Other Generic Collections in C#

LinkedList<T>

LinkedList<T> is a general-purpose linked list. It supports enumerators and implements the ICollection interface, consistent with other collection classes in the .NET Framework. LinkedList<T> provides separate nodes of type LinkedListNode<T>, so insertion and removal are O(1) operations. You can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap. Because the list also maintains an internal count, getting the Count property is an O(1) operation. Because the LinkedList<T> is doubly linked, each node points forward to the Next node and backward to the Previous node. If the LinkedList<T> is empty, the First and Last properties contain null.

Example:

```
public class Example
{
    public static void Main()
    {
        // Create the link list.
        string[] words = { "the", "fox", "jumps", "over", "the", "dog" };
        LinkedList<string> sentence = new LinkedList<string>(words);
        Display(sentence, "The linked list values:");
        Console.WriteLine("sentence.Contains(\"jumps\") = {0}",
                                            sentence.Contains("jumps"));
        // Add the word 'today' to the beginning of the linked list.
        sentence.AddFirst("today");
        Display(sentence, "Test 1: Add 'today' to beginning of the list:");
        // Move the first node to be the last node.
        LinkedListNode<string> mark1 = sentence.First;
        sentence.RemoveFirst();
        sentence.AddLast(mark1);
        Display(sentence, "Test 2: Move first node to be last node:");
        // Change the last node to 'yesterday'.
        sentence.RemoveLast();
        sentence.AddLast("yesterday");
        Display(sentence, "Test 3: Change the last node to 'yesterday':");
        // Move the last node to be the first node.
        mark1 = sentence.Last;
        sentence.RemoveLast();
        sentence.AddFirst(mark1);
        Display(sentence, "Test 4: Move last node to be first node:");
        // Indicate the last occurence of 'the'.
        sentence.RemoveFirst();
        LinkedListNode<string> current = sentence.FindLast("the");
        IndicateNode(current, "Test 5: Indicate last occurence of 'the':");
        // Add 'lazy' and 'old' after 'the' (the LinkedListNode named current).
        sentence.AddAfter(current, "old");
        sentence.AddAfter(current, "lazy");
        IndicateNode(current, "Test 6: Add 'lazy' and 'old' after 'the':");
        // Indicate 'fox' node.
        current = sentence.Find("fox");
        IndicateNode(current, "Test 7: Indicate the 'fox' node:");
        // Add 'quick' and 'brown' before 'fox':
        sentence.AddBefore(current, "quick");
        sentence.AddBefore(current, "brown");
        IndicateNode(current, "Test 8: Add 'quick' and 'brown' before 'fox':");
        // Keep a reference to the current node, 'fox',
        // and to the previous node in the list. Indicate the 'dog' node.
        mark1 = current;
        LinkedListNode<string> mark2 = current.Previous;
        current = sentence.Find("dog");
        IndicateNode(current, "Test 9: Indicate the 'dog' node:");
        // The AddBefore method throws an InvalidOperationException
        // if you try to add a node that already belongs to a list.
        Console.WriteLine("Test 10: Throw exception by adding node (fox) already
                               in the list:");
        try
        {
            sentence.AddBefore(current, mark1);
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Exception message: {0}", ex.Message);
        }
        Console.WriteLine();
        // Remove the node referred to by mark1, and then add it before the node
        // referred to by current. Indicate the node referred to by current.
        sentence.Remove(mark1);
        sentence.AddBefore(current, mark1);
        IndicateNode(current, "Test 11: Move a referenced node (fox) before the
                         current node (dog):");
        // Remove the node referred to by current.
        sentence.Remove(current);
        IndicateNode(current, "Test 12: Remove current node (dog) and attempt to
                         indicate it:");
        // Add the node after the node referred to by mark2.
        sentence.AddAfter(mark2, current);
        IndicateNode(current, "Test 13: Add node removed in test 11 after a
                         referenced node (brown):");
        // The Remove method finds and removes the first node that that has the
        // specified value.
        sentence.Remove("old");
        Display(sentence, "Test 14: Remove node that has the value 'old':");
        // When the linked list is cast to ICollection(Of String),
        // the Add method adds a node to the end of the list.
        sentence.RemoveLast();
        ICollection<string> icoll = sentence;
        icoll.Add("rhinoceros");
        Display(sentence, "Test 15: Remove last node, cast to ICollection, and
                         add 'rhinoceros':");
        Console.WriteLine("Test 16: Copy the list to an array:");
        // Create an array with the same number of elements as the linked list.
        string[] sArray = new string[sentence.Count];
        sentence.CopyTo(sArray, 0);
        foreach (string s in sArray)
        {
            Console.WriteLine(s);
        }
        // Release all the nodes.
        sentence.Clear();
        Console.WriteLine("Test 17: Clear linked list. Contains 'jumps' = {0}",
                       sentence.Contains("jumps"));
        Console.ReadLine();
    }
    private static void Display(LinkedList<string> words, string test)
    {
        Console.WriteLine(test);
        foreach (string word in words)
        {
            Console.Write(word + " ");
        }
        Console.WriteLine();
    }
    private static void IndicateNode(LinkedListNode<string> node, string test)
    {
        Console.WriteLine(test);
        if (node.List == null)
        {
            Console.WriteLine("Node '{0}' is not in the list.\n", node.Value);
            return;
        }
        StringBuilder result = new StringBuilder("(" + node.Value + ")");
        LinkedListNode<string> nodeP = node.Previous;
        while (nodeP != null)
        {
            result.Insert(0, nodeP.Value + " ");
            nodeP = nodeP.Previous;
        }
        node = node.Next;
        while (node != null)
        {
            result.Append(" " + node.Value);
            node = node.Next;
        }
        Console.WriteLine(result);
    }
}
//This code example produces the following output:
//
//The linked list values:
//the fox jumps over the dog
//Test 1: Add 'today' to beginning of the list:
```

```
//today the fox jumps over the dog
//Test 2: Move first node to be last node:
//the fox jumps over the dog today
//Test 3: Change the last node to 'yesterday':
//the fox jumps over the dog yesterday
//Test 4: Move last node to be first node:
//yesterday the fox jumps over the dog
//Test 5: Indicate last occurence of 'the':
//the fox jumps over (the) dog
//Test 6: Add 'lazy' and 'old' after 'the':
//the fox jumps over (the) lazy old dog
//Test 7: Indicate the 'fox' node:
//the (fox) jumps over the lazy old dog
//Test 8: Add 'quick' and 'brown' before 'fox':
//the quick brown (fox) jumps over the lazy old dog
//Test 9: Indicate the 'dog' node:
//the quick brown fox jumps over the lazy old (dog)
//Test 10: Throw exception by adding node (fox) already in the list:
//Exception message: The LinkedList node belongs a LinkedList.
//Test 11: Move a referenced node (fox) before the current node (dog):
//the quick brown jumps over the lazy old fox (dog)
//Test 12: Remove current node (dog) and attempt to indicate it:
//Node 'dog' is not in the list.
//Test 13: Add node removed in test 11 after a referenced node (brown):
//the quick brown (dog) jumps over the lazy old fox
//Test 14: Remove node that has the value 'old':
//the quick brown dog jumps over the lazy fox
//Test 15: Remove last node, cast to ICollection, and add 'rhinoceros':
//the quick brown dog jumps over the lazy rhinoceros
//Test 16: Copy the list to an array:
//the
//quick
//brown
//dog
//jumps
//over
//the
//lazy
//rhinoceros
//Test 17: Clear linked list. Contains 'jumps' = False
```

Stack<T>

Stack<T> is implemented as an array. Stacks and queues are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue<T> if you need to access the information in the same order that it is stored in the collection. Use System.Collections.Generic.Stack<T> if you need to access the information in reverse order. Use the System.Collections.Concurrent.ConcurrentStack<T> and System.Collections.Concurrent.ConcurrentQueue<T> types when you need to access the collection from multiple threads concurrently. A common use for System.Collections.Generic.Stack<T> is to preserve variable states during calls to other procedures. Three main operations can be performed on a System.Collections.Generic.Stack<T> and its elements:

- Push inserts an element at the top of the Stack.
- Pop removes an element from the top of the Stack<T>.
- Peek returns an element that is at the top of the Stack<T> but does not remove it from the Stack<T>.

The capacity of a Stack<T> is the number of elements the Stack<T> can hold. As elements are added to a Stack<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

Example:

```
class Example
{
    public static void Main()
    {
        Stack<string> numbers = new Stack<string>();
        numbers.Push("one");
        numbers.Push("two");
        numbers.Push("three");
        numbers.Push("four");
        numbers.Push("five");
        // A stack can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }
        Console.WriteLine("\nPopping '{0}'", numbers.Pop());
        Console.WriteLine("Peek at next item to stack: {0}", numbers.Peek());
        Console.WriteLine("Popping '{0}'", numbers.Pop());
        // Create a copy of the stack, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Stack<string> stack2 = new Stack<string>(numbers.ToArray());
        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in stack2 )
        {
            Console.WriteLine(number);
        }
        // Create an array twice the size of the stack and copy the
        // elements of the stack, starting at the middle of the array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);
        // Create a second stack, using the constructor that accepts an
        // IEnumerable(Of T).
        Stack<string> stack3 = new Stack<string>(array2);
        Console.WriteLine("\nContents of the second copy, with duplicates and
                         nulls:");
        foreach( string number in stack3 )
        {
            Console.WriteLine(number);
        }
        Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
                         stack2.Contains("four"));
        Console.WriteLine("\nstack2.Clear()");
        stack2.Clear();
        Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
    }
}
/* This code example produces the following output:
five
four
three
two
one
Popping 'five'
Peek at next item to stack: four
Popping 'four'
Contents of the first copy:
one
two
three
Contents of the second copy, with duplicates and nulls:
one
two
three
stack2.Contains("four") = False
stack2.Clear()
stack2.Count = 0
 */
```

Queue<T>

This class implements a generic queue as a circular array. Objects stored in a Queue<T> are inserted at one end and removed from the other. Use Queue<T> if you need to access the information in the same order that it is stored in the collection. Three main operations can be performed on a Queue<T> and its elements:

- Enqueue adds an element to the end of the Queue<T>.
- Dequeue removes the oldest element from the start of the Queue<T>.
- Peek peek returns the oldest element that is at the start of the Queue<T> but does not remove it from the Queue<T>.

The capacity of a Queue<T> is the number of elements the Queue<T> can hold. As elements are added to a Queue<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

Example:

```
class Example
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");
        // A queue can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }
        Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
        Console.WriteLine("Peek at next item to dequeue: {0}", numbers.Peek());
        Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());
        // Create a copy of the queue, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Queue<string> queueCopy = new Queue<string>(numbers.ToArray());
        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in queueCopy )
        {
            Console.WriteLine(number);
        }
        // Create an array twice the size of the queue and copy the
        // elements of the queue, starting at the middle of the array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);
        // Create a second queue, using the constructor that accepts an
        // IEnumerable(Of T).
        Queue<string> queueCopy2 = new Queue<string>(array2);
        Console.WriteLine("\nContents of the second copy, with duplicates and
                         nulls:");
        foreach( string number in queueCopy2 )
        {
            Console.WriteLine(number);
        }
        Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
            queueCopy.Contains("four"));
        Console.WriteLine("\nqueueCopy.Clear()");
        queueCopy.Clear();
        Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
    }
}
/* This code example produces the following output:
one
two
three
four
five
Dequeuing 'one'
Peek at next item to dequeue: two
```

```
Dequeuing 'two'
Contents of the copy:
three
four
five
Contents of the second copy, with duplicates and nulls:
three
four
five
queueCopy.Contains("four") = True
queueCopy.Clear()
queueCopy.Count = 0
 */
```