Sorting Project Specifications

DoublyLinkedList::merge_sort

Purpose: Runs a recursive merge sort program using a public helper function and a private recursive algorithm.
Assumptions: There is an existing DoublyLinkedList which is the calling object that holds integer value in each node.
Inputs: There are no inputs to the helper function but the recursive function will take in a pointer to the first node of the sublist that needs to be sorted.
Outputs: The helper function does not have any outputs, the recursive function returns a Node pointer that represents the new head pointer.
State Changes: The head pointer will be updated by the return value of the recursive function, the tail will be updated after the head pointer in the helper function before the function returns.
Cases and Expected Behavior: In the case of an empty list or a list of size one, no recursive calls are made. In the case that there is more than one element, the merge sort algorithm will recursively call itself on the left half of the list and the right half of the list. The first half is defined by the first index to the middle index, the second half begins at middle + 1 and ends with the final index. The result of the recursive calls on the two lists (each should be a sorted sublist) is then merged using the merge function. The result of the merge function is then returned.

DoublyLinkedList::merge

Purpose: A helper function for merge_sort taking two sorted sublists and merging into one singular sorted sublist.
Assumptions: There are two sorted sublists which are to be merged
Inputs: Two pointers to two sorted sublists.
Outputs: one singular sorted sublist
State Changes: merging two sublists into one big sublist
Cases and Expected Behavior:
Both of the sublists are empty: return nullptr
One of the sublists is empty: return head of the full sublist
Both the sublists are full of sorted values to be merged : merge the sorted sublists

DoublyLinkedList::quick_sort

Purpose: Runs a recursive quick sort program using a public helper function and a private recursive algorithm.
Assumptions: There is an existing DoublyLinkedList which is the calling object that holds integer value in each node.

Inputs: The recursive function will take in a pointer to the first node of the sublist that needs to be sorted.
Outputs: The recursive function returns a Node pointer that represents the new head pointer.
State Changes: The head pointer will be updated by the return value of the recursive function, the tail will be updated after the head pointer in the helper function before the function returns.
Cases and Expected Behavior: In the case of an empty list or a list of size one, no recursive calls are made. In the case that there is more than one element the first node will be the pivot and will partition the list into a two segments based on if the nodes values are less than or greater than the pivot value. The pivot is then placed in the correct position and we recursively call the sort on either sides of the pivot until the entire list is sorted. The result of the quick sort is then returned.

DoublyLinkedList::partition

Purpose: Rearrange the passed doubly linked list to have values less than the pivot to the left of the pivot node and values greater than the pivot to the right of the pivot node.
Assumptions: There exists a doubly linked list of size two or more with integer values
Inputs: The lowest node and the highest node in the remaining list to be sorted
Outputs: Returns the pivot node
State Changes: The function may change the data stored in each node if the node's stored value is less than or greater than the pivot while in the wrong place.
Cases and Expected Behavior: When taking in a doubly linked list's low and high node, it will return the pivot node and sort both side of the pivot into less than on the left and greater than on the right.

DoublyLinkedList::insertion_sort

Purpose: Runs an insertion sort program by inserting each element into its correct position within the doubly linked list
Assumptions: There is an existing doubly linked list which is the calling object that holds integer values in each node
Inputs: A pointer to the first node in the doubly linked list
Outputs: A node pointer that represents the head pointer
State Changes: The doubly linked list will become sorted
Cases and Expected Behavior: In the case of an empty list or a list of size one, no changes are made. In the case that there is more than one element the insertion sort will traverse the list, comparing values and swapping the minimum element with the first element of the unsorted doubly linked list until the list is fully sorted. In the event the list is already sorted, it will just traverse the list, make no changes, and return the original list.

VectorSorter::merge_sort

Purpose: Recursively sort a vector by splitting it in half repeatedly, into smaller and smaller sub-arrays, before merging them into a single sorted vector. Uses helper functions merge_sort(vector<int>& arr) and merge(). merge_sort(vector<int>& arr) calls the main merge_sort with correct inputs.
Assumptions: There exists a vector.
Inputs: A vector of integers, a left integer representing the leftmost index, and a right integer representing the rightmost index.
Outputs: Void return type, but the vector is now sorted.
State Changes: The vector is now sorted.
Cases and Expected Behavior:

Case 1: If the case of either a single element or no elements, it returns the vector immediately.

Case 2: In the case of more than one element, it will recursively call itself, dividing the vector into smaller sub-arrays before merging them into a correct order. If the array is already sorted, no changes will be made.

VectorSorter::merge

Purpose: A helper function to the merge_sort function. Merge two vector subarrays into a single, now sorted, vector by comparing the first element of each vector and moving the lowest element into the new vector.
Assumptions: There is a vector of integers with two sorted sub-arrays
Inputs: A vector of integers, a left, middle, and right integer
Outputs: Void
State Changes: The vector will now be fully sorted, without sub-arrays.
Cases and Expected Behavior: A valid vector is passed, and assuming the sub-arrays are presorted, sorts the sub-arrays into a single array.

VectorSorter::quick_sort

Purpose: Sort a vector with its elements in ascending order. A public helper function (named quick_sort) aids a private member function (of the same name) that recursively calls itself to perform the quick sort algorithm.
Assumptions: A vector is declared and is potentially assigned some number of elements (e.g. integer values). If the vector is partitioned, in each recursive call the first value in the given sub-vector is chosen as the pivot. (Note there exists other methods for choosing pivots.)
Inputs: The address of the vector is passed into the private recursive function. Two integer variables marking the end and beginning of the given (sub-)vector, respectively, should be declared.
Outputs: A partitioned vector in which elements have been recursively sorted around their pivot is the output. The final returned vector will be the same length as the original vector, only now its elements are arranged / sorted in ascending order.

State Changes: Though the length of the vector is the same as the original vector, its elements are now sorted in ascending order.
Cases and Expected Behavior:

Case 1: The vector contains zero elements or only one element. No further partitioning of the vector is required, and the vector, as it is, is returned.

Case 2: The vector contains more than one element. In the first call the first element is chosen as the pivot. All other elements are then rearranged around this value, with smaller values being partitioned to its left, and larger values being partitioned to its right. The smaller values form a left sub-vector, and the larger values form a right sub-vector. The function then recursively calls itself on the left and right sub-vector, where in each call the first value in the sub-vector is chosen as the pivot.

- **In the case that the vector is already sorted** (its elements are in ascending order) the vector will be partitioned, and the function will recursively call itself. However, because the vector is already sorted no rearrangements of elements are required. With each call
- **The vector is in reverse order (its elements are in descending order)**. The function recursively calls itself and
- **The vector contains duplicate elements**.

VectorSorter::insertion_sort

Purpose: Sort a vector with its elements in ascending order. A public helper function (named insertion_sort) aids a private member function (of the same name) that recursively calls itself to perform the insertion sort algorithm.
Assumptions: A vector is declared and is potentially assigned some number of elements (e.g. integer values).
Inputs: The address of the vector is passed into the private recursive function. An integer value marking the end of the vector should be declared (to be passed into the recursive function).
Outputs: A vector with its elements in ascending / sorted order is the output.
State Changes: Though the length of the vector has not changed from its original length, its elements are now sorted in ascending order.
Cases and Expected Behavior:

Case 1: The vector contains no elements or one element. The insertion sort algorithm does not need to be applied to the vector.

Case 2: The vector contains more than one element. The first element of the vector is assumed to be sorted, and the second element is compared to it. If it is greater than the first element, then it is swapped with the first element. The third element is then compared to these values, and, if applicable, swapped accordingly into its correct position. This recursive process continues until all elements are sorted into their correct position in the vector.

- **If the vector is already sorted** each element needs to be compared with only its immediate predecessor, and no swapping needs to occur.

- **If the vector is sorted in reverse order** each element will need to be compared and based upon its original position in the vector, swapped through the entire length of the vector.

Evaluator::ingest

Purpose: Function to ingest and store each of the three groups of 4 lines.
Assumptions: There is a file named evaluation_cases.txt with three groups of four lines where in the first group, each line has 1000 four-digit integers separated by spaces. The next group is 10,000 five-digit integers and the last group is 100,000 six-digit integers.
Inputs: A string representing the file name
Outputs: None
State Changes: Updates a 2D vector to hold each of the three testing cases blocks.
Cases and Expected Behavior: If the file could not be opened, we will return an error. Else, we expect it to contain the right values and update the 2D vector test_cases to hold all of the test blocks by reading each of the four lines in each of the 3 blocks.

Evaluator::merge_comparison

Purpose: Run a merge sort algorithm on both a doubly linked list and a vector for each of the three test cases. This function will time the process of sorting for both the doubly linked list and the vector and store the results in appropriate member variables. These results will be used in later comparisons.
Assumptions: The test_cases member variable is initialized with data, where each case is a vector containing integer values. Merge sort implementation for both the doubly linked list and the vector exists and both should perform as expected. The time measurement of sorting on both data structures is assumed to be correct and accurate, and there exist member variables, properly defined, to store the timing results for structure.
Inputs: No inputs, instead we just access member variables.
Outputs: No outputs, instead we just set member variables
State Changes: The member variable merge_results will be updated with the sorting time for both the doubly linked list and the vector.
Cases and Expected Behavior: Because each element (i.e. each test case) contains a large number of integers we expect the sorting time for the doubly linked list and the vector to be larger (than if each element were to have significantly fewer integers). For this particular sorting algorithm, we may expect that it takes less time to sort on the vector than on the doubly linked list.

Evaluator::quick_comparison

Purpose: Run a quick sort algorithm on both a doubly linked list and a vector for each of the three test cases. This function will time the process of sorting for both the doubly linked list and the vector and store the results in appropriate member variables. These results will later be used in comparisons.

Assumptions: The test_cases member variable is initialized with data, where each case is a vector containing integer values. Quick sort implementation for both the doubly linked list and the vector exists and both should perform as expected. The time measurement of sorting on both data structures is assumed to be correct and accurate, and there exist member variables, properly defined, to store the timing results for each structure.
Inputs: No inputs, instead we just access member variables.
Outputs: No outputs, instead we just set member variables
State Changes: The member variable quick_results will be updated with the sorting time for both the doubly linked list and the vector
Cases and Expected Behavior: Because each element (i.e. each test case) contains a large number of integers we expect the sorting time for the doubly linked list and the vector to be larger (than if each element were to have significantly fewer integers). For this particular sorting algorithm, we may expect that it takes less time to sort on the vector than on the doubly linked list.

Evaluator::insertion_comparison

Purpose: Run an insertion sort algorithm on both a doubly linked list and a vector for each of the three test cases. This function will time the process of sorting for both the doubly linked list and the vector and store the results in appropriate member variables. These results will later be used in comparisons.
Assumptions: The test_cases member variable is initialized with data, where each case is a vector containing integer values. Insertion sort implementation for both the doubly linked list and the vector exists and both should perform as expected. The time measurement of sorting on both data structures is assumed to be correct and accurate, and there exist member variables, properly defined, to store the timing results for each structure.
Inputs: No inputs, instead we just access member variables.
Outputs: No outputs, instead we just set member variables
State Changes: The member variable insertion_results will be updated with the sorting time for both the doubly linked list and the vector.
Cases and Expected Behavior: Because each element (i.e. each test case) contains many integers we expect the insertion sorting time for the doubly linked list and the vector to be larger (than if each element were to have significantly fewer integers). For this particular sorting algorithm, we may expect that it takes less time to sort on the vector than on the doubly linked list.

Evaluator::evaluate

Purpose: Print a **pretty** table showing the results of our sort_comparison functions.
Assumptions: We assume that each of the three sorting comparison functions have already been called, thus filling the necessary member variables with the right data.

Inputs: No inputs.
Outputs: It will output a pretty table to the terminal
State Changes: A table will be printed to the terminal
Cases and Expected Behavior: If the sort comparison functions are properly called before the evaluate() function call, the function will output a neat table showing the sorting times for the dll and vectors.