The objective of Lab 4 is to observe the activation records discussed beginning 6 pages into DSO Chapter 9 on recursion. In the lab you will observe these activation records when the computer runs the recursive write_vertical function discussed at the beginning of Chapter 9. So read all these pages first!

You will find an expanded version of the “vertical” program from ITSUNIX directory ~acsi310/Lab4. Study the source file vertical.cxx Copy, build (remember to use the -g option!), and run it under the debugger.

As usual, set break main first, then activate run. Scroll to the body of write_vertical() and set another breakpoint, or simply command “break write_vertical”. Set a third breakpoint before the last line of code in write_vertical(). Finally, click or command continue to get the body of main() to run.

Observe how the program waits for you to type a positive number input. After you enter a sample number, the second breakpoint will be hit. At that point, pull down the “Status” menu and select “Backtrace”

Give the “continue” command a few times and observe what happens. The “backtrace” is really a report of what activation records, also called stack frames, are currently in the execution stack. To see a more detailed record, give the “backtrace” and the “where” commands in the text command window.

Pull down the “Data” menu and command ddd to “Display local variables” and “Display Arguments” For a really detailed view of the current stack frame, give the text command “info frame” You can command ddd to put this frame information “on display” by checking “All about selected frame” in the “Status Displays” window that appears when you active “Status Displays” on the “Data” menu.

Observe how our one local variable write_verticals_LOCAL changes as you progress through running the program. Sometimes use the “next” command instead of “continue” all the time. When does a display change indicates that ACTUAL VARIABLE changes, and when does only its value change? Tip: Put the following expression on display: “&write_verticals_LOCAL”

You probably will have to restart the program several times using the “run” command to observe what we are looking for. It is handy that all the breakpoint and display settings are kept for successive program runs.

Another important exercise: When the Backtrace shows several frames of write_vertical() activations, experiment with the “Up” and “Down” operations to observe the changes in the display window.

(When you are debugging for real, the “Up” operation is handy for finding out about the parts of the program that called functions leading to the spot where the debugger stopped the program. This is important to know when you want to figure out why your program crashed and left a “core” file. To do that, (next time you get a core file), run ddd as follows: ddd executable-file-name core and ddd will show you the state of the executing program at the time it crashed!)

Checkoff Credit—Due Friday night, March 11—Write a lab report in a text file, no more than 300 words, and submit it to project Lab4
Project 3

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

A build script named `build.sh` must be supplied, as it was for previous projects. When executed in its own directory, `build.sh` should produce an executable file named `sorter`.

When `sorter` is executed, it runs first the “input phase”, second the “command phase”, and it finally exits upon processing the “exit” command.

1 Input Phase

During the input phase, `sorter` reads (from standard input) lines of input and stores them as strings in a singly or doubly linked list in the order they are read.

Each normal input line is up to 90 characters in length including the newline. Your `sorter` could read them into a char array using the code given for reading the City names in Project 2 and then produce a new C-string to store each input line, or it may store them in any other other way (such as using C++ STL strings!)

However you implement the input, `sorter` must not “crash” on abnormally long input lines. For simplicity, it may “exit gracefully” by printing the message “Input line too long. Exiting.”. Or (another option, your choice) you may choose to recover after printing the message “Input line too long. Ignored.”. Or (a third option) you may remove the 90 character limitation. Your program will be tested against excessively long input lines and if it “crashes” you will lose one test-case worth of points.

An unusual requirement: Your program should “ignore”, i.e., skip over and not store, empty input lines (which consist of the newline character only). The reason is you will be invited to try your program on large text files supplied by Project Guttenberg [http://www.gutenberg.net](http://www.gutenberg.net) and they have a lot of boring empty lines. It is amusing to see just how fast certain algorithms on computers really are, and what happens to great literature plus publisher’s notices when their lines are sorted together lexicographically. More seriously, this project is the start for writing concordance computing software, a CSI402 level assignment.

The end of input is marked by a single line `==end==` whose only characters are those 7 characters followed by the newline. Upon detecting this input line, the program should not store it but should print the message on one line:

```
nnn non-empty input lines stored.
```

where `nnn` is the number of non-empty input lines read and stored.

2 Command Phase

Recognize and do the specified action for each command. Each command will start with a colon:

1. `:exit`

   Exit the program.
2. :printforward

Print all the non-empty input lines in the order they were read.

3. :selectionsort

This command should use your implementation of the selection sort algorithm to re-arrange the list of stored lines so they are in non-decreasing order based on the result of comparisons done by the C-string library function `strcmp()` or equivalent. The sorted list should replace the originally read list in your data structure, so :selectionsort may be run again.

Note that :selectionsort should not print the lines, either before or after sorting. Instead, it should print a report of the approximate time that elapsed during the sorting operation.

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 elapsed time.

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

```c
#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 list into non-decreasing lexicographic order using code that implements the selection sort algorithm explained in a previous lecture, in DSO under project 6 at the end of Chapter 5, and also in section 13.1. You must use `strcmp()` or equivalent to compare strings for sorting. Some explanations were 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 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 C-strings (or other implementations) in a linked list is to swap the addresses of the strings 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 (or strings).

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:
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 :printforward to verify those lines were stored properly, running :selectionsort, and then re-running :printforward to see whether the lines are properly sorted.

If the first :printforward 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?

4. 2 Commands: :split and :printsprint

The :split command must construct a second linked list implemented by the same data structure (type) as your first linked list. Then, it must move the best approximate second half of the original list to the new 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. :split must work even if the original list has 0 or 1 element, so :split it makes one or both result lists be empty.)

(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.)

You will get no credit for the :split command unless it can be tested with the :printforward and :printsprint commands.

The :printsprint command should do the following in order:

(a) Print the second list, one stored line per output line.
(b) Print a blank line.

If :printsprint is given before a subsequent split, there will be no second list. So it should make no changes but print “Error: No split list to print!” and go on for more commands.

On the other hand, if :split is given when there is a second list as the result of a previous split, it should similarly make no changes print “Error: Split done already!”
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 :printforward to check whether your program inputs the lines correctly.

(c) Try :split

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

(e) Try :printsplit to check that the second half (or approximately half if the number of lines is odd) of the original lines are now in the second list, in their original input order.

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

5. :mergesort (Study recursion and do Lab4 first!)

Provide the same functionality as :selectionsort, except use the mergesort algorithm to do the sorting. Mergesort for arrays is described in section 13.2 of DSO. However, your job is to adapt it to linked lists. (Implementing mergesort of linked lists is easier than of arrays.) This data structure approach is described our current lecture notes.

For this project, the mergesort algorithm must be implemented using recursive code. Recursion will be introduced in lectures. When we 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 the lectures.

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() or its equivalent must be used to compare the C-strings or other strings for sorting. The string comparison operation used in the merge operation must be exactly the same as the one used for selection sort, so that differences in elapsed time will not depend on how the strings were compared.

Of course, :mergesort must report the the gettimeofday() results and their difference just as :selectionsort did, except for name of the algorithm is “mergesort”.

5
: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.)

3 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 to specify function documentation written in header files, turning in a directory, etc.)

Full credit for modularization requires at least 3 different implementation files for (1) the user interface/input phase reader (2) the selection sort algorithm implementation and (3) the merge sort algorithm implementation. You are encouraged to use more, but that’s optional.

You might fashion your Project 3 program from your Project 2 program, or start from scratch. Reminder: :selectionsort and :mergesort MUST NOT PRINT the sorted lines, only the time reports. To see the sorted results, you must give the :printforward command. My reasons are to make time reports not include the time for printing and to make the count of lines sorted and time reports easy to see when the number of lines is quite large (thousands or more).

Selection sort merely moves 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 command operations for Proj3 should ever run a C++ new or delete operation!! The new operation is of course important during the input phase. 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 Proj3 using “turnin-csi310 -c csi310 -p Proj3 Directory”. Verify with “turnin-csi310 -v -c csi310 -p Proj3”.

Future Work

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 line “==end==”, followed by a line with the command :selectionsort (or :mergesort), and finally the :exit command at the end of the file.

2. Supply the file as input to your project program by redirecting the standard input to sorter from the file. This is done by the following shell command: proj3 < testinput.txt

If you followed the instructions, only the line count, day time and elapsed time reports will be printed.