ТИТУЛЬНЫЙ ЛИСТ

СОДЕРЖАНИЕ

[АКТУАЛЬНОСТЬ 4](#_Toc103778293)

[ВВЕДЕНИЕ 3](#_Toc103778294)

[ОБЗОР АНАЛОГОВ 6](#_Toc103778295)

[ВВЕДЕНИЕ В HASH ARRAY MAPPED TRIE 9](#_Toc103778296)

[АЛГОРИТМ ЛОК-ФРИ HASH ARRAY MAPPED TRIE 19](#_Toc103778297)

[РАЗРАБОТКА 23](#_Toc103778298)

[ТЕСТИРОВАНИЕ 24](#_Toc103778299)

[ЗАКЛЮЧЕНИЕ 25](#_Toc103778300)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 26](#_Toc103778301)

# АКТУАЛЬНОСТЬ

Интерес к лок-фри алгоритмам не уходит. Так, например, Google Trends, веб-сайт, который анализирует информацию поисковых запросов в Google, приводит следующую статистику [] для словосочетания lock-free (рисунок X.X). Данные были взяты по поисковым запросам всего мира за последние пять лет. По горизонтали – даты, по вертикали – шкала популярности (0 – недостаточно данных, 100 – самый большой уровень популярности запроса, 50 – популярность запроса, которая вдвое меньше, чем 100).

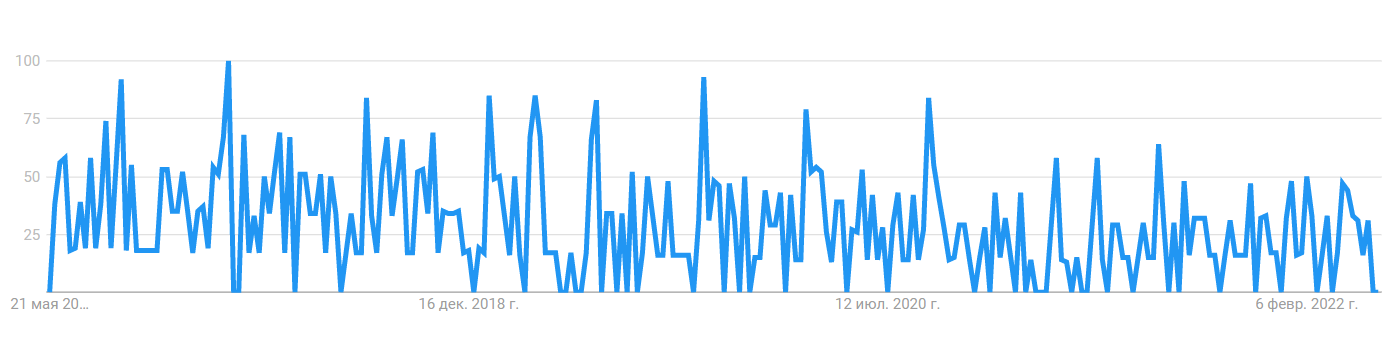


Рисунок X.X – Статистика популярности запроса «lock-free»

Интерес говорит о том, что у разработчиков программного обеспечения есть потребность использовать лок-фри алгоритмы или лок-фри структуры данных. Но писать свой собственный лок-фри код – процесс экстремально сложно и трудозатратный. Поэтому, если задача не слишком специфична, намного эффективнее воспользоваться уже готовыми решениями.

Одним из таких решений является библиотека libcds. [] В основном поддерживаемая Максимом Хиджинским, она включает в себя множество лок-фри структур данные на языке C++ таких как: MichaelHashMap, SkipListMap, FeldmanHashMap, SplitOrderedList и так далее.

Интерес к libcds за последние 5 лет тоже не уходит (рисунок X.X) []



Рисунок X.X – Статистика популярности запроса «libcds»

На данный момент, в libcds есть множество нереализованных структур данных. Одна из них, lock-free Hash Array Mapped Trie. Это и станет главное задачей работы: внедрить алгоритм [] lock-free алгоритм Hash Array Mapped Trie в библиотеку libcds.

# ВВЕДЕНИЕ

Идея создания многопоточных подходов появилась и использовалась ещё в середине прошлого века [4]. Но её бурное развитие началось с повсеместным распространением многоядерных процессов [5]. Это раскрыло их потенциал через использование многопоточного кода, где задействуется вся мощность компьютера.

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 серьёзным проблемам: дедлок, ухудшение быстродействия из-за затрат на перевод режим ядра.

Альтернативой служит неблокирующий подход, который не переводит потоки в режим ядра, а заставляет каждый беспрерывно исполнять код. В этом случае понятие критической секции, в которой в любой момент времени может находится 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 в зависимости от гарантий, которые они дают: [6]

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 |

Из трёх вышеприведённых типов наиболее используемый тип «без блокировок (то есть лок-фри). Он 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 *лок-фри структур данных*, которые позволяют производить с ними операции многопоточно.

Одной из таких лок-фри структур данных является Hash Array Mapped Trie. Впервые Hash Trie была описана Брандисом и Фредкином. [2] Позже, она была модифицирована Бегвилом [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 лок-фри 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 c глубины.

В данный момент реализация лок-фри Hash Array Mapped Trie есть на языке Java, но отсутствует для С++. Так, в библиотеке лок-фри структур данных libcds [3] необходима реализация этой структуры данных на языке С++. Именно это и является главной целью данной работы.

В качестве задач, шагов к выполнению были предложены следующие:

1. Анализ Hash Array Mapped Trie структуры данных
2. Реализация однопоточной версии
3. Реализация блокирующей многопоточной версии
4. Переход к неблокирующей многопоточной версии HAMT
5. Решение проблемы ABA
6. Внедрение в библиотеку libcds
7. Добавление в libcds

# ОБЗОР АНАЛОГОВ

Напомним, какая проблема решает при внедрении лок-фри 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 операции 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 посредством одной из следующих структур данных.

Таблица 2 – Структуры данных и амортизационное время операций

|  |  |  |  |
| --- | --- | --- | --- |
| Структура данных | 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 тоже 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 это 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 лок-фри структура. Это даёт мощный прирост к производительности её базовых операций.

# ВВЕДЕНИЕ В HASH ARRAY MAPPED TRIE

Идея внедряемого алгоритма была описана в статье Cache Aware Lock-Free Concurrent Hash Tries [1]. Перед тем как изложить соответствующие устройства лок-фри 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. Если число, то на биты. Далее для каждой части создаётся репрезентирующий её узел. Например, на рисунке X.X показано разбиение строки «hton»

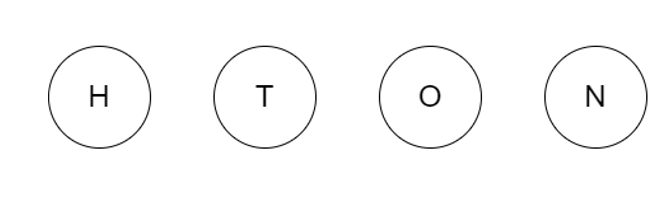


Рисунок X.X – Узлы из «hton»

А число 9 будет предварительно переведено в двоичную систему разбито на биты 1001 как показано на рисунке X.X

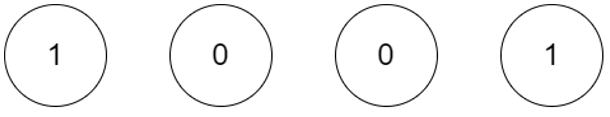


Рисунок X.X – Узлы из ключа 9

После разбития 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 , … ... узлы, мы находимся в узле дерева и уже вставили все узлы до .

Тогда,

* если уже имеет ребёнка (назовём его ), то мы пропускаем вставку (так, как он уже есть = ) и вставляем начиная c .
* если не имеет ребёнка , то мы добавляем в ребёнка (назовём его ) и вставляет начиная с .

В обоих случаях, если был обработан последний узел (), то 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 X.X. Узлы красного цвета являются терминальными, белого – промежуточными, зелёный узел соответствует корню.

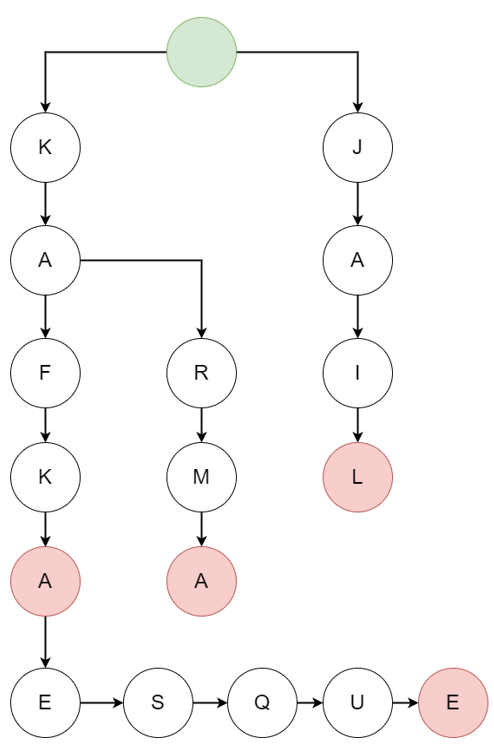


Рисунок X.X – Префиксное дерево, хранящее строки

Стоит заметить, trie компактно хранит ключи: узлы строк с одинаковыми префиксами не дублируются, а хранятся в единственном экземпляре («kafka» и «karma» имеют общий префикс «ka»). 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 символов (обычно = 256, что соответствует размерности таблицы ASCII). То есть, всего может быть различных видов узлов в дереве и у каждого узла может быть не больше, чем различных детей. Тогда пусть каждый узел хранит ссылки на своих детей в собственном фиксированном массиве на элементов (). Если , значит данный узел не имеет ребёнка, узла с типом . На рисунке X.X изображён узел A, который имеет только трёх детей B, C, D, но хранит информацию об отсутствии 253 других возможных детей (например, , то есть отсутствует узел C).

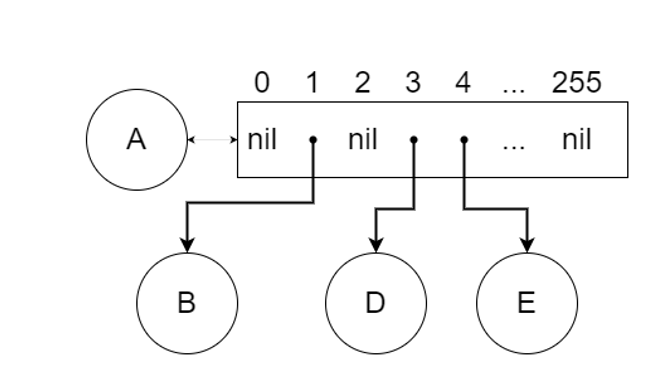


Рисунок X.X – Узел в наивной реализации префиксного дерева

Данный подход имеет плюс в скорости поиска следующего узла (асимптотика – Вместе с тем, огромное количество памяти расходуется впустую на хранение .

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

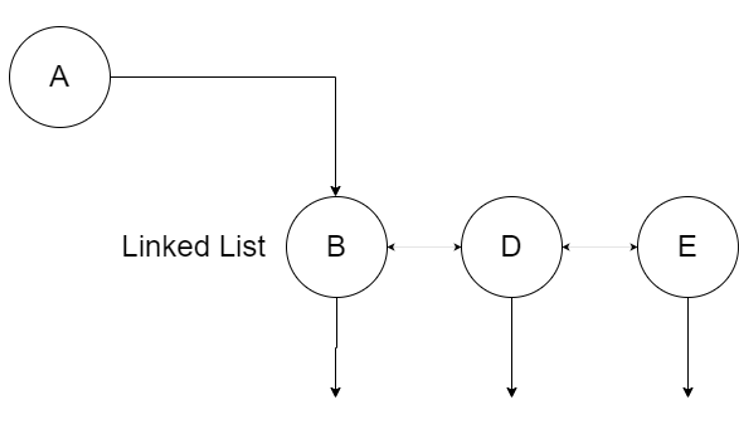


Рисунок X.X – Узел в реализации префиксного дерева со связным списком

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.

После детального рассмотрения префиксного дерева, можно приступить к устройству Hash Array Mapped Trie. Hash Array Mapped Trie (HAMT) 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, HAMT разбивает свой ключ на части. Эти части определяют путь до ключа/значения. Ключи могу 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 . Каждые пять бит – это отдельное число. Например, если в дерево добавляется ключ, который хешируется в число 2172 (двоичную форму будем обозначать так ), то он будет разбит на (рисунок X.X)

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

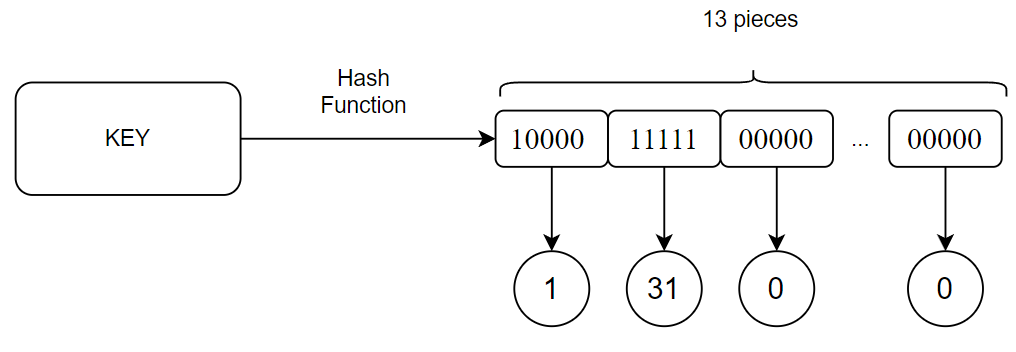
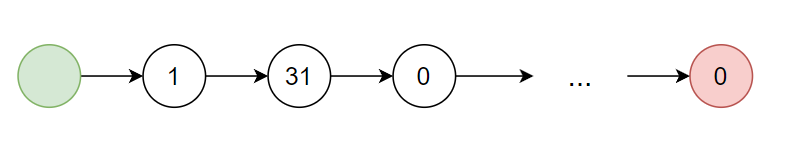


Рисунок X.X – Разбиение хеша ключа на части

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

Рисунок X.X – 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 X.X)



Рисунок X.X – Hash Array Mapped Trie не содержащий ни одного ключа

Добавим в дерево ключ с хешом 2172 (рисунок X.X). 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 понадобиться для проверки при поиске).

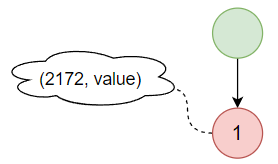


Рисунок X.X – Hash Array Mapped Trie c единственным ключом.

Теперь, если нам необходимо будет найти ключ 2172, мы проходить не по всем узлам (ведь дерево их не содержит), а лишь по части и них. 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 () 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 X.X). Эти узлы вычисляются уже по второй части хеша.

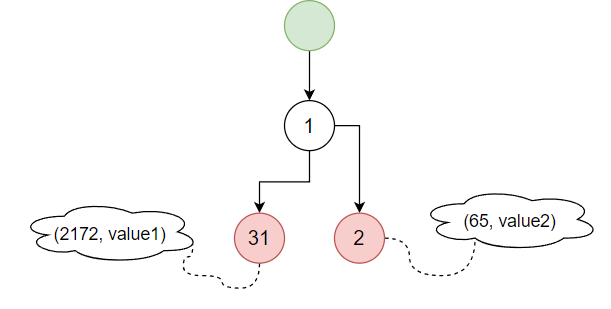


Рисунок X.X – Hash Array Mapped Trie c двум ключами с общим префиксом у хешей.

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 рисунках X.X и X.X показан процесс разрешения коллизии, при добавлении двух ключей с идентичными первыми 12 частями хеша.

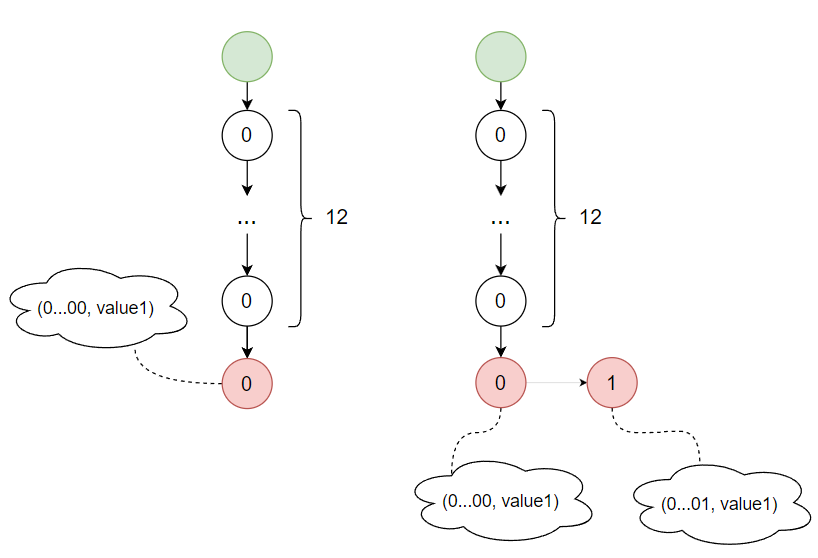


Рисунок X.X – Разрешение коллизии

Теперь осуществим удаление. Оно будет производится рекурсивно, начиная с терминальной вершины, соответствующей ключу: если текущая вершина не имеет детей, то она подлежит удалению. То есть, HAMT 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 , где , … ... , мы находимся в узле дерева и уже вставили все узлы до . Тогда:

* если у текущего узла дерева нет ребёнка , мы добавляем его, помечаем терминальным и вкладываем
* если у текущего узла есть нетерминальный ребёнок , то пропускаем вставку и производим добавление следующей части хеша , но уже c узла .
* если (то есть, мы находимся ниже предпоследнего уровня глубины дерева) и у текущего узла есть терминальный ребёнок с ключом и значением , то сначала проверяем не совпадает ли он с . Если да, то заменяем внутри этого ребёнка на. Иначе, делаем из него промежуточный узел и добавляем , и
* если (то есть, мы находимся на предпоследнем уровне глубины дерева) и у текущего узла есть терминальный ребёнок с ключом и значением , то мы заменяем его на терминальный узел-список, состоящий из
* если (то есть, мы находимся на предпоследнем уровне глубины дерева) и у текущего узла есть терминальный узел-список ребёнок, то проверяем есть ли в этом списке пара , где . Если имеется, то заменяем на . Иначе, добавляем в список новую

*Поиск:* пусть нам необходимо найти ключ , где , … ... , мы находимся в узле дерева и уже спустились на уровень . Тогда:

* если у текущего узла дерева нет ребёнка , то ключ/значение не найдены.
* если у текущего узла есть нетерминальный ребёнок , то производим поиск следующей части хеша , но уже с узла .
* если у текущего узла есть терминальный ребёнок с ключом и значением, то проверяем есть ли в этом списке пара , где . Если да, то и есть искомая пара. Иначе, ключ/значения не найдены.

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

* если текущая промежуточная вершина не имеет детей, то удаляем её (удаляем ссылку на неё у родителя)
* если текущая терминальная вершина не имеет детей, то удаляем её, если она хранит единственный ключ. Если это узел-список, то удаляем из неё такую пару , что .
* 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 HAMT 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 , 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 ) у узла P (рисунок X.X) говорит о наличии трёх детей: 0, 1, 3.

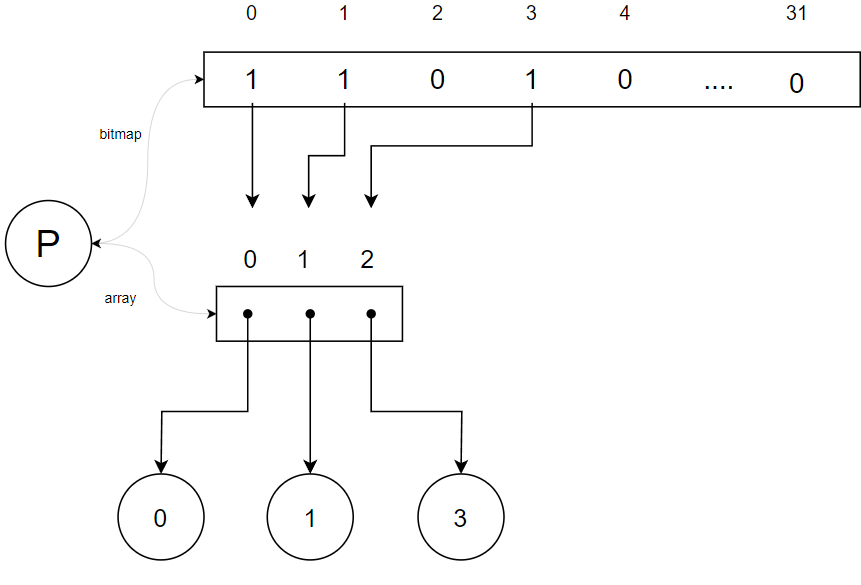


Рисунок X.X – Узел с битмапой и массивом(array), содержащий трёх детей.

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

Формула искомого индекса: , где popcount – функция подсчёта поднятых битов в числе, – позиция в битмапе с установленным битом, & - операция побитового «И», – побитовый сдвиг числа 1 раз влево. Получения бита в битмапе производится тоже через побитовые операции: . 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, возратить 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 ребёнок узла или его отсутствие.

# АЛГОРИТМ ЛОК-ФРИ HASH ARRAY MAPPED TRIE

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

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

Все три типа унаследованы от абстрактного узла Node.

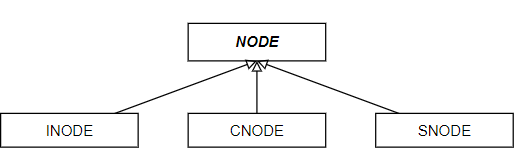


Рисунок X.X – UML-диаграмма для узлов

Начнём объяснения используя SNode и CNode. В дальнейших рассуждениях будем усложнять структуру данных.

Пусть у нас есть дерево с узлом SNode(рисунок X.X). Надпись на направленном ребре обозначает тип ребёнка.

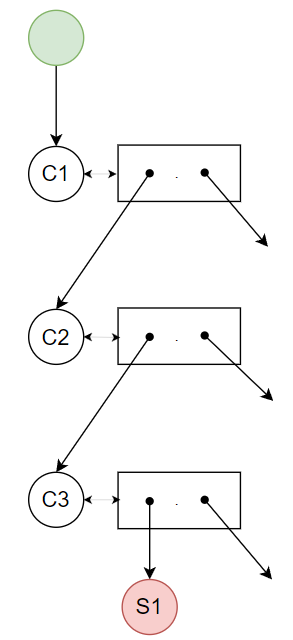


Рисунок X.X – Hash Array Mapped Trie с узлом S1

Тогда, если мы хотим втавить в С3 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 соответсвенно 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 создержать версию потока T2, который не учивает 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 , меняет его локально, и уже потом обновляет в С2 указатель на С3 на

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 )
2. значение, которое лежало по этому указателю (назовём это )
3. обновлённый узел (назовём это )

Если , то мы меняем содержимое указателя на и возвращаем , как показатель того, что значение было обновлено. Иначе возвращаем .

В нашем случае, поток T1 сначала зафиксируем локально чему равно С3. Потом сделает его копию , начнёт менять (добавит новый узел). И воспользуется сas, куда передаст указатель на C3 в качесте p, зафиксироованный ранее в качестве и в качестве . 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 иструкцию, что позволяет считать её лок-фри, то есть неблокирующей.

# РАЗРАБОТКА

# ТЕСТИРОВАНИЕ

# ЗАКЛЮЧЕНИЕ

# 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)