symmetric difference between sets A and B is the set of elements in either A or B, but not both. Implement this by making use of the `union` and `intersects` methods provided by the `SET` class, as well as the `setminus` method you wrote in the previous step. Your method should have signature:
  `public static SET<String> symmdiff(SET<String> a, SET<String> b)`

Using the test client in the `main` method of the `SET` class as inspiration, write a `main` method for your class testing your methods.

6 Evaluating Boolean expressions

Modify Dijkstra's two-stack algorithm to evaluate Boolean expressions: write a function `evaluate` taking a `String` representing the boolean expression. Assume the expression uses the Boolean operators AND (`&&`), OR (`||`) and NOT (`!`), while operands may be general Java variable names. Your algorithm can assume the expression is fully parenthesized (see p. 571), and that there is a space between each bracket, operator, and operand.

To evaluate the expression, values must be assigned to the variable names: you should get these from a symbol table also passed to the evaluate function. Thus, the signature of your `evaluate` method should be something like:

```
public static boolean evaluate(BST<String,Boolean> vars, String expr)
```

Finally, to test your method, write a driver in the `main` method of the class. This main method should read from standard input: each line should either set a (new or existing) variable name to `true` or `false` (with a command like `found false`), or provide an expression to be evaluated (with a command like `eval ( found && ( ( ! hungry ) || lost ) )`.

Hints:

1. The variables that are being set should be put into the symbol table to be used by evaluate.
2. The `String` class provides a `trim` method, which removes leading and trailing whitespace from a string.
3. You can use the `split` method of the `String` function to split an expression into an array of Strings, each containing a bracket, operator, or variable name.

7 Doublets (Book 4.5.31)

Write a program `WordLadder` which can be used for playing the game of *doublets* (also known as word golf). In doublets, you are given a starting word, and a target word, and your aim is to find the shortest sequence of words starting at the first word and ending at the target, such that each intermediate word in the list is also a valid word, and differs from the previous word by either adding or removing a character, or by changing one character. For example, if *goat* is a word in the sequence, the next word might be *gloat*, *got*, or *moat*.

The `main` method of your class should accept a single command-line parameter specifying a file containing a dictionary of legal lower-case words, one per line. It should then repeatedly accept a starting and target word, and print out the word sequence, until an empty starting word is entered. If no legal path can be found, print a suitable message.

To perform this task, apply the `Graph` and `PathFinder` classes, making valid words as vertices, and putting edges between words if they may be adjacent in the word sequence. In this case, a path in the graph corresponds to a valid word sequence, and we can search for shortest paths as we do for Bacon numbers. A useful addition to the `Graph` class provided in the resources file over that in the textbook, is the `addVertex` method, allowing you to add vertices to the graph without connecting them to other edges immediately.

Some additional notes that may be useful:

- `dictionary.txt` in the provided resources has around 145000 words, and ends up needing around 460000 edges. It can take a while to construct a graph this size, so test on smaller examples first, or be patient.
- When a path is not found, `PathFinder`'s `distanceTo` method returns `Integer.MAX_VALUE`.

Example usage

```
$ java WordLadder dictionary.txt
love
hate
love -> hove -> have -> hate
green
beret
green -> reen -> been -> beet -> beret
johnny
bravo
No path found.
STOP
$
```