ECE368 Programming Assignment #2

Due Monday, February 19, 2018, 11:59pm

Description:

This project is to be completed on your own. You will implement Shell sort on an array and Shell sort on a linked list. In both cases, you will use the following sequence for Shell sort:

$$\{h(1) = 1, h(2) = 4, h(3) = 13, \ldots, h(i), h(i+1) = 3 \times h(i) + 1, \ldots, h(p)$$

where $h(p)$ is the largest number in the sequence that is smaller than the size of the array to be sorted.

You are not allowed to store the sequence in your program. The sequence has to be generated (but not stored) as part of your Shell sort functions.

Functions you will have to write for `sorting.c`:

All mentioned functions and their support functions, if any, must reside in the program module `sorting.c`. The declarations of these functions should reside in the include file `sorting.h`. These functions should not call each other (unless you implement any such function as a recursive function). They should be called by the `main` function. `sorting.h` should not contain the declaration of other functions.

There are three functions that deal with performing Shell sort on an array. The first two functions `Load_Into_Array` and `Save_From_Array`, are not for sorting, but are needed to transfer the `long` integers to be sorted from and to a file in **binary form** to and from an array, respectively.

```
long *Load_Into_Array(char *Filename, int *Size)
```

The size of the binary file whose name is stored in the `char` array pointed to by `Filename` should determine the number of `long` integers in the file. The size of the **binary** file should be a multiple of `sizeof(long)`. You should allocate sufficient memory to store all `long` integers in the file into an array and assign to `*Size` the number of integers you have in the array. The function should return the address of the memory allocated for the `long` integers.

You may assume that all input files that we will use to evaluate your code will be of the correct format.

Note that we will not give you an input file that stores more than `INT_MAX` `long` integers (see `limits.h` for `INT_MAX`). If the input file is empty, an array of size 0 should still be created and `*Size` be assigned 0.

```
int Save_From_Array(char *Filename, long *Array, int Size)
```

The function saves `Array` to an external file specified by `Filename` in **binary format**. The output file and the input file have the same format. The integer returned should be the number of `long` integers in the `Array` that have been successfully saved into the file.

If the size of the array is 0, an empty output file should be created.

```
void Shell_Sort_Array(long *Array, int Size, double *N_Cmp)
```

The function takes in an `Array` of `long` integers and sort them. `Size` specifies the number of integers to be sorted, and `*N_Cmp` should store the number of comparisons involving items in `Array` throughout the entire process of sorting. You may choose to use insertion sort or bubble sort to sort each subarray.

A comparison that involves an item in `Array`, e.g., `temp < Array[i]` or `Array[i] < temp`, corresponds to one comparison. A comparison that involves two items in `Array`, e.g., `Array[i] < Array[i-1]`, also corresponds to one comparison. Comparisons such as `i < j` where `i` or `j` are indices are not considered as comparisons for this programming assignment.

There are also a set of three functions that deal with performing Shell sort on a linked list. In this assignment, you will use the following user-defined type to store integers in a linked list:

```
typedef struct _Node {
    long value;
    struct _Node *next;
} Node;
```

You have to define this structure as provided above in `sorting.h`

You may also use the following user-defined type to store a linked-list of linked-lists. To be exact, the following structure can be used to implement a linked-list of addresses pointing to the `Node` structure.

```
typedef struct _List {
    Node *node;
    struct _List *next;
} List;
```

This structure is probably useful for you to maintain k linked-lists, where k is a number in your sequence. However, it is not necessary that you use this structure in your implementation. My implementation uses only the structure `Node`. You can really have a shorter run-time if you use a list of linked-list. Using only the structure `Node` can really slow down the sorting.

Given the definition of the structure `Node`, these are the three functions you have to write to deal with performing Shell sort on a linked list:

```
Node *Load_Into_List(char *Filename)
```

```
int Save_From_List(char *Filename, Node *list)
```

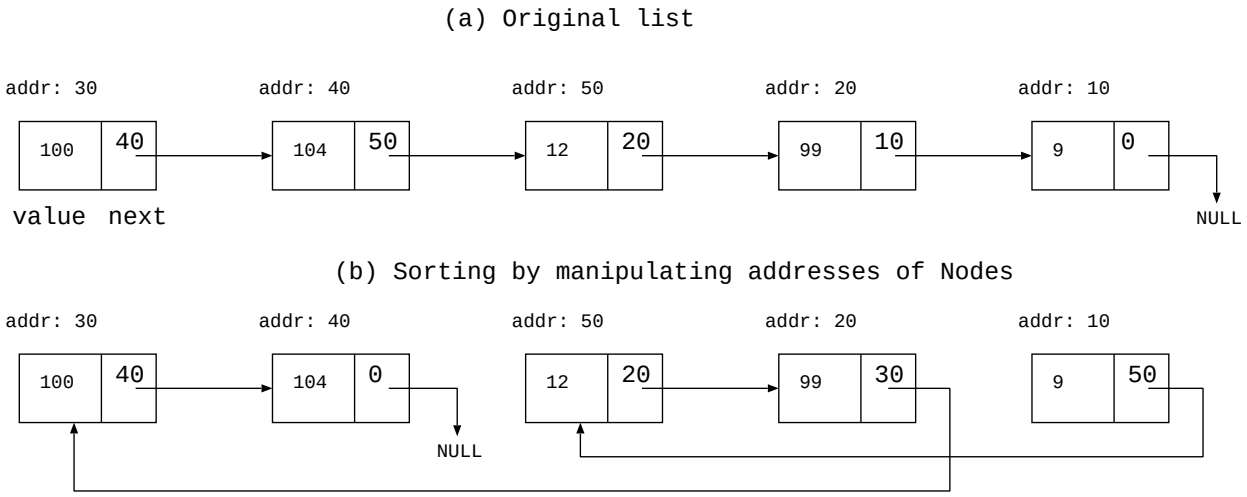
```
void Shell_Sort_List(Node *list, double *N_Cmp)
```

The input and output files are of the same format as in the case of array. `Filename` in each of the two functions is the address of the `char` array containing the name of the input or output file.

The load function should read all (`long`) integers in the input file into a linked-list and return the address pointing to the first node in the linked-list. The linked-list must contain as many `Nodes` as the number of `long` integers in the file.

The save function should write all (`long`) integers in a linked-list into the output file. This function returns the number of integers successfully written into the file.

The Shell sort function takes in a `list` of long integers and sort them. To correctly apply Shell sort, you would have to know the number of elements in the `list` and generate the sequence accordingly. The address pointing to the first node of the sorted list is returned by the function. Similar to the case of an array, a comparison here is defined to be any comparison that involves the field `value` in the structure `Node`.



The `Shell_Sort` function must perform sorting by manipulating the `next` fields of the `Nodes`. Figure (a) shows an original list that is unsorted. Figure (b) shows how the list is sorted by storing the correct addresses in the `next` fields. The `long` integers stored in the `value` fields remain in the original `Nodes`. For example, the integer 99 is stored in a `Node` with an address 20 in the original list. The `field` of the same `Node` stores the address 10, allowing it to point to the `Node` storing the value 9.

After sorting, 99 is still stored in the `value` field of the `Node` with address 20. However, the `next` field of the `Node` now stores 30, allowing it to point to the `Node` storing the value 100.

In other words, each `long` integer must reside in the same `Node` in the original list before and after sorting.

`main` function you will have to write:

You have to write another file called `sorting_main.c` that would contain the `main` function to invoke the functions in `sorting.c`. You should be able to compile `sorting_main.c` and `sorting.c` with the following command

```
gcc -g -std=c99 -Wall -Wshadow -Wvla -pedantic sorting.c sorting_main.c -o proj2
```

When the following command is issued,

```
./proj2 -a input.b output.b
```

the program should load from `input.b` the integers to be sorted and store them in an array, run Shell sort on the array, and save the sorted integers in `output.b`. The program should also print to the standard output (i.e., a screen dump), the following information:

```
I/O time:  AAAA
Sorting time:  BBBB
Number of comparisons:  CCCC
```

where `AAAA`, `BBBB`, and `CCCC` are all in `%le` format. Essentially, you report the statistics you have collected in your program.

When the following command is issued,

```
./proj2 -l input.b output.b
```

the program should load from `input.b` the integers to be sorted and store them in a linked-list, run Shell sort on the linked-list, and save the sorted integers in `output.b`. The program should also print to the standard output (i.e., a screen dump), the following information:

```
I/O time:  AAAA
Sorting time:  BBBB
Number of comparisons:  CCCC
```

You may declare and define other help functions in `sorting.c` and `sorting_main.c`. However, these help functions should not be declared in `sorting.h`. It is best that these helper functions be declared as `static`. Do not name these help functions with a prefix of two underscores "__".

The function that you may use to keep track of `I/O time` and `Sorting time` is `clock()`, which returns the number of clock ticks (of type `clock_t`). You can call `clock()` at two different locations of the program. The difference gives you the number of clock ticks that pass by between the two calls. You would have to divide the difference by `CLOCKS_PER_SEC` to get the elapsed time in seconds. There are typically 1 million clock ticks in one second.

It is important that if the instructor has a working version of `sorting_main.c`, it should be compilable with your `sorting.c` to produce an executable. Similarly, if the instructor has a working version of `sorting.c`, it should be compilable with your `sorting_main.c` to produce an executable.

Report you will have to write:

You should write a (brief, at most a page) report that contains the following items:

- An analysis of the time- and space-complexity of your algorithm to generate the sequence (not sorting).

- A tabulation of the run-time, number of comparisons, and number of moves obtained from running your code on some sample input files. You should comment on how the run-time and the and number of comparisons grow as the problem size increases, i.e., the time complexity of your sorting routines.

- A comparison of the time complexity for sorting an array and the time complexity for sorting a linked list. You should mention the basic sorting routine(s) (insertion or bubble) that you are using for the two Shell sort routines.

- A summary of the space complexity of your sorting routines, i.e., the complexity of the additional memory required by your routines. Do not include the space required by the linked-list to store the integers (and the addresses). However, if you have to create a linked-list of lists (or a linked list of addresses), you would have to account for the additional space for that linked list of lists (or pointers).

Each report should not be longer than 1 page and should be in PDF or plain text format (using the textbox in the submission window). The report will account for 10% of the overall grade of this project.

Submission and Grading:

The project requires the submission (electronically) of a zip file called `proj2.zip` through Blackboard. The zip file should contain the programs `sorting.c` and `sorting_main.c` and the header file `sorting.h` you have created. The zip file should also contain the report. We do not expect you to turn in a Makefile.

The grade depends on the correctness of your program, the efficiency of your program, the clarity of your program documentation and report.

The two sorting functions will account for at least 50% and at most 70% of the entire grade. The other functions and the report will account for the remainder of the grade.

It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Memory issues will result in 40% penalty.

Given:

We provide sample input files (in `proj2_samples.zip`) for you to evaluate the runtimes, and numbers of comparisons and moves of your sorting algorithms. All ".b" files are binary files. The number in the name refers to the number of `long` integers the file is associated with. For example, `15.b` contains 15 `long` integers, `15s.b` contains 15 sorted `long` integers from `15.b`. In particular, `15s.b` is created by `proj2` by the following command:

```
./proj2 -a 15.b 15s.b
```

My implementation of `proj2` prints the following output to the screen when the above command is issued:

```
I/O time: 1.980000e-04
Sorting time: 2.000000e-06
Number of comparisons: 6.600000e+01
```

My implementation of `proj2` prints the following output to the screen when the following command is issued:

```
./proj2 -l 15.b 15sl.b
I/O time: 3.490000e-04
Sorting time: 5.000000e-06
Number of comparisons: 1.360000e+02
```

My implementation of `proj2` also created `1000s.b` and `1000sl.b`. Of course, `15s.b` and `15sl.b` are identical and `1000s.b` and `1000sl.b` are also identical. For the input files `10000.b`, `100000.b`, and `1000000.b`, the output files of my implementation of `proj2` are not included.

Your implementation should not try to match the number of comparisons that my implementation reported. That is not the purpose of the assignment.

Getting started:

Copy over the files from the Blackboard website. Any updates to these instructions will be announced through Blackboard.

Given that the input files are in binary format, you probably want to write some helper functions to print the array of `long` integers before and after sorting in text (instead of binary) for debugging purpose.

You also have to ask the question of whether you have performed (Shell) sorting correctly. If the array of `long` integers is in ascending order after sorting, have you sorted correctly?

Start sorting!