**Для чего используются таблицы хеширования?**  
Хеш-таблицы используются для быстрого хранения, поиска, обновления и удаления данных по ключу. Они обеспечивают эффективный доступ к данным с помощью хеш-функции, которая преобразует ключ в индекс массива, где хранится значение. Это делает хеш-таблицы идеальными для задач, требующих высокой скорости, таких как словари, базы данных, кэширование, проверка уникальности (например, в множествах) и реализация ассоциативных массивов. Благодаря средней сложности операций O(1) O(1) O(1) при хорошей хеш-функции, они широко применяются в программировании и системах обработки данных.

**Что используется для программного поиска информации по содержанию?**  
Для программного поиска информации по содержанию используются структуры данных и алгоритмы, такие как хеш-таблицы, деревья поиска (например, бинарные деревья, AVL-деревья, красно-чёрные деревья), индексы в базах данных, а также алгоритмы полнотекстового поиска (например, инвертированные индексы или алгоритмы на основе трисов). Хеш-таблицы применяются, когда поиск осуществляется по точному ключу, тогда как для поиска по содержанию (например, частичному совпадению) чаще используются структуры, поддерживающие префиксный поиск, или специализированные алгоритмы, такие как BMH или KMP.

**Назовите основные функции хеширования.**  
Основные функции хеширования включают: 1) **Преобразование ключа в индекс**: хеш-функция преобразует входной ключ (строку, число и т.д.) в числовой индекс для доступа к массиву. 2) **Обеспечение равномерного распределения**: хорошая хеш-функция минимизирует коллизии, распределяя ключи равномерно по таблице. 3) **Детерминированность**: для одного и того же ключа всегда возвращается одинаковый хеш. 4) **Эффективность**: хеш-функция должна быть быстрой для вычисления, чтобы обеспечить высокую производительность операций. Примеры включают деление, умножение или сложение байтов ключа.

**Приведите структуру хеш-таблицы.**  
Хеш-таблица состоит из массива фиксированного размера (слотов), где каждый слот может содержать данные или указатель на структуру данных (например, список или дерево). Каждая ячейка хранит пару ключ-значение или ссылку на них. В hash.txt структура ячейки включает ключ, значение и флажки (C, U, T, L, D, P0, Pi) для обработки коллизий и удаления. В реализации с цепочками (как в 6lab.txt) слот содержит указатель на AVL-дерево, где хранятся ключи с одинаковым хешем. Хеш-функция определяет, в какой слот помещается ключ.

**Приведите способы перевода ключевых слов в числовые значения.**

1. **Суммирование кодов символов**: суммируются ASCII/Unicode-коды символов ключа (например, ABC → 65+66+67). 2) **Полиномиальный метод**: каждый символ умножается на степень основания (например, V=s0⋅pn+s1⋅pn−1+… V = s\_0 \cdot p^n + s\_1 \cdot p^{n-1} + \ldots V=s0​⋅pn+s1​⋅pn−1+…, где p p p — константа). 3) **Метод первых символов**: используются коды или индексы первых k k k символов (как в hash.txt: V=индекс\_первой\_буквы⋅26+индекс\_второй\_буквы V = \text{индекс\\_первой\\_буквы} \cdot 26 + \text{индекс\\_второй\\_буквы} V=индекс\_первой\_буквы⋅26+индекс\_второй\_буквы). 4) **Свертка**: ключ разбивается на блоки, и их значения комбинируются (например, через XOR или сложение). 5) **Криптографические хеш-функции** (MD5, SHA): для сложных ключей, но медленнее.

**Приведите способы преобразования числового значения ключевого слова в хеш-адрес.**

1. **Метод деления**: h=Vmod  M h = V \mod M h=VmodM, где V V V — числовое значение ключа, M M M — размер таблицы (как в hash.txt, h=Vmod  20 h = V \mod 20 h=Vmod20). 2) **Метод умножения**: h=⌊M⋅(V⋅kmod  1)⌋ h = \lfloor M \cdot (V \cdot k \mod 1) \rfloor h=⌊M⋅(V⋅kmod1)⌋, где k k k — константа (например, золотое сечение). 3) **Серединный квадрат**: берутся средние цифры V2 V^2 V2. 4) **Сдвиг и маскирование**: применяются побитовые операции для получения индекса. Метод деления наиболее распространён из-за простоты и эффективности, особенно если M M M — простое число, чтобы минимизировать коллизии.

**Что такое коллизия? В каких случаях она возникает?**  
Коллизия — это ситуация, когда два разных ключа получают одинаковый хеш-адрес (h h h) после применения хеш-функции. Она возникает, когда хеш-функция отображает разные ключи в один и тот же индекс массива, что неизбежно при ограниченном размере таблицы (M M M) и большом числе ключей (из-за принципа Дирихле). Например, в hash.txt ключи Ковалев, Козлов, Кот дают V=378 V = 378 V=378, h=18 h = 18 h=18. Коллизии чаще происходят при плохой хеш-функции, малом размере таблицы или высокой плотности данных.

**Назовите способы обработки коллизий.**

1. **Цепочки (область переполнения)**: ключи с одинаковым хешем хранятся в структуре данных (например, список, AVL-дерево, как в 6lab.txt) в одном слоте. 2) **Открытая адресация (пробинг)**: при коллизии ищется следующая свободная ячейка: а) **Линейный пробинг** (проверяет h,h+1,h+2,… h, h+1, h+2, \ldots h,h+1,h+2,…); б) **Квадратичный пробинг** (h,h+1,h+4,h+9,… h, h+1, h+4, h+9, \ldots h,h+1,h+4,h+9,…); в) **Двойное хеширование** (использует вторую хеш-функцию для шага). 3) **Реорганизация таблицы**: увеличение размера таблицы и перехеширование при высокой загрузке. В hash.txt используется линейный пробинг, а в 6lab.txt — цепочки.

**В каких случаях производится процедура пробинга?**  
Процедура пробинга применяется в хеш-таблицах с открытой адресацией, когда происходит коллизия, и нужно найти свободную ячейку для размещения ключа. Пробинг запускается при вставке (если слот занят), поиске (для проверки всех возможных ячеек) или удалении (для поиска ключа). Например, в hash.txt при вставке ключа с h=18 h = 18 h=18 (если ячейка занята) линейный пробинг проверяет ячейки 19, 0, 1 и т.д., пока не найдёт свободную или подходящую. Пробинг не используется в цепочках, как в 6lab.txt, где коллизии разрешаются через AVL-дерево.

**Какой порядок элементов в хеш-таблице?**  
В хеш-таблице элементы не имеют фиксированного порядка, так как их расположение определяется хеш-функцией, которая преобразует ключ в индекс массива (слот). Порядок зависит от значений хеш-адресов (h=hash(key)mod  table\_size h = \text{hash}(key) \mod \text{table\\_size} h=hash(key)modtable\_size) и метода разрешения коллизий. Например, в реализации с цепочками (как в 6lab.txt с AVL-деревом) элементы с одинаковым хешем хранятся в AVL-дереве, отсортированном по ключам, но между слотами нет порядка (слоты заполняются хаотично). В хеш-таблицах с открытой адресацией (например, линейный пробинг в hash.txt) элементы располагаются в слотах по правилу пробинга, что также не гарантирует логический порядок (например, алфавитный или по времени вставки). Таким образом, хеш-таблицы не поддерживают упорядоченность, и для получения элементов в определённом порядке требуется дополнительная обработка (например, сортировка).

**В чем преимущества хеш-таблицы?**  
Хеш-таблицы обладают несколькими преимуществами: 1) **Высокая скорость операций**: в среднем операции вставки, поиска, обновления и удаления выполняются за O(1) O(1) O(1), если хеш-функция равномерно распределяет ключи и коллизии редки. 2) **Эффективное использование памяти**: при правильном выборе размера таблицы и низкой загрузке хеш-таблицы компактны. 3) **Гибкость ключей**: поддерживают ключи любого типа (строки, числа, объекты), если есть хеш-функция. 4) **Широкое применение**: идеальны для словарей, кэшей, баз данных и проверки уникальности. Например, в 6lab.txt хеш-таблица с AVL-деревом обеспечивает быстрый доступ к данным по ключам вроде Dima. Эти преимущества делают хеш-таблицы популярными в высокопроизводительных системах.

**В чем недостатки хеш-таблицы?**  
Хеш-таблицы имеют и недостатки: 1) **Коллизии**: при плохой хеш-функции или высокой загрузке коллизии увеличивают время операций (до O(n) O(n) O(n) в худшем случае, например, если все ключи попадают в один слот). 2) **Отсутствие порядка**: как упомянуто, элементы не упорядочены, что затрудняет задачи, требующие сортировки или обхода. 3) **Зависимость от хеш-функции**: неравномерное распределение ключей снижает производительность. 4) **Дополнительная память**: при использовании цепочек (как в 6lab.txt) или пробинга требуется память для хранения структур данных (деревьев, списков) или резервных ячеек. 5) **Сложность настройки**: выбор размера таблицы и хеш-функции требует оптимизации, чтобы минимизировать коллизии. Например, в hash.txt линейный пробинг может привести к кластеризации, увеличивая время поиска. Эти недостатки делают хеш-таблицы менее подходящими для задач с упорядоченными данными или частыми коллизиями.

**Используем ли мы такую же хеш-функцию в нашем коде?**  
В последней версии кода (с латинским алфавитом) хеш-функция аналогична описанной: V=индекс\_первой\_буквы⋅26+индекс\_второй\_буквы V = \text{индекс\\_первой\\_буквы} \cdot 26 + \text{индекс\\_второй\\_буквы} V=индекс\_первой\_буквы⋅26+индекс\_второй\_буквы для A=0, ..., Z=25. Для русского алфавита (в более ранней версии) использовалась та же формула: V=индекс\_первой\_буквы⋅33+индекс\_второй\_буквы V = \text{индекс\\_первой\\_буквы} \cdot 33 + \text{индекс\\_второй\\_буквы} V=индекс\_первой\_буквы⋅33+индекс\_второй\_буквы, где А=0, ..., Я=32, как в примере (например, V[Вя]=2⋅33+32=98 V[\text{Вя}] = 2 \cdot 33 + 32 = 98 V[Вя]=2⋅33+32=98). Таким образом, хеш-функция идентична по принципу.

**Способ формирования хеш-адреса**  
В коде хеш-адрес вычисляется как h=Vmod  H h = V \mod H h=VmodH, где H=20 H = 20 H=20 (размер таблицы), что соответствует h(V)=Vmod  H h(V) = V \mod H h(V)=VmodH из описания. Параметр B B B не используется (B=0 B = 0 B=0).

**Структура ячейки и размер таблицы**  
Структура ячейки в коде отличается: вместо флажков (C, U, T, L, D, P0, Pi) из hash.txt используется AVL-дерево в каждом слоте, хранящее пары ключ-значение. Размер таблицы H=20 H = 20 H=20, как в примере.

**Способы обработки коллизий**  
Код использует **цепочки** (область переполнения) с AVL-деревом, а не внутреннюю адресацию с резервными ячейками, как в hash.txt. Коллизии разрешаются путём добавления ключей в дерево в одном слоте.

**Пробинг**  
Пробинг (линейный) в коде не применяется, так как используется метод цепочек, а не внутренняя адресация. Линейный пробинг упомянут в hash.txt, но не реализован в 6lab.txt.

**Принцип работы сбалансированного дерева (AVL-дерева) для разрешения коллизий в хеш-таблице**

В хеш-таблице из 6lab.txt для разрешения коллизий используется метод **цепочек**, где каждый слот таблицы (table[i]) содержит указатель на **AVL-дерево**. AVL-дерево — это сбалансированное бинарное дерево поиска, которое обеспечивает логарифмическую сложность операций (O(log⁡n) O(\log n) O(logn)) за счёт поддержания баланса. Вот как это работает:

1. **Коллизия в хеш-таблице**:
   * Когда два ключа (например, Dima и Dino) получают одинаковый хеш-адрес (h=6 h = 6 h=6, так как V=3⋅26+8=86 V = 3 \cdot 26 + 8 = 86 V=3⋅26+8=86, h=86mod  20=6 h = 86 \mod 20 = 6 h=86mod20=6), они должны быть помещены в один слот таблицы.
   * Вместо хранения ключей в простом списке, слот содержит AVL-дерево, в которое добавляются все ключи с этим хешем.
2. **Структура AVL-дерева**:
   * Каждый узел дерева (AVLNode) хранит ключ, значение, указатели на левое и правое поддерево, а также высоту для отслеживания баланса.
   * Дерево организовано как бинарное дерево поиска: ключи в левом поддереве меньше текущего узла, в правом — больше (по лексикографическому порядку).
3. **Балансировка**:
   * AVL-дерево поддерживает баланс, обеспечивая, что разница высот левого и правого поддерева (balance factor=height(left)−height(right) \text{balance factor} = \text{height(left)} - \text{height(right)} balance factor=height(left)−height(right)) находится в диапазоне [−1,1][-1, 1][−1,1].
   * При вставке или удалении, если баланс нарушается (∣balance factor∣>1 |\text{balance factor}| > 1 ∣balance factor∣>1), выполняются **повороты** (Left-Left, Right-Right, Left-Right, Right-Left), чтобы восстановить баланс. Например:
     + **Left-Left**: поворот вправо для случая, когда левое поддерево слишком глубокое.
     + **Right-Right**: поворот влево для правого поддерева.
     + **Left-Right**, **Right-Left**: комбинация поворотов для смешанных случаев.
4. **Операции в AVL-дереве для коллизий**:
   * **Вставка (insert)**: ключ добавляется в дерево по правилам бинарного поиска. Если ключ уже существует, обновляется его значение. После вставки проверяется баланс, и при необходимости выполняются повороты.
   * **Поиск (search)**: ключ ищется в дереве, начиная с корня, сравнивая его с узлами. Если ключ найден, возвращается значение; иначе выбрасывается исключение.
   * **Удаление (remove)**: ключ удаляется из дерева, после чего выполняется ребалансировка, если структура изменилась.
   * Все операции имеют сложность O(log⁡n) O(\log n) O(logn), где n n n — число ключей в дереве (то есть ключей с одинаковым хешем).
5. **Преимущества использования AVL-дерева для коллизий**:
   * **Эффективность**: Логарифмическая сложность (O(log⁡n) O(\log n) O(logn)) быстрее, чем линейный поиск в списке (O(n) O(n) O(n)), особенно при большом числе коллизий.
   * **Автоматическая балансировка**: Поддерживает высоту дерева близкой к log⁡n \log n logn, минимизируя время операций.
   * **Упорядоченность**: Ключи в дереве хранятся отсортированными, что может быть полезно для дополнительных операций (хотя в хеш-таблице это не требуется).
6. **Как это реализовано в 6lab.txt**:
   * Каждый слот table[h] — это AVLTree\*, куда добавляются ключи с хеш-адресом h h h.
   * При вставке (ht.insert("Dima", "Carpenter")) хеш-функция вычисляет h=6 h = 6 h=6, и пара ключ-значение добавляется в table[6] (AVL-дерево).
   * Если вставляется Dino (тоже h=6 h = 6 h=6), он добавляется в то же дерево, которое балансируется при необходимости.
   * Поиск, обновление и удаление работают аналогично, обращаясь к нужному дереву по хешу.
7. **Сравнение с hash.txt**:
   * В hash.txt коллизии разрешаются через **линейный пробинг** (внутренняя адресация), где при коллизии ищется следующая свободная ячейка.
   * В 6lab.txt цепочки с AVL-деревом более эффективны для большого числа коллизий, так как пробинг может привести к кластеризации (многие ключи в соседних ячейках), увеличивая время поиска.

**Кратко**: AVL-дерево в хеш-таблице хранит ключи с одинаковым хеш-адресом в сбалансированной структуре бинарного поиска, обеспечивая быструю вставку, поиск и удаление (O(log⁡n) O(\log n) O(logn)) за счёт автоматической балансировки через повороты, что делает его эффективным для разрешения коллизий по сравнению с простыми списками или пробингом