Harrison Weiss
Data Structures
Dr. Fine
4.23.2018

Project 34 Write-Up

The results of the timing tests are about what I expected: the STL performed the best out of the three data structures. This is obviously due to the fact that the STL version of a doubly-linked list, *std::list*, has been perfected over the years by people who have been studying the topic for a majority of their careers. In addition to that, any functions not directly implemented by *std::list*, like random_shuffle, were implemented by me using *std::vector* to perform operations that could not be done directly on the list. So, despite the fact that my *list::random_shuffle* function copies the info (using *std::copy*) to a vector, performs the necessary operation, and then puts the new content into a cleared list, it still outperformed the direct implementation of the same function for both my linked-list and dynamic array. Again, this is because the *std::algorithm* and *std::vector* functions have been (nearly) perfect. Most of those functions, according to documentation, perform at linear, near-linear, or constant time.

For my implementations of a linked-list and dynamic array, the linked-list generally performed better than the linked list. This is likely due to the fact that the dynamic array has to resize to prevent segmentation faulting, whereas the linked-list is dealing with non-sequential memory, meaning it can append, prepend, and insert with near-linear, if not linear, time. The dynamic array *remove_all* function, however, is much more efficient than the linked-list: because the memory is dynamically allocated for the dynamic array, to delete the contents of the array, we only have to use the keywork **delete** to release the memory and 'remove' the elements from our array. The linked-list would perform the same action in linear time since it needs to travel through the entire list and null pointers and destruct objects to prevent memory leaks. If we wanted to, we could leak memory everywhere and make it a constant time operation by just changing the head and tail pointers. Since I was unable to implement the rotate, reverse, and *random_shuffle* functions for the linked-list, I looked at documentation for the same functions within the STL. Within a vector, *std::rotate* is an up-to linear time operation. With this in mind, we can think about what that means for rotating the pointers in a linked-list. Since we have to iterate through the list to get the next pointer, a linear time operation, and likely have to hold those pointers within another data structure, a constant time copy operation, and perform those actions *n* number of times, a linked-list rotate function would likely be *n(log n)*. According to cplusplus.com, *std::reverse* is a linear function in list size on a doubly-linked list, whereas *std::reverse* is a linear function in container size for a singly-linked list (*std::forward_list*). My implementation of *std::reverse*, likely would have involved a second data structure to store the pointers of the list, a linear time traversal through the list and a constant time copy, and then moving from the front of the list and the back of the data structure, appropriately reassign the pointers, an *n(log n)* operation. Finally, to implement *random_shuffle*, there would be a swap function. This swap function would have a time complexity of O(n-1) for a singly-linked list. There would also need to be constant time checks for conditions where we are swapping a head to tail node to prevent losing a way into the list and mismanaging the tail pointer.