Assignment One

Ethan Morton

Ethan.Morton1@Marist.edu

September 7, 2024

1 Problem One

Concept	Line Number	Explaination	Result
Node	node.hpp 8	A struct that contains a generic piece of data, and a pointer to a node of the same generic type. It is commonly used in data structures that use a linked-list to store a collection of elements. In this case it is used for the Stack and Queue structs.	
Stack	stack.hpp 14	A struct that wraps a pointer to a Node, and provides some Stack-related functions. The Stack is LIFO which means the last item to be added is the first item to be removed. Functions: * push - adds to the top of the stack * pop - removes the element at the top of the stack * peek - views the element at the top of the stack * isEmpty - if the pointer to the top of the stack is null, then the stack is empty	
Queue	queue.hpp 14	A struct that contains 2 pointers to Nodes, one being the head and the other being the tail. The Queue is FIFO which means that the first item to be added is the first item to be removed. Functions: * enqueue - adds to the tail of the queue * dequeue - removes from the head of the queue * peek - views the element at the head of the queue * isEmpty - if the pointers to the head or tail are null, then the queue is empty	

Concept	Line Number	Explaination	Result
Palindromes	main.cpp 159	Lines were read one at a time from magicitems.txt, and the strings were adjusted to ignore case and space. Characters were added to the stack and queue in the same order, one at a time. After all characters were added, they were removed one at a time from the stack and queue. If the characters ever didn't match, then the line wasn't a palindrome. If all characters matched, then the line was a palindrome. This is because the stack reverses the order of elements when they are all pushed then all popped.	666 lines read 15 palindromes found
Selection Sort	main.cpp 31	This algorithm iterates over the entire array in the outer loop. Then in the inner loop, it iterates over the array from the current index onwards. In the inner loop, it's looking for the smallest element in the sub-array. Once the inner loop finishes, it swaps the element at the current index with the smallest element. Then the outer loop increments the current index, and the process repeats until the list is sorted.	221,445 comparisons every time. O(n^2) time complexity. 666^2 = 443,556. The exact time complexity is n(n-1)/2 which is 221,445. On the first pass, we do n-1 comparisons because the first element is assumed to be sorted. Each succeeding pass does one less comparison than the last. That gives us the equation (n-1)+(n-2)+(n-3)+...+1, which simplifies to n(n-1)/2.
Insertion Sort	main.cpp 52	This algorithm iterates over the entire array in the outer loop. The element at the current index is called the key. Everything before the current index is sorted, and everything after is unsorted. In each iteration of the outer loop, we are trying to find where the key should be inserted. The inner loop iterates the array in reverse order from the current index - 1 down until it finds where the key should go. Then the outer loop increments the current index, and the process repeats until the list is sorted.	111,575 comparisons on average with 10 samples. O(n^2) time complexity. The approximate time complexity is around n(n-1)/4 which is 110,722.5. The insertion sort is just about as efficient as expected. On the first pass, we only need to do 1 comparison since only 1 item is assumed to be sorted. On the second pass, we need to do 2 comparisons. We do this all the way up to n-1 comparisons on the final pass. We get the equation 1+2+3+...+n-1, which simplifies to n(n-1)/2. Since the array is randomly sorted, we only need to make about half of the comparisons since half of them are already sorted relative to adjacent items.

Concept	Line Number	Explaination	Result
Merge Sort	main.cpp 120	This algorithm recursively divides the list in half, into a left side and right side, until each sub-array cannot be further divided. Then the tiny sub-arrays are compared with the ones next to each other. The sub-arrays are merged back together and sorted as the recursion call-stack makes its way back to the beginning.	6,087 comparisons on average with 10 samples. O(n*log2(n)) time complexity. 666*log2(666) = 6,246.6659... The merge sort is slightly more efficient than the time complexity suggests. The number of elements in a sub-array halves each time it's divided. You can divide the array log2(n) times before it cannot be further divided. When merging arrays back together, they are compared in linear time. So for each time you divide an array, you have to then compare it, which gives the equation n*log2(n).
Quick Sort	main.cpp 149	This is a recursive algorithm that starts by picking an element to be a pivot. Elements smaller than the pivot are moved into one sub-array, and elements bigger are moved into another. This is done until the array cannot be further divided. At this point, the elements should already be sorted, so it just merges the array back together.	Anywhere between 6,000 and 8,500 comparisons, and an average of 6,719 with 10 samples. O(n*log2(n)) time complexity. The quick sort is a little less efficient than expected, and this is probably due to the pivot choice not being optimized. When you pick a pivot, you have to compare each element in the sub-array relative to the pivot, which is done in linear time. Since you can make log2(n) divisions, we get the equation n*log2(n). If the pivot is bad, then the recursion tree is unbalanced and it can have up to n divisions, leading to a time complexity of O(n^2) in the worst case.
Knuth Shuffle	main.cpp 17	This algorithm iterates over an array in reverse order. For each index, it generates a random number between 0 and the index. Then it swaps the elements at the index and the RNG number.	A nicely shuffled array.