**Зв'язний список**

Основне призначення зв'язного списку - надання механізму для зберігання і доступу до довільного кількості даних. Як випливає з назви, це досягається зв'язуванням даних разом в список.

Перш ніж ми перейдемо до розгляду зв'язного списку, давайте згадаємо, як зберігаються дані в масиві.



Як показано на малюнку, дані в масиві зберігаються в безперервному ділянці пам'яті, розділеному на осередки певного розміру. Доступ до даних в осередках здійснюється за посиланням на їх розташування - індексу.

Це відмінний спосіб зберігати дані. Більшість мов програмування дозволяють так чи інакше виділити пам'ять у вигляді масиву і оперувати його вмістом. Послідовне зберігання даних збільшує продуктивність (data locality), дозволяє легко ітерованих у вмісті і отримувати доступ до довільного елементу за індексом.

Проте, іноді масив - не найкраща структура.

Припустимо, що у нашої програми наступні вимоги:

• Прочитати кілька цілих чисел з джерела (метод NextValue), поки не зустрінеться число 0xFFFF.

• Надіслати лічені числа в метод ProcessItems

Оскільки в вимогах зазначено, що лічені числа передаються в метод ProcessItems за один раз, очевидним рішення буде масив цілих чисел:

void LoadData()

{

// Припустимо, що елементів буде не більше 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)

{

// ... обробити дані.}

У цього рішення є ряд проблем, але найочевидніша з них - що трапиться, якщо буде необхідно прочитати більше 20 значень? У даній реалізації значення з 21 і далі просто проігнорують. Можна виділити більше пам'яті - 200 або 2000 елементів. Можна дати користувачеві можливість вибрати розмір масиву. Або виділити пам'ять під новий масив більшого розміру при заповненні старого і скопіювати елементи. Але всі ці рішення ускладнюють код і марно витрачають пам'ять.

Нам потрібна колекція, яка дозволяє додати довільне число елементів і перебрати їх в порядку додавання. Розмір колекції повинен бути необмежений, а довільний доступ нам не потрібен. Нам потрібен зв'язний список.

Перш ніж перейти до його реалізації, давайте подивимося на те, як могло б виглядати рішення нашої задачі.

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)

{

// ... обробити дані.

}

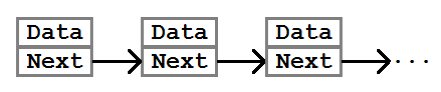
Зверніть увагу: проблем, властивих першим варіантом вирішення більше немає - ми не можемо виділити недостатньо або, навпаки, занадто багато пам'яті під масив.

Крім того, з цього коду можна побачити, що наш список буде приймати параметр типу <T> і реалізовувати інтерфейс IEnumerable

### Реалізація класу LinkedList

#### Клас Node

В основі зв'язного списку лежить поняття вузла, або елемента (Node). Вузол - це контейнер, який дозволяє зберігати дані і отримувати наступний вузол.



У найпростішому випадку клас Node можна реалізувати так:

public class Node

{

public int Value { get; set; }

public Node Next { get; set; }

}

Тепер ми можемо створити примітивний зв'язний список. Виділимо пам'ять під три вузла (first, middle, last) і з'єднаємо їх послідовно:

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

// | 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;

Тепер у нас є список з трьох елементів, починаючи з first і закінчуючи last. Поле Next останнього вузла має значення null, що показує, що це останній елемент. З цим списком вже можна робити різні операції. Наприклад, надрукувати дані з кожного елемента:

private static void PrintList(Node node)

{

while (node != null)

{

Console.WriteLine(node.Value);

node = node.Next;

}

}

Метод PrintList ітеріруется за елементами списку: друкує значення поля Value і переходить до наступного вузла за посиланням в поле Next.

Тепер, коли ми знаємо, як повинен виглядати вузол пов'язаного списку, давайте подивимося на приклад реалізації класу LinkedListNode.

public class LinkedListNode

{

///

/// Конструктор нового вузла зі значенням Value.

///

public LinkedListNode(T value)

{

Value = value;

}

///

/// Поле Value.

///

public T Value { get; internal set; }

///

/// Посилання на наступний вузол списку (якщо вузол останній, то null).

///

public LinkedListNode Next { get; internal set; }

}

#### Клас LinkedList

Перш ніж реалізовувати наш зв'язний список, потрібно зрозуміти, як ми будемо з ним працювати.

Раніше ми побачили, що колекція повинна підтримувати будь-який тип даних, а значить, нам потрібно реалізувати узагальнений інтерфейс.

Оскільки ми використовуємо платформу .NET, має сенс реалізувати наш клас таким чином, щоб його поведінка була схоже на поведінку вбудованих колекцій. Найпростіший спосіб зробити це - реалізувати інтерфейс ICollection <T>. Зауважте, що ми реалізуємо ICollection <T>, а не Ilist <T>, оскільки інтерфейс Ilist <T> дозволяє отримувати доступ до елементів за індексом. Незважаючи на те, що довільний доступ до елементів в цілому корисний, його неможливо ефективно реалізувати в зв'язковому списку.

З огляду на все вищесказане, давайте накидаємо приблизний план класу, а потім заповнимо відсутні методи.

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

* **Поведінка: Додає елемент в кінець списку**.
* **Складність:** O(1)

Додавання елемента в зв'язний список проводиться в три етапи:

1. Створити екземпляр класу LinkedListNode.

2. Знайти останній вузол списку.

3. Встановити значення поля Next останнього вузла списку так, щоб воно вказувало на створений вузол.

Основна складність полягає в тому, щоб знайти останній вузол списку. Можна зробити це двома способами. Перший - зберігати покажчик на перший вузол списку і перебирати вузли, поки не дійдемо до останнього. У цьому випадку нам не потрібно зберігати покажчик на останній вузол, що дозволяє використовувати менше пам'яті (в залежності від розміру покажчика на вашій платформі), але вимагає проходу по всьому списку при кожному додаванні вузла. Це означає, що метод Add займе O (n) часу.

Другий метод полягає в збереженні покажчика на останній вузол списку, і тоді при додаванні нового вузла ми поміняємо покажчик так, щоб він вказував на новий вузол. Цей спосіб краще, оскільки виконується за O (1) часу.

Перше, що нам необхідно зробити - додати два приватних поля в клас LinkedList: посилання на перший (head) і останній (tail) вузли.

private LinkedListNode \_head;

private LinkedListNode \_tail;

Тепер ми можемо додати метод, який виконує три необхідних кроку.

public void Add(T value)

{

LinkedListNode node = new LinkedListNode(value);

if (\_head == null)

{

\_head = node;

\_tail = node;

}

else

{

\_tail.Next = node;

\_tail = node;

}

Count++;

}

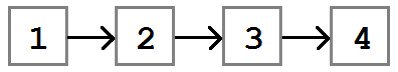
Спочатку ми створюємо екземпляр класу LinkedListNode. Потім перевіряємо, чи є список порожнім. Якщо список порожній, ми просто встановлюємо значення полів \_head і \_tail так, щоб вони вказували на новий вузол. Цей вузол в даному випадку буде одночасно і першим, і останнім у списку. Якщо у списку ще порожній, вузол додається в кінець списку, а поле \_tail тепер вказує на новий кінець списку.

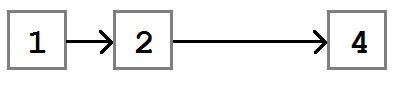
Поле Count инкрементируется при додаванні вузла для того, щоб зберігався контракт інтерфейсу ICollection <T>. Поле Count повертає точну кількість елементів списку.

#### Метод Remove

* **Поведінка**: Видаляє перший елемент списку зі значенням, рівним переданому. Повертає true, якщо елемент був вилучений і false в іншому випадку.
* **Складність:** O(n)

Перш ніж розбирати метод Remove, давайте подивимося, чого ми хочемо досягти. На наступному малюнку список з чотирма елементами. Ми видаляємо елемент зі значенням «3».



Після видалення вузла поле Next вузла зі значенням «2» буде вказувати на вузол із значенням «4».

Основний алгоритм видалення елемента такої:

1. Знайти вузол, який необхідно видалити.
2. Змінити значення поля Next попереднього вузла так, щоб воно вказувало на вузол, наступний за видаляється.

Як завжди, основна проблема криється в дрібницях. Ось деякі з випадків, які необхідно передбачити:

* Список може бути порожнім, або значення, яке ми передаємо в метод може не бути присутнім в списку. В цьому випадку список залишиться без змін.
* Видаляється вузол може бути єдиним в списку. В цьому випадку ми встановимо значення полів \_head і \_tail рівними null.
* Видаляється вузол буде на початку списку. В цьому випадку ми записуємо в \_headссилку на наступний вузол.
* Видаляється вузол буде в середині списку.
* • видаляється вузол буде в кінці списку. В цьому випадку ми записуємо в \_tail посилання на передостанній вузол, а в його поле Next записуємо null.

public bool Remove(T item)

{

LinkedListNode previous = null;

LinkedListNode current = \_head;

// 1: Порожній список: нічого не робити.

// 2: Один елемент: встановити Previous = null.

// 3: кілька елементів:

// a: Видаляється елемент перший.

// b: Видаляється елемент в середині або кінці.

while (current != null)

{

if (current.Value.Equals(item))

{

// Вузол в середині або в кінці.

if (previous != null)

{

// Случай 3b.

// До: Head -> 3 -> 5 -> null

// Після: Head -> 3 ------> null

previous.Next = current.Next;

// Якщо в кінці, то міняємо \_tail.

if (current.Next == null)

{

\_tail = previous;

}

}

else

{

// Випадок 2 або 3a.

// До: Head -> 3 -> 5

// Після: Head ------> 5

// Head -> 3 -> null

// Head ------> null

\_head = \_head.Next;

// Список тепер порожній?

if (\_head == null)

{

\_tail = null;

}

}

Count--;

return true;

}

previous = current;

current = current.Next;

}

return false;

}

Поле Count декрементируется при видаленні вузла.

#### Метод Contains

* **Поведінка:** Повертає true або false залежно від того, чи присутній шуканий елемент в списку.
* **Складність:** O(n)

Метод Contains досить простий. Він переглядає кожен елемент списку, від першого до останнього, і повертає true як тільки знайде вузол, чиє значення дорівнює переданому параметру. Якщо такий вузол не знайдений, і метод дійшов до кінця списку, то повертається 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

* **Поведінка:** Повертає екземпляр IEnumerator, який дозволяє ітерованих за елементами списку.
* **Складність:** Отримання ітератора - O (1). Прохід по всіх елементах— O(n).

Повертається итератор проходить по всьому списку від першого до останнього вузла і повертає значення кожного елемента за допомогою ключового 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

* **Поведінка:** Видаляє всі елементи зі списку.
* **Складність:** O(1)

Метод Clear просто встановлює значення полів \_head і \_tail рівними null. Оскільки C # - мова з автоматичним управлінням пам'яттю, немає необхідності явно видаляти невикористовувані вузли. Клієнт, що викликає метод, повинен переконатися в коректному видаленні значень вузлів, якщо це необхідно.

public void Clear()

{

\_head = null;

\_tail = null;

Count = 0;

}

#### Метод CopyTo

* **Поведінка:** Копирует содержимое списка в указанный массив, начиная с указанного индекса.
* **Складність:** 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

* **Поведінка:** Повертає кількість елементів списку. Повертає 0, якщо список порожній.
* **Складність:** O(1)

Count — поле з публічним геттером і приватним сетером. Зміна його значення здійснюється в методах Add, Remove і Clear.

public int Count

{

get;

private set;

}

#### Метод IsReadOnly

* **Поведінка:** Повертає false, якщо список тільки для читання.
* **Складність:** O(1)

public bool IsReadOnly

{

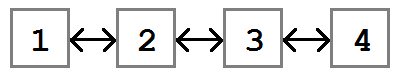
get { return false; }

}

**Двусвязний список**

Зв'язний список, який ми тільки що створили, називається також «однозв'язного». Це означає, що між вузлами тільки одна зв'язок в єдиному напрямку від першого вузла до останнього. Є також досить поширений варіант списку, який надає доступ до обох кінців - двусвязний список.

Для того, щоб створити двусвязний список, ми повинні додати в клас LinkedListNode поле Previous, яке буде містити посилання на попередній елемент списку.



Далі ми розглянемо тільки відмінності в реалізації однозв'язного і двусвязного списку.

#### Клас Node

Єдина зміна, яку треба внести в клас LinkedListNode - додати поле з посиланням на попередній вузол.

public class LinkedListNode

{

///

/// Конструктор нового вузла зі значенням Value.

///

///

public LinkedListNode(T value)

{

Value = value;

}

///

/// Поле Value.

///

public T Value { get; internal set; }

///

/// Посилання на наступний вузол списку (якщо вузол останній, то null).

///

public LinkedListNode Next { get; internal set; }

///

/// Посилання на попередній вузол списку (якщо вузол перший, то null).

///

public LinkedListNode Previous { get; internal set; }

}

#### Метод AddFirst

У той час, як однозв'язний список дозволяє додавати елементи тільки в кінець, використовуючи двусвязний список ми можемо додавати елементи як на початок, так і в кінець, за допомогою методів AddFirst і AddLast відповідно. Метод ICollection <T> .Add викликатиме AddLast для сумісності з однозв'язного списком.

* **Поведінка:** Додає переданий елемент в початок списку.
* **Складність:** O(1)

При додаванні елемента в початок списку послідовність дій приблизно така ж, як і при додаванні елемента в однозв'язний список.

1. Встановити значення поля Next в новому вузлі так, щоб воно вказувало на колишній перший вузол.
2. Встановити значення поля Previous в колишньому першому вузлі так, щоб воно вказувало на новий вузол.
3. Оновити поле \_tail при необхідності і інкрементіровать поле Count

public void AddFirst(T value)

{

LinkedListNode node = new LinkedListNode(value);

// Зберігаємо посилання на перший елемент.

LinkedListNode temp = \_head;

// \_head вказує на новий вузол.

\_head = node;

// Вставляємо список позаду першого елемента.

\_head.Next = temp;

if (Count == 0)

{

// Якщо список був порожній, то head and tail повинні

// вказувати на новій вузол.

\_tail = \_head;

}

else

{

// До: head -------> 5 7 -> null

// Після: head -> 3 5 7 -> null

temp.Previous = \_head;

}

Count++;

}

#### Метод AddLast

* **Поведінка:** Додає переданий елемент в кінець списку.
* **Складність:** O(1)

Додавання вузла в кінець списку легше, ніж на початок. Ми просто створюємо новий вузол і оновлюємо поля \_head і \_tail, а потім інкрементіруем 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

// Після:Head -> 3 5 7 -> null

// 7.Previous = 5

node.Previous = \_tail;

}

\_tail = node;

Count++;

}

Як було сказано раніше, ICollection <T> .Add просто кличе AddLast.

public void Add(T value)

{

AddLast(value);

}

#### Метод RemoveFirst

Як і метод Add, Remove буде розділений на два методи, що дозволяють видаляти елементи з початку і з кінця списку. Метод ICollection <T> .Remove буде також видаляти елементи з початку, але тепер буде ще оновлювати поля Previous в тих вузлах, де це необхідно.

* **Поведінка:** Видаляє перший елемент списку. Якщо список порожній, не робить нічого. Повертає true, якщо елемент був вилучений і false в іншому випадку.
* **Складність:** O(1)

RemoveFirst встановлює посилання head на другий вузол списку і обнуляє поле Previous цього вузла, видаляючи таким чином всі посилання на попередній перший вузол. Якщо список був порожній або містив тільки один елемент, то поля \_head і \_tailстановятся рівні null.

public void RemoveFirst()

{

if (Count != 0)

{

// До: Head -> 3 5

// Після: 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

* **Поведінка:** Видаляє останній елемент списку. Якщо список порожній, не робить нічого. Повертає true, якщо елемент був вилучений і false в іншому випадку.
* **Складність:** O(1)

RemoveLast встановлює значення поля \_tail так, щоб воно вказувало на передостанній елемент списку і, таким чином, видаляє останній елемент. Якщо список був порожнім, або містив тільки один елемент, то поля \_head і \_tailстановятся рівні null.

public void RemoveLast()

{

if (Count != 0)

{

if (Count == 1)

{

\_head = null;

\_tail = null;

}

else

{

// До: Head --> 3 --> 5 --> 7

// Tail = 7

// Після: Head --> 3 --> 5 --> null

// Tail = 5

// Обнуляем 5.Next

\_tail.Previous.Next = null;

\_tail = \_tail.Previous;

}

Count--;

}

}

#### Метод Remove

* **Поведінка:** Видаляє перший елемент списку зі значенням, рівним переданому. Повертає true, якщо елемент був вилучений і false в іншому випадку.
* **Складність:** O(n)

Метод ICollection <T> .Remove () майже такий же, як і в однозв'язного списку. Єдина відмінність - тепер нам необхідно поміняти значення поля Previous при видаленні вузла. Для того, щоб не повторювати код, цей метод кличе RemoveFirst при видаленні першого вузла.

public bool Remove(T item)

{

LinkedListNode previous = null;

LinkedListNode current = \_head;

// 1: Порожній список: нічого не робити.

// 2: Один елемент: встановити Previous = null.

// 3: кілька елементів:

// a: Видаляється елемент перший.

// b: Видаляється елемент в середині або кінці.

while (current != null)

{

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

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

if (current.Value.Equals(item))

{

// Вузол в середині або в кінці.

if (previous != null)

{

// Случай 3b.

previous.Next = current.Next;

// Якщо в кінці, то міняємо \_tail.

if (current.Next == null)

{

\_tail = previous;

}

else

{

// До: Head -> 3 5 7 -> null

// Після: Head -> 3 7 -> null

// previous = 3

// current = 5

// current.Next = 7

// Означає... 7.Previous = 3

current.Next.Previous = previous;

}

Count--;

}

else

{

// Случай 2 или 3a.

RemoveFirst();

}

return true;

}

previous = current;

current = current.Next;

}

return false;

}

#### Навіщо потрібен двусвязний список?

Отже, ми можемо додавати елементи в початок списку і в його кінець. Що нам це дає? У тому вигляді, в якому він реалізований зараз, немає особливих переваг перед звичайним однозв'язного списком. Але якщо додати геттери для полів head і tail, користувач нашого списку зможе реалізувати безліч різних алгоритмів.

public LinkedListNode Head

{

get

{

return \_head;

}

}

public LinkedListNode Tail

{

get

{

return \_tail;

}

}

Так ми зможемо ітерованих за списком вручну, в тому числі від Післяднего елемента до першого.

У цьому прикладі ми використовуємо поля Tail і Previous для того, щоб обійти список задом наперед.

public void ProcessListBackwards()

{

LinkedList list = new LinkedList();

PopulateList(list);

LinkedListNode current = list.Tail;

while (current != null)

{

ProcessNode(current);

current = current.Previous;

}

}

Крім того, двусвязний список дозволяє легко реалізувати ДЕК, яка, в свою чергу, є будівельним блоком для інших структур даних. Ми повернемося до неї пізніше, у відповідній частині.