Which Data Structure to use?

The first thing to do is look at the fundamental requirements - what are the inputs, what are the outputs, what are the operations in between?

Then, the performance and scale requirements - how much data are we talking about? How frequently will it be accessed?

Then, ponder how those requirements align with the simple classic data structures that we all know and love - lists, arrays, hash sets, dictionaries, queues, trees, heaps, and so on. Usually there's a good match; if there is only a near-match, you can usually wrap one of those simple structures to give it the API that you really need (like turning a list into a stack or queue).

Sometimes, there's no good fit with the usual structures. Then you have to consider stuff that isn't used often, but that works wonderfully for some special cases. Bloom filters are a great example. Or stuff that's "bigger" than the basic structures, like relational databases or the various NoSQL databases.

The last two steps depend on having a mental catalog of structures to consider. That's something that comes from experience. The more code you write, the more familiar you will be with various data structures, algorithms, patterns, and so on. Reading books also helps, as does reading code, getting feedback on your code, reading other peoples' feedback on other peoples' code, and so on. It all takes time. But you can pick up a lot of that stuff without even trying - just by writing software on your own, or better yet as part of a team.

Study all the important common data structures, including their supposed strengths and weaknesses. Understand the public interfaces, build your own implementations, then study versions from others. After that, dream up different scenarios, and write lots of code using them. Form opinions about which you prefer in those situations.

This should give you a good sense for the ‘right’ data structure for some new task.

Based on time and patience, your study list of “important common data structures” could include these (and perhaps much more):

* Arrays / Vectors
* Queues / Stacks
* Singly Linked Lists / Doubly Linked Lists
* Binary Trees / Binary Search Trees
* Priority Queues / Binary Heaps
* Hash Sets / Hash Maps
* Graphs

A **HashSet**, similar to a **Dictionary**, is a hash-based collection, so look ups are very fast with O(1). But unlike a **dictionary**, it doesn't store key/value pairs; it only stores values. So, every objects should be unique

[**5 C# Collections that Every C# Developer Must Know**](https://programmingwithmosh.com/net/csharp-collections/)

Finding the right collection in .NET is like finding the right camera in a camera shop! There are so many options to choose from, and each is strong in certain scenarios and weak in others. If looking for a collection in .NET has left you confused, you’re not alone.

 These are the collections that you’ll use 80 – 90% of the time, if not more.

* List
* Dictionary
* HashSet
* Stack
* Queue

**List<T>**

Represents a list of objects that can be accessed by an index. <T> here means this is a generic list.

Unlike arrays that are fixed in size, lists can grow in size dynamically. That’s why they’re also called *dynamic arrays* or *vectors*. Internally, a list uses an array for storage. If it becomes full, it’ll create a new larger array, and will copy items from the existing array into the new one.

These days, it’s common to use lists instead of arrays, even if you’re working with a fixed set of items.

To create a list:

|  |  |
| --- | --- |
| 1 | var list = new List<int>(); |

If you plan to store large number of objects in a list, you can reduce the cost of reallocations of the internal array by setting an initial size:

|  |  |
| --- | --- |
| 1  2 | // Creating a list with an initial size  var list = new List<int>(10000); |

Here are some useful operations with lists:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // Add an item at the end of the list  list.Add(4);    // Add an item at index 0  list.Insert(4, 0);    // Remove an item from list  list.Remove(1);    // Remove the item at index 0  list.RemoveAt(0);    // Return the item at index 0  var first = list[0];    // Return the index of an item  var index = list.IndexOf(4);    // Check to see if the list contains an item  var contains = list.Contains(4);    // Return the number of items in the list  var count = list.Count;    // Iterate over all objects in a list  foreach (var item in list)      Console.WriteLine(item); |

Now, let’s see where a list performs well and where it doesn’t.

**Adding/Removing Items at the Beginning or Middle**

If you add/remove an item at the beginning or middle of a list, it needs to shift one or more items in its internal array. In the worst case scenario, if you add/remove an item at the very beginning of a list, it needs to shift all existing items. The larger the list, the more costly this operation is going to be. We specify the cost of this operation using [Big O notation](https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/): **O(n)**, which simply means the cost increases linearly in direct proportion to the size of the input. So, as n grows, the execution time of the algorithm increases in direct proportion to n.

**Adding/Removing Items at the End**

Adding/removing an item at the end of a list is a relatively fast operation and does not depend on the size of the list. The existing items do not have to be shifted. This is why the cost of this operation is relatively constant and is not dependent on the number of items in the list. We represent the execution cost of this operation with Big O notation: **O(1)**. So, 1 here means constant.

**Searching for an Item**

When using methods that involve searching for an item(e.g. **IndexOf**, **Contains** and **Find**), List performs a linear search. This means, it iterates over all items in its internal array and if it finds a match, it returns it. In the worst case scenario, if this item is at the end of the list, all items in the list need to be scanned before finding the match. Again, this is another example of **O(n)**, where the cost of finding a match is linear and in direct proportion with the number of elements in the list.

**Accessing an Item by an Index**

This is what lists are good at. You can use an index to get an item in a list and no matter how big the list is, the cost of accessing an item by index remains relatively constant, hence **O(1)**.

**List in a Nutshell**

So, adding/removing items at the end of a list and accessing items by index are fast and efficient operations with O(1). Searching for an item in a list involves a linear search and in the worst case scenario is O(n). If you need to search for items based on some criteria, and not an index (e.g. customer with ID 1234), you may better use a Dictionary.

**Dictionary<TKey, TValue>**

Dictionary is a collection type that is useful when you need fast lookups by keys. For example, imagine you have a list of customers and as part of a task, you need to quickly look up a customer by their ID (or some other unique identifier, which we call *key*). With a list, looking up a customer involves a linear search and the cost of this operation, as you learned earlier, is O(n) in the worst case scenario. With a dictionary, however, look ups are very fast with O(1), which means no matter how large the dictionary is, the look up time remans relatively constant.

When storing or retrieving an object in a dictionary, you need to supply a key. The key is a value that uniquely identifies an object and cannot be null. For example, to store a Customer in a Dictionary, you can use CustomerID as the key.

To create a dictionary, first you need to specify the type of keys and values:

|  |  |
| --- | --- |
| 1 | var dictionary = new Dictionary<int, Customer>(); |

Here, our dictionary uses **int**keys and **Customer**values. So, you can store a Customer object in this dictionary as follows:

|  |  |
| --- | --- |
| 1 | dictionary.Add(customer.Id, customer); |

You can also add objects to a dictionary during initialization:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var dictionary = new Dictionary<int, Customer>  {       { customer1.Id, customer1 },       { customer2.Id, customer2 }  } |

Later, you can look up customers by their IDs very quickly:

|  |  |
| --- | --- |
| 1  2 | // Return the customer with ID 1234  var customer = dictionary[1234]; |

You can remove an object by its key or remove all objects using the **Clear**method:

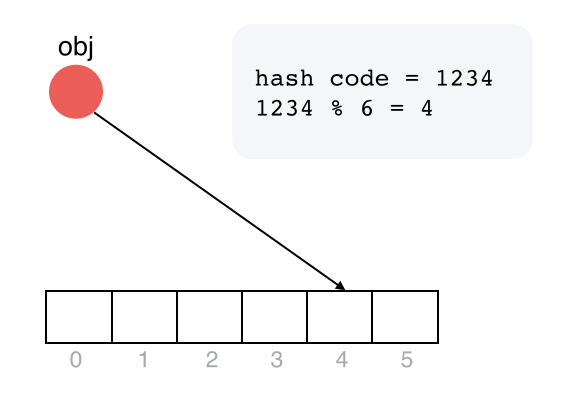
|  |  |
| --- | --- |
| 1  2  3  4  5 | // Removing an object by its key  dictionary.Remove(1);    // Removing all objects  dictionary.Clear(); |

And here are some other useful methods available in the Dictionary class:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20 | var count = dictionary.Count;    var containsKey = dictionary.ContainsKey(1);    var containsValue = dictionary.ContainsValue(customer1);    // Iterate over keys  foreach (var key in dictionary.Keys)       Console.WriteLine(dictionary[key]);    // Iterate over values  foreach (var value in dictionary.Values)       Console.WriteLine(value);    // Iterate over dictionary  foreach (var keyValuePair in dictionary)  {       Console.WriteLine(keyValuePair.Key);       Console.WriteLine(keyValuePair.Value);  } |

So, why are dictionary look ups so fast? A dictionary internally stores objects in an array, but unlike a list, where objects are added at the end of the array (or at the provided index), the index is calculated using a hash function. So, when we store an object in a dictionary, it’ll call the **GetHashCode** method on the key of the object to calculate the hash. The hash is then adjusted to the size of the array to calculate the index into the array to store the object. Later, when we lookup an object by its key, **GetHashCode** method is used again to calculate the hash and the index. As you learned earlier, looking up an object by index in an array is a fast operation with O(1). So, unlike lists, looking up an object in a dictionary does not require  scanning every object and no matter how large the dictionary is, it’ll remain extremely fast.

So, in the following figure, when we store this object in a dictionary, the **GetHashCode** method on the key is called. Let’s assume it returns 1234. This hash value is then adjusted based on the size of the internal array. In this figure, length of the internal array is 6. So, the remainder of the division of 1234 by 6 is used to calculate the index (in this case 4). Later, when we need to look up this object, its key used again to calculate the index.

[](https://i0.wp.com/programmingwithmosh.com/wp-content/uploads/2015/10/Screenshot-2015-10-19-23.23.25.png?ssl=1)

Now, this was a simplified explanation of how hashing works. There is more involved in calculation of hashes, but you don’t really need to know the exact details at this stage (unless for personal interests). All you need to know as a C# developer is that dictionaries are *hash-based collections* and for that reason lookups are very fast.

**HashSet<T>**

A HashSet represents a set of unique items, just like a mathematical set (e.g. **{ 1, 2, 3 }**). A set cannot contain duplicates and the order of items is not relevant. So, both **{ 1, 2, 3 }** and **{ 3, 2, 1 }** are equal.

Use a HashSet when you need super fast lookups against a unique list of items. For example, you might be processing a list of orders, and for each order, you need to quickly check the supplier code from a list of valid supplier codes.

A HashSet, similar to a Dictionary, is a hash-based collection, so look ups are very fast with **O(1)**. But unlike a dictionary, it doesn’t store key/value pairs; it only stores values. So, every objects should be unique and this is determined by the value returned from the **GetHashCode** method. So, if you’re going to store custom types in a set, you need to override **GetHashCode**and **Equals**methods in your type.

To create a HashSet:

|  |  |
| --- | --- |
| 1 | var hashSet = new HashSet<int>(); |

You can add/remove objects to a HashSet similar to a List:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17 | // Initialize the set using object initialization syntax  var hashSet = new HashSet<int>() { 1, 2, 3 };    // Add an object to the set  hashSet.Add(4);    // Remove an object  hashSet.Remove(3);    // Remove all objects  hashSet.Clear();    // Check to see if the set contains an object  var contains = hashSet.Contains(1);    // Return the number of objects in the set  var count = hashSet.Count; |

HashSet provides many mathematical set operations:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | // Modify the set to include only the objects present in the set and the other set  hashSet.IntersectWith(another);    // Remove all objects in "another" set from "hashSet"  hashSet.ExceptWith(another);    // Modify the set to include all objects included in itself, in "another" set, or both  hashSet.UnionWith(another);    var isSupersetOf = hashSet.IsSupersetOf(another);  var isSubsetOf = hashSet.IsSubsetOf(another);  var equals = hashSet.SetEquals(another); |

**Stack<T>**

Stack is a collection type with Last-In-First-Out (LIFO) behaviour. We often use stacks in scenarios where we need to provide the user with a way to go back. Think of your browser. As you navigate to different web sites, these addresses that you visit are *pushed* on a stack. Then, when you click the back button, the item on the stack (which represents the current address in the browser) is *popped* and now we can get the last address you visited from the item on the stack. The undo feature in applications is implemented using a stack as well.

Here is how you can use a Stack in C#:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | var stack = new Stack<string>();    // Push items in a stack  stack.Push("[http://www.google.com](http://www.google.com/)");    // Check to see if the stack contains a given item  var contains = stack.Contains("[http://www.google.com](http://www.google.com/)");    // Remove and return the item on the top of the stack  var top = stack.Pop();    // Return the item on the top of the stack without removing it  var top = stack.Peek();    // Get the number of items in stack  var count = stack.Count;    // Remove all items from stack  stack.Clear(); |

Internally, a stack is implemented using an array. Since arrays in C# have a fixed size, as you push items into a stack, it may need to increase its capacity by re-allocating a larger array and copying existing items into the new array. If re-allocation doesn’t need to happen, push is O(1) operation; otherwise, if re-allocation is required, assuming the stack has n elements, all these elements need to be copied to the new array. This leads to runtime complexity of O(n).

**Pop** is an O(1) operation.

**Contains** is a linear search operation with O(n).

**Queue<T>**

Queue represents a collection with First-In-First-Out (FIFO) behaviour. We use queues in situations where we need to process items as they arrive.

Three main operations on queue include:

* Enqueue: adding an element to the end of a queue
* Dequeue: removing the element at the front of the queue
* Peek: inspecting the element at the front without removing it.

Here is how you can use a queue:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | var queue = new Queue<string>();    // Add an item to the queue  queue.Enqueue("transaction1");    // Check to see if the queue contains a given item  var contains = queue.Contains("transaction1");    // Remove and return the item on the front of the queue  var front = queue.Dequeue();    // Return the item on the front without removing it  var top = queue.Peek();    // Remove all items from queue  queue.Clear();    // Get the number of items in the queue  var count = queue.Count; |

**Summary**

Lists are fast when you need to access an element by index, but searching for an item in a list is slow since it requires a linear search.

Dictionaries provide fast lookups by key. Keys should be unique and cannot be null.

HashSets are useful when you need fast lookups to see if an element exists in a set or not.

Stacks provide LIFO (Last-In-First-Out) behaviour and are useful when you need to provide the user with a way to go back.

Queues provide FIFO (First-In-First-Out) behaviour and are useful to process items in the order arrived.