ECE 223: Computer Systems Engineering **Spring 2014**
Machine Problem 3 **Due: 11:59 pm, Thursday, February 13**

The goal of this machine problem is to extend the two-way linked list ADT from machine problem 2 to include four algorithms for sorting a list. We also extend machine problem 2 to include four new commands: APPENDL *ip-addr*, SORTL *x*, PRINTL, and SCANL *threshold*. These new commands operate on a new third list. APPENDL simply adds the new packet to the end of the list (and does *not* remove any duplicate packets). SORTL *x* sorts the new list in ascending order based on dest_ip_addr, where *x* is one of the four sorting algorithms defined below. PRINTL prints the list in the same format as for PRINT. SCANL *threshold* is identical to SCAN but operates on the new list if it has been sorted. If the new list has not been sorted since the last APPENDL, then the SCANL function should print a message stating that the operation cannot be performed until the list has been sorted. Also, update STATS to report the sizes of all three lists.

The code must consist of the following files:

`lab3.c`	– contains the `main()` function for testing the sort algorithms. Use `lab2.c` as a template.
`list.c`	– Extension of the two-way linked list ADT from lab2
`hpot_support.c`	– Use the same hpot_support.c file from machine problem 2.
`hpot_support.h`	– The data structures for the specific format of the trace route records, and the prototype definitions.
`list.h`	– The data structures and prototype definitions for the list ADT.
`datatypes.h`	– Key definitions of the trace route structure and a procedure needed by the list ADT
`makefile`	– Compiler commands for all code files

Sorting the two-way linked list ADT

Create the prototype definition `void list_sort(pList list_ptr, int sort_type)` in your `list.h` file. For this function, `sort_type` can take one of the following values:

1. 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 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.
2. 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. Except, change the code so the sorted list is increasing. 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 Blackboard.

3. 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.
4. 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.

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

```
     list_ptr->llist_sort = LIST_SORTED;
    list_validate(list_ptr);
```

Note that APPENDL is implemented using **`list_insert()`**, and the **`list_insert`** function changes **`llist_sort`** to LIST_UNSORTED. Thus, SORTL 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 `lab3.c` use:

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

(when the SORTL command is found do)
int initialsize = list_size(L);
start = clock();
list_sort(L, sort_type);
end = clock();
elapse_time =  1000.0 * ((double) (end - start)) / CLOCKS_PER_SEC;
assert(list_size(L) == initialsize);
printf("%d\t%f\t%d\n", initialsize, elapse_time, sort_type);
```

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

Additional requirements for SORTL

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.

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

The `hpot_record_fill()` function generates an unreasonable number of prints when working with large input files. Once debugging is completed and you are preparing the final tests with large input files, comment out the call to `hpot_record_fill()`. While this leaves the record with uninitialized values, when testing list with large sizes the record details are not needed. Skipping reading in record details reduces the memory requires to pipe in the input data, allowing you to test large lists sizes on systems with limited memory (such as virtual machines).

The `list_validate()` function is **very** inefficient. After all your code works correctly, remove `list_validate()` from **all** `list_()` functions except `list_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.

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 4 | ./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 **`list_sort`**) and the call to fill in the details of a packet.

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. You **must** create graphs to illustrate your results. In particular, for the tests involving <u>random</u> 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 Blackboard.

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 substantially 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(list_t *A, llist_node_t *m, llist_node_t *n)
```

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code, a makefile, a test script, and a test log must be turned in by email to assign@assign.ece.clemson.edu. Use as subject header ECE223-1,#3. Work must be completed by each individual student, and see the course syllabus for additional policies.