LinkedList<T> Class

Namespace: System.Collections.Generic

Assemblies: System.Collections.dll, System.dll, netstandard.dll

Represents a doubly linked list.

In this article

C# Copy

```
[System.Runtime.InteropServices.ComVisible(false)]
[System.Serializable]
public class LinkedList<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.ICollection, System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

Type Parameters

T

Specifies the element type of the linked list.

Inheritance Object → LinkedList<T>

Attributes ComVisibleAttribute, SerializableAttribute

Implements ICollection<T> , IEnumerable<T> , IReadOnlyCollection<T> , ICollection , IEnumerable , IDeserializationCallback , ISerializable

Examples

The following code example demonstrates many features of the LinkedList<T> class.

C# Copy

```
using System;
using System.Text;
using System.Collections.Generic;

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 inked 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();
        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();
        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);
        Console.WriteLine();
    }
}
```

```
//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
//
```

Remarks

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.

Each node in a LinkedList<T> object is of the type LinkedListNode<T>. Because the LinkedList<T> is doubly linked, each node points forward to the Next node and backward to the Previous node.

Lists that contain reference types perform better when a node and its value are created at the same time. LinkedList<T> accepts `null` as a valid Value property for reference types and allows duplicate values.

If the LinkedList<T> is empty, the First and Last properties contain `null`.

The LinkedList<T> class does not support chaining, splitting, cycles, or other features that can leave the list in an inconsistent state. The list remains consistent on a single thread. The only multithreaded scenario supported by LinkedList<T> is multithreaded read operations.

Constructors

LinkedList<T>()	Initializes a new instance of the LinkedList<T> class that is empty.
LinkedList<T>(IEnumerable<T>)	Initializes a new instance of the LinkedList<T> class that contains elements copied from the specified IEnumerable and has sufficient capacity to accommodate the number of elements copied.
LinkedList<T>(SerializationInfo, StreamingContext)	Initializes a new instance of the LinkedList<T> class that is serializable with the specified SerializationInfo and StreamingContext.

Properties

Count	Gets the number of nodes actually contained in the LinkedList<T>.
First	Gets the first node of the LinkedList<T>.
Last	Gets the last node of the LinkedList<T>.

Methods

AddAfter(LinkedListNode<T>, LinkedListNode<T>)	Adds the specified new node after the specified existing node in the LinkedList<T>.
AddAfter(LinkedListNode<T>, T)	Adds a new node containing the specified value after the specified existing node in the LinkedList<T>.
AddBefore(LinkedListNode<T>, LinkedListNode<T>)	Adds the specified new node before the specified existing node in the LinkedList<T>.
AddBefore(LinkedListNode<T>, T)	Adds a new node containing the specified value before the specified existing node in the LinkedList<T>.
AddFirst(LinkedListNode<T>)	Adds the specified new node at the start of the LinkedList<T>.
AddFirst(T)	Adds a new node containing the specified value at the start of the LinkedList<T>.
AddLast(LinkedListNode<T>)	Adds the specified new node at the end of the LinkedList<T>.
AddLast(T)	Adds a new node containing the specified value at the end of the LinkedList<T>.
Clear()	Removes all nodes from the LinkedList<T>.
Contains(T)	Determines whether a value is in the LinkedList<T>.
CopyTo(T[], Int32)	Copies the entire LinkedList<T> to a compatible one-dimensional Array, starting at the specified index of the target array.

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
Find(T)	Finds the first node that contains the specified value.
FindLast(T)	Finds the last node that contains the specified value.
GetEnumerator()	Returns an enumerator that iterates through the LinkedList<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize the LinkedList<T> instance.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(LinkedListNode<T>)	Removes the specified node from the LinkedList<T>.
Remove(T)	Removes the first occurrence of the specified value from the LinkedList<T>.
RemoveFirst()	Removes the node at the start of the LinkedList<T>.

RemoveLast()	Removes the node at the end of the LinkedList<T>.
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array, starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection.
ICollection<T>.Add(T)	Adds an item at the end of the ICollection<T>.
ICollection<T>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the linked list as a collection.
IEnumerable<T>.GetEnumerator()	Returns an enumerator that iterates through a collection.

Extension Methods

CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter `T` is DataRow.
CopyToDataTable<T>	Copies DataRow objects to the specified DataTable, given an input IEnumerable<T> object where

(IEnumerable<T>, DataTable, Load Option)	the generic parameter `T` is DataRow.
CopyToDataTable<T> (IEnumerable<T>, DataTable, Load Option, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable, given an input IEnumerable<T> object where the generic parameter `T` is DataRow.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
DescendantNodes<T> (IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

.NET Core

3.0, 2.2, 2.1, 2.0, 1.1, 1.0

.NET Framework

4.8, 4.7.2, 4.7.1, 4.7, 4.6.2, 4.6.1, 4.6, 4.5.2, 4.5.1, 4.5, 4.0, 3.5, 3.0, 2.0

.NET Standard

2.1, 2.0, 1.6, 1.4, 1.3, 1.2, 1.1, 1.0

UWP

10.0

Xamarin.Android

7.1

Xamarin.iOS

10.8

Xamarin.Mac

3.0

Thread Safety

This type is not thread safe. If the LinkedList<T> needs to be accessed by multiple threads, you will need to implement their own synchronization mechanism.

A LinkedList<T> can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. In the rare case where an enumeration contends with write accesses, the collection must be locked during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

See also

- LinkedListNode<T>

Is this page helpful?

Yes No