Assignment Sorting and Searching: Recursive Sorting and Searching with Merge Sort and Binary Search

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- More practice with classes and arrays in C++.
- Practice writing member functions of C++ classes.
- More examples of and practice with writing recursive functions.
  - Specifically divide-and-conquer recursive algorithms.
- Explore $\mathcal{O}(n \log n)$ sorting algorithms by implementing the recursive merge sort.
- Explore $\mathcal{O}(\log_n)$ searching using a recursive Binary Search implementation.

Description

In this assignment the primary goal is to learn more about how sorting and searching algorithms work. We will be implementing merge sort, which is an interesting sorting algorithm that has $\mathcal{O}(n \log n)$ performance and is implemented most naturally using a recursive algorithm. As you should have learned in this unit, naive sorting algorithms, like Bubble Sort, have average and worst case performance of $\mathcal{O}(n^2)$, which is not very good. The worst and average case performance of merge sort is $\mathcal{O}(n \log n)$ which is better, though merge sort will required $2n$ space, which means it needs to make a copy of the values to perform the sorting, thus doubling the amount of in core memory needed to perform the sort.

In addition to implementing a $\mathcal{O}(n \log n)$ sort, we will take advantage of the fact that we can sort a list of values to also implement a $\mathcal{O}(\log_2 n)$ binary search. We looked at an iterativeversion of the binary search in our textbook readings and lecture videos for this unit. Binary search can easily be implemented as a recursive algorithm, and is also an example of divide-and-conquer recursion. As was covered in this unit, binary search only works on a list of items that is already sorted. But if the list is presorted, we can take advantage of this fact to search the least very rapidly, in $\mathcal{O}(\log_2 n)$ time.

This week you will be returning back to working with and implementing member functions of a class. We will be (re)using the `List` class user defined type that may have been given to you in a previous assignment, though the `List` class has been modified this week to hold a list of `string` values for variety. You will be adding methods to the `List` so that you can sort the list of values by performing a merge sort, and then search the resulting sorted list of values using a binary search.

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 sorting and searching member functions you are to implement.
`include/List.hpp`	Header file for the declarations of the `List` class you will be modifying and its defined API.
`src/List.cpp`	Implementation file for the `List` member functions that implement the API and class functionality.

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

You will start by implementing a `sort()` member method that will be added to the `List` class to enable it to sort its list of values. merge sort can be specified as a recursive algorithm. The base case is similar to what we did for several function in a previous assignment, whenever a list is of size 0 or 1 then trivially the list is already sorted.

The general case for merge sort can be specified as

1. Divide the list into half. Create two new lists where the first one has a copy of the values in the lower half of the list we are sorting, and the second one has a copy of the values in the upper half.
2. Call the sort recursively on the lower and upper half lists.
3. On return these sublists should now be sorted, perform a sorted merge of these two sublists back into the original list.

So to support a merge sort, for part 1 you will implement a new type of copy constructor that can construct a copy of a sub portion of a list. To support part 3 of the algorithm, you will add in a `merge()` member function that will expect two sorted lists, and will merge those into the existing list of values. Then of course once we have those two methods, we can easily write the `sort()` method itself to use them to recursively sort the original `List`.

Task 1: Implement Sub List Copy Constructor

If you haven't already, create the Task 1 issue on GitHub and link this issue with the `Feedback` pull request.

We first need a way to easily make a copy of a portion of an existing list, so that we can first divide a list (the divide part of divide-and-conquer), before we sort recursively and merge back together. The `List` data type already has a copy constructor defined that we used in the previous assignment with this `List` type. You are going to add yet another constructor to our `List` data type, but one which will make a new list that is a copy of a portion of another list. So for this task, you need to add a new `List` constructor that is the same as the copy constructor, but add in two additional parameters, a `begin` index and an `end` index (both integers). These will indicate the portion of the passed in list that should be copied into the newly created list. Your implementation of the copy constructor will then be similar to the existing one, but you should only allocate enough memory to hold the values that will be copied, and then only copy over the indicated values.

It is suggested you perform the usual activities before trying to implement this task in earnest. The first test case in the testing file was already uncommented and was testing the existing `List` functionality. Uncomment the second test case, which will test the copy constructor you are to be creating. Then add in the function prototype to the `List.hpp` file for the new copy constructor, and add in a stub function that doesn't do anything. The code should compile and run, but the new tests you uncommented will not be passing.

Once you have the project compiling and running the tests again, implement the sublist copy constructor as described. When you are able to pass the tests and are satisfied with your work, commit your changes and push them to your repository `Feedback` pull request.

Task 2: Implement `merge()` Member Function

Make sure you have created Issue 2 for this task and linked it with the `Feedback` pull request.

For the second task, you need to add a `merge()` member function to your `List` class. This is a special case of merging lists, and will be a bit tricky, you may want to watch/read some of the materials below on merge sort if you need to understand better what is being done with this `merge()` function.

This function will take in references to two other `List` instances. This member function will not modify either of these input lists, so they should be passed in as constant `const` reference parameters. Call these parameters `lower` and `upper`, though really neither list is positionally before or after the other. It is required, in order for this method to work correctly, that both `lower` and `upper` are sorted before calling this method. We will not test this assumption in the code for now, but may add this in later. This member function does not return an explicit result, so it will be a `void` function.

Remember the result will be that all of the values of the `lower` and `upper` lists will be merged together and will replace the list `values` of the object this method is called on. We could implement this by freeing up (`delete`) the current list of `values`, and dynamically allocating a new block of memory before doing the merge. However this causes extra memory (space) needs if done this way, and we are implementing this function specifically to be a part of our merge sort implementation.

So instead we will just merge the results into the existing dynamically allocated set of `values`. Therefore it is a prerequisite that the `size` of the list that is being merged into is big enough to hold all of the values of the passed in `lower` and `upper` lists. You will need to test that the size is big enough before starting the merge and throw an exception if it is not.

The algorithm to do the merge is as follows

1. Start at the beginning index of both `lower` and `upper` lists.
2. Compare the current index of the two input arrays,
   - copy whichever value is smaller to the destination.
   - increment the index of whichever array we just copied from
   - repeat 2 until we have exhausted all of the values in one of the input arrays
3. After the loop, one of the arrays `lower` or `upper` will still have values left in it. Copy all remaining values from the array to the destination.

Conceptually this is not too difficult, but you need to keep track of the index into the two input arrays and the destination array. And you need to handle copying the remaining values after this from one of the arrays onto the end of the destination.

Do the usual before you begin a task. Uncomment the next test case in the testing file, and create a stub function and declare the member function signature in the header file. The merge function is a `void` function so the stub can be empty. Make sure your code compiles and runs the tests with the new set of tests uncommented before trying to begin your implementation.

Once you have implemented your `merge()` member function and it is passing the given tests, do the usual. Commit your changes and push them to the `Feedback` pull request of your GitHub repository.

Task 3: Implement the `sort()` Member Function using Merge Sort

Make sure you have done the usual to commit and document the previous task in your repository and you have created the issue for Task 3.

With your sublist copy constructor and the sorted `merge()` member function, the implementation of a `sort()` member function is relatively simple. We gave the general algorithm at the beginning of the tasks above.

You need to split the current list into two sublists of as equal as possible number of values. The dividing in half is what makes the merge sort a $\mathcal{O}(n \log n)$ algorithm. If you always divided the list into two parts where one was of size 1 and the other was the rest of the list, then merge sort devolves into something similar to bubble sort, a $\mathcal{O}(n^2)$ algorithm.

If the size of `this` instance is even, then you can divide the list exactly in half. For example, if the list size is 4, then you can divide into lists using the values 0,1 for the lower and 2,3 for the upper. If you divide the size by 2, you end up with the first index for your upper list that needs to be created. If the size of `this` is odd, then you can't divide

into exactly even lists. For example, if the size is 5, if you divide by 2 you get a result of 2.5. But if you perform integer division, this will be chopped off to 2 again. This works fine as the index to divide along, you again end up with values 0,1 in the lower list, and the values 2, 3 and 4 go in upper in that case.

If you are not certain how you can use your sublist copy constructor to perform the split, remember that the `this` implicit pointer name is a pointer to the instance of this object that you want to split. So you can do something like:

```
List lower = List(*this, 0, 2);
```

This will call the sublist copy constructor on this instance, and return a copy of the values from index 0 to index 2 inclusive, if you implemented the copy constructor correctly.

The `sort()` method will take no input parameters, and it is a `void` function because it returns no explicit result. The result should be to end up sorting the list of values that is being managed by `this` list instance that the sort method is called on.

Once again the merge sort algorithm is a recursive algorithm. Your base case is that, if you are asked to sort a list of size 1 or an empty list of size 0 then the list is already trivially sorted, so simply return and do nothing.

But if the list has 2 or more items you need to

1. Split this instance into 2 equally sized lists, the lower and upper copies of the halves of this list.
2. Call `sort()` recursively on both of these new copied sublists.
3. Call `merge()` on this instance with the returned, now sorted, `lower` and `upper` list copies.

And that is it. If you implement that correctly, your lists should end up being sorted in place, and you will be able to pass the tests in the next set of unit tests. Once you are satisfied with your implementation, commit your work and push it to the `Feedback` pull request of your repository.

Task 4: Implement `search()` Member Functions using Recursive Binary Search

Perform the usual prerequisite steps before starting task 4.

Once we have the ability to sort our `List` of strings, we should be able to now efficiently search it using a binary search. You will be required to implement the binary search for your `List` class using a recursive binary search algorithm. For now don't worry about checking if the list is sorted before performing the binary search, which needs the list to be sorted in order to work. We will assume that the list is sorted externally before the `search()` method is called, and fix that assumption next.

For the `search()` method, the first parameter is a string that is to be searched for. This should be passed in as a `const` parameter. Also the `search` method will be passed a `begin` and `end` integer index again, which indicate the current unsearched portion of the list of items. This method, as in our textbook and lecture examples, will return an `int` which is the index where the item was found in the list. If the item is not found, the global constant `NOT_FOUND` should be returned, which has been defined for you in the `List` header file.

The recursive algorithm for the binary search is as follows. The base case is, if `end` is less than or equal to `begin`, then there is 1 or less items left in the list to search. In that case you should test the item in the `begin` index and if the item is what we were looking for, you return the `begin` index where it was found. But if the item is not what was being looked for, then you return `NOT_FOUND` to indicate a failed search.

The recursive general case is another example of a divide-and-conquer approach. The algorithm is as follows:

1. calculate the middle index of the remaining portion of the list, e.g. $\frac{begin+end}{2}$
2. test the value at the middle index. If it is what we are searching for, return the middle index to indicate success.
3. otherwise the value must either be in the portion of the list before the middle or after the middle.
    - if the value we are searching for is less than the value in the middle index, recursively call the search again on the list from begin to middle - 1.
    - else the value should be greater than the value in the middle index, so recursively search from middle + 1 to end.

Once you are satisfied with your implementation of `search()` and it is passing the tests, perform the usual steps to commit and push your work document to the `Feedback` pull request.

Task 5: Implement `isSorted()` Member Function and Improve `search()` API

At the end of the last task it is up to the user to ensure that the list is sorted before doing a search. This could be dangerous, it would be better if we ensure that the list is sorted, and sort it if it is not before the search.

Also there is another wart or annoyance with the implementation of the `search()` method. To call it the user has to specify the begin and end index of the list they want to search. If there is a use case where they might only want to search a portion of the list, then this API (interface) would make sense to leave available to the user. However, almost always what the user really wants to do is to just specify the value to search for, and it is implied that the whole list should be searched for that value. It is common to need extra values like `begin` and `end` on divide-and-conquer recursive algorithms to keep track of what portions of something are being worked on in the recursive calls. However, it is also common to hide these. We could use default parameters, so that if the user does not specify the `begin` and `end` they would default to 0 and `size - 1` respectively. However, we are going to instead overload the `search()` function, using a second version of the function as a convenient place to first determine if the list needs to be sorted before we can perform the search.

However, we first need a method we can call that will give us an answer of `true` or `false` of whether or not the list is currently sorted. We could add another member variable that is set to true whenever the list is sorted, but this would not detect cases where we initialize or create a list with an already sorted set of values. You will instead write a method that will determine dynamically anytime it is needed whether or not the list is currently sorted or not.

This method is relatively simple conceptually. This method returns a `bool` result of `true` if the list is currently sorted and `false` if not. To determine if a list is sorted or not, you can start by looking at the first two values in index 0 and 1. If they are out of order (e.g. value in 0 is greater than the value in 1), then we can return `false` immediately, the list is not sorted. If they are in the correct order, we can test indexes 1 and 2 to see if they are in order, etc. and iterate through all adjacent pairs of values to see if we find any out of order. If we test all pairs and don't find any out of order then the list is sorted.

Add in the described member method named `isSorted()`. This method returns a `bool` result as we described and it has no parameters as input. This member should be a `const` member function, because it is returning information only, it does not modify the `List` when called.

Implement this method and get it to pass the next set of unit tests after uncommenting them in the test file. When you commit and push this work to the `Feedback` pull request, check that you are successfully passing all of the GitHub autograding actions.

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