public class LinkedList<E>  
 extends AbstractSequentialList<E>  
 implements List<E>, Deque<E>, Cloneable, java.io.Serializable  
{

|  |  |
| --- | --- |
| **Методы Linked*List*** | |
| Boolean aList.**removeAll**(aList) | Также удаляет все элементы из списка |
| Collections.sort(aList) | Отсортировать список |
| Collections.sort(aList,Collections.reverseOrder()) | Отсортировать в обратном порядке |
| String[] arr = (String[])aList.toArray(new String[aList.size()]) | Преобразование списка в массив |
| ArrayList < String> aList= new ArrayList < String>(Arrays.asList(array)) | Преобразование массива в список |
| Boolean isEmpty() | Проверить, пуст ли список. Возвращаете true or false |
| Boolean retainAll(Collection<?> c) | Оставляет в списке только те элементы, которые входят в список-аргумент |
| E getFirst() | Возвращает первый элемент списка |
| E getLast() | Возвращает последний элемент списка |
| E removeFirst() | Удаляет и возвращает первый элемент списка |
| E removeLast() | Удаляет и возвращает последний элемент списка |
| Void addFirst(E e) | Добавляет элемент в самое начала списка |
| Void addLast(E e) | Добавляет элемент в конец списка |
| Void clear() | Удаляет все элементы коллекции |
| Boolean contains(Object o) | Возвращает true/false при наличии/отсутствии указанного объекта в коллекции |
| Int size() | Возвращает размер коллекции |
| Boolean add(E e) | Добавляет указанный объект в конец коллекции. Возвращает true. |
| Boolean remove(Object o) | Удаляет первое появление указанного объекта. Возвращает true если список такой объект содержал. |
| Boolean addAll(Collection < ? extends E > c) | Добавляет содержимое указанной коллекции в конец списка. Возвращает true если список был изменен в результате вызова метода |
| Boolean addAll(int index, Collection < ? extends E > c) | Добавляет содержимое указанной коллекции в список. Index – позиция первого элемента коллекции. Индексы элементов справа от добавленных меняются соответственно. Возвращает true если список был изменен в результате вызова метода |
| E get(int index) | Возвращает элемент по индексу |
| E set(int index, E element) | Заменяет элемент по указанному индексу на указанный элемент. Возвращает старый элемент. |
| Void add(int index, E element) | Добавляет элемент в указанный индекс, смещщая все последующие элементы вправо |
| E remove(int index) | Удаляет элемент по указанному индексу, сдвигая последующие элементы влево. Возвращает удаляемый элемент. |
| Int indexOf(Object o) | Возвращает индекс первого такого элемента-аргумента. Если такого елемента нет в коллекции – возвращает -1 |
| int lastIndexOf(Object o) | Если в списке есть повторяющиеся элементы, то метод возвращает индекс последнего из них. Если такого элемента нет вообще, то возвращает -1 |
| E peek() | Возвращает, но не удаляет из списка первый его элемент. Если список пуст – возвращает null. |
| E element() | Возвращает, но не удаляет из списка первый его элемент. Если список пуст – выбрасывает исключение |
| E poll() | Возвращает и удаляет из списка первый его элемент. Если список пуст – возвращает null. |
| E remove() | Возвращает и удаляет из списка первый его элемент. Если список пуст – выбрасывает исключение |
| Boolean offer(E e) | Добавляет указанный элемент в конец списка. Возвращает true при изменении списка. |
| Boolean offerFirst(E e) | Добавляет указанный элемент в начало списка. Возвращает true при изменении списка. |
| Boolean offerLast(E e) | Добавляет указанный элемент в конец списка. Возвращает true при изменении списка. |
| E peekFirst() | Возвращает, но не удаляет из списка первый его элемент. Если список пуст – возвращает null. |
| E peekLast() | Возвращает, но не удаляет из списка последний его элемент. Если список пуст – возвращает null. |
| E pollFirst() | Возвращает и удаляет из списка первый его элемент. Если список пуст – возвращает null. |
| E pollLast() | Возвращает и удаляет из списка посдедний его элемент. Если список пуст – возвращает null. |
| Void push(E e) | Добавляет элемент в самое начала списка |
| E pop() | Удаляет и возвращает первый элемент списка. Если список пуст – бросает исключение |
| Boolean removeFirstOccurrence(Object o) | Удаляет первое появление указанного элемента в списке. Возвращается true. Если такого в списке нет – список не меняется – возвращается false. |
| Boolean removeLastOccurrence(Object o) | Удаляет последнее появление указанного элемента в списке. Возвращается true. Если такого в списке нет – список не меняется – возвращается false. |
| Object clone() | Возвращает копию самой сущности списка. Элементы в нем не копируются |
| Object[] toArray() | Возвращает массив из всех элементов по порядку в списке. |
|  |  |

В LinkedList элементы фактически представляют собой звенья одной цепи. У каждого элемента помимо тех данных, которые он хранит, имеется **ссылка на предыдущий и следующий элемент**. По этим ссылкам можно переходить от одного элемента к другому. Создается он так:

**public** **class** Main {

**public** **static** **void** main(java.lang.String[] args) {

String str1 = **new** String("Hello World!");

String str2 = **new** String("My name is Earl");

String str3 = **new** String("I love Java");

String str4 = **new** String("I live in Moscow");

LinkedList<String> earlBio = **new** LinkedList<>();

earlBio.add(str1);

earlBio.add(str2);

earlBio.add(str3);

earlBio.add(str4);

System.out.println(earlBio);

}

}

**Вывод:**

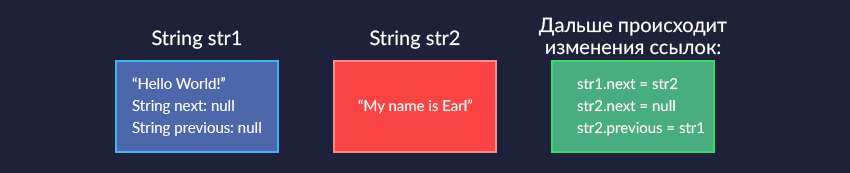
***[Hello World! My name is Earl, I love Java, I live in Moscow]***

Вот так будет выглядеть строение нашего списка:



Давай посмотрим, как производится добавление нового элемента. Это делается с помощью метода **add().**

earlBio.add(str2);

На момент этой строчки кода наш список состоит из одного элемента — строки str1. Посмотрим, что происходит дальше на картинке: 

В результате str2 и str1 становятся связанными через хранящиеся в них ссылки **next** и **previous**:



Теперь тебе должна стать понятной главная идея двусвязного списка. Элементы LinkedList являются единым списком именно благодаря вот этой цепочке ссылок. Внутри LinkedList нет массива, как в ArrayList, или чего-то похожего. Вся работа с [ArrayList](https://javarush.ru/quests/lectures/questsyntax.level07.lecture05) (по большому счету) сводится к работе с внутренним массивом.

**Вся работа с LinkedList сводится к изменению ссылок.**

Это очень хорошо видно на примере добавления элемента в середину списка:

**public** **class** Main {

**public** **static** **void** main(java.lang.String[] args) {

String str1 = **new** String("Hello World!");

String str2 = **new** String("My name is Earl");

String str3 = **new** String("I love Java");

String str4 = **new** String("I live in Moscow");

LinkedList<String> earlBio = **new** LinkedList<>();

earlBio.add(str1);

earlBio.add(str3);

earlBio.add(1, str2);

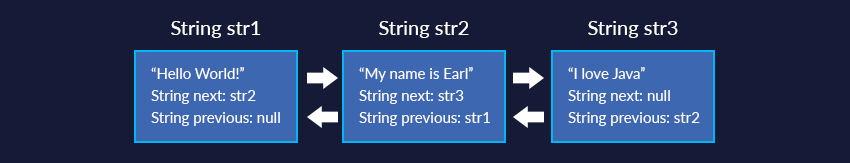
System.out.println(earlBio);

}

}

Как видишь, перегруженный метод **add()** позволяет указать конкретный индекс для нового элемента. В данном случае мы хотим добавить строку str2 между str1 и str3. Вот что будет происходить внутри:



И в результате изменения внутренних ссылок элемент str2 успешно добавлен в

список: Теперь все 3 элемента связаны. От первого элемента по цепочке next можно дойти до последнего и обратно. Со вставкой мы более-менее разобрались, а что с удалением элементов? Принцип работы тот же. Мы просто переопределяем ссылки у двух элементов “по бокам” от удаляемого:

**public** **class** Main {

**public** **static** **void** main(java.lang.String[] args) {

String str1 = **new** String("Hello World!");

String str2 = **new** String("My name is Earl");

String str3 = **new** String("I love Java");

String str4 = **new** String("I live in Moscow");

LinkedList<String> earlBio = **new** LinkedList<>();

earlBio.add(str1);

earlBio.add(str3);

earlBio.add(1, str2);

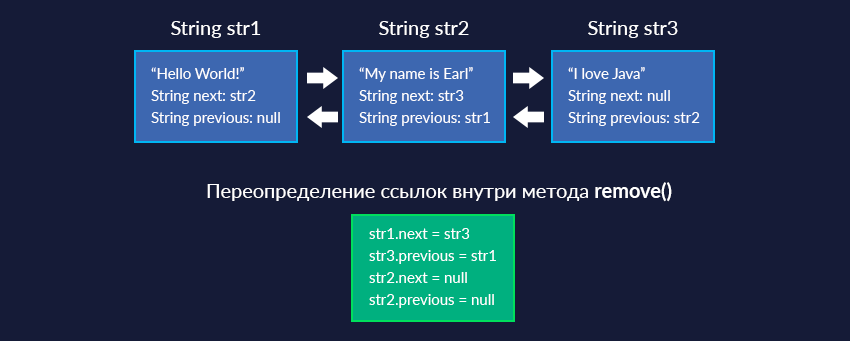
earlBio.remove(1);

System.out.println(earlBio);

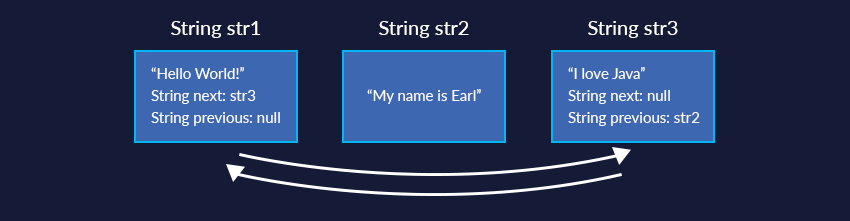
}

}

Вот что будет происходить, если мы удалим элемент с индексом 1 (он находится посередине списка):



После переопределения ссылок мы получаем нужный результат:



В отличие от удаления в ArrayList здесь нет никаких сдвигов элементов массива и тому подобного. Мы просто переопределяем ссылки у элементов str1 и str3. Теперь они указывают друг на друга, а объект str2 “выпал” из этой цепочки ссылок, и больше не является частью списка.

## Обзор методов

У LinkedList есть много общих с ArrayList методов. Например, такие методы как **add()**, **remove()**, **indexOf()**, **clear()**, **contains()** (содержится ли элемент в списке), **set()** (вставка элемента с заменой) и **size()** есть в обоих классах. Хотя (как мы выяснили на примере add() и remove()) внутри многие из них работают по другому, но в конечном итоге они делают то же самое.

Однако, у LinkedList есть отдельные методы для работы с началом и концом списка, которых нет в ArrayList:

* **addFirst()**, **addLast()**: методы для добавления элемента в начало/конец списка

**public** **class** Car {

String model;

**public** Car(String model) {

**this**.model = model;

}

**public** **static** **void** main(String[] args) {

LinkedList<Car> cars = **new** LinkedList<>();

Car ferrari = **new** Car("Ferrari 360 Spider");

Car bugatti = **new** Car("Bugatti Veyron");

Car lambo = **new** Car("Lamborghini Diablo");

Car ford = **new** Car("Ford Modneo");

Car fiat = **new** Car("Fiat Ducato");

cars.add(ferrari);

cars.add(bugatti);

cars.add(lambo);

System.out.println(cars);

cars.addFirst(ford);

cars.addLast(fiat);

System.out.println(cars);

}

@Override

**public** String toString() {

**return** "Car{" +

"model='" + model + '\'' +

'}';

}

}

**Вывод:**

***[Car{model='Ferrari 360 Spider'}, Car{model='Bugatti Veyron'}, Car{model='Lamborghini Diablo'}]***

***[Car{model='Ford Modneo'}, Car{model='Ferrari 360 Spider'}, Car{model='Bugatti Veyron'}, Car{model='Lamborghini Diablo'}, Car{model='Fiat Ducato'}]***

В итоге “Форд” оказался в начале списка, а “Фиат” — в конце.

* **peekFirst()**, **peekLast()**: возвращают первый/последний элемент списка. Возвращают null, если список пуст.

**public** **static** **void** main(String[] args) {

LinkedList<Car> cars = **new** LinkedList<>();

Car ferrari = **new** Car("Ferrari 360 Spider");

Car bugatti = **new** Car("Bugatti Veyron");

Car lambo = **new** Car("Lamborghini Diablo");

cars.add(ferrari);

cars.add(bugatti);

cars.add(lambo);

System.out.println(cars.peekFirst());

System.out.println(cars.peekLast());

}

**Вывод:**

***Car{model='Ferrari 360 Spider'} Car{model='Lamborghini Diablo'}***

* **pollFirst()**, **pollLast()**: возвращают первый/последний элемент списка **и удаляют его из списка**. Возвращают null, если список пуст

**public** **static** **void** main(String[] args) {

LinkedList<Car> cars = **new** LinkedList<>();

Car ferrari = **new** Car("Ferrari 360 Spider");

Car bugatti = **new** Car("Bugatti Veyron");

Car lambo = **new** Car("Lamborghini Diablo");

cars.add(ferrari);

cars.add(bugatti);

cars.add(lambo);

System.out.println(cars.pollFirst());

System.out.println(cars.pollLast());

System.out.println("Что осталось в списке?");

System.out.println(cars);

}

**Вывод:**

***Car{model='Ferrari 360 Spider'}***

***Car{model='Lamborghini Diablo'}***

***Что осталось в списке?***

***[Car{model='Bugatti Veyron'}]***

* **toArray()**: возвращает массив из элементов списка

**public** **static** **void** main(String[] args) {

LinkedList<Car> cars = **new** LinkedList<>();

Car ferrari = **new** Car("Ferrari 360 Spider");

Car bugatti = **new** Car("Bugatti Veyron");

Car lambo = **new** Car("Lamborghini Diablo");

cars.add(ferrari);

cars.add(bugatti);

cars.add(lambo);

Car[] carsArray = cars.toArray(**new** Car[3]);

System.out.println(Arrays.toString(carsArray));

}

**Вывод:**

**[Car{model='Ferrari 360 Spider'}, Car{model='Bugatti Veyron'}, Car{model='Lamborghini Diablo'}]**

Теперь мы знаем, как устроен LinkedList и чем он отличается от ArrayList. В чем же заключаются выгоды от использования LinkedList? Прежде всего, **в работе с серединой списка**. Вставка и удаление в середину LinkedList устроены гораздо проще, чем в ArrayList. Мы просто переопределяем ссылки соседних элементов, а ненужный элемент “выпадает” из цепочки ссылок. В то время как в ArrayList мы:

* проверяем, хватает ли места (при вставке)
* если не хватает — создаем новый массив и копируем туда данные (при вставке)
* удаляем/вставляем элемент, и сдвигаем все остальные элементы вправо/влево (в зависимости от типа операции). Причем сложность этого процесса сильно зависит от размера списка. Одно дело — скопировать/сдвинуть 10 элементов, и совсем другое — сделать то же самое с миллионом элементов.

То есть, если в твоей программе чаще происходят операции вставки/удаления с серединой списка, LinkedList должен быть быстрее, чем ArrayList.

## Теоретически

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<Integer> list = **new** LinkedList<>();

**for** (**int** i = 0; i < 5\_000\_000; i++) {

list.add(**new** Integer(i));

}

**long** start=System.currentTimeMillis();

**for**(**int** i=0;i<100;i++){

list.add(2\_000\_000, **new** Integer(Integer.MAX\_VALUE));

}

System.out.println("Время работы для LinkedList (в милисекундах) = " + (System.currentTimeMillis()-start));

}

}

**Вывод:**

**Время работы для LinkedList (в милисекундах) = 1873**

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<Integer> list = **new** ArrayList<>();

**for** (**int** i = 0; i < 5\_000\_000; i++) {

list.add(**new** Integer(i));

}

**long** start=System.currentTimeMillis();

**for** (**int** i=0;i<100;i++){

list.add(2\_000\_000, **new** Integer(Integer.MAX\_VALUE));

}

System.out.println("Время работы для ArrayList (в миллисекундах) = " + (System.currentTimeMillis()-start));

}

}

**Вывод:**

**Время работы для ArrayList (в миллисекундах) = 181**

Неожиданно! Казалось бы, мы проводили операцию, где LinkedList должен быть намного эффективнее — вставку 100 элементов в середину списка. Да и список у нас огромный — 5000000 элементов: ArrayList’у приходилось сдвигать по паре миллионов элементов каждый раз при вставке! В чем же причина его победы?

**Во-первых, доступ к элементу осуществляется в** ArrayList **за фиксированное время.** Когда ты указываешь:

list.add(2\_000\_000, **new** Integer(Integer.MAX\_VALUE));

то в случае с ArrayList [2\_000\_000] это конкретный адрес в памяти, ведь у него внутри массив. В то время как у LinkedList массива нет. Он будет искать элемент номер 2\_000\_000 по цепочке ссылок. Для него это не адрес в памяти, а ссылка, до которой еще надо дойти:

***fistElement.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next.next………***

В итоге при каждой вставке (удалении) в середине списка ArrayList уже знает точный адрес в памяти, к которому он должен обратиться, а вот LinkedList’у еще надо до нужного места “дотопать”.

**Во-вторых,** дело в структуре самого ArrayList’a. Расширение внутреннего массива, копирование всех элементов и сдвиг элементов осуществляет специальная внутренняя функция — System.arrayCopy(). Она работает очень быстро, потому что специально оптимизирована для этой работы. А вот в ситуациях, когда “топать” до нужного индекса не нужно, LinkedList действительно показывает себя лучше.

Например, если вставка происходит в начало списка. Попробуем вставить туда миллион элементов:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

getTimeMsOfInsert(**new** ArrayList());

getTimeMsOfInsert(**new** LinkedList());

}

**public** **static** **long** getTimeMsOfInsert(List list) {

//напишите тут ваш код

Date currentTime = **new** Date();

insert1000000(list);

Date newTime = **new** Date();

**long** msDelay = newTime.getTime() - currentTime.getTime(); //вычисляем разницу

System.out.println("Результат в миллисекундах: " + msDelay);

**return** msDelay;

}

**public** **static** **void** insert1000000(List list) {

**for** (**int** i = 0; i < 1000000; i++) {

list.add(0, **new** Object());

}

}

}

**Вывод: *Результат в миллисекундах: 43448 Результат в миллисекундах: 107*** Совсем другой результат! На вставку миллиона элементов в начало списка ArrayList затратил больше 43 секунд, в то время как LinkedList справился на 0,1 секунды! Сказался именно тот факт, что в этой ситуации LinkedList’у не пришлось “пробегать” каждый раз по цепочке ссылок до середины списка. Он сразу находил нужный индекс в начале списка, а там уже разница в принципах работы была на его стороне:) На самом деле, дискуссия “ArrayList против LinkedList” имеет очень широкое распространение, и сильно в нее углубляться на текущем уровне мы не будем.

**Главное, что тебе нужно запомнить:**

* Не все преимущества той или иной коллекции “на бумаге” будут действовать в реальности (мы разобрали в это на примере с серединой списка)
* Не стоит бросаться в крайности при выборе коллекции (“ArrayList всегда быстрее, используй его и не ошибешься. LinkedList давно никто не пользуется”).

Хотя даже создатель LinkedList Джошуа Блох так говорит:) Тем не менее, эта точка зрения верна далеко не на 100%, и мы в этом убедились. В нашем предыдущем примере LinkedList отработал в 400 (!) раз быстрее. Другое дело, что ситуаций, когда LinkedList будет лучшим выбором, действительно немного. Но они есть, и в нужный момент LinkedList может тебя серьезно выручить. Не забывай про то, о чем мы говорили в начале лекции: для разных задач наиболее эффективными являются разные структуры данных. Нельзя на 100% уверенно говорить, какая структура данных будет лучше, пока неизвестны точно все условия задачи. Позднее ты будешь больше знать об этих коллекциях, и сделать выбор будет проще. Но самый простой и действенный вариант всегда один: испытай и то, и другое на реальных данные своей программы. Тогда ты сможешь своими глазами увидеть результаты работы обоих списков и точно не ошибешься :)