The objectives of this project are (1) giving you additional practice with the linked list data structure, and (2) implementing two sorting algorithms to operate on a linked list so that their performance (speed) can be compared.

This project can be done by adding 3 new commands to project 2. The format of this assignment and its grading policies, requirements, etc. are the same as project 2 except where specified.

1 New Command Specifications

Add the following 3 new commands to project 2: The program name must be proj3.

1. :selectionsort

First, call the gettimeofday() function before doing the sort so that the time of day can be saved. This time value will eventually be printed and used to calculate the time it took to do the sorting.

Use the following following code: (Study man gettimeofday output to find out what it means.)

```cpp
#include <sys/time.h>
#include <iostream>
using namespace std;
...
struct timeval time_started;
struct timeval time_finished; // needed for later.. see below
gettimeofday( &time_started, 0 );
```

Second, sort the main list into non-decreasing lexicographic order using code that implements the selection sort algorithm explained on pages 495-602 of Main and Savitch’s textbook. You must use strcmp() to compare the C-strings for sorting. Main and Savitch’s explanation is in terms of sorting an array. However, your job is to adapt the selection sort principle so it applies to a linked list.

Selection sort in an array uses two index variables: One keeps track of the position within the sorted sequence where the element now being selected will finally be put. The other keeps track of the positions of elements to be compared with the so-far-smallest element among those not placed in their final positions. See the Lecture 12 for notes on using selection sort to sort the characters in an array. Both index variables are used to traverse the array.

You will use two pointer-to-node variables to keep track of the same things as the two array index variables. The usual way to use pointers to traverse a linked list is easy to program.

Selection sort operates by comparing certain pairs of sequence items and swaps them if they are out of order. The most efficient way to swap the C-strings in the proj2 and proj3 data
structures is to swap the addresses of the char arrays stored in two linked list nodes. So, you must use this most efficient way. For the sake of simplicity and efficiency, the selection sort code **must not modify any pointer to NODE variables**, only pointers to chars.

These topics will be covered in Lecture 12.

The :selectionsort command must leave the “cursor” (described in project 2) indicating the end of the main list.

**Third**, Call gettimeofday() again, saving the data in a different struct timeval. Then, print the time of day before sorting, and then the time after sorting. Use the code:

```cpp
cout << "Time before selection sort=
   " time_started.tv_sec << "seconds+
   " time_started.tv_usec << "microseconds" << endl;
cout << "Time after selection sort= " //..... etc..
```

This will enable you to observe exactly how time elapsed during the period when the computer was busy running your selection sort code.

**Finally**, compute the number of microseconds of time that elapsed between the two time of day reports by a combination of addition, subtraction and multiplication operations on the two fields of each of the two struct timeval’s filled by gettimeofday(). (Your program will have to multiply something by a million (1000000) since there are 1 million microseconds in 1 second.) Print the result exactly as follows:

*The wallclock time to run selection sort on <n> elements was <mmm> microseconds.*

Here <n> signifies the number of lines in the list (that was sorted) and <mmm> is the number of microseconds, both printed as decimal integers.

You and our staff will be able to test your :selectionsort operation by inputting some lines, trying :printall to verify those lines were stored properly, running :selectionsort, and then re-running :printall to see whether the lines are properly sorted.

If the first :printall after input doesn’t report the very same lines that were typed in for the test, your program will be rejected right away and get 0 points for functionality.

Question: What is the disadvantage of printing the “Time before selection sort=” message before running the selection sort algorithm?

2. :split

**First**, make the cut-list empty.

**Then**, move the best approximate second half of the main list to the cut-list. (“Best approximate second half” is as close to half as possible if the main list has an odd number of lines, and exactly half otherwise.)

Like the :selectionsort command, the :split command must leave the “cursor” indicating the end of the main list.

(The purpose of this item is to give you credit for implementing a key operation that will be required for implementing mergesort. Do not attempt to go on to finish :mergesort until you have correctly working code for the split operation.)
Here’s a good way to test :split

(a) Input some number of lines, sometimes 0, 1, 2, a larger even number and a larger odd number, for different tests. (You should do at least 5 tests.) (A good choice of sample lines is “One”, “Two”, “Three”, “Four”, etc.)

(b) Try :printall to check whether your program inputs the lines correctly.

(c) Try :split

(d) Try :printall to check that the first half (or approximately half if the number of lines is odd) of the original lines remain in the main list.

(e) Try :paste and then :printall to check that the originally inputted lines, in their original order, are now in the main list.

Word to the wise: This is how we are going to test your :split!

3. :mergesort

Provide the same functionality as :selectionsort, except use the mergesort algorithm to do the sorting. Mergesort for arrays is described on pages 608-620 of Main and Savitch. However, your job is to adapt it to linked lists. (Implementing mergesort of linked lists is easier than of arrays.)

For this project, the mergesort algorithm must be implemented using recursive code. Recursion will be introduced in Lecture 12. When we apply apply, in the simplest way, the “divide-and-conquer” principle to the sorting problem to obtain a recursive sorting algorithm, the algorithm we get is mergesort. This topic will be covered in Lecture 14.

The recursive function that does mergesort should have (at least):

(a) One parameter: Address of the first node in the linked list to be sorted. If its parameter is NULL, the function should simply return NULL.

(b) Two local pointer-to-node type variables, to hold the addresses of the two sublists supplied by the split function.

(c) Two recursive calls, one for each sublist.

(d) One call to a function to merge the two sublists after the two recursive calls both return, each having sorted one of the sublists.

(e) The mergesort function should return the address of the first node of the now sorted list.

If you have not implemented the :split command with a function that takes one address parameter and somehow supplies the addresses of the two sublists, you will have to adapt the original splitting code to create such functions.

Like for :selectionsort, the C-string library function strcmp() must be used to compare the C-strings for sorting.

Of course, :mergesort must report the the gettimeofday() results and their difference just as :selectionsort did, except for name of the algorithm is “mergesort”.
:mergesort must set the “cursor” exactly specified for :selectionsort.
:mergesort will be tested the same way as :selectionsort.

We will also check that your actually implemented and called a recursive mergesort algorithm, rather than cheat by calling the selection sort code again (or a library function)! (We will also be able to tell by the time reports.)

2 How to Do This Project; Some Graded Items!

All project specifications and rules from the handout on doing programming projects, and Project 2 apply to Project 3 as well. (RCS databases, build script, header/implementation/main file separation, pre and post conditions, turning in a directory, etc.)

Modify your :help and :about outputs to make them relevant to project 3.

You might as well keep all the commands from project 2 in your project 3 program. However, the only ones that will be used for project 3 grading are: :about, :help, :quit, (non-command line insert), :printall, :paste, and of course :selectionsort, :split and :mergesort.

Reminder: :selectionsort and :mergesort MUST NOT PRINT the sorted lines, only the time reports. To see the sorted results, you must give the :printall command. My reasons are to make time reports not include the time for printing and the make time reports easy to see when the number of lines is quite large (thousands or more).

Since the selection sort, split and merge sort operations modify the main list, the cut-list and the cursor, they should be implemented as member functions belonging to editorCore. You may want to make these member functions call other functions that do the appropriate operations on linked lists of C-strings identified by the addresses of their first nodes.

Selection sort merely moves C-string addresses from one linked list node to another. Mergesort just splits one linked list into two, and, during the merge operation, moves a linked list node from the front of one list (one of the two lists being merged) to the end of a third list (which accumulates the result of merging). (Of course, many splits and merges will be done because of the recursion.) This analysis proves that NONE of the 3 new operations for Proj3 should ever run a C++ new or delete operation!! If you think you need to code new or delete, you are probably misunderstanding something, so ask the professor or a TA.

Submit a single directory to project name proj2 using “turnin-csi310 -c csi310 -p proj3 Directory ”. Verify with “turnin-csi310 -v -c csi310 -p proj3”.

Future Work

A subsequent project will include implementing another sorting algorithm AND obtaining and analyzing time reports for sorting files of lines with various sizes. Some will have thousands or more of lines!

You can easily make this project program sort the lines from a file and report the time to do it by:

1. Prepare the file named for example “testinput.txt” by appending the command :selectionsort (or :mergesort) after the last in the file.
2. Supply the file as input to your project program by *redirecting* the standard input to `proj3` from the file. This is done by the following shell command:

```
proj3 < testinput.txt
```

If you followed the instructions, you now should only see the time and time difference reports.