Если вы выбрали **Separate Chaining Hashmap** для реализации в рамках лабораторной работы, то вы будете работать с хеш-таблицей, использующей метод цепочек для разрешения коллизий. Этот метод заключается в том, что каждый элемент хеш-таблицы представляет собой связанный список, который хранит все элементы с одинаковым хеш-значением.

**1. Основные понятия Separate Chaining Hashmap**

* **Хеш-таблица (Hashmap)** — это структура данных, которая хранит элементы в виде пар ключ-значение. Она использует хеш-функцию для вычисления индекса в массиве для каждого ключа.
* **Метод цепочек (Separate Chaining)** — при коллизии (когда два ключа имеют одинаковое хеш-значение) создается список элементов, которые хешируются в тот же индекс, а не перезаписывается значение.

**Структура данных:**

* Массив с фиксированным размером, где каждый элемент — это указатель на связанный список, хранящий пары **ключ-значение**.

**2. Типы операций для sc-dict (Separate Chaining Hashmap)**

**2.1. Добавление элемента (insert)**

* При добавлении элемента вычисляется его хеш, и элемент вставляется в соответствующий список в соответствующем индексе хеш-таблицы.

**2.2. Поиск элемента (lookup)**

* Для поиска значения по ключу вычисляется хеш этого ключа, и затем происходит поиск в списке, который соответствует этому хешу.

**2.3. Удаление элемента (remove)**

* Для удаления элемента из хеш-таблицы также нужно вычислить хеш ключа и затем пройти по связанному списку в соответствующем индексе, чтобы найти и удалить элемент.

**2.4. Фильтрация (filter)**

* Функция, которая применяет фильтрацию (например, по значению или ключу) ко всем элементам в хеш-таблице.

**2.5. Отображение (map)**

* Функция, которая применяет операцию ко всем элементам (например, изменяет значение или выполняет другую операцию с ключом и значением).

**2.6. Свертки (fold)**

* Логика свертки для агрегации данных из хеш-таблицы.

**3. Реализация sc-dict**

Теперь давайте рассмотрим пример реализации хеш-таблицы с методом цепочек.

**3.1. Определение структуры данных**

Для начала определим структуру хеш-таблицы, которая использует отдельные цепочки для разрешения коллизий.

-- Узел для связанного списка

data Node k v = Node k v (Node k v) | EmptyNode

-- Хеш-таблица, которая хранит список для каждой позиции в массиве

data HashMap k v = HashMap { size :: Int, buckets :: [Node k v] }

-- Пустая хеш-таблица

empty :: HashMap k v

empty = HashMap 0 (replicate 16 EmptyNode)

**3.2. Хеш-функция**

Для начала нужно будет определить хеш-функцию, которая преобразует ключ в индекс массива.

haskell

Copy code

-- Пример хеш-функции (простейшая для целых чисел)

hash :: Int -> Int

hash k = k `mod` 16

**3.3. Операция добавления (insert)**

-- Вставка нового элемента в хеш-таблицу

insert :: (Eq k) => k -> v -> HashMap k v -> HashMap k v

insert key value (HashMap size buckets) = HashMap size (insertInBucket index key value (buckets !! index) : take index buckets ++ drop (index + 1) buckets)

where

index = hash key

-- Вставка в цепочку

insertInBucket :: (Eq k) => Int -> k -> v -> Node k v -> Node k v

insertInBucket \_ key value EmptyNode = Node key value EmptyNode

insertInBucket idx key value (Node k v next)

| key == k = Node key value next -- Если ключ уже есть, заменяем значение

| otherwise = Node k v (insertInBucket idx key value next)

**3.4. Операция поиска (lookup)**

-- Поиск элемента по ключу

lookup :: (Eq k) => k -> HashMap k v -> Maybe v

lookup key (HashMap \_ buckets) = lookupInBucket (hash key) key (buckets !! (hash key))

where

-- Поиск в цепочке

lookupInBucket :: (Eq k) => Int -> k -> Node k v -> Maybe v

lookupInBucket \_ \_ EmptyNode = Nothing

lookupInBucket idx key (Node k v next)

| key == k = Just v

| otherwise = lookupInBucket idx key next

**3.5. Операция удаления (remove)**

-- Удаление элемента по ключу

remove :: (Eq k) => k -> HashMap k v -> HashMap k v

remove key (HashMap size buckets) = HashMap size (removeFromBucket (hash key) key (buckets !! (hash key)) : take (hash key) buckets ++ drop (hash key + 1) buckets)

where

removeFromBucket :: (Eq k) => Int -> k -> Node k v -> Node k v

removeFromBucket \_ \_ EmptyNode = EmptyNode

removeFromBucket idx key (Node k v next)

| key == k = next -- Удаляем текущий узел

| otherwise = Node k v (removeFromBucket idx key next)

**3.6. Операция фильтрации (filter)**

-- Фильтрация элементов хеш-таблицы

filterDict :: (Eq k) => (k -> v -> Bool) -> HashMap k v -> HashMap k v

filterDict predicate (HashMap size buckets) = HashMap size (map (filterBucket predicate) buckets)

where

filterBucket :: (Eq k) => (k -> v -> Bool) -> Node k v -> Node k v

filterBucket \_ EmptyNode = EmptyNode

filterBucket pred (Node k v next)

| pred k v = Node k v (filterBucket pred next)

| otherwise = filterBucket pred next

**3.7. Операция отображения (map)**

-- Отображение значений с применением функции

mapDict :: (k -> v -> a) -> HashMap k v -> HashMap k a

mapDict f (HashMap size buckets) = HashMap size (map (mapBucket f) buckets)

where

mapBucket :: (k -> v -> a) -> Node k v -> Node k a

mapBucket \_ EmptyNode = EmptyNode

mapBucket f (Node k v next) = Node k (f k v) (mapBucket f next)

**4. Тестирование**

**4.1. Юнит-тесты**

Можно написать простые тесты для проверки работы операций. Например, тест на добавление элемента:

testInsert :: Bool

testInsert = lookup 5 (insert 5 "five" empty) == Just "five"

**4.2. Property-based тесты**

Для тестов на свойства можно использовать библиотеку, как QuickCheck, для проверки свойств, таких как ассоциативность операций и корректность работы с хеш-таблицей.

prop\_insert\_lookup :: Int -> String -> HashMap Int String -> Bool

prop\_insert\_lookup k v map = lookup k (insert k v map) == Just v

**5. Заключение**

В этой реализации мы построили хеш-таблицу с использованием метода цепочек для разрешения коллизий. Мы реализовали базовые операции: добавление, поиск, удаление, фильтрацию и отображение, а также написали несколько тестов, чтобы проверить корректность работы структуры данных.

Для успешного выполнения лабораторной работы вам нужно будет дополнительно написать более сложные тесты и проработать особенности производительности, учитывая неизменяемость структуры данных и требования к полиморфизму и моноидным свойствам.

أكيد! خليني أشرح لك خطوة بخطوة وباللهجة السورية عن كيفية بناء **هاش ماب (Hashmap)** باستخدام طريقة **Separate Chaining** وكيفية تنفيذ العمليات عليها.

**1. شو هي Separate Chaining Hashmap؟s**

الـ **Hashmap** هي هيكل بيانات بيخزن العناصر بشكل "مفتاح - قيمة" (key-value)، وبيستخدم دالة **هاش (Hash)** ليوزع العناصر داخل مصفوفة.

أما **طريقة Separate Chaining** هي طريقة لحل مشكلة **الاصطدامات (Collisions)** لما يكون في مفتاحين بيأخذوا نفس قيمة الهاش. بدل ما نعيد حساب الهاش أو نكتب فوق القيمة القديمة، بنخزن كل العناصر اللي جابوا نفس القيمة في "سلسلة" (Linked List) واحدة بكل خانة من المصفوفة.

**2. كيف بيشتغل ال Separate Chaining؟**

1. **المصفوفة**: بنستخدم مصفوفة ذات حجم ثابت لتخزين القيم. كل عنصر في المصفوفة هو عبارة عن **سلسلة مرتبطة** (Linked List).
2. **دالة الهاش**: لما بدنا نخزن قيمة معينة، بنحسب لها هاش وبنخزنها في المصفوفة في المكان يلي طلعناه من الهاش.
3. **الاصطدامات (Collisions)**: إذا صار في مفتاحين بيأخذوا نفس قيمة الهاش، بدل ما نكتب فوق العنصر القديم، بنضيف العنصر الجديد في السلسلة المرتبطة.

**3. العمليات الرئيسية على sc-dict**

**3.1 إضافة عنصر (insert)**

أول شي بنحسب الهاش للمفتاح، وبنخزّن القيمة ضمن السلسلة المرتبطة في الخانة المناسبة بالمصفوفة.

**3.2 بحث عن عنصر (lookup)**

بنحسب الهاش للمفتاح وبنبحث في السلسلة المرتبطة في الخانة المناسبة بالمصفوفة.

**3.3 حذف عنصر (remove)**

بنحسب الهاش للمفتاح وبعدين منمر على السلسلة المرتبطة في الخانة المناسبة ونحذف العنصر إذا كان موجود.

**3.4 فلترة (filter)**

بتطبق شرط معين على كل العناصر وبتفلترهم.

**3.5 تطبيق دالة على العناصر (map)**

بتطبق دالة معينة على كل العناصر الموجودة في الـ **Hashmap**.

**3.6 سويْرت أو جمع (fold)**

بتعمل جمع أو سويْرت للعناصر الموجودة.

**4. كيفية بناء الـ sc-dict باستخدام Haskell**

**4.1 تعريف الـ Hashmap**

أولاً، بنعرف هيكل البيانات للمصفوفة اللي بتخزن السلاسل المرتبطة.

-- عقدة في السلسلة المرتبطة

data Node k v = Node k v (Node k v) | EmptyNode

-- هاش ماب (Hashmap) بيحتوي على حجم المصفوفة و المصفوفة نفسها

data HashMap k v = HashMap { size :: Int, buckets :: [Node k v] }

-- هاش ماب فارغ

empty :: HashMap k v

empty = HashMap 0 (replicate 16 EmptyNode)

* **Node**: هي عقدة في السلسلة المرتبطة. بتحتوي على المفتاح، القيمة، والعقدة الجاية.
* **EmptyNode**: بتمثل النهاية أو حالة السلسلة الفارغة.

**4.2 دالة الهاش**

بنحسب الهاش باستخدام عملية بسيطة:

-- دالة هاش للمفاتيح من نوع Int

hash :: Int -> Int

hash k = k `mod` 16

**4.3 إضافة عنصر (insert)**

لما نضيف عنصر جديد، بنحسب الهاش للمفتاح وبنضيفه في السلسلة المناسبة في المصفوفة.

-- إضافة عنصر في هاش ماب

insert :: (Eq k) => k -> v -> HashMap k v -> HashMap k v

insert key value (HashMap size buckets) = HashMap size (insertInBucket index key value (buckets !! index) : take index buckets ++ drop (index + 1) buckets)

where

index = hash key -- نحسب الهاش

-- إضافة العنصر في السلسلة

insertInBucket :: (Eq k) => Int -> k -> v -> Node k v -> Node k v

insertInBucket \_ key value EmptyNode = Node key value EmptyNode

insertInBucket idx key value (Node k v next)

| key == k = Node key value next -- إذا المفتاح موجود، نغير القيمة

| otherwise = Node k v (insertInBucket idx key value next)

**4.4 بحث عن عنصر (lookup)**

لما بدنا نبحث عن عنصر، بنحسب الهاش وبنمر على السلسلة المرتبطة اللي في نفس المكان.

-- بحث عن عنصر في هاش ماب

lookup :: (Eq k) => k -> HashMap k v -> Maybe v

lookup key (HashMap \_ buckets) = lookupInBucket (hash key) key (buckets !! (hash key))

where

-- بحث في السلسلة

lookupInBucket :: (Eq k) => Int -> k -> Node k v -> Maybe v

lookupInBucket \_ \_ EmptyNode = Nothing

lookupInBucket idx key (Node k v next)

| key == k = Just v -- إذا لقيت المفتاح، رجع القيمة

| otherwise = lookupInBucket idx key next

**4.5 حذف عنصر (remove)**

لما نريد نحذف عنصر، بنحسب الهاش للمفتاح ونمر على السلسلة المرتبطة ونحذفه إذا كان موجود.

-- حذف عنصر من هاش ماب

remove :: (Eq k) => k -> HashMap k v -> HashMap k v

remove key (HashMap size buckets) = HashMap size (removeFromBucket (hash key) key (buckets !! (hash key)) : take (hash key) buckets ++ drop (hash key + 1) buckets)

where

removeFromBucket :: (Eq k) => Int -> k -> Node k v -> Node k v

removeFromBucket \_ \_ EmptyNode = EmptyNode

removeFromBucket idx key (Node k v next)

| key == k = next -- نحذف العنصر

| otherwise = Node k v (removeFromBucket idx key next)

**4.6 فلترة (filter)**

تطبق دالة الفلترة على كل العناصر وبتحذف العناصر اللي ما بتطابق الشرط.

-- فلترة عناصر هاش ماب

filterDict :: (Eq k) => (k -> v -> Bool) -> HashMap k v -> HashMap k v

filterDict predicate (HashMap size buckets) = HashMap size (map (filterBucket predicate) buckets)

where

filterBucket :: (Eq k) => (k -> v -> Bool) -> Node k v -> Node k v

filterBucket \_ EmptyNode = EmptyNode

filterBucket pred (Node k v next)

| pred k v = Node k v (filterBucket pred next)

| otherwise = filterBucket pred next

**5. الاختبارات**

**5.1 اختبار الوحدة (Unit Test)**

مثال لاختبار إضافة عنصر:

testInsert :: Bool

testInsert = lookup 5 (insert 5 "five" empty) == Just "five"

**5.2 اختبار الخصائص (Property-based Test)**

اختبار للخاصية: "إذا أضفنا عنصر للماب، لازم نقدر نبحث عنه ونرجع قيمته":

prop\_insert\_lookup :: Int -> String -> HashMap Int String -> Bool

prop\_insert\_lookup k v map = lookup k (insert k v map) == Just v

**6. الخلاصة**

في هالمثال، بنيت **هاش ماب** باستخدام طريقة **Separate Chaining** لتخزين العناصر بشكل **غير قابل للتغيير**. أضفنا العمليات الأساسية مثل **إضافة العناصر**، **البحث**، **الحذف**، **الفلترة**، و **تطبيق دالة على العناصر**. بالإضافة إلى أنه تم تطبيق بعض **الاختبارات** لضمان عمل العمليات بشكل صحيح.