### Связный список

Основное назначение связного списка — предоставление механизма для хранения и доступа к произвольному количеству данных. Как следует из названия, это достигается связыванием данных вместе в список.

Прежде чем мы перейдем к рассмотрению связного списка, давайте вспомним, как хранятся данные в массиве.



Как показано на рисунке, данные в массиве хранятся в непрерывном участке памяти, разделенном на ячейки определенного размера. Доступ к данным в ячейках осуществляется по ссылке на их расположение — индексу.

Это отличный способ хранить данные. Большинство языков программирования позволяют так или иначе выделить память в виде массива и оперировать его содержимым. Последовательное хранение данных увеличивает производительность *(data locality)*, позволяет легко итерироваться по содержимому и получать доступ к произвольному элементу по индексу.

Тем не менее, иногда массив — не самая подходящая структура.

Предположим, что у нашей программы следующие требования:

* Прочесть некоторое количество целых чисел из источника (метод NextValue), пока не встретится число 0xFFFF.
* Передать считанные числа в метод ProcessItems

Поскольку в требованиях указано, что считанные числа передаются в метод ProcessItems за один раз, очевидным решение будет массив целых чисел:

void LoadData()

{

// Предположим, что элементов будет не более 20

int[] values = new int[20];

for (int i = 0; i < values.Length; i++)

{

if (values[i] == 0xFFFF)

{

break;

}

values[i] = NextValue();

}

ProcessItems(values);

}

void ProcessItems(int[] values)

{

// ... обработать данные.

}

У этого решения есть ряд проблем, но самая очевидная из них — что случится, если будет необходимо прочесть больше 20 значений? В данной реализации значения с 21 и далее просто проигнорируются. Можно выделить больше памяти — 200 или 2000 элементов. Можно дать пользователю возможность выбрать размер массива. Или выделить память под новый массив большего размера при заполнении старого и скопировать элементы. Но все эти решения усложняют код и бесполезно тратят память.

Нам нужна коллекция, которая позволяет добавить произвольное число элементов и перебрать их в порядке добавления. Размер коллекции должен быть неограничен, а произвольный доступ нам не нужен. Нам нужен связный список.

Прежде чем перейти к его реализации, давайте посмотрим на то, как могло бы выглядеть решение нашей задачи.

static void LoadItems()

{

LinkedList list = new LinkedList();

while (true)

{

int value = NextValue();

if (value != 0xFFFF)

{

list.Add(value);

}

else

{

break;

}

}

ProcessItems(list);

}

static void ProcessItems(LinkedList list)

{

// ... обработать данные.

}

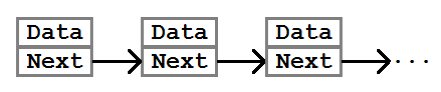
Обратите внимание: проблем, присущих первому варианту решения больше нет — мы не можем выделить недостаточно или, наоборот, слишком много памяти под массив.

Кроме того, из этого кода можно увидеть, что наш список будет принимать параметр типа <T> и реализовывать интерфейс IEnumerable

### Реализация класса LinkedList

#### Класс Node

В основе связного списка лежит понятие узла, или элемента *(Node)*. Узел — это контейнер, который позволяет хранить данные и получать следующий узел.



В самом простом случае класс Node можно реализовать так:

public class Node

{

public int Value { get; set; }

public Node Next { get; set; }

}

Теперь мы можем создать примитивный связный список. Выделим память под три узла (first, middle, last) и соединим их последовательно:

// +-----+------+

// | 3 | null +

// +-----+------+

Node first = new Node { Value = 3 };

// +-----+------+ +-----+------+

// | 3 | null + | 5 | null +

// +-----+------+ +-----+------+

Node middle = new Node { Value = 5 };

// +-----+------+ +-----+------+

// | 3 | \*---+--->| 5 | null +

// +-----+------+ +-----+------+

first.Next = middle;

// +-----+------+ +-----+------+ +-----+------+

// | 3 | \*---+--->| 5 | null + | 7 | null +

// +-----+------+ +-----+------+ +-----+------+

Node last = new Node { Value = 7 };

// +-----+------+ +-----+------+ +-----+------+

// | 3 | \*---+--->| 5 | \*---+-->| 7 | null +

// +-----+------+ +-----+------+ +-----+------+

middle.Next = last;

Теперь у нас есть список из трех элементов, начиная с first и заканчивая last. Поле Next последнего узла имеет значение null, что показывает, что это последний элемент. С этим списком уже можно производить различные операции. Например, напечатать данные из каждого элемента:

private static void PrintList(Node node)

{

while (node != null)

{

Console.WriteLine(node.Value);

node = node.Next;

}

}

Метод PrintList итерируется по элементам списка: печатает значение поля Value и переходит к следующему узлу по ссылке в поле Next.

Теперь, когда мы знаем, как должен выглядеть узел связанного списка, давайте посмотрим на пример реализации класса LinkedListNode.

public class LinkedListNode

{

///

/// Конструктор нового узла со значением Value.

///

public LinkedListNode(T value)

{

Value = value;

}

///

/// Поле Value.

///

public T Value { get; internal set; }

///

/// Ссылка на следующий узел списка (если узел последний, то null).

///

public LinkedListNode Next { get; internal set; }

}

#### Класс LinkedList

Прежде чем реализовывать наш связный список, нужно понять, как мы будем с ним работать.

Ранее мы увидели, что коллекция должна поддерживать любой тип данных, а значит, нам нужно реализовать обобщенный интерфейс.

Поскольку мы используем платформу .NET, имеет смысл реализовать наш класс таким образом, чтобы его поведение было похоже на поведение встроенных коллекций. Самый простой способ сделать это — реализовать интерфейс ICollection<T>. Заметьте, что мы реализуем ICollection<T>, а не Ilist<T>, поскольку интерфейс Ilist<T> позволяет получать доступ к элементам по индексу. Несмотря на то, что произвольный доступ к элементам в целом полезен, его невозможно эффективно реализовать в связном списке.

Учитывая все вышесказанное, давайте набросаем примерный план класса, а затем заполним недостающие методы.

public class LinkedList :

System.Collections.Generic.ICollection

{

public void Add(T item)

{

throw new System.NotImplementedException();

}

public void Clear()

{

throw new System.NotImplementedException();

}

public bool Contains(T item)

{

throw new System.NotImplementedException();

}

public void CopyTo(T[] array, int arrayIndex)

{

throw new System.NotImplementedException();

}

public int Count

{

get;

private set;

}

public bool IsReadOnly

{

get { throw new System.NotImplementedException(); }

}

public bool Remove(T item)

{

throw new System.NotImplementedException();

}

public System.Collections.Generic.IEnumerator GetEnumerator()

{

throw new System.NotImplementedException();

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

throw new System.NotImplementedException();

}

}

#### Метод Add

* **Поведение:** Добавляет элемент в конец списка.
* **Сложность:** O(1)

Добавление элемента в связный список производится в три этапа:

1. Создать экземпляр класса LinkedListNode.
2. Найти последний узел списка.
3. Установить значение поля Next последнего узла списка так, чтобы оно указывало на созданный узел.

Основная сложность заключается в том, чтобы найти последний узел списка. Можно сделать это двумя способами. Первый — сохранять указатель на первый узел списка и перебирать узлы, пока не дойдем до последнего. В этом случае нам не требуется сохранять указатель на последний узел, что позволяет использовать меньше памяти (в зависимости от размера указателя на вашей платформе), но требует прохода по всему списку при каждом добавлении узла. Это значит, что метод Add займет O(n) времени.

Второй метод заключается в сохранении указателя на последний узел списка, и тогда при добавлении нового узла мы поменяем указатель так, чтобы он указывал на новый узел. Этот способ предпочтительней, поскольку выполняется за O(1) времени.

Первое, что нам необходимо сделать — добавить два приватных поля в класс LinkedList: ссылки на первый (head) и последний (tail) узлы.

private LinkedListNode \_head;

private LinkedListNode \_tail;

Теперь мы можем добавить метод, который выполняет три необходимых шага.

public void Add(T value)

{

LinkedListNode node = new LinkedListNode(value);

if (\_head == null)

{

\_head = node;

\_tail = node;

}

else

{

\_tail.Next = node;

\_tail = node;

}

Count++;

}

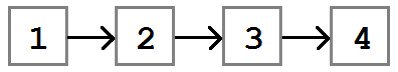
Сначала мы создаем экземпляр класса LinkedListNode. Затем проверяем, является ли список пустым. Если список пуст, мы просто устанавливаем значения полей \_head и \_tail так, чтобы они указывали на новый узел. Этот узел в данном случае будет являться одновременно и первым, и последним в списке. Если список не пуст, узел добавляется в конец списка, а поле \_tail теперь указывает на новый конец списка.

Поле Count инкрементируется при добавлении узла для того, чтобы сохранялся контракт интерфейса ICollection<T>. Поле Count возвращает точное количество элементов списка.

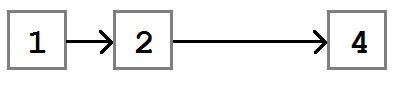
#### Метод Remove

* **Поведение:** Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.
* **Сложность:** O(n)

Прежде чем разбирать метод Remove, давайте посмотрим, чего мы хотим добиться. На следующем рисунке список с четырьмя элементами. Мы удаляем элемент со значением «3».



После удаления узла поле Next узла со значением «2» будет указывать на узел со значением «4».



Основной алгоритм удаления элемента такой:

1. Найти узел, который необходимо удалить.
2. Изменить значение поля Next предыдущего узла так, чтобы оно указывало на узел, следующий за удаляемым.

Как всегда, основная проблема кроется в мелочах. Вот некоторые из случаев, которые необходимо предусмотреть:

* Список может быть пустым, или значение, которое мы передаем в метод может не присутствовать в списке. В этом случает список останется без изменений.
* Удаляемый узел может быть единственным в списке. В этом случае мы установим значения полей \_head и \_tail равными null.
* Удаляемый узел будет в начале списка. В этом случае мы записываем в \_headссылку на следующий узел.
* Удаляемый узел будет в середине списка.
* Удаляемый узел будет в конце списка. В этом случае мы записываем в \_tail ссылку на предпоследний узел, а в его поле Next записываем null.

public bool Remove(T item)

{

LinkedListNode previous = null;

LinkedListNode current = \_head;

// 1: Пустой список: ничего не делать.

// 2: Один элемент: установить Previous = null.

// 3: Несколько элементов:

// a: Удаляемый элемент первый.

// b: Удаляемый элемент в середине или конце.

while (current != null)

{

if (current.Value.Equals(item))

{

// Узел в середине или в конце.

if (previous != null)

{

// Случай 3b.

// До: Head -> 3 -> 5 -> null

// После: Head -> 3 ------> null

previous.Next = current.Next;

// Если в конце, то меняем \_tail.

if (current.Next == null)

{

\_tail = previous;

}

}

else

{

// Случай 2 или 3a.

// До: Head -> 3 -> 5

// После: Head ------> 5

// Head -> 3 -> null

// Head ------> null

\_head = \_head.Next;

// Список теперь пустой?

if (\_head == null)

{

\_tail = null;

}

}

Count--;

return true;

}

previous = current;

current = current.Next;

}

return false;

}

Поле Count декрементируется при удалении узла.

#### Метод Contains

* **Поведение:** Возвращает true или false в зависимости от того, присутствует ли искомый элемент в списке.
* **Сложность:** O(n)

Метод Contains достаточно простой. Он просматривает каждый элемент списка, от первого до последнего, и возвращает true как только найдет узел, чье значение равно переданному параметру. Если такой узел не найден, и метод дошел до конца списка, то возвращается false.

public bool Contains(T item)

{

LinkedListNode current = \_head;

while (current != null)

{

if (current.Value.Equals(item))

{

return true;

}

current = current.Next;

}

return false;

}

#### Метод GetEnumerator

* **Поведение:** Возвращает экземпляр IEnumerator, который позволяет итерироваться по элементам списка.
* **Сложность:** Получение итератора — O(1). Проход по всем элементам — O(n).

Возвращаемый итератор проходит по всему списку от первого до последнего узла и возвращает значение каждого элемента с помощью ключевого слова yield.

IEnumerator IEnumerable.GetEnumerator()

{

LinkedListNode current = \_head;

while (current != null)

{

yield return current.Value;

current = current.Next;

}

}

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable)this).GetEnumerator();

}

#### Метод Clear

* **Поведение:** Удаляет все элементы из списка.
* **Сложность:** O(1)

Метод Clear просто устанавливает значения полей \_head и \_tail равными null. Поскольку C# — язык с автоматическим управлением памятью, нет необходимости явно удалять неиспользуемые узлы. Клиент, вызывающий метод, должен убедиться в корректном удалении значений узлов, если это необходимо.

public void Clear()

{

\_head = null;

\_tail = null;

Count = 0;

}

#### Метод CopyTo

* **Поведение:** Копирует содержимое списка в указанный массив, начиная с указанного индекса.
* **Сложность:** O(n)

Метод CopyTo проходит по списку и копирует элементы в массив с помощью присваивания. Клиент, вызывающий метод должен убедиться, что массив имеет достаточный размер для того, чтобы вместить все элементы списка.

public void CopyTo(T[] array, int arrayIndex)

{

LinkedListNode current = \_head;

while (current != null)

{

array[arrayIndex++] = current.Value;

current = current.Next;

}

}

#### Метод Count

* **Поведение:** Возвращает количество элементов списка. Возвращает 0, если список пустой.
* **Сложность:** O(1)

Count — поле с публичным геттером и приватным сеттером. Изменение его значения осуществляется в методах Add, Remove и Clear.

public int Count

{

get;

private set;

}

#### Метод IsReadOnly

* **Поведение:** Возвращает false, если список только для чтения.
* **Сложность:** O(1)

public bool IsReadOnly

{

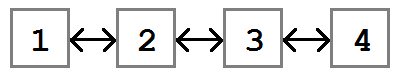
get { return false; }

}

### Двусвязный список

Связный список, который мы только что создали, называется также «односвязным». Это значит, что между узлами только одна связь в единственном направлении от первого узла к последнему. Есть также достаточно распространенный вариант списка, который предоставляет доступ к обоим концам — двусвязный список.

Для того, чтобы создать двусвязный список, мы должны добавить в класс LinkedListNode поле Previous, которое будет содержать ссылку на предыдущий элемент списка.



Далее мы рассмотрим только отличия в реализации односвязного и двусвязного списка.

#### Класс Node

Единственное изменение, которое надо внести в класс LinkedListNode — добавить поле со ссылкой на предыдущий узел.

public class LinkedListNode

{

///

/// Конструктор нового узла со значением Value.

///

///

public LinkedListNode(T value)

{

Value = value;

}

///

/// Поле Value.

///

public T Value { get; internal set; }

///

/// Ссылка на следующий узел списка (если узел последний, то null).

///

public LinkedListNode Next { get; internal set; }

///

/// Ссылка на предыдущий узел списка (если узел первый, то null).

///

public LinkedListNode Previous { get; internal set; }

}

#### Метод AddFirst

В то время, как односвязный список позволяет добавлять элементы только в конец, используя двусвязный список мы можем добавлять элементы как в начало, так и в конец, с помощью методов AddFirst и AddLast соответственно. Метод ICollection<T>.Add будет вызывать AddLast для совместимости с односвязным списком.

* **Поведение:** Добавляет переданный элемент в начало списка.
* **Сложность:** O(1)

При добавлении элемента в начало списка последовательность действий примерно такая же, как и при добавлении элемента в односвязный список.

1. Установить значение поля Next в новом узле так, чтобы оно указывало на бывший первый узел.
2. Установить значение поля Previous в бывшем первом узле так, чтобы оно указывало на новый узел.
3. Обновить поле \_tail при необходимости и инкрементировать поле Count

public void AddFirst(T value)

{

LinkedListNode node = new LinkedListNode(value);

// Сохраняем ссылку на первый элемент.

LinkedListNode temp = \_head;

// \_head указывает на новый узел.

\_head = node;

// Вставляем список позади первого элемента.

\_head.Next = temp;

if (Count == 0)

{

// Если список был пуст, то head and tail должны

// указывать на новой узел.

\_tail = \_head;

}

else

{

// До: head -------> 5 7 -> null

// После: head -> 3 5 7 -> null

temp.Previous = \_head;

}

Count++;

}

#### Метод AddLast

* **Поведение:** Добавляет переданный элемент в конец списка.
* **Сложность:** O(1)

Добавление узла в конец списка легче, чем в начало. Мы просто создаем новый узел и обновляем поля \_head и \_tail, а затем инкрементируем поле Count.

public void AddLast(T value)

{

LinkedListNode node = new LinkedListNode(value);

if (Count == 0)

{

\_head = node;

}

else

{

\_tail.Next = node;

// До: Head -> 3 5 -> null

// После:Head -> 3 5 7 -> null

// 7.Previous = 5

node.Previous = \_tail;

}

\_tail = node;

Count++;

}

Как было сказано ранее, ICollection<T>.Add просто зовет AddLast.

public void Add(T value)

{

AddLast(value);

}

#### Метод RemoveFirst

Как и метод Add, Remove будет разделен на два метода, позволяющих удалять элементы из начала и из конца списка. Метод ICollection<T>.Remove будет также удалять элементы из начала, но теперь будет еще обновлять поля Previous в тех узлах, где это необходимо.

* **Поведение:** Удаляет первый элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.
* **Сложность:** O(1)

RemoveFirst устанавливает ссылку head на второй узел списка и обнуляет поле Previous этого узла, удаляя таким образом все ссылки на предыдущий первый узел. Если список был пуст или содержал только один элемент, то поля \_head и \_tailстановятся равны null.

public void RemoveFirst()

{

if (Count != 0)

{

// До: Head -> 3 5

// После: Head -------> 5

// Head -> 3 -> null

// Head ------> null

\_head = \_head.Next;

Count--;

if (Count == 0)

{

\_tail = null;

}

else

{

// 5.Previous было 3; теперь null.

\_head.Previous = null;

}

}

}

#### Метод RemoveLast

* **Поведение:** Удаляет последний элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.
* **Сложность:** O(1)

RemoveLast устанавливает значение поля \_tail так, чтобы оно указывало на предпоследний элемент списка и, таким образом, удаляет последний элемент. Если список был пустым, или содержал только один элемент, то поля \_head и \_tailстановятся равны null.

public void RemoveLast()

{

if (Count != 0)

{

if (Count == 1)

{

\_head = null;

\_tail = null;

}

else

{

// До: Head --> 3 --> 5 --> 7

// Tail = 7

// После: Head --> 3 --> 5 --> null

// Tail = 5

// Обнуляем 5.Next

\_tail.Previous.Next = null;

\_tail = \_tail.Previous;

}

Count--;

}

}

#### Метод Remove

* **Поведение:** Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.
* **Сложность:** O(n)

Метод ICollection<T>.Remove() почти такой же, как и в односвязном списке. Единственное отличие — теперь нам необходимо поменять значение поля Previous при удалении узла. Для того, чтобы не повторять код, этот метод зовет RemoveFirst при удалении первого узла.

public bool Remove(T item)

{

LinkedListNode previous = null;

LinkedListNode current = \_head;

// 1: Пустой список: ничего не делать.

// 2: Один элемент: установить Previous = null.

// 3: Несколько элементов:

// a: Удаляемый элемент первый.

// b: Удаляемый элемент в середине или конце.

while (current != null)

{

// Head -> 3 -> 5 -> 7 -> null

// Head -> 3 ------> 7 -> null

if (current.Value.Equals(item))

{

// Узел в середине или в конце.

if (previous != null)

{

// Случай 3b.

previous.Next = current.Next;

// Если в конце, то меняем \_tail.

if (current.Next == null)

{

\_tail = previous;

}

else

{

// До: Head -> 3 5 7 -> null

// После: Head -> 3 7 -> null

// previous = 3

// current = 5

// current.Next = 7

// Значит... 7.Previous = 3

current.Next.Previous = previous;

}

Count--;

}

else

{

// Случай 2 или 3a.

RemoveFirst();

}

return true;

}

previous = current;

current = current.Next;

}

return false;

}

#### Зачем нужен двусвязный список?

Итак, мы можем добавлять элементы в начало списка и в его конец. Что нам это дает? В том виде, в котором он реализован сейчас, нет особых преимуществ перед обычным односвязным списком. Но если добавить геттеры для полей head и tail, пользователь нашего списка сможет реализовать множество различных алгоритмов.

public LinkedListNode Head

{

get

{

return \_head;

}

}

public LinkedListNode Tail

{

get

{

return \_tail;

}

}

Так мы сможем итерироваться по списку вручную, в том числе от последнего элемента к первому.

В этом примере мы используем поля Tail и Previous для того, чтобы обойти список задом наперед.

public void ProcessListBackwards()

{

LinkedList list = new LinkedList();

PopulateList(list);

LinkedListNode current = list.Tail;

while (current != null)

{

ProcessNode(current);

current = current.Previous;

}

}

Кроме того, двусвязный список позволяет легко реализовать двусвязную очередь, которая, в свою очередь, является строительным блоком для других структур данных. Мы вернемся к ней позже, в соответствующей части.