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 sub-list 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 sub-list) is then merged using the merge function. The result of the merge function is then returned.

DoublyLinkedList::merge

Purpose: Merges two sorted sub-arrays within a vector into a single sorted sub-array as part of the merge sort algorithm.

Assumptions: The vector contains at least two sorted sub-arrays defined by the indices left, mid and right.

Inputs: A reference to a vector of integers and 3 integers: left, mid and right representing the range to merge.

Outputs:This function returns void and performs the merge operation in place on the vector.

State Changes: The range of elements from left to right in the input vector will be sorted.

Cases and Expected Behavior: In the case where one of the sub-arrays is empty, all elements from the non-empty sub-array are copied directly. In the general case, the function creates two temporary vectors to hold the elements of the left and right sub-arrays. It then compares elements from both and inserts them back into the original vector in sorted order. If there are remaining elements in either temporary vector after the main comparison loop, they are appended to the result. After completion, the sub-array defined from ‘left’ to ‘right’ will be fully sorted.

DoublyLinkedList::quick_sort

Purpose: Runs an in-place quicksort algorithm on a doubly linked list using a public helper function and a private recursive algorithm that partitions and sorts sublists.

Assumptions: There is an existing DoublyLinkedList which is the calling object that holds integer values in each node.

Inputs: The public quick_sort() function takes no inputs and sorts the entire list. The private recursive version takes two pointers: start (the first node in the sub-list) and end (the last node in the sub-list). The partition function also takes these two pointers and returns the final pivot position.

Outputs: All functions return void, and sorting is done in-place by swapping values.

State Changes: The node values are rearranged such that the list becomes sorted in ascending order. The node links (prev, next) remain unchanged during the sort. The head and tail pointers remain valid.

Cases and Expected Behavior: If the list is empty or has only one element, the function returns immediately without changes. The quick_sort(start, end) function recursively partitions the listA pivot is selected as the start node. The partition function compares elements from the end toward the start, collecting larger-than-pivot values toward the end. All values greater than the pivot are swapped toward the middle, and finally the pivot is placed in its correct position. After the pivot is positioned, the function recursively sorts the left and right sublists. The result is a sorted linked list from smallest to largest value.

DoublyLinkedList::insertion_sort

Purpose: Sort a Doubly Linked List filled with random integer values

Assumptions: The list exists and contains integer values

Inputs: None

Outputs: None

State Changes: Structure of the list stays the same but values at each Node pointer is switched to the in order integer value of that node position

Cases and Expected Behavior:
Makes comparisons with values at a certain node, if the value being compared is smaller than the original value, the original value will be shifted further in the list and the smaller value will be correctly inserted before the original value.

VectorSorter::merge_sort

Purpose: Run a recursive merge sort on a vector of integers using a public wrapper function and a private recursive algorithm that divides the vector into sub-lists and merges them.

Assumptions: The input is a std::vector<int> passed by reference. The vector can contain zero or more elements and is modifiable in-place.

Inputs: The public wrapper version takes a reference to a vector of integers and begins the sort from index 0 to vec.size()-1. The private recursive version takes the same vector reference, along with two integers left and right representing the current bounds of the sub-list to sort.

Outputs: None. Sorting is done in place; the original vector is modified.

State Changes: The elements of the vector are reordered in ascending order. No elements are added or removed.

Cases and Expected Behavior: If the vector is empty or contains only one element, the function exits without making any recursive calls. If there are two or more elements, the midpoint is calculated using mid= left + (right-left)/2. The function recursively calls itself to sort the left half [left,mid] and right half [mid+1,right]. The merge function is then used to merge these two sorted sub-lists into a single sorted portion of the vector.

VectorSorter::merge

Purpose: Merges two sorted sub-arrays of a vector (left and right) into one sorted segment. Used internally by the merge sort function.

Assumptions: The vector is being sorted in place. The sub-arrays defined by left, mid, and right are contiguous and valid: Left sub-array: vec[left] to vec[mid], and the right sub-array: vec[left] to vec[right]

Inputs: A reference to a vector of integers.

Three integers: left, mid, and right.

Outputs: None. The sorted result is written directly into the original vector.

State Changes: The range of indices from left to right in the input vector is modified and becomes sorted. And the temporary vectors are used to hold copies of the left and right sub-arrays, and elements are copied back in sorted order.

Cases and Expected Behavior: Temporary vectors leftSortesVec and rightSortesVecare created by copying elements from the original vector. Pointers i, j, and k are used to merge the temporary vectors back into the original one: i traverses the left half, j traverses the right half,k writes into the main vector.

If one side runs out of elements during the merge, the remaining elements from the other side are copied directly.

VectorSorter::quick_sort

Purpose: Sorts a vector of integers using the quick sort algorithm with a public wrapper function and a private recursive function.

Assumptions: The vector contains integer values.

Inputs: Public: a reference to a vector of integers. Recursive: a reference to the vector, and two integers left and right representing the current sub-range to sort.

Outputs: Both versions return void and sort the vector in place.

State Changes: The elements in the input vector are reordered such that the vector is sorted in ascending order.

Cases and Expected Behavior: If the vector is empty or contains one element, no sorting is performed.If the vector contains multiple elements A pivot is selected (in this case, the first element of the current sub-array). The partition function rearranges the sub-array so that elements less than or equal to the pivot come before it, and those greater come after. Quick sort is then recursively applied to the left and right partitions of the pivot. At all, sorting is performed in place without the use of additional vectors.

VectorSorter::partition (inner function)

Purpose:Reorders elements in the sub-array such that all elements less than or equal to the pivot are on the left, and all greater elements are on the right.

Assumptions:The pivot is chosen as the first element of the current sub-array.

Inputs: A reference to the vector and 2 integers, left and right, defining the sub-range to partition.

Outputs: Returns an integer representing the final position of the pivot element after partitioning.

State Changes:Reorders the sub-array vec[left…right] around the pivot.

Cases and Expected Behavior: This function would scan inward from both ends: It could increment low until an element greater than the pivot is found and decrement high until an element less than or equal to the pivot is found. If low is less than highs, the 2 elements swapped at low and high. After exiting the loop, swaps the pivot with the element at high, ensuring it's in the correct sorted position. In the end, return the index high so that quick_sort can recur on both sides of the pivot.

VectorSorter::insertion_sort

Purpose: Sort a vector of integers from smallest to largest.

Assumptions: The vector is a vector of integers.

Inputs: vector of integers

Outputs: None

State Changes:The vector of integers will be changed, where the smallest integer value will be placed at the beginning and each next smallest value will come after the smallest.

Cases and Expected Behavior:In the case of one value in the vector, nothing will change because the vector will already be sorted, and the same if it is an empty vector. An integer will be set to a certain index in the list and each value will be compared to it, depending on being greater than or less than that value, each value will be inserted accordingly.

Evaluator::ingest

Purpose: Loads numerical datasets from evaluation_cases.txt into vectors and doubly linked lists. Selects subsets of different sizes for small (100), medium (1000), and large (10000) datasets.

Assumptions: evaluation_cases.txt exists and follows the expected format. Each dataset section contains exactly four lines of numbers.Random selection of consecutive numbers is valid for testing.

Inputs: None

Outputs: Populates data1_vec, data1_dll, data2_vec, data2_dll, data3_vec, and data3_dll. But no outputs directly.

State Changes: Modifies the data1_vec, data1_dll, data2_vec, data2_dll, data3_vec, and data3_dll attributes.

Cases and Expected Behavior: If the file is missing, an error is thrown. If fewer numbers are present than expected, the function may fail or select smaller ranges.

Evaluator::MergeComparison

Purpose: Runs merge sort on small, medium, and large datasets and measures sorting time for both vectors and doubly linked lists.

Assumptions: data1_vec, data1_dll, data2_vec, data2_dll, data3_vec, and data3_dll contain valid data and merge_sort is correctly implemented.

Inputs: Internal datasets loaded by Ingest.

Outputs: Updates dll_merge.time1, dll_merge.time2, dll_merge.time3, vec_merge.time1, vec_merge.time2, and vec_merge.time3 while printing sorting times.

State Changes: Stores merge sort timing results in the corresponding timing attributes.

Cases and Expected Behavior: If sorting succeeds, the times are printed and stored; if merge_sort fails, an error is thrown.

Evaluator::QuickComparison

Purpose: Runs quick sort on small, medium, and large datasets and measures sorting time for both vectors and doubly linked lists.

Assumptions: data1_vec, data1_dll, data2_vec, data2_dll, data3_vec, and data3_dll contain valid data and quick_sort is correctly implemented.

Inputs: Internal datasets loaded by Ingest.

Outputs: Updates dll_quick.time1, dll_quick.time2, dll_quick.time3, vec_quick.time1, vec_quick.time2, and vec_quick.time3 while printing sorting times.

State Changes: Stores quick sort timing results in the corresponding timing attributes.

Cases and Expected Behavior: If sorting succeeds, the times are printed and stored; if quick_sort fails, an error occurs.

Evaluator::InsertionComparison

Purpose: Runs insertion sort on small, medium, and large datasets and measures sorting time for both vectors and doubly linked lists.

Assumptions: data1_vec, data1_dll, data2_vec, data2_dll, data3_vec, and data3_dll contain valid data and insertion_sort is correctly implemented.

Inputs: Internal datasets loaded by Ingest.

Outputs: Updates dll_insert.time1, dll_insert.time2, dll_insert.time3, vec_insert.time1, vec_insert.time2, and vec_insert.time3 while printing sorting times.

State Changes: Stores insertion sort timing results in the corresponding timing attributes.

Cases and Expected Behavior: If sorting succeeds, the times are printed and stored; if insertion_sort fails, an error occurs.

Evaluator::Evaluate

Purpose: Runs all sorting comparisons (insertion, merge, and quick) and prints a performance table summarizing the results.

Assumptions: Ingest has been called successfully and all sorting functions are correctly implemented.

Inputs: Internal datasets loaded by Ingest.

Outputs: Calls InsertionComparison, MergeComparison, and QuickComparison and prints a detailed performance table with timing results.

State Changes: Populates all timing variables with sorting durations.

Cases and Expected Behavior: If all sorts work correctly, a performance table is printed; if any sorting function fails, an error message is printed and the error is thrown.