ECE 2230: Computer Systems Engineering **Fall 2021**
Machine Problem 3 **Due: 11:59 pm, Thursday, September 23**

The goal of this machine problem is to extend the two-way linked list ADT from machine problem 2 to include five algorithms for sorting a list. Extend machine problem 2 to include the new commands:

SORT *x*
APPEXP ep_id

SORT *x* sorts the exposure queue in ascending order based on **ep_id**, where *x* is one of the five sorting algorithms defined below.

APPEXP ep_id is similar to **ADDID** except for the following changes. Simply append the new EP ID record to the back of the exposure list. Note this will allow there to be duplicate **ep_id**'s in the exposure list. Each time **APPEXP** is called, the **rec_num** should be incremented and saved in the EP ID. The first time **APPEXP** is called the **rec_num** should be one. The remaining fields in the **epid_rec_t** structure are all set to zero by default. Don't call the **private_make_rec** function, though you can make your own version without all the `fget/sscanf` calls if you like.

The code must consist of the following files:

`lab3.c` – the `main()` function for testing the sort algorithms. Use **`lab2.c`** as a template.
`linkedlist.c` – Extend the two-way linked list ADT from lab2
`ens_support.c` – Extend the **`ens_support.c`** file from machine problem 2.

`ens_support.h` – The IDS interface definitions.
`linkedlist.h` – The data structures and prototype definitions for the list ADT.
`datatypes.h` – Key definitions of the alter structure
`makefile` – Compiler commands for all code files

Sorting the two-way linked list ADT

Create the prototype definition `void linked_sort(linked_t *L, int sort_type, int (*fcomp)(const data_t *, const data_t *))` in your `linkedlist.h` file. For this function, `sort_type` can take one of the following values:

1. Insertion sort. We can use the other functions in `linkedlist.c` to implement a very simple sort function. To sort the list, use `linked_remove` to take the first (or front) item from the list and `linked_insert_sorted` to put the item into a second list that is sorted. When all the elements have been inserted into the sorted list, adjust the pointers in `list_ptr` to point to the newly sorted list. Note this is a simple variation of the priority queue sort described by Standish in Section 4.3 of the book.
2. Extremum sort. Like insertion sort, a simple approach is to use `linked_elem_remove_extremum` to remove the smallest `data_t` memory block from the list and `linked_insert` to put the item into a second list at the tail. The smallest `data_t` element is determined using the `comp_proc` from the list header block. Just like for insertion sort, when all the elements have been inserted into the sorted list, adjust the pointers in `list_ptr` to point to the newly sorted list. This simple sort has basic similarity to the next two versions of selection sort. However, it is (potentially) less efficient because it removes items from one list and adds to a second.

The two version of selection sort defined next simply swap the positions of the `data_t` memory blocks by updating the two `data_ptr`'s.

3. Recursive Selection Sort. Implement the recursive version of the selection sort as defined by program 5.19 on page 152 in the book by Standish. Update the algorithm so that it properly handles our two-way linked list (as opposed to the implementation for an array in program 5.19). The calculations of the sort algorithm should not change; just change the algorithm to work with pointers instead of indexes into an array. You will also need to implement Standish's `FindMax` algorithm defined in program 5.20 on page 152. The source code from Standish is available on Canvas. See also the note at the end of this document. Note it is critical that you don't change the logic of the algorithm. Simply update the algorithm to use pointers instead of integer indices.
4. Iterative Selection Sort. Implement the iterative version of the selection sort as defined by program 5.35 on page 171 in the book by Standish. Read Section 5.4 for an explanation of how the recursive version of the program is transformed into the iterative version.
5. Merge Sort. Implement the recursive version of merge sort as defined by the program 6.19 on page 237 of the book by Standish. Note that you will need to implement two support functions. The first is a function to partition a list into two half-lists (this is easy to implement as you just step through the linked list until half the list size and then break the list into two lists). This second function merges two lists that are sorted into a single list. Read the paragraph on page 237 that discusses how to merge two lists, and note that it is a simple process of using `linked_remove` at the front of either the left or right list and `linked_insert` at the back of the merged list. See also the MergeSort powerpoint file for an illustration of the algorithm in the textbook.

Be careful to design your algorithms so that you do not change their complexity class. Do not redesign the algorithms! The final two lines of the function `linked_sort` **must** be

```
list_ptr->list_sorted_state = SORTED_LLIST;
my_debug_validate(list_ptr);
```

The **`linked_sort()`** function must also update the **`comp_proc`** structure member in the list header to be equal to the function pointer **`fcomp`**. Note that **`APPEXP`** is implemented using **`linked_insert()`**, and the **`linked_insert`** function changes **`list_sorted_state`** to UNSORTED_LLIST. Thus, the sort command is used the change the list status to the sorted state.

Measuring time to sort

To measure the performance of a sorting algorithm use the built in C function **`clock`** to count the number of cycles used by the program. In **`ens_support.c`** add a function for sorting and use code like this:

```
#include <time.h>
clock_t start, end;
double elapse_time;  /* time in milliseconds */

int initialcount = linked_entries(L);
start = clock();
linked_sort(L, sort_type, ens_compare_epid);
end = clock();
elapse_time =  1000.0 * ((double) (end - start)) / CLOCKS_PER_SEC;
assert(linked_entries(L) == initialcount);
printf("%d\t%f\t%d\n", initialcount, elapse_time, sort_type);
```

where `CLOCKS_PER_SEC` and `clock_t` are defined in `<time.h>`.

Additional requirements for SORT

Your final code must use the exact `printf()` statement given in the above example. You will collect output from multiple runs to plot performance curves, showing run time for various list sizes. Your program **must** verify that the size of the list after the completion of the call to `linked_sort` matches the size before the list is sorted. Your program must not have any memory leaks or array boundary violations.

Suppress prints and unnecessary validation calls during performance evaluation and for final submission

Do not include any **`printf`** calls for your new **`ens_append_rear`** function. We don't need to see 250,000 prints about prompting and adding to the queue. Also, comment out the welcome menu and goodbye prints in **`lab3.c`** to avoid unnecessary chatter. The **only** output should be the one new print statement added to measure the sorting time.

The `my_debug_validate()` function is **very** inefficient. After all your code works correctly, remove `my_debug_validate()` from **all** `linked_()` functions except `linked_sort()`, where it must remain as the last line called before the function returns.

Generating large inputs for testing

See the supplemental program **`geninput.c`** to create input for testing. The program takes three options on the command line. The first specifies the size of the list, the second specifies the type of list, and the third the type of sorting algorithm. There are three possible types:

1. List with elements in a random order.
2. List with elements already in ascending order.
3. List with elements in descending order.

Run `./geninput` with no arguments to print a help message. To run, pipe the output of the `geninput` program into `lab3`. For example, for a merge sort trial on a list with 10,000 elements in random order use:

```
./geninput 10000 1 5 | ./lab3
```

The final code you submit **must** operate with **`geninput.c`**. In particular, you must have commented out the calls to the validation function (except the call in **`linked_sort`**). Your program should be able to sort approximately 12,000 items (or more) in about one second for insertion sort or the selection sorts. For merge sort, your program should be able to sort approximately 250,000 items (or more) in about one second

Final testing and PDF for the testing log

Test each of the five sorting algorithms and each of the three list types (random, ascending, descending) with at least **five** different list sizes. You **must** create graphs to illustrate your results. In particular, for the tests involving random list types, you **must** include in your graph the result for at least **one** list size that requires more than one second to sort as determined using the C function `clock`.

In your Test Log document, in addition to reporting your data using graphs, describe

(a) For lists that are initially random, explain the differences in running time for the sorting algorithms. Do your iterative and recursive selection sort algorithms show dramatic differences in running

times or are they similar? Why does the mergesort algorithm show a dramatic improvement in run time? If the runtime for merge sort is not dramatically faster than the other algorithms you have a bug.

(b) If a list is already in ascending or descending order, some sort algorithms are very fast while others still have to perform a similar number of comparisons as when the list is not sorted. Describe which algorithm(s) show extremely fast performance if the list is already sorted, and explain why.

You must submit your test log as a PDF file. Both Microsoft Word and LibreOffice have tools that convert the document to PDF format.

Your test script is a simple listing of the lab3 commands you used to generate the data for your test log. See the example script posted on Canvas. You just need to make small changes to the test script to account for the speed of your computer.

An optional experiment is to recompile your final code taking out the –Wall and –g options and adding the –O option (capital letter o, not zero) to all calls with gcc. The –O option turns on compiler optimizations and you should find your code runs faster. While the run times are reduced and you can sort larger lists, has the complexity class for any of the sorting algorithms changed?

Hint

When converting Standish's selection sort algorithms from working with arrays to working with pointers, don't try to add array-like features to our two-way linked list. Instead rewrite Standish's code to use pointers. For example, change

```
void SelectionSort(InputArray A, int m, int n)
```

to

```
void SelectionSort(linked_t *A, linked_elem_t *m, linked_elem_t
*n)
```

The changes you need to make should be very simple. For example, change from incrementing index values to moving roving pointers.

Submission

Submit all the files listed on page 1, your test script, and a PDF document that includes graphs showing performance of all sorting algorithm. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#3. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes. If you do not get a confirmation email or the email reports an error, your submission was not successful. You must include your files as attachments, and your email must be in text only format. You can make more than one submission but we will only grade the final submission. A re-submission must include all files. You cannot add files to an old submission.

To receive credit for this assignment your code must compile and at a minimum sort small lists using all five sorting algorithms. Code that does not compile or does not sort sets of 100 records will not be accepted or graded.

See the ECE 2230 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student, and see the course syllabus for additional policies.