Assignment Basic Sorting: Simples Sorts and their Performance

CSCI 520: Data Structures and Algorithms

Spring 2022

Objectives

- More practice with classes and arrays in C++.
- Practice writing member functions of C++ classes.
- Explore basic $\mathcal{O}(n^2)$ sorting algorithms. by implementing

Description

In this assignment we will look at some of the variations of basic sorts given in our textbook, and try and compare the performance of them.

In particular we will implement the basic insertion sort given in our textbook. We will then also implement the Shellsort variation. You will create helper methods to implement these sorts, within the usual assignment framework. Then we will create some code to generate lists of random items, and compare the performance of the insertion sort and the shell sort variation on this data.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-List.cpp`	Unit tests for the List class.
`src/test-libsort.cpp`	Unit tests for the sorting library assignment functions.
`include/List.hpp`	Header file for the declarations of the `List` class you will be modifying and its defined API.
`include/libsort.hpp`	Header file for function prototypes for our libsort library functions.
`src/List.cpp`	Implementation file for the `List` member functions that implement the API and class functionality.
`src/libsort.cpp`	Implementation file for our sorting library assignment functions.

This week you will be adding in several new member functions to the `List` class. So all of your work will be to add in code into the `List.[hpp|cpp]` header and implementation files.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment.
3. Configure the project by running the `configure` script from a terminal.
4. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
5. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Also make sure you link the issues with the `Feedback` pull request.

Assignment Tasks

Task 1: Generate a list of random strings

We are going to (re)implement the insertion sort from our textbook, but using a defined `List` data type. The `List`, described above, keeps a list of strings. It keeps track of the items in the list, and the lists current size.

We will first create a method that will generate a list of random string items and return this list of items. As usual uncomment the task 1 unit tests in the `test-libsort.cpp` file and create function prototypes and a stub function to get started.

You will find that there is already a function implemented in the `libsort` library called `generateRandomString()`. You will use this function to implement your function to generate a list of random strings.

Your function will take 3 `const int` parameters as inputs and it will return a `List` as the result of calling the function. The signature of the function should look like this:

```
List generateRandomListOfStrings(const int numItems, const int itemSize = 5, const int seed = 42);
```

The first parameter is the number of random string items to be generated and put into the List that will be created and returned. The last 2 parameters have default parameter values. The `itemSize` is the size of the random string that should be generated and put into the `List`, that defaults to strings of size 5 by default. And to make testing easier, this method allows you to specify a random seed value that should be used to seed the random number generator, which defaults to a value of 42.

The pseudocode of your function implementation should look like this:

1. Create a new `List` of the indicated size so it can hold the indicated `numItems`. Use the `List` constructor to do this, specifiying the list size on creation.
2. Seed the random number generator with the indicated seed parameters. The `srand()` function from the c standard library is used to set the random number generator seed value.
3. Use a loop to create the indicated numItems random strings. You will call the `generateRandomString()` function you are given in this loop to do the actual work of generating random strings of the indicated `itemSize`. The `List` data type has the `operator[]` overloaded, so you can use it to assign strings to your `List` in the loop.
4. Return your `List` that you just initialized with the randomly generated strings.

Once you are satisfied with your work and it is passing the unit tests for the first task, commit your work and push it to your GitHub assignment repository.

Task 2: Implement Insertion Sort

In task 2 you will add in a function to implement the same insertion sort we saw in class, modified to sort our `List` of string values. This function should take a reference to a `List` as its only input parameter, and it will be a `void` function as from our textbook.

You will need to modify the code from our textbook to work on sorting the given `List`. For example, instead of being given the left and right indexes, you need to get those indexes from the size of the list (e.g. the left index will be 0 if we want to sort the whole list, and the right can be found using the `getSize()` member method).

Implement the insertion sort. Make sure you use the second version given in our textbook, that has an initial bubble pass to get the smallest value to the left most index, and then uses shifting to move items so can insert values to their correct location.

Task 3: Implement function to calculate h

We will write one helper function to be used by the shell sort implementation.

Write a function named `calculateInitialH`. This function calculates the h-step value to be used by shell sort. It should take a `const int` of size as the input. This will be the size of the list - 1, e.g. if you look at our textbook it uses (right - left) in the algorithm, which would for example be 9 for a list of size 10. This function should return the initial h-ste size as an int result.

You can look at the tests to see what the correct value of h should be that is returned. You will need to implement the for loop of the shell sort from our textbook for this function.

Task 4: Implement Shell Sort

Implement the shell sort function from our textbook. This function should have the same signature as the insertion sort from task 2. It will take a reference to a List as its only input parameter. It will sort the items using the shell sort algorithm from our textbook. You should reuse the `calculateInitialH` function to calculate the initial h-step value. Then implement the shell sort as a sequence of insertion sorts of h-step values, as shown in the text.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 25 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 5 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop

index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).

4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U05-1 Searching
- Lecture U05-2 Sorting
- Lecture U06-1 Analysis of Algorithms
- Binary Search Playlist MyCodeSchool
- Sorting Algorithms Playlist MyCodeSchool
- Time Complexity Analysis Playlist MyCodeSchool
- Recursion Playlist MyCodeSchool
- Recursive Sorting Algorithms