**C# Framework**

[Big O Notation 2](#_Toc374451817)

[Data Structures 8](#_Toc374451818)

[1. Array (Non Generic Collection Types) 12](#_Toc374451819)

[Array.Sort 13](#_Toc374451820)

[2. ArrayList (Non Generic Collection Types) 14](#_Toc374451821)

[3. Hastable 17](#_Toc374451822)

[Hashing Overview 18](#_Toc374451823)

[Hashing Functions 18](#_Toc374451824)

[Hash Table Internal DataStructure 19](#_Toc374451825)

[The Hashtable Class's Hash Function 19](#_Toc374451826)

[Collisions 21](#_Toc374451827)

[Growing the Collection 23](#_Toc374451828)

[Disadvantage of HashCode 23](#_Toc374451829)

[HashTable <<<>>> Dictionary 24](#_Toc374451830)

[Additional Comments on Hastable (from code) 25](#_Toc374451831)

[4. Stacks 26](#_Toc374451832)

[5. Queue 30](#_Toc374451833)

[6. Linked List 31](#_Toc374451834)

[Definition of Linked Lists 31](#_Toc374451835)

[Structure of .Net Linked List 31](#_Toc374451836)

[Structure of Custom Linked List 31](#_Toc374451837)

[Doubly Linked List 32](#_Toc374451838)

[Operations with Linked List 32](#_Toc374451839)

[7. Binary Tree 33](#_Toc374451840)

[Operations within Binary Search Tree 34](#_Toc374451841)

[Insert Into BST 34](#_Toc374451842)

[Traversal of BST 34](#_Toc374451843)

[Finding a Node (Searching) and Min/Max Values 35](#_Toc374451844)

[Removing a Node From a BST 35](#_Toc374451845)

[Finding the Depth of BST 35](#_Toc374451846)

[Comparison of Data Structures 36](#_Toc374451847)

[Basic Sorting Algorithms 38](#_Toc374451848)

[1. Bubble Sort 39](#_Toc374451849)

[2. Insertion Sort 41](#_Toc374451850)

[3. Selection Sort 43](#_Toc374451851)

[4. Merge Sort 45](#_Toc374451852)

[Quick Sort 47](#_Toc374451853)

[Basic Searching Algorithms 49](#_Toc374451854)

[1. Sequential Searching 49](#_Toc374451855)

[Searching for Maximum and Minimum Values 49](#_Toc374451856)

[2. Binary Searching 51](#_Toc374451857)

[Recursive Binary Search Algorithm 51](#_Toc374451858)

[Array Characteristics 52](#_Toc374451859)

[Strings, String Class and String Builder 53](#_Toc374451860)

[Frequently Used String Class Methods 54](#_Toc374451861)

[Methods for Comparing Strings 54](#_Toc374451862)

[Methods for Manipulating Strings 56](#_Toc374451863)

[StringBuilder Class 57](#_Toc374451864)

[Constructing StringBuilder Objects 57](#_Toc374451865)

[Modifying StringBuilder Objects 57](#_Toc374451866)

[Append Method 57](#_Toc374451867)

[Insert Method 58](#_Toc374451868)

[Remove Method 58](#_Toc374451869)

[Replace Method 59](#_Toc374451870)

[Convert to String Class 59](#_Toc374451871)

[Efficiency of the String Class to StringBuilder: 59](#_Toc374451872)

[Questions 60](#_Toc374451873)

[Data Structure - Collections 60](#_Toc374451874)

[Sorting and Searching Questions 61](#_Toc374451875)

[String Concepts 62](#_Toc374451876)

Big O Notation

|  |  |
| --- | --- |
| **For Data Structures** | |
| Array |  |
| ArrayList |  |
| HashTable |  |
| Stack | Internally uses Array – so similar as above  While Insertion - If Count is less than the capacity of the stack, Push is an O(1) operation. If the capacity needs to be increased to accommodate the new element, Push becomes an O(n) operation, where n is Count. Pop is an O(1) operation. |
| Queue |  |
| Singly LinkedList |  |
| Doubly Linked List |  |
| Binary Search Trees |  |

|  |  |  |
| --- | --- | --- |
| **Sorting Options** | | |
| 1 | Bubble |  |
| 2 | Insertion |  |
| 3 | Selection |  |
| 4 | Merge |  |
| 5 | Quick |  |

|  |  |
| --- | --- |
| **Sorting Mechanisms used in Collections** | |
| **Array.Sort** | Uses QuickSort |
| **List<T>.Sort** | This method uses Array.Sort internally, which uses the QuickSort algorithm. This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal. |
|  |  |

<http://en.wikipedia.org/wiki/Sorting_algorithm>  
<http://bigocheatsheet.com/>

|  |  |  |
| --- | --- | --- |
| **Searching Options** | | |
| 1 | Sequential Searching |  |
| 2 | Binary Search |  |
| 3 | Depth First Search |  |
| 4 | Breadth First Search |  |

**Collections**

- Have defined set of properties that describe them and operations that can be performed on them.  
- Example of Collection property  
 Count (which holds the number of items in collection)  
- Collection operations called methods include   
 Add (for adding a new element to a collection)  
 Insert (for adding a new element to a collection at a specified index)  
 Remove (for removing a specified element from a collection)  
 Clear (for removing all the elements from a collection)  
 Contains (for determining if a specified element is a member of a collection)  
 IndexOf (for determining the index of a specified element in a collection).

Non-Linear

- Holds elements that do not have positional order within collection  
- Examples Real World   
 An Org Chart  
 Rack of billiard balls

**Linear**

- List of elements where 1 element follows the previous element  
- Elements are normally ordered by position  
- Examples Real World   
 Grocery list

Hierarchical Access

Grouped Access

Sequential Access

Direct Access

Tree

Generalized Index Collection

ArrayList

Array

Struct

String

Graphs

Networks

Sets

Heap

Hashtable

Binary Tree

Stacks

ArrayList

Queue

Collection Types

Nongeneric Collection Types

ArrayList

Hash Tables

BitArray

Queue

Stack

Array

Generic Collection Types

List<T>

LinkedList<T>

Dictionary<TKey, TValue>

SortedDictionary<Tkey, TValue>   
&   
SortedList<Tkey, TValue>

HashSet<T>

SortedSet<T>

Queue<T>  
&  
Stack<T>

Data Structures

The data structures described here are found in the .NET Framework class library System.Collections. We develop the concept of a collection by first discussing the implementation of our own Collection class (using the array as the basis of our implementation) and then by covering the Collection classes in the .NET Framework.

An important addition to C# 2.0 is generics. Generics allow the C# programmer to write one version of a function, either independently or within a class, without having to overload the function many times to allow for different data types. C# 2.0 provides a special library, System.Collections.Generic, that implements generics for several of the System.Collections data structures.

**Collections Defined**

A collection is a structured data type that stores data and provides operations for adding data to the collection, removing data from the collection, updating data in the collection, as well as operations for setting and returning the values of different attributes of the collection.

**Direct Access Collection**

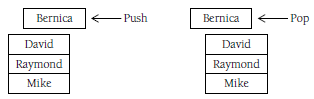
* Array: In C# arrays are not only built-in data type, they are also a class.
* String:
* A string is a collection of characters that can be accessed based on their index, in the same manner we access the elements of an array. Strings are also implemented as class objects in C#.
* The class includes a large set of methods for performing standard operations on strings, such as concatenation, returning substrings, inserting characters, removing characters, and so forth
* C# strings are immutable, meaning once a string is initialized it cannot be changed. When you modify a string, a copy of the string is created instead of changing the original string. This behavior can lead to performance degradation in some cases, so the .NET Framework provides a StringBuilder class that enables you to work with mutable strings
* Struct:
* A struct is a composite data type that holds data that may consist of many different data types. For example, an employee record consists of employee’ name (a string), salary (an integer), identification number (a string, or an integer), as well as other attributes. Since storing each of these data values in separate variables could become confusing very easily, the language provides the struct for storing data of this type.
* A powerful addition to the C# struct is the ability to define methods for performing operations stored on the data in a struct. This makes a struct somewhat like a class, though you can’t inherit or derive a new type from a structure.

**Sequential Access Collection**

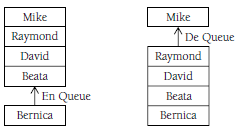
* A sequential access collection is a list that stores its elements in sequential order.
* We call this type of collection a linear list. Linear lists are not limited by size when they are created, meaning they are able to expand and contract dynamically.
* Items in a linear list are not accessed directly; they are referenced by their position



* Linear lists can be either ordered or unordered. The order of a list makes a big difference when performing searches on the data on the list.
* Some types of linear lists restrict access to their data elements. Examples of these types of lists are stacks and queues.
* Stack:
* A stack is a list where access is restricted to the beginning (or top) of the list.
* Items are placed on the list at the top and can only be removed from the top. Hence they are known as Last-in, First-out structures.
* When we add an item to a stack, we call the operation a push. When we remove an item from a stack, we call that operation a pop.



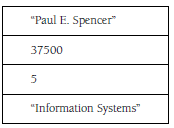
* The stack is a very common data structure, especially in computer systems programming. Stacks are used for arithmetic expression evaluation and for balancing symbols, among its many applications.
* Queue:
* A queue is a list where items are added at the rear of the list and removed from the front of the list.
* This type of list is known as a First-in, First-out structure.
* Adding an item to a queue is called an EnQueue, and removing an item from a queue is called a Dequeue.



* Queues are used in both systems programming, for scheduling operating system tasks, and for simulation studies. Queues make excellent structures for simulating waiting lines in every conceivable retail situation.

**Generalized Index Collections**

* Hashtable:
* This stores a set of data values associated with a key.
* In a hash table, a special function, called a hash function, takes one data value and transforms the value (called the key) into an integer index that is used to retrieve the data. The index is then used to access the data record associated with the key.
* C# has a class, called HashTable, for storing data in a hash table.



* Dictionary:
* A dictionary is made up of a series of key–value pairs, called associations.
* The key is an index into the value associated with the key.

**Hierarchical Access Collection**

A hierarchical collection is a group of items divided into levels. An item at one level can have successor items located at the next lower level. One common hierarchical collection is the tree.

* Tree:
* A tree collection looks like an upside-down tree, with one data element as the root and the other data values hanging below the root as leaves.
* The elements of a tree are called nodes, and the elements that are below a particular node are called the node’s children.

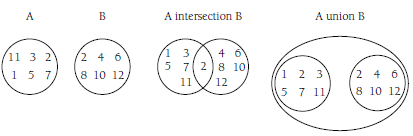


* Trees have applications in several different areas. The file systems of most modern operating systems are designed as a tree collection, with one directory as the root and other subdirectories as children of the root.
* A binary tree is a special type of tree collection where each node has no more than two children. A binary tree can become a binary search tree, making searches for large amounts of data much more efficient. This is accomplished by placing nodes in such a way that the path from the root to a node where the data is stored is along the shortest path possible.
* Yet another tree type, the heap, is organized so that the smallest data value is always placed in the root node. The root node is removed during a deletion, and insertions into and deletions from a heap always cause the heap to reorganize so that the smallest value is placed in the root. Heaps are often used for sorts, called a heap sort. Data elements stored in a heap can be kept sorted by repeatedly deleting the root node and reorganizing the heap.

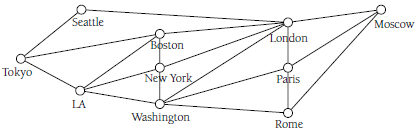
**Grouped Access Collection**

A nonlinear collection of items that are unordered is called a group. The three major categories of group collections are sets, graphs, and networks.

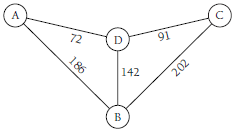
* Set:
* A set is a collection of unordered data values where each value is unique. The list of students in a class is an example of a set, as is, of course, the integers. Operations that can be performed on sets include union and intersection.



* Graph:
* A graph is a set of nodes and a set of edges that connect the nodes.
* Graphs are used to model situations where each of the nodes in a graph must be visited, sometimes in a particular order, and the goal is to find the most efficient way to “traverse” the graph. Graphs are used in logistics and job scheduling and are well studied by computer scientists and mathematicians. You may have heard of the “Traveling Salesman” problem. This is a particular type of graph problem that involves determining which cities on a salesman’s route should be traveled in order to most efficiently complete the route within the budget allowed for travel.



A network is a special type of graph where each of the edges is assigned a weight. The weight is associated with a cost for using that edge to move from one node to another. Figure below depicts a network of cities where the weights are the miles between the cities (nodes).



**Collection Base Class**

The .NET Framework library does not include a generic Collection class for storing data, but there is an abstract class you can use to build your own Collection class—CollectionBase. The CollectionBase class provides the programmer with the ability to implement a custom Collection class. The class implicitly implements two interfaces necessary for building a Collection class, ICollection and IEnumerable, leaving the programmer with having to implement just those methods that are typically part of a Collection class.

**.Net Built-In Data Structures**

Data structures are classes that are used to organize data and provide various operations upon their data. Probably the most common and well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index.

1. Array (Non Generic Collection Types)

NameSpace: System.Array

Everyone's Favorite Linear, Direct Access, Homogeneous Data Structure

* The contents of an array are stored in contiguous memory.
* All of the elements of an array must be of the same type or of a derived type; hence arrays are referred to as homogeneous data structures.
* Array elements can be directly accessed. With arrays if you know you want to access the “i” element, you can simply use one line of code: arrayName[i].
* Using an array in C# involves creating an array object of System.Array type, the abstract base type for all arrays.
* The data in an array can be of either a built-in type or a user-defined type.

arrayName = new arrayType[allocationSize];

This allocates a contiguous block of memory in the CLR-managed heap large enough to hold the allocationSize number of arrayTypes.

* If arrayType is a value type, then allocationSize number of unboxed arrayType values are created.
* If arrayType is a reference type, then allocationSize number of arrayType references are created.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Another example:  myPersons[0] = new Person { FirstName="Ayrton", LastName="Senna" };  myPersons[1] = new Person { FirstName="Michael", LastName="Schumacher" }; |  |

* The thing to keep in mind is that the ten elements in the files array are references to FileInfo instances. The above figure hammers home this point, showing the memory layout if we assign some of the values in the files array to FileInfo instances.
* In managed code, array lookups are a slight bit more involved than this because with each array access the CLR checks to ensure that the index being requested is within the array's bounds. If the array index specified is out of bounds, an IndexOutOfRangeException is thrown. This check help ensures that when stepping through an array we do not accidentally step past the last array index and into some other memory.

Methods and Properties for Retrieving Array Metadata

The Array class provides several properties for retrieving metadata about an array:

* Length: Returns the total number of elements in all dimensions of an array.
* GetLength: Returns the number of elements in specified dimension of an array.
* Rank: Returns the number of dimensions of an array.
* GetType: Returns the Type of the current array instance. The GetType method returns not only the type of the array, but also lets us know that the object is indeed an array: System.Int32[]

Advantage of using Array

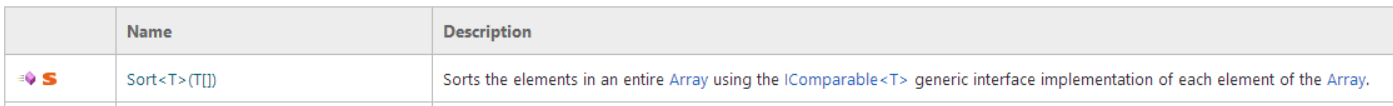
* Arrays are excellent data structures to use when storing a collection of homogeneous types that you only need to access directly.

Disadvantage of using Array

* Searching an unsorted array has linear running time. While this is acceptable when working with small arrays, or when performing very few searches, if your application is storing large arrays that are searched frequently, there are a number of other data structures better suited for the job
* Arrays, while simple to use, can quickly become a nuisance if you find yourself needing to regularly resize the array, or don't know how many elements you'll need when initializing the array.

Array.Sort

Internally it uses the quick sort algorithm



|  |  |
| --- | --- |
|  |  |

**Multi-Dimensional Arrays**

So far we have limited our discussion to arrays that have just a single dimension. In C#, an array can have up to 32 dimensions, though arrays with more than three dimensions are very rare (and very confusing). Multidimensional arrays are declared by providing the upper bound of each of the dimensions of the array. The two-dimensional declaration: int[,] grades = new int[4,5];

This declares an array that consists of 4 rows and 5 columns. Two-dimensional arrays are often used to model matrices.

**Jagged Arrays**

A jagged array is an array of arrays where each row of an array is made up of an array. Each dimension of a jagged array is a one dimensional array. We call it a “jagged” array because the number of elements in each row may be different. A picture of a jagged array would not be square or rectangular, but would have uneven or jagged edges.

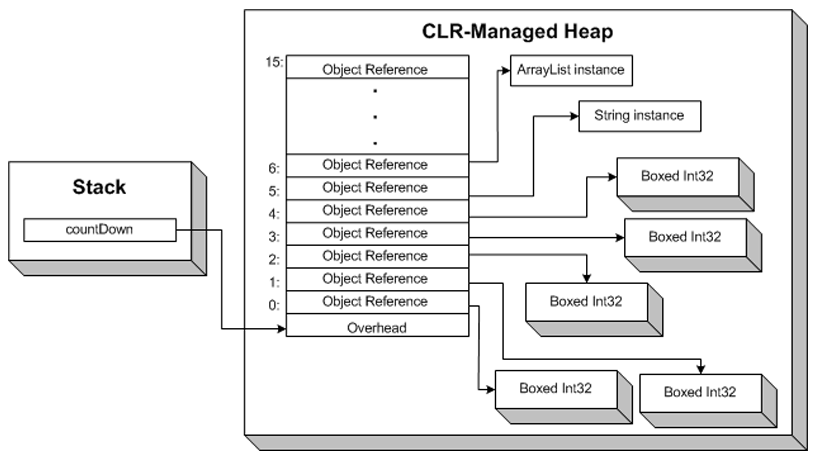
A jagged array is declared by putting two sets of parentheses after the array variable name. The first set of parentheses indicates the number of rows in the array. The second set of parentheses is left blank. This marks the place for the one-dimensional array that is stored in each row.  
int[][] jagged = new int[12][];  
This statement looks strange, but makes sense when you break it down. jagged is an Integer array of 12 elements, where each of the elements is also an Integer array.

2. ArrayList (Non Generic Collection Types)

NameSpace: System.Collections.ArrayList

An interesting alternative to using arrays in C# is the ArrayList class. The .NET Framework already contains a data structure that provides the below functionality. The ArrayList maintains an internal object array and provides automatic resizing of the array as the number of elements added to the ArrayList grows. Because the ArrayList uses an object array, developers can add any type—strings, integers, FileInfo objects, Form instances, anything.

An arraylist is an array that grows dynamically as more space is needed. For situations where you can’t accurately determine the ultimate size of an array, or where the size of the array will change quite a bit over the lifetime of a program, an arraylist may be a better choice than an array.



*Note: The same memory layout occurs for reference types in both ArrayLists and arrays.*

Inner Working:

An ArrayList object has a Capacity property that stores its size. **The initial value of the property is 16**. When the number of elements in an ArrayList reaches this limit, the Capacity property adds another 16 elements to the storage space of the ArrayList. Using an ArrayList in a situation where the number of elements in an array can grow larger, or smaller, can be more efficient than using ReDim Preserver with a standard array.

Members of the ArrayList Class

The ArrayList class includes several methods and properties for working with ArrayLists. Here is a list of some of the most commonly used methods and properties:

* Add(): Adds an element to the ArrayList.
* AddRange(): Adds the elements of a collection to the end of the ArrayList.
* Capacity: Stores the number of elements the ArrayList can hold.
* Clear(): Removes all elements from the ArrayList.
* Contains(): Determines if a specified item is in the ArrayList.
* CopyTo(): Copies the ArrayList or a segment of it to an array.
* Count: Returns the number of elements currently in the ArrayList.
* GetEnumerator(): Returns an enumerator to iterate over the ArrayList.
* GetRange(): Returns a subset of the ArrayList as an ArrayList.
* IndexOf(): Returns the index of the first occurrence of the specified item.
* Insert(): Insert an element into the ArrayList at a specified index.
* InsertRange(): Inserts the elements of a collection into the ArrayList starting at the specified index.
* Item(): Gets or sets an element at the specified index.
* Remove(): Removes the first occurrence of the specified item.
* RemoveAt(): Removes an element at the specified index.
* Reverse(): Reverses the order of the elements in the ArrayList.
* Sort(): Alphabetically sorts the elements in the ArrayList.
* ToArray(): Copies the elements of the ArrayList to an array.
* TrimToSize(): Sets the capacity of the ArrayList to the number of elements in the ArrayList.

Advantage and disadvantage of ArrayList

* While the ArrayList provides added flexibility over the standard array, this flexibility comes at the cost of performance.

Disadvantage of using ArrayList

* Because the ArrayList stores an array of objects, when reading the value from an ArrayList you need to explicitly cast it to the data type being stored in the specified location. Recall that an array of a value type—such as a System.Int32, System.Double, System.Boolean, and so on—is stored contiguously in the managed heap in its unboxed form. The ArrayList's internal array, however, is an array of object references. Therefore, even if you have an ArrayList that stores nothing but value types, each ArrayList element is a reference to a boxed value type.
* The boxing and unboxing comes with using value types in an ArrayList can hamper the performance of your application when using large ArrayLists with many reads and writes.
* Having an object array also introduces potential bugs that won't be noticed until run-time. A developer may intend to only add elements of a particular type to an ArrayList, but since the ArrayList allows any type to be added, adding an incorrect type won't be caught during compilation.

Using an ArrayList Class

ArrayLists are not used like standard arrays. Normally, items are just added to an ArrayList using the Add method, unless there is a reason why an item should be added at a particular position, in which case the Insert method should be used.

The first thing we have to do with an ArrayList is declare it, as follows:

ArrayList grades = new ArrayList();

Addition to Array List:

* Objects are added to an ArrayList using the Add method. This method takes one argument—an Object to add to the ArrayList. The Add method also returns an integer indicating the position in the ArrayList where the element was added, though this value is rarely used in a program.

grades.Add(100);  
grades.Add(84);  
int position;  
position = grades.Add(77);

* If you want to add an element to an ArrayList at a particular position, you can use the Insert method. This method takes two arguments: the index to insert the element, and the element to be inserted.

grades.Insert(1, 99);

grades.Insert(3, 80);

* In addition to adding individual objects to an ArrayList, you can also add ranges of objects. The objects must be stored in a data type that is derived from ICollection. This means that the objects can be stored in an array, a Collection, or even in another ArrayList.
  + There are two different methods you can use to add a range to an ArrayList. These methods are AddRange and InsertRange. The AddRange method adds the range of objects to the end of the ArrayList, and the InsertRange method adds the range at a specified position in the ArrayList.

Displaying Items from ArrayList:

The objects in an ArrayList can be displayed using a For Each loop. The ArrayList has a built-in enumerator that manages iterating through all the objects in the ArrayList, one at a time.

Check Current Capacity of ArrayList

You can check the current capacity of an ArrayList by calling the Capacity property and you can determine how many elements are in an ArrayList by calling the Count property:

Console.WriteLine("The current capacity of grades is:" + grades.Capacity);  
Console.WriteLine("The number of grades in grades is:" + grades.Count);

Position of an object in ArrayList

You can determine the position of an object in an ArrayList by calling the IndexOf method. This method takes one argument, an object, and returns the object’s position in the ArrayList. If the object is not in the ArrayList, the method returns -1. Here’s a short code fragment that uses the IndexOf method in conjunction with the RemoveAt method:

int pos;  
pos = grades.IndexOf(70);  
grades.RemoveAt(pos);

Delete Items from ArrayList:

There are several ways to remove items from an ArrayList.

* If you know the item you want to remove, but don’t know what position it is in, you can use the Remove method. This method takes just one argument—an object to remove from the ArrayList. If the object exists in the ArrayList, it is removed. If the object isn’t in the ArrayList, nothing happens.Whena method like Remove is used, it is typically called inside an If–Then statement using a method that can verify the object is actually in the ArrayList, such as the Contains method.

if (grades.Contains(54))

grades.Remove(54)

else

Console.Write("Object not in ArrayList.");

* If you know the index of the object you want to remove, you can use the RemoveAt method. This method takes one argument—the index of the object you want to remove. The only exception you can cause is passing an invalid index to the method. The method works like this:

grades.RemoveAt(2);

Two Other Methods

Two other methods that many programmers find useful are the ToArray method and the GetRange method.

* The GetRange method returns a range of objects from the ArrayList as another ArrayList. The GetRange method takes two arguments: the starting index and the number of elements to retrieve from the ArrayList. GetRange is not destructive, in that the objects are just copied from the original ArrayList into the new ArrayList.

ArrayList someNames = new ArrayList();

someNames = names.GetRange(2,4);

Console.WriteLine("someNames sub-ArrayList: ");

foreach (Object name in someNames)

Console.WriteLine(name);

* The ToArray method copies all the elements of the ArrayList to an array. Let’s look first at the GetRange method. The ToArray method allows you to easily transfer the contents of an ArrayList to a standard array. The primary reason you will use the ToArray method is because you need the faster access speed of an array. The ToArray method takes no arguments and returns the elements of the ArrayList to an array.

Object[] arrNames;

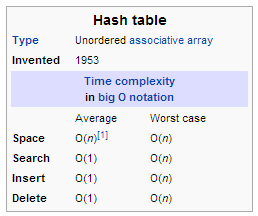
arrNames = names.ToArray();

Console.WriteLine("Names from an array: ");

for(int i = 0; i <= arrNames.GetUpperBound(0); i++)

Console.WriteLine(arrNames[i]);

3. Hastable



NameSpace: System.Collections.Hashtable

A hash table provides key/value pairing with support for efficient lookup by key.

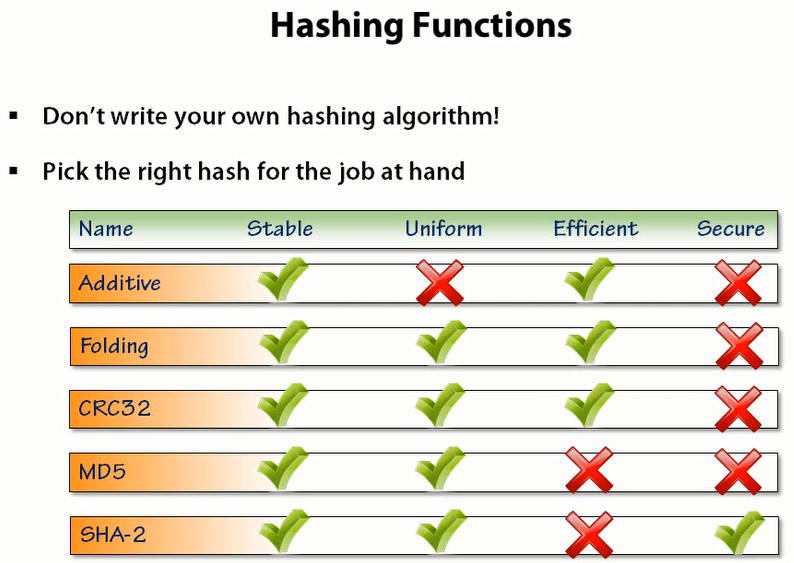
* Being nongeneric, both the key and item can be of any type and are represented as System.Object
* When adding an item to the Hashtable (through Add()), you must provide not only the item, but the unique key by which the item is accessed

|  |  |
| --- | --- |
|  |  |

Hashing Overview



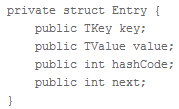
Hashing Functions



Hash Table Internal DataStructure

(Explanation for Dictionary but applies equally to HashTable)

To store its data, the Dictionary uses an array of Entry structs:



* key and value should be self-explanatory.
* hashCode provides a fast lookup to the original hash code for the functions that need it
* next is used when dealing with hash collisions.

So, the backing array is a Entry[] called entries. However, where the entries go in this array isn’t determined by the hash code. There’s actually a separate array of ints called buckets alongside entries. How is this used?

As an example, say you’ve got a Dictionary with a capacity of 5:

|  |  |
| --- | --- |
| Lets say you add an item which has hashcode 3. The actual value is just stored in the first slot of entry and the pointer stored in buckets array |  |
| You then add two more items to it, with hash codes 0 and 2, respectively: |  |

The Hashtable Class's Hash Function

* Internally, keys are hashed by use of the key type’s GetHashCode implementation.
* The idea of a hash code is to provide an integer-valued identifier for an object with as good uniqueness properties as possible.
* Whenever an entry is requested from the table by using a key, the hash code of the supplied key gets computed. Based on that value, a fast lookup in the underlying storage can be made as opposed to a sequential scan of the key/value pairs.

GetHashCode()

* Defined in the System.Object class
* The Object class's default implementation of GetHashCode() returns a unique integer that is guaranteed not to change during the lifetime of the object.



* Because every type is derived, either directly or indirectly, from Object, all objects have access to this method.
* This method can be overridden to provide a hash function more suitable to a specific class – eg: The Point class in the System.Drawing namespace, for example, overrides GetHashCode(), returning the XOR of its x and y member variables
* The Hashtable class uses a technique referred to as rehashing. (Some sources refer to rehashing as double hashing). The Hashtable class's hash function is defined as follows:

**H(key) = [GetHash(key) + 1 + (((GetHash(key) >> 5) + 1) % (hashsize – 1))] % hashsize**

* Rehashing works as follows: there is a set of hash different functions, H1 ... Hn, and when inserting or retrieving an item from the hash table, initially the H1 hash function is used. If this leads to a collision, H2 is tried instead, and onwards up to Hn if needed.

Characteristics of Hash Code Implementation:

* As long as an object does not mutate its state, the hash code must remain the same. This is important for hash tables to be able to find back an object based on its hash code-based key.
* The more random the distribution of a hash code across all objects of the same type, the better. For hash tables, this allows faster lookup times.
* For two objects that compare equal (by means of Object.Equals), the hash code must be the same.
* If the hash-code for two items does not match, they may never be considered equal (Equals will simply never be called).

The GetHashCode() method should reflect the Equals logic; the rules are:

* if two things are equal (Equals(...) == true) then they must return the same value for GetHashCode()
* if the GetHashCode() is equal, it is not necessary for them to be the same; this is a collision, and Equals will be called to see if it is a real equality or not.

Hence both the GetHashCode() and Equals() should both be overloaded (they are defined on System.Object) if there are plans to overload either of them. By overriding Equals you're basically stating that you are the one who knows better how to compare two instances of a given type, so you are the best candidate to provide the best hash code.

Note: The little mentioned equals method acts just like the == operator in that they both check to see if two objects are identical. Two object are identical if they both refer to the same instance of a class... that is, they share the same address in memory. The inherent contract of the equals method for objects that extend/inherit from Object class is that it tests for object equality rather than identity. Two objects are said to to be equivalent if they are both instances of the same Type and if the value of each field of the first object is equal to the value of the same field in the second object. Essentially this means we are recursively calling the equals method for each attribute on the objects.

Equality vs IComparer

The simplest fix would be to override GetHashCode and Equals directly rather than implementing a comparer (comparers are generally only interesting when you need to supply multiple different comparison types (case sensitive and case insensitive for example) or when you need to be able to perform comparisons on a class which you don't control.

Extra Reading - <http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/>

Collisions

|  |  |
| --- | --- |
| Collision Resolution Techniques   * Linear Probing (s+1, s+2….) * Quadratic Probing (Starts checking spots a quadratic distance away s + 12, s - 12 , s + 22 ) * Rehashing – also called double hashing - used by .Net Framework’s Hashtable class * Chaining – used by Dictionary Class |  |

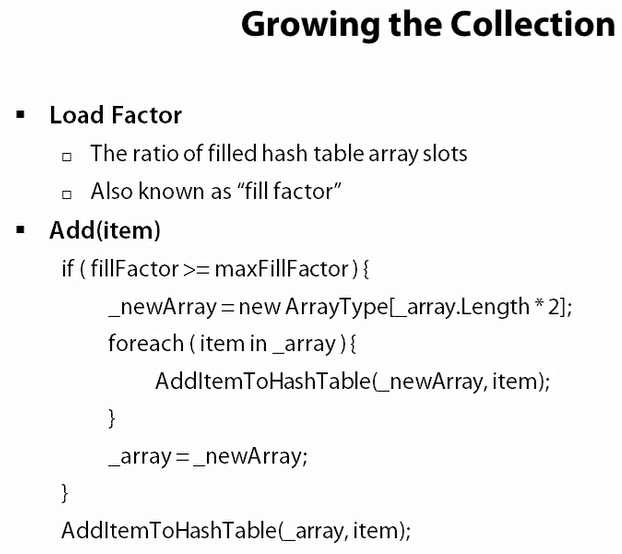
|  |  |
| --- | --- |
|  |  |
|  | |
| Removing Items with Rehash Collision Technique | Finding Items with Rehash Collision Technique |
|  |  |
| Another Example | |

In addition to being strongly-typed, the Dictionary also employs a different collision resolution strategy than the Hashtable class, using a technique referred to as chaining.

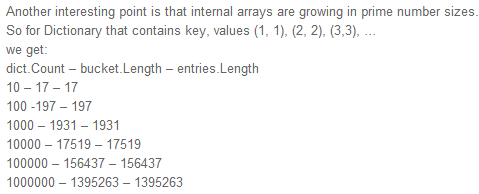
Recall that with probing, in the event of a collision another slot in the list of buckets is tried. (With rehashing, the hash is recomputed, and that new slot is tried.) With chaining, however, a secondary data structure is utilized to hold any collisions. Specifically, each slot in the Dictionary has an array of elements that map to that bucket. In the event of a collision, the colliding element is prepended to the bucket's list.

|  |  |
| --- | --- |
|  | Another Example |
|  | Another Example |
| Removing Items with Chaining Strategy | Finding Items with Chaining Strategy |
|  |  |

Growing the Collection



Loadfactor this factor helps a lot if you looking for performance, The smaller the load size (fraction of the hashtable full .6 measn 60% full) , the more efficiently our hash table works and the more memory it occupy.



A hashtable's capacity is used to calculate the optimal number of hashtable buckets based on the load factor. If you do not provide the initial capacity, the default initial capacity is zero.

When the actual load factor reaches the load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

Disadvantage of HashCode

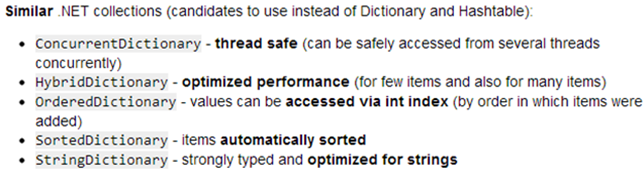
* Hash codes are used to make object lookup more efficient but don’t provide many (or any) guarantees beyond that use. For example, given two equal hash codes, you cannot conclude the two corresponding objects are equal.
* Don’t rely on hash codes to make security decisions, as a replacement for object identity, or to find objects back in a persistent store.
* The reason not to use hash codes for persistency purposes (for example, in a database) is that the default hash code implementation, as provided by System.Object, can (and will) change between different releases of the .NET Framework. All that is required from a hash code are the properties outlined earlier on, for the duration of a program’s execution. More specifically, it’s even valid for an application to produce different hash codes for objects across runs.

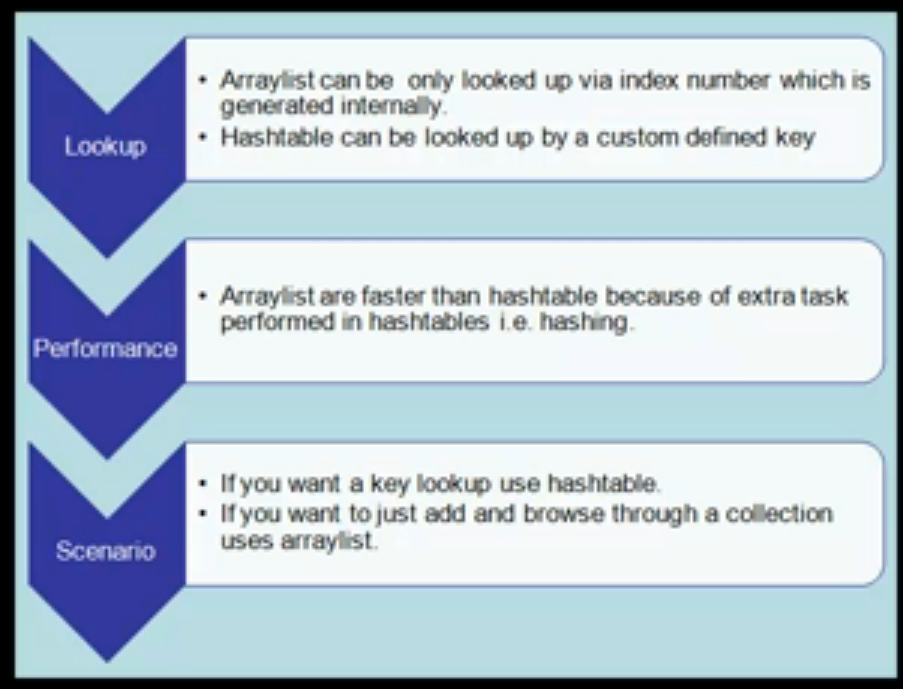
HashTable <<<>>> Dictionary

* Non-Generic <<<>>> Generic
* Everything is cast as Object <<<>>> Strongly Typed
* Leverages Rehashing/Double Hashing <<<>>> Uses Chaining for collision resolution technique
* Offers thread safe version through Synchronized() method <<<>>> Needs own thread synchronization
* Enumerated Item: DictionaryEntry <<<>>>> Enumerated item: KeyValuePair
* Request to non-existing key returns null <<<>>> Request to non-existing key throws exception
* Bit Slower (needs boxing/unboxing) for value types <<<>>> Potentially a bit faster for value types
* Is in System.Collections <<<>>> is in Systems.Collections.Generic

**HashTable / Dictionary similarities:**

* Both are internally implemented as hashtables == fast access to item data according to key
* For reference types (unlike value types mentioned above) – the speed is around the same
* Both need immutable and unique keys
* Keys of both need own GetHashCode() method





Additional Comments on Hastable (from code)

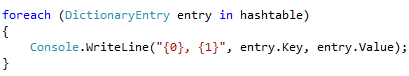
* Contains 15 constructors but you will mostly use:

Hashtable \_table = new Hashtable(); //Default one

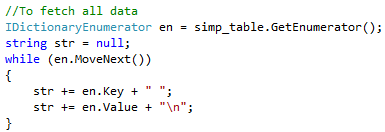
Hashtable \_table2 = new Hashtable(10); // capacity or size.  
        Hashtable \_table3 = new Hashtable(10, 06); //capacity , loadfactor

* For enumerating a Hashtable – there are various approaches:

1. Use DictionaryEntry



1. If you need to look at all the keys and values in pairs, it is best to enumerate the Hashtable instance itself. Generally we use IDictionaryEnumerator iterator in a while loop

  
 ……..ADD ITEMS TO HASHTABLE…….  
 

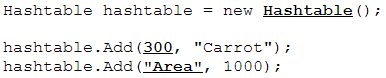
* When looking for a specific item in Hashtable – there are 2 ways to find them:
  1. Using Contains method:



* 1. Using the index



* We can add multiple types in Hashtable since everything is stored as an object



* During the Add method - A key cannot be null, but a value can be.

**Stacks And Queues**

Two list-oriented data structures that provide easy-to-understand abstractions are stacks and queues.

Data in a stack are added and removed from only one end of the list, whereas data in a queue are added at one end and removed from the other end of a list. Stacks are used extensively in programming language implementations, from everything from expression evaluation to handling function calls. A stack is known as a Last-in, First-out (LIFO) data structure.

Queues are used to prioritize operating system processes and to simulate events in the real world, such as teller lines at banks and the operation of elevators in buildings.

C# provides two classes for using these data structures: the Stack class and the Queue class

4. Stacks

Definition

* The **Stack class** is an implementation of the ICollection interface that represents a LIFO collection
* Stack<T> is implemented as an array.

Internal Structure and Details

* Stack<T> accepts null as a valid value for reference types and allows duplicate elements.
* Both Stack<T> and Queue<T> use an array internally (T[]). By default, it's setup with a length of 0. On the first item added, it's set to length of 4. Whenever you add an item >=length, the capacity is doubled.

O Notation while insertion

* If Count is less than the capacity of the stack, Push is an O(1) operation. If the capacity needs to be increased to accommodate the new element, Push becomes an O(n) operation, where n is Count. Pop is an O(1) operation.

Methods

* The Stack class includes methods for pushing, popping, and peeking values.
* There are also methods for determining the number of elements in the stack, clearing the stack of all its values, and returning the stack values as an array.

Backing Store Details

* If you don’t like the internal structure of Array (which is inherently used by Stack), you could create a backing store through
* LinkedList<T> (no fixed number of items)
* ArrayList<T> (auto resizes)
* ConcurrentStack<T>: storage mechanism more like a LinkedList than Array.

Instantiate a Stack

Three ways to instantiate a stack object:

1. The default constructor instantiates an empty stack with an initial capacity of 10 values. The default constructor is called as follows:

Stack myStack = new Stack();

1. A generic stack is instantiated as follows:   
   Stack<string> myStack = new Stack<string>();
2. a) The second Stack constructor method allows you to create a stack object from another collection object. For example, you can pass the constructor as an array and a stack is built from the existing array elements:  
   string[] names = new string[] {"Raymond", "David", "Mike"};  
   Stack nameStack = new Stack(names);

Executing the Pop method will remove “Mike” from the stack first.

1. You can also instantiate a stack object and specify the initial capacity of the stack. The code for instantiating a Stack object with an initial capacity looks like this:

Stack myStack = new Stack(25);

More Details on the Stack Methods

**Peek Method**The Peek method lets us look at the value of an item at the top of a stack without having to remove the item from the stack. Without this method, you would have to remove an item from the stack just to get at its value.  
if (IsNumeric(Nums.Peek())  
num = Nums.Pop():

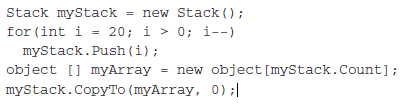
**Clear Method**The Clear method removes all the items from a stack, setting the item count to zero. It is hard to tell if the Clear method affects the capacity of a stack, since we can’t examine the actual capacity of a stack, so it’s best to assume the capacity is set back to the initial default size of 10 elements.

A good use for the Clear method is to clear a stack if there is an error in processing. For example, in our expression evaluator, if a division by 0 operation occurs, that is an error and we want to clear the stack:  
if (oper2 == 0)  
Nums.Clear();

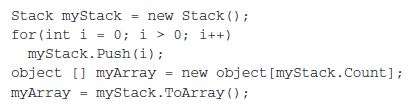
**Contains Method**The Contains method determines if a specified element is located in a stack. The method returns True if the element is found; False otherwise.We can use this method to look for a value in the stack but not currently at the top of the stack, such as a situation where a certain character in the stack might cause a processing error:

if (myStack.Contains(" "))  
 StopProcessing();  
else  
 ContinueProcessing();

**The CopyTo and ToArray Methods**The CopyTo method copies the contents of a stack into an array. The array must be of type Object since that is the data type of all stack objects. The method takes two arguments: an array and the starting array index to begin placing stack elements. The elements are copied in LIFO order, as if they were popped from the stack. Here’s a short code fragment demonstrating a CopyTo method call:



The ToArray method works in a similar manner. You cannot specify a starting array index position, and you must create the new array in an assignment statement. Here’s an example:



Use-Cases of Stack

1. When a C# program is executed, the CLR maintains a call stack which, among other things, keeps track of the function invocations. Each time a function is called, its information is added to the call stack. Upon the function's completion, the associated information is popped from the stack. The information at the top of the call stack represents the current function being executed. (For a visual demonstration of the function call stack, create a project in Visual Studio .NET, set a breakpoint and go to Debug/Start. When the breakpoint hits, display the Call Stack window from Debug/Windows/Call Stack.)
2. PostFix Calculator
3. UndoFunctionality
4. Test for String Palindrome

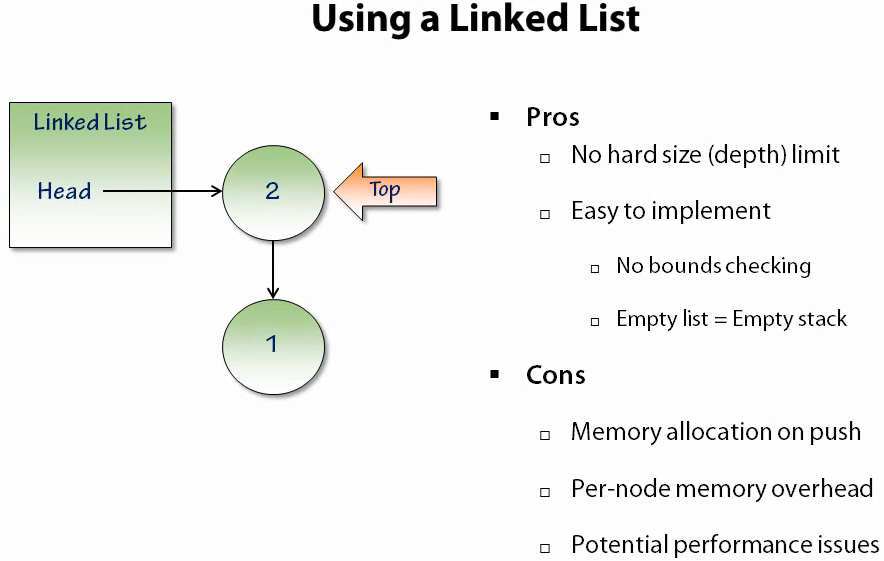
Code Sample Explanation

*Code Sample 1*

* Using .Net built Stack Class – showing Push, Pop, Peek and a few other methods

*Code Sample 2*

* This example shows a Custom Stack class which leverages LinkedList as a backing store.
* This example uses the built in LinkedList Class provided by .Net Framework.
* You can also implement the Stack with custom implemented LinkedList. See code example here - <http://www.dzone.com/snippets/implementing-stack-using>



*Code Sample 3*

* This example shows a Custom Stack class which leverages Array as a backing store.
* It still is generic Stack implementation by creating a backing store of T[]

*Code Sample 4*

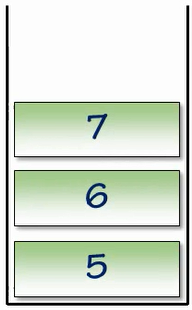
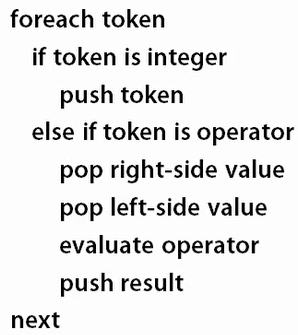
* A Stack implementation has to use an underlying structure to hold data. We’ll choose an ArrayList since we don’t have to worry about resizing the list when new items are pushed onto the stack.

*Internal Code Explanation*

* The constructor method does nothing except initialize the index variable to −1.
* The first method to implement is Push. The code calls the ArrayList Add method and adds the value passed to it to the ArrayList.
* The Pop method does three things: calls the RemoveAt method to take the top item off the stack (out of the ArrayList), decrements the index variable by 1, and, finally, returns the object popped off the stack.
* The Peek method is implemented by calling the Item method with the index variable as the argument.
* The Clear method simply calls an identical method in the ArrayList class.
* The Count property is written as a read-only property since we don’t want to accidentally change the number of items on the stack.

*Code Sample 5*

* PostFix notation: 5 2 +. They remove ambiguity.
* Infix notation: 5 + 2 \* 3

* Here we use only 1 stack where integers are stored. Everytime we hit a operator – we pop last 2 values and apply the operation to it and push the result (new integer) back to the Stack

*Code Sample 6*

The primary operations you perform with a stack are Push and Pop. Let’s look at these methods in the context of using a stack to evaluate simple arithmetic expressions.

This expression evaluator uses two stacks: one for the operands (numbers) and another one for the operators.

* An arithmetic expression is stored as a string. We parse the string into individual tokens, using a For loop to read each character in the expression.
* If the token is a number, it is pushed onto the number stack. If the token is an operator, it is pushed onto the operator stack.
* Since we are performing infix arithmetic, we wait for two operands to be pushed on the stack before performing an operation. At that point, we pop the operands and an operand and perform the specified arithmetic.
* The result is pushed back onto the stack and becomes the first operand of the next operation. This continues until we run out of numbers to push and pop.

5. Queue

A queue is a data structure where data enters at the rear of a list and is removed from the front of the list. Queues are used to store items in the order in which they occur. Queues are an example of a first-in, first-out (FIFO) data structure.

Queue Constructor Methods

* When a new Queue object is instantiated, the default capacity of the queue is 32 items.
* By definition, when the queue is full, it is increased by a growth factor of 2.0. This means that when a queue is initially filled to capacity, its new capacity becomes 64.
* You are not limited to these numbers however. You can specify a different initial capacity when you instantiate a queue. This sets the queue’s capacity to 100 items.

Queue myQueue = new Queue(100);

* You can change the growth factor as well. It is the second argument passed to the constructor:

Queue myQueue = new Queue(32, 3);

This line specifies a growth rate of 3 with the default initial capacity. You have to specify capacity even if it’s the same as the default capacity since constructor is looking for a method with diff signature

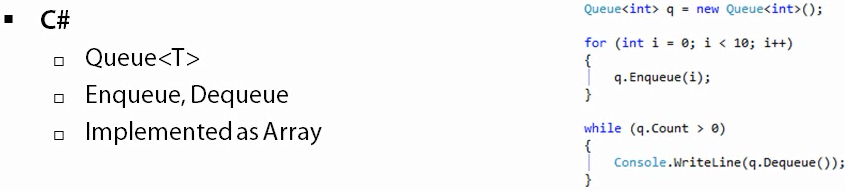
* A generic Queue is instantiated like this: Queue<int> numbers = new Queue<int>();

Primary Queue Operations

The two primary operations involving queues are adding a new item to the queue and removing an item from the queue. The operation for adding a new item is called Enqueue, and the operation for removing an item from a queue is called Dequeue.

The Enqueue operation adds an item at the end of the queue and the Dequeue operation removes an item from the front (or beginning) of the queue.

Internal Backing Store



Code Sample Explanation

*Code Sample 1*

* Using .Net built Queue Class – showing basic methods

*Code Sample 2*

|  |  |
| --- | --- |
| * This example shows a Custom Queue class which leverages LinkedList as a backing store. * This example uses the built in LinkedList Class provided by .Net Framework. |  |

*Code Sample 3*

|  |  |
| --- | --- |
| * This example shows a Custom Stack class which leverages Array as a backing store. * It still is generic Stack implementation by creating a backing store of T[] * Logic is implemented as a Circular Queue |  |

*Code Sample 4*

* This example shows a Thread Safe Circular Queue Implementation. See documentation attached with code

6. Linked List

Definition of Linked Lists

* A linked list is a collection of class objects called nodes.
* Link - The reference to another node is called a link. Each node is linked to its successor node in the list using a reference to the successor node.
* Header Node – Special node to denote beginning of the linked list.
* End of the linked list is identified by pointing it to null
* The Linked List class internally implements a doubly linked list.
* Insertion is very efficient when using Linked List. Only thing involved is adjusting link at the point of insertion
* Removing a node is similarly as above.

Insertion and Deletion are two methods that define why we use linked lists over arrays.

Structure of .Net Linked List

System.Collection.Generic namespace provides two generic classes for building linked lists:

* LinkedList class
* LinkedListNode class

The Node class provides two data fields for storing a value and a link, whereas the LinkedList class implements a doubly linked list with methods for inserting before a node as well as inserting after a node. The class also provides method for removing nodes, finding the first ad last nodes in the linked list as well as other useful methods.





Structure of Custom Linked List

A node is made up of a field for storing data and the field for the node reference.

|  |  |
| --- | --- |
| Non Generic Version | Generic Version |
|  |  |

|  |  |
| --- | --- |
|  |  |

Doubly Linked List

A doubly linked list makes it easier to move backward through a linked list and to remove a node from the list.



Advantages of Linked List

* Elements of a linked list are referenced by their relationship to the other elements of the chain and hence provide the capability to rearrange the items efficiently.
* Another advantage of linked lists over arrays is that the links provide us with the LinkedList provides good performance on insertion or deletion.

Disadvantages of Linked List

1. In terms of memory usage, the LinkedList is often less efficient than a properly-sized array or List of the elements. This is because of the memory allocation in the .NET Framework, and how objects are allocated.
2. Each node in the LinkedList will require a separate root in the garbage collector. In an array or List, many references are stored in a single block on the managed heap together, reducing the work required for a garbage collection.

Operations with Linked List

Reversing Single Linked List -   
<http://stackoverflow.com/questions/8686168/reversing-single-linked-list-in-c-sharp>

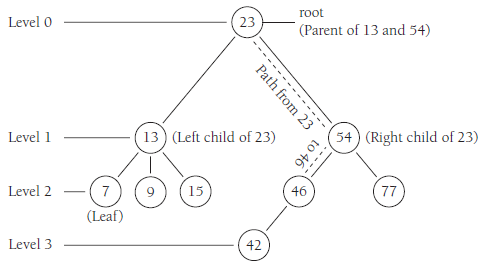
<http://www.mytechinterviews.com/reverse-a-linked-list>

Another link - <http://www.youtube.com/watch?v=DTZlz8KsSSQ&noredirect=1>

7. Binary Tree

**Definition of a Tree**

* A tree is a set of nodes connected by edges.
* Root node - The top node of a tree is called the root node.
* Parent - If a node is connected to other nodes below it, the top node is called the parent, and the nodes below it are called the parent’s children
* Leaf - A node without any child node is called a leaf.
* Path - The series of edges you follow to get from one node to another is called a path
* Depth - A tree can be broken down into levels. The root node is at Level 0, its children at Level 1, those node’s children are at Level 2, and so on. We can define the depth of a tree as the number of levels in the tree.
* Tree Traversal - Visiting all the nodes in a tree in some particular order is known as a tree transversal.
* Special Trees - Special types of trees, called binary trees, restrict the number of children to no more than two. Binary trees have certain computational properties that make them very efficient for many operations.



**Binary tree**: Tree where each node has up to two leaves (no more)



* A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right".
* By limiting the number of children to 2, we can write efficient programs for inserting data, deleting data, and searching for data in a binary tree.
* The child nodes of a parent node are referred to as the left node and the right node.
* Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists.
* Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child.
* A tree which does not have any node other than root node is called a null tree.
* In a binary tree, a degree of every node is maximum two. A tree with n nodes has exactly n−1 branches or degree.

Binary trees are often chosen over more fundamental structures, such as arrays and linked lists, because you can search a binary tree quickly (as opposed to a linked list) and you can quickly insert data and delete data from a binary tree (as opposed to an array).

**Binary search tree**:

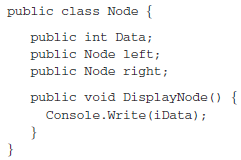
* A binary search tree is a binary tree where data with lesser values are stored in left nodes and values with greater values are stored in right nodes.
* This property provides for very efficient searches



|  |  |
| --- | --- |
|  |  |

Building a Binary Search Tree

A binary search tree is made up of nodes, so we need a Node class that is similar to the Node class we used in the linked list implementation.



Operations within Binary Search Tree

Insert Into BST

Traversal of BST

Three Traversal Methods used with Binary Search Tree:

1. Inorder
2. PreOrder
3. PostOrder

InOrder Traversal

* An inorder traversal can best be written as a recursive procedure.
* Since the method visits each node in ascending order, the method must visit both the left node and the right node of each subtree, following the subtrees under the left child of the root before following the subtrees under the right side of the root.

|  |  |
| --- | --- |
|  |  |

Example:

To demonstrate how this method works, let’s examine a program that inserts a series of numbers into a BST. Then we’ll call the inOrder method to display the numbers we’ve placed in the BST. Here’s the code:

|  |  |
| --- | --- |
|  |  |
|  | |

PreOrder Traversal

PostOrder Traversal

Finding a Node (Searching) and Min/Max Values

Removing a Node From a BST

3 use-cases:

1. Removing a Leaf Node from BST
2. Delete a Node with One Child
3. Delete a Node with Two Children

Finding the Depth of BST

Comparison of Data Structures

**Array:**

* Arrays are liner direct-access, fixed length homogeneous structure
* All elements of the array must be of the same type
* Elements can be directly accessed through index

**ArrayList:**

* An arraylist is a heterogeneous self-redimensioning array
* For situations where you can’t accurately determine the ultimate size of an array, or where the size of the array will change quite a bit over the lifetime of a program, an arraylist may be a better choice than an array. But this flexibility comes at a cost of performance
* ArrayList are much faster than hashtable because of the extra task performed in hastables i.e. hashing

**HashTable:**

* The Hashtable extends the ArrayList by allowing items to be indexed by an arbitrary key, as opposed to indexed by an ordinal value.
* If you want a key lookup use hashtable

If you want to just add and browse through a collection – use arraylist

**Dictionary:**

* The Dictionary class provides a type-safe Hashtable, with an alternate collision resolution strategy.
* More differences between Hashtable and Dictionary in previous pages

**Queue & Stack:**

The Queue and Stack provide List -like capabilities in that they can store an arbitrary number of elements. The Queue and Stack differ from the List in the sense that while the List allows direct, random access to its elements, both the Queue and Stack limit how elements can be accessed.

The Queue uses a FIFO strategy, or first in, first out. That is, the order with which items can be removed from the Queue is precisely the order with which they were added to the Queue. To provide these semantics, the Queue offers two methods: Enqueue() and Dequeue(). Queues are useful data structures for job processing or other tasks where the order with which the items are processed is based by the order in which they were received.

The Stack, on the other hand, offers LIFO access, which stands for last in, first out. Stacks provide this access scheme through its Push() and Pop() methods. Stacks are used in a number of areas in computer science, from code execution to parsing.

**Binary Search Tree:**

* A hash table is an unordered data structure.

A binary search tree is a sorted data structure

**Array**

* The array is the natural data structure to use when working with lists.
* Arrays provide fast access to stored items and are easy to loop through.
* And, of course, the array is already part of the language and you don’t have to use extra memory and processing time using a user-defined data structure.

Disadvantages of Array

The array is not the perfect data structure.

1. Random Access - elements in an array are referenced by position (the index) (like a[10]), and hence random access to the items in the list is fast and efficient
2. Unordered Array - Searching for an item in an unordered array is slow because you have to possibly visit every element in the array before finding the element you’re searching for.
3. Ordered Array - Ordered (sorted) arrays are much more efficient for searching, but insertions and deletions are slow because you have to shift the elements up or down to either make space for an insertion or remove space with a deletion.

**LinkedList**

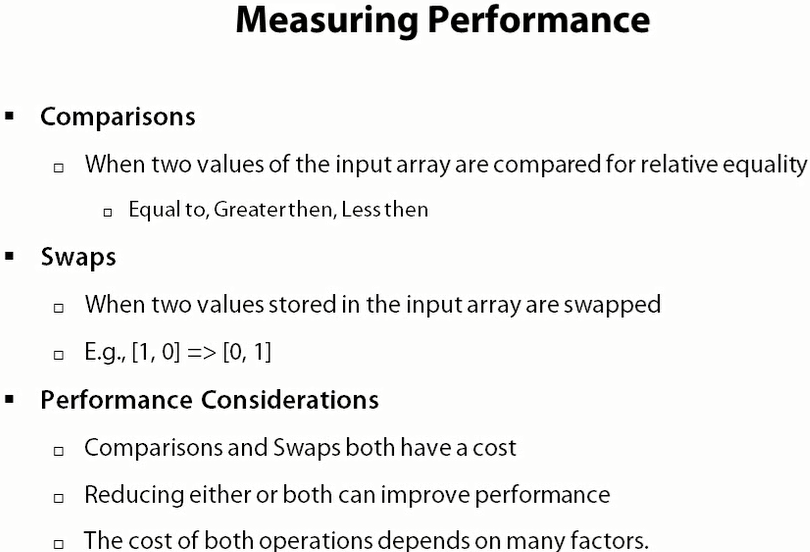
* Elements of a linked list are referenced by their relationship to the other elements of the chain and hence provide the capability to rearrange the items efficiently.
* Another advantage of linked lists over arrays is that the links provide us with the LinkedList provides good performance on insertion or deletion.

Disadvantages of LinkedList

1. In terms of memory usage, the LinkedList is often less efficient than a properly-sized array or List of the elements. This is because of the memory allocation in the .NET Framework, and how objects are allocated.
2. Each node in the LinkedList will require a separate root in the garbage collector. In an array or List, many references are stored in a single block on the managed heap together, reducing the work required for a garbage collection.

Basic Sorting Algorithms

The two most common operations performed on data stored in a computer are sorting and searching.



Sorting Algorithms

Bubble

Insertion

Selection

Merge

Quick

Each of these algorithms is easy to understand and easy to implement. They are not the best overall algorithms for sorting by any means, but for small data sets and in other special circumstances, they are the best algorithms to use.

|  |  |  |
| --- | --- | --- |
| **Sorting Options** | | |
| 1 | Bubble |  |
| 2 | Insertion |  |
| 3 | Selection |  |
| 4 | Merge |  |
| 5 | Quick |  |

1. Bubble Sort

Features

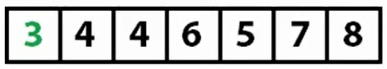
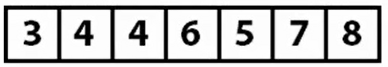
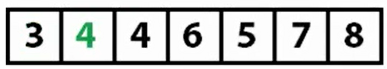
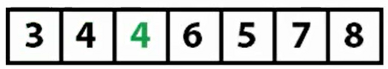
* Simplest sorting algorithm, but is one of the slowest sorting algorithms
* On Each Pass
  + Compare each array item to its right neighbor
  + If the right neighbor is smaller than Swap them
  + Repeat for the remaining array items.
* Imp – You keep going through the pass till the last pass had atleast had one swap.
* In-place sort
* Not appropriate for large unordered dataset
* The sort gets its name because values “float like a bubble” from one end of the list to another. Higher values float to the right whereas lower values float to the left. This behavior is caused by moving through the list many times, comparing adjacent values and swapping them if the value to the left is greater than the value to the right.

Example1

Pass 1

Pass 2

Pass 3

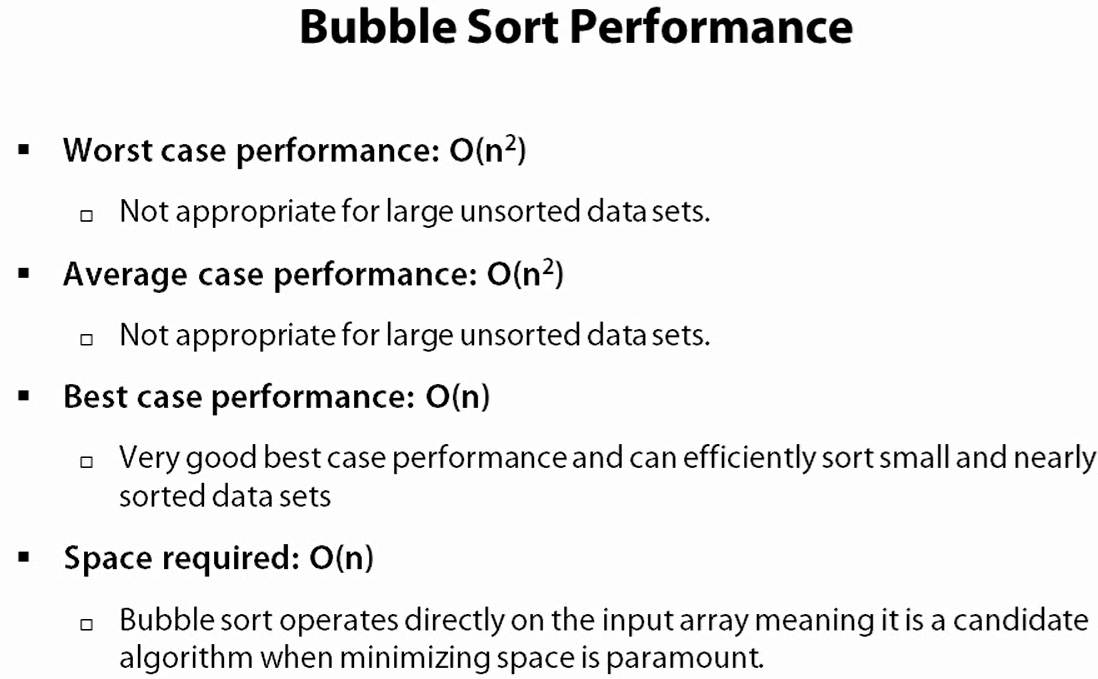
Example2

Two numbers from the numbers inserted into the array (2 and 72) are highlighted with circles. You can watch how 72 moves from the beginning of the array to the middle of the array, and you can see how 2 moves from just past the middle of the array to the beginning of the array.

Notice that the outer loop starts at the end of the array and moves toward the beginning of the array. If you look back at Figure 3.1, the highest value in the array is in its proper place at the end of the array. This means that the array indices that are greater than the value in the outer loop are already in their proper place and the algorithm doesn’t need to access these values any more. The inner loop starts at the first element of the array and ends when it gets to the next to last position in the array. The inner loop compares the two adjacent positions indicated by inner and inner +1, swapping them if necessary.

|  |  |
| --- | --- |
|  |  |

Performance



2. Insertion Sort

Features

* Sorts each item in the array as they are encountered (single pass not multiple passes to sort)
* As the current item works from left to right
  + Everything from left of the item is known to be sorted
  + Everything to the right is unsorted
* In-place sort
* Not appropriate for large unordered dataset

Example1





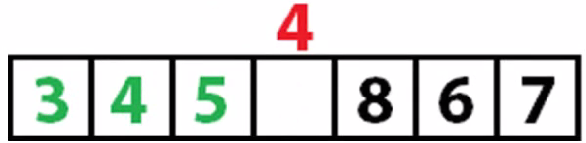














8 is already in place -> 











Example2

The Insertion sort is an analog to the way we normally sort things numerically or alphabetically. Let’s say that I have asked a class of students to turn in index card with their names, id numbers, and a short biographical sketch. The students return the cards in random order, but I want them to be alphabetized so I can build a seating chart. I take the cards back to my office, clear off my desk, and take the first card. The name on the card is Smith. I place it at the top left position of the desk and take the second card. It is Brown. I move Smith over to the right and put Brown in Smith’s place. The next card is Williams. It can be inserted at the right without having to shift any other cards. The next card is Acklin. It has to go at the beginning of the list, so each of the other cards must be shifted one position to the right to make room. That is how the Insertion sort works.

The Insertion sort has two loops. The outer loop moves element by element through the array whereas the inner loop compares the element chosen in the outer loop to the element next to it in the array. If the element selected by the outer loop is less than the element selected by the inner loop, array elements are shifted over to the right to make room for the inner loop element, just as described in the preceding example.

|  |
| --- |
|  |



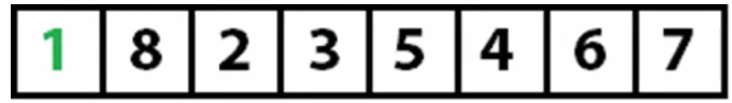
3. Selection Sort

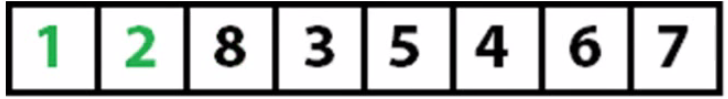
Features

* Sorts each item in the array by finding the smallest item and swapping it into the array in the first unsorted location. The smallest element is placed in position 0, and the sort then begins again at position 1. This continues until each position except the last position has been the starting point for a new loop.
* Algorithm:
  + Enumerate the array from the first unsorted item to the end
  + Identify the smallest item
  + Swap the smallest item with the first unsorted item
* In-place sort
* Not appropriate for large unordered dataset
* Typically performs better than bubble sort but worse than insertion sort

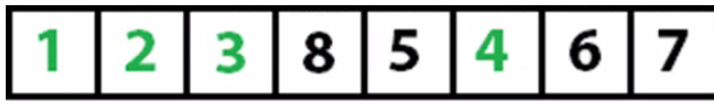
Example1



No swap necessary since 5 is already in place 

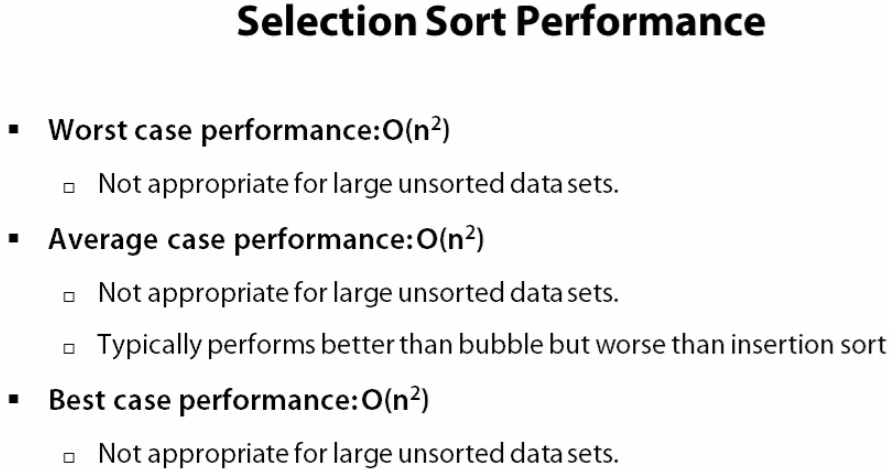
 



Example2

Two loops are used in the SelectionSort algorithm. The outer loop moves fromthe first element in the array to the next to last element, whereas the inner loop moves from the second element of the array to the last element, looking for values that are smaller than the element currently being pointed at by the outer loop. After each iteration of the inner loop, the most minimum value in the array is assigned to its proper place in the array.

|  |  |
| --- | --- |
|  |  |



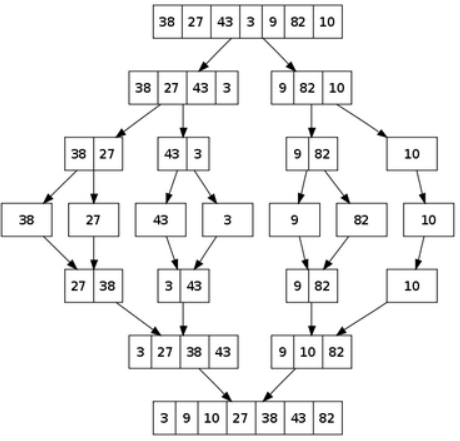
4. Merge Sort

* A Merge Sort is an example of divide and conquer paradigm. It is a comparison based sorting algorithm. It takes a list and divides the list in two lists of almost equal lengths. It then sorts the list by applying merge sort recursively, which divides the divided lists into two sublists for each and applying the merge sort to them as well.

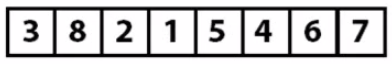
A merge sort works as follows:

* + 1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
    2. Divide the unsorted list into two sublists of about half the size.
    3. Sort each sublist recursively by re-applying merge sort.
    4. Merge the two sublists back into one sorted list.
* Performance is impressive but is fixed – which is always good
* Very appropriate for large unordered dataset (since performance is always fixed)
* Like the Quick Sort, the merge sort is recursive which can make it a bad choice for applications that run on machines with limited memory.

Example 1



Example 2

Step 1 - 

Step 2 - 

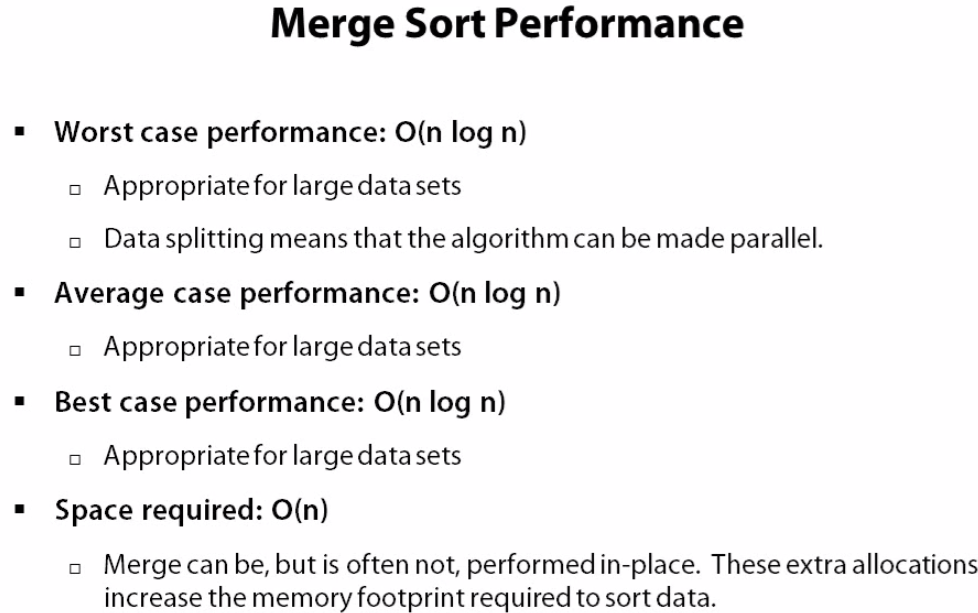
Step 3 - 

Step 4 - 

Step 5 - 

Step 6 - 

Step 7 - 



Code Links-

1. <http://www.softwareandfinance.com/CSharp/MergeSort_Recursive.html>

2. <http://tutorials.csharp-online.net/Merge_Sort>

Quick Sort

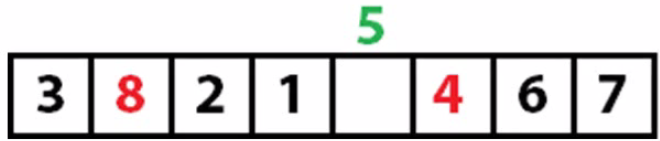
Features

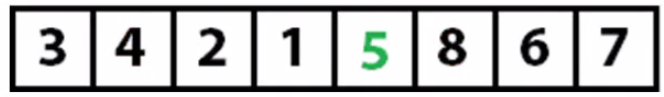
* The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code.
* Divide and conquer algorithm
* The recursive algorithm consists of four steps:

1. If there are one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used).
3. Split the array into two parts - Put all the values before the pivot to the left and above to the right
4. The pivot point is now sorted – everything right is larger, everything left is smaller. Recursively Perform pivot and partition algorithm on the left and right partitions.

* The quick sort is by far the fastest of the common sorting algorithms. It is possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster.
* The quick sort is recursive, which can make it a bad choice for applications that run on machines with limited memory.

Example 1

5 is picked as a pivot point -   
Move all the values less than 5 to left and all values greater than 5 to the right - 

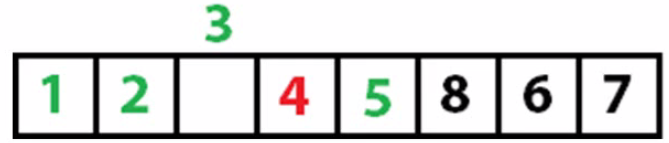


Now another value is picked 2 (as pivot value) - 

Move all the values less than 2 to left and all values greater than 5 to the right - 

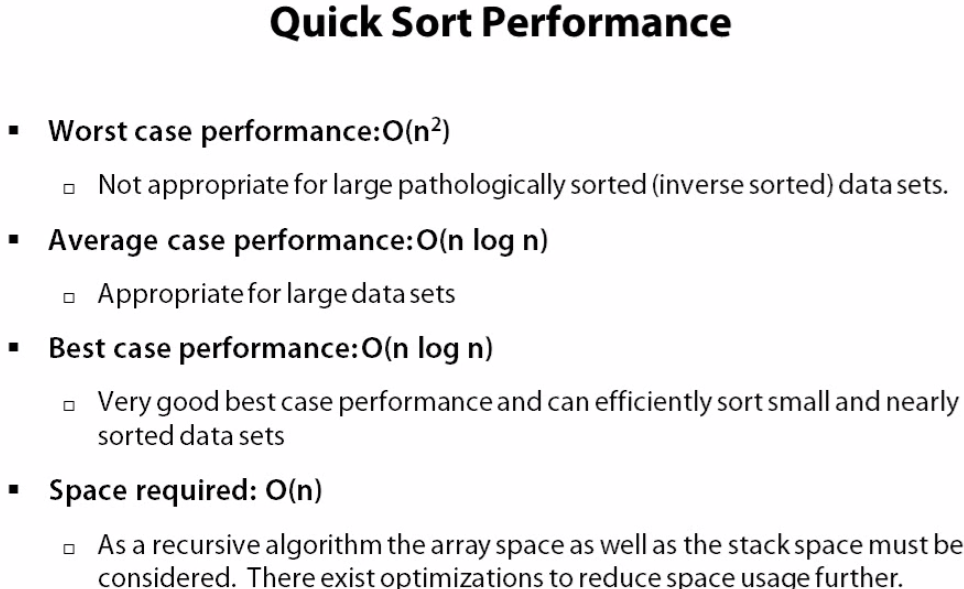


Now 1 is picked as a pivot value - 

Now 3 is picked as a pivot value -  

Moving to other side - Now 7 is picked as a pivot value -  

All sorted now - 



Code Sample - <http://tutorials.csharp-online.net/Quick_Sort>

**Timing Comparisons Of The Basic Sorting Algorithms**

These three sorting algorithms are very similar in complexity and theoretically, at least, should perform similarly when compared with each other.

The size of the array makes a big difference in the performance of the algorithm. The Selection sort is over 100 times faster than the Bubble sort and over 200 times faster than the Insertion sort.

The performance of all three algorithms degrades considerably, though the Selection sort is still many times faster than the other two. Clearly, none of these algorithms is ideal for sorting large data sets. There are sorting algorithms, though, that can handle large data sets more efficiently.

Basic Searching Algorithms

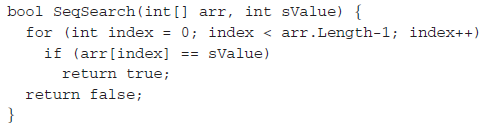
Searching for data is a fundamental computer programming task and one that has been studied for many years. There are two fundamental ways to search for data in a list: the sequential search and the binary search. Sequential search is used when the items in the list are in random order; binary search is used when the items are sorted in the list.

The simplest method of searching a data set is to start at the beginning and search for the item until either the item is found or the end of the data set is reached. This searching method works best when the data set is relatively small and unordered. If the data set is ordered, the binary search algorithm is a better choice. Binary search works by continually subdividing the data set until the item being searched for is found. You can write the binary search algorithm using both iterative and recursive codes.

1. Sequential Searching

The most obvious type of search is to begin at the beginning of a set of records and move through each record until you find the record you are looking for or you come to the end of the records. This is called a sequential search. A sequential search (also called a linear search) is very easy to implement. Start at the beginning of the array and compare each accessed array element to the value you’re searching for. If you find a match, the search is over. If you get to the end of the array without generating a match, then the value is not in the array.

Here is a function that performs a sequential search:



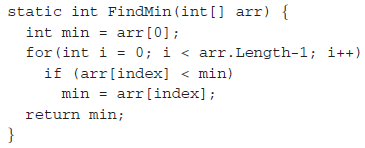
If a match is found, the function immediately returns True and exits. If the end of the array is reached without the function returning True, then the value being searched for is not in array and the function returns False.

Searching for Maximum and Minimum Values

Computer programs are often asked to search an array (or other data structure) for minimum and maximum values. In an ordered array, searching for these values is a trivial task. Searching an unordered array, however, is a little more challenging. Let’s start by looking at how to find the minimum value in an array. The algorithm is:

1. Assign the first element of the array to a variable as the minimum value.
2. Begin looping through the array, comparing each successive array element with the minimum value variable.
3. If the currently accessed array element is less than the minimum value, assign this element to the minimum value variable.
4. Continue until the last array element is accessed.
5. The minimum value is stored in the variable.

Let’s look at a function, FindMin, which implements this algorithm:



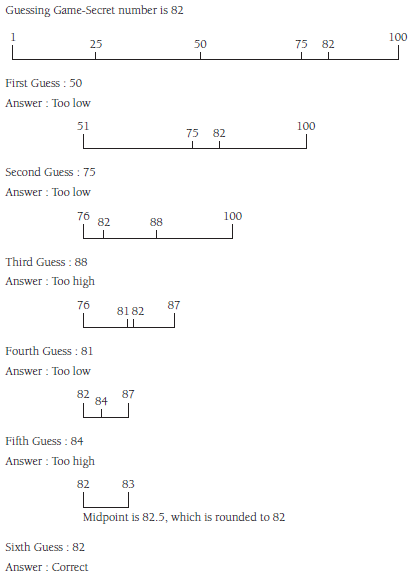
Notice that the array search starts at position 1 and not at position 0. The 0th position is assigned as the minimum value before the loop starts, so we can start making comparisons at position 1.

<http://www.c-sharpcorner.com/Forums/Thread/159465/linear-search-algorithm-its-complexity.aspx>

2. Binary Searching

When the records you are searching through are sorted into order, you can perform a more efficient search than the sequential search to find a value. This search is called a binary search.

To understand how a binary search works, imagine you are trying to guess a number between 1 and 100 chosen by a friend. For every guess you make, the friend tells you if you guessed the correct number, or if your guess is too high, or if your guess is too low. The best strategy then is to choose 50 as the first guess. If that guess is too high, you should then guess 25. If 50 is to low, you should guess 75. Each time you guess, you select a new midpoint by adjusting the lower range or the upper range of the numbers (depending on if your guess is too high or too low), which becomes your next guess. As long as you follow that strategy, you will eventually guess the correct number.



Recursive Binary Search Algorithm

Although the version of the binary search algorithm developed in the previous section is correct, it’s not really a natural solution to the problem. The binary search algorithm is really a recursive algorithm because, by constantly subdividing the array until we find the item we’re looking for (or run out of room in the array), each subdivision is expressing the problem as a smaller Sequential Searching version of the original problem. Viewing the problem this ways leads us to discover a recursive algorithm for performing a binary search.

The main problem with the recursive binary search algorithm, as compared to the iterative algorithm, is its efficiency. When a 1,000-element array is sorted using both algorithms, the recursive algorithm is consistently 10 times slower than the iterative algorithm.

Of course, recursive algorithms are often chosen for other reasons than efficiency, but you should keep in mind that anytime you implement a recursive algorithm, you should also look for an iterative solution so that you can compare the efficiency of the two algorithms.

Array Characteristics

The Array class has a built-in binary search method. It takes two arguments, an array name and an item to search for, and it returns the position of the item in the array, or -1 if the item can’t be found.

When the built-in binary search method is compared with our custombuilt method, it consistently performs 10 times faster than the custom-built method, which should not be surprising. A built-in data structure or algorithm should always be chosen over one that is custom-built, if the two can be used in exactly the same ways.

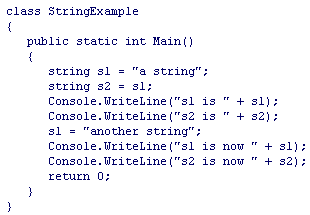
Strings, String Class and String Builder

Strings in C# have a schizophrenic nature—they are both native types and objects of a class. Actually, to be more precise, we should say that we can work with strings as if they are native data values, but in reality every string created is an object of String class.

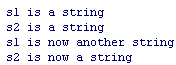
The StringBuilder class is used when a program must make many changes to a String object because strings and String objects are immutable, whereas StringBuilder objects are mutable.

Strings are created like this: string name = "Jennifer Ingram"  
Though you can of course, declare the variable and assign it data in two separate statements. The declaration syntax makes name look like it is just a regular variable, but it is actually an instance of a String object.

strings are **immutable**. Should you make changes to one of these strings, this will create an entirely new string object, leaving the other string unchanged.



The output from this is as follows:



Changing the value of s1 has no effect on s2, contrary to what you'd expect with a reference type! What's happening here is that when s1 is initialized with the value a string, a new string object is allocated on the heap. When s2 is initialized, the reference points to this same object, so s2 also has the value a string. However, when you now change the value of s1, instead of replacing the original value, a new object will be allocated on the heap for the new value. The s2 variable will still point to the original object, so its value is unchanged. Under the hood, this happens as a result of **operator overloading.**

An escape character begins with a backslash (\) and is followed by a single letter that represents the format. For example, \n indicates a newline (line break) and \t indicates a tab.

string name = "Mike McMillan\nInstructor, CIS\tRoom 306";

String Literal: You can prefix a string literal with the at character (@) and all the characters in it will be treated at face value; they won't be interpreted as escape sequences:

string filepath = @"C:\ProCSharp\First.cs";

Frequently Used String Class Methods

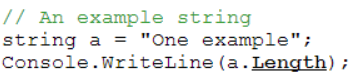
Although there are many operations you can perform on strings, a small set of operations dominates. Three of the top operations are as follows:

1. Finding a substring in a string

The Substring method takes two arguments: a starting position and the number of characters to pull. Look at the following example:

string s = "Now is the time";  
string sub = s.Substring(0,3);

1. Determining the length of a string



1. Determining the position of a character in a string.

Strings in C# are zero-based and therefore the first character in the string is at position 0, the second character is at position 1, and so on. If the character can’t be found in the string, a −1 is returned.

The IndexOf method reports the zero-based index of the first occurrence of a specified Unicode character or string within this instance. The method returns -1 if the character or string is not found in this instance.

|  |  |
| --- | --- |
|  |  |

1. Split and Join Methods

The Split method takes a string, breaks it into constituent pieces, and puts those pieces into a String array. The method works by focusing on a separating character to determine where to break up the string.

We can go the other way, from an array to a string, using the Join method. This method takes two arguments: the original array and a character to separate the elements. A string is built consisting of each array element followed by the separator element.

Methods for Comparing Strings

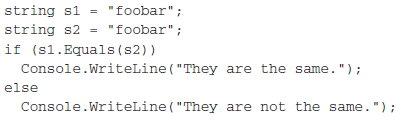
Methods for Comparing StringsStrings are compared with each other much as we compare numbers. However, since it’s not obvious if “a” is greater than or less than “H”, we have to have some sort of numeric scale to use. That scale is the Unicode table. Each character (actually every symbol) has a Unicode value, which the operating system uses to convert a character’s binary representation to that character. You can determine a character’s Unicode value by using the ASC function.

To find the ASCII value for a character, simply convert the character to an integer using a cast, like this:

int charCode;  
charCode = (int)'a';

The value 97 is stored in the variable.

Comparison Methods:

1. Equals Method (or strA == strB)
2. CompareTo Method
3. Compare Method
4. StartsWith
5. EndsWith
6. Equals Method: The first comparison method we’ll examine is the Equals method. This method is called from a String object and takes another String object as its argument. It then compares the two String objects character-by-character. If they contain the same characters (based on their numeric codes), the method returns True. Otherwise, the method returns False.  
     
   Also another syntax which is same as .Equals(..), as it just calls that method.  
   stringValue == otherStringValue:
7. CompareTo Method: This method also takes a String as an argument but it doesn’t return a Boolean value. Instead, the method returns either 1, −1, or 0.   
   strA: The first string to compare.   
   strB: The second string to compare.

|  |  |
| --- | --- |
| **Value** | **Condition** |
| Less than zero | strA is less than strB |
| zero | strA equals strB |
| Greater than zero | strA is greater than strB |

1. Compare Method:

This is a static method provided by String to compare 2 strings. Ultimately it boils down to calling the same code as CompareTo. It is strictly a helper function but it is useful when the first string can be null since it handles that case.



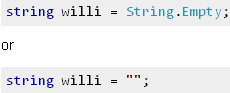
Which one to use

* The CompareTo method was designed primarily for use in sorting or alphabetizing operations. It should not be used when the primary purpose of the method call is to determine whether two strings are equivalent. To determine whether two strings are equivalent, call the Equals method."
* Because a string reference can be null, you should avoid using the equality symbol == when comparing strings. Instead, use the static String.Compare method. This method has the advantage that it can handle null string references, compare strings ignoring case, and compare strings using a specific culture.

4. and 5. StartsWith and EndsWith:

These instance methods take a string as an argument and return True if the instance either starts with or ends with the string argument.

Difference between String.Empty and null



String.Empty does not create an object whereas "" does.

Methods for Manipulating Strings

1. Insert Method: This method inserts a string into another string at a specified position.



1. Remove Method: This method takes two Integer arguments: a starting position and a count, which is the number of characters you want to remove



1. Replace Method: This method takes two arguments: a string of characters to remove and a string of characters to replace them with. The method returns the new string.



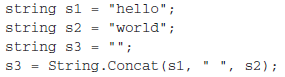
1. PadLeft: The PadLeft method right-aligns a string

|  |  |
| --- | --- |
|  |  |

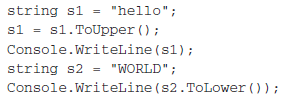
1. PadRight: PadRight method left-aligns a string.

|  |  |
| --- | --- |
|  |  |

1. Concat: We already know that the & (ampersand) operator is used for string concatenation. The String class also includes a method Concat for this purpose. This method takes a list of String objects, concatenates them, and returns the resulting string.



1. ToUpper and ToLower:



StringBuilder Class

* The StringBuilder class provides access to mutable String objects. Objects of the String class are immutable, meaning that they cannot be changed.
* Every time you change the value of a String object, a new object is created to hold the value.
* StringBuilder objects, on the other hand, are mutable. When you make a change to a StringBuilder object, you are changing the original object, not working with a copy.
* The StringBuilder class is found in the System.Text namespace so you must import this namespace into your program before you can use StringBuilder objects.

Constructing StringBuilder Objects

There are three ways to do this:

* The first way is to create the object using the default constructor: StringBuilder stBuff1 = new StringBuilder();
* This line creates the object stBuff1 with the capacity to hold a string 16 characters in length. Every time the capacity of a StringBuilder object is exceeded, the capacity is increased by 16 characters. This capacity is assigned by default, but it can be changed by passing in a new capacity in a constructor call, like this:

StringBuilder stBuff2 = New StringBuilder(25);

* The final constructor call takes a string as the argument:   
  StringBuilder stBuff3 = New StringBuilder("Hello, world");

Modifying StringBuilder Objects

We can modify a StringBuilder object by appending new characters to the end of the object, inserting characters into an object, replacing a set of characters in an object with different characters, and remove characters from an object.

Append Method

You can add characters to the end of a StringBuilder object by using the Append method. This method takes a string value as an argument and concatenates the string to the end of the current value in the object.

|  |  |
| --- | --- |
|  |  |

A formatted string can be appended to a StringBuilder object. A formatted string is a string that includes a format specification embedded in the string.

|  |  |
| --- | --- |
|  |  |

Insert Method

This method allows us to insert a string into the current StringBuilder object. The method can take up to three arguments. The first argument specifies the position to begin the insertion. The second argument is the string you want to insert. The third argument, which is optional, is an integer that specifies the number of times you want to insert the string into the object.

|  |  |
| --- | --- |
|  |  |

Remove Method

The StringBuilder class has a Remove method for removing characters from a StringBuilder object. This method takes two arguments: a starting position and the number of characters to remove.

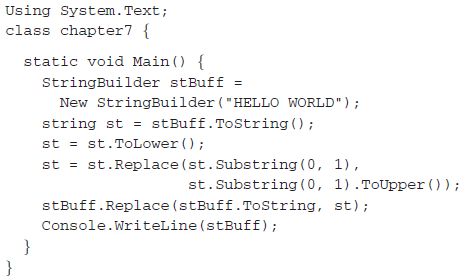
|  |  |
| --- | --- |
|  |  |

Replace Method

This method takes two arguments: the old string to replace and the new string to put in its place.

|  |  |
| --- | --- |
|  |  |

Convert to String Class



This program displays the string “Hello world” by first converting stBuff to a string (the st variable), making all the characters in the string lowercase, capitalizing the first letter in the string, and then replacing the old string in the StringBuilder object with the value of st. The ToString method is used in the first argument to Replace because the first parameter is supposed to be a string. You can’t call the StringBuilder object directly here.

Efficiency of the String Class to StringBuilder:

We know that String objects are immutable and StringBuilder objects are not. It is reasonable to believe, then, that the StringBuilder class is more efficient. However, we don’t want to always use the StringBuilder class because the StringBuilder class is lacking several methods we need to perform reasonably powerful string processing. It is true that we can transform StringBuilder objects into String objects (and then back again) when we need to use String methods but we need to know when we need to use StringBuilder objects and when it’s okay to just stick with String objects.

Results: For relatively small objects, there is really no difference between String objects and StringBuilder objects. In fact, you can argue that for strings of up to 1,000 characters, using the String class is just as efficient as using the StringBuilder class. However, when we get to 10,000 characters, there is a vast increase in efficiency for the StringBuilder class. There is, though, a vast difference between 1,000 characters and 10,000 characters.

When to use string classes: Stings are immutable, so when you are not performing any operations on strings, then use string class.

Questions

|  |  |
| --- | --- |
| Data Structure - Collections | |
|  | How is list collection implemented?  Internally it uses an array |
|  | System.Collection.Generic  Introduced in .Net 2.0, this namespace contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.   1. Dictionary<TKey, TValue> : Internally is implemented as Hash-table 2. List<T> 3. LinkedList<T> 4. Queue<T> 5. Stack<T>   Collections above are not thread-safe by default.  System.Collection.Concurrent  Introduced in.Net 4.0 and namespace provides several thread-safe collection classes that should be used in place of the corresponding types in the System.Collections and System.Collections.Generic namespaces whenever multiple threads are accessing the collection concurrently.   1. ConcurrentDictionary<TKey, TValue> 2. ConcurrentQueue<T> 3. ConcurrentStack<T> 4. ConcurrentBag<T> : Represents a thread-safe, unordered collection of objects. |
|  | Which of the following are collections? - Structs - Enum - Dictionary All collections directly or indirectly should implement the ICollection interface or the ICollection(Of T) generic interface. If you check System.Array you will see ICollection interface in there. Both "struct" and "enum" are value types. Correct answer is Dictionary. |
|  | What’s the difference between the System.Array.CopyTo() and System.Array.Clone()?  The Clone() method returns a new array (a shallow copy) object containing all the elements in the original array. The CopyTo() method copies the elements into another existing array. Both perform a shallow copy. |
|  | How can you sort the elements of the array in descending order?  By calling Array.Sort() and then Array.Reverse() methods. |
|  | How do you add objects to hashtable   * With Add method * With insert Method * With = operator   Both the Add method and the = operator. Example for the = operator below:    GetEnumerator( ) of IEnumerable interface returns  GetEnumerator() returns a reference to System.Collections.Ienumerator class which allows a simple iteration over a collection |
|  | In a singly linked list, if I give a pointer to a node, how would you delete it and what other steps would you take? |
|  | What is the different between binary tree and arraylist data structure. When would you use one over the other? |
|  | What class is underneath the SortedList class?  A sorted HashTable. |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Sorting and Searching Questions | |
|  | In .Net what is the difference between SortedDictionary and SortedList. |
|  | How does QuickSort work? Complexity? - Implement pseudo QuickSort function |
|  | Which data structure uses Binary Search? |
|  | **Binary Search Question –**  I have 10 balls and want to find the lightest one with smallest number of tries – how would you do it  <http://en.wikipedia.org/wiki/Binary_search_algorithm>  <http://www.programming-free.com/2012/07/c-list-linear-search-vs-binary-search.html#.UUthIhzvsmM> |
|  | **Merge Sort Question –**  <http://stackoverflow.com/questions/15557481/algorithm-approach-suggestions-merging-two-arrays-in-order>  <http://en.wikipedia.org/wiki/Merge_sort>  <http://stackoverflow.com/questions/12148712/c-sharp-merge-sort-performance> |
|  | Quick Sort Algorithm  <http://en.wikipedia.org/wiki/Quicksort> |
|  | <http://csharp-algos.blogspot.com/> |
|  |  |

|  |  |
| --- | --- |
| String Concepts | |
|  | In order to use stringbuilder in our class we need to refer:  - System.Text  - System.IO  - System.StringBuilder  System.Text |
|  | For performing repeated modification on string which class is preferred   * String * StringBuilder * Both   StringBuilder class is preferred for performing repeated operations on string since its value is mutable. String class represents a text value which is immutable. All operations on string class which appear to be modifying its value actually return the new value. So for modifying text contents StringBuilder class is preferred. |
|  | Which of the following is not a member of stringbuilder   1. append() 2. insert() 3. replace() 4. substring()   Substring() is not a member of the StringBuilder |
|  | Explain String.Intern |
|  | What’s the difference between System.String and System.StringBuilder classes?  OR  What’s the difference between System.String and System.Text.StringBuilder classes?  System.String is immutable, System.StringBuilder was designed with the purpose of having a mutable string where a variety of operations can be performed. |
|  | When should you use the StringBuilder class instead of the String class?  When building a string from shorter strings.  When working with text data longer than 256 bytes.  When you want to search and replace the contents of a string.  When a string is a value type. |
|  | What is the difference between a.Equals(b) and a == b?  For value types, the expressions are the same  For reference types, a==b is true only when the objects have the same reference (pointer)  a.Equals(b), on the other hand, is true for reference types with different pointers provided they have the same value. An example usage would be for a class that is mapped to a database table. There could be two different instances of the class with an id of 5. When using ==, the value would return false. However, if Equals is appropriately overridden to compare the id, than a.Equals(b) should return true since they have the same id |