Задача 6. Словари (основанные на деревьях или хеш-таблицах)

Целью данной задачи является знакомство со стандартными для любого современного языка программирования структурами данных – словарями (другое название – ассоциативные массивы), которые в библиотеке языка Java называются Map (стандартные реализации – TreeMap, HashMap, LinkedHashMap и др.).

Но для того, чтобы не только уметь с этими структурами работать, но и знать, как они внутри устроены (чтобы хорошо понимать возможности и ограничения этих структур), данную задачу вам необходимо будет выполнить как со стандартными классами из библиотеки Java (TreeMap<T>, HashMap<T>, LinkedHashMap<T>), так и со своей собственной реализацией данных структур. Не пугайтесь, в примерах к лекциям по деревьям поиска и хеш-таблицам эти структуры уже реализованы, и вам можно ими воспользоваться, но естественно, при желании вы можете выполнить полностью свою собственную реализацию указанных структур данных.

Собственные реализации структур должны соответствовать стандартному интерфейсу Map<K, V> и именно этот тип данных необходимо использовать в решении, но чтобы можно было как-то указать, какой конкретно тип данных необходимо использовать – стандартные классы или собственную реализацию (примерно также, как со списками – объявляем тип List<T>, а создаем экземпляры, как правило, ArrayList<T>). В вашем решении можно передавать конкретный экземпляр Map<T> в качестве параметра методу, который непосредственно реализует алгоритм задачи.

После решения задачи вы должны четко понимать, как рассматриваемые алгоритмы и структуры данных (деревья, хеш-таблицы) связаны со словарями (Map<T>) в языке Java.

В большинстве случаев (на усмотрение преподавателя) задачи предполагают ввод данных из файла и оконный интерфейс. В 2021 году допускается консольная реализация.

Если в условии задачи что-то непонятно – попросить пояснить преподавателя.

Варианты:

1. Отсортировать массив чисел с помощью словаря, основанного на бинарных деревьях поиска (стандартный TreeMap и собственная реализация/реализации). Для этого необходимо в качестве ключа использовать числа из массива, а в качестве значения – встречаемость данного числа. Далее пройти по словарю (порядок обхода будет соответствовать числам в отсортированном порядке и записать в исходный массив нужные числа требуемое число раз). Оцените сложность такого алгоритма сортировки.

2. Реализовать множество (класс – наследник Set<T>) на основе словаря (Map<K, V>). С использованием этого множества реализовать алгоритм, который по списку элементов строит другой список, в котором значения из переданного списка встречаются ровно по одному разу.

3. Реализовать очередь с приоритетами на основе дерева (или классов Map / Set, построенных на основе дерева).

4. Для массива/списка целых чисел найти все пары чисел, которые в сумме дают S (S задается пользователем). Алгоритм должен работать за время O(n*log(n)).

5. Для массива/списка целых чисел найти все тройки чисел, которые в сумме дают S (S задается пользователем). Алгоритм должен работать за время $O(n^2*\log(n))$.

6. Найти все позиции в массиве/списке целых чисел наибольшего из наиболее часто встречающихся значений. Алгоритм должен работать за время O(n*log(n)). Использовать структуру данных Map<Integer, List<Integer>> (ключ – число из массива, значение – позиции этого числа). Далее пройти по данному словарю для поиска нужного числа и его позиций.

7. (*) Есть файл с информацией об оценках студентов по предметам. Формат файла – CSV (предмет, ФИО, оценка). Оценка – число или зачет/незачет (т.е. строка). Записи в файле расположены в произвольном порядке. Обеспечить поиск информации обо всех оценках указанного студента. Использовать структуру данных Map<String, Map<String, String>> (Map<ФИО, Map<Предмет, Оценка>>). Реализовать в виде оконного приложения, где можно выбрать ФИО, а в таблице ниже отобразиться список оценок студента. (Задача повышенной сложности из-за того, что надо много всего сделать в плане реализации интерфейса, сами алгоритмы – тривиальны).

8. (*) Исходные данные соответствуют предыдущей задаче, только теперь необходимо найти все группы студентов имеющих абсолютно одинаковые оценки по предметам (вывести в виде (вариант набора оценок по предметам –> список студентов, имеющий такой набор оценок).
   Необходимо реализовать класс StudentMarks для хранения оценок одного студента. Данный класс должен быть сравним (Comparable<StudentMarks>). Проще всего внутри класса StudentMarks хранить оценки в виде словаря, а сравнение обеспечить по упорядоченным парам (предмет, оценка) – реализация на основе итераторов по словарям.
   Создаем словарь Map<ФИО, StudentMarks>, заполняем. Далее на основе данных в этом словаре создаем словарь Map<StudentMarks, List<ФИО>>, который и является ответом на задачу.
   Программа может быть консольной.

9. Найти N самых встречаемых слов в тексте длины не менее K букв (N и K задаются пользователем). Если какие-то слова используются одинаково часто, то их также вывести (т.е. если N равно 3-м, и есть одно слово, которое встречается 5 раз, и 7 слов, которые встречаются по 4 раза, а остальные слова встречаются реже, то необходимо вывести первые 8 слов). Одни и те же слова в различной форме считаются разными словами. Регистр букв при подсчете слов необходимо игнорировать (приводить слова к одному регистру). В качестве тестовых данных использовать литературные произведения (большие по объему тексты).

10. (*) Реализовать программу, которая ищет дубли файлов на диске (в указанной директории). Дублями считаются файлы с одинаковым содержимым (байт в байт). Для простоты сравнивать файлы будем не по содержимому, а по md5-хешу от содержимого (почитать про md5, код для подсчета md5 для файла можно найти в интернете, например, в этом обсуждении: https://stackoverflow.com/questions/304268/getting-a-files-md5-checksum-in-java).
    Собственно реализация: рекурсивно обходим файловую систему, начиная с указанной директории и в словарь по ключу в виде md5-хеша пишем в виде списка имена файлов. Далее проходим по словарю и для всех ключей, для которых в списке содержится более одного имени файла, распечатываем эти имена (дубли).

11. (*) Реализовать словарь PutOrderMap<K, V>, который обеспечивает итерацию по словарю в порядке добавления элементов. При этом все операции со словарем должны остаться эффективными (со сложностью не более O(log(n))). Для этого внутри словаря

должно храниться два других словаря, где в первом хранятся пары (ключ -> (значение + порядок добавления)), а во втором – пары (порядок добавления –> (значение + ключ)). Все операции поиска осуществляются по первому словарю, вставка / удаление затрагивают два словаря одновременно, а итерация проходит по второму словарю.

12. (*) Реализовать такое множество (Set<T>), для которого можно быстро, за время O(log(n)), находить K-ый по порядку элемент множества. Для этого необходимо разработать бинарное дерево поиска (можно модифицировать какой-то из классов SimpleRBTree / AVLTree / RBTree из проекта примеров к лекциям) такое, чтобы в каждом узле дерева дополнительно хранить информацию о количестве элементов в поддереве с вершиной в данном узле. Естественно, операции вставки и удаления элементов должны остаться O(log(n)). Далее на основе такого дерева необходимо реализовать множество, которое позволяет быстро найти K-ый по счету элемент.

13. Описать класс для представления Тарифов на международную связь. Данный класс хранит в себе список направлений в виде код (префикс) направления, название направления, цена минуты разговора. Класс должен уметь загружать информацию из текстового файла заданного формата (проще всего CSV), сохранять тарифы в такой же файл, иметь возможность добавления/удаления/модификации направления по префиксу. Внутри класс направления должны храниться в виде Map<Префикс, Направление>.
    И самое главное, класс должен уметь считать стоимость звонка, заданного вызываемым номером и длительностью в секундах. Для этого надо найти направление с самым длинным кодом (префиксом), подходящим к номеру, перевести длительность звонка в минуты (с округлением вверх, звонки короче 6 секунд не тарифицируются).

14. Реализовать приложение, которое осуществляет сравнение производительности различных вариантов Map<K, V> (как минимум, TreeMap<> и HashMap<>) для различных вариантов использования – только вставка случайных элементов, только поиск случайных элементов, только удаление случайных элементов, комбинация вставки и удаления случайных элементов. В качестве ключей использовать строки фиксированной длины (задается пользователем). Построить графики зависимости времени работы от количества вставляемых / запрашиваемых / удаляемых элементов. (При тестировании надо вначале генерировать последовательность действий (ключей), которые необходимо выполнить, а уже потом, по готовой последовательности ключей, измерять производительность последовательности операций, чтобы учитывать только время работы со словарем.)

15. Вывести в порядке встречаемости (от наиболее часто встречаемых) пары подряд идущих букв в тексте. Вывести информацию в виде – пара букв, относительная встречаемость (т.е. отношение кол-ва этих букв к общему количеству пар букв в тексте). В качестве тестовых данных использовать литературные произведения (большие по объему тексты). Сделать выводы, какие пары подряд идущих букв наиболее часто встречаются в русском и английском языке (эти пары, по идее, должны встречаться наиболее часто в различных текстах, т.к. это свойство языка, а не конкретного текста).

16. Найти в тексте все аббревиатуры (за таковые считать слова, состоящие только из заглавных букв, не длиннее 5 символов). Вывести информацию в виде аббревиатуры –> сколько раз встречается.

17. Найти в тексте все имена собственные (за таковые считать слова, которые начинаются с большой буквы не в начале предложения) и посчитать, сколько каждое из них встречается.

18. Найти в тексте наиболее часто встречающиеся словосочетания (два подряд идущих слова, не разделенных знаком препинания). Одни и те же слова в различной форме считаются разными словами. Регистр букв при подсчете словосочетаний необходимо игнорировать (приводить слова к одному регистру).

19. Получить список слов, которые встречаются в тексте ровно N раз (N задается пользователем). Одни и те же слова в различной форме считаются разными словами. В качестве тестовых данных использовать литературные произведения (большие по объему тексты).

20. (*) Реализовать словарь RndBSTreeMap<K, V> на основе рандомизированных деревьев поиска (https://habr.com/ru/post/145388/, https://neerc.ifmo.ru/wiki/index.php?title=Рандомизированное_бинарное_дерево_поиска). Имеет смысл для этого сначала сделать класс для рандомизированного дерева поиска RndBSTree<T> (лучше всего переделать класс SimpleBSTree из примеров к лекциям), а уже из него сразу же получится соответствующий словарь.

21. (*) Примитивный обфускатор Java-кода. Необходимо все идентификаторы в программе (слова, которые не являются ключевыми словами, методом main, некоторым стандартным набором типов данных, а также не являются частью строк "строка" и импортов) заменить на значения v1, v2, v3 и так далее. Одни и те же идентификаторы должны получить одно и то же новое имя. Программа по возможности должна остаться работоспособной.

22. (*) Реализовать дополнительный эффективный итератор по TreeMap по ключам от A до B включительно (эффективный означает, что перебираются только нужные элементы, а не все подряд). Если модифицировать проект с примерами к лекциям, то такой итератор следует сначала реализовать в DefaultBSTree (default-метод), а затем на его основе итератор в DefaultBSTreeMap (также default-метод). После этого данный итератор станет доступным в классах SimpleRBTreeMap / AVLTreeMap / RBTreeMap.

23.

24.

25.

26.

27.

28.

29.

30.

31.