Ta Quang Tung – 104222196

COS20007 – 9.3HD – Research Project Plan

Project Title: Comparison of different Collection classes offered by the .NET library

Abstract

The .NET framework provides various classes to organize data into collections of items. Some of the most common ones include Arrays, Lists, Dictionaries, Stacks, Queues, and Linked Lists. Each of these data structures differs in terms of use case, feature support, and time complexity. This paper compares these classes by measuring the time it takes to perform common operations such as insertion and deletion on them. Results show that execution time varies significantly across operations and collection classes. This means that programmers will have to carefully consider the collection class they use to optimize their application performance.

Introduction

Collections are one of the fundamental building blocks of programs, offering programmers the ability to group items of the same data type together into a single logical unit called a collection. The most primitive of these is arrays, which in many languages are contiguous blocks of memory used to store same-type values. In C#, arrays are fixed-size, meaning that programmers must declare the size on array creation and that growing the array requires copying it to a larger block of memory. This rigidity makes expanding or shrinking the collection difficult. To make the process of using a dynamic array easier, C# offers lists, which can grow dynamically. C# also offers many other types of collections to serve different purposes and access patterns, such as stacks and queues.

Picking the correct type of collections is critical to the speed and resource usage of an application. With so many collection types to choose from, however, it can be difficult for programmers to pick the right one for the job. The .NET framework offers a range of collection classes for different use cases, such as dictionaries for fast look-up by key, queues for first-in-first-out access, or stacks for last-in-first-out access. These collection classes also have generic, thread-safe, and immutable variants provided by the System.Collections.Generic, System.Collections.Concurrent, and System.Collections.Immutable namespaces respectively for advanced use cases.

This paper will compare six generic collection classes offered by the .NET framework in terms of their time performance in various collection operations. The collections examined are arrays, lists, dictionaries, stacks, queues, and linked lists, all of which are commonly used data structures. The non-generic, thread-safe, and immutable variants of these classes are beyond the scope of this paper. The array class is provided by the System namespace while the other classes are provided by the System.Collections.Generic namespace. The operations performed are adding to the beginning, adding to the end, adding at an index, removing by value, removing at the beginning, removing at the end, removing at an index, and finding. The table below summarizes the collection classes' support for these operations and the methods that enable them.

	Add to beginning	Add to end	At at index	Remove by	Remove at beginning	Remove at end	Remove at index	Find

				value				
Array	X	X	X	X	X	X	X	Find
List	Insert	Add	Insert	Remove	RemoveAt	RemoveAt	RemoveAt	Find
Dictionary	X	Add	X	X	X	X	X	X
Queue	X	Enqueue	X	X	Dequeue	X	X	X
Stack	Push	X	X	X	X	Pop	X	X
LinkedList	AddFirst	AddLast	X	Remove	RemoveFirst	RemoveLast	X	Find

It should be noted that the notion of beginning and end is not built into some of the above collections. For instance, a stack does not have a notion of beginning or end; programmers are only concerned with what is on top of the stack. The notion of beginning and end for such collections in this paper is defined in terms of what comes out first or last when looping over the collection with foreach. For example, if item A comes out first from stack S in a foreach loop, A is considered the beginning of stack S.

Additionally, when discussing operations that involve an index such as adding a value at an index, this paper is only concerned with zero-based integer indexes. As such, even though a collection like dictionaries have keys to identify values, operations that involve keys are not considered.

Finally, this paper only examines operations that have methods built into their corresponding classes in the .NET library. However, just because an operation has no built-in support does not mean that programmers have no other methods of performing it. For instance, .NET does not have a native method to find and get the value of an item in a queue; however, programmers can easily implement this with a foreach loop.

Method

To obtain the results, a test case is defined for each operation and run on the collection classes that support that operation. Each test case is run 100 times per collection class. The execution time of each iteration is measured in milliseconds using the System.Diagnostics.Stopwatch class. The execution times of the iterations are then summed and averaged to find the mean execution time for an operation on a particular collection class. The test cases per operation are defined as follows:

Operation	Test case
Add to beginning	Add 100,000 values to the beginning of the collection.
Add to end	Add 1,000,000 values to the end of the collection.
Add at index	Add 100,000 values to the collection, each at a randomly chosen index.
Remove by value	Removes 100 values from the collection containing 1,000,000 values.
Remove at beginning	Removes 100 values from the start of the collection.
Remove at end	Removes 100 values from the end of the collection.
Remove at index	Removes 100 values from the collection, each at a randomly chosen valid index.

Find	Find 100 values from the collection containing 1,000,000 values.

The following code illustrates the implementation of the “add to end” operation on a list. Other operations are implemented in a similar fashion.

```
public static void MeasureAddEndList() {
      var timer = Stopwatch.StartNew();

      List<int> list = new List<int>();

      long[] results = new long[100];

      for (int t = 0; t < 100; t++) {
          timer.Reset();
          timer.Start();

          for (int i = 0; i < 1000000; i++) {
              list.Add(i);
          }

          timer.Stop();
          list = new List<int>();
          results[t] = timer.ElapsedMilliseconds;
      }

      long sum = 0;
      double average;

      foreach (long r in results) sum += r;
      average = (double)sum / 100;

      Console.WriteLine("Average elapsed time: {0}ms.", average);
  }
```

Results

The following results were obtained on .NET version 6.0. Data is measured in milliseconds.

	Add to beginning	Add to end	At at index	Remove by value	Remove at beginning	Remove at end	Remove at index	Find

Array	X	X	X	X	X	X	X	1.21
List	470.55	7.9	211.27	29.91	63.54	0	26.8	1.24
Dictionary	X	26.72	X	X	X	X	X	X
Queue	X	6.42	X	X	0	X	X	X
Stack	0	X	X	X	X	0	X	X
LinkedList	3.2	118.2	X	218.61	0	0	X	1.72

Discussions

It is clear from the results some collections are more suitable to a particular operation than others. For instance, linked lists perform significantly better than lists when it comes to inserting a value at the beginning of the collection. This is because a linked list merely adds a new node at the beginning, which requires the creation of a new LinkedListNode object. Meanwhile, lists have to shift all items by one index to the right to make room for the new value, which significantly impacts performance. Removing values from the beginning of a list are also slow for the same reason.

Lists and queues are better at adding values to the end of the collection. This is because in most cases, this operation only requires the value at the end position of the collection to change. In other cases, however, the underlying data structure needs to grow to accommodate the new value, which requires copying the entire collection to a bigger block of memory. Lists are more performant than linked lists with regard to removing a specific value, possibly due to the necessary relinking of list nodes which incurs some extra time.

Adding and removing at an index of a list can take some time, as it requires some or all elements in the list to be shifted left or right. The experiment randomly chooses valid indexes to add/remove items, so the results may not accurately reflect real world usage. However, one should expect adding/removing near the beginning of a list to be slower than near the end of it. The find operation performs relatively uniformly across collection classes. This is probably because it merely iterates through the collection, comparing each value to the given target value or predicate, which is not too different among collections.

It should be noted that the purpose of this experiment is only to measure the performance difference of the same operation across multiple collection classes, not different operations on the same class. Additionally, the choice of collection classes should depend on the overall use case. For instance, dictionaries may have slower insertion than lists but offers instant retrieval.

Conclusion

This paper examined six collection classes offered by the .NET framework which are arrays, lists, dictionaries, stacks, queues, and linked lists. They were compared in terms of time performance in different operations such as inserting, deleting, and finding. To measure the time performance of each operation per collection, test cases were written and run on the collection classes that support them. The average execution time of each operation on a particular collection class is captured in table 3. The results show that the same operation performs differently depending on the collection class, which makes each collection more suitable for a particular use case. This paper only compares the same operation across

different collection classes, and not different operations on the same class. More research is needed to measure the latter. The paper hopes to serve as a useful reference when programmers are considering the right collection choice for their application.

Reference

dotnet-bot. (n.d.). *Array Class (System)*. Retrieved July 31, 2023, from https://learn.microsoft.com/en-us/dotnet/api/system.array?view=net-7.0

dotnet-bot. (n.d.). *Stopwatch Class (System.Diagnostics)*. Retrieved July 31, 2023, from https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-7.0

dotnet-bot. (n.d.). *System.Collections.Generic Namespace*. Retrieved July 31, 2023, from https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-7.0

IEvangelist. (2022, August 12). *Collections and Data Structures*. https://learn.microsoft.com/en-us/dotnet/standard/collections/