COP 4530: Data Structures
Project 4

You are not allowed to use the Internet. You may only consult approved references[*].
This is an individual project.
This policy is strictly enforced.

You must submit a **hard copy** of all of the items requested below. You must also submit your *code*[†] to Canvas.

For full credit, the code that is submitted must:

- Use the specified signature, if applicable.
- Be implemented in a file using the specified file name, if applicable.
- Be correct (i.e., it must always return the correct result).
- Be efficient (i.e., it must use the minimum amount of time and the minimum amount of space necessary to be a correct implementation).
- Be readable and easy to understand. You should include comments to explain when needed, but you should not include excessive comments that makes the code difficult to read.
  - Every class definition should have an accompanying comment that describes what is for and how it should be used.
  - Every function should have declarative comments which describe the purpose, preconditions, and postconditions for the function.
  - In your implementation, you should have comments in tricky, non-obvious, interesting, or important parts of your code.
  - Pay attention to punctuation, spelling, and grammar.
- Follows ALL coding guidelines from section 1.3 of the textbook. Additional coding guidelines:
  - No magic numbers. Use constants in place of hard-coded numbers. Names of constants must be descriptive.
  - No line of the text of your source code file may have more than 80 characters (including whitespace).
  - All header files should have #define guards to prevent multiple file inclusion. The form of the symbol name should be `<FILENAME>_H_`
  - Do not copy and paste code. If you need to reuse a section of code, then write a function that performs that code.
  - Define functions inline only when they are simple and small, say, 5 lines or less
  - Function names, variable names, and filenames must be descriptive. Avoid abbreviation.
  - Use only spaces (no tabs), and indent 3 spaces at a time.
- Compile and run on the C4 Linux Lab machines (g++ compiler, version 4.8.2). *The shell script and makefile that I will use to compile and run your code will be posted on Canvas. Please note that I may use my own `main.cpp` file to test the code you submit.*
- Have no memory leaks.

[*] The list of approved references is posted on Canvas. You must cite all references used.
[†] Your code must compile and run on the C4 Linux Lab machines

Project Tasks

1. Create a file named `functions.h`.
   (a) [45 points] Write a function to convert a postfix expression to the corresponding fully-parenthesized infix expression.
   `string getInfixExpression(const string&)`, where the input parameter is a string representing an expression in postfix notation.
   - Examples:
     1. The expression `ab+` should produce `a+b` (note that `b+a` is incorrect)
     2. The expression `ab+cd-*` should produce `((a+b)*(c-d))`
   - There only characters of the input string will be either operators or operands
     - The operators used by the expression are + (addition), - (subtraction), * (multiplication), / (division)
     - The operands used by the expression are lowercase letters of the English alphabet.

     You may assume that the function always has a valid input. (A string with a single operand is a valid input.)
   - The only container that you are permitted to use to implement this function is one STL stack container.
     Read the input string left-to-right putting operands on the stack. When you encounter an operator, use the top two items from the stack to make an operand. After the string has been read in then the only item on the stack will be the infix expression.

   (b) [45 points] Write a function to implement the Radix Sort algorithm.
   `void radixSort(vector<string>& numbers, const int& digitsPerNumber, const int& radix)`, where numbers is the vector (array) to be sorted, digitsPerNumber is the maximum length of any number in the array, and radix is the base of the integers (e.g., base-10 numbers will have radix 10).
   The algorithm for radix sort is:

Algorithm 1: Radix Sort (numbers, digits, radix)

```
   /* Create a queue for each possible number that can be in a number. */
1  vector<queue<string>> buckets
   /* Iterate over the digits of each number, starting with the least significant digit */
2  for each digit of the numbers (i.e., 0 to digitsPerNumber), starting with the least significant digit do
       /* Iterate over each the numbers in the vector and put them in a bucket */
3      for each value, j, in the numbers vector do
4          int digitI = get the i^th significant digit of integer j
5          put integer j in the bucket corresponding to digitI
6      end
       /* Iterate over each of the buckets and pull out the numbers to put them back into numbers */
7      for each bucket (in order from 0 to radix) in buckets do
8          Pull out each number of this queue and put it back into the numbers vector, in order
9      end
10 end
```

   - If the base is higher than 10, then uppercase letters of the English alphabet will be used to denote digits.
     For example, in base-16 we use A to denote '10', B to denote '11', C to denote '12', D to denote '13', E to denote '14', and F to denote '15'.
     You may assume that for any set of numbers input to the algorithm, that the characters 0-9 and A-Z will be sufficient to represent the integer (i.e., the highest base will be 36)
   - You are permitted to use one STL vector container to hold r STL queue containers (where r is the radix of the integers to be sorted). No other containers may be used to implement this function.
   - To walk over the numbers vector (the loop on line 3 of the pseudocode), you must use an STL iterator.

2. [10 points] Write a main function to thoroughly test each of the functions from Task 1 above.
   Note that I will be creating my own `main.cpp` file to test your code from Task 1 above.