

СОДЕРЖАНИЕ

АКТУАЛЬНОСТЬ	5
ВВЕДЕНИЕ	7
ОБЗОР АНАЛОГОВ	10
1 ОПИСАНИЕ HASH ARRAY MAPPED TRIE.....	13
1.1 Описание Trie	13
1.2 Описание однопоточной Hash Array Mapped Trie	17
1.3 Описание lock-free Hash Array Mapped Trie.....	24
2 РАЗРАБОТКА	29
2.1 Настройка рабочего окружения.....	29
2.2 Реализация однопоточной версии	33
2.3 Реализация блокирующей многопоточной версии	38
2.4 Реализация неблокирующей версии.....	39
2.5 Решение проблемы АВА	41
2.6 Внедрение в библиотеку libcds	42
3 ТЕСТИРОВАНИЕ	43
4 ЗАКЛЮЧЕНИЕ	46
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	47

АКТУАЛЬНОСТЬ

От современного программного обеспечения требуется всё больше производительности. Один из способов улучшить показатели – это использовать многопоточные алгоритмы. В определённых случаях они оказываются быстрее однопоточных. Если в однопоточной версии единственный поток выполняет всю работу единолично, то в многопоточной несколько потоков делят работу на меньшие части и выполняют их одновременно (это позволяет системе прогрессировать быстрее).

Сложность написания таких алгоритмов заключается в том, что потоки зачастую имеют доступ к одним и тем же данным. При одновременной работе с ними, данные могут прийти в неконсистентное состояние. Это вызовет неправильную работу самих потоков (гонки данных), а значит, системы в целом.

Проблема с некорректной работы с данными может решаться через блокировки потенциально опасных частей кода. В таких критических местах может исполняться одновременно лишь один поток, что делает работу с данными безопасной. Пока один поток работает в критической секции, другие, желающие исполниться, ожидают (останавливают свою работу). Такой подход иногда приводит к снижению производительности из-за затрат на использование критических секций. Более того, при неправильной реализации возможна бесконечная остановка системы, когда каждый поток входит в ожидание.

Альтернативой являются алгоритмы без блокировок (lock-free). В них потоки постоянно выполняют необходимую работу, не ожидая вход в какой-либо участок кода. Такие алгоритмы могут улучшить производительность за счёт отсутствия бесконечной остановки системы, отсутствия накладных расходов на критические секции и отсутствия ожидания потоков.

Использование lock-free подхода распространено в библиотеке libcds (библиотека для многопоточных структур данных). Она включает в себя множество высокопроизводительных, неблокирующих структур на языке C++. Например, многопоточные реализации хеш-таблиц (MichaelHashMap, FeldmanHashMap) или деревьев (BronsonAvlTree). Такие структуры данных зачастую необходимы, так как они решают распространённую в программировании задачу поддержания словаря. К сожалению, во многих случаях их производительности недостаточно из-за особенностей таких структур данных. Например, хеш-таблицы в некоторые моменты снижают производительность из-за внутреннего устройства, а определённые структуры данных основанных на деревьях часто имеют большое время исполнения операций. Компромиссным решением для эффективного поддержания словаря является структура данных Hash Array Mapped Trie. Она сочетает в себе свойства и деревьев, и хеш-таблиц.

Среди аналогов библиотеки libcds (таких как boost, ConcurrencyKi), как и в самой библиотеке, неблокирующая структура данных Hash Array Mapped Trie не реализована. Поэтому появилась необходимость внедрения такой структуры в библиотеку libcds. Она может быть производительнее своих аналогов в задаче поддержания словаря.

ВВЕДЕНИЕ

В многопоточных приложениях потокам (отдельные единицы исполнения кода) зачастую необходима синхронизация через передачу друг другу данных, обмен сигналами, сообщений. Потоки могут делать это посредством как *блокирующей синхронизации*, так и *неблокирующей*.

Блокирующая синхронизация использует специальные примитивы синхронизации (мьютекс, семафор, условные переменные), чтобы в любой момент только один поток мог исполняться в *критической секции*. Для обеспечения эксклюзивности выполнения другие потоки переводятся операционной системой в режим ядра, где они ожидают исполнения. Такой подход может привести к проблемам: дедлок (остановка и бесконечное ожидание потоков), ухудшение быстродействия из-за затрат на перевод в режим ядра.

Альтернативой служит неблокирующий подход, который не переводит потоки в режим ядра, а заставляет каждый непрерывно исполнять код. В этом случае понятие критической секции, в которой в любой момент времени может находиться лишь один поток, уходит. Точнее, теперь блокирующая секция сужается лишь до одной ассемблерной инструкции *compare-and-swap*. И только этот код может исполняться потоком эксклюзивно. Данный подход во многих случаях эффективнее по времени, так как не требует временных затрат на блокировку.

В свою очередь, неблокирующие алгоритмы могут быть поделены на три типа (Таблица 1) в зависимости от гарантий, которые они дают: [1]

Таблица 1 – Типы неблокирующих алгоритмов и их гарантии

Тип	Гарантии
Без препятствий	Любой поток при любой начальной конфигурации завершает свою работу через ограниченное количество шагов.
Без блокировок	Хотя бы один из потоков совершает прогресс в любой момент времени.
Без ожиданий	Любой поток завершается через конечное количество шагов. И это число не зависит от других потоков

Тип «без блокировок» является компромиссом между не всегда возможным типом «без ожиданий» и типом «без препятствий», который даёт слишком слабые гарантии. Именно на основе типа «без блокировок» создано большое число *lock-free структур данных*, которые позволяют производить с ними операции многопоточно.

Одной из таких lock-free структур данных является Hash Array Mapped Trie. Впервые Hash Trie была описана Брандисом и Фредкином. [2] Позже, она была модифицирована Бегвиллом [2] в Hash Trie Mapped Trie – подтип префиксных деревьев, поддерживающие операции поиска значения по ключу, удаление ключа, добавление значения по ключу. Также, операции производятся по хешированному ключу. Каждый узел префиксного дерева хранит ссылки на поддеревья внутри массива, которые индексируется с помощью *bitmap*. Это делает структуру эффективной по памяти и использованию кэша.

В 2017 году трое учёных из Швейцарии [1] предложили многопоточную lock-free реализацию *Hash Array Mapped Trie*. В данном алгоритме, они избавились от проблемы stop-the-word, которая свойственна хеш-таблицам; а также сделали структуру данных эффективной по памяти – с динамическим с глубиной.

В данный момент реализация lock-free Hash Array Mapped Trie есть на языке Java, но отсутствует для C++. Так, в библиотеке lock-free структур данных libcds [3] необходима реализация этой структуры данных на языке C++. Целью данной работы является повышение производительности

программного обеспечения за счет реализации потокобезопасной структуры данных Hash Array Mapped Trie.

Были поставлены следующие задачи:

1. Анализ существующих структур данных
2. Реализация однопоточной версии Hash Array Mapped Trie
3. Реализация блокирующей потокобезопасной версии Hash Array Mapped Trie
4. Реализация неблокирующей потокобезопасной версии Hash Array Mapped Trie
5. Решение проблемы АВА в неблокирующей реализации
6. Тестирование разработанной структуры данных
7. Внедрение в библиотеку libcds

ОБЗОР АНАЛОГОВ

Напомним, какая проблема решается при внедрении lock-free Hash Array Mapped Trie. Существует базовая задача поддержания *словаря* [5], где каждый элемент (значение) идентифицируется уникальным ключом. Под поддержанием имеется ввиду возможность производить некоторые операции. Обычно, в этот список входят:

- Вставка значения по ключу
- Удаление по ключу
- Поиск значения по ключу

Особо важно, чтобы каждая операция работала за наименьшее время, а также затраты на хранения словаря должны быть минимальны.

Данная прикладная задача решается повсеместно. Начиная от самой популярной [3] базы данных Redis и заканчивая таблицей символов в большинстве компиляторов [1]. Причём, в зависимости от специфики области применения некоторыми требованиями можно пренебречь в пользу других. Например, пожертвовать временем исполнения для сокращения затрачиваемой памяти или памятью для того, чтобы предотвратить снижения производительности.

Задача поддержания такого словаря, в свою очередь, решается посредством одной из следующих структур данных (Таблица 2).

Таблица 2 – Структуры данных и их амортизационное время операций

	Добавление	Удаление	Поиск
Хеш-Таблица	$O(1)^*$	$O(1)$	$O(1)$
Красно-Чёрное Дерево	$O(\log n)$	$O(1)$	$O(1)$
Список	$O(n)$	$O(n)$	$O(n)$

Из таблицы 2 можно сделать вывод, что хеш-таблица является наиболее предпочтительной структурой данных. Она выигрывает по всем параметрам. Но стоит отметить:

- во-первых, линейное время исполнения достигается не всегда, это лишь амортизационное время
- во-вторых, хеш-таблица время от времени необходим рефазинг, то есть время, когда её необходимо расширить
- в-третьих, нельзя удалить элемент в хеш-таблице
- в-четвёртых, хеширование является затратным по времени. Более того, оно необходимо при каждой операции с деревом.

Второе замечание ведёт к понижению производительности в случайный момент времени. Это может быть недопустимы в некоторых системах реального времени. Третье замечание показывает неэффективность хеш-таблиц по памяти (хранятся лишние элементы).

В то же время, красно чёрное дерево проигрывает хеш-таблице по операции добавление. Зато не имеет проблем с рефазингом и неэффективности по памяти.

Из всех структур данных из Таблицы 2 список является самым эффективным по памяти, но скорость работы неприемлемо низкая.

Хотело иметь структуру данных, которая является компромиссом между быстрой хеш-таблицей и эффективным по памяти красно-чёрным деревом. И если определить к структуре данных следующие требования:

- Отсутствие время на рефазинг
- Быстродействие по сравнению с аналогами
- Эффективность по памяти по сравнению с аналогами

То подходящей является Hash Array Mapped Trie. Она сочетает в себе свойства хеш-таблиц и префиксных деревьев (trie). Все операции у этой структуры данных являются выполняются за $O(1)$. Также необходимости в рефазинге – структура динамически при каждой новой операции меняет свой размер (сокращается, увеличивается), а не откладывает это как в хеш-таблице. То

есть, структура ещё эффективна по памяти. И главное, Hash Array Mapped Trie может быть реализована как lock-free структура. Это даёт прирост к производительности её базовых операций.

1 ОПИСАНИЕ HASH ARRAY MAPPED TRIE

1.1 Описание Trie

Идея внедряемого алгоритма была описана в статье Cache Aware Lock-Free Concurrent Hash Tries [1]. Перед тем как изложить соответствующие устройства lock-free Hash Array Mapped Trie, необходимо понять, как работает однопоточная версия. Это, в свою очередь, требует знаний о структуре данных Trie.

Trie – дерево поиска (оно же *префиксное дерево* или *бор*) позволяет хранить, удалять, искать ключи, а также привязанные к ним значения. Обычно, ключами выступают строки, но могут быть и числа. Чтобы добавить элемент в дерево, ключ разбивается на части. Если это строка, то разбиение зачастую идёт по символам. В случае, если это число, то на биты. Далее для каждой части создаётся репрезентирующий её узел. Например, на рисунке 1.1 показано разбиение строки «hton»

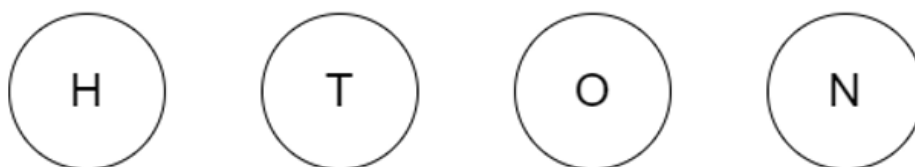


Рисунок 1.1 – Узлы из «hton»

Число 9 в качестве ключу будет предварительно переведено в двоичную систему и разбито на биты 1001 как показано на рисунке 1.2



Рисунок 1.2 – Узлы из ключа 9

После разбиения ключа, необходимо поочерёдно вставить узлы в дерево. Вставка всегда происходит из корня. Её алгоритм может быть описан так: пусть нам необходимо вставить ключ k , который разбивается на $k_1, k_2 \dots k_s \dots k_n$ узлы, мы находимся в узле дерева t_i и уже вставили все узлы до k_s .

Тогда,

- если t_i уже имеет ребёнка k_s (назовём его t_j), то мы пропускаем вставку k_s (так, как он уже есть $k_s = t_j$) и вставляем k_{s+1} начиная с t_j .
- если t_i не имеет ребёнка k_s , то мы добавляем в t_i ребёнка k_s (назовём его t_j) и вставляем k_{s+1} начиная с t_j .

В обоих случаях, если был обработан последний узел ($k_i = k_n$), то t_j помечается как терминальный. Это указывает, что данный узел является в дереве концом какого-то ключа.

Что касается значений, ассоциированных с ключом, то они могут быть привязаны к терминальным узлам. Таким образом, не каждый узел в префиксном дереве связан с определённым значением.

Как пример, префиксное дерево, включающее в себя строки «kafka», «karma», «kafkaesque», «jail» показано на рисунке 1.3. Узлы красного цвета являются терминальными, белого — промежуточными, зелёный узел соответствует корню.

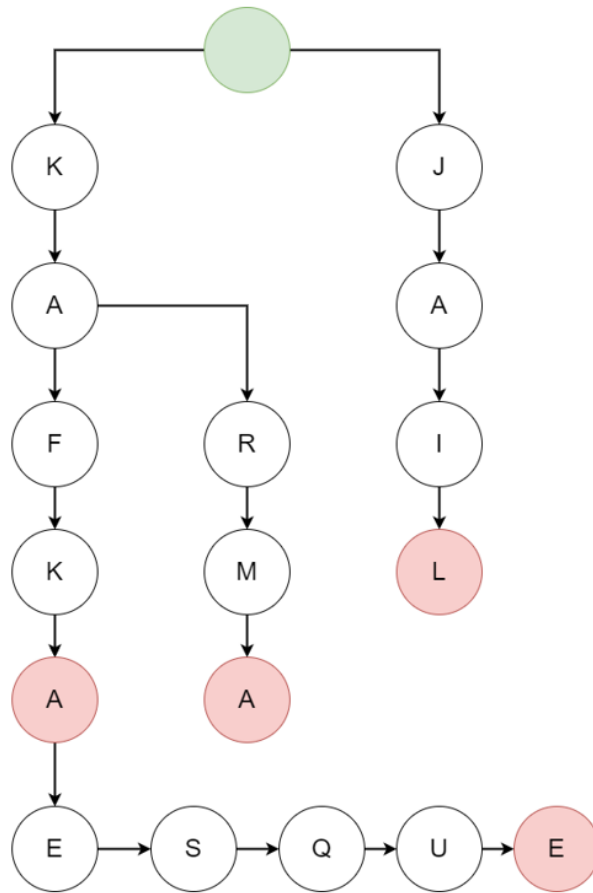


Рисунок 1.3 – Префиксное дерево, хранящее строки

Стоит заметить, trie компактно хранит ключи: узлы строк с одинаковыми префиксами не дублируются, а хранятся в единственном экземпляре («kafka» и «karma» имеют общий префикс «ка»). Это и делает эту структуру данных эффективной по памяти. То же можно сказать и про Hash Array Mapped Trie, которая использует свойства префиксных деревьев.

Trie может быть реализовано несколькими способами. Первый способ – наивный. Пусть у нас есть определённый алфавит, состоящий из N символов (обычно $N = 256$, что соответствует размерности таблицы ASCII). То есть, всего может быть N различных видов узлов в дереве и у каждого узла может быть не больше, чем N различных детей. Тогда пусть каждый узел хранит ссылки на своих детей в собственном фиксированном массиве на N элементов ($array[N]$). Если $array[i] = nil$, значит данный узел не имеет ребёнка, узла с типом i . На рисунке 1.4 изображён узел A, который имеет только трёх детей

B, C, D, но хранит информацию об отсутствии 253 других возможных детей (например, $array[2] = nil$, то есть отсутствует узел C).

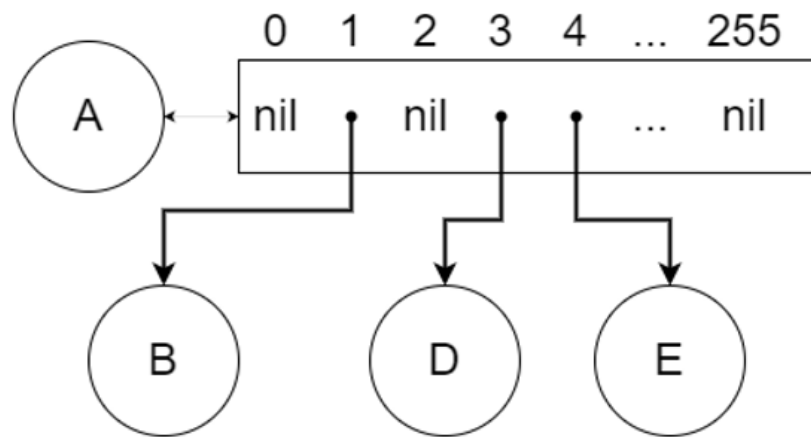


Рисунок 1.4 – Узел в наивной реализации префиксного дерева
 Данный подход имеет плюс в скорости поиска следующего узла (асимптотика – $O(1)$). Вместе с тем, огромное количество памяти расходуется впустую на хранение *nil*.

Закономерным решением проблемы с памятью было бы использование списков вместо массивов. Это увеличит время операции до $O(m)$, где m – количество детей у узла, но позволит избавиться от лишнего расхода памяти. На рисунке 1.5 показано префиксное дерево, реализующее данный подход: узел A хранит связный список (linked list) из трёх элементов B, D, E, что разы меньше, чем в наивной реализации.

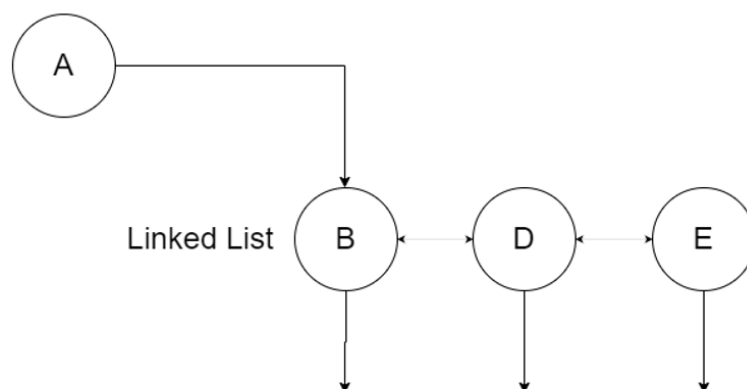


Рисунок 1.5 – Узел в реализации префиксного дерева со связным списком

Hash Array Mapped Trie снова находится на середине между двумя подходами. Для хранения детей использует массив как в наивном методе (но не статический, а динамический. И так же, как и реализации со связным списком, не хранит лишние элементы.

Основные положения Trie:

- Позволяет поддерживать словарь (операции: вставка, удаление, поиск)
- Ключ делится на части, которые репрезентируются в узлы.
- Узлы бывают трёх видов: терминальные (хранят в себе значение), корень (представляет особый узел, у которого нет родителей), промежуточные
- При вставке узлы ключа последовательно добавляются в дерево. Если два ключа имеют общий, префикс, их узлы, соответствующие этому префиксу, не дублируются.
- Каждый узел хранит детей в связном списке, что позволяет экономить память (в отличии от статического массива)

1.2 Описание однопоточной Hash Array Mapped Trie

После детального рассмотрения префиксного дерева, можно приступить к устройству Hash Array Mapped Trie. Hash Array Mapped Trie (НАМТ) представляет собой такое же дерево поиска для хранения, удаления, поиска ключей/значений. Как было сказано ранее, она сочетает в себе свойства хеш-таблиц и префиксных деревьев. Но также включает отличные идеи для повышения быстродействия.

Как и для префиксных деревьев, НАМТ разбивает свой ключ на части. Эти части определяют путь до ключа/значения. Ключи могут иметь общие префиксы, что позволяет экономить память. Но, если в префиксных деревьях разбиение определялось для каждого типа ключа по-разному (для строки – на символы, для числа – на биты), то Hash Array Mapped Trie разбивает единообразно: получает 64-битный хеш ключа, а затем делит его части по пять битов. Итого, частей $[64 / 5] = 13$. Каждые пять бит – это отдельное число.

Например, если в дерево добавляется ключ, который хешируется в число 2172 (двоичную форму будем обозначать так $0b1000011111$), то он будет разбит на (рисунок 2.1)

$$p_0 = 0b10000, p_1 = 0b11111, p_2 = 0b00000, \dots, p_{12} = 0b00000 \dots$$

Части репрезентируются в узлах 1, 31, 0, 0 и так далее.

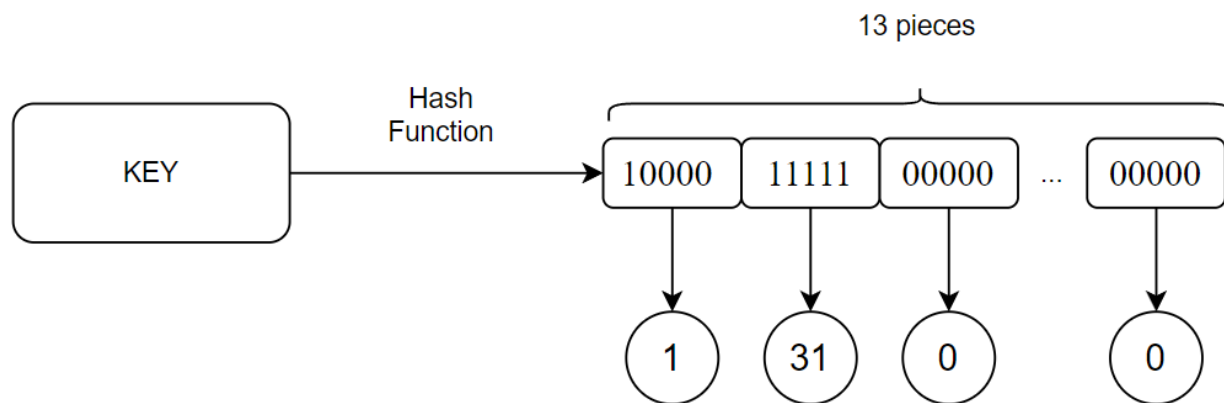


Рисунок 2.1 – Разбиение хеша ключа на части

Таким образом, данный ключ в дереве будет располагаться по пути, который показан на рисунке 2.2. Зелёным цветом обозначен корень дерева, белым – промежуточные вершины, красным – терминальная вершина, где хранится значение для ключа.



Рисунок 2.2 – Путь в дерева для ключа с хешом 2172

На самом деле, для хеша 2172 не обязательно хранить все узлы в контексте дерева. Нужно хранить ровно такое количество, чтобы можно было уникальное идентифицировать путь до ключа в дереве. Приведём несколько примеров.

Пусть у нас есть пустое дерево (рисунок 2.3)



Рисунок 2.3 – Hash Array Mapped Trie не содержащий ни одного ключа

Добавим в дерево ключ с хешом 2172 (рисунок 2.4). Как говорилось ранее, нужно добавлять минимальное количество узлов, чтобы можно было уникально идентифицировать путь до ключа. В данном случае в пустом дереве хватит лишь одного узла. Этот узел нужно сделать терминальным и положить в него значение и полный ключ (это понадобится для проверки при поиске).

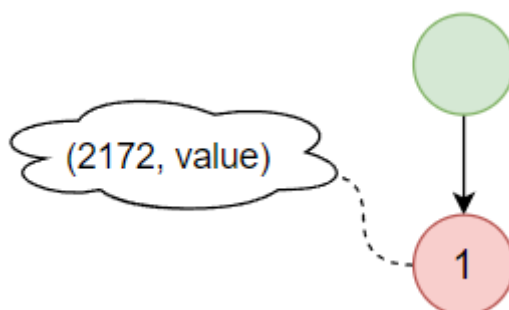


Рисунок 2.4 – Hash Array Mapped Trie с единственным ключом.

Теперь, если нам необходимо будет найти ключ 2172, мы проходить не по всем узлам (ведь дерево их не содержит), а лишь по части и ним. И когда поиск приведёт к терминальной вершине, необходимо провести проверку на полное совпадение ключей. Ниже будет описан более строгий алгоритм поиска. При добавлении ключа с хешом 65 появляется неоднозначность. Действительно, у обоих хешей первая часть совпадает. Значит, узел 1 (0b00001) уже нельзя сделать терминальным (неоднозначно какому ключу он будет принадлежать: 65 или 2172). Поэтому необходимо сделать узел 1 промежуточным, и отвести от него два узла для 65 и 2172 (рисунок 2.5). Эти узлы вычисляются уже по второй части хеша.

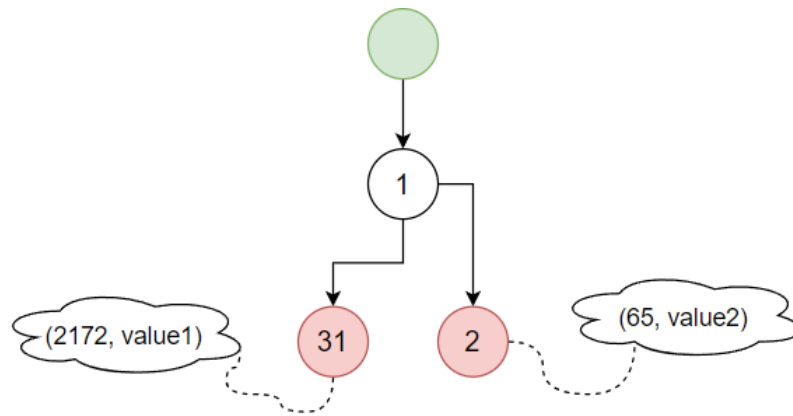


Рисунок 2.5 – Hash Array Mapped Trie с двум ключами с общим префиксом у хешей.

Может случиться так, что и вторые части хеша будут совпадать. Тогда производим те же самые операции (создаём общий узел и отводим узлы для детей). В случае, если все части хешей совпадают, то на последнем уровне вместо терминальной вершины создаём *терминальный узел-список* из двух ключей. Терминальный узел-список, это лист, который хранит себе множество пар ключ/значение. На рисунке 2.6 показан процесс разрешения коллизии, при добавлении двух ключей с идентичными первыми 12 частями хеша.

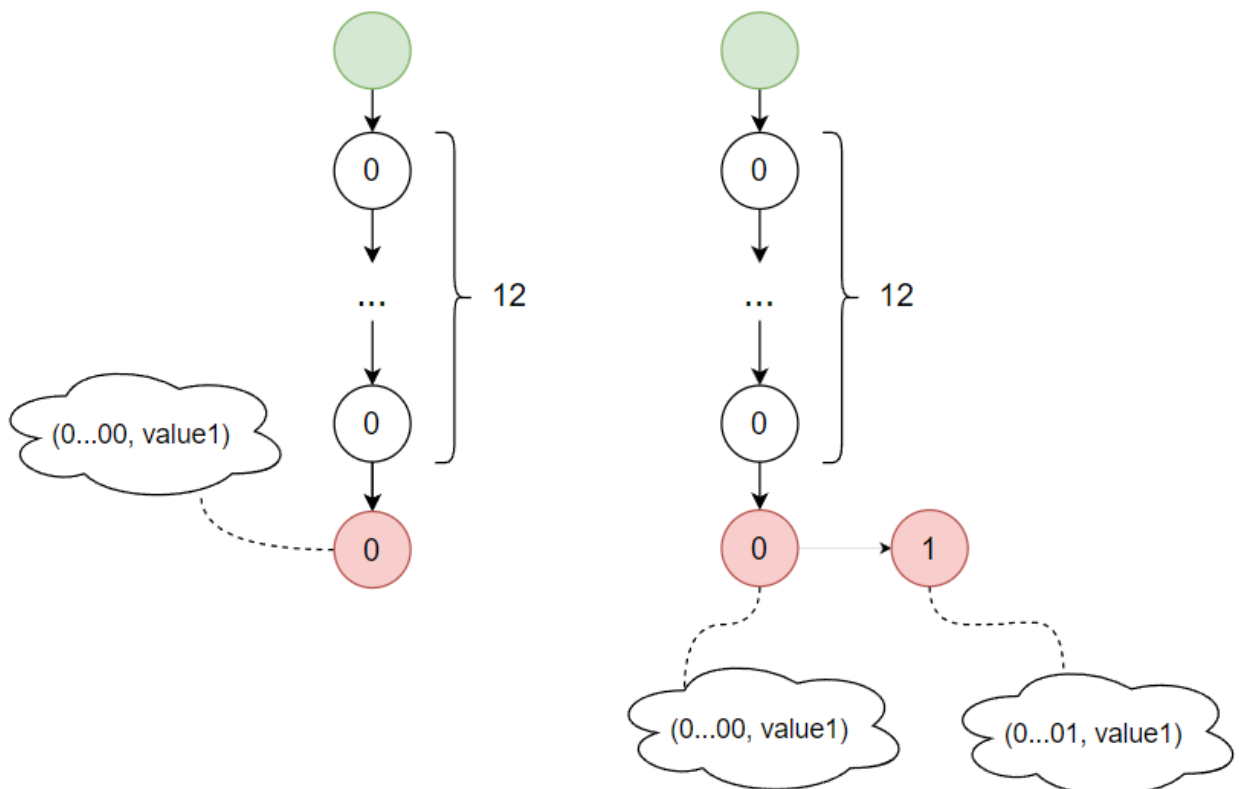


Рисунок 2.6 – Разрешение коллизии

Теперь осуществим удаление. Оно будет производиться рекурсивно, начиная с терминальной вершины, соответствующей ключу: если текущая вершина не имеет детей, то она подлежит удалению. То есть, НАМТ сжимает дерево сразу же после удаления, поэтому не хранит лишних узлов.

Более строго операции вставки, поиска, удаления можно описать следующим образом.

Вставка: пусть нам необходимо вставить пару (k, v) , где $hash(k) = h_1, h_2 \dots h_s \dots h_{12}$, мы находимся в узле дерева n_i и уже вставили все узлы до h_s . Тогда:

- если у текущего узла дерева n_i нет ребёнка h_s , мы добавляем его, помечаем терминальным и вкладываем (k, v)
- если у текущего узла есть нетерминальный ребёнок n_j , то пропускаем вставку h_s и производим добавление следующей части хеша h_{s+1} , но уже с узла n_j
- если $s < 11$ (то есть, мы находимся ниже предпоследнего уровня глубины дерева) и у текущего узла есть терминальный ребёнок с ключом и значением (k_1, v_1) , то сначала проверяем не совпадает ли он с $k = k_1$. Если да, то заменяем v_1 внутри этого ребёнка на v . Иначе, делаем из него промежуточный узел и добавляем (k, v) , и (k_1, v_1)
- если $s = 11$ (то есть, мы находимся на предпоследнем уровне глубины дерева) и у текущего узла есть терминальный ребёнок с ключом и значением (k_1, v_1) , то мы заменяем его на терминальный узел-список, состоящий из (k_1, v_1) , (k, v) .
- если $s = 11$ (то есть, мы находимся на предпоследнем уровне глубины дерева) и у текущего узла есть терминальный узел-список ребёнка, то проверяем есть ли в этом списке пара (k_1, v_1) , где $k = k_1$. Если имеется, то заменяем v_1 на v . Иначе, добавляем в список новую (k_1, v_1)

Поиск: пусть нам необходимо найти ключ k , где $hash(k) = h_1, h_2 \dots h_s \dots h_n$, мы находимся в узле дерева n_i и уже спустились на уровень h_{s-1} . Тогда:

- если у текущего узла дерева n_i нет ребёнка h_s , то ключ/значение не найдены.
- если у текущего узла есть нетерминальный ребёнок n_j , то производим поиск следующей части хеша h_{s+1} , но уже с узла n_j .
- если у текущего узла есть терминальный ребёнок с ключом и значением, то проверяем есть ли в этом списке пара (k_1, v_1) , где $k = k_1$. Если да, то (k_1, v_1) и есть искомая пара. Иначе, ключ/значения не найдены.

Удаление: пусть нам необходимо удалить ключ k . Произведём поиск этого ключа. Если ключ отсутствует, то завершаем удаление. Иначе, существует терминальная вершина n_j соответствующая ему. Начнём удаление с вершины n_j . Тогда:

- если текущая промежуточная вершина n_i не имеет детей, то удаляем её (удаляем ссылку на неё у родителя)
- если текущая терминальная вершина n_i не имеет детей, то удаляем её, если она хранит единственный ключ. Если это узел-список, то удаляем из неё такую пару (k_1, v_1) , что $k = k_1$.
- иначе, прекращаем процесс.

Следующий важный момент в Hash Array Mapped Trie – это использование связки битмапа и динамический массив. Идея заключается в том, что узел хранит ссылки только на существующих детей в динамическом массиве. Поиск по такому массиву занимает $O(n)$, как это было в одной из реализаций trie ранее. Но НАМТ также хранит битмапу, по которой можно равен определить индекс необходимого элемента в массиве. Индекс вычисляется с использованием побитовых операций, что значительно быстрее предыдущей версии.

Структура данных битмапа (bitmap) – это число с определённой разрядностью N , где каждый бит говорит об присутствии (если равен единице) или отсутствии (если равен нулю) чего-либо. Битмапа идёт всегда в связке с другим объектом и описывает его состояние. В случае Hash Array Mapped Trie битмапа есть у каждого узла, её разрядность 32, а характеризует она наличие или отсутствие детей определённого. Например, битмапа 104 (или в двоичном виде $0b1101000$) у узла P (рисунок 2.7) говорит о наличии трёх детей: 0, 1, 3.

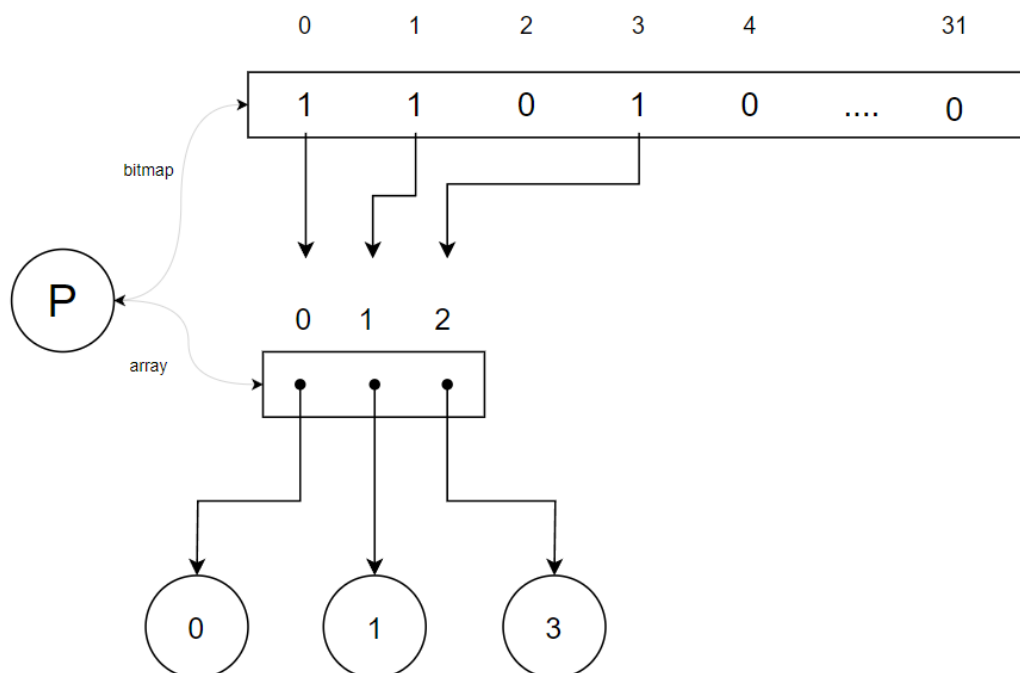


Рисунок 2.7 – Узел с битмапой и массивом(array), содержащий трёх детей.

Зная индекс битмапы, где поднят флаг единицы, можно вычислить индекс в массиве.

Формула искомого индекса вычисляется по формуле (1):

$$array_index = popcount(((1 \ll pos) - 1) \& bitmap), \quad (1)$$

где *popcount* – функция подсчёта поднятых битов в числе, *pos* – позиция в битмапе с установленным битом, $\&$ - операция побитового «И», $1 \ll pos$ – побитовый сдвиг числа 1 *pos* раз влево. Получения бита в битмапе производится тоже через побитовые операции: $((bitmap \gg pos) \& 1)$. Итого, чтобы получить элемент массива по битмапе, необходимо сделать два шага:

1. Проверить, установлен ли бит на определённой позиции (используются только побитовые операции)
 2. Если в предыдущем шаге бит не установлен, вернуть ничего. Иначе, вычисляем индекс массива (используются только побитовые операции)
- Благодаря битмапе поиск ребёнка в узле происходит не за время $O(n)$, а за $O(1)$.

Основные положения о Hash Array Mapped Trie:

- Ключ хешируется и разбивается на части.
- У каждого узла может быть не больше, чем 32 ребёнка
- Максимальная высота дерева равна 13
- При добавлении ключей с одинаковыми префиксами хешей, общая часть не дублируется.
- Добавляется минимальное количество узлов, при котором можно уникально идентифицировать ключ
- При удалении, дерево сжимается и не хранит лишних узлов.
- Каждый узел хранит детей в динамическом массиве и битмапе. Через побитовые операции можно за время $O(1)$ вычисляет ребёнка узла или его отсутствие.

1.3 Описание lock-free Hash Array Mapped Trie

Идея алгоритма [1] построена на введение 3 типов узлов (рисунок 3.1):

1. SNode – узел, который содержит список пар ключ/значение
2. CNode – узел с битмапой и массивом детей. Дети могут быть двух типов: SNode и INode
3. INode – узел, который хранит в себе ссылку на CNode. Это искусственная обёртка над CNode для поддержания lock-free.

Все три типа унаследованы от абстрактного узла Node.

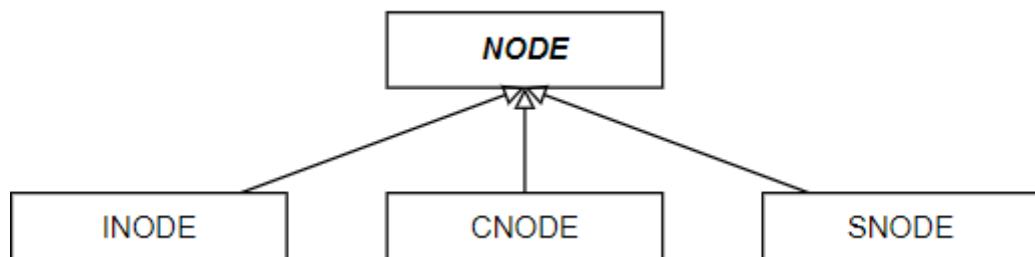


Рисунок 3.1 – UML-диаграмма для узлов

Начнём объяснения используя SNode и CNode. В дальнейших рассуждениях будем усложнять структуру данных. Пусть у нас есть дерево с узлом SNode (рисунок 3.2). Надпись на направленном ребре обозначает тип ребёнка.

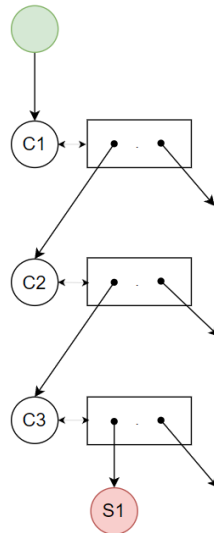


Рисунок 3.2 – Hash Array Mapped Trie с узлом S1

Тогда, если мы хотим вставить в C3 ещё одного ребёнка, нам необходимо будет совершить два действия:

1. Обновить битмапу, чтобы она указывала и на новый элемент
2. Вставить новый узел в массив C3

Операции 1 и 2 происходят неатомарно и это приводит к логической гонке данных. Например, если два потока T1 и T2 одновременно пытаются добавить соответственно новые узлы S2 и S3, то может произойти следующее:

1. T1 копирует битмапу и меняет её локально.
2. T2 копирует битмапу (которая указывает только на S1) меняет её локально.
3. T1 меняет битмапу в C3 на изменённую локальную версию
4. T2 перезаписывает битмапу C3 на свою изменённую локальную версию.

В результате, битмапа будет содержать версию потока T2, который не учитывает изменения T1. Решение этой проблемы заключается в том, что обновляют не битмапу и массив, а целые узлы. Так, в данном случае T1 не меняет содержимое C3, а копирует этот узел C3', меняет его локально, и уже потом обновляет в C2 указатель на C3 на C3'.

Для того, чтобы избежать техническую гонку данных при изменении указателя, необходимо воспользоваться операцией *compare-and-swap* (cas).

Это функция принимает три аргумента:

1. указатель на старый узел (назовём это *p*)
2. значение, которое лежало по этому указателю (назовём это *old*)
3. обновлённый узел (назовём это *new*)

Если $*p = old$, то мы меняем содержимое указателя на *new* и возвращаем *true*, как показатель того, что значение было обновлено. Иначе возвращаем *false*.

В нашем случае, поток T1 сначала зафиксирует локально чему равно C3. Потом сделает его копию C3', начнёт менять (добавит новый узел). И воспользуется cas, куда передаст указатель на C3 в качестве *p*, зафиксированный ранее C3 в качестве *old* и C3' в качестве *new*. Если cas прошёл успешно, то есть никакой другой поток не поменял C3, то наш поток

обновил его. Иначе, производим вставку заново с самого начала. Операция `cas` происходит за одну ассемблерную инструкцию, что позволяет считать её `lock-free`, то есть неблокирующей.

Даже при таком решении с атомарны `compare-and-swap` на целом узлом, отстаются проблемы с гонками данных. Пусть есть некоторое дерево в следующей конфигурации (рисунок 3.3) и пусть есть поток `T1`, который хочет вставить в `C3` новый узел `S2`. Для этого он копирует ноду `C3` (жёлтым цветом обозначена копия).

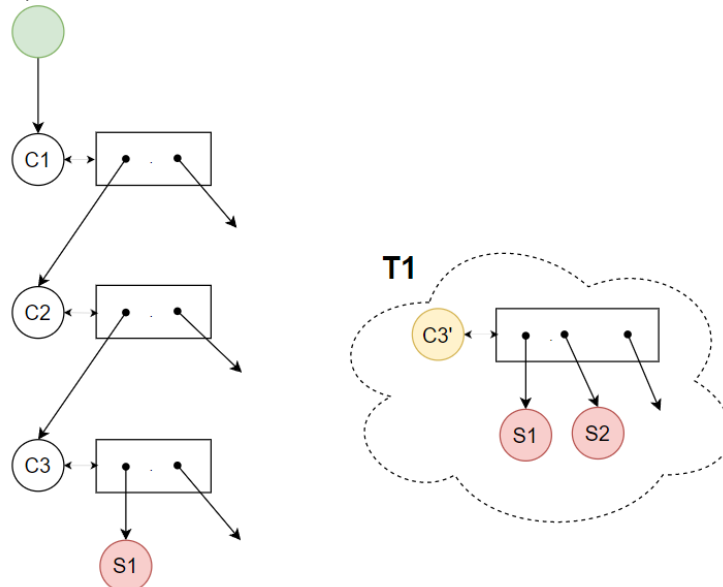


Рисунок 3.3 – поток `T1` копирует `C3` и добавляет в копию новый узел `S2`

Пока `T1` занимается копирование и локальным добавлением нового узла, другой поток `T2` решает добавить новый узел `S3` в `C2`. Для этого он также создаёт копию `C2'`, добавляет узел `S3` (рисунок 3.4)

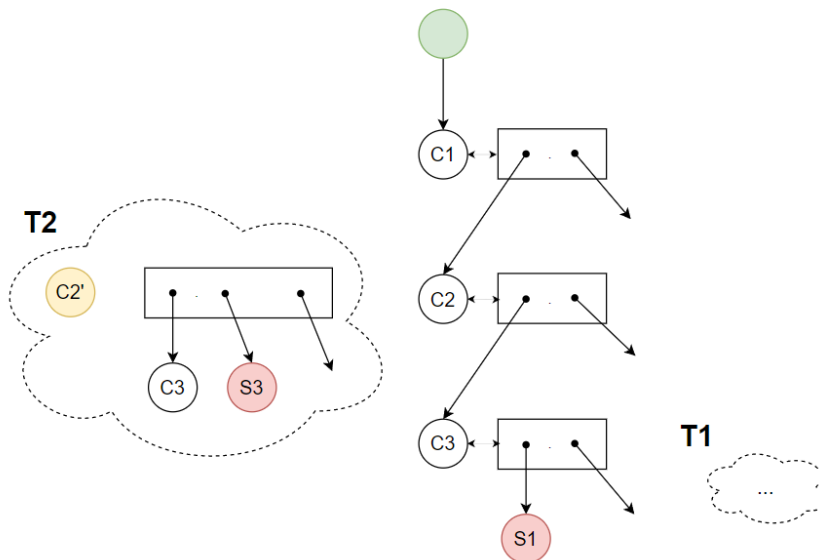


Рисунок 3.4 – поток `T1` копирует `C3` и добавляет в копию новый узел `S3`

И если сначала `T1` сделает `cas`-операцию, а потом `T2`, то узел `S2` потеряется (рисунок 3.5).

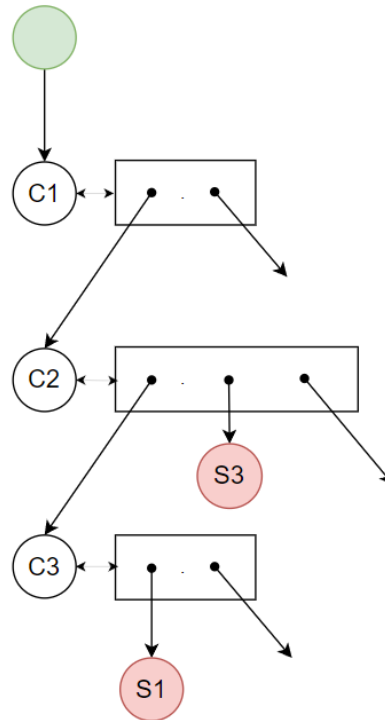


Рисунок 3.5 – дерево после обновление C3 потоком T1 и C2 потоком T1. Для решения данной проблемы, вводится новый тип узла INode. Он является обёрткой над каждым CNode. То есть, для каждого CNode узла существует родитель INode, который ссылается только на него. Тогда CNode в качестве детей может иметь INode или SNode. узлы. Теперь, любая cas-операция производится над содержимым промежуточного узла INode, что позволяет избавиться от проблемы, описанной ранее.

На данном этапе, lock-free Hash Array Mapped Trie работает корректно для операций вставка и поиск. Но есть ещё операция удаления. Она происходит следующим образом: удаляется пара с искомым ключом из соответствующего узла. Если в этом узле SNode не осталось ключей, то необходимо удалить его и произвести сокращение дерева. Сокращение дерева случается, когда существует INode, ребёнок которого имеет одного ребёнка. Тогда этот ребёнок должен заменить INode. Во время этого процесса может возникнуть проблема. Представим дерево, которое, должно сократиться (рисунок 3.6) потоком T1. Пока T1 копирует и изменяет C1(меняет ребёнка I2 на S2), другой поток T2 добавляет в C2 новое значение S3. Для этого также начнёт копировать C2 и обновлять копию. В этот момент, T1 совершает

сокращение (I2 заменяется на S2). Затем поток T1 совершит cas и изменит ребёнка I2. Но I2 уже удалён. Поэтому, в дереве не будет вершины S3.

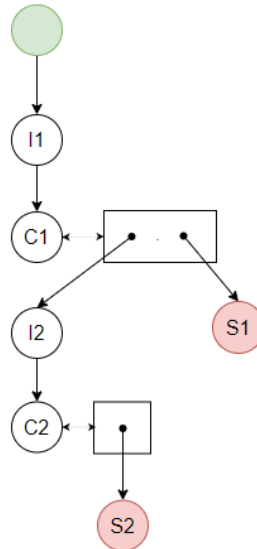


Рисунок 3.6 – дерево, перед тем, как сократиться

Решение этой проблемы – это помечать сокращаемые узлы CNode как гробовые (tomb-node). Такие узлы все потоки могут только сокращать. Если CNode уже нельзя сократить, то необходимо его воскресить (из гробового узла сделать обычный).

Основные положения lock-free Hash Array Mapped Trie, описанные в этой главе:

- вводятся три типа узлов INode, CNode, SNode
- при помощи compare-and-swap атомарно происходит изменение целого узла, а не битмапы и массива отдельно
- после удаления узла, дерево начинает сжиматься
- CNode, который может быть сокращён помечается как гробовой. В него нельзя добавлять или удалять детей.

2 РАЗРАБОТКА

В предыдущей главе описаны теоретические аспекты алгоритма lock-free Hash Array Mapped Trie и также заложены предпосылки для внедрения. Внедрение такого алгоритма — процесс трудозатратный. Он включает в себя множество пунктов: постоянная отладка программы, ручное и автоматическое тестирование, анализ кода, проверка гипотез, исправление ошибок и так далее. Чтобы облегчить эти процессы, необходимо иметь чёткий план действий: что реализовывать в первую очередь, а что можно отложить на потом.

План:

- Настройка рабочего окружения
- Реализация однопоточной версии НАМТ
- Реализация блокирующей многопоточной версии
- Переход к неблокирующей многопоточной версии НАМТ
- Решение проблемы АВА
- Внедрение в библиотеку libcds

2.1 Настройка рабочего окружения

В качестве контроля версий был выбран Git. Он является свободно распространяемым, наиболее популярным и имеет необходимый функционал:

- создание веток разработки
- слияние одной ветки в другую
- просмотр истории файла
- создание именованных коммитов
- удобное взаимодействие через командную строку
- множество интеграций с другими сервисами

Для того, чтобы код не хранился в одном месте, а также была возможность поделиться с ним, использовался GitHub. Это бесплатный

сервис, который работает с системой версией Git и позволяет хранить, анализировать код, производить Git команды, совместно редактировать код и тестировать его благодаря GitHub Actions.

Благодаря GitHub Actions можно настроить систему так, чтобы при каждом коммите тестировался код.

В основной репозитории был положен файл Dockerfile, который запускает докер контейнер при коммите. В него устанавливается операционная система, копируется код из текущего.

Благодаря GitHub Actions можно настроить систему так, чтобы при каждом коммите тестировался код. В основной репозитории был положен файл Dockerfile (рисунок 1.1), который запускает докер контейнер при коммите. В него устанавливается операционная система, копируется код из текущего репозитория GitHub и запускается скрипт entrypoints.sh

```
1  >> FROM ubuntu:18.04
2
3  RUN apt-get -y update
4  RUN apt-get -y install software-properties-common git wget clang-10
5  RUN wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null | apt-key add -
6  RUN apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main'
7
8  RUN apt-get -y install build-essential cmake qtbase5-dev
9
10 COPY entrypoint.sh /entrypoint.sh
11
12 RUN echo "$GITHUB_WORKSPACE"
13 COPY ./ /hamt
14 RUN chmod +x entrypoint.sh
15
16 ENTRYPOINT ["/entrypoint.sh"]
```

Рисунок 1.1 – Код файла Docker

В скрипте entrypoints.sh (рисунок 1.2) производится подготовка для сборки и запуска исполняемого файла для тестирования.

```

1  ►  #!/usr/bin/env bash
2
3  cd "$GITHUB_WORKSPACE" || exit 1
4  cd test
5
6  cmake -DCMAKE_C_COMPILER=clang-10 -DCMAKE_CXX_COMPILER=clang++-10 .
7  make
8
9  ./run_tests
10

```

Рисунок 1.2 – Код файла entrypoints.sh

Для сборки был написан CmakeFileLists.txt. Этот файл используется утилитой CMake, которая генерирует Makefile. А Makefile, в свою очередь, используется утилитой Make, которая создаёт на его основе исполняемый файл для тестирования.

```

1  cmake_minimum_required(VERSION 3.16)
2  project(hash_array_mapped_trie)
3
4  set(CMAKE_CXX_STANDARD 14)
5
6  set(CMAKE_CXX_FLAGS -pthread)
7
8  # MAIN TARGET
9  add_executable(
10     hamt include/visualize.h main.cpp src/utils.cpp
11 )

```

Рисунок 1.3 – Код файла CmakeFileLists.txt

Cmake для сборки был выбран так как прост в использовании и описания действий выражаются проще и короче, чем в утилите Make. В качестве интегрированной среды разработки был выбран Clion от компании JetBrains. Помимо встроенного отладчика, статических анализаторов кода, Clion предоставляет удобный графический интерфейс к Git, а также интегрируется с GitHub. Одна из особенностей lock-free алгоритмов, что ошибка на конкретных тестах может появляться не постоянно. Поэтому было необходимо тестировать код как можно чаще, чтобы была больше вероятность воспроизвести потенциальные ошибки в коде. GitHub action позволяет запускать тесты, не только при каждом коммите, но и по расписанию. Так, в

специальный файл, отвечающий за запуск докер контейнера, было добавлена следующая строка:

```
schedule:  
- cron: '* / 5 * * * *'
```

В поле `cron` в синтаксисе подобном `crontab` находится указание запускать докер контейнер с тестами по расписанию «каждые пять минут». В случае, если тесты проходят неуспешно, на почту отсылается соответствующее уведомление. Автоматическое тестирование при коммитах и по расписанию облегчает внесения изменений в код. Также это ускоряет разработку, так как избавляет от необходимости ручного тестирования и ожидания результатов.

Ещё одна проблема разработке — отладка программы. Интегрированная среда разработки Clion (в ней писался код) предоставляет возможности, но таких функций недостаточно. Зачастую необходима визуализация самой структуры данных Hash Array Mapped Trie. Для этого был добавлен код для визуализации (`visualize.h`), в котором была реализована логика сбора информации о переданном НАМТ, её обработка, а затем передача библиотеке `boost graphviz`. Данная библиотека умеет визуализировать графы в формате `png`. На рисунке 1.4 показан пример визуализации произвольного Hash Array Mapped Trie. Узел `I0` является `INode` и имеет `CNode` ребёнка `C1`. В листьях дерева содержатся `SNode`. Так узел `C6` хранит ссылку на ребёнка `SNode` с ключом `8` и значением `32`.

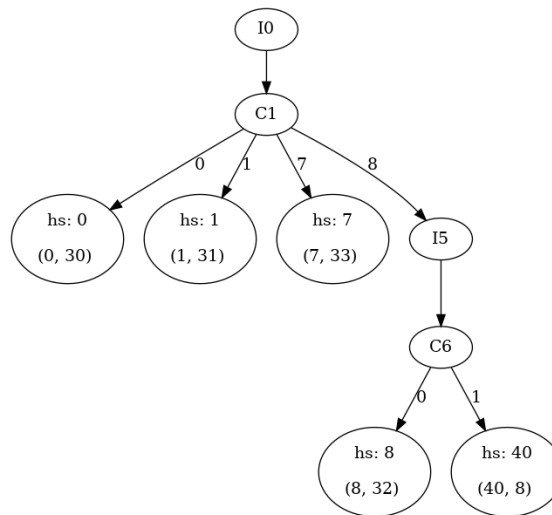


Рисунок 1.4 – Дерево, построенное при помощи visualize.h

2.2 Реализация однопоточной версии

Приступая к описанию однопоточной версии НАМТ на языке C++, необходимо сказать, что с самого код писался с учётом будущей трансформации под lock-free версию НАМТ. Это было сделано, чтобы для ускорения разработки и для логического структурирования всей работы.

Начнём с перечисления сущностей (классов, структур), опишем их взаимодействия друг с другом. А затем перейдём к трём основным операциям insert, remove, lookup.

Есть класс Hamt, представляющий контейнер с Hash Array Mapped Trie структурой (рисунок 2.1). В нём есть поле root, в котором хранится корневой узле и три публичных метода: вставка, удаление и поиск. K и V означают типы один из типов int или string.

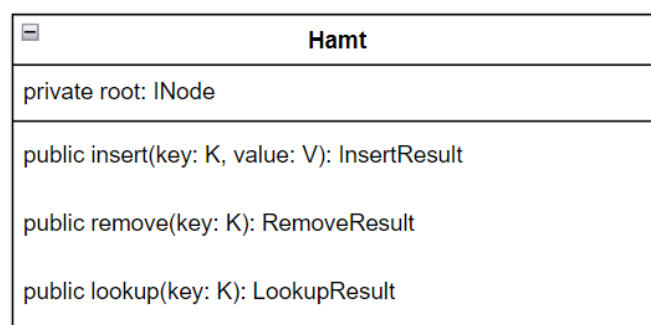


Рисунок 2.1 – UML-диаграмма для Hamt класса

Так, при создании контейнера, необходимо указать какого типа будут ключи и значения (рисунок 2.2)

```
Hamt<int, int> hamt;
```

Рисунок 2.2 – Создание контейнера с ключами и значениями типа int

Все три метода insert, remove, lookup возвращают соответственно структуры InsertResult, RemoveResult, LookupResult. Это необходимо для того, чтобы хранить информацию о том было ли значение заменено при вставке, удалено ли ключ. Например, на рисунке 2.3 показана UML-диаграмма для структуры RemoveResult. Тип Status – тип перечисления. Он может хранить в себе три значения:

- Removed (если удаление прошло успешно, тогда в поле value будет храниться значение у удаляемой пары)
- Failed (заготовка для lock-free версии, где удаление может не получиться)
- NotFound (в дереве нет такого ключа, которого хотя удалить)

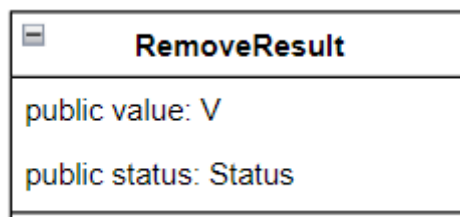


Рисунок 2.3 – UML-диаграмма для узлов

Внутри дерева есть вида узлов, наследующиеся от базового абстрактного типа Node (рисунок 2.4).

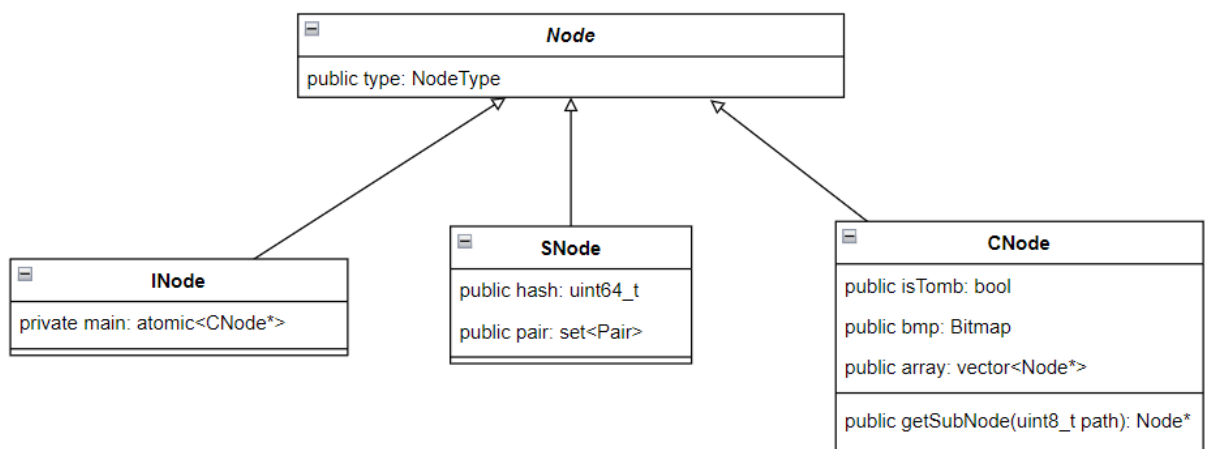


Рисунок 2.4 – UML-диаграмма для узлов

INode, как говорилось в прошлой главе, является обёрткой на CNode. Это позволяет обеспечить потокобезопасное обновление узлов. Именно поэтому у него есть одно поле `main` типа `atomic < CNode * >`. С таким типом можно совершать атомарные операции над указателем на CNode (том числе и `cas` операцию). Сейчас это лишь заготовка для будущей lock-free версии, и она не противоречит однопоточной.

SNode – узел состоящий из двух полей: хеш и множество из пар ключ/значение с таким хешом. Поле `hash` имеет тип `uint64_t`, что занимает 64 бита. Ровно столько, чтобы вместить в себя максимальное значение хеша. Это позволяет не хранить лишние биты и не тратить память. Pair представляет собой класс с двумя публичными полями (один для ключа другой для значения) и перегруженным оператором меньше (это необходимо для создания `set` на его основе).

CNode – хранит в себе поле `isTomb` (заготовка для lock-free версии, указывает гробовой это узел или обычный), поле типа `Bitmap` битмапа у данного узла, и динамический массив указателей на узлы типа `Node`. Один из важнейших методов – `getSubNode`, который принимает тип ребёнка (число от 0 до 31) и возвращает указатель на ребёнка или `nil`, если такого ребёнка нет.

Подробнее стоит остановиться на вспомогательной структуре данных `Bitmap` и методов, работающих с ней (рисунок 2.5). `Bitmap` характеризует состояние массива `array`. То есть, указывает на отсутствие или наличие определённых детей. При удалении или добавлении новых узлов в массив, поле типа `Bitmap` у `CNode` должно изменяться. Для этого у `Bitmap` есть публичные методы `set`, `unset` и `isSet`.

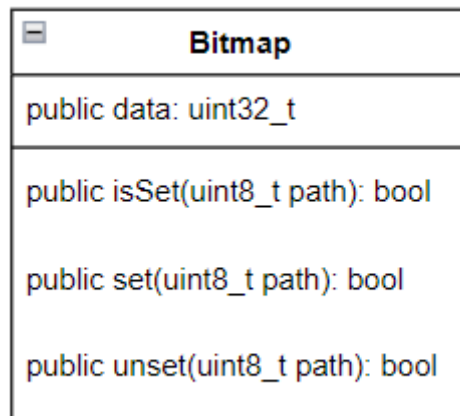


Рисунок 2.5 – UML-диаграмма для класса Bitmap

Все три метода (рисунок 2.6) используют побитовые операции «И» (&), сдвига вправо (>>), инвертирование (~).

```

bool Bitmap::isSet(uint8_t pos) const {
    return ((data >> pos) & 0b1) == 0b1;
}

void Bitmap::set(uint8_t pos) {
    data = data | (1 << pos);
}

void Bitmap::unset(uint8_t pos) {
    data = data & ~(1 << pos);
}
  
```

Рисунок 2.6 – Реализация методов Bitmap

На основе данных битмапы также вычисляет индекс массива (рисунок 2.7). Эта логика реализована в отдельном методе класса CNode.

```

uint8_t getArrayIndexByBmp(uint8_t pos) const {
    return __builtin_popcount(
        ((1 << pos) - 1) & bmp.data
    );
}
  
```

Рисунок 2.7 – Вычисление индекса массива для установленного бита

Команда `__builtin_popcount` вычисляет количество установленных битов в числе, `bmp.data` – значение битмапы в соответствующем узле CNode.

Теперь, когда были описаны основные классы и структуры, можно перейти к реализации `insert`, `remove`, `lookup` операций. Они определены в

публичных методах класса `Namt` и каждый ссылается на соответствующий непубличный метод. Так было сделано по двум причинам:

- Декомпозиция метода (в публичном создаются вспомогательные структуры, которые идут в приватные)
- Необходимость повторного вызова вставки, удаления и поиска (это заготовка для lock-free версии)
- Такая структура легко трансформируется в блокирующую многопоточную версию.

Для примера разберём операцию `insert` (рисунок 2.8). На вход в публичную функцию поступают ключ и значение. В строчка 282-284 идёт проверка особого случая (проверка на пустоту дерева). В строчке 285 создаётся новый узел `SNode`, который нужно будет вставить. Далее, вызывается приватный метод `insert` (рисунок 2.9).

```
281 bool insert(K key, V value) {
282     if (root == nullptr) {
283         root = new INode<K, V>(new CNode<K, V>());
284     }
285     SNode<K, V> *subNode = createSNode( &key, value, generateSimpleHash(key));
286     return insert(root, subNode, 0);
287 }
```

Рисунок 2.8 – Вычисление индекса массива для установленного бита

В приватном методе, чтобы определить по какому пути идти, вычисляет часть хеша (строчка 347). Далее вычисляет ребёнка, на которого мы, возможно перейдём на следующем шаге. И затем обрабатывается одна из следующих ситуаций, которая была описана в предыдущей главе.

```
346 static bool insert(INode<K, V> *startNode, SNode<K, V> *newNode, uint8_t level) {
347     int path = extractHashPartByLevel( hash: newNode->getHash(), level);
348     Node *subNode = startNode->main->getSubNode(path);
349
350     if (subNode == nullptr) {
351         startNode->swapToCopyWithInsertedChild( child: newNode, path);
352         return true;
353     } else {
354         if (subNode->type == S_NODE) {
355             if (level == 12 || static_cast<SNode<K, V> *>(subNode->contains(newNode)) {
356                 startNode->swapToCopyWithMergedChild( newChild: newNode, path);
357                 return true;
358             } else {
359                 startNode->swapToCopyWithDownChild( child: subNode, level);
360             }
361         }
362         return insert(static_cast<INode<K, V> *>(startNode->main->getSubNode(path)), newNode, level + 1);
363     }
364 }
```

Рисунок 2.9 – Вычисление индекса массива для установленного бита

Так, например, если по данному пути нет ребёнка, то есть *subNode == nullptr* (строка 350), то для текущего узла вызывается специальный метод *swapToCopyWithInsertedChild*, который вставляет ребёнка по пути *path* в текущий узел (строка 351).

Методы *swapToCopyWithMergedChild*, *swapToCopyWithDownChild*, *swapToCopyWithInsertedChild* занимаются обработкой одной из возможных ситуаций. Все их перечисленных изменяют состояние текущего узла.

2.3 Реализация блокирующей многопоточной версии

Код, описанный в предыдущей главе, легко трансформируется в многопоточную блокирующую версию. Всё что, необходимо поменять – это добавить примитив синхронизации в класс *Hamt* (рисунок 3.1) и определить, в каких местах он будет использоваться.

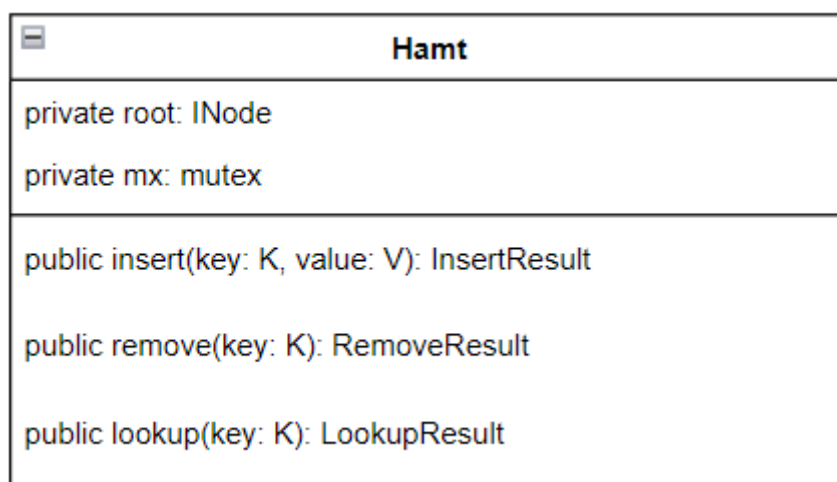
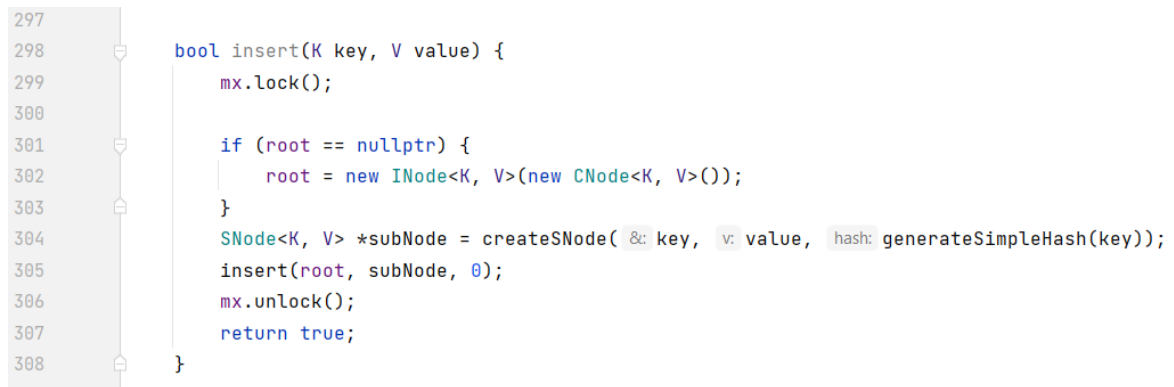


Рисунок 3.1 – UML-диаграмма класса *Hamt* для блокирующе многопоточной версии

В качестве примитива синхронизации был выбран мьютекс. Он позволяет создавать критические секции. Чтобы попытаться захватить мьютекс, потоку необходимо воспользоваться методов *lock()*. Если это удаётся сделать, то он единолично входит в критическую секцию. Иначе, ожидает. После завершения работы в критической секции потоку необходимо отпустить примитив при помощи метода *unlock()*.

Как говорилось ранее, методы вставки, удаления, поиска логически разделены на публичные и приватные. Такая структура позволяет легко добавить использование мьютекса. Идея в том, что потокобезопасный код внутри приватных методов необходимо окружить взятием и отпусканьем мьютекса (рисунок 2.3.2).



```
297
298 bool insert(K key, V value) {
299     mx.lock();
300
301     if (root == nullptr) {
302         root = new INode<K, V>(new CNode<K, V>());
303     }
304     SNode<K, V> *subNode = createSNode(&key, value, generateSimpleHash(key));
305     insert(root, subNode, 0);
306     mx.unlock();
307     return true;
308 }
```

Рисунок 2.3.2 – Реализация публичного метода вставки в блокирующей версии

2.4 Реализация неблокирующей версии

Lock-free версия НАМТ является закономерным продолжением блокирующей многопоточной версии, которая, в свою очередь, была трансформирована из однопоточной.

Поэтому при реализации lock-free версии необходимо было постепенно сужать критические секции в блокирующей версии и вводить cas-операции.

Главным отличием lock-free является то, что узлы дерева изменяются только при cas операции. Поток всегда сначала копирует узел, который он хочет изменить, меняет эту локальную копию и уже потом пытается вставить свою версию. Если вставка своей версии (операция cas) проходит неуспешно, значит другой поток уже заменил соответствующий узел и наши изменения неактуальны. Поэтому необходимо повторить операцию.

Код для вставки проходит через следующие этапы (рисунки 4.1, 4.2):

1. Запускается публичный метод, в бесконечном цикле происходит попытка вставить новый узел. Либо вставка происходит в корне, либо вызывается приватный метод insert для вставки ниже корня.

2. В приватном методе фиксируется текущая CNode и родитель, если он есть. (строки 398-399)
3. Попытка сократить дерево (строки 441-443). Метод *contractParent* производит сокращение текущего узла, если в этом есть необходимость. Возвращаемое значение сигнализирует о том, необходимо ли заново провести операцию вставки или нет.
4. Получение копии текущего CNode (строка 445)
5. В зависимости от ребёнка, произвести изменения локальной копии (строки 450-475)
6. Попытаться вставить при помощи новую версию CNode при помощи cas-операции (строки 453, 463, 467, 453)
7. Если не получилось, то повторяем вставку через вызов публичного метода (строка 474)

```

353 InsertResult insert(K key, V value) {
354     while (true) {
355         CNode *old = root->main.load();
356         if (old == nullptr) {
357             auto *c = new CNode();
358             auto *s = new SNode(key, value);
359             c->insertChild(s, extractHashPartByLevel( hash: s->getHash(), level: 0));
360             if (root->main.compare_exchange_strong(old, c)) {
361                 return InsertResult{.status = InsertResult::Status::Inserted};
362             }
363         } else {
364             InsertResult res = insert(root, nullptr, new SNode(key, value), 0);
365             if (res != INSERT_RESTART) {
366                 return res;
367             }
368         }
369     }
370 }
371

```

Рисунок 4.1 – Реализация приватного метода вставки в lock-free версии

```

437 InsertResult insert(INode *currentNode, INode *parent, SNode *newNode, uint8_t level) {
438     CNode *pm = parent ? parent->main.load() : nullptr;
439     CNode *m = currentNode->main.load();
440
441     if (contractParent(parent, i: currentNode, pm, m, level, hash: newNode->getHash())) {
442         return INSERT_RESTART;
443     }
444
445     CNode *updated = getCopy( node: m);
446     uint8_t path = extractHashPartByLevel( hash: newNode->getHash(), level);
447
448
449     Node *subNode = updated->getSubNode(path);
450     if (subNode == nullptr) {
451         transformToWithInsertedChild(updated, child: newNode, path);
452         updated->isTomb = isTombed( c: updated, root, parent: currentNode);
453         return currentNode->main.compare_exchange_strong(m, updated) ? INSERT_SUCCESSFUL : INSERT_RESTART;
454     } else if (subNode->type == SNODE) {
455         auto *s = static_cast<SNode *>(subNode);
456         if (s->contains(newNode)) {
457             transformToWithReplacedPair(updated, subNode: s, newNode, path);
458             updated->isTomb = isTombed( c: updated, root, parent: currentNode);
459             return currentNode->main.compare_exchange_strong(m, updated) ? INSERT_SUCCESSFUL : INSERT_RESTART;
460         } else if (level == MAX_LEVEL_COUNT) {
461             transformToWithMergedChild(updated, subNode: s, newNode, path);
462             updated->isTomb = isTombed( c: updated, root, parent: currentNode);
463             return currentNode->main.compare_exchange_strong(m, updated) ? INSERT_SUCCESSFUL : INSERT_RESTART;
464         } else {
465             transformToWithDownChild(updated, newChild: newNode, oldChild: s, level, path);
466             updated->isTomb = isTombed( c: updated, root, parent: currentNode);
467             return currentNode->main.compare_exchange_strong(m, updated) ? INSERT_SUCCESSFUL : INSERT_RESTART;
468         }
469     } else if (subNode->type == INODE) {
470         return insert(static_cast<INode *>(subNode), currentNode, newNode, level + 1);
471     } else {
472         fprintf( stream: stderr, format: "Node with unknown type\n");
473         assert(false);
474         return INSERT_RESTART;
475     }
476 }
477

```

Рисунок 4.2 – Реализация приватного метода вставки в lock-free версии

В данной реализации рекурсивному алгоритму был предпочтён не рекурсивный через бесконечный цикл. Это помогает избежать проблемы переполнения стека при большом количестве неуспешных попыток произвести вставку, а значит большом количестве рекурсивных вызовов функций.

2.5 Решение проблемы АВА

Проблема АВА возникает, когда поток считывает из ячейки данных дважды значение, это значение оказывается одинаковым и это воспринимается как «ячейка данных не менялась». Однако, это может быть неверным, ведь между двумя считываниями другой поток мог поменять указатель, изменить значение по нему, а потом вернуть прежний указатель.

Существует несколько подходов для решения проблемы АВА. Один из них hazard pointers (опасные указатели). Данная схема позволяет освобождать ячейку памяти только когда на неё никто не ссылается.

Hazard pointer реализованы в библиотеке libcds. Это позволяет внедрить их в код и избавиться от АВА. Для этого любые места памяти, которые необходимо удалить, помечаются как hazard pointer. Такие указатели удаляются отложено, когда это можно будет сделать корректно.

Чтобы подключить hazard pointers, соответствующий объект класса передаётся через шаблонные параметры (рисунок 5.1)

```
90         template<class GC, typename K, typename V>
91         class Hamt {
92             typedef GC gc;
93             typedef K key_type;
94             typedef V value_type;
95
```

Рисунок 5.1 – Класс Hamt, принимающий в виде шаблона hazard pointer

2.6 Внедрение в библиотеку libcds

Для внедрения в библиотеку libcds потребовалось изучение типовой структуры других контейнеров. Код декомпозировался на различные namespaces, внедрялся созданный в исходном коде аллокатор памяти.

3 ТЕСТИРОВАНИЕ

Для внедрённого алгоритма были написаны unit-тесты. Часть из них была однопоточной, часть использовало множество потоков, чтобы проверить потокобезопасность структуры. Были покрыты все сценарии во всех методах Hamt, также была проверена работа структуры bitmap. Unit-тесты были написаны при помощи библиотеки googletest.

На рисунке 1.1 проверяется, что дерево сократилось правильно. Добавляются три узла, а затем удаляет тот, что должен вести к сокращению. Далее с помощью ASSERT_EQ проверяется структура сокращённого дерева.

```
170  TEST(TRIE, HAPPY_FLOW_CONTRACTED_CHECK_WITH_THREE_KEYS) {
171      // arrange
172      Hamt<int, int> hamt;
173
174      // act & assert
175      hamt.insert(0b00000, 1);
176      hamt.insert(0b00001, 2);
177      hamt.insert(0b100001, 3);
178      ASSERT_EQ(hamt.root->main.load()->getChildCount(), 2);
179      hamt.remove(0b100001);
180
181
182      ASSERT_EQ(hamt.root->main.load()->array[0]->type, SNODE);
183      ASSERT_EQ(hamt.root->main.load()->getChildCount(), 2);
184      ASSERT_EQ(hamt.root->main.load()->array[1]->type, SNODE);
185  }
```

Рисунок 1.1

На рисунке 1.2 проверяется работа операции метода unset, который должен с помощью побитовых операций установить 0 на определённой позиции бита в числе.


```

95  TEST(BITMAP, HAPPY_FLOW__UNSET_AT_FALSE) {
96      // arrange
97      Bitmap bmp = {0b10101};
98
99      // act
100     bmp.unset(3);
101
102     // assert
103     ASSERT_EQ(bmp.data, (Bitmap) {0b10101}.data);
104 }

```

Рисунок 1.2

В качестве многопоточных тестов добавлялись сценарии:

- Вставка по 10000 значений одновременно 4 потоками
- Вставка по 10000 значений одновременно 4 потоками и последующее удаление
- Случайное количество потоков случайно выполняют одну из трёх операций (вставка, удаление, поиск)
- Предварительное заполнение структуры 100000 ключей и одновременное удаление несколькими потоками.

При помощи библиотеке google benchmark были написаны тесты производительности. Были рассмотрены структуры:

- однопоточная версия НАМТ
- блокирующая многопоточная НАМТ
- lock-free неблокирующая НАМТ
- set из стандартной библиотеки C++
- MichaelHashMap из библиотеки libcds.

Каждая структура был протестирована на следующих сценариях:

- Добавление множества значений
- Удаление множества значений из дерева
- Поиск по множества ключей

- Добавление, затем поиск, затем удаление, затем снова поиск.

Результаты тестирования можно увидеть на рисунке 1.3. Было сделано 10 замеров, затем они сортировались по времени исполнения.

4 ЗАКЛЮЧЕНИЕ

В данной работе удалось внедрить в библиотеку libcds lock-free структуру данных Hash Array Mapped Trie. Замеры производительности показали, что данная реализация при одновременном удалении множества ключей и при одновременном добавлении множества ключей показывает скорость исполнения ниже, чем в аналогичных структурах данных и однопоточной и блокирующей версии Hash Array Mapped Trie. Были протестированы основные сценарии исполнения и покрытие кода составило больше 90%.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Aleksandar Prokopec, Phil Bagwell, Martin Odersky Cache Aware Lock-Free Concurrent Hash Tries [Электронный ресурс] – 2011. – URL: https://www.researchgate.net/publication/313432754_Cache-Aware_Lock-Free_Concurrent_Hash_Tries (дата обращения 15.04.2022)
2. Fredkin E. Trie memory. Communications of the ACM 3 [Электронный ресурс] – 1960 – URL: <https://dl.acm.org/doi/10.1145/367390.367400> (дата обращения 15.04.2022)
3. Smotherman M. History of multithreading [Электронный ресурс] – 2005. – URL: <https://people.cs.clemson.edu/~mark/multithreading.html#:~:text=History%20of%20Multithreading&text=Summary%3A%20Multithreading%20first%20appeared%20in,investigated%20by%20IBM%20in%201968.&text=Most%20attempts%20at%20a%20history,PPUs%20in%20the%20CDC%206600> (дата обращения 15.04.2022)
4. Venu B. Multi-core processor – An overview [Электронный ресурс] – 2011. – URL: https://www.researchgate.net/publication/51945986_Multi-core_processors_-_An_overview (дата обращения 15.04.2022)
5. Phil Bagwell. Ideal Hash Trees [Электронный ресурс] – 2001. – URL: https://www.researchgate.net/publication/2378571_Ideal_Hash_Trees (дата обращения 15.04.2022)
6. Time and Space Lower Bounds for Non-Blocking Implementation [Электронный ресурс] – 1996 – URL: <https://dl.acm.org/doi/10.1145/248052.248105> (дата обращения 15.04.2022)