

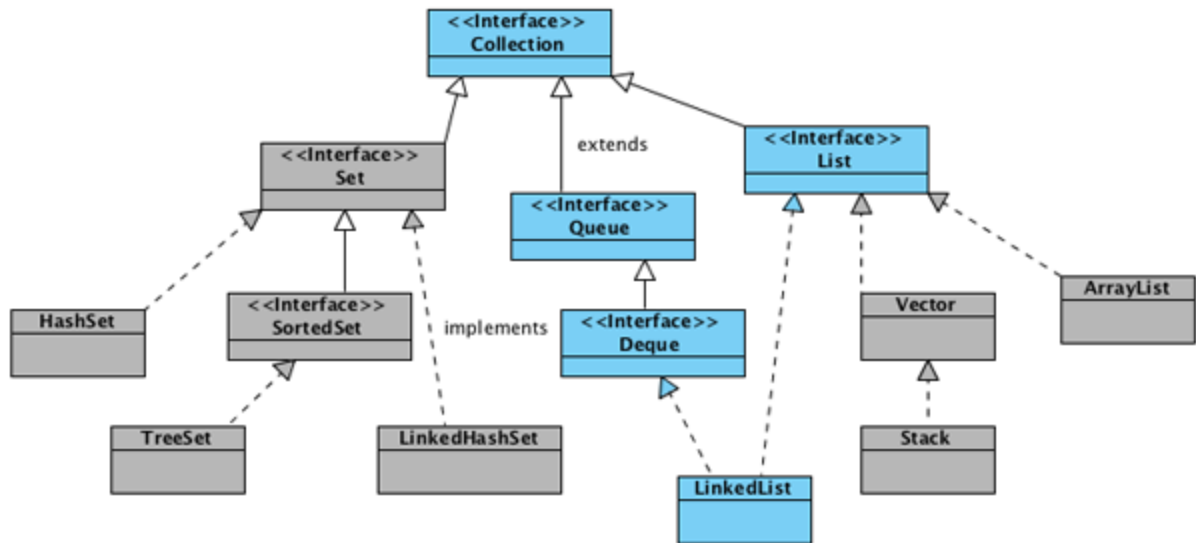
 tarzan82 22 сентября 2011 в 10:32

Структуры данных в картинках. LinkedList

Java*

Приветствую вас, хабражители!

Продолжаю начатое, а именно, пытаюсь рассказать (с применением визуальных образов) о том как реализованы некоторые структуры данных в Java.



В прошлый раз мы говорили об [ArrayList](#), сегодня присматриваемся к LinkedList.

LinkedList — реализует интерфейс List. Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Итератор поддерживает обход в обе стороны. Реализует методы получения, удаления и вставки в начало, середину и конец списка. Позволяет добавлять любые элементы в том числе и null.

Создание объекта

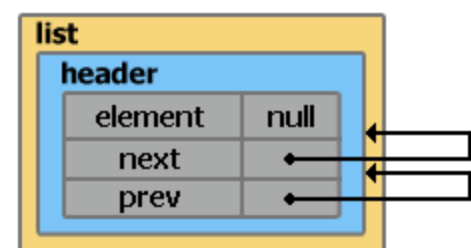
```
List<String> list = new LinkedList<String>();
```

Footprint{Objects=2, References=4, Primitives=[int x 2]}
Object size: 48 bytes

Только что созданный объект list, содержит свойства **header** и **size**.

header — псевдо-элемент списка. Его значение всегда равно **null**, а свойства **next** и **prev** всегда указывают на первый и последний элемент списка соответственно. Так как на данный момент список еще пуст, свойства **next** и **prev** указывают сами на себя (т.е. на элемент **header**). Размер списка **size** равен 0.

```
header.next = header.prev = header;
```



Добавление элементов

```
list.add("0");
```

Footprint{Objects=5, References=8, Primitives=[int x 5, char]}

Object size: 112 bytes

Добавление элемента в конец списка с помощью методом **add(value)**, **addLast(value)** и добавление в начало списка с помощью **addFirst(value)** выполняется за время O(1).

Внутри класса **LinkedList** существует static inner класс **Entry**, с помощью которого создаются новые элементы.

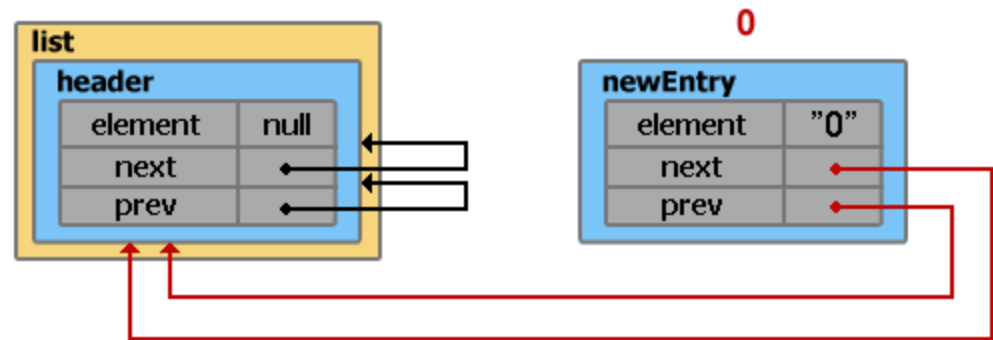
```
private static class Entry<E>
{
    E element;
    Entry<E> next;
    Entry<E> prev;

    Entry(E element, Entry<E> next, Entry<E> prev)
    {
        this.element = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Каждый раз при добавлении нового элемента, по сути выполняется два шага:

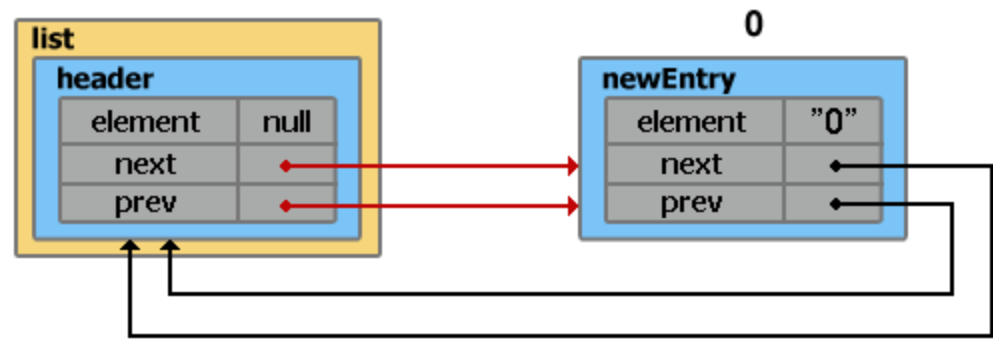
1) создается новый новый экземпляр класса **Entry**

```
Entry newEntry = new Entry("0", header, header.prev);
```



2) переопределяются указатели на предыдущий и следующий элемент

```
newEntry.prev.next = newEntry;
newEntry.next.prev = newEntry;
size++;
```



Добавим еще один элемент

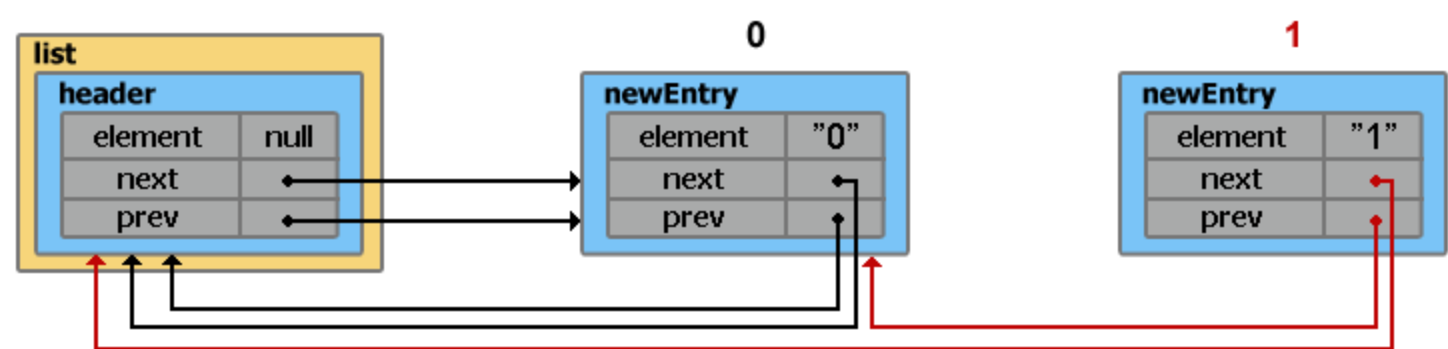
```
list.add("1");
```

Footprint{Objects=8, References=12, Primitives=[int x 8, char x 2]}

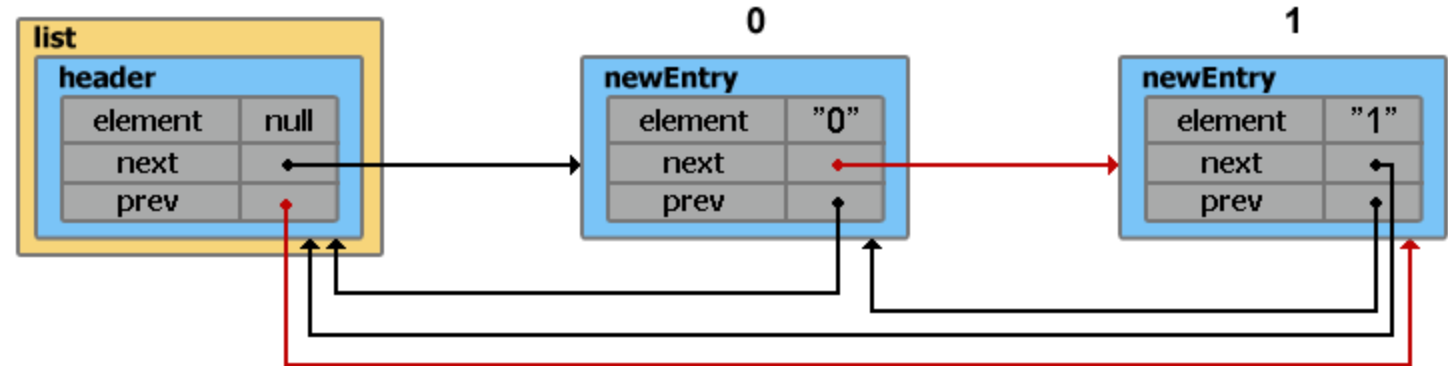
Object size: 176 bytes

1)

```
// header.prev указывает на элемент с индексом 0
Entry newEntry = new Entry("1", header, header.prev);
```



2)



Добавление элементов в «середину» списка

Для того чтобы добавить элемент на определенную позицию в списке, необходимо вызвать метод **add(index, value)**. Отличие от **add(value)** состоит в определении элемента перед которым будет производиться вставка

```
(index == size ? header : entry(index))
```

Метод **entry(index)** пробегает по всему списку в поисках элемента с указанным индексом. Направление обхода определяется условием **(index < (size >> 1))**. По факту получается что для нахождения нужного элемента перебирается не больше половины списка, но с точки зрения асимптотического анализа время на поиск растет линейно — $O(n)$.

```
private Entry<E> entry(int index)
{
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);

    Entry<E> e = header;

    if (index < (size >> 1))
    {
        for (int i = 0; i <= index; i++)
            e = e.next;
    }
    else
    {
        for (int i = size; i > index; i--)
            e = e.prev;
    }

    return e;
}
```

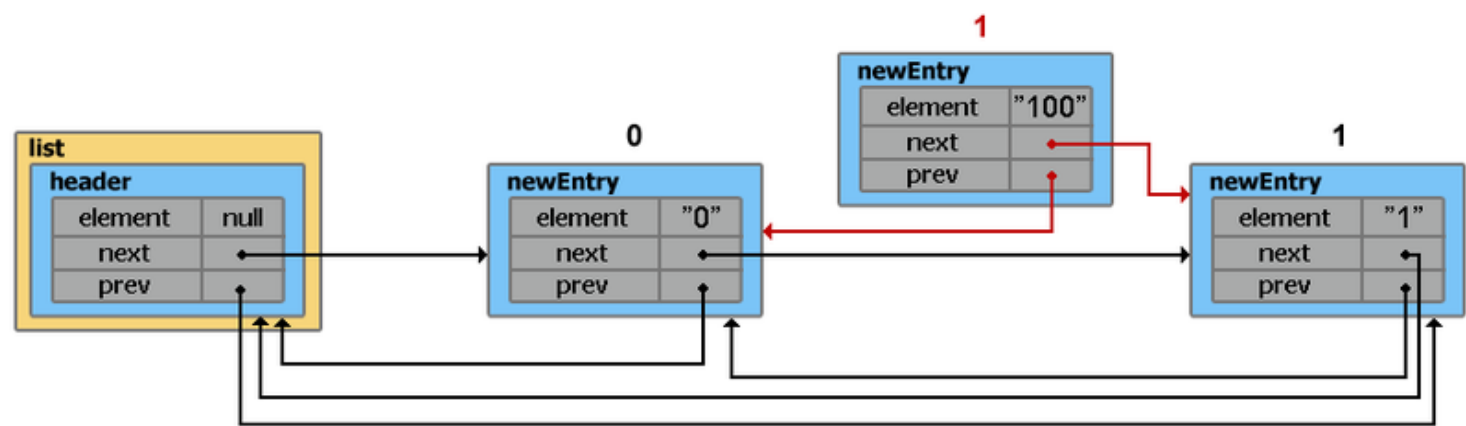
Как видно, разработчик может словить **IndexOutOfBoundsException**, если указанный индекс окажется отрицательным или большим текущего значения **size**. Это справедливо для всех методов где в параметрах фигурирует индекс.

```
list.add(1, "100");
```

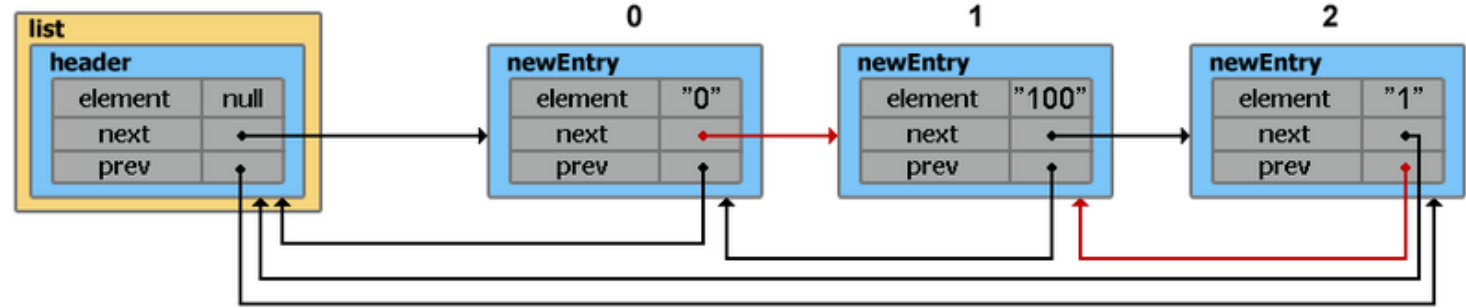
Footprint{Objects=11, References=16, Primitives=[int x 11, char x 5]}
Object size: 248 bytes

1)

```
// entry указывает на элемент с индексом 1, entry.prev на элемент с индексом 0
Entry newEntry = new Entry("100", entry, entry.prev);
```



2)



Удаление элементов

Удалять элементы из списка можно несколькими способами:

- из начала или конца списка с помощью **removeFirst()**, **removeLast()** за время $O(1)$;
- по индексу **remove(index)** и по значению **remove(value)** за время $O(n)$.

Рассмотрим удаление по значению

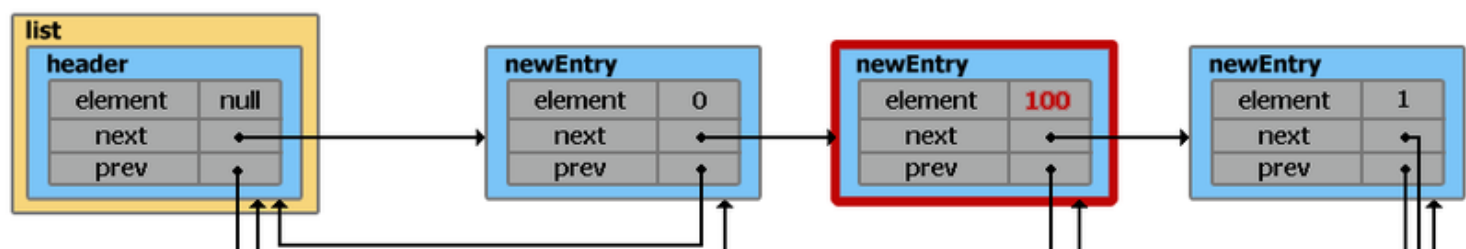
```
list.remove("100");
```

Footprint{Objects=8, References=12, Primitives=[int x 8, char x 2]}
Object size: 176 bytes

Внутри метода **remove(value)** просматриваются все элементы списка в поисках нужного. Удален будет лишь первый найденный элемент.

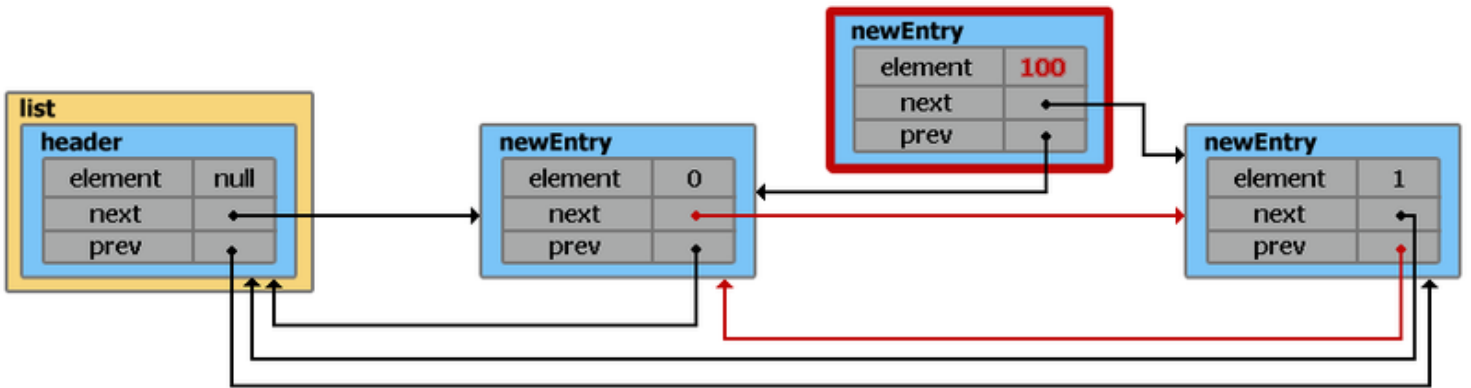
В общем, удаление из списка можно условно разбить на 3 шага:

1) поиск первого элемента с соответствующим значением



2) переопределяются указатели на предыдущий и следующий элемент

```
// Значение удаляемого элемента сохраняется
// для того чтобы в конце метода вернуть его
E result = e.element;
e.prev.next = e.next;
e.next.prev = e.prev;
```



3) удаление указателей на другие элементы и предание забвению самого элемента

Все потоки

Разработка

Администрирование

Дизайн

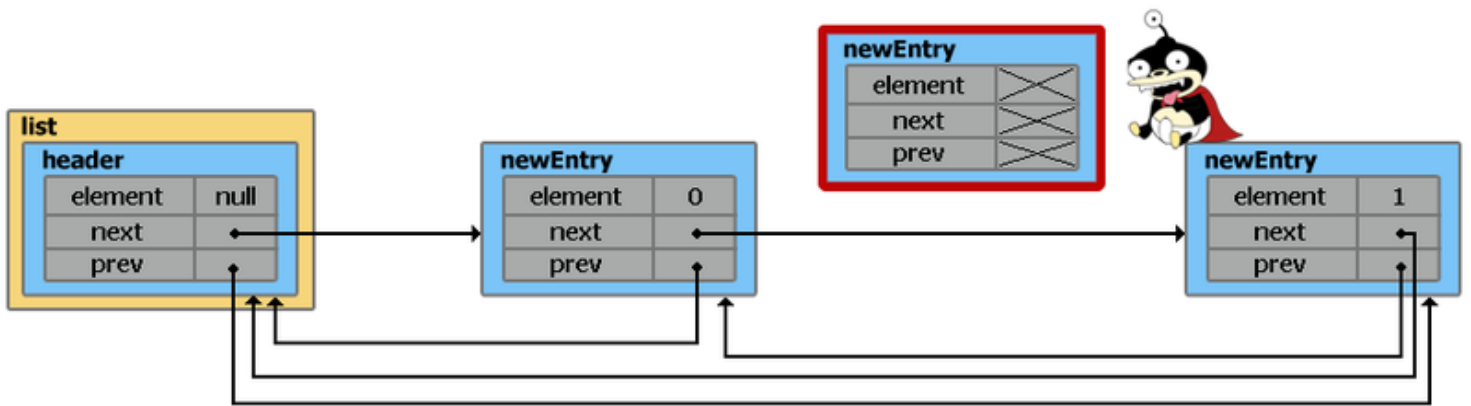
Менеджмент

Маркетинг

Научпоп

🔍

size--;



Итераторы

Для собственноручного перебора элементов можно воспользоваться «встроенным» итератором. Сильно углубляться не буду, процессы протекающие внутри, очень похожи на то что описано выше.

```
ListIterator<String> itr = list.listIterator();
```

Приведенный выше код поместит указатель в начало списка. Так же можно начать перебор элементов с определенного места, для этого нужно передать индекс в метод **listIterator(index)**. В случае, если необходимо начать обход с конца списка, можно воспользоваться методом **descendingIterator()**.

Стоит помнить, что **ListIterator** свалится с **ConcurrentModificationException**, если после создания итератора, список был изменен не через собственные методы итератора.

Ну и на всякий случай примитивный пример перебора элементов:

```
while (itr.hasNext())
    System.out.println(itr.next());
```

Итоги

- Из LinkedList можно организовать стек, очередь, или двойную очередь, со временем доступа O(1);
- На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);
- Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-обертки;
- Не синхронизирован.

Ссылки

Исходники LinkedList
Исходники LinkedList из JDK7
Исходники JDK OpenJDK & trade 6 Source Release — Build b23

Объем занимаемой памяти измерялся с помощью инструмента memory-measurer. Для его использования также понадобится Guava (Google Core Libraries).

 **+42**

 **493K**

 **1105**




Хабы: Java

Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронпочта



113
Карма


0
Рейтинг

@tarzan82

Пользователь

Комментарии 22



 **Akvel** 22.09.2011 в 11:25


Спасибо — очень просто написано и легко для понимания.

Nibbler порадовал :-)


 **+1**

Ответить



 


 **knekrasov** 22.09.2011 в 11:59

Хороший цикл статей. Спасибо!

 **0**

Ответить

 **alist** 22.09.2011 в 12:07

В связи со статьей возник вопрос: кто-нибудь в реальном проекте когда-либо использовал LinkedList?

Из своего опыта скажу, что даже в тех случаях, когда он по идее должен был дать прирост в производительности — т.е. большой список с частым удалением и добавлением элементов в середину и начало списка — ArrayList все равно оказывался быстрее.

Объяснение простое: ArrayList внутри себя использует System.arrayCopy — нативную функцию, которая

отрабатывает на удивление шустро. В то же время для LinkedList'a приходится переопределять несколько указателей, но это не самое главное. Все эти указатели на предыдущего и следующего обходятся сборщиком мусора. Из-за этого растет время, необходимое для стадии маркинга, а с учетом того, что в одном списке оказываются элементы из разных поколений, вся процедура затягивается еще больше.

В единственный раз, когда LinkedList почти догнал ArrayList по производительности, я решил попробовать PersistentVector из Clojure, и тот оказался в полтора раза быстрее обоих. PersistentVector внутри представляет из себя дерево массивов, поэтому операции вставки приводят к изменению только одного или нескольких сегментов дерева, т.е. одной или несколькими операциям arrayCopy, которая для небольших массивов остается непревзойденной по скорости.

Так что остается ощущение, что LinkedList остается в JDK только для того, чтобы у кандидатов на интервью про его эффективность вопросы задавать.

 **+5**

Ответить

 **Damaskus** 22.09.2011 в 12:29 

В LinkedList добавление в конец или начало списка будет производиться всегда за одинаковое время независимо от текущего размера списка. Если вы не знаете какого размера будет ваш список заранее, то (ИМХО) для операции составления больших списков, лучше использовать LinkedList.

 **+5**

Ответить

 **alist** 22.09.2011 в 13:19 

Да, это ясно, но ведь список будет какое-то время жить в рантайме, по нему нужно будет итерироваться, и с учетом того, что мы добавляем туда элементы, итерироваться придется неоднократно. А это довольно дорогая операция по сравнению с ArrayList'ом.

Если нам так уж нужно, чтобы элементы находились в определенном порядке, не лучше ли формализовать этот порядок в виде компаратора и использовать SortedSet?

А если порядок не нужен, почему бы не добавлять элементы в конец списка?

А если ожидаемое количество элементов известно, не проще ли создать ArrayList с заданным первоначальным размером?

А если все-таки, несмотря на все вышеперечисленные предложения, больше подходит LinkedList, не стоит ли посмотреть в сторону [B-деревьев](#)?

Есть только один сценарий, при котором LinkedList уместен — когда он стоит в основе реализации очереди, используемой несколькими потоками.

 **0**

Ответить

 **Damaskus** 22.09.2011 в 15:01 

В бинарных деревьях удобно искать, а не добавлять. Процесс добавления может иногда может вызывать перестроение дерева, что «дорого». В LinkedList же все наоборот, по крайней мере при добавлении в конец(начало).

 **0**

Ответить

 **Colwin** 23.09.2011 в 05:30 

Думаю, что от связанных списков вообще важна идея) а в качестве отдельных элементов подобного списка уже давно используют массивы.

 **+2**

Ответить

 **Siper** 22.09.2011 в 14:08 

Я использовал. LinkedList реализует Queue. Там где нет потенциальных состязаний LinkedList меня устраивает.

 **+1**

Ответить

 **Colwin** 23.09.2011 в 05:30 

ArrayList тоже устроит, не так ли? :-)

 **+1**

Ответить

o  **Throwable** 22.09.2011 в 14:42 ^

Точно такой же вопрос возник с самого начала статьи. Помню времена, когда на еще на Паскале я делал разные динамические структуры в качестве задания... Кольцевой LinkedList я использовал для хранения виджетов в контейнере, чтобы удобней было Tab-ом фокус переключать.

Вообще LinkedList не позволяет random-access и может обрабатываться только последовательно. Поэтому область его применения сильно ограничена. ArrayList реально работает и быстрее и эффективнее.

 0 Ответить  ...

o  **edhell** 22.09.2011 в 16:58 ^

Согласен, что очень специфичный и редко используемый контейнер. Для интереса запустил поиск по исходникам, которые когда-то писал... Примеры нашёл, хотя везде бы и ArrayList нормально сработал.

Вот пример: история введённых URL в программе с GUI. При введении нового адреса надо добавить его в начало списка, а если он уже был где-то в списке, то старый элемент удалить (чтобы не было дубликатов).

В этом случае у ArrayList по логике вещей будет больше операций (т. к. удаление будет всегда долгим, а изредка и добавление), хотя на практике еще непонятно кто будет работать быстрее.

Еще нашёлся пример с задачей с diofant.ru: в алгоритме надо было в списке городов часто удалять из середины города по известному названию.

 0 Ответить  ...

o  **Colwin** 23.09.2011 в 05:33 ^

В LinkedList итерирование займет больше, чем копирование части массива в ArrayList. Это утверждение верно как минимум для ≤ 1024 элементов (проверено)

 +1 Ответить  ...

o  **erl** 27.04.2012 в 14:09 ^

В обоих случаях (ArrayList и LinkedList) удаление дубликата потребует линейного времени. Здесь подойдет LinkedHashSet: как и в списке, порядок элементов сохраняется, а удаление и добавление занимают постоянное время.

 0 Ответить  ...

o  **Siper** 22.09.2011 в 14:05

В итогах опечаточка. $O(1)$ — из **концов** списка.

 0 Ответить  ...


o  **tarzan82** 22.09.2011 в 14:23 ^

Если вы про

на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется $O(1)$

то здесь всё верно написано. Загляните в исходник, в этих методах нет перебора элементов.

 0 Ответить  ...

o  **aiwan9** 09.08.2012 в 17:16 ^

$O(1)$, когда итератор уже «дошел» до нужного элемента. Если посмотреть в доку, то там сказано, что будет удален последний элемент, возвращенный методом *next()* (или *previous()*). А вот чтобы дойти до нужного элемента потребуется $O(n/2) = O(n)$.

 0 Ответить  ...

o  22.09.2011 в 15:26

НЛО прилетело и опубликовало эту надпись здесь

 0 Ответить  ...

ice9

22.09.2011 в 18:21

Знания всё-таки освежают, освежевывать их — это как-то уж слишком сурово :)

+3

Ответить

...

agarus

22.04.2017 в 16:12

«header — псевдо-элемент списка. Его значение всегда равно null...»
Что-то я не вижу у себя никакого header`а. Или его и не должно быть? Тогда как он может быть null?

Да я понимаю, что статье уже 6 лет. Но все же джава привержена обратной совместимости. Если все же с тех пор все поменялось, то буду благодарен если кто-то поделится в какой версии это header был.

0

Ответить

...

Maccimo

23.04.2017 в 09:46

Java 6:
<http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/5f4bfda58ef8/src/share/classes/java/util/LinkedList.java#l95>

Если наблюдаемое поведение не меняется, то и проблем с совместимостью нет.
Сейчас и пустой ArrayList , созданный без явного указания capacity , массива из 10 элементов не содержит и создаёт его только после начала заполнения.

0

Ответить

...

agarus

24.04.2017 в 03:04

Спасибо. Чуть промахнулся веткой: https://habrahabr.ru/post/127864/#comment_10187886

0

Ответить

...

agarus

24.04.2017 в 03:01

О, спасибо. Теперь вижу — в Java 7 header тоже еще был, а в восьмой исчез.
То есть, это private поле, только в пустом списке оно не равно null, как указано в статье, а имеет null поля: header.next = header.prev = header.element = null.
А я себе думаю, что за магия такая — откуда у него могут быть поля (пусть и null`евые) если он сам null.

0

Ответить

...

Только полноправные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

ПОХОЖИЕ ПУБЛИКАЦИИ

2 января в 12:00

LJV: Чему нас может научить визуализация структур данных в Java

+85

23K

236

11

+11

4 сентября 2021 в 16:46

Как снизить зависимость кода от структуры данных?

+2

8.9K

59

41

+41

18 июля 2021 в 16:55

Две открытые библиотеки для обучения байесовских сетей и идентификации структуры данных

+6

2.3K

48

1

+1

МИНУТОЧКУ ВНИМАНИЯ

[Разместить](#)

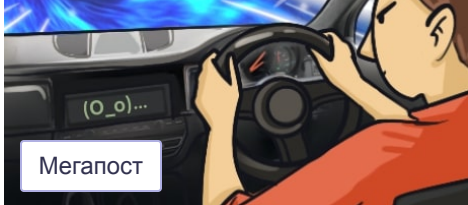




Промокод — твой билет в общество потребления



Тетрис на стероидах: тестируем War Robots на Steam Deck

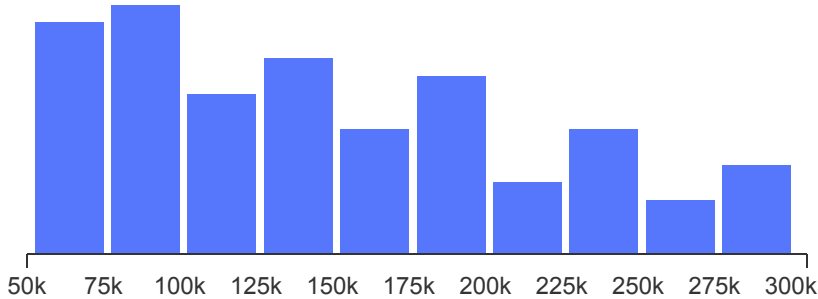


Назад в сезон Java: читаем лучшие статьи

СРЕДНЯЯ ЗАРПЛАТА В IT

157 820 /мес.

— средняя зарплата во всех IT-специализациях по данным из 5 374 анкет, за 2-ое пол. 2022 года. Проверьте «в рынке» ли ваша зарплата или нет!



Проверить свою зарплату

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 20:49

Что не так с ДЭГ Москвы на этот раз?

+261 26K 33 98 +98

вчера в 23:33

Смерть Mozilla — это смерть открытого Интернета

+88 23K 36 213 +213

вчера в 16:03

Бифуркация (фантастический рассказ)

+22 3K 12 15 +15

сегодня в 07:06

ЦБ хочет компенсировать страдания российских инвесторов за счет «недружественных» нерезидентов

+21 3.9K 5 3 +3

сегодня в 00:19

Гайд по межсетевому экранированию (nftables)

+17 4.5K 100 2 +2

Финтех и геймдев: где зеленой трава, а нагрузка больше

Подкаст

ЧИТАЮТ СЕЙЧАС

Смерть Mozilla — это смерть открытого Интернета

23K 214 +214

Что не так с ДЭГ Москвы на этот раз?

26K 98 +98


Активность найма на IT-рынке в августе 2022

19K 15 +15

Крякнул софт? Суши сухари

 27K  132 +132

Американские компании начали убирать кнопки Facebook** для авторизации со своих сайтов

 5.7K  7 +7

Хотим узнать, что вы думаете о своём месте работы (анонимно)

Опрос

РАБОТА

Java разработчик
425 вакансий

Все вакансии

Ваш аккаунт

Войти
Регистрация

Разделы

Публикации
Новости
Хабы
Компании
Авторы
Песочница

Информация

Устройство сайта
Для авторов
Для компаний
Документы
Соглашение
Конфиденциальность

Услуги

Корпоративный блог
Медийная реклама
Нативные проекты
Образовательные
программы
Стартапам
Мегапроекты



Настройка языка

Техническая поддержка

Вернуться на старую версию