**Содержание:**

1. Коллекции
2. Методы возвращающие представления или immutable коллекции
3. Устаревшие коллекции
4. Concurrent коллекции

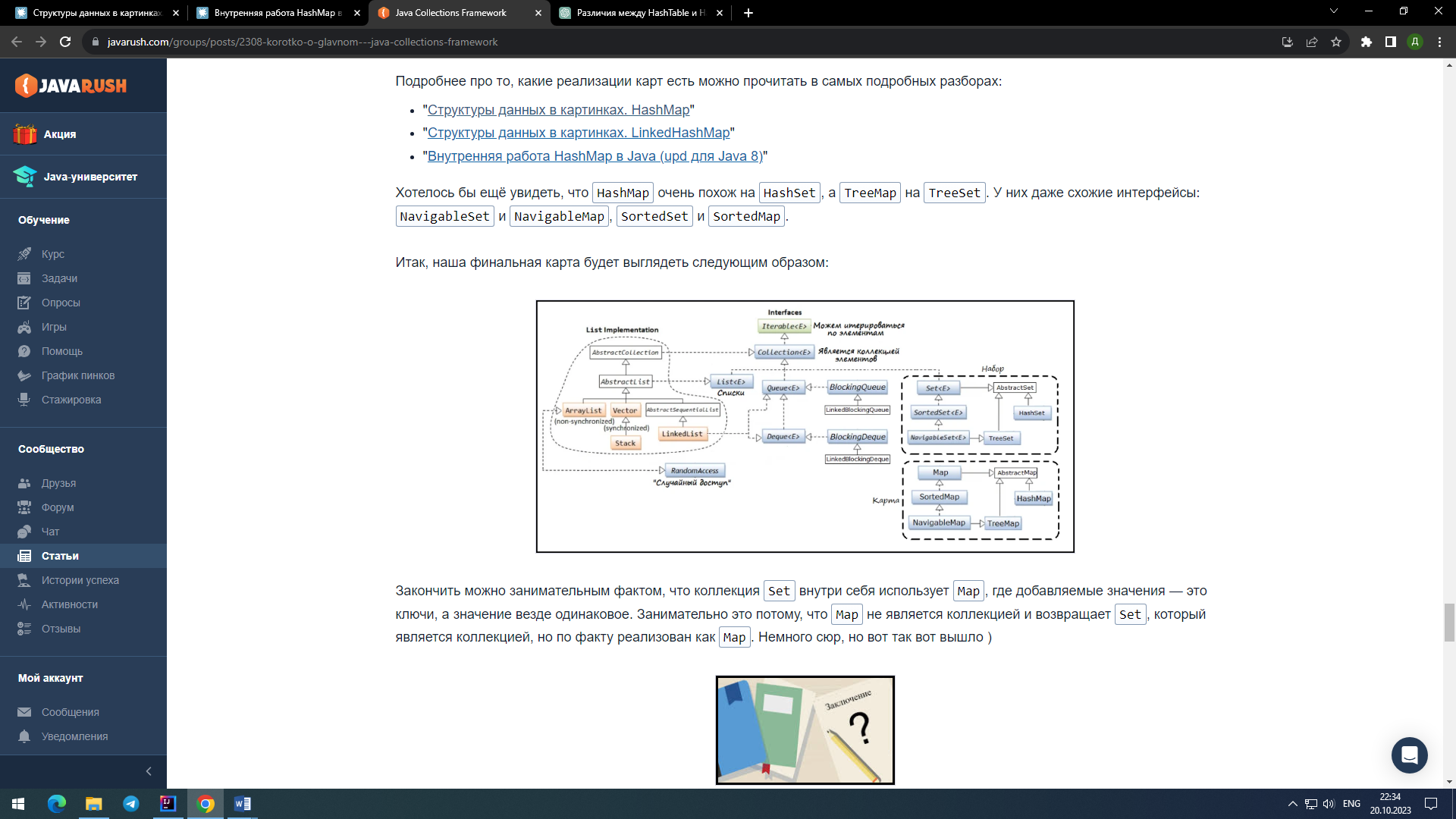
**Структуры данных**

**Массив** – самая низкоуровневая структура данных, является базовой структурой для некоторых видов коллекций.

Структурно является непрерывной областью в памяти, имеет фиксированный размер после создания и может содержать примитивы одинакового типа или ссылки на объекты, отношения которых ковариантны по отношению к переменной. Создаются динамически – размер массива не обязан быть известен на этапе компиляции, а может быть результатом вычислений в рантайме.

Непрерывность и фиксированный размер наполнения дают возможность быстрого доступа к элементам массива по индексу, т.к. можно рассчитать нахождение любого элемента умножив индекс элемента на его размер.

Такая структура массива дает более быстрый доступ к объектам по отношению к спискам, т.к. в списке нет возможности рассчитать нахождение элемента в памяти.



**Java Collection Framework** — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет пользоваться большим количеством структур данных из «коробки».

Включает в себя интерфейс **Collection**, который определяет контракт, который описывает основное поведение (абстрактные методы) и базовые концепции этих классов (инструкция по эксплуатации), например, наличие у любой реализации 2х конструктором, соблюдение контрактов equals и hashCode, выбрасывание определенных типов исключений в специфических ситуациях и др. Наследник Iterable.

Интерфейсы наследники Collection**: List, Queue, Set. Map** считается коллекцией, но к иерархии не относится.

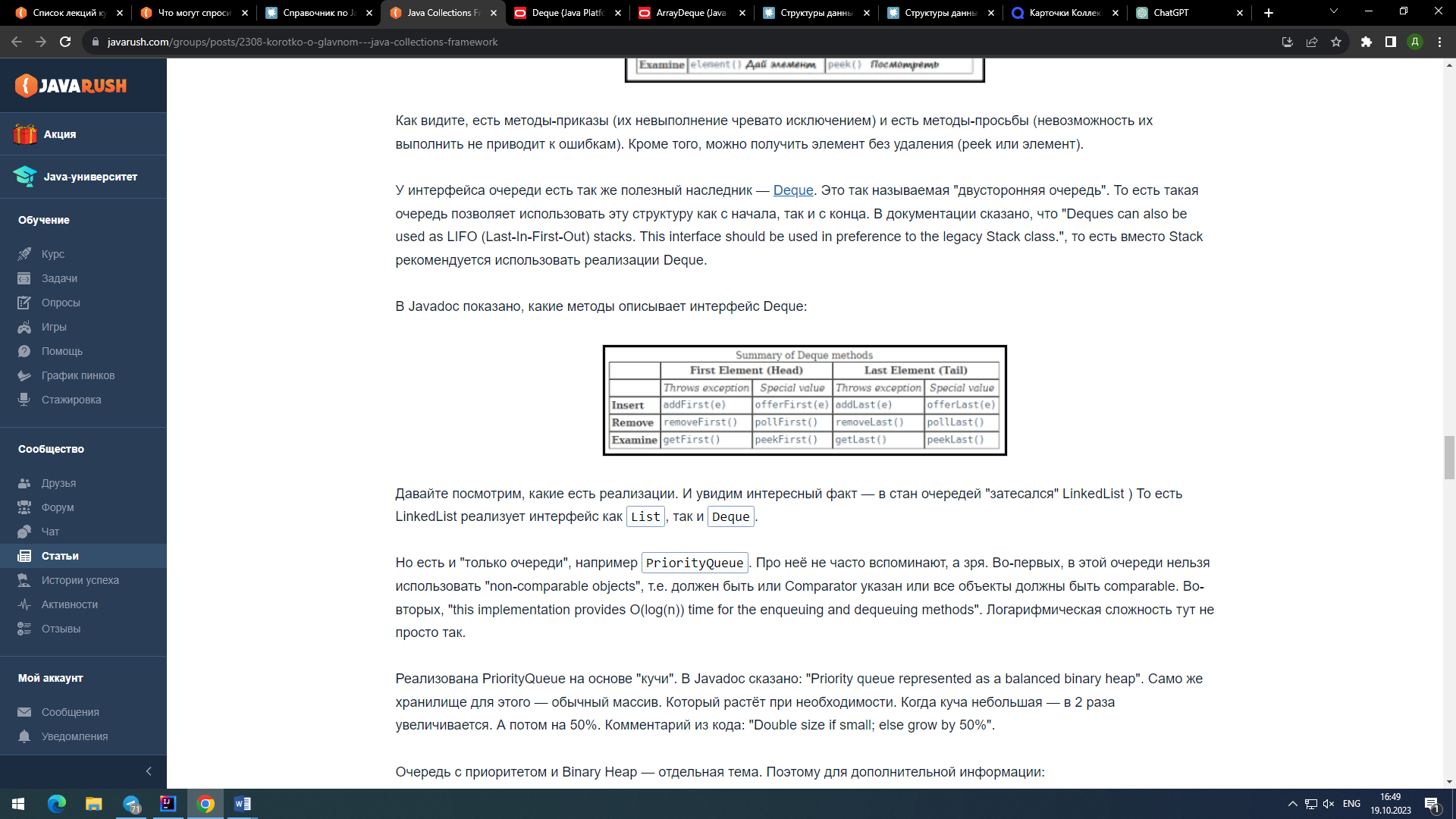
**Интерфейс List** – интерфейс списков, добавляет в контракт новые методы, которые определяют доступ по индексу, возможность иметь повторяющиеся элементы и работа с null.

**Интерфейс Queue** – интерфейс однонаправленных очередей, добавляет в контракт новые методы, которые определяют доступ FIFO (т.к. добавление в конец очереди, удаление из начала), не допускают null. Разделяет методы на fail-fast методы (кидают исключения) и fail-save (возвращают специальное значение).

Методы add (IllegalStateException), remove (NoSuchElementException), element (NoSuchElementException) – бросают RuntimeException.

offer, poll, peek – возвращают null или false.

**Интерфейс Deque** – интерфейс двунаправленных очередей, расширяет Queue, добавляет в контракт методы для работы с двумя концами очереди, которые определяют доступ FIFO и LIFO. Также разделяет все свои метода на fail-fast и fail-safe.



**Интерфейс Set** – определяет интерфейс множества, не определяет собственных методов, но содержит спецификацию для реализаций унаследованных методов и спецификацию контракта (уникальные элементы – полностью опирается на equals и hashcode, неупорядоченность элементов). \*Каждая реализация Set делегирует к соответствующим классам в Map.

**Интерфейс SortedSet –** определяет интерфейс сортированного множества, элементы упорядочены, все элементы должны быть взаимосравнимы (Comparator comparable)

**Интерфейс NavigableSet –** расширяетконтрактSortedSet, определяя методы для навигации по SortedSet.

**Интерфейс Map –** определяет интерфейс словаря (пара ключ-значение), где каждый ключ представлен множеством (неупорядоченные неповторяющиеся значения ключей, но не их значений).

Не является наследником Collection из-за различий в семантике и функциональности: Collection – интерфейс для коллекций с набором объектов, Map – набор объектов и ассоциированных с ними значений, также не стыкуются некоторые методы, например, add – принимает 1 параметр, а map требует добавление ключ-значение.

Так же Map не наследуется от Itarable напрямую, итерироваться можно с помощью entrySet/keyset/values, которые возвращают Set.

**Основные реализации:**

**ArrayList** – динамически расширяющийся массив в основе которого лежит массив, реализует List и RandomAccess – интерфейс маркер, обозначает, что можно достучаться до элементов по индексу.

Поля:

size – кол-во заполненных элементов массива   
default\_Capacity – размер по умолчанию = 10   
elemData – массив Object

**Динамическое расширение/сужение:**

Расширение – заполнение внутреннего массива инициирует его расширение, что подразумевает создание нового массива с помощью метода grow, создает новый массив (size + 1) \* 1,5, затем переносит значения массива в этот массив с помощью System.arrayCopy.

Сужение – trimToSize, необходим, если elemData содержит много незаполненных значений. Уменьшает elemData до size через System.arrayCopy(). Инициируется пользователем.

**Добавление и удаление элементов:**

*Добавление в начало/середину*: требует сдвига части массива от индекса куда вставляется новый элемент до конца массива.

*Добавление в конец*: не требует дополнительных манипуляций с элементами массива.

Любая операция добавления может потребовать инициализации нового массива.

*Удаление из начала или середины*: Также требует сдвига части массива.

*Удаление с конца*: не требует дополнительных манипуляций с элементами массива.

**Сложность операций:**

Доступ к элементам по индексу О(1)  
Поиск элемента по значению O(n)  
Добавление в конец О(1)  
Добавление в начало/середину O(n) + возможно нужно создать новый массив  
Добавление в конец О(1) + возможно нужно создать новый массив  
Удаление из начала/середины – O(n)  
Удаление с конца – О(1)

\*Из-за того, что массив имеет непрерывную область памяти он может кэшироваться процессором и метод System.arrayCopy является нативным и работает с машинными инструкциями, что позволяет выполнять это копирование крайне быстро, поэтому, хоть LinkedList в теории быстрее вставляет и удаляет элементы, чем ArrayList, на практике это не всегда так. Преимущества ArrayList перед LinkedList так же в том, что он занимает в 4 раза меньше памяти, а также не нагружает GC, так как по сути это 1 объект, а LinkedList это связанные вместе объекты.

**LinkedList** – двусвязный список, в основе лежит двунаправленная связь каждого элемента списка посредством хранения ссылок каждым элементом на предыдущий и последующий элементы, реализует интерфейсы List (имеет ложный доступ по индексу) и Deque.

**Структура списка:**

Для каждого хранимого элемента списка создается его представление в виде объекта Node, которые связываются друг с другом посредством сохранения ссылок на соседние элементы в полях next и previous.

public class List<E> {  
  
 Node<E> first;  
 Node<E> last;  
 int size = 0;  
  
 private static class Node<E> {  
 E element;  
 Node<E> next;  
 Node<E> previous;  
  
 public Node(E element, Node<E> next, Node<E> previous) {  
 this.element = element;  
 this.next = next;  
 this.previous = previous;  
 }  
 }  
}

**Добавление и удаление элементов:**

*Добавление начало/конец*: Список хранит в себе ссылки на первый и последний элементы, что сводит это добавление к переустановке ссылок на один из этих указателей и соответствующее присвоение новых ссылок в поля next и previous.

*Добавление в середину*: Помимо манипуляций со ссылками необходим доступ к нужному месту в списке, это вызывает сложности, т.к. структура списка требует пройти по каждому элементу, начиная с начала или конца до нужного места, а затем переприсвоить ссылки у соседних элементов, т.к. список ложно поддерживает доступ по индексу, но по сути это просто счетчик пройденных элементов.

*Удаление элемента с начала/конца*: ситуация как с добавлением

*Удаление из середины:* ситуация как с добавлением в середину.

**Сложность операций:**

Доступ по индексу О(n/2) – О(n/2) потому что исходя из индекса и длины списка можно определить откуда лучше начать, с начала или конца.  
 Доступ по значению О(n)  
 Добавление в начало/конец О(1)  
 Добавление в середину(n/2) – т.к. нужно дойти до места вставки  
Удаление начало/конец О(1)  
Удаление середина(О/2) – т.к. нужно дойти до места удаления.  
Последовательное удаление/добавление нескольких элементов с помощью ListIterator – первое добавление будет иметь О(n/2), т.к. нужно дойти до нужного индекса, все последующие операции будут О(1), т.к. итератор сохранит расположение индекса, что снизит время на поиск нужного места.

\*linkedList не имеет реального применения в коде, см выше, с его помощью можно реализовать различные LIFO (Stack) или FIFO (Queue) очереди.

**HashMap/HashSet –** структуры данных, в основе которых лежат свойства множеств, структурно организованная в виде хэш-таблицы (HashSet полностью полагается на реализацию HashMap, а вместо значений для ключей использует PRESENT – объект заглужку).

**Хэш-таблица:**

Представлена динамическим массивом Node[ ] (стандартный размер 16), каждая ячейка массива формирует односвязный список в случае, если ячейка не пуста, после достижения длины этого списка значения 8, он реорганизуется в красно-черное дерево (с Java 8).

В собственных реализациях возможно использование любой структуры данных внутри массива бакетов, включая открытую адресацию – способ хранения в хэш-таблице, который требует хранение всех элементов непосредственно в бакете – в случае коллизии приводит к поиску свободного бакета (не организацию структуры данных в нем) с помощью алгоритма пробирования.

Динамичность массива бакетов выражается в его увеличении по мере его наполнения для уменьшения вероятности коллизии. Конфигурируется параметром loadFactor (по умолчанию 0,75). Пример: 16\*0,75 = 12. Превышение этого порогового значения инициирует выделение нового массива бакетов увеличенного в 2 раза. Этот процесс инициирует процесс реиндексирования всей структуры – для каждого элемента вычисляется новый индекс свежевыделенного массива бакетов.

**Определение позиции (бакета) для добавления/поиска:**

Включает определение индекса для помещения элемента в Node[ ] (в бакет) – определяется на основании вычисления HashCode значения ключа и вычисление индекса массива, в который элемент будет помещен с помощью index = hash&(array.length - 1), что соответствует hash% (array.length – 1).

**Поддержание уникальности ключей и замена их значений:**

Уникальность ключей определяется на основании их сравнения с помощью equals, ситуация, когда добавляется существующий ключ с новым значением приводит к замене прошлого значения новым.

**Пример алгоритма поиска:**

1. Вычисляется Hash ключа
2. Вычисляется индекс бакета
3. Каждый элемент в бакете проверяется на equals

\*Добавление значения также требует выполнения этого алгоритма, т.к. необходимо знать существует такой ключ или нет.

**Итерация:**

Map не имплементит Iterable, итерация осуществляется за счет внутреннего класса KeySet, который является представлением множества ключей и поддерживает итерацию.

**Особенности ключей:**

Использование null –допускается использование null в качестве ключа или значения, повторное добавление null в качестве ключа будет приводить к перетиранию значения.

Immutable ключи – необходимо использовать неизменяемые ключи, т.к. изменение любого поля, которое участвует в вычислении hashCode приведет к его изменению, что повлечет за собой невозможность получения его из hash-структуры, т.к. она будет утверждать, что такого ключа нет.

\*Нельзя использовать массивы в качестве ключей, т.к. HashCode и equals для массивов невозможно переопределить, и реализация hashCode по умолчанию использует identityHash (генерится не на основании содержимого – для этого Arrays.hahsCode), а equals сравнивает ссылки.

**Сложность операций:**

Поиск: Если нет коллизий О(1), если список меньше 8 элементов, то О(n), если больше 8 О(LogN).  
Добавление/удаление: все тоже самое, т.к. операции завязаны на поиск места для добавления.

**TreeMap/TreeSet –** структуры данных, в основе которых лежат свойства деревьев, а именно красно-черного дерева (бинарное, ацикличное, сбалансированное дерево поиска). Как и во всех реализациях TreeSet полагается на TreeMap. Наследник SortedMap, NavigableMap. Упорядоченное, поэтому требуется, чтобы содержимое было Comparable или передан Comparator для таких элементов (null в качестве ключа только если передан Comparator.nullsFirst или nullsLast).

**Деревья –** общеепонятие для структур, обладающих свойством связанности и ацикличности. Связанность – наличие маршрута между любой парой вершин (узлов), ацикличность – отсутствие связей между вершинами, как следствие - между каждой парой вершин (узлов) существует только один путь (частный случай графа).

**Структура:**

Узлы – корень, промежуточные узлы, листья:

Корень – узел, не имеющий родительского узла.  
Промежуточные узлы – узел, имеющий родителя и одного или двух потомков.  
Лист – узел, не имеющий ни одного потомка.

Ветви – представления связи между узлами.

Свойства:

Кол-во ветвей – кол-во связей, которые способен хранить узел (Бинарное и др.).  
Порядок – наличие правил размещения узлов в структуре(Упорядоченное, неупорядоченное)  
Сбалансированность – организация узлов дерева таким образом, чтобы расстояние от каждого листа до корня при их сравнении давали «небольшую разницу».

**Бинарное дерево** – неупорядоченное, несбалансированное, бинарное дерево.

**Бинарное дерево поиска** – упорядоченное, несбалансированное, бинарное дерево.

Свойство упорядоченности обеспечивает эффективный поиск log(n). Не гарантирует указанную сложность, если вставляются постепенно увеличивающиеся данные - 1-2-3-4-5, такое добавление преобразует дерево в связанный список и сложность O(n).

**Авл-дерево** – упорядоченное, сбалансированное, бинарное дерево.

Правило упорядоченности:

Значения левого ребенка узла должно быть меньше значения узла-родителя, значение правого ребенка узла должно быть больше узла-родителя.

Добавлена новая характеристика узла - высота – кол-во узлов от указанного узла до самого нижнего листа.

Добавляет новое состояние в узлы – разница высот – отражает разность высот дочерних узлов.

Свойство сбалансированности достигается соблюдением условия: Разница между высотами дочерних узлов должна быть <= |1|. Которое обеспечивается корректировкой высот узлов во время вставки и удаления и проверки на сбалансированность дерева, если дерево разбалансировано, то инициируется процесс балансировки.

Балансировка – реорганизация дерева таким образом, чтобы соблюдалось условие высот, при этом дерево не изменяло свою структуру. Достигается с помощью поворотов (левый поворот, правый поворот, правый левый поворот, левый правый поворот).

Свойство сбалансированности позволяет дереву сохранить Log(n) вне зависимости от поступающих данных.

\*Постоянная необходимость вычисления и изменения высот узлов по пути рекурсии, вычисление всех баланс факторов на пути рекурсии и выполнение множественных поворотов по пути рекурсии замедляет весь алгоритм работы, особенно при больших объемах данных.

Все операции с таким деревом будут иметь сложность log(n).

**Красно-черное дерево** – упорядоченное, сбалансированное, бинарное дерево с более эффективной балансировкой.

Добавлена новая характеристика узла:

Цвет (черный/красный).  
Черная высота - количество черных узлов на пути от исходного узла до null (исходный узел не считается).  
Ссылка на родительский узел – дает возможность уйти от рекурсии (минусы см. ниже).

Правила балансировки:

1) Каждый узел такого дерева окрашен либо в красный, либо в черный.  
 2) Листом является не узел без детей, а несуществующий узел – null (в роли него выступает внутренний класс – заглушка, все листья ссылаются на него).  
3) Корень и листья всегда черные.   
4) У каждого красного узла оба ребенка черные.   
5) Все черные высоты дерева должны быть равны.  
6) Добавляемый узел – красный (нарушает меньшее кол-во правил балансировки)

*АВЛ – поиск быстрее:* Красно-черное дерево не гарантирует идеальной сбалансированности, в отличие от АВЛ дерева, т.е. может быть выше него примерно в 1.4 раза, что дает меньшую производительность при поиске элемента по сравнению с АВЛ.  
*Красно-черное – быстрее вставка/удаление:* Та же причина – меньшая строгость балансировки, что уменьшает ее частоту и уменьшает накладные расходы.

Проблема рекурсии – код проще, но дороже из-за того, что стэк имеет меньший размер, можно улететь по StackOverFlow, если дерево достаточно глубокое. Также вызов функции достаточно дорогая операция (для каждой функции нужно создать свой фрейм с адресами возвратов и локальными переменными). Можно отметить это как плюс для КЧ дерева, т.к. в узлы добавлена ссылка на родителя.

**Оставшиеся реализации:**

**PriorityQueue** – единственная прямая реализация Queue. Имеет возможность управлять порядком извлечения элементов основываясь на приоритете (только Comparable объекты или передать Comparator). Имеет структуру двоичной кучи, в качестве хранителя объектов выступает динамический массив Object[ ] (не список Node, а массив).

**Двоичная куча** – полное, бинарное, упорядоченное дерево. Полное означает, что дерево заполняется слева направо на каждом уровне (не требуются операции балансировки структуры, она сбалансирована сама по себе).

Правило упорядоченности:

Родитель должен быть (больше/меньше своих предков) т.е. в корне должен находится максимальный/минимальный узел.

Сложность:

Поиск О(1) – т.к. получается всегда корневой элемент.

Вставка/удаление log(n) – т.к. после таких операций кучу нужно просеивать (организовывать узлы по правилу упорядоченности).

**ArrayDeque** – двусторонняя очередь на основе массива. Подходит для реализации очередей LIFO и FIFO.

*Stack* – ArrayDeque лучше, так как Stack использует синхронизацию (LIFO).  
 *LinkedList* – требует больше памяти для хранения каждого элемента, при этом ArrayDeque проигрывает в производительности, т.к. это динамический массив (создание массива и перемещение элементов) (LIFO, FIFO). Но возможно, что оптимизированный System.arrayCopy делает выигрышным именно ArrayDeque.

**LinkedHashMap/LinkedhashSet –** наследник HashMap и SequenceMap. Реализует функциональность Хэш-таблицы с возможностью сохранения порядка элементов.

Реализация:

Стандартная реализация HashMap, где каждый Node имеет доп. поля next, prev и структура имеет ссылки на head и tail.

Порядок итерации:

Порядок вставки: По умолчанию, LinkedHashMap сохраняет порядок добавления элементов. Это значит, что при итерации по карте элементы будут выводиться в порядке их добавления.

Порядок доступа: Если конструктор LinkedHashMap создан с параметром accessOrder равным true, то порядок элементов будет изменяться в зависимости от их последнего доступа. Последний использованный элемент будет перемещен в конец списка, что делает LinkedHashMap подходящим для реализации LRU (Least Recently Used) кэшей.

**WeakHashMap** – реализация хэш таблицы с использованием weak-references. Ключи, на которые нет сильных ссылок могут быть удалены GC. Имеет оверхэд на обработку слабых ссылок.

**Методы представлений и неизменяемых коллекций:**

**Представления коллекций:**

1. List.subList()
2. NavigableSet.subSet()
3. Map.keySet()
4. Map.values()
5. Map.entrySet()

Эти методы возвращают представления исходной коллекции, что означает связь с исходной коллекцией и изменение в представлении отразятся на исходной коллекции.

**Неизменяемые коллекции:**

**Неизменяемые представления:**

1. Collections.unmodifiableList()
2. Collections.unmodifiableSe()
3. Collections.unmodifiableMap()

**Неизменяемые коллекции:**

1. List.of() / List.copyOf()
2. Set.of() / Set.copyOf()
3. Map.of() / Map.copyOf()

Также есть метод Arrays.asList(), который возвращает частично неизменяемое представление массива – не позволяет изменять размер (add, remove), но разрешает set.

Под неизменяемостью понимается невозможность модифицировать коллекцию или представление, включая редактирование значений.

**Устаревшие классы**

Не рекомендуются к использованию, т.к. имеют устаревший API, используют синхронизацию и синхронизованы на уровне метода, что блокирует всю коллекцию как на чтение, так и на запись. Это ставит их проигрышное положение в ситуации, когда не нужна синхронизация, а когда синхронизация все-таки необходима, то предпочтительнее использовать concurrent коллекции, т.к. они имеют более производительные блокировки.

**Vector –** Появился до Collection Framework в Java 1. Имеет схожую реализацию с ArrayList, но в отличие от ArrayList является потокобезопасным, что накладывает на него издержки в производительности.

**Stack** – Обертка над классом Vector (может использоваться как список), реализующая принцип LIFO (Last in first out), частично синхронизированная коллекция (кроме push). Так же основан на динамическом массиве. Push – добавить; pop – удалить; peek – получить элемент.

**HashTable** – Появилась в Java 1.Реализует структуру данных – hash-таблица. Синхронизована.

**Enumeration** – устаревший интерфейс с функционалом типа Iterator, имеет методы hasMoreElements() и nextElement(), используется в классах Vector и Hashtable, не поддерживает добавление и удаление элементов.

**Потокобезопасные коллекции:**

*Использование устаревших коллекций* – полностью блокируют коллекцию на чтение и на запись при любой операции (используют synchronized на методе), что дает низкую производительность (замедляет доступ к коллекции при чтении), плюс имеют устаревший API и не рекомендуются к использованию.

*Использование синхронизированных оберток для коллекций (Collections.synchronized\*\*\*\*)* – также блокируют коллекцию на чтение и запись (используют synchronized на методе), но представляют из себя потокобезопасную обертку над «современными» коллекциями.

**Использование коллекций из пакета concurrent:**

**CopyOnWriteArrayList (CopyOnWriteArraySet) –** покобезопасный аналог ArrayList, оптимизированный для сценариев частых операций чтения и нечастых операций записи из-за особенностей реализации.

Запись – использует стратегию полного копирования массива при записи/удалении элементов, то есть массив создается ровно под кол-во элементов +1. Операция записи блокирующая.

Чтение – содержит ссылку на «старый» массив, все операции чтения происходят по этой ссылке, как только добавление элемента завершается, ссылка подменяется новой и дальнейшие чтения используют ее. Под операциями чтения понимаются также итерация и получение size массива (итерация по снепшоту ограничивает возможности итератора – невозможна операция remove). Чтение не блокируется.

CopyOnWriteArraySet полностью делегирует к CopyOnWriteArrayList, но при этом обеспечивает уникальность элементов.

Сложность:

Добавление/удаление – всегда O(n), т.к. копируется массив  
 Получение индексу – O(1)  
 Поиск – O(n)

**ConcurrentHashMap** (**ConcurrentHashSet**) – потокобезопасный аналог HashMap, представляет сегментированную таблицу, блокировка сегментов не приводит к блокировке всей таблицы, что уменьшает конкуренцию между потоками при записи в разные сегменты. Чтение неблокирующее, запись блокирует только сегмент (первая запись неблокирующая).

**ConcurrentHashSet** не выделен в качестве класса, если нужен подобный функционал для Set, то можно воспользоваться keyset у Map коллекции.

**ConcurrentSkipListMap (ConcurrentSkipListSet)** – потокобезопасный аналог TreeMap, основан на пропускающих списках - каждый слой представлен отсортированным связанным списком, каждый верхний слой представлен таким же списком, но более разряженным (1/2 шанс, что он будет добавлен в слой выше), совокупность таких добавлений на каждом слое – башня, которая содержит одинаковые значения, но расположенные на разных слоях, такая структура данных позволяет обеспечить практически Log(n) для всех операций (есть погрешности, т.к. имеет место случайность), а потокобезопасность достигается за счет использования оптимистических блокировок (CAS) с помощью varhandle, unsafe, AtomicReference или volatile.

[Видео](https://www.youtube.com/watch?v=Q9MdwzewSZg), [серия статей про конкурентные структуры данных](https://habr.com/ru/articles/250815/)

**ConcurrentLinkedQueue (ConcurrentLinkedDeque)** – потокобезопасные очереди на основе связанного списка, используют оптимистический подход к блокировке (используют CAS с помощью классов Unsafe, Varhandle, AtomicReference или volitile).

**Блокирующие очереди (BlockingQueue и BlockingDeque) –** интерфейсы определяющие контракты для поддержания операций ожидания – ожидает, когда очередь станет не пустой при извлечении элемента или, когда появится свободное место для извлечения элемента. Их реализации являются потокобезопасными.

Представители (есть аналоги для Deque):

SynchronousQueue (при попытке добавить поток заблокируется, пока кто-то не попросит элемент и наоборот   
ArrayBlockingQueue (массив фиксированного размера, добавление сверх capacity блокирует поток, пока место не освободится и наоборот)   
LinkedBlockingQueue (блокирующая очередь на основе списка)  
PriorityBlockingQueue (блокирующая очередь с приоритетом)