Big-O

https://www.raywenderlich.com/123100/collection-data-structures-swift-2

The most commonly seen Big-O performance measures are as follows, in order from best to worst performance:

O(1) – (constant time)
No matter how many items are in a data structure, this function calls the same number of operations. This is considered ideal performance.

O(log n) – (logarithmic)
The number of operations this function calls grows at the rate of the logarithm of the number of items in the data structure.
This is good performance, since it grows considerably slower than the number of items in the data structure.

O(n) – (linear)
The number of operations this function calls will grow linearly with the size of the structure.
This is considered decent performance, but it can grind along with larger data collections.

O(n (log n)) – (“linearithmic”)
The number of operations called by this function grows by the logarithm of the number of items in the structure multiplied by the number of items in the structure.
Predictably, this is about the lowest level of real-world tolerance for performance.
While larger data structures perform more operations, the increase is somewhat reasonable for data structures with small numbers of items.

O(n^2) – (quadratic)
The number of operations called by this function grows at a rate that equals the size of the data structure, squared – poor performance at best.
It grows quickly enough to become unusably slow even if you’re working with small data structures.

O(2^n) – (exponential)
The number of operations called by this function grows by two to the power of the size of the data structure.
The resulting very poor performance becomes intolerably slow almost immediately.

O(n!) (factorial)

The number of operations called by this function grows by the factorial of the size of the data structure.
Essentially, you have the worst case scenario for performance.
For example, in a structure with just 100 items, the multiplier of the number of operations is 158 digits long.
Witness it for yourself on wolframalpha.com.

https://koenig-media.raywenderlich.com/uploads/2014/09/Screen-Shot-2014-09-14-at-11.54.21-AM.png

Did you notice that you can't even see the green O(log n) line because it is so close to the ideal O(1) at this scale?
That's pretty good!
On the other hand, operations that have Big-O notations of O(n!) and O(2^n) degrade so quickly that by the time you have more than 10 items in a collection,
the number of operations spikes completely off the chart.
Yikes! As the chart clearly demonstrates, the more data you handle, the more important it is to choose the right structure for the job.
Now that you've seen how to compare the performance of operations on data structures,
it's time to review the three most common types used in iOS and explore how they perform in theory and in practice.

Lesser-known Foundation Data Structures

- NSCache
Using NSCache is very similar to using NSMutableDictionary – you just add and retrieve objects by key.
The difference is that NSCache is designed for temporary storage for things that you can always recalculate or regenerate.
If available memory gets low, NSCache might remove some objects. They are thread-safe, but Apple's documentation warns:

- NSCountedSet
NSCountedSet tracks how many times you've added an object to a mutable set. It inherits from NSMutableSet, so if you try to add the same object again it will only be reflected once in the set.
However, an NSCountedSet tracks how many times an object has been added. You can see how many times an object was added with countForObject().
Note that when you call count on an NSCountedSet, it only returns the count of unique objects, not the number of times all objects were added to the set.

- NSOrderedSet
An NSOrderedSet along with its mutable counterpart, NSMutableOrderedSet,
is a data structure that allows you to store a group of distinct objects in a specific order.
"Specific order" -- gee, that sounds an awful lot like an array, doesn't it?
Apple succinctly sums up why you'd want to use an NSOrderedSet instead of an array (emphasis mine):

You can use ordered sets as an alternative to arrays when element order matters and performance
while testing whether an object is contained in the set is a consideration
-- testing for membership of an array is slower than testing for membership of a set.
Because of this, the ideal time to use an NSOrderedSet is when you need to store an ordered collection of objects that cannot contain duplicates.
Note: While NSCountedSet inherits from NSMutableSet, NSOrderedSet inherits from NSObject.
This is a great example of how Apple names classes based on what they do, but not necessarily how they work under the hood.

- NSHashTable and NSMapTable
NSHashTable is another data structure that is similar to Set, but with a few key differences from NSMutableSet.
You can set up an NSHashTable using any arbitrary pointers and not just objects, so you can add structures and other non-object items to an NSHashTable.
You can also set memory management and equality comparison terms explicitly using NSHashTableOptions enum.
NSMapTable is a dictionary-like data structure, but with similar behaviors to NSHashTable when it comes to memory management.
Like an NSCache, an NSMapTable can hold weak references to keys. However, i
t can also remove the object related to that key automatically whenever the key is deallocated.
These options can be set from the NSMapTableOptions enum.

- NSIndexSet
An NSIndexSet is an immutable collection of unique unsigned integers intended to represent indexes of an array.
If you have an NSArray of ten items where you regularly need to access items' specific positions,
you can store an NSIndexSet and use NSArray's objectsAtIndexes: to pull those objects directly:

An NSIndexSet retains the behavior of NSSet that only allows a value

to appear once.
Hence, you shouldn't use it to store an arbitrary list of integers unless only a single appearance is a requirement.
With an NSIndexSet, you're storing indexes as sorted ranges, so it is more efficient than storing an array of integers.