# COSC2320: Data Structures and Algorithms
# HW1: Editor with Doubly Linked Lists

## 1 Introduction

You will write a C++ program that will implement basic functionality of an editor. Since creating an interactive editor would be too time consuming you will program a command-line editor which does only insertion and deletion of words. This editor is similar to early editors and UNIX utilities.

You must use **doubly linked lists** to store a list of words; you cannot use arrays. Words will be inserted and deleted with dynamic memory manipulation via linked lists. The first major requirement is that your program produces correct results. The second requirement is that it manages memory efficiently. You must develop your own C++ classes to solve the homework. Therefore, you are not allowed to use the STL Library.

## 2 Input

The input is a text file, with an unbounded number of different words, separated by spaces, commas or periods. The input file can have several lines terminated by carriage return. You can assume words will be at most 30 characters long and you must store them as char *, or char [].

## 3 Program and input specification

The main program should be called "editor", which will as input parameter one file with list manipulation commands (i.e. a script file).

```
editor script=mytest.script
```

The script file will contain commands to read, write, insert and delete words from a doubly linked list. Your program must read the words and store them in a linked list. *After* the list is processed it must be written to the output file. If "insert(list,'word1/word2')" command is given word2 must be inserted after word1 in the list; you should insert one space as separator after word2 by default. If "delete(list,'word1')" is given word1 must be deleted from the list. After executing the insert/delete commands your program must write the modified linked list back to disk into the output file.

# 4 Examples

## Example 1

```
in.txt
--------------
abc wrt cvbb
<eof>

example1.script
-------------------------
read(L1,'in.txt')
insert(L1,'abc/xyz')
write(L1,'out.txt')

out.txt
----------------
abc xyz wrt cvbb
```

## Example 2

```
in.txt
--------------------
dog tiger lion zebra giraffe
<eof>


example2.script
--------------------
read(L1,'in.txt')
insert(L1,'lion/elephant')
insert(L1,'elephant/cheetah')
write(L1,'out-fwd.txt',forward)
write(L1,'out-rev.txt',reverse)

out-fwd.txt
-------
dog tiger lion elephant cheetah zebra giraffe
<eof>

out-rev.txt
-------
giraffe zebra cheetah elephant lion tiger dog
<eof>
```

## Example 3

```
in.txt
--------------------
Jaguar Ford Renault Citroen Audi Mercedes Chevrolet
<eof>

example3.script
----------------
read(L1,'in.txt')
insert(L1,'')
insert(L1,'Jaguar/Peugeot')
delete(L1,'Citroen')
delete(L1,'Audi')
```

```
delete(L1,'Chrysler')             /* error */
write(L1,'out-fwd.txt',forward)
write(L1,'out-rev.txt',reverse)
write(L2,'out.txt')               /* error */

out-fwd.txt
-------------------
Jaguar Peugeot Ford Renault Mercedes Chevrolet
<eof>

out-rev.txt
-------
Chevrolet Mercedes Renault Ford Peugeot Jaguar
```

# 5 Requirements

- Use doubly linked lists. Singly linked lists are not acceptable. Arrays are forbidden to store the list of all words. Each word and separating string will be stored as one node in the linked list. Numbers can be treated as separator strings, but they will be used as inpuit in future homeworks.
- Your program will be tested with input files of very different sizes (2 words, 100 words, 5000 words, etc). No, we cannot tell you how large the input file will be.
- Output: use one space as separator by default. Do not use other separators. If you find carriage returns (end of line), leave them ”as is”. Remember your programm will be automatically tested.
- Parameters: You can use the Command Line Parser that is provided in the TAs homepage. The TAs will also provide a class to parse an script file with commands.
- Output must be written to a log file (call it 'log.txt'). In this file you can display error messages and # of bytes following this format (size=120). You should display the size of the list every time there is an insertion or deletion. You should also display it righ after the read() and write() commands.

  Your program must display the number of bytes used after reading the input file and after writing the output file. You can use ”sizeof()” in C++ or hard code the number of bytes per variable. This number of bytes will show you are dynamically allocating memory. Notice it is invalid to read the input file several times to “guess” the array size.
- Optional: empty files, repeated words, numbers, strange symbols (to be tested in future homeworks).
- Your program must ignore input words not found in the linked list.
- Your program will be called several times using a chain of files. Primitive operators: Notice that with insert() and delete() it is feasible to perform all edition operations.
- Your program must write the modified list to disk. Your program must preserve the original separators given in the input file.
- Source code verification:

  Your program will be automatically scanned for array declarations. If the TAs detect the words from the input files are stored on large arrays you will get 50 points maximum. Notice it is OK to use arrays for the program parameters (given in the command line), but not to store the entire list of input words.

If your program gets a 50 score or higher and it complies with the doubly linked list requirement it will be checked for code plagiarism. The TAs will also automatically compare your source code to other students' source code as well as popular programming sources like Wikipedia, Stackoverflow and other universities. Notice we also have a repository of all previous editions of this course, in case you intend to ask a friend for help.

- You cannot use the STL library.
- Your program should write error messages to the log file. Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input files, zeroes and unexpected symbols.
- Test cases. Your program will be tested with 10 test cases, going from easy to difficult. You can assume 80% of test cases will be clean, valid input files. If your program fails an easy test case 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points maximum. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.