CS 315 Week 4 (Feb 19 and 21) summary and review questions

Topics covered

- Discussion of project # 1, permutation generation.
- Stack ADT, array implementation of stack operations – push, pop, top, isEmpty, isFull etc.
- Stack applications – (a) checking balanced parentheses (b) evaluation of a postfix arithmetic expression and (c) infix to postfix conversion.

SUMMARY:

- Pseudo-code for **`buildp`**, the additional function needed in the Combination class to implement permutation generation. (Note: The error in the pseudo-code A[j-1] instead of A[j] has been corrected in the updated PDF file posted.)
- Stack – basic operations.
- Array implementation takes a constant time (denoted usually as O(1)) to support PUSH, POP, TOP, ISEMPTY and ISFULL. Printing a stack takes O(N) operations where N = size of the stack.
- Informal definition of balanced parentheses involving multiple types of brackets. Algorithm to test if a given string of parentheses is balanced.
- Definition of postfix notation (informal). Examples
- Algorithm to evaluate an expression in postfix notation.
- Algorithm to convert infix to postfix using priority rules. (We defined in-coming and in-stack priority for operators and open bracket symbol.)
- General introduction to queue and project # 3 based on queues.

Review questions:

1) Exercise 3.3
2) Exercise 3.4 If both lists are of size N, what is the number of operations performed by your algorithm?
3) Write a function balance_check that takes as input a string of parentheses (using three types of them { }, [ ] and ( )) and returns true (false) if the string represents a balanced (not a balanced) parentheses. You can assume that you have access to a stack class.

4) In class, we presented complete code for evaluating an arithmetic expression in postfix (where we assumed that the operands are positive integers and operators can be one of + , – , * , / or **. You can download the complete code from the home page for the course. Test this program for some inputs and make sure that it works under the stated assumptions.
5) Extend the scope of the program for expression evaluation by adding an error check. When the input is not a valid expression in postfix, your program should output appropriate error message. (This can happen in the following cases: (a) when an operator is encountered, the stack has less than two operands. (b) when the input has been completely read, the stack has more than one item. Make appropriate changes in these two cases. You need not consider other errors such as illegal characters in the input.)
6) Extend the scope of the program for expression evaluation by allowing unary minus. Unary minus will be represented by the symbol @. Thus, for example, the following postfix expression: 5 6 4 – 8 * + 9 @ * will have the value –189. (Hint: when @ is encountered, just pop one value x off the stack, and push –x on to stack.)