Computer Science E214

Tutorial 9

2014

For this tutorial, see the `Stack`, `BST`, and `SET` implementations provided in the tutorial's resources section.

Do the following exercises.

1 Book 4.3.9

Add a method `peek()` to `Stack` (PROGRAM 4.3.4) that returns the top element on the stack (without popping it).

2 4.3.1

Add a method `isFull()` to `ArrayStackOfStrings`. Write a client to test it.

3 4.3.24

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, `false` otherwise. Test your function in a test client. This test client may be the main function in the class `LinkedStackOfStrings`.

4 4.3.23

Write an iterative method `delete()` for `LinkedStackOfStrings` that takes an integer parameter k and deletes the kth element (assuming it exists).

5 4.3.28

Write a method `removeAfter()` that takes an instance of a `LinkedStackOfStrings` `Node` as a argument and removes the node following the given one (and does nothing if the argument or the next field in the argument node is null). Test your function in a test client, use the one you used in `4.3.24`.

6 4.3.29

Write a method `insertAfter()` that takes two `LinkedStackOfStrings` `Node` arguments and inserts the second after the first in the list (and does nothing if either argument is null). Test your function in a test client, use the one you used in `4.3.24` and `4.3.28`.

7 4.3.44 Most recently used

Read in a sequence of characters from the standard input and maintain the characters in a linked list with no duplicates. When you read in a previously unseen character, insert it at the front of the list. When you read in a duplicate character, delete it from the list and reinsert it at the beginning of the list.

Name your program `MRU` – it implements the well-known *Most Recently Used* strategy which is useful for caching, data compression, and many other applications where items that have been recently accessed are more likely to be used in the near future.

8 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 `[(])`.

9 Book 4.3.18

Develop a class `ArrayQueueOfStrings` that implements the queue abstraction 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!

Optional (more advanced) extensions:

- Modify the above code to make use of doubling and halving when the array becomes too full or too empty to remove the size parameter to the constructor.
- Modify the above code to create a queue for a parametrized data type, rather than strings. Note the Q+A in the textbook on p. 594.

10 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()`.

11 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.

12 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.