Project 02: Sorting List-Based Strings and Numbers

Overview

Your second project will be to build a sorting utility called `volsort` in C++ that will mimic the UNIX tool "sort" with a twist: the underlying container must be a linked list (https://en.wikipedia.org/wiki/Linked_list).

Your program must implement the following sorting methods or modes:

1. **Oblivious**: This sorting method does no sorting; it just reads and returns the numbers similar to the benchmark in Dr. Plank's notes.
2. **STL**: This sorting method simply uses the std::sort (http://en.cppreference.com/w/cpp/algorithm/sort) function.
3. **qsort**: This sorting method simply uses the qsort (http://man7.org/linux/man-pages/man3/qsort.3.html) function.
4. **merge**: This sorting method uses a custom implementation of the merge sort (https://en.wikipedia.org/wiki/Merge_sort) algorithm.
5. **quick**: This sorting method uses a custom implementation of the quick sort (https://en.wikipedia.org/wiki/Quicksort) algorithm.

This project is to be done in groups of **1 - 4** students and is due **Wednesday February 6th, 2019** at 12:01pm.

Specification

My implementation, which you should mimic, has the following **usage** message:

```
$ volsort -h
usage: volsort
    -m MODE   Sorting mode (oblivious, stl, qsort, merge, quick)
    -n        Perform numerical ordering
```

`volsort` therefore must take two optional arguments: the `-m` flag, which lets the user select which sorting method to use; and the `-n` flag, which informs the program to treat the input values as numeric integers rather than textual strings.

As with most Unix filters (https://en.wikipedia.org/wiki/Filter_(software)#Unix), `volsort` reads from [standand input] and then outputs to standard output (https://en.wikipedia.org/wiki/Standard_streams#Standard_output_.28stdout.29):

```
# Use the volsort solution to order 10 random numbers in the file input
$ volsort -m quick < input
12
32
37
37
50
61
69
71
90
97
```

Node and List

Your header file should contains the definition of the `struct Node` and `struct List` such as this:

```
struct Node {
  std::string string;
  int         number;
  Node       *next;
};

struct List {
  Node       *head;
  size_t      size;

  List();
  ~List();

  void push_front(const std::string &s);
};
```

This `struct List` is a singly-linked list (https://en.wikipedia.org/wiki/Linked_list#Singly_linked_lists) with a **constructor**, **destructor**, and a single **push_front** method. This `struct List` consists of `struct Node` entries that contain both **string** values, the corresponding **number** value, and a pointer to the next `struct Node` .

It is required that you implement the `struct List` **constructor**, **destructor**, and **push_front** method. The **push_front** method should insert a new `struct Node` into the front of the list. To get the **number** value for the new `struct Node` , you should use the std::stoi (http://en.cppreference.com/w/cpp/string/basic_string/stol) function (if this conversion fails, you can default to `0` for the **number** value).

You should implement the `struct Node` comparison functions in C ++ style as:

```
// C++ Style comparison function
bool node_number_compare(const Node *a, const Node *b);
bool node_string_compare(const Node *a, const Node *b);
```

Thus, there are two comparision functions in total: one for comparing the **number** field and the other for comparing the **string** field of the `struct Node`

Additionally, it will help to debug if you also implement the `dump_node` utility function which outputs the contents of each node start from `n` until `nullptr` is reached:

```
// Utility function
void dump_node(Node *n);
```

STL Sort

The first sorting mode is to use the std::sort (http://en.cppreference.com/w/cpp/algorithm/sort) method. Because our `struct List` does not implement random access iterators, we will need to copy the `struct Node` s in the `struct List` into another container such as an array. Once we have this copy, we can then call the std::sort (http://en.cppreference.com/w/cpp/algorithm/sort) function on this copy with the appropriate comparison function. Note that our Node struct has three values, one of which is a string. The most efficient way to reconstitute the sorted order is to update the links between the `struct Node` s -- not to simply replacing the list data with node data. For example, for a list of *n* strings you may have to run atoi *n* times, versus simply updating *n* pointers as suggested here.

Hint: You should store `struct Node *` in your copy array.

Hint: Make sure you set the **head** of the `struct List` after you have set the links of all the `struct Node` s.

QSort

The second sorting mode could be nearly identical to the first function, except you will use qsort (http://man7.org/linux/man-pages/man3/qsort.3.html) instead of std::sort (http://en.cppreference.com/w/cpp/algorithm/sort).

Merge Sort

Both of the custom merge sort (https://en.wikipedia.org/wiki/Merge_sort) and quick sort (https://en.wikipedia.org/wiki/Quicksort) implementations must use the following divide-and-conquer (https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms) strategy. Msort is used here as the initial example:

```
Node *msort(Node *head, CompareFunction compare) {
    // Handle base case

    // Divide into left and right sublists

    // Conquer left and right sublists

    // Combine left and right sublists
}
```

Specifically, you could implement the third sorting mode as follows:

```
// Public functions
void merge_sort(List &l, bool numeric);

// Internal functions
Node *msort(Node *head, CompareFunction compare);
void split(Node *head, Node *&left, Node *&right);
Node *merge(Node *left, Node *right, CompareFunction compare);
```

- `merge_sort` takes a `struct List` and whether or not the comparison should be `numeric` and performs the top-down merge sort (https://en.wikipedia.org/wiki/Merge_sort) algorithm. This function serves as a wrapper or helper function for the recursive `msort` function.
- `msort` is the recursive portion of the algorithm and calls `split` to **divide** and calls `merge` to **conquer**. It returns the new `head` of the list.
- `split` is a helper function that splits the singly-linked list (https://en.wikipedia.org/wiki/Linked_list#Singly_linked_lists) in half by using the *slow* and *faster* pointer trick (https://www.quora.com/What-is-a-slow-pointer-and-a-fast-pointer-in-a-linked-list) (aka [tortoise and hare]).
- `merge` is a helper function that combines both the `left` and `right` lists and returns the new `head` of the list.

Online Code -- don't look!

There are many examples of merge sort (https://en.wikipedia.org/wiki/Merge_sort) on linked lists. For instance, GeeksForGeeks (http://www.geeksforgeeks.org/merge-sort-for-linked-list/) has an article about Merge Sort for Linked Lists (http://www.geeksforgeeks.org/merge-sort-for-linked-list/). You should resist the temptation to look at such sources.

Remember that any code you submit must be your own and that you should understand everything you submit. The best way to do that is to not look at online solutions.

If you still don't believe me, the case above from GeeksForGeeks (http://www.geeksforgeeks.org/merge-sort-for-linked-list/) is suboptimal for two reasons: 1) We are not covering or requiring recursion, so its a red flag; and 2) the reason we are stressing an iterative version of `merge` for this project is to ensure you don't overflow the stack on large input sizes we will provide during grading. We recognize libraries can be useful but we are requiring you to implement this on your own.

Quick Sort

The fourth sorting mode is a custom implementation of quick sort (https://en.wikipedia.org/wiki/Quicksort). You are to implement a simple version of the algorithm where the first element is the pivot.

```
// Public functions
void quick_sort(List &l, bool numeric);

// Internal functions
Node *qsort(Node *head, CompareFunction compare)
void partition(Node *head, Node *pivot, Node *&left, Node *&right, CompareFunction compare);
Node *concatenate(Node *left, Node *right);
```

- `quick_sort` takes a `struct List` and whether or not the comparison should be `numeric` and performs the quick sort (https://en.wikipedia.org/wiki/Quicksort) algorithm. This function serves as a wrapper or helper function for the recursive `qsort` function.
- `qsort` is the recursive portion of the algorithm and calls `partition` to **divide** the list and calls `concatenate` to **conquer**. It returns the new `head` of the list.
- `partition` is a helper function that splits the singly-linked list (https://en.wikipedia.org/wiki/Linked_list#Singly_linked_lists) into two `left` and `right` lists such that all elements less than the `pivot` are in the `left` side and everything else is on the `right` .
- `concatenate` is a helper function that combines both the `left` and `right` lists and returns the new `head` of the list.

Online Code -- don't look!

Again, there are many examples of quick sort (https://en.wikipedia.org/wiki/Quicksort) on linked lists. For instance, GeeksForGeeks (http://www.geeksforgeeks.org/merge-sort-for-linked-list/) has an article about QuickSort on Singly Linked Lists (http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/). Again, you should resist the temptation to look at such sources.

Just like above, this example from GeeksForGeeks (http://www.geeksforgeeks.org/merge-sort-for-linked-list/) is much more complex than the proposed divide-and-conquer (https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms) strategy shown above.

Finishing up

To test your programs, simply run `make test` . Your programs must correctly process the `input` correctly, have no memory errors, and pass the test cases provided.

When you are finished implementing your `volsort` sorting utility answer the following questions in the `project02/README.md` :

1. Benchmark all four different sorting modes on files with the following number of integers:

   ```
   10, 100, 1000, 10000, 100000, 1000000, 10000000
   ```

   Record the elapsed time and memory consumption of each mode and file size in a Markdown (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet) table:

   ```
   | Mode    | Size  | Elapsed Time  |
   |---------|-------|---------------|
   | STL     | 10    | 0             |
   | STL     | 100   | 0             |
   | ...     | ...   | ...           |
   | QSORT   | 10    | 0             |
   | ...     | ...   | ...           |
   | MERGE   | 10    | 0             |
   | ...     | ...   | ...           |
   | QUICK   | 10    | 0             |
   | ...     | ...   | ...           |
   ```

   Hint: Use the Unix utility **time** to get the elapsed time per test.

   Hint: Write a simple C/C++ program or create a Python (https://www.python.org/) script to generate the input files.

   Optional: If you have prior experience, write another simple script (shell script for example) to automate the benchmarking.

   ### Scratch Space and Output

   You may wish to create your test files in `/tmp` as that is likely larger than your EECS **local** directory.

   Likewise, you should redirect the output of `volsort` to `/dev/null` for the benchmarking.

2. After you have performed your benchmark:

   - Discuss the relative performance of each sorting method and try to explain the differences.
   - What do these results reveal about the relationship between theoretical **complexity** discussed in class and actual performance?
   - In your opinion, which sorting mode is the best? Justify your conclusion by examining the trade-offs for the chosen mode.

In addition to the questions, please provide a **brief summary** of each **individual** group members **contributions** to the project.

Testing your code prior to submission

To faciliate testing, you were previously asked to clone the course Github repository as follows:

```
git clone https://github.com/semrich/ds302_20.git cs302
```

For this assignment, update this clone by using the following:

```
git pull
```

We'll discuss this in class but note that your program must be compilable using make. To test your solution against ours, type:

```
make test
```

Rubric

As we grade the assignment, "as intended" means you achieve the correct output in a given mode as outlined in this project document above. We may choose to assign up to 50% of the total points for as many alternative implementations, e.g., volsort -m stl copies list into a vector, sorts using std:sort(v.begin(), v.end()), and copies back into the list if:

1. Corresponding markdown entries are included (see part 3)
2. You articulate in your writeups what challenges you had implementing the code as outlined in the project description above.

Full rubric in three total parts follows:

Part 1: Project high-level requirements

- +2 code well formatted, commented, with reasonable variable names
- +2 You partner(s) names (if done in a group and Git repo link (1 point each)
- +4 Your writeup about the group dynamics, contributions, or summary of solution (solo projects)

Part 2: Make test

- + 4 Volsort -m stl passes (no diff, code as intended)
- + 4 Volsort -m stl -n passes (no diff, code as intended)
- +4 Volsort -m qsort passes (no diff, code as intended)

- +4 Volsort -m qsort -n passes (no diff, code as intended)
- +5 Volsort -m merge passes (no diff, code as intended)
- +5 Volsort -m merge -n passes (no diff, code as intended)
- +5 Volsort -m quick passes (no diff, code as intended)
- +5 Volsort -m quick -n passes (no diff, code as intended)
- +4 No reported memory issues

Part 3: Report in Git readme (as requested) or separate doc

- +6 markdown table as requested (1.5 point per sort)
- +6 answers to requested questions (2 points each)

Submission

To submit your solution, you must submit a single archive (.zip, .tar, .gz, etc.) on Canvas prior to the deadline.

Note: Although submission will be faciliated by Canvas, we will compile and test on EECS lab machines!

If you develop your solution elsewhere please make sure it works on the lab computers prior to the deadline