Выводы на тему: Параметризация, Java – коллекции

Для оценки примерного времени выполнения основных операций – добавления, поиска, удаления элемента в начало, середину и конец воспользовался методом nanoTime(). Взяв 10000 итераций для каждого метода и проведя 100 таких замеров для усреднения результата, я пришел к выводам:

1. Поиск и добавление элементов в списках LinkedList<E> и MyLinkedList<E> выполняется пропорционально количеству элементов в обоих случаях и, при этом, результаты в целом схожи. При удалении элемента в середине списка всегда проигрывает MyLinkedList<E> из-за того, что в нем предоставлена анонимная реализация итератора.

Сравнение производительности LinkedList<E> с MyLinkedList<E>

RemMid	19349423	22995566

Выводы: реализация MyLinkedList<E> аналогична по производительности выполнения основных операций с LinkedList<E>, за исключением удаления элемента из середины списка.

2. LinkedList<E> и ArrayListt<E>.

   a. Вставка в начало выполняется в ArrayListt<E> за линейное время, а за o(1) в LinkedListt<E>. Добавление в конец за константное время у обеих коллекций. Чтобы добавить новый элемент, мы должны сначала инициализировать новый массив с большей емкостью и скопировать все существующие элементы в новый массив. Только после копирования текущих элементов мы можем добавить новый элемент. Теоретически добавление нового элемента происходит за амортизированное константное время, вследствие предоставления native arraycopy() метода.

Сравнение производительности LinkedList<E> с ArrayList<E>

AddMid	95034386	19552394

   b. Получение по индексу: чтобы получить элемент с индексом просто возвращаем элемент, находящийся в i-ом индексе для ArrayListt<E>. Следовательно, временная сложность равна константе. LinkedListt<E>, в отличие от ArrayListt<E>, не поддерживает быстрый произвольный доступ. Чтобы найти элемент по индексу, мы проходим часть списка вручную, в худшем случае весь список, то есть получаем линейное время доступа.

c. При удалении первого элемента в ArrayList нам пришлось перемещать все оставшиеся элементы обратно на один индекс, из-за чего и проигрываем по времени LinkedList, который просто находит первый элемент и переопределяет ссылки элементов.

Сравнение производительности LinkedList<E> с ArrayList<E>-1

RemFirst	156402	4511762

LinkedList пришлось пройти половину элементов списка чтобы удалить ссылки в среднем элементе. В данном случае время доступа определяет временную сложность.

Сравнение производительности LinkedList<E> с ArrayList<E>-2

RemMid	350702368	10811714

LinkedList реализует интерфейс Deque<E>, поэтому время нахождения последнего элемента в списке одного порядка, что и нахождение первого элемента; в то время как ArrayList просто нужно удалять последний элемент, не сдвигая никаких элементов.

Сравнение производительности LinkedList<E> с ArrayList<E>-3

RemLast	137210	73495

Выводы: в большинстве случаев ArrayList<E> оказывается эффективнее LinkedList<E>, исключением является лишь вставка в начало и удаление первого элемента. Соответственно, если эти операции не используются лучше использовать ArrayList<E>.

3. HashSet<E>, LinkedHashSet<E>, TreeSet<E>.

Сравнение производительности HashSet<E> с LinkedHashSet<E> и TreeSet<E>

Operation	**HashSet**	**LinkedHashSet**	**TreeSet**
Add	787580	885323	2488437
Contains	244005	167061	1027566
Rem	447455	877068	1646327

В данном случае вместо метода get(), проверялся метод contains() - содержится ли элемент во множестве. У этих коллекций нет метода get(), так как нет смысла

пытаться получить тот же самый объект, который уже есть. В случае первых двух коллекций добавление (add() у HashSet вызывает метод put() у внутреннего объекта HashMap, у LinkedHashSet также вызывает метод put() у внутреннего объекта HashMap, у TreeSet вызывает метод put() у внутреннего объекта Map) и проверка на содержание этого объекта во множестве происходят за константное время, а у TreeSet за log(n) в силу того, что HashSet хранит объекты в случайном порядке, тогда как TreeSet применяет естественный порядок элементов (через интерфейс Comparable).

Выводы: HashSet и LinkedHashSet имеют практически одинаковую производительность, в то время как TreeSet работает медленнее из-за необходимости выполнять сортировку при каждой вставке. Если требуется, чтобы элементы множества были отсортированы, то лучше выбрать TreeSet. В том случае, если порядок элементов в множестве не важен — лучше использовать HashSet. Если мы хотим сохранить порядок вставки и извлечь выгоду из постоянного доступа во времени, мы можем использовать LinkedHashSet.

4. HashMap<K,V>, LinkedHashMap<K,V>, TreeMap<K,V>.

Сравнение производительности HashMap<K,V> с LinkedHashMap<K,V> и TreeMap<K,V>

Operation	**HashMap<K,V>**	**LinkedHashMap<K,V>**	**TreeMap<K,V>**
Put	1112592	780208	2330507
Contains	371117	311726	1425367
Rem	423365	576312	1643626

HashMap, будучи реализацией на основе хэш-таблицы, внутренне использует структуру данных на основе массива для организации своих элементов в соответствии с хэш-функцией. TreeMap хранит свои данные в иерархическом дереве с возможностью сортировки элементов с помощью пользовательского Comparator.
Выводы: Мы должны использовать TreeMap, если хотим, чтобы наши записи были отсортированы. Мы можем использовать LinkedHashMap, если хотим сохранить порядок вставки, используя при этом постоянный доступ во времени и если порядок ключей в карте отображения не важен — использовать HashMap.