Capstone Project? ArrayStringListHelpers.cs

(Capstone Module)

InsertIntoArray(int[] arr, int index, int value)

What it does?

Inserts a value into an array at a specific index then shifts everything to the right. Arrays have a fixed size so anything getting shifted out of bound is gone..

Why this complexity?

There is a linear cost to O(n), indexes have to move to n-1 element to the right to make room.

Patterns / performance notes?

Inserts at the front take the biggest hit since every element has to move down one spot, while inserts near the end barely cost anything. Arrays keep good performance from staying in continuous memory, but it's still a full copy of the elements. Expanding an array is not realistic, you have to adjust and copy it onto a new array with the proper element capacity. For multiple inserts, a List<T> makes more sense since it can expand automatically and handle shifting for you.

DeleteFromArray(int[] arr, int index)

What it does:

Deletes an element at a specific index by sliding everything after it to the left until one last slot gets cleared out after.

Why this complexity:

It's O(n) because each element after the deleted one has to move over by one. It's the same idea as insert, just shifting in the other direction.

Patterns / performance notes:

Deleting near the start costs the most since the entire array has to move, while deleting at the very end is more cost efficient. Arrays stay efficient thanks to their continuous memory layout. Clearing the last slot helps avoid leftover values showing up where they shouldn't.

ConcatenateNamesNaive(string[] names)

What it does:

Join all the names together using += inside a loop. It's straightforward and easy to read, but slow.

Why this complexity:

It's $O(n^2)$ because every time you use +=, a new string is created, and all the previous characters have to be copied again. The work load grows as the string grows..

Patterns / performance notes:

Works fine for smaller inputs, but doesn't handle scaling very well. Each new string adds more on memory since new copies keep piling up. You can actually see the slowdown once the input size gets large, and it becomes clear why StringBuilder exists.

ConcatenateNamesBuilder(string[] names)

What it does:

Builds a full string builder, appending names efficiently without constantly recreating new strings.

Why this complexity:

It runs in O(n) because it just adds characters to a single growing string, skipping the copy work from the naive approach.

Patterns / performance notes:

If you already have a good idea of how big the final string will be, setting the StringBuilder's capacity upfront helps. For most joining tasks, string.Join is even cleaner, but StringBuilder is best when you have a lot of conditional or step-by-step appending.

InsertIntoList(List<int> list, int index, int value)

What it does:

Inserts a new value into a List<int> at a given position. The list shifts elements to the right to make space, just like an array would.

Why this complexity:

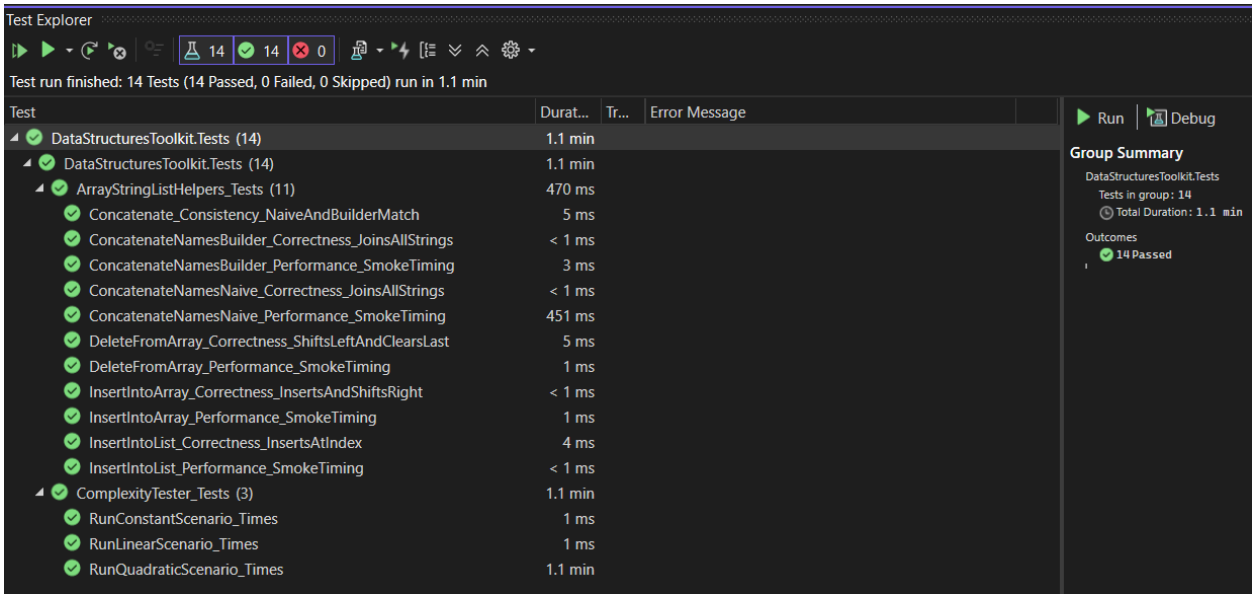
It’s O(n) for inserts in the middle because everything after the index has to move one slot. Adding to the end is O(1) since lists grow automatically in chunks when they need more room.

Patterns / performance notes:

When you’re mostly adding to the end, List<T> is the best for it as it is fast, clean, and easy to manage. If you’re doing a lot of inserts or deletes in the middle, you’ll still hit that linear cost. In those cases, something like a linked list or another data structure might fit better. List<T> has the right balance between speed and flexibility.

Summary

List<T> still feels like the best all-around option when adding at the end, where it’s both fast and effortless. Arrays are solid when you need fixed size and tight memory, inserts and deletes can quickly become costly. The naive string concatenation works fine for quick tests, issues when input grows, while StringBuilder keeps things running smooth and consistent. For simple data handling, List<T> gives the best performance and convenience.


Test Explorer
14
14
0
Test run finished: 14 Tests (14 Passed, 0 Failed, 0 Skipped) run in 1.1 min
Test
Durat...
Tr...
Error Message
DataStructuresToolkit.Tests (14) 1.1 min
DataStructuresToolkit.Tests (14) 1.1 min
ArrayStringListHelpers_Tests (11) 470 ms
Concatenate_Consistency_NaiveAndBuilderMatch 5 ms
ConcatenateNamesBuilder_Correctness_JoinsAllStrings < 1 ms
ConcatenateNamesBuilder_Performance_SmokeTiming 3 ms
ConcatenateNamesNaive_Correctness_JoinsAllStrings < 1 ms
ConcatenateNamesNaive_Performance_SmokeTiming 451 ms
DeleteFromArray_Correctness_ShiftsLeftAndClearsLast 5 ms
DeleteFromArray_Performance_SmokeTiming 1 ms
InsertIntoArray_Correctness_InsertsAndShiftsRight < 1 ms
InsertIntoArray_Performance_SmokeTiming 1 ms
InsertIntoList_Correctness_InsertsAtIndex 4 ms
InsertIntoList_Performance_SmokeTiming < 1 ms
ComplexityTester_Tests (3) 1.1 min
RunConstantScenario_Times 1 ms
RunLinearScenario_Times 1 ms
RunQuadraticScenario_Times 1.1 min
Run
Debug
Group Summary
DataStructuresToolkit.Tests
Tests in group: 14
Total Duration: 1.1 min
Outcomes
14 Passed