ECE 2230: Computer Systems Engineering
Machine Problem 3

Spring 2021

Due: 11:59 pm, Wednesday, March 10

1 Introduction

The goal of this machine problem is to extend the two-way linked list ADT from MP2 to include four algorithms for sorting a list. We also extend MP2 to include three new commands. The first is **SORT x order**, where **SORT x 1** sorts the queue in descending order based on the atom ID numbers, **SORT x 2** is the same except in ascending order. Here x is one of the four sorting algorithms defined below. In addition, we will add a new **ADDTAIL atom ID** command to allow us to efficiently build a large list without all the unnecessary details in a atom record. Finally, **PRINTMP3** prints the newly formed queue.

The code must consist of the following files:

lab3.c – contains the `main()` function for testing the sort algorithms.

list.c – Extension of the two-way linked list ADT from machine problem 2.

atom_support.c – Extension of the `atom_support.c` file from machine problem 2.

atom_support.h – The data structures for the specific format of the atom records, and the prototype definitions.

list.h – The data structures and prototype definitions for the list ADT.

datatypes.h – Key definitions of the atom structure and a procedure needed by the list ADT

geninput.c – Generates inputs for lab3.c

makefile – Compiler commands for all code files

runner.sh – script to run multiple experiments efficiently

2 Sorting the two-way linked list ADT

The file `list.h` contains a new prototype definition `void list_sort(list_t **list_ptr, int sort_type, int sort_order);`. Update the new `list.h` and `list.c` with your completed functions from MP2. For this function, `sort_order` is 1 if the items should be in descending order, and 2 for ascending order. The `sort_type` can take one of the following values:

1. Bubble sort. We can use the other functions in `list.c` to implement this sort. Since this repeatedly sort swaps two elements, you can build an auxiliary routine that swaps two elements in the list using `list_remove and list_insert.`

2. Insertion sort. We can use the other functions in `list.c` to implement a very simple sort function. To sort the list, use `list_remove` to take the first (or head) item from the list and `list_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 Standish describes in Section 4.3 of the book.

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. You may also find it useful to create a `FindMin` function. The source code from Standish is available on Canvas. Note that the `FindMax/Min` functions you create should be local to `list.c`.
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 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 `list_remove` at the head of either the left or right list and `list_insert` at the tail 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 time complexity class. Do not redesign the algorithm! The final two lines of the function `list_sort` must be

```
(*list_ptr)->list_sorted_state = LIST_SORTED_?;// ? is ASCENDING or DESCENDING;
list_debug_validate(*list_ptr);
```

Note that ADDTAIL is implemented using `list_insert()`, and the `list_insert` function changes `list_sorted_state` to LIST_UNSORTED. Thus, SORT is used to change the list status to one of the sorted states. For MP3, we will create a new version of **ADDTAIL atom ID** that collects the atom ID with the command and does not call `fill_atom_record`. Instead, it sets only the only the atom ID and initializes all other values in the memory block to zero, such as with `calloc`). A prototype for this function is added to `atom_support.h` and a stub is added to `atom_support.c`.

3 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 `atom_support.c` the function for sorting must be extended to time the sort by using the following:

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

int initial_size = list_size(*list_ptr);
start = clock();
list_sort(list_ptr, sort_type, sort_order);
end = clock(); elapse_time =  1000.0 * ((double) (end - start)) / CLOCKS_PER_SEC;

assert(list_size(*list_ptr) == initial_size);
printf("Sorting: %d %f %d %d\n", initial_size, elapse_time, sort_type, sort_order);
```

where CLOCKS_PER_SEC and clock_t are defined in <time.h>. Most of this code is commented out. Uncomment the code to time the sorts once you have everything working.

4 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 list_sort matches the size before the list is sorted. Your program must not have any memory leaks or array boundary violations.

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

Do not include any printf calls in your new function for adding to the tail of the unsorted list. We don't need to see 250,000 prints about adding to the queue.

The list_debug_validate() function is **very** inefficient. After all your code works correctly, remove list_debug_validate() from **all** list_*() functions. The easiest way to do this is through the use of the pre-processor directive -DVALIDATE in the CFLAGS variable in the makefile. Removing this compiler flag will not define VALIDATE in the compiler's pre-processor, which results in the code inside list_debug_validate being removed and not compiled. Thus, this disables the validation check.

6 Generating large inputs for testing

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

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

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

```
./geninput 10000 2 5 1 | ./lab3
```

The final code you submit **must** operate with geninput.c. Your program should be able to sort approximately 12,000 items (or more) in about one second for insertion or either selection sort. For merge sort, your program should be able to sort approximately 250,000 items (or more) in about one second. Note that this estimation depends on the underlying hardware you are running on and serves as a target.

7 Final testing and PDF for the testing log

Test each of the four sorting algorithms and each of the three list types (random, ascending, descending) with at least **five** different list sizes. Sort in ascending order. 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. Remember that you are not timing the sort in isolation (i.e., other applications are running, competing for

CPU time and cache space). Run each experiment multiple times and average the results for each data point in your graph.

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

1. 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.

2. 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 log should include the commands you used to generate the data for your test log. If you modify geninput for testing purposes, submit the new file and detail what you changed and why.

8 Bonus (10 points)

An optional experiment is to recompile your final code after adding the `-O` option (capital letter o, not zero) to the CFLAGS variable in the makefile. The `-O` option turns on compiler optimizations and you should find your code runs substantially faster values run between `-O0` to `-O3`. Redo the previous analysis and plots for each optimization level (0–3). While the run times are reduced and you can sort larger lists, has the complexity class for any of the sorting algorithms changed? Explain.

9 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(i.e., do not use `list_access()`). Instead rewrite Standish's code to use pointers. For example, change

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

to something like

```
void SelectionSort(list_t **A, list_node_t *m, list_node_t *n)
```

you are not required to use that exact interface, but it should be very similar.

10 Submission

Compress all code, test cases, testing report that includes your plots, test case document defining what each test cases tests into a ZIP file, and submit the ZIP file to Canvas by the deadline. **Your last submission is the one that will be 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.