**Caesar Cipher Lab**

You will be deciphering a given message. It is encrypted with a Caesar Cipher (https://en.wikipedia.org/wiki/Caesar\_cipher) that increases by 2 after every 3 characters (including symbol characters, which are not encoded), starting at key = 5. You should save this message to a file using a text editor. Then, your program will prompt the user for the name of the file, decrypt the message stored in the file, and then write the decrypted message to a new file called solution.txt.

A sample message talking about getting 5 points of extra credit appears below:

Htsnyhcdjwlevbah! Pfl zxo afsb dwusb srnsyz!

(We may try other sample messages when we test your program.)

HINTS:

* Make sure your key does not exceed 26, the number of letters in the alphabet. Consider using modulo division to help.
* Make sure you "roll" around the alphabet correctly. For example, if the input letter is "D" and the key is currently 5, subtracting 5 from D's code would result in you being 1 before A (64). You should add back the code for the letter Z to roll to the end of the alphabet.
* For JavaScript/TypeScript coders: You can convert letters to their numerical equivalent numbers by using String.fromCharCode(...) and String.charCodeAt(...). For example, String.fromCharCode(65) produces 'A' and 'A'.charCodeAt(0) produces 65.
* For coders in other languages: You can do math directly with letters, such as 'D' - 'A'.

# Wrapper & Iterator Lab

Write a fake vector class. The class should use the built-in vector/list/array in your language of choice to effectively "wrap" inside of it an existing vector, while presenting a limited vector-like functionality to the user. Here is a TypeScript interface that represents what your class should look like:

interface Vector<T> extends Iterable<T> {  
 get(index: number);  
 set(index: number, value: T);  
 length: number;  
 push(value: T);  
 pop(): T;  
 insert(index: number, value: T);  
 // remember to implement the iterable functionality  
}

If you are working in another language, you may translate this interface into a C# interface, Java interface, VB interface, or C++ pure virtual class, because the class you write must implement/inherit the interface/class that is shown above. If you are programming in raw JavaScript (ES6/ES2015), you cannot implement the interface, but you should carefully program your class to work identically to the presented interface.

Please ensure you implement an iterator (or Enumerable, for C# programmers). You may use generators if your language supports them, or you may implement the iterator classes manually if you choose to do so.

# Bouncy Bubble Sort Lab

In standard bubble sort, the algorithm makes multiple passes through the list, swapping items so that the smaller value comes first and the larger value comes second.

Implement a "bouncing" bubble sort. In this version of bubble sort, instead of making passes through a list that starts at the beginning and runs through to the end, you should reverse the direction of each pass. That is, if the first pass starts at the beginning of the list and runs through to the end, the second pass would run from the end of the list back to the beginning, and then the third pass would start at the beginning again.

Assume items in the list are of the type integer.

# Sorted Set Lab

Write a sorted set data structure using binary search trees. Then, read in numbers from a file named *infile.dat*, inserting them into an instance of your sorted set. You will then prompt the user for a value, and search the tree to determine if the value is found in the tree. If the value is present, output "Yes" and no other text. If the value is missing, output "No" and no other text.

SortedSet {  
 isEmpty()  
 add(value)  
 remove(value)  
 contains(value)  
}

Example:

**infile.dat**

1, 2, 3, 4, 5, 6, 7

**Program**

Sorted Set A Contains 1, 2, 3, 4, 5, 6, 7

User Input = 9

**Output**

No

# Red-Black Tree Lab

There's many instances of red-black trees available throughout the internet for a variety of languages. Pick one and include it with your source code if the language you use does not have a built in one.

Using a red-black tree, build a sorted dictionary. This dictionary should be able to insert key/value pairs, retrieve values associated with keys, delete keys, and inform you whether or not a key exists in the dictionary. You do not need to do any error checking to determine if a key already exists when someone attempts to retrieve/delete an existing key.

Demonstrate that your sorted dictionary works by writing your program to insert into the dictionary the following key/value pairs:

|  |  |
| --- | --- |
| **Key** | **Value** |
| hello | world |
| goodbye | everyone |
| name | student |
| occupation | student |
| year | 2016 |
| gpa | 4.0 |
| lab | yes |
| assignment | no |
| department | cs |

Then, retrieve the value of *gpa*and *department* and display it on the screen.

# Topological Sort Lab

Implement the pseudocode presented in the lecture for both creating a graph and for topological sort. Put each algorithm into a separate function. Construct a data structure to store a graph, which can be based on an adjacency matrix, adjacency list, or any other underlying structure you create.

When the program starts, read in a graph from a file called infile.dat. Then, print out **two different**topological orderings for the graph to the screen. The graph we input to test is guaranteed to have at least two valid orderings, and you do not need to error check for that. You will need to modify the topological sort algorithm slightly to allow it to come up with a different valid topological ordering the second time, by tweaking one of the *id(0)* nodes that it selects.

# BFS Lab

Create or use any graph data structure that reveals the graph using an adjacency matrix instead of an an adjacency list. Implement the breadth-first search algorithm as described in the lecture notes, using a queue to represent S'. Just like you can generate *dfn*for a depth-first search, generate a *bfn*number for this breadth-first search. Use the bfn number to represent S, do not use a separate data structure.

Read in a graph in the same format as last week from infile.dat. Assume node 0 is the start point (s). Output a valid *bfn* number for each node to the screen in the following format:

*node#* *bfn*

For example, a 3-node graph might generate the following output:

0 1  
2 3  
1 2