Data Structures and Algorithms

Programming Assignment 1

Department of Computer Science & Technology
United International College

Rubrics

Criteria for assessment	Performance levels				
	Excellent 10 / A / 4	Good 8 / B / 3	Satisfactory 6 / C / 2	Marginal Pass 4 / D / 1	Fail F / 0
Function test **(80 % weighting)**	All systems test case run successfully.	Most system test case run successfully.	Some unit test case runs successfully.	Only a few test cases successfully.	The code runs none.
Program structure **(10 % weighting)**	Needed program structures are evident.	Program structures are clear.	Program structures are obscure.	Needed program structures are lacking.	None.
Comment **(5 % weighting)**	Comments are adequately provided and are at levels of abstraction appropriate for conveying specifics about the programs.	Comments are mostly provided and at levels of abstraction appropriate for conveying specifics about the program.	Comments are provided somewhere, but at too low a level of abstraction to be of much use.	Comments are sparse or vague, and give little information about the purpose of the program or how it goes about carrying it out.	No comments and no information about the purpose of the program.
Code style **(5 % weighting)**	A clear coding style is evident, and consistently applied, greatly enhancing program readability	A clear coding style with mostly consistency in application, aiding readability in a majority of the program.	A clear coding style is hinted at, with some consistency in application, aiding readability in some of the program.	A clear coding style is lacking, or applied very inconsistency, with readability suffering accordingly.	None

Comments on the Rubrics

- You will get full mark for Function test if
  - Your code produces correct output for all our test inputs.
    - The test inputs are not provided to you.
    - Try your code against all possible inputs (that you can think of) to test correctness
  - No memory leak is found in any case
- Program Structure refers to
  - Reasonable file structure in the project
  - Reasonable placement of function declarations and implementations
- Code style includes
  - Reasonable naming of identifiers
  - Reasonable indentation
  - Code neatness

PROBLEM LIST

Problem 1 – List Methods

- Given the Linked list ADT introduced in Lecture 3, implement two more methods:
  - `InverseNodes`
  - `RemoveDuplicates`
- Submit the complete code set including
  - Struct definition
  - Declaration and implementation for the existing and the new methods
    - You may use the sample solution provided on iSpace or your own implementation of the existing methods (e.g., *InsertNode*)
  - A main function which runs your own test cases

InverseNodes

- void InverseNodes(Node** phead);
  - Inverse the order of all the elements in a linked list.
- Sample Input and output

Input List	List Update
2 --> 6 --> 5	5 --> 6 --> 2
Null	Null
6	6

RemoveDuplicates

- void RemoveDuplicates(Node** phead);
  - Deletes all nodes that have duplicate values, leaving only nodes with distinct values
  - You may assume that the node values are non-decremental in this method

– Sample Input and output

Input	List Update
1 --> 2 --> 2 --> 4 --> 6 --> 6	1 --> 4
Null	Null
6 --> 6 --> 6 --> 7 --> 7	Null
6 --> 7	6 --> 7

Problem 2 – ValidBrackets

- Complete function: bool ValidBrackets(char* str)
  - *str* is a string containing only '(', ')', '{', '}', '[', ']', '<' and '>'
  - Returns *True* if the input string is valid and *False* otherwise
  - In a valid string,
    - The brackets must match
    - The brackets must close in the correct order
- Sample Input and output

Input	Output
"{()}"	True
"(<)>"	False
"{()}["	False
""	True
NULL	False

Problem 2 – ValidBrackets

- Hint
  - You can make use of the Stack ADT
  - Consider what action you will take when you process the following characters in the string
    - '{', '[', '(', '<': opening brackets
    - '}', ']', ')', '>': closing brackets
- Submit the complete code set including
  - Struct definition
  - Declaration and implementation for every necessary method
  - A main function which runs your own test cases

Problem 3 - Uncompress

- Complete function: char* Uncompress(char* str)
  - *str* is a compressed string
    - The rule of compression is: *k(encoded_string)*, where the *encoded_string* inside the square brackets is being repeated *k* times and *k* is a positive integer.
    - You may assume that the input string is always of valid format
    - You may assume that the original data contain only English letters and that digits are only for those repeat numbers, *k*. For example, there won't be input like *"(c"* or *"21(4)"*.
  - Returns a string produced by uncompressing *str*
  - You may assume that the uncompressed string is no longer than 10000

Problem 3 - Uncompress

- Sample Input and output

Input	Output
NULL	NULL
""	""
"abc"	"abc"
"a1(b)c"	"abc"
"a11(c)d"	"acccccccccccd"
"3(a)2(bc)"	"aaabcbc"
"3(a2(c))"	"accaccacc"
"2(abb3(cd))ef"	"abbcdcdcdabbcdcdcdef"

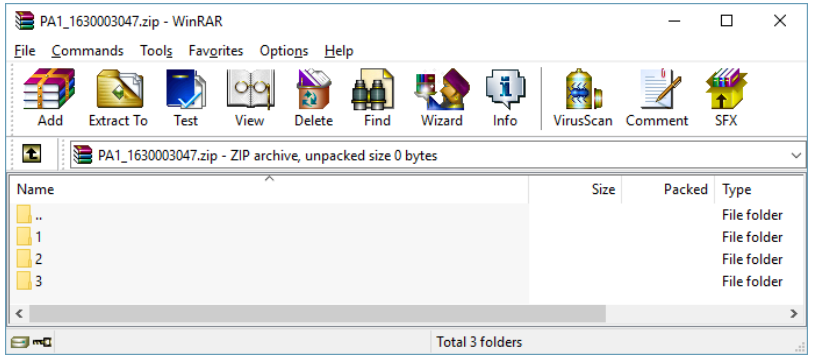
Problem 3 - Uncompress

- Hint
  - You can make use of the Stack ADT
  - Consider what action you will take when you process the following characters in the compressed string
    - a digit
    - "("
    - a letter
    - ")"
  - Attention: Stack push/pop reverse the order of a string.
- Submit the complete code set including
  - Struct definition
  - Declaration and implementation for every necessary method
  - A main function which runs your own test cases

Submission

1. Put the complete set of source files for each problem into a folder named with the problem ID.
   - For example, the code set (.h and .cpp files) for Problem 1 should be in folder 1.
2. Compress all the folders into a zip with name: PA1_<your student id>.zip
   - For example, if you are 王安博, you file should be:

   PA1_1630003047.zip
3. Submit the .zip file to iSpace.



Plagiarism Policy

- We will check between everyone's submission.
- We will check with online solutions.
- If copies are found, everyone involved gets ZERO mark.