**Алгоритм** – это конечная последовательность действий для решения конкретной задачи. **Асимптотическая сложность** показывает, как количество операций, выполняемых алгоритмом, будет меняться в зависимости от объема входных данных. Всего существует около 20ти видов сложности. Сложность обозначается через O-нотацию. Виды сложности алгоритмов:

1. Линейная O(n). Например, найти единственный белый шар в мешке с черными шарами. Количество операций растет прямо пропорционально общему количеству шаров.
2. Логарифмическая O(log(n)). Например, поиск слова в словаре. Сложность тут растет медленнее.

**Бинарный (двоичный) поиск** основан на идее дихотомии – разделении набора объектов на два непересекающихся множества. Отсортированный массив разбивается на две части, и одна из них отбрасывается. Деление продолжается до тех пор, пока элемент не будет найден. Бинарный поиск есть в стандартной библиотеке *Arrays.binarySearch()*.

1. Константная (постоянная) O(1). Например, подойти к библиотекарю, спросить, где книга, и взять её.
2. Квадратичная O(n2). Например, найти дубликаты в массиве чисел.



**Рекурсия** – это функция, вызывающая сама себя. Она обязательно должна содержать **базовый случай** (условие выхода из рекурсии). Недостатки рекурсии:

* может замедлить работу программы по сравнению с циклами
* занимает много памяти
* может привести к переполнению стека и завершению программы с ошибкой

Пример рекурсии – подсчет факториала.

*int factR(int n) {*

*int result;*

*if (n == 1) return 1;*

*return factR(n-1) \* n; }*

Наиболее распространенные виды сортировок:

1. **Пузырьковая сортировка** (***buble sort***). Применяется только в учебных целях, т.к. существуют более эффективные алгоритмы. Массив обрабатывается в (n - 1) проходов. В каждой итерацию сравниваются соседние элементы и, при необходимости, меняются местами.

Сложность равна *O(n2)*. Является **стабильной сортировкой** (сохраняет относительный порядок элементов, что помогает при нескольких условиях сортировки).

*int[] array = { 10, 2, 10, 3, 1, 2, 5 };*

*boolean needIteration = true;*

*while (needIteration) {*

*needIteration = false;*

*for (int i = 1; I < array.length; i++) {*

*if (array[i] < array[i – 1]) {*

*int tmp = array[i];*

*array[i] = array[i – 1];*

*array[i – 1] = tmp;*

*needIteration = true; } } }*

1. **Сортировка слиянием** (***merge sort***). Рекурсивно разбивает задачи на меньшие, пока размер массива не достигнет единицы. Затем рекурсивно соединяет два соседних массива в упорядоченный третий массив.

Сложность равна *O(n \* log(n))*. Требуется, как минимум, *n* **дополнительной памяти**. Алгоритм **стабильной сортировки** (сохраняет относительный порядок равных элементов).

*public static void mergeSort(int[] array) {*

*if (array.length <= 1) {*

*return array; }*

*int[] result = new int[array.length]; // массив для результата сортировки*

*int[] left = mergeSortDescending(Arrays.copyOfRange(array, 0, array.length / 2));*

*int[] right = mergeSortDescending(Arrays.copyOfRange(array, array.length / 2, array.length));*

*int leftIdx = 0;*

*int rightIdx = 0;*

*int resultIdx= 0;*

*while (leftIdx < left.length && rightIdx < right.length) { // сливаем результаты*

*if (left[leftIdx] >= right[rightIdx]){*

*result[resultIdx] = left[leftIdx];*

*leftIdx++;*

*} else {*

*result[resultIdx] = right[rightIdx];*

*rightIdx++; }*

*resultIdx++; }*

*while (leftIdx < left.length) { // если один массив закончился раньше, чем второй*

*result[resultIdx] = left[leftIdx];*

*leftIdx++;*

*resultIdx++; }*

*while (rightIdx < right.length) {*

*result[resultIdx] = right[rightIdx];*

*rightIdx++;*

*resultIdx++; }*

*return result; }*

1. **Быстрая сортировка** (***quicksort***). Аналог пузырьковой сортировки. Выбирается опорный элемент (любой). Массив сортируется так, чтобы разбить его на две-три части (меньше опорного, равны опорному и больше опорного). Рекурсивно повторить к полученным частям.

Сложность равна *O(n \* log(n))*. Дополнительной памяти не требуется. Является **нестабильной сортировкой** (поддерживает только один порядок элементов).

*public static void quickSort(int[] source, int leftBoarder, int rightBoarder) {*

*int leftMarker = leftBoarder;*

*int rightMarker = rightBoarder;*

*int pivot = source[(leftMarker + rightMarker)] / 2];*

*do {*

*while (source[leftMarker] < pivot) {*

*leftMarker++; }*

*while (source[rightMarker] > pivot) {*

*rightMarker--; }*

*if (leftMarker <= rightMarker) {*

*if (leftMarker < rightMarker) {*

*int tmp = source[leftMarker];*

*source[leftMarker] = source[rightMarker];*

*source[rightMarker] = tmp; }*

*leftMarker++;*

*rightMarker--; }*

*} while (leftMarker <= rightMarker);*

*if (leftMarker <= rightBoarder) {*

*quicksort(source, leftMarker, rightBoarder); }*

*if (leftBoarder <= rightMarker) {*

*quicksort(source, leftBoarder, rightMarker); } }*

1. **Сортировка вставками** (***insertion sort***). Разделяет входной массив на две части (отсортированную и нет). Затем каждый элемент из неотсортированной части помещается на корректное место в отсортированной части со сдвигом массива.

Сложность равна *O(n2)*. Дополнительной памяти не требуется. Медленно работает на больших объемах.

*int[] array = { 10, 2, 10, 3, 1, 2, 5 };*

*for (int left = 0; left < array.length; left++) {*

*int value = array[left];*

*int i = left – 1;*

*for (; i >=0; i--) {*

*if (value < array[i]) {*

*array[i + 1] = array[i];*

*} else { break; } }*

*array[i + 1] = value; }*

1. **Поразрядная сортировка** (***radix sort***). Легкий вариант - считает общее количество вхождений каждого числового элемента и сохраняет его в массив-счетчик, а затем выводит его необходимое количество раз.

Тяжелый вариант - сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения. Затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Сложность равна *O(n)*. Требуется дополнительная память, объёмом в размере количества уникальных значений.

Подробнее о сортировках [тут](https://javarush.com/groups/posts/1997-algoritmih-sortirovki-v-teorii-i-na-praktike).

**Структура данных** – это совокупность набора данных, отношений между ними и всех методов по управлению этими данными.



Интерфейс ***Iterable<T>*** предоставляет инструменты для того, чтобы перебирать (итерировать) коллекцию:

* Цикл *forEach*

*for (String element : list) {*

*System.out.println(element); }*

* Метод *iterator()*

*final Iterator<String> iterator; // объявили объект iterator*

*iterator = list.iterator(); // проинициализировали iterator через метод iterator()*

*while (iterator.hasNext()) { // проверяем, есть ли следующий элемент в коллекции*

*final String element = iterator.next(); // возвращаем элемент*

*System.out.print(element);}*

Интерфейс ***Collection<T>*** предоставляет инструменты для того, чтобы добавлять/удалять элементы, вычислять размер коллекции и осуществлять поиск внутри нее:

* Методы *add(E e)* и *addAll(Collection c)* для одного и нескольких элементов
* Методы *contains(Object o)* и *containsAll(Collection c)*
* Методы *remove(Object o)* и *removeAll(Collection c)*
* Метод *size()*
* Метод *isEmpty()*
* Другие полезные методы типа *clear()*, *toArray()* и т.д.

Интерфейс ***Set<T>*** позволяет реализовать структуры данных, где каждый элемент уникален (может встречаться только один раз).



**Множество** (***HashSet***) – это коллекция, в которой содержатся неупорядоченные уникальные элементы без индекса. Оно создает хеш-таблицу и использует только ячейки-ключи. Множество содержит методы:

* Метод ***add(E e)*** добавляет новый элемент в множество и возвращает значение *boolean*.
* Метод ***addAll(Collection<? Extends E> c)*** добавляет все элементы любой коллекции и возвращает значение *boolean*. Если хотя бы один из элементов будет успешно добавлен – вернется *true*.
* Метод ***contains(Object o)*** позволяет выяснить наличие объекта в множестве. Возвращает *boolean*.
* Метод ***remove(Object o)*** удаляет элемент из множества.
* Метод ***isEmpty()*** проверяет что множество пустое. Возвращает *boolean*.
* Метод ***size()*** возвращает количество элементов в множестве. Возвращает *int*.

Для реализации *HashSet* можно использовать следующие конструкторы:

* *public HashSet()* – стандартная реализация.
* *public HashSet(Collection<? Extends E> c)* – создает множество на основе списка или множества.
* *public HashSet(int initialCapacity, float loadFactor)* – задает начальное количество пустых ячеек и коэффициент заполнения.

Сложность поиска элемента в множестве всегда равна *O(1)*, т.к. оно всегда знает, где находится нужный элемент.

**Связанное множество** (***LinkedHashSet***) – это множество, построенное на основе хеш-таблицы *LinkedHashMap*. Для реализации *LinkedHashSet* можно использовать те же самые конструкторы, что и для *HashSet.* Сложность поиска элемента в связанном множестве всегда равна *O(1)*.

**Упорядоченное множество** (***TreeSet***) – это множество, построенное на основе хеш-таблицы *TreeMap*. Элементы должны реализовывать интерфейс *Comparable* (единственно верная сортировка для всех объектов класса) или при создании в конструктор нужно передать тип *Comparator* (экземпляр класса с логикой сортировки).

Для реализации *TreeSet* можно использовать следующие конструкторы:

* *public TreeSet()* – стандартная реализация.
* *public TreeSet(Comparator<? super E> comparator)* – реализация с логикой упорядочивания.
* *public TreeSet(Collection<? Extends E> c)* – создание на основе любого списка или множества.
* *public TreeSet(SortedSet<E> s)* – создание на основе уже отсортированного множества.

Сложность операций поиска, добавления и удаления элемента в упорядоченном множестве равна *O(log(n))*.

Интерфейс ***Queue<T>*** (**очередь**) описывает структуры данных, где элементы располагаются один за другим согласно очередности. Работает по принципу “первым зашел – первым вышел”.

Интерфейс ***List<T>*** позволяет создавать списки с повторяющимися элементами в произвольном порядке, которые могут расширяться и поддерживают сортировку. При этом он дает возможность контролировать процесс добавления нового элемента в список, организовать поиск и вносить изменения:

* Метод *add(E e)* добавляет элемент в конец списка
* Метод *add(int index, E e)* добавляет элемент в любое место списка
* Метод *get(int index)* получает элемент по индексу
* Метод *set(int index, E e)* заменяет элемент по индексу
* Метод *remove(int index)* удаляет элемент по индексу
* Метод *remove(E e)* удаляет элемент по значению
* Другие полезные методы

Список работает только со ссылочными типами, поэтому для примитивных типов нужно использовать класс-обертку.

Методы *Arrays.asList()* и *List.of()* позволяют создать список из массива или набора значений. Списки, созданные с помощью этих методов, считаются **неизменяемыми** (***ImmutableCollections***).

Метод *toArray(T[ ])* позволяет преобразовать список обратно в массив.

*List<String> cities = Arrays.asList("Воронеж", "Южно-Сахалинск", "Иркутск"); // создание списка*

*String[] citiesArray = cities.toArray(new String[ ]{ }); // преобразование списка в массив*

**Стек** – это набор элементов с организацией доступа по принципу “первым зашел – последним вышел”.

Список ***ArrayList<T>*** – это динамический массив, который может автоматически увеличиваться в размере во время исполнения программы. При вставке/удалении элемента, происходит сдвиг элементов вправо/влево.

У динамического массива есть два основных параметра – размер и вместимость. **Размер** – текущее количество элементов в списке. **Вместимость** – потенциальное количество элементов, которое может храниться в списке. По умолчанию равна 10 и автоматически увеличивается на 50% (в меньшую сторону) при превышении лимита.

*ArrayList<String> arrayListWithDefaultCapacity = new ArrayList<>(); // вместимость 10 элементов*

*ArrayList<String> arrayListWithSetCapacity = new ArrayList<>(33); // вместимость 33 элемента*

Сложность операции добавления элемента в конец списка ***add(T)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. В зависимости от необходимости увеличивать вместимость. **Амортизированная асимптотическая сложность** (когда список уже большой и увеличение происходит редко) равна *O(1)*.

Сложность операции добавления элемента по индексу ***add(int, T)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. В зависимости от количества сдвигаемых вправо элементов. Если список уже заполнен на момент вставки, то потребуется дополнительные *O(n)* на расширение. Аналогичная сложность с операциями удаления элемента ***remove(T)*** и по индексу ***remove(int)***.

Сложность операций получения элемента по индексу ***get(int)*** и замены элемента по индексу ***set(int, T)*** всегда оценивается как *O(1)*.

Список ***LinkedList*** реализует двунаправленный связанный список. Это набор связанных узлов, которые хранят ссылки на предыдущий и следующий элементы (как вагоны в поезде). Также двусвязный список хранит указатели на хвост и голову. Этот список поддерживает последовательный обход элементов друг за другом сначала или с конца.



Сложность операций добавления элемента в начало списка ***addFirst(T)*** и в конец списка ***addLast(T)*** всегда оценивается как *O(1)*. Их результатом становится перемещение ссылок головного и конечного элементов. Аналогичная сложность с операциями удаления из начала списка ***removeFirst()*** и из конца списка ***removeLast()***.

Сложность операций получения первого элемента ***getFirst()*** и последнего элемента ***getLast()*** всегда оценивается как *O(1)*.

Сложность операций добавления элемента по индексу ***add(int, T)***, получения элемента по индексу ***get(int)*** и удаления элемента по индексу ***remove(int)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. Чем ближе элемент расположен к центру списка, тем дольше его поиск.

Сложность операций получения размера списка ***size()*** всегда оценивается как *O(1)*, т.к. она динамически изменяется вместе со списком.

**Преимущества *ArrayList<T>*:**

1. Каждый новый элемент *ArrayList<T>* занимает меньше памяти, чем *LinkedList<T>*.
2. Массив *ArrayList<T>* расположен в памяти последовательно, в отличии от хаотично распределенных узлов *LinkedList<T>*. Повышается производительность.
3. Операции копирования массивов на практике работают быстрее, т.к. они часто оптимизированы на аппаратном уровне.

**Преимущества *LinkedList<T>*:**

1. В больших списках эффективнее работает со вставками/удалениями элементов в начале списка.
2. В больших списках эффективнее работает со вставками/удалениями элементов из середины списка.
3. Гарантирует вставку в конец списка за константное время *O(1)*.

Интерфейс ***Comparator<T>*** задает правила сортировки произвольных классов. Можно наследовать много раз. Для сравнения используется метод ***int compare(T, T)***, который возвращает:

* Отрицательное число, если первый объект меньше второго
* Ноль, если объекты равны
* Положительное число, если первый объект больше второго

Для сортировки коллекции в обратном порядке используется метод ***reverse()***.

Для каждого варианта сортировки нужно писать свой класс для сравнения. Например:

*import java.util.Comparator;*

*public class ItemPriceComparator implements Comparator<Item> { // на месте T - класс Item*

*@Override*

*public int compare(Item item1, Item item2) {*

*if (item1.price > item2.price) { // более дорогой должен быть дальше в списке*

*return 1;*

*} else if (item1.price < item2.price) { // более дешёвый — ближе к началу списка*

*return -1;*

*} else { // если стоимость равна, нужно вернуть 0*

*return 0; } } } // или упростить до строки return item1.price - item2.price;*

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

*List<Item> items = new ArrayList<>(); // создаём список товаров*

*items.add(new Item("Рубашка", 4500, 37));*

*items.add(new Item("Носки", 55, 8));*

*items.add(new Item("Толстовка", 1399, 74));*

*ItemPriceComparator itemPriceComparator = new ItemPriceComparator(); // создаём объект-компаратор*

*items.sort(itemPriceComparator); // сортировка по убыванию цены*

*System.out.println(items);*

*Comparator<Item> reversedItemPriceComparator = itemPriceComparator.reversed();*

*items.sort(reversedItemPriceComparator); // сортировка по возрастанию цены*

*System.out.println(items);}*

Для стандартных типов существуют готовые решения. Например, статическая переменная ***CASE\_INSENSITIVE\_ORDER*** хранит в себе *Comparator<String>*, сортирующий строки без учета регистра:

*static class ItemStringInSensitiveNameComparator implements Comparator<Item> {*

*@Override*

*public int compare(Item item1, Item item2) {*

*return String.CASE\_INSENSITIVE\_ORDER.compare(item1.name, item2.name); } }*

Классы-обертки не реализуют компараторы, но содержат статические методы ***compare()***. Например:

*public class ItemPriceComparator implements Comparator<Item> {*

*@Override*

*public int compare(Item item1, Item item2) {*

*return Integer.compare(item1.price, item2.price); } }*

В *Java* существует утилитарный класс ***Collections*** с более чем 20 методами для работы с различными структурами данных. **Утилитарные классы** не имеют конструкторов (либо они приватные), т.к. все его методы либо приватные, либо статические (доступны без создания экземпляра класса). **Утилитарные классы** не содержат изменяемых внутренних полей.

Следующие методы быстро создают неизменяемые списки нужного размера:

* Метод ***Collections.emptyList()*** создает пустой список.
* Метод ***Collections.singletonList(T)*** создает список из единственного элемента.
* Метод ***Collections.nCopies(int, T)*** создает список, где элемент *T* повторен *int* раз.

Превратить неизменяемый список в изменяемый можно с помощью передачи его в *ArrayList*:

*final List<String> immutableList = List.of("Ёж", "Медведь", "Лиса"); // неизменяемый список*

*final ArrayList<String> mutableList = new ArrayList<>(immutableList); // уже можно изменить*

Метод ***Collections.fill(List<T>, T)*** позволяет присвоить всем элементам списка *List<T>* единое значение *T*.

Метод ***Collections.copy(List<T>, List<T>)*** позволяет скопировать все элементы из второго списка в первый. Требует гораздо меньше ресурсов, чем пересоздание списка, если списки очень большие.

Метод ***Collections.sort(List<T>, Comparator<T>)*** позволяет сортировать списки (идентичен методу из *List<T>*).

Методы ***Collections.min(Collection, Comparator<T>)*** и ***Collections.max(Collection, Comparator<T>)*** помогают искать в списке минимум и максимум соответственно.

Интерфейс ***Comparable<T>*** задает естественную (единственно верную) сортировку для класса объектов. Его можно наследовать только один раз. Он предоставляет единственный метод ***int compareTo(T)***, который ведет себя аналогично методу *compare(T, T)*. С его помощью работают метод сортировки ***Collections.sort(List<T>)***, а также методы нахождения минимума ***Collections.min(Collection<T>)*** и максимума ***Collections.max(Collection<T>)***. Этот интерфейс лучше реализовывать для простых классов.

**Отображение** (***Map***) — это такое сопоставление, при котором объекту из первого множества соответствует только один объект из второго множества. Например, сопоставление множества стран и множества континентов можно записать кодом в виде хеш-таблицы *HashMap<>*.



Интерфейс ***Map<K, V>*** задает между элементами однозначную одностороннюю связь и позволяет работать с данными по принципу ключ-значение. Он предоставляет следующие методы:

* Метод ***put(K value, V value)*** добавляет новую запись. *K* – ключ, *V* – значение. Если ключ уже есть в хеш-таблице, то значение будет переписано.
* Методы ***containsKey(Object key)*** и ***containsValue(Object value)*** позволяют понять, содержится ли в хеш-таблице конкретный ключ или значение. Возвращают булево значение.
* Метод ***get(Object key)*** возвращает значение по ключу. Если ключ не найден, то вернет *null*.
* Метод ***getOrDefault(Object key, V defaultValue)*** возвращает значение по умолчанию, если не находит искомый ключ в хеш-таблице.
* Метод ***size()*** возвращает количество записей в хеш-таблице в типе *int*. Максимальное количество записей, которое можно хранить в хеш-таблице совпадает с макс. значением *int* и равно 2 147 483 647.
* Метод ***remove(Object key)*** возвращает значение, которое хранилось по этому ключу, и удаляет из хеш-таблицы соответствующую пару ключ-значение. Если значение или ключ отсутствуют, то вернется *null*.
* Метод ***clear()*** удаляет все данные из хеш-таблицы.
* Метод ***isEmpty()*** проверяет, хранится ли что-то в хеш-таблице (f*alse*) или она пустая (*true*). Возвращают булево значение.
* Метод ***keySet()*** итерирует по всем ключам.
* Метод ***values()*** итерирует по всем значениям.
* Метод ***entrySet()*** итерирует по всем записям. Возвращаемая запись ***Entry*** содержит одновременно ключ и значение. Доступ к ним осуществляется с помощью геттеров ***getKey()*** и ***getValue()***.

*for (Entry<String, Integer> entry : actorsMap.entrySet()) {*

*String actor = entry.getKey();*

*int filmsCount = entry.getValue(); }*

Хеш-таблицы работают только со ссылочными типами. В качестве значения можно передать и список.

Если в качестве ключа используется объект собственного класса, то для правильной работы хеш-таблицы нужно переопределить методы ***hasCode()*** и ***equals()***.

*@Override*

*public boolean equals(Object o) {*

*if (this == o) return true;*

*if (o == null || getClass() != o.getClass()) return false;*

*User user = (User) o;*

*return id.equals(user.id); }*

*@Override*

*public int hashCode() {*

*return id.hashCode(); }*

У каждого объекта в java есть метод ***hashCode()*** для его представления в виде числа. **Хеш-функция** (или ***функция-свёртки***) обеспечивает преобразование значения, возвращенного методом ***hashCode()***, в уникальное целое число - **хеш**. Хеш определяет, в какой ячейке хеш-таблицы будет сохранена запись.

Алгоритм хеш-функции обладает следующими свойствами:

* Детерминизм – для одного и того же ключа возвращается одинаковый хеш.
* Эффективность – должен работать быстро.
* Ограниченность – результат должен принадлежать определенному диапазону.
* Равномерность – данные должны быть распределены по ячейкам равномерно.

**Коллизия** – ситуация, при которой хеш-функция направляет два совершенно разных ключа в одну и ту же ячейку. В результате хеш-таблица продолжит корректно работать, но время её работы увеличится, т.к. придется перебирать все значения в ячейке на предмет соответствия вызываемому ключу. При некорректно реализованном методе *hashCode()*, все ключи скопятся в одной ячейке.

При увеличении количества ячеек, все ключи заново хешируются и перераспределяются. **Коэффициент заполнения** (***load factor***) – это соотношение между занятыми и свободными ячейками. По умолчанию он равен 75%, но его можно задать в момент создания хеш-таблицы:

*public HashMap(int initialCapacity, float loadFactor) // max initialCapacity = 1 073 741 824*

**Хеш-таблица** (***HashMap***) – это список пар ключ-значение, реализующий интерфейс ***Map***. Ключи уникальны, не генерируются автоматически и хранятся не упорядочено (при итерировании перебираются хаотично). Ячейки реализованы в виде массива длинной равной степени двойки (16, 32, 64 и т.д.). По умолчанию создаются 16 пустых ячеек и коэффициентом заполнения 75%.

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(1)*. Сложность итерации всегда будет равна *O(n)*.

**Связанная хеш-таблица** (***LinkedHashMap***) – это хеш-таблица, в которой ключи хранятся упорядоченно. Данные могут быть упорядочены двумя способами:

* В порядке добавления записей – при итерации первой будет запись, которая была добавлена в хеш-таблицу самой первой.
* В порядке доступа к записям – при итерации первой будет запись, к которой дольше всего не обращались. Обращением считается вызов одного из методов выборки по ключу – *get(Object key)* и *getOrDefault(Object key, V defaultValue)*.

По умолчанию создаются 16 пустых ячеек и коэффициентом заполнения 75%. Для создания новой связанной хеш-таблицы можно использовать следующие конструкторы:

* *public LinkedHashMap()* – стандартные настройки, сортировка в порядке добавления.
* *public LinkedHashMap(Map<? Extends K, ? extends V> m)* – копирование хеш-таблицы *m*.
* *public LinkedHashMap(int initialCapacity)* – стандартные настройки с созданием *int* ячеек.
* *public LinkedHashMap(int initialCapacity, float loadFactor) –* стандартные настройки с созданием *int* ячеек и указанием коэффициента заполнения *float*.
* *public LinkedHashMap(int initialCapacity, float loadFactor, Boolean accessOrder)* – стандартные настройки с созданием *int* ячеек, указанием коэффициента заполнения *float* и типом сортировки (*true* – по порядку добавления записей, *false* – в порядке доступа к записям).

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(1)*. Из-за наличия дополнительного двусвязного списка и накладных расходов на его поддержание, связанная хеш-таблица потребляет чуть больше памяти и совершает чуть больше действий, чем простая хеш-таблица. Сложность итерации всегда будет равна *O(n)*.

**Упорядоченная хеш-таблица** (***TreeMap***) – это хеш-таблица, в которой применяются сложные правила сортировки элементов (по заданному признаку). Например, для строк это “от первой буквы алфавита к последней”, а для чисел – “от меньшего к большему”. Наличие четкого порядка упрощает итерацию и создает полезные возможности.

Для создания новой упорядоченной хеш-таблицы можно использовать следующие конструкторы:

* *public TreeMap()* – создает пустую хеш-таблицу без пустых ячеек.
* *public TreeMap(Comparator<? super K> comparator)* – создает хеш-таблицу и задает правила её сортировки.
* *public TreeMap(Comparator<? extends K, ? extends V> m)* – создает новый объект *TreeMap* со стандартными настройками и заполняет данными из переданной хеш-таблицы любого типа.
* *public TreeMap(SortedMap<K, ? extends V> m)* – создает новый объект *TreeMap* со стандартными настройками и заполняет данными из переданной хеш-таблицы типа *SortedMap* (с уже отсортированными данными).

Например, зададим порядок сортировки при передаче в конструктор объекта *Comparator*. Обычно этот метод применяется, когда нельзя отредактировать код в исходном классе, или когда в разных местах нужна разная логика сортировки.

*Comparator<User> userComparator = new Comparator<>() {*

*@Override*

*public int compare(User user1, User user2) {*

*return user1.userId - user2.userId; } };*

*Map<User, String> users = new TreeMap<>(userComparator);*

Например, зададим порядок сортировки через интерфейс *Comparable*. Этот метод удобен, когда есть только один способ сортировки данных.

*class User implements Comparable<User> {*

*public Integer userId;*

*public String username;*

*@Override*

*public int compareTo(User o) {*

*return this.userId - o.userId; } }*

Методы ***firstEntry()*** и ***lastEntry()*** возвращают первую и последнюю по порядку запись.

Методы ***firstKey()*** и ***lastKey()*** возвращают первый и последний по порядку ключ.

Методы ***pollFirstEntry()*** и ***pollLastEntry()*** возвращают первую и последнюю по порядку запись, а также удаляют их из хеш-таблицы.

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(log(n))*. Это ненамного медленнее чем *O(1)*, но выигрышнее за счет сортировки. Для поддержания сортировки упорядоченная хеш-таблица потребляет чуть больше памяти и совершает чуть больше действий, чем простая хеш-таблица. Сложность итерации всегда будет равна *O(n)*.

На примере иерархии сотрудников рассмотрим из чего состоят деревья.



* **Вершина дерева** (***vertex***), или **узел/нода** — объект, который может хранить информацию и ссылаться на другие вершины. Может иметь не более одного родителя (руководителя). У вершины могут быть потомки (подчиненные). Если потомков нет, то она называется **листом** (***leaf***). В иерархии организации вершинами являются все сотрудники.
* **Ребро дерева** (***edge***) — направленная, то есть иерархическая, связь между двумя вершинами. Направление рёбер дерева однозначно определяет какая из двух связанных вершин выше по иерархии, а какая — ниже. В иерархии организации Рита подчиняется Ёлкину.
* **Путь в дереве** (***path***), или **маршрут** — набор вершин, в котором следующие друг за другом узлы соединены рёбрами. Так как рёбра имеют строго одно направление, то и путь можно проложить только в одну сторону — сверху вниз по иерархии и никак иначе. Количество вершин в самом длинном пути от корня до листа называется **высотой**/**глубиной** древа с корнем.
* **Дерево** — совокупность вершин и рёбер без циклов. Если из вершины X1​ существует путь до вершины X2​, то гарантируется, что не существует пути от X2​ к X1​. Это значит, что сотрудник не может быть одновременно и подчинённым, и руководителем для одного и того же человека.

Если мы ограничим количество дочерних узлов в вершинах дерева числом *N*, то это число будет называться **арностью дерева**. Если арность равна двум, то дерево называют **бинарным** или **двоичным**.

Чтобы бинарное дерево превратилось в **дерево поиска**, оно должно отвечать нескольким условиям:

1. Если в узле дерева записано значение Х, то в левом поддереве располагаются только узлы со значениями, меньшими или равными Х.
2. Если в узле дерева записано значение Х, то в правом поддереве располагаются только узлы со значениями, большими или равными Х.
3. Левое и правое поддеревья отвечают тем же двум условиям. Это рекурсивное правило. Из него следует, что любое поддерево в дереве поиска будет являться деревом поиска.

Если в процессе поиска обнаруживается, что отсутствует ссылка на нужное поддерево, то искомого элемента не существует.