Задание 1 – реализация собственного связного списка и сравнение с библиотечным LinkedList’ом. (графики в Приложении снизу)

В таблице ниже представлена зависимости разности времени исполнения простейших операций над таблицами. Время в таблице представлено в **наносекундах**. Измерялась разность (**MyLinkedList- LinkedList**) в эквиваленте времени на выполнение разных операций.

Данные в таблице – это средние значения после 100 итераций (для лучшей точности эксперимента). При добавлении 100000000 элементов дефолтовая память заканчивалась, но для чистоты эксперимента будет достаточно данных до 10000000 объектов.

Данные:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Кол-во объектов** | **ADD** | **GET** | **SET** | **REMOVE** |
| 100 | 190 | 165 | -138 | -168 |
| 1000 | 171 | 126 | 4569 | -566 |
| 10000 | 40492 | 31547 | 35642 | 36145 |
| 100000 | 121514 | 132356 | 109895 | 108896 |
| 1000000 | 222265 | 569883 | 208344 | 296230 |
| 10000000 | 5695750 | 7571000 | 4011050 | 6451149 |
| 100000000 | java.lang.OutOfMemoryError: Java heap space | | | |

Интерпретация результатов:

-LinkedList и MyLinkedList работают с одинаковой скоростью до 10000 объектов, хотя это не объективная оценка с точки зрения объектов, заполняющих листы. Я использовал класс MyComplex из самой первой домашки.

-При большом количестве объектов LinkedList выигрывает в скорости.

Почему? – не могу сказать Реализации простейших операций практически не отличаются, кроме того, что они не ссылаются на отдельные методы, а имеют реализацию внутри.

Так же clear() здесь не проверен, но он просто разрывает связи первого и последнего без удаления всех элементов, так что он будет работать быстрее.

Так же я реализовал метод getSequence, который возвращает последовательность с текущего элемента (хотя с этим может справиться и итератор), зато теперь есть такой метод))

Вывод: в целом реализация моего LinkedList’а практически не отличается от оригинального (потому что она и правда неплоха), но джавовый работает немного быстрее.

Задание 2 - Сравнительный анализ производительности основных Java коллекций.  
(Графики в самом конце)

1. Сравнить производительность ArrayList и LinkedList для основных операций:  
   Ниже представлена таблица разности времени выполнения различных запросов в формате (**ArrayList-LinkedList**). Результат – среднее от 100 итераций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Кол-во объектов** | **ADD** | **GET** | **SET** | **REMOVE** |
| 100 | -20580 | -689 | 589 | 400 |
| 1000 | -21292 | -6380 | -7199 | -5869 |
| 10000 | -26399 | -33849 | -33930 | -29020 |
| 100000 | -109507 | -147446 | -97342 | -81857 |
| 1000000 | -1403310 | -1770815 | -2117490 | -1388892 |

ArrayList выигрывает по скорости, что не удивительно, ведь у него есть маркер RandomAccess и доступ к объекту осуществляется проще, чем у LinkedSilt’a.

Процесс добавления нового объекта в середину списка должен быть проще и быстрее у LinkedList’a из-за его реализации, но почему-то все тесты выдавали обратный результат.

Вывод:

-В следствии реализации LinkedList удобнее использовать, когда нам постоянно нужно обновлять коллекцию, но не слишком часто иметь к ней доступ (брать из неё объекты)

-ArrayList подойдет для постоянного взаимодействия со списком (использование объектов, хранящихся в нем). Операции вставки и удаления у него более затратные по сравнению с LinkedList’om.

1. Сравнить производительность HashSet, LinkedHashSet, TreeSet для основных операций:  
   Ниже представлена таблица времени выполнения различных запросов(в нс).

Тестировал с обычными объектами. Результат – среднее от 100 итераций.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | HashSet |  | LinkedHashSet | | TreeSet |  |
| Кол-во объектов | ADD | REMOVE | ADD | REMOVE | ADD | REMOVE |
| 100 | 524 | 1128 | 348.94 | 1608 | 894 | 1307 |
| 1000 | 365 | 1746 | 442 | 2203 | 713 | 1364 |
| 10000 | 1363 | 3228 | 987 | 4016 | 1425 | 2061 |
| 100000 | 3565 | 8808 | 5607 | 9354 | 4308 | 4731 |
| 1000000 | 4553 | 9734 | 5243 | 9757 | 5889 | 7818 |

Простейшие операции по удалению и вставке объектов выполняются во всех трех реализациях множества примерно за одинаковое время.

Преимущество использования:

**HashSet** – обычно используется эта реализация, которая оптимизирована для быстрого поиска в коллекции. Но порядок элементов определяется по сложному алгоритму.

**LinkedHashMap** – расширяет класс **HashSet**, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.

**TreeSet –** если важна сортировка и порядок объектов по заданному правилу, используем его. Полезная и удобная штука и благодаря лямбдам можно не париться насчет Compare и Comparable, а закинуть принцип сравнения при создании коллекции.

1. Сравнить производительность HashMap, LinkedHashMap, TreeMap:  
     
   Ниже представлена таблица времени выполнения различных запросов(в нс).

Тестировал с обычными объектами. Результат – среднее от 100 итераций.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | HashMap |  |  |  |
| Кол-во объектов | put | get | replace | remove |
| 100 | 227 | 574 | 635 | 719 |
| 1000 | 457 | 813 | 1420 | 1629 |
| 10000 | 1353 | 3765 | 1675 | 3080 |
| 100000 | 4216 | 4309 | 2798 | 4800 |
| 1000000 | 3609 | 4386 | 2946 | 5205 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | LinkedHashMap | |  |  |
| Кол-во объектов | put | get | replace | remove |
| 100 | 369 | 810 | 582 | 736 |
| 1000 | 478 | 981 | 827 | 1047 |
| 10000 | 1525 | 2396 | 1468 | 2957 |
| 100000 | 3100 | 4044 | 2636 | 4221 |
| 1000000 | 4032 | 5223 | 3840 | 5068 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | TreeMap |  |  |  |
| Кол-во объектов | put | get | replace | remove |
| 100 | 575 | 4377 | 940 | 597 |
| 1000 | 737 | 2469 | 1293 | 1720 |
| 10000 | 2026 | 4580 | 2898 | 1926 |
| 100000 | 3086 | 4277 | 3101 | 3495 |
| 1000000 | 4419 | 6139 | 4084 | 2499 |

Как и по множествам, простейшие операции в мапах выполняются практически за одинаковые промежутки времени. Возможно я не могу заметить четкой разницы из-за типов, использованных объектов (Ключи Integer, Объекты String – не литералами, а через конструктор). По полученным данным можно сделать вывод, что простейшие операции в этих 3 реализациях мапы построены одинаково или похожим образом. Во всяком случае у **HashMap и LinkedHashMap** метод get реализован одинаково.

Особенности использования:

**HashMap** – обычная мапа реализующая хэш таблицу с доступом по ключу к объекту, но она не отсортированная. Используется для хранения в отображении, когда порядок не важен.

**LinkedHashMap –** наследует HashMap но так же хранит информацию о первом и последнем объектах и хранит их связным списком. Используется, когда важен порядок хранения элементов.

**TreeMap –** отсортированное отображение, используют когда важно в каком порядке будут располагаться объекты в мапе.

Приложение:

Прим. Результат отрицательный, т.к. ArrayList отработал быстрее.

Вывод – простейшие операции выполняются примерно с одинаковой скоростью.

Далее я разделил графики по разным действиям для Map

Общий вывод по графикам:

По всем простейшим действиям получается такая градация по скорости  
TreeMap – HashMap – LinkedHashMap  
Но это нельзя трактовать как неоспоримую истину по причине не объективности эксперимента, а именно выбор сортировки в TreeMap и виде объектов в мапах.

Хотя в общем и целом можно сказать, что их скорость примерно одинакова, ведь измерялось все в ns и диапазон измерений не превышал один порядок.