Assignment 2: mysort

Objectives

You must implement a list API and a number sorting program with several command-line options.

Requirements

Your sorting program must be named `mysort` and its basic operation is as follows:

- it reads zero or more numbers from standard input;
- it prints the same numbers in sorted order on its standard output, one number per line.

Numbers in the input are non-negative and separated by any whitespace, including newlines. You can test your program by typing your input in the terminal [1], or you can use bash input redirection to send your program a whole text file [2].

While reading the input, your program must sort by maintaining a linked list in memory. It should implement a version of the insertion sort [3] algorithm to do this, but *some careful consideration of the problem might help you avoid implementing that complete algorithm on your linked list*.

Your program should take several command-line options that modify the sorted list. You can find the complete list of options in the **Grading** section. We provide the code to parse the command-line arguments with the `getopt` function in `main.c`. For the complete description of `getopt` you can consult the manual pages using the command `man 3 getopt` in the terminal. Other useful man pages for this assignment include `fgets`, `strtol` and `strtok`.

You must submit your work as a tarball. The command `make tarball` will create a tarball called `insertion_sort_submit.tar.gz` containing the relevant files.

Getting started

1. Read all the function prototypes and descriptions in `list.h`. Decide on what type of *linked list* you will create and draw representations of your intended *list* and *node* structs on paper.
2. Implement the function prototypes from `list.h` in `list.c` in their given order and run `make check` after completing every function. If you think you have implemented enough functions to pass a test and the test still fails, fix the problem before moving on to the next set of functions.
3. Start with the processing of a simplified version of the input description in `main.c`. Only parse numbers separated by newlines, and do not consider multiple numbers on the same line just yet. Sort the input numbers using your linked list implementation and print them in the specified output format.
4. Add the code to parse the complete input description and combine this with your existing sorting code. Run `make check` and ensure that your code passes all sorting tests.

5. Reread all the function descriptions in `list.h` and verify that you implemented all these functions correctly. Carefully consider all edge case inputs for your functions and ensure you handle them in your implementation (i.e. they should not crash or break the program).
6. Implement the option flags described in the grading section. A flag on the command line sets the related variable in the `config` struct to `1`. Check this struct and modify the program output accordingly.

Grading

Your grade starts from 0, and the following tests determine your grade:

- +2pt if you have submitted an archive in the right format, and you have made a real effort to implement to implement `list.c` and `main.c`.
- +2pt if your list correctly implements all described functions in `list.h` and also handles invalid operations such as invalid inserts and invalid unlinks correctly.
- +2.5pt if your `mysort` correctly processes any input meeting the described requirements and produces output in sorted order according to the specified format.
- -1pt if your code produces any warnings using the flags `-Wpedantic -Wall -Wextra` when compiling.

The provided tests only partially check the following features. You will have to validate the correctness of these features yourself by writing your own tests.

- +0.5pt if your `mysort` supports the option `-d`, which causes the values in the list to be sorted in descending order.
- +0.5pt if your `mysort` supports the option `-c` which causes the sorted list values to be added in a pair-wise manner as shown in this example:

```
$ echo "5 4 3 2 1" | ./mysort -c
3
7
5
```

- +0.5pt if your `mysort` supports the option `-o`, which removes all odd valued numbers from the sorted list.
- +0.5pt if your `mysort` supports the option `-z` which causes the sorted list to be cut into two equal halves (with the first half being longer in case of odd length) and then zips the two halves back together, starting with the first element of the first half and then alternating between elements from the second and first half until all elements have been joined back into the single list. This example shows the output for a simple example:

```
$ echo "5 4 3 2 1" | ./mysort -z
1
4
2
5
3
```

- +0.5pt if your `mysort` handles interactions between the flags correctly.
- +1pt If your implementation has the correct style and the correct complexity.

- -0.5pt if your program misbehaves on zero-sized inputs.
- -0.5pt if your program misbehaves when the last line does not terminate with a newline character.
- -1pt if your program does not correctly handle out-of-memory situations.

Your program should also be able to handle combinations of these options in the same order they are described here, e.g. *-c* always happens before *-z*. The list is first sorted (in ascending or descending order), and then all options are applied in the above order. With any combination of options, the output should still be produced only once after all the options have been applied, according to the specified output format.

---

1 https://thoughtbot.com/blog/input-output-redirection-in-the-shell#standard-input
2 https://thoughtbot.com/blog/input-output-redirection-in-the-shell#redirecting-output
3 https://en.wikipedia.org/wiki/Insertion_sort
4 https://github.com/google/sanitizers/wiki/AddressSanitizer
5 http://valgrind.org/docs/manual/quick-start.html#quick-start.intro