

## Programming Assignment #1

(Individual Programming Project)

**WARNING:** Programming assignment #1 is due at the end of week 2, and will **not** be accepted **late**.

**Rationale:** You must successfully complete individual assignment #1 before you can join a team for assignment 2. Teams must be formed in time to complete the design in week three. Therefore, if you are not able to complete assignment #1 on time, you will not be allowed to move forward in this course.

### Part 1: The Program

Write a program to analyze words in a text file. The text file will contain paragraphs of sentences of English words and possibly numbers. The program will:

- Read, count and organize the words into lists, based on starting letters (words containing any digits will be ignored).
- Determine the number of unique words in the text file that begin with each alphabetic letter.
- Determine which letter or letters begin(s) the most unique words.

The program must be modular (i.e. broken down into logical functions) and use correct parameter passing. Use of global variables will NOT be allowed. And object-oriented programming (classes, objects, templates, STL, etc) may NOT be used

### Data Structure Overview

A double linked list is represented by a pointer to the top node in the list. This program will maintain a double linked list for EACH letter of the alphabet.

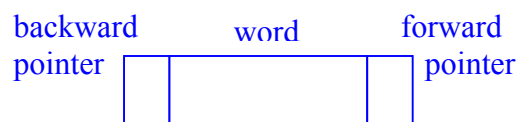
So you will need to use an **array of pointers** to implement this program. Each pointer will represent the head of one list (for 26 separate lists).

*You must implement this data structure from scratch (no use of templates from the STL allowed).*

The individual **nodes** in each list should contain one word and two pointers (one pointer pointing forward to the next node and one pointer pointing backwards to the previous node) only.

C-string usage is ONLY required for storing the command line **file name** parameter. The words in each list should be stored using the C++ **string** datatype.

#### Sample Node:



Initially, the lists will not contain any nodes. After the lists are built, the backwards pointing pointer of the **first** node in each list will be NULL, and the forward pointing pointer of the **last** node in each list will be NULL.

Each list should contain **word nodes only**. Do **not** store a word count with the list, or any

other additional information.

Since each double linked list in the array will contain words starting with a particular letter, the array will have 26 cells, indexed by the integers 0 to 25 (for 26 letters).

To create an array of pointers:

- a) Define a **struct** type that contains the **node** data as described above (string word, pointer forward, and pointer backwards)
- b) Create an **array of pointers** to the **node struct** type, just like you would any other array. Since it will be indexed by letters, the size should be 26.  
(NOTE: The **new** operator is **not** needed to define the array)
- c) Initialize each pointer in the array to NULL, to represent 26 empty lists.  
(NOTE: Again, the **new** operator is not needed to initialize the array).

So each cell in the array will be treated as the TOP/FIRST pointer for a list.

### Implementation Details

The text input data filename will be given as a **command line argument** to the program.

Your program should validate that a filename argument was given on the command line, **and** that the file is valid (i.e. can be opened). The program should give an error message and exit gracefully, if the filename argument is missing or the file does not exist.

After validating the data file exists, your program should initialize all the lists to be empty.

Then process the input data file. Read one word at a time from the input text file. Words will be separated by whitespace.

For each word the program reads, it should:

- 1) Determine if the word contains any digits.  
If the word contains any digits (examples: **3B**, **RS232**), then the word will not be considered a valid word. The program should ignore the entire word and should not count the word in any of the word counts. The program will simply read the next word.
- 2) Strip off any extra spaces and punctuation ( . , " ! ? ( ) etc. ) from the beginning and ends of each word.

Note that apostrophes and internal dashes within words should not be stripped (for example, **she's** and **part-time** are valid words).

Valid words will all be common English words, found in an English dictionary.

- 3) Format the word in either all uppercase or all lowercase (your choice).
- 4) Determine which double linked list the word belongs in, using the first letter of the word.

There will be one cell in the array of pointers for each letter of the alphabet. The array will be indexed by the integers 0 to 25 (for 26 letters). The program will determine which index to use by looking at the first letter of each word processed. For example, all words starting with the letter 'A' will be stored in the list at index 0. Words starting with 'B' will be stored in the list at index 1, etc.

So the program needs to map the starting letter of each word to the correct array index. If the word starts with 'A', it should convert the 'A' to a 0 (zero) to get the right index into the array of pointers. And 'B' would be converted to a 1, etc.

The easiest way to do this is to subtract 'A' from the letter you have:

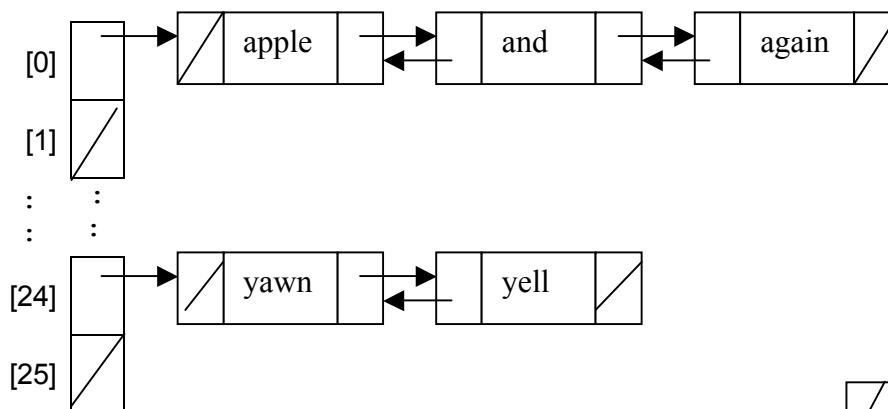
$$\begin{aligned} \text{'A'} - \text{'A'} &= 0 \\ \text{'B'} - \text{'A'} &= 1 \\ \text{'C'} - \text{'A'} &= 2 \\ &\vdots \\ \text{'Z'} - \text{'A'} &= 25 \end{aligned}$$


5) Each list will contain only unique words (no duplicates). So the next step will be to check if the word is already in the double linked list that corresponds to that starting letter.

If the word is not yet in the list, the program should add the word to the **top** (beginning) of the appropriate double linked list (i.e. only add words that are not already on the list).

6) As you read each word from the file, keep track of the total word count (number of valid words read from the file), excluding words containing digits. Do **not** track the unique word count – it will be calculated later with the results.

#### Diagram of Sample Array:



NOTE:  means NULL

**After** all of the words in the file have been read, processed, and inserted into the proper list, output the results as follows:

Display the filename and total number of words processed (counting duplicates, but not counting any ignored invalid words that contained digits).

Then for each word list:

Using the **forward-pointing links** in your double linked list, the program should begin at the top of the list and should count the words as it traverses the list. Then display the number of words counted in the list (i.e. unique words for one letter).

After counting the words, the program should be at the last node. Using the **backward-pointing links** in the double linked list, the program should display the words that start with that letter as it traverses the list backwards. Note that traversing backwards will cause the words to be output in the order that they first appeared in the file.

**NOTE: If a list is empty, do not display the count OR the word list.**

Keep track of the total unique word count by summing the counts from each list. Then after displaying all counts and word lists, display the number of **unique** words in the whole file (i.e. the total number of words in all the lists).

Finally, determine which letter began the most unique words (i.e. the letter whose list contains the most words). Display the letter and word count. If there is a tie or ties, output **all letters** with the largest count.

### Sample Output

```
Results for file memo.txt:  20 total words processed
```

```
4 words beginning with 'a'/'A':
```

```
    A, ALSO, AND, ALWAYS
```

```
2 words beginning with 'b'/'B':
```

```
    BE, BOTH
```

```
2 words beginning with 'f'/'F':
```

```
    FULL-TIME, FOR
```

```
4 words beginning with 'w'/'W'
```

```
    WHEN, WHERE, WHAT'S, WHOSE
```

```
1 word beginning with 'y'/'Y'
```

```
    YOUTH
```

```
There were 13 unique words in the file.
```

```
The highest word count was 4.
```

```
Letter(s) that began words 4 times were
```

```
    'a'/'A'
```

```
    'w'/'W'
```

### Part 2: Efficiency Analysis

Analyze your program (algorithm analysis is covered in week 2) as follows:

a) Analyze the efficiency of each function (except **main**) using:

**n** = number of words in the text file

Explain your logic for calculating the efficiency of each function within the analysis

document. Also include the resulting **f(n)** efficiency for each function in the comment header of the function.

NOTE: The word size should not affect the efficiency, as it can be reduced to a constant representing the "average" English word size.

b) Using the efficiencies for each function from (a) above, combine them to determine the efficiency of the **main** function. Use this to determine the big-O efficiency for the entire program. Explain your logic.

c) Describe what type of data file would produce the best-case, average-case, and worst-case for this program (i.e. What types data, for the same value of **n**, would affect the runtime to give the best, average and worst cases?).

### Part 3: Items Due

By **midnight Sunday of week 2:**

Your **program source code** and **program analysis**.

Also include any data files you used to test your program.

#### WARNING:

Programs that do not compile, are not modular, or that use ANY global variables, will NOT be accepted.

Submit your assignment, to the **Week 2 Drop Box** (located in the *Assignment Drop Box Folder* under the *Week by Week*) attaching all of the necessary files.

Before submitting your program and data files, you MUST name them as follows:

**Lastname-Assn1-DLLProg.cpp**

**Lastname-Assn1-BigOAnalysis.doc**

**Lastname-Assn1-Testdata.txt**

If you have multiple test data files, append a letter on the end of each filename.

Examples:

**Smith-Assn1-DLLProg.cpp**

**Smith-Assn1-BigOAnalysis.cpp**

**Smith-Assn1-TestdataA.cpp**

**Smith-Assn1-TestdataB.cpp**

### Part 4: Grading

The grading rubric that will be used for this assignment is included on the next pages.