Тестовое задание для стажера
Java-разработчик в команду корпоративной шины данных и микросервисов

Выбрал за основу TreeMap т. к. тут есть точная гарантия что будет не хуже log(n), в отличии от HashMap, где существует вероятность коллизии при хешировании и соответственно все может ухудшится до O(n), хоть и вероятность этого мала при хорошей функции хеширования.

Структура BTree даёт преимущества из-за меньшей ветвистости, к примеру каждое хранимое значение в RBtree или avlTree это отдельный узел и это дополнительный расход на хранение ссылки на него, заголовка объекта и его дочерних элементов, а внутри BTree несколько хранимых значений могут лежать внутри одного узла и следовательно экономить память.

Для сохранения account и быстрого поиска выбрал BTreeMap, внутри которой для ключей используются примитивы и это позволяет экономить память за счет хранения значений account в примитиве, а не в ссылочной обертке, что даст экономию в 16 байт за каждую сохранённую запись (т.к храня Long мы тратим память на хранение ссылки на него, на сам объект- 8 байт заголовок и 8 байт содержимое). Для сохранения value и быстрого поиска также выбрал BTreeMap что также дает экономию памяти за счет использования примитивов при хранении ключей.

При хранении строк выбрал byte[] т.к это снижает накладные расходы и мы храним только содержимое строки в юникоде в виде массива, а не обёртку в виде String, дополнительно расходующую память на себя. Т.к можно использовать Arrays.compare для сравнения массивов и там сравнивается их содержимое, TreeMap по именам правильно работает.

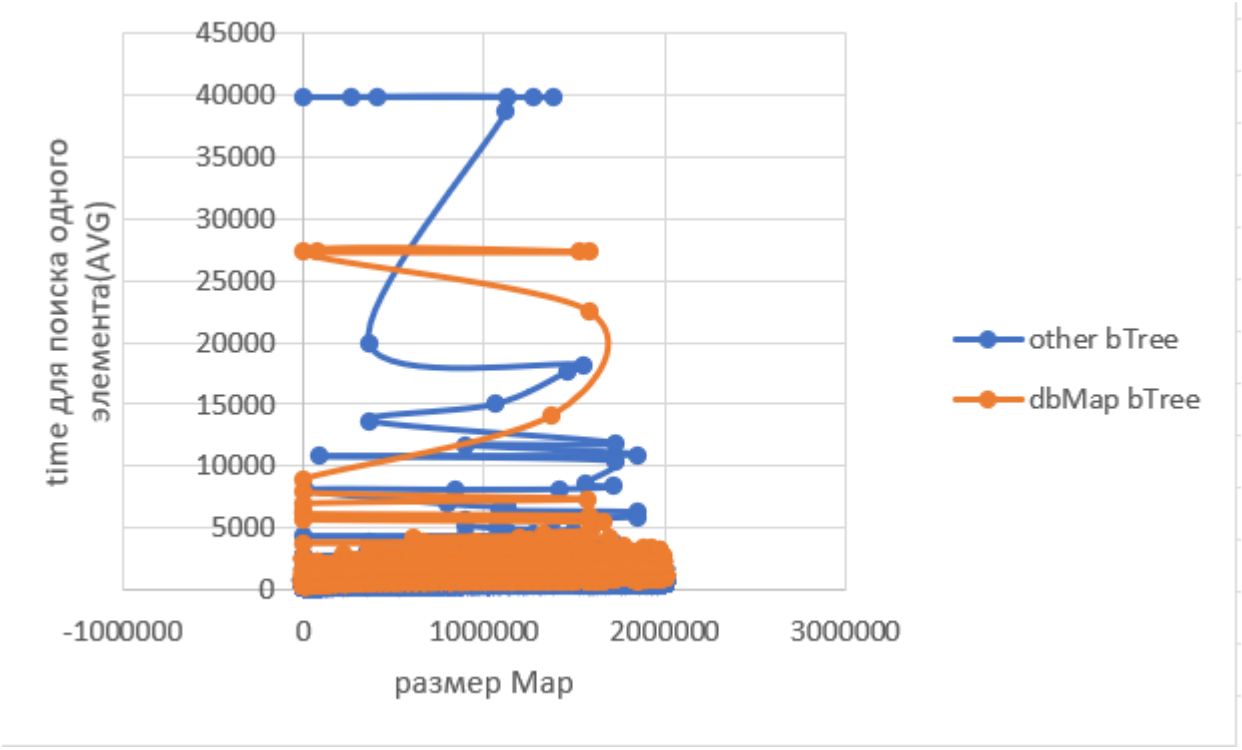
Саму запись acc, name, value храню в объекте соответствующего класса т. к. потом можно в кешах по различным полям хранить только ссылку на него, а сам объект создадим один раз для всех кешей.

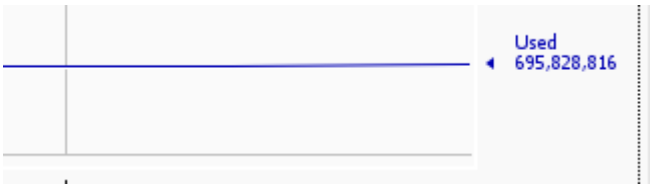
Там, где может быть несколько значений использовал ArrayList т. к. несмотря на то, что обычно его емкость выше, чем кол-во элементов, на не сильно больших размерах (от 3х элементов и до какого-то не сильно большого числа) он экономнее LinkedList, который хранит большое кол-во ссылок в своих узлах, т. к. это двухсвязный список.

Выбрал библиотеку MapDB и её BTreeMap т. к при тестировании эта структура показала минимальное потребление памяти в сравнении с uk.co.omegaprime.btreemap и it.unimi.dsi.fastutil. Насколько я понял это

вызвано тем, что в MapDB объект Node представлен внутри массивами, хранящими ключи и значения узла дерева, и размерность этих массивов соответствует кол-ву элементов данного узла. В отличии от других библиотек, где идет хранение не в массивах, а в отдельных полях, причем кол-во полей не соответствует кол-ву значений и, следовательно, может быть перерасход выделяемой на Node памяти, когда кол-во полей в Node 20, а реально хранимых элементов 6.

Тесты показали, что данная структура имеет сравнимую с её аналогами скорость по поиску значений (при использовании такого способа хранения как heapDB, т.к не идёт лишняя сериализация). Также путем внутреннего обследования структуры данных через дебаггер увидел упорядоченность, присущую В дереву, что даёт асимптотические оценки как у сбалансированного дерева поиска.





OTHER B-tree (2000000 entries)

With use mapDB bTree (2000000 entries)

Тестовое задание 2 для стажера
Java-разработчик в команду технологий фонт-офиса

O(1)

Для реализации O(1) надо будет использовать HashMap, тип ключа будет строка с номером телефона, а значением строка с именем клиента которую будем хранить в виде byte[] для экономии места, т.к врятли будет много дубликатов и следовательно пул строк String скорее всего не сэкономит существенное количество памяти, а затраты на память String превышают затраты на обычный byte[], поскольку придется выделять память на сам объект строки и на её содержимое. Используя Encoder для UTF-8, сможем быстро перевести байты в строку имени. Считать память будем по частям, сначала вычислим длину массива, на основе которого работает HashMap

С помощью небольшого приложения рассчитал размер мапы при условии, если заполненность массива достигает 75%, то массив копируется в новый массив длины в 2 раза больше предыдущего. Рассчитанная длина массива = 33554432 элемента

```
static int n = 18758328;
static int calc() {
    int cap = 16;
    for (int i = 1; i <= n; i++) {
        if (i >= cap * 0.75) {
            cap *= 2;
        }
    }
    return cap;
}
```

Перейдем к вычислению места на хранение мапы, будем учитывать только хранение массива, остальные расходы являются не существенными. Массив должен будет хранить 18758328 ссылок на хранимые данные, остальная емкость будет занята null значениями. Однако память выделяется под ту емкость которую я вычислил выше, получаем следующую оценку: 8 байт на заголовок объекта массива, 4 байта на хранение длины и 33554432 * 8 байт размер памяти занимаемой содержимым, поскольку массив состоит из ссылок а каждая ссылка занимает 8 байт в 64 битной JVM. Переводя в мегабайты все расходы получим **256 Мб.**

Теперь посчитаем сколько занимают объекты хранящие наши данные. Каждый элемент массива в хеш таблице занимает 8 байт на заголовок объекта, 4 байт на хранение значение хеша, 8 байт на хранение ссылки на ключ, 8 байт на хранение ссылки на значение и 8 байт на хранение следующего элемента для случая если возникают коллизии при хешировании разных объектов, т.к сумма не делится на 8, прибавляем

дополнительно 4 байта. Полученную сумму умножаем на кол-во элементов, заданное в задаче. Получим **716 Мегабайт.**

Последний шаг — это подсчет места для хранения значений наших данных.

Хранение номера идёт в строке. Строка будет иметь длину 16 символов, т.к 3 цифры это код страны, после идёт пробел и далее 12 цифр самого номера. Одна строка будет занимать 8 байт на заголовок объекта, 4 байта на хранение хеша, 8 байт на хранение ссылки на массив байт, хранящий содержимое строки и 4 байта для выравнивания, т. к. JVM нужно чтобы итоговый объем делился на 8.

Сам массив байт займет 8 байт на заголовок, 4 байта на хранение длины и 16 байт на хранение содержимого, т. к. весь набор символов для телефона умещается в ASCII то каждый символ занимает один байт, дополнительно прибавляем 4 байта для выравнивания. Умножая полученную сумму на объем данных, получим **1002 Мегабайта.**

Считать будем что у нас 100% имён на кириллице, один массив занимает 8 байт на заголовок, 4 байта на длину массива и 20*2 байта на содержимое и 4 байта для выравнивания. Просуммируя и умножив на объем данных и переведя в мегабайты получим **1002 Мегабайта**.

Теперь суммируем все вычисленное выше и получаем **2976 Мегабайт.** Оценка примерная т. к. JVM тоже потребляет память для своей работы и не факт, что каждый номер телефона будет в отдельной ячейке массива, поскольку могут быть коллизии, а HashMap может превратить элемент массива, хранящий коллекцию значений с одинаковым хешем в сбалансированное бинарное дерево поиска, и влияние таких моментов я не могу предусмотреть.

Самый экономный способ

Для экономии памяти пишем все в массив байт, предварительно проходимся по данным и считаем какую длину массива устанавливать учитывая, что после имени будем оставлять байт со значением 0, для того чтобы потом не возникали сложности с прочтением содержимого, дополнительно для простоты работы с массивом прибавим один байт для того, чтобы добавить 0 после последней пары имя-номер телефона

Далее создаем массив байт под наши входные данные, каждую строку для имени переводим в массив байт с кодировкой юникод, записываем в наш массив байт, далее пишем байт со значением 0, после этого переводим строку с номером телефона в массив байт используя кодировку UTF-8 и записываем, далее пишем байт со значением 0.

Для того чтобы найти телефон и полное имя клиента будем проходится по нашему массиву используя, например обход массива и

при каждом обнаружении байта со значением 0 будем понимать перед нами имя, если мы встретили данный байт на нечетном шаге или телефон если мы встретили на четном шаге. Дополнительно будем следить за тем какая позиция у предыдущей пары имя-телефон, для того чтобы получить позицию, где начинается байтовое представление текущей пары. Когда получили байты соответствующие паре то переводим их в строку используя кодировку юникод. Данный способ считаю работоспособным т. к. проверял как кодируются русские и английские буквы и лишних байтов со значением 0 я не заметил внутри бинарного представления строк, следовательно все должно однозначно раскодироваться, когда мы будем обратно считывать массив байт.

Расходы на хранение имён считаем в самом плохом случае, если нам все символы в именах на кириллице: (20 символов * 2 байта на символ) * кол-во пар клиент-телефон = 715 Мегабайт

Расходы на хранение номеров телефонов считаем исходя из того, что все цифры и символ пробела кодируются в UTF-8 одним битом для обратной совместимости с ASCII. Из 3х символов на код страны, 12 на номер телефона и пробел между кодом страны и номером, получаем: (16*1 байт) * кол-во пар клиент-телефон=286 мегабайт.

Дополнительно прибавляя кол-во байт равных кол-ву пар клиент-телефон, используемых для отделения имени от номера телефона получаем приблизительно такой объем памяти равный: 18 Мб + 715 Мб + 286 Мб= 1019 Мегабайт , точно рассчитать сложно из-за виртуальной машины джава и нюансов её работы.

Для дальнейшей оптимизации можно попробовать сжать полученный массив байт на своей рабочей машине через джавовский Deflater, записать получившийся массив байт в файл, также записать исходный массив, и если при сравнении объема побеждает сжатый вариант то осуществляем его перенос на тот компьютер где подразумевается хранить информацию, и потом уже работать не напрямую с массивом байт, а сначала распаковать часть сжатого массива во временный массив байт заранее выбранной нами длины, например 5 байт через Deflater, а потом производить чтение из временного массива и уже учитывать что мы не распаковываем весь исходный массив сразу экономя память за счет более сложной логики работы, поскольку теперь надо будет накапливать байты относящиеся к паре имя-телефон, которую мы пытаемся прочитать в данный момент и следить когда конкретно мы прочитали все байты относящиеся к паре

имя-телефон, основываясь на том же байте со значением 0 и уже после полного прочтения конвертировать прочитанную информацию в строку аналогично варианту без сжатия.

Объем памяти в таком случае зависит от внутренних свойств данных, в лучшем случае можно сократить расходы по памяти приблизительно в два раза. Вероятнее всего сэкономим не более сотни мегабайт. В худшем расходы немного возрастут.