|  |
| --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ высшего образования |
| **«Национальный исследовательский ядерный университет «МИФИ»** |
| ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ |
| КАФЕДРА «КОМПЬЮТЕРНЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ» (№12) |

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**на курсовую учебно-исследовательскую работу по дисциплине**

**ЛОГИЧЕСКИЕ ОСНОВЫ СИСТЕМ СПЕЦИАЛЬНОГО НАЗНАЧЕНИЯ С ЭЛЕМЕНТАМИ СИСТЕМНОГО АНАЛИЗА**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тема: Анализ специальной структуры данных.** | | | | |
| Студент | Солоненко Никита Павлович | | Группа | С22-501 |
|  | ФИО | |  |  |
| Руководитель | | Кулик Сергей Дмитриевич, д.т.н., профессор НИЯУ МИФИ | | |
|  | | ФИО, степень, звание, должность | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент |  |  | Солоненко Н.П. |
|  | подпись |  | ФИО |
| Руководитель |  |  | Кулик С.Д. |

подпись

**Москва, 2024**

СОДЕРЖАНИЕ

1. Введение
2. Префиксные деревья
   1. Определение
   2. Основные свойства префиксных деревьев
   3. Область применения префиксных деревьев
   4. Структура узла
   5. Вставка в дерево
   6. Поиск ключа
   7. Удаление ключа
   8. Обход дерева
   9. Выводы
3. Сравнительный анализ с существующими структурами данных
   1. RB-tree в языке С++
   2. Hash-tables в языке С++
   3. B-tree в языке С++
   4. Вставка ключа
   5. Поиск ключа
   6. Выводы
4. The Adaptive Radix Trie
   1. Radix Trie
   2. The Adaptive radix Trie
   3. Выводы
5. Заключение
6. Список использованных источников
7. Приложения
8. **Введение**

После десятилетий увеличения объема оперативной памяти даже большие транзакционные базы данных помещаются в оперативную память. Когда большая часть данных кэшируется, традиционные системы баз данных привязаны к процессору, поскольку они тратят значительные усилия на то, чтобы избежать доступа к диску. Это привело к очень интенсивным исследованиям и коммерческой деятельности в таких системах баз данных с оперативной памятью. Системы оптимизированы для нового аппаратного обеспечения и, следовательно, намного быстрее.

Более 25 лет назад в качестве структуры индексации в памяти было предложено T-дерево [4]. К сожалению, кардинальные изменения в архитектуре процессоров сделали T-деревья, как и все традиционные деревья бинарного поиска, неэффективными на современном оборудовании. Причина в том, что постоянно растущий объем кэш-памяти процессора и растущая скорость основной памяти сделали устаревшим базовое предположение о равномерном времени доступа к памяти. Варианты B+-дерева, такие как B+-дерево с чувствительностью к кэшу [5], имеют более удобные для кэширования схемы доступа к памяти, но требуют более дорогостоящих затрат операции обновления. Кроме того, эффективность как бинарных, так и B+-деревьев страдает из-за другой особенности современных процессоров: поскольку результат сравнений не может быть легко предсказан, длинные конвейеры современных процессоров останавливаются, что приводит к дополнительным задержкам после каждого второго сравнения (в среднем).

Эти проблемы традиционных деревьев поиска были решены в ходе недавних исследований структур данных, специально разработанных для обеспечения эффективности в современных аппаратных архитектурах. Дерево поиска k-ary[6] и дерево, чувствительное к быстрой архитектуре (FAST) [7] используйте параллелизм на уровне данных для одновременного выполнения нескольких сравнений с помощью одной инструкции с несколькими данными (SIMD). Кроме того, FAST использует структуру данных, которая позволяет избежать ошибок в кэше за счет оптимального использования строк кэша и буфер предварительного просмотра перевода (TLB). Хотя эти оптимизации повышают производительность поиска, обе структуры данных не могут поддерживать постепенное обновление.

Хэш-таблицы - еще одна популярная структура данных в оперативной памяти. В отличие от деревьев поиска, которые имеют время доступа O(log n), хэш-таблицы имеют ожидаемое время доступа O(1) и , следовательно, намного быстрее хранятся в основной памяти. Тем не менее, хэш -таблицы реже используются в качестве индексов баз данных. Одна из причин заключается в том, что хэш-таблицы распределяют ключи случайным образом и, следовательно, поддерживают только точечные запросы. Другая проблема заключается в том, что большинство хэш-таблиц не справляются с ростом, а требуют дорогостоящей реорганизации при переполнении с O(n) сложностью. Следовательно, современные системы сталкиваются с неудачным выбором между быстрыми хэш-таблицами, которые позволяют выполнять только точечные запросы, и полнофункциональными, но относительно медленными деревьями поиска.

1. **Префиксные деревья**
   1. **Определение**

**Префиксное дерево (Trie)** – это древовидная структура данных, которая используется для хранения набора ключей, в качестве которых могут выступать любые объекты или структуры, которые можно однозначно разбить на разряды.

* 1. **Основные свойства префиксных деревьев**
     1. Каждый внутренний узел помечен некоторым

«разрядом ключа» – частичным ключом.

* + 1. Все дочерние узлы для данного помечены различными, не совпадающими по значению разрядами.
    2. Дерево содержит данный ключ только тогда, когда целый ключ может быть восстановлен из частичных, путём прохода от корня дерева к одному из внутренних узлов или листьев.
    3. Ключи не хранятся явно, а вычисляются при операциях поиска
    4. Можно организовать обход дерева, при котором сохраняется свойство упорядоченности ключей.
  1. **Область применения префиксных деревьев**
     1. Эффективные алгоритмы поиска подстрок (Ахо-Корасик).
     2. Алгоритмы сжатия данных.
     3. Анализ естественных языков (синтаксический анализ)
     4. Эффективное хранения словарей (в том числе для орфографического анализа).
     5. Эффективное представление индексированных данных в ядре Linux.
     6. Индексы таблиц в реляционных базах данных.
  2. **Структура узла**
     1. Набор дочерних узлов. Может быть просматриваемой, упорядоченной и хэш таблицей, а также таблицей биективного отображения.
     2. Флаг действительного ключа.
     3. (Опционально) Ссылка на родительский элемент.
     4. (Опционально) Частичный ключ данного узла.
     5. (Опционально) Количество потомков.

Пример: предположим, что мы хотим реализовать дерево-словарь, то есть каждый узел дерева будет хранить слово. Важно: приведенная реализация верная для любых узлов, если разряды имеют размер в 1 байт.

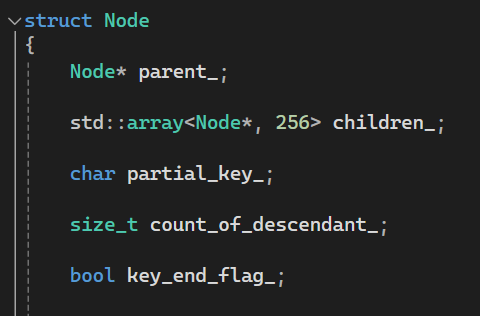


Рис. 2.1. Структура узла простейшего префиксного дерева на языке С++

* 1. **Вставка в дерево**

Начиная с корневого узла идем по дереву вниз. Если у очередного узла отсутствует потомок, соответствующий следующему фрагменту ключа – создаем. Узел, соответствующий последнему разряду ключа, помечаем флагом действительного ключа.

Для реализации вставки в дерево, необходимо для удобства реализовать функции для получения, добавления и проверки дочернего узла.

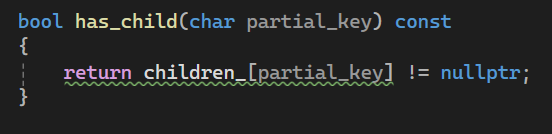


Рис. 2.2. Функция проверки наличия дочернего узла на языке С++

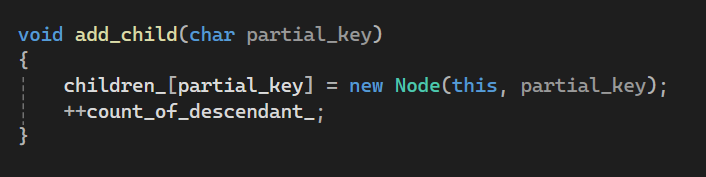


Рис. 2.3 Функция добавления дочернего узла на языке С++

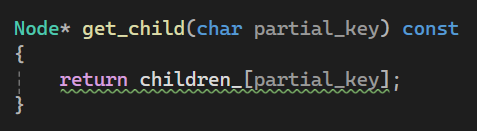


Рис. 2.4. Функция получения указателя на дочерний узел на языке С++

При наличии приведенных трех функций (в моем коде они представлены в качестве методов структуры Node) реализуем вставку ключа в префиксное дерево на языке программирования С++.

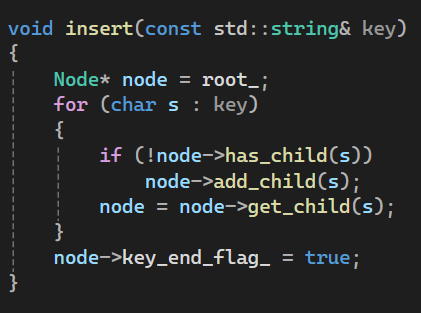


Рис. 2.5. Вставка ключа в префиксное дерево на языке С++

**Временная сложность операции получения и добавления дочерних элементов**, зависит от способа их хранения. В предложенной реализации доступ к дочернему элементу по частичному ключу осуществляется за О(1).

**Временная сложность вставки** элемента в предложенной реализации О(|k|), где |k| - длина искомого ключа. Время вставки не зависит от количества элементов в дереве.

Для хранения набора дочерних элементов используется дополнительная память в каждом узле дерева. Объем памяти на узел при такой реализации зависит от количества значений, которые может принимать частичный ключ элемента.

* 1. **Поиск ключа**

Начиная с корневого узла идём по дереву вниз. Если у очередного узла отсутствует потомок, соответствующий следующему фрагменту ключа – ключ не содержится в данном дереве. Проверяем, что узел, соответствующий последнему разряду ключа, помечаем флагом

действительного ключа, иначе – ключ не содержится в дереве.

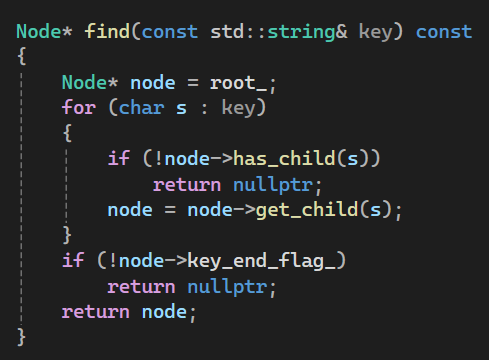


Рис. 2.6. Поиск ключа в префиксном дереве на языке С++

**Временная сложность поиска элемента** в предложенной реализации О(|k|), где |k| - длина искомого ключа. Время поиска не зависит от количества элементов в дереве.

По временной и пространственной сложности операция поиска имеет те же характеристики, что и операция вставки.

* 1. **Удаление ключа**
     1. Используя операцию поиска, находим позицию финального узла для данного ключа, если он есть в дереве.
     2. Снимаем с данного узла пометку «финального».
     3. Начиная с текущего узла, идём вверх по дереву к родительским узлам, удаляя из их детей текущий узел, если он не является финальным и, если у него не осталось потомков.
     4. Заканчиваем третий этап алгоритма, как только мы не удалили очередной узел или дошли до корня дерева.

Для реализации удаления из дерева, необходимо для удобства реализовать функцию для удаления дочернего узла.

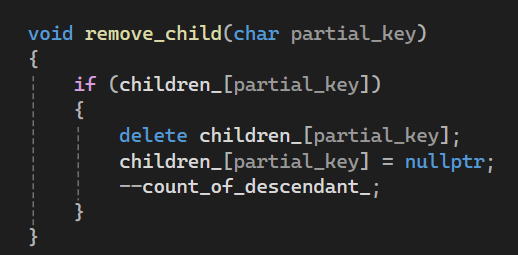


Рис. 2.7. Функция для удаления дочернего узла на языке С++

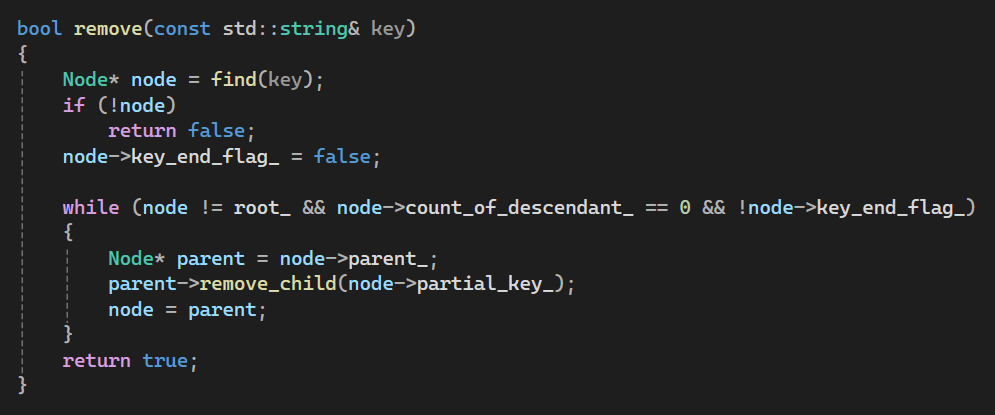


Рис. 2.8. Удаление ключа из префиксного дерева на языке С++

* 1. **Обход дерева**

Для сохранения свойства упорядоченности можно использовать следующий порядок обхода:

* + 1. Начинаем с корня. Кладём в конец списка частичный ключ текущего посещаемого узла.
    2. Если текущий посещаемый узел «финальный» – выводим текущее состояние списка.
    3. Посещаем всех потомков текущего узла в заданном порядке обхода (по возрастанию или убыванию значения частичного ключа).
    4. Если у текущего узла нет потомков - извлекаем из конца списка частичный ключ текущего узла и возвращаемся «вверх» по дереву к родительскому узлу.

Для реализации обхода префиксного дерева потребуется дополнительная память для хранения префиксов и рекурсивная функция, которая будет реализовывать приведенный выше алгоритм.

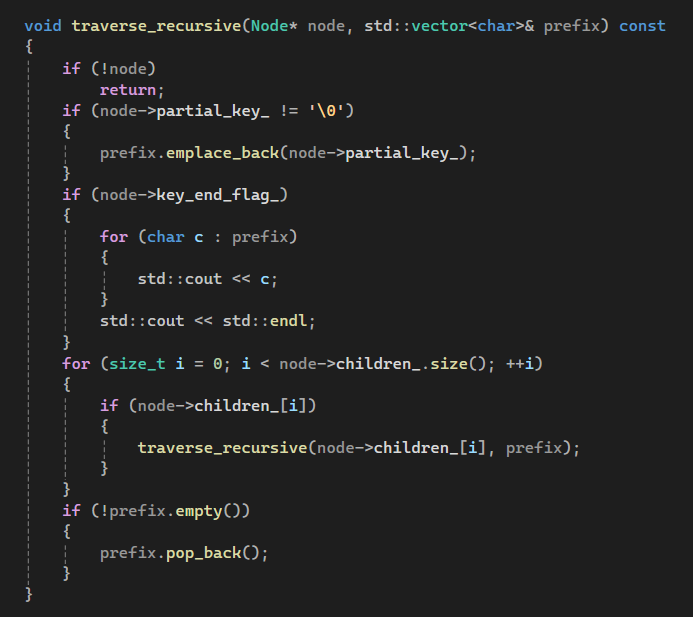


Рис. 2.9. Рекурсивная функция обхода префиксного дерева на языке С++

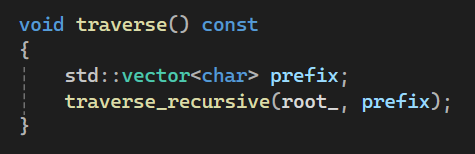


Рис. 2.10. Обход префиксного дерева на языке С++

* 1. **Выводы**

В данном разделе я описал принцип работы простейших префиксных деревьев. Подробно описал все базовые алгоритмы, с помощью которых можно осуществить реализацию, на любом языке программирования, для любого типа данных (который может быть представлен в виде последовательности байт) префиксного дерева. Оценил и описал сложность всех операций, которые можно проводить над данным деревом. На языке программирования С++ реализовал все эти алгоритмы для префиксного дерева-словаря.

Познакомил с тем, в каких областях можно применять префиксные деревья и чем они удобны, в следующей главе будет проведен сравнительный анализ, который наглядно продемонстрирует все преимущества и недостатки данных деревьев.

1. **Сравнительный анализ с существующими структурами данных.**

Весь сравнительный анализ будет построен на двух параметрах – N, |k|. Где N – количество элементов, уже содержащихся в структуре данных. |k| - разрядность ключа.

Для сравнительного анализа я буду использовать следующие структуры данных: сбалансированные деревья (в качестве балансировки будут выбраны RB и B), хэш таблица, а также непосредственно сами префиксные деревья.

Таблица 2.1. Сравнительная таблица известных структур данных

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Структура данных** | **Порядок ключей** | **Явное хранение ключей** | **Сложность вставки ключа** | **Сложность поиска ключа** | **Требования к типу ключа** | **Локальность данных** |
| **RB-дерево** | Да | Да | О(log(N)|k|) | О(log(N)|k|) | ==, <, > | Низкая |
| **B-дерево** | Да | Да | O(tlogt(N)|k|) | O(log(N)|k|) | ==, <, > | Высокая |
| **Хэш таблица** | Нет | Да | O(|k|) – хорошо  O(N|k|) – плохо | O(|k|) – хорошо  O(N|k|) – плохо | ==, hash | Высокая (Низкая при цепочка) |
| **Префиксное дерево** | Опционально | Опционально | O(|k|) | O(|k|) | ==, разбиение на разряды | Средняя |

Для сравнения я также приведу код на языке С++, чтобы наглядно показать разницу во времени, требуемого на выполнение тех или иных операций.

* 1. **RB-tree в языке С++.**

В языке С++ нет конкретной реализации красно-черного дерева, но есть ряд шаблонных контейнеров, который по умолчанию предполагают rb-балансировку. Самые популярные из них:

* + 1. std::set – представляет из себя красно-черное дерево, которое хранит в себе упорядоченное множество ключей. Ключи не могут повторяться.
    2. std::map – также, как и std::set представляет из себя красное-черное дерево, но хранит в себе дополнительную информацию.
    3. std::multiset – аналогичное красно-черное дерево, но ключи в данном случае могут повторяться, дополнительная информация отсутствует.
  1. **Hash-tables в языке С++.**

В языке С++ существует также несколько шаблонных контейнеров, реализующих поведение хэш таблицы. Отличия контейнеров между собой аналогичны, как и в случае rb-деревьев, где-то есть дополнительная информация, где-то ключи могут повторяться. Вот некоторые из них: std::unordered\_set, std::unordered\_map, std::unordered\_multiset.

* 1. **B-tree в языке С++.**

В языке С++ нет библиотечной реализации B-дерева, поэтому я воспользуюсь внешними библиотеками. Реализация B-дерева представлена в библиотеке absl. Для того, что в проекте подключить ее, можно прописать следующий код в CMakeLists.txt:

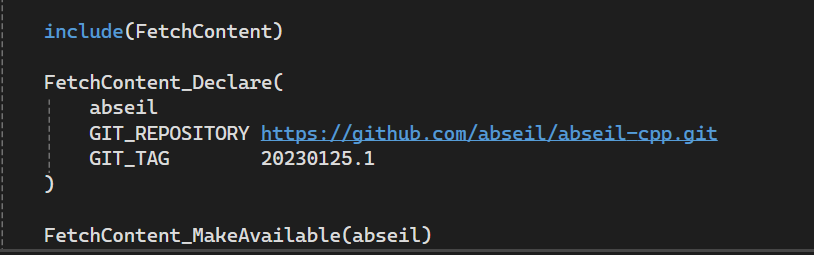


Рис.3.1. CMakeLists.txt для B-tree

* 1. **Вставка ключа.**

Произведем вставку сначала 10000 элементов, с длиной ключа 10 и 15 символов соответственно, а после 100000 элементов, с длиной ключа 10 и 15 символов аналогично. Для случайной генерации слов-ключей была написана функция (см. приложение 1). Код сравнения операции вставки представлен на языке С++ (см. приложение 2).

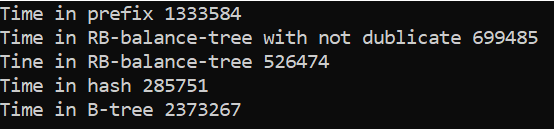


Рис.3.2. Результат выполнения программы для 100000 элементов для длины ключа 10 символов

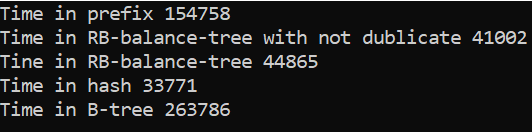


Рис.3.3. Результат выполнения программы для 10000 элементов для длины ключа 10 символов

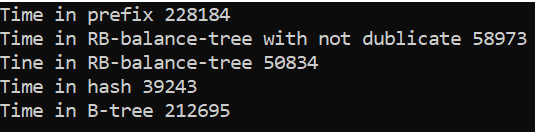


Рис.3.4. Результат выполнения программы для 10000 элементов для длины ключа 15 символов

* 1. **Поиск ключа.**

Произведем вставку 100000 элементов, с длиной ключа 10 и 15 символов соответственно, а после произведем поиск 10000 и 100000 элементов с заданными длинами ключа. Код сравнения операции поиска представлен на языке С++ (см. приложение 3).

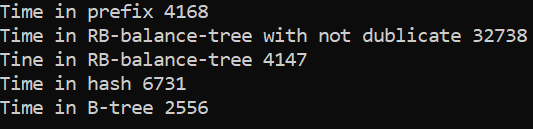


Рис.3.5. Результат выполнения программы для длины ключа 10 символов для 10000 элементов

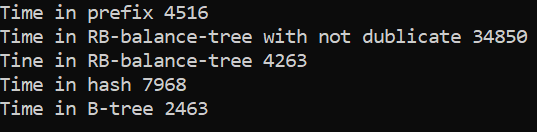


Рис.3.6. Результат выполнения программы для длины ключа 15 символов для 10000 элементов

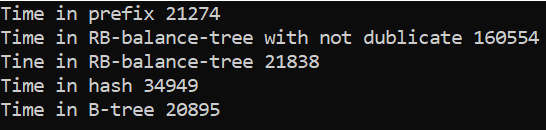


Рис.3.7. Результат выполнения программы для длины ключа 10 символов для 100000 элементов

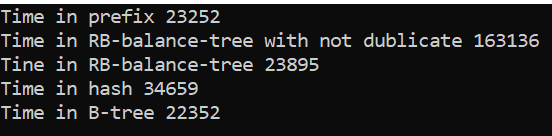


Рис.3.8. Результат выполнения программы для длины ключа 15 символов для 100000 элементов

* 1. **Выводы**
     1. Префиксные деревья имеют лучшие асимптотические характеристики временной сложности операций. (При этом довольно существенную константу).
     2. Время выполнения операций стабильнее, чем у хэш таблиц, при этом сохраняется еще и упорядоченность ключей.
     3. Кэш дружественность значительно выше, чем у бинарных деревьев, требуется значительно меньше выделений памяти. Однако уступают по этим показателям отдельным реализациям B деревьев и хэш таблиц.
     4. Высокий коэффициент использования пространства при «плотном» заполнении, но крайне низкий в разреженных случаях.
     5. Сложные требования к типу ключа, что значительно усложняет использование этих структур данных для общих случаев.

1. **The Adaptive Radix Trie**
   1. **Radix Trie.**

Перед тем, как рассмотреть Adaptive radix Trie, необходимо рассмотреть сжатое префиксное дерево или Radix дерево – модификация классического Trie, которая позволяет уменьшить высоту дерева в случае его разреженного заполнения. Radix дерево в своих узлах хранит не по одному разряду ключа, а последовательность элементов, у которых был бы только один дочерний узел при хранении в префиксном дереве того же набора данных.

* + 1. **Сравнение Trie и Radix Trie**

Пусть в дерево вставляются следующий набор ключей: abs, cdf, absd, absdf.

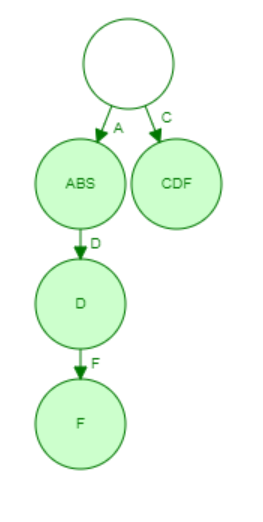
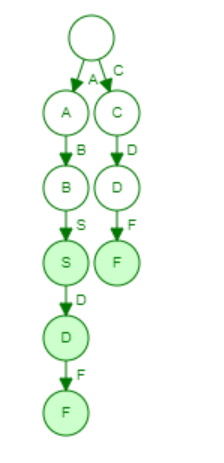


Рис. 4.1. Сравнение Radix Trie и Trie

Как мы видим из рисунка, Radix Trie представляет из себя дерево, которое предоставляет более оптимизированное хранение ключей, в сравнении с Trie, а как следствие уменьшает время, затрачиваемое на выполнение тех или иных операций.

* + 1. **Ленивое расширение.**

1. Операция применяется при вставке нового элемента в дерево.
2. Внутренний узел создаётся только если он необходим для различения по крайней мере двух дочерних узлов.
3. При таком подходе высота поддеревьев существенно уменьшается в тех случаях, когда существуют внутренние узлы, с уникальным фрагментом ключа.

Алгоритм ленивого расширения:

1. Поиск первого отличающегося разряда фрагмента искомого ключа и сжатого пути текущего узла.
2. Сдвиг фрагмента искомого ключа до первого отличающегося разряда.
3. Проверка. Если сжатый путь текущего узла является подстрокой (в том числе пустой) искомого ключа, то расширение не требуется, а фрагмент искомого ключа сдвигается.
4. Создание нового узла, сжатый путь которого – общий префикс у искомого и текущего.
5. В родительском узле текущего по ключу текущего узла устанавливается указатель на новый узел.
6. В новый узел добавляется текущий по частичному ключу равному первому отличающемуся разряду фрагмента искомого и текущего сжатого пути.
7. Сжатый путь текущего узла сдвигается.

Иллюстрация: произведем вставку в дерево сначала ключа abcdef, затем abcdegh, а после aba.

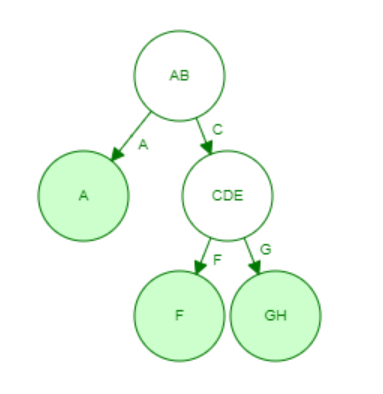
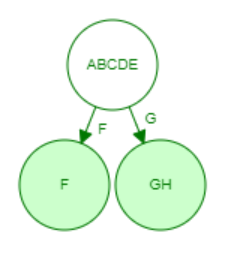


Рис. 4.2. Ленивое расширение

* + 1. **Сжатие пути.**

1. Операция применяется при удалении элемента.
2. Внутренний узел объединяется с родительским, если он имеет только одного потомка.
3. При таком подходе высота поддеревьев существенно уменьшается в тех случаях, когда появляются внутренние узлы, с уникальным фрагментом ключа.

Алгоритм сжатия пути:

1. Если количество потомков узла > 1 сжатие не требуется.
2. Находим ключ единственного ребенка.
3. Находим единственного ребенка по его ключу.
4. В родительском узле текущего узла устанавливаем по текущему ключу узел ребенка.
5. Изменяем сжатый путь потомка, добавляя к его началу сжатый путь текущего узла.
6. Удаляем текущий узел.
7. Производим операцию сжатия пути уже непосредственно для ребёнка.

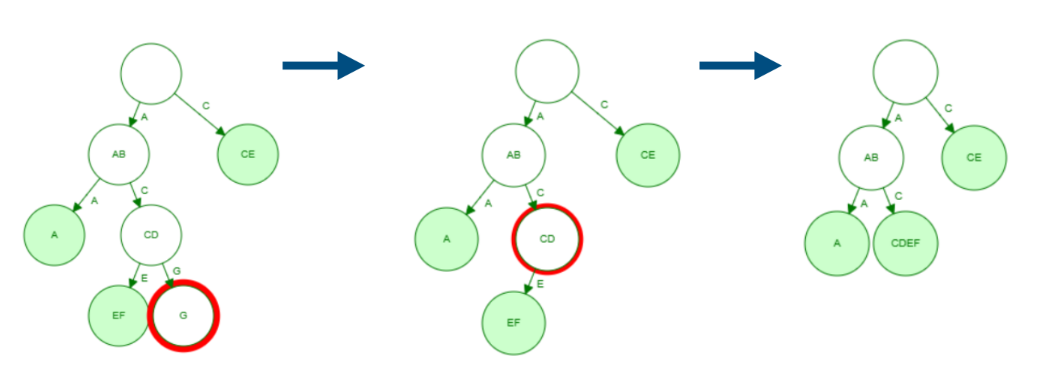


Рис 4.3. Иллюстрация сжатия пути при удалении ключа (удаляем ключ abcdg)

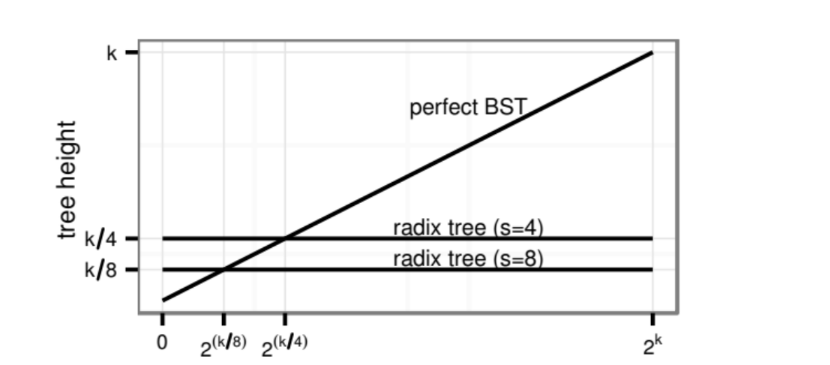


Рис 4.4. Зависимость высоты дерева от количества элементов и параметра span [15].

* 1. **The Adaptive Radix Trie.**

1. Дерево состоит из двух типов узлов: внутренние и листовые.
2. Внутренние узлы сопоставляют частичные ключи со следующими узлами.
3. Листовые узлы содержат значения элементов, соответствующие определенным ключам.
4. Наиболее эффективное представление внутренних узлов – массив из указателей размера 2^s. Во время обхода дерева s-разрядный фрагмент ключа используется в качестве индекса в этом массиве и, таким образом, определяет следующий дочерний узел без каких-либо дополнительных сравнений.
5. Параметр s (span) крайне сильно влияет на эффективность дерева, фактически определяя его высоту при заданной длине ключа.
6. Высота дерева при k-битном ключе определяется как [k/s]. (При s = 1 и 32 битном ключе высота будет равна 32, в то время как при s = 8 всего 4).
   * 1. **Структура внутренних узлов.**
7. Внутренние узлы префиксных деревьев биективно отображают множество частичных ключей на множество своих дочерних элементов.
8. Для экономии памяти в реализации используется 4 вида внутренних узлов различной ёмкости. Каждый из видов узлов позволяет эффективно, получить указатель на дочерний элемент, соответствующий данному частичному ключу, при этом не нарушается свойство упорядоченности.
9. Предложенная реализация использует параметр s равный 8 битам.
10. Вместо использования списка пар ключ/значение список разделён на одну ключевую часть, одну указательную часть. Это позволяет сохранить компактность представления, обеспечивая при этом эффективный поиск.
    * 1. **Node4**
11. Наименьший тип узла, способный вмещать до 4 элементов.
12. Представляет собой массив из 4-х ключей и массив из 4-х указателей на дочерние элементы.
13. Ключи отсортированы и лежат в позициях в соответствии с указателями на следующий элемент.

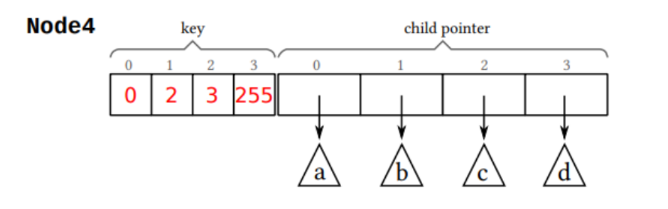


Рис. 4.5. Представление Node4 [15].

* + 1. **Node16.**

1. Тип узла, предназначенный для хранения от 5 до 16 элементов.
2. Как и узел для 4-х элементов, состоит из упорядоченного массива ключей и массива указателей на следующий элемент.
3. Ключ может быть эффективно найден бинарным поиском или с помощью параллельного сравнения нескольких элементов посредством векторных (SIMD) инструкций на современных CPU.

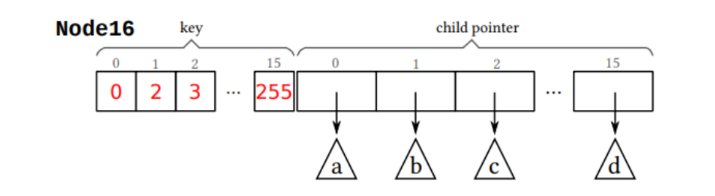


Рис. 4.6. Представление Node16 [15].

* + 1. **Node48.**

1. Тип узла, вмещающий от 17 до 48 элементов.
2. Представлен двумя массивами: массив ключей, который содержит 256 (всевозможные значения ключа) 6-битных (8-битных для эффективности реализации) индексов массива из 48 указателей на следующий элемент.

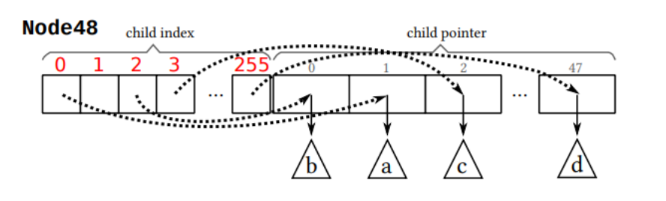


Рис. 4.7. Представление Node48 [15].

* + 1. **Node256.**

1. Узел, для хранения до 256 элементов.
2. Реализован, как в классических префиксных деревьях с s = 8 бит, в виде массива из 256 указателей на следующий элемент.

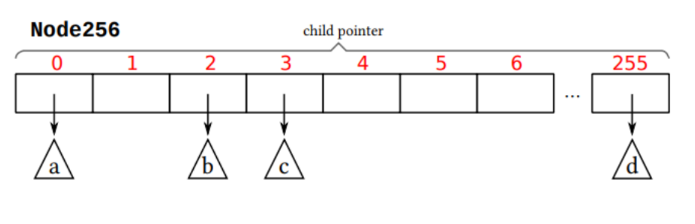


Рис. 4.8. Представление Node256 [15].

* + 1. **Листовые узлы**

Значения в листьях могут храниться несколькими различными способами:

1. Хранение с использованием дополнительного типа конечного узла, который хранит одно значение.
2. Хранение в одном из четырех различных типов конечных узлов, которые отражают структуру внутренних узлов, но содержат значения вместо указателей.
3. Если значения помещаются в указатели, нет необходимости в отдельных типах узлов. Вместо этого каждое место хранения указателя во внутреннем узле может хранить либо указатель, либо значение. Значения и указатели можно различать, используя один дополнительный бит на указатель или помощью тегирования указателя.

Явное хранение одного значения является самым простым в реализации и самым гибким способом хранения информации, однако увеличивает высоту дерева и требует дополнительного разыменования указателя при операциях поиска, что, в том числе, уменьшает кэш-дружественность операций обхода такого дерева.

Хранение нескольких значений в соответствии с типом узла является удобным подходом, всё ещё требующим дополнительные типы для листьев, однако не увеличивающим высоту дерева. Недостатком такого подхода является требование фиксированной длины ключа.

Третий подход является самым эффективным в случае, если размер значения не превышает размер указателя. Не требует введения дополнительных типов и увеличения высоты дерева.

* + 1. **Хранение сжатого пути.**

1. **Пессимистичный**. В каждом внутреннем узле хранится вектор частичных ключей (возможно пустой). В векторе содержатся все удалённые частичные ключи. При поиске данный вектор сравнивается с искомым ключом перед обработкой следующего дочернего узла.
2. **Оптимистичный**. В каждом внутреннем узле хранится только число удалённых предшествующих внутренних узлов, но не сами частичные ключи. При операциях поиска в искомом ключе просто пропускается соответствующее число частичных ключей. Когда при поиске достигнут лист, происходит сравнение искомого ключа с найденным, чтобы исключить неверный результат.

Оба подхода гарантируют, что у каждого внутреннего узла есть по крайней мере два дочерних узла.

Оптимистический подход особенно полезен для длинных ключей, но требует одной дополнительной проверки.

Пессимистический метод использует больше места и имеет узлы переменного размера, что приводит к повышенной фрагментации памяти.

На практике применяется гибридный подход – с сохранением вектора в каждом узле, как в пессимистическом подходе, но с постоянным размером (8 байт) для всех узлов. Только при превышении этого размера алгоритм поиска динамически

переключается на оптимистическую стратегию. Не тратя впустую слишком много места и не фрагментируя память, что позволяет избежать дополнительной проверки части случаев.

* + 1. **Вставка ключа.**

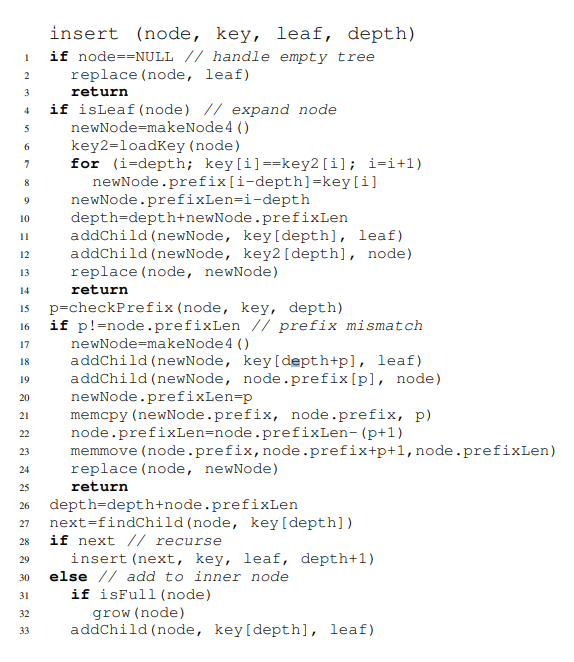


Рис. 4.9. Вставка ключа в ART

На рисунке 4.9. представлен псевдокод операции вставки в дерево. С помощью рекурсивного вызова в строке 29 выполняется обход дерева до тех пор, пока не будет найдена позиция для нового листа. Обычно лист можно просто вставить в существующий внутренний узел, предварительно вырастив его, если это необходимо (строки 31-33). Если из-за отложенного расширения обнаруживается существующий лист, он заменяется новым внутренним узлом, хранящим существующий и новый лист (строки 5- 13). Другой особый случай возникает, если ключ нового листа отличается от сжатого пути: над текущим узлом создается новый внутренний узел, а сжатые пути корректируются соответственно (строки 17-24). Я опускаю некоторые вспомогательные функции из-за нехватки места: replace заменяет узел в дереве другим узлом, addChild добавляет новый дочерний узел к внутреннему узлу, checkPrefix сравнивает сжатый путь к узлу с ключом и возвращает количество равных байт, grow заменяет узел на узел большего типа, и loadKey извлекает ключ листа.

* + 1. **Поиск ключа.**

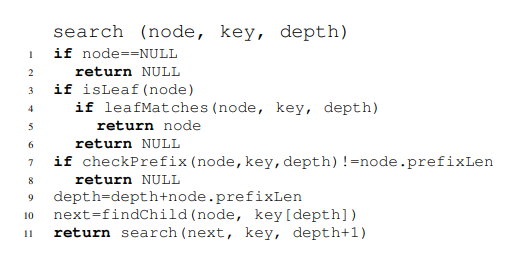


Рис. 4.10. Поиск ключа в ART

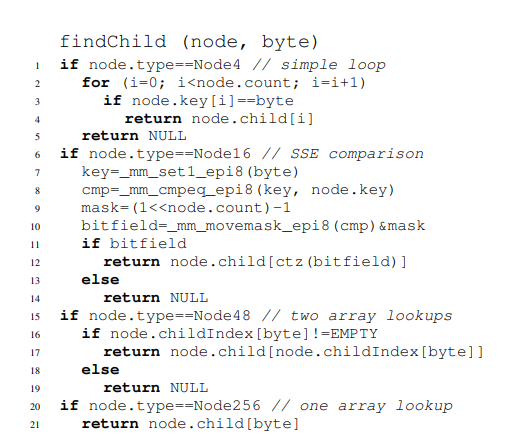


Рис. 4.11. Алгоритм поиска дочернего элемента во внутреннем узле по частичному ключу

Дерево просматривается с использованием последовательных байтов массива ключей, пока не будет найден конечный узел или нулевой указатель. Строка 4 обрабатывает отложенное расширение, проверяя, что найденный конечный узел полностью соответствует ключу. Сжатие пессимистического пути выполняется в строках 7 и 8 путем прерывания поиска, если сжатый путь не соответствует ключу.

Следующий дочерний узел находится с помощью функции findChild, которая показана на рисунке 4.11.

В зависимости от типа узла выбирается соответствующий алгоритм поиска: поскольку Node4 содержит только 2-4 записи, мы используем простой цикл. Для Node16 псевдокод показывает реализацию SIMD с использованием инструкций SSE, которые позволяют параллельно сравнивать 16 ключей с одной инструкцией. Во-первых, в псевдокоде искомый ключ копируется (строка 7), а затем сравнивается с 16 ключами, хранящимися во внутреннем узле (строка 8). На следующем шаге создается маска (строка 9), поскольку в узле может быть менее 16 допустимых записей. Результат сравнения преобразуется в битовое поле и применяется маска (строка 10). Наконец, битовое поле преобразуется в индекс с использованием значения count, завершающегося нулем инструкция (строка 12). В качестве альтернативы, если инструкции SIMD недоступны, можно использовать двоичный поиск. Поиск в Node48 выполняется путем предварительной проверки правильности записи childIndex, а затем возврата соответствующего указателя. Поиск по Node256 состоит только из одного обращения к массиву.

* + 1. **Эффективность поиска.**

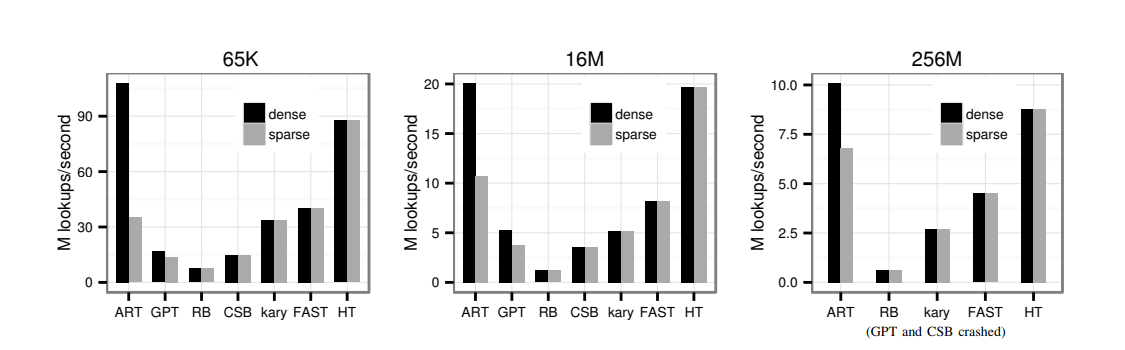
****

Рис. 4.12. Производительность однопоточного поиска в индексе с 65 тыс., 16 млн и 256 млн ключей [15].

На рисунке 3.11 показано, что наилучшая производительность у ART и хэш-таблицы. ART работает более чем в два раза быстрее, чем GPT – это дерево корней с половиной размаха и, следовательно, в два раза большей высотой. Красно-черное дерево – это самое медленное дерево, основанное на сравнении, за ним следуют CSB+-дерево, k-арный поиск и, наконец, быстрый поиск. Несмотря на то, что FAST не поддерживает обновления и оптимизирован для современных архитектур, он работает медленнее, чем ART и хэш-таблица. Относительная производительность структура данных очень похожа для трех размеров индексов. Это указывает на то, что небольшие индексы (65 тыс. В 10 раз быстрее, чем у больших индексов (256 млн), это в основном обусловлено эффектами кэширования, а не асимптотическими свойствами структур данных.

* + 1. **Эффекты кэширования.**

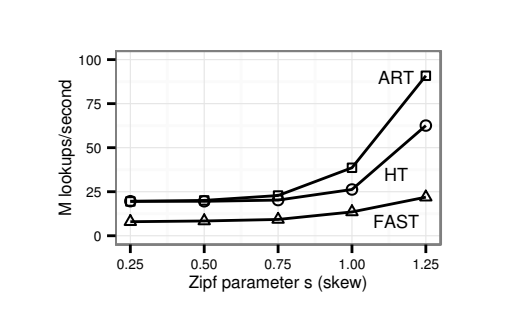


Рис 4.13. Влияние перекоса на производительность поиска (16 миллионов ключей) [15].

Для современных процессоров кэширование чрезвычайно важно, поскольку задержка DRAM составляет сотни циклов процессора. Древовидные структуры, в частности, очень выигрывают от кэширования, поскольку часто используемые узлы и верхние уровни обычно кэшируются. Чтобы количественно оценить эти эффекты кэширования, мы сравниваем две древовидные структуры, ART (с плотными ключами) и FAST, с хэш-таблицей. Случайный поиск, который мы выполняли до сих пор, является наихудшим вариантом для кэшей, поскольку этот шаблон доступа имеет плохую временную локальность. На практике очень распространены асимметричные схемы доступа, например, к последним заказам обращаются чаще, чем к старым. Мы смоделировали такой сценарий, используя распределенные ключи Zipf вместо случайных ключей. На рисунке 3.12. показано влияние увеличения асимметрии на производительность трех структур данных. Все структуры данных работают намного лучше при наличии перекоса, поскольку количество пропусков в кэше уменьшается. С увеличением перекоса производительность ART и хэш-таблицы приближается к их скорости в небольших резидентных деревьях кэша. Для FAST ускорение меньше, поскольку требуется больше сравнений и вычисления смещения, которые не улучшаются при кэшировании.

* 1. **Выводы.**

Я подробно проанализировал adaptive radix tree (ART) – быструю и экономичную структуру индексации в оперативной памяти. Высокая степень разветвления, сжатие путей и медленное расширение уменьшают высоту дерева и, следовательно, обеспечивают превосходную производительность. В худшем случае потребление пространства, распространенная проблема радикальных деревьев, ограничивается динамическим выбором компактных внутренних структур данных. Я сравнил ART с другими современными структурами данных в основной памяти. Результаты показывают, что ART работает намного быстрее, чем красно-черное дерево, чувствительное к кэшированию B+-дерево и GPT, еще одно предложение по созданию дерева оснований. Даже чувствительное к архитектуре, доступное только для чтения дерево поиска FAST, разработанное специально для современных процессоров, работает медленнее, чем ART, даже без учета обновлений. Из всех рассмотренных структур данных конкурентоспособной оказалась только хэш-таблица. Но хэш -таблицы неупорядочены и поэтому не подходят в качестве индексных структур общего назначения.

1. **Заключение**

В процессе выполнения курсовой работы были получены следующие результаты.

Изучен и проанализирован один из видов деревьев поиска – префиксное дерево. Данный вид позволяет оптимизировать алгоритмы поиска и хранения данных. Были реализованы алгоритмы поиска, вставки, удаления ключа для префиксного дерева на языке программирования С++.

Был проведен сравнительный анализ с существующими структурами данных, такие, как хэш-таблицы, B-tree, красно-черные деревья.

В рамках данной курсовой работы был проведен подробный анализ adaptive radix tree. Оно представляет из себя быструю и экономичную структуру индексации для баз данных в оперативной памяти. Высокая степень разветвления, сжатие путей и медленное расширение уменьшают высоту дерева и, следовательно, обеспечивают превосходную производительность. В худшем случае потребление пространства, распространенная проблема радикальных деревьев, ограничивается динамическим выбором компактных внутренних структур данных. Я сравнил ART с другими современными структурами данных в основной памяти. Результаты показывают, что ART работает намного быстрее, чем красно-черное дерево, чувствительное к кэшированию B+-дерево и GPT, еще одно предложение по созданию дерева оснований.

В будущем я намерен поработать над синхронизацией параллельных обновлений. В частности, я планирую разработать схему синхронизации без защелок, используя атомарные примитивы, такие как compare-swap. Другая идея заключается в разработке компактного дерева оснований, которое имеет узлы одинакового размера. Вместо динамической адаптации разветвления в зависимости от разреженности ключей, количество битов, используемых в ключе, должно изменяться динамически, в то время как разветвление должно оставаться приблизительно постоянным. Такое дерево также можно использовать для данных, хранящихся на диске.

1. **Список использованных источников**

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, “H-store: a high-performance, distributed main memory transaction processing system,” PVLDB, vol. 1, 2008.

[2] F. Farber, S. K. Cha, J. Primsch, C. Bornh ¨ ovd, S. Sigg, and W. Lehner, ¨ “SAP HANA database: data management for modern business applications,” SIGMOD Record, vol. 40, no. 4, 2012.

[3] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,” in ICDE, 2011.

[4] T. J. Lehman and M. J. Carey, “A study of index structures for main memory database management systems,” in VLDB, 1986.

[5] J. Rao and K. A. Ross, “Cache conscious indexing for decision-support in main memory,” in VLDB, 1999.

[6] B. Schlegel, R. Gemulla, and W. Lehner, “k-ary search on modern processors,” in DaMoN workshop, 2009.

[7] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, “FAST: fast architecture sensitive tree search on modern cpus and gpus,” in SIGMOD, 2010.

[8] R. Bayer and E. McCreight, “Organization and maintenance of large ordered indices,” in SIGFIDET, 1970.

[9] D. Comer, “Ubiquitous B-tree,” ACM Comp. Surv., vol. 11, no. 2, 1979.

[10] R. Bayer, “Symmetric binary B-trees: Data structure and maintenance algorithms,” Acta Informatica, vol. 1, 1972.

[11] L. Guibas and R. Sedgewick, “A dichromatic framework for balanced trees,” IEEE Annual Symposium on the Foundations of Computer Science, vol. 0, 1978.

[12] J. Rao and K. A. Ross, “Making B+ trees cache conscious in main memory,” in SIGMOD, 2000.

[13] G. Graefe and P.-A. Larson, “B-tree indexes and CPU caches,” in ICDE, 2001.

[14] R. De La Briandais, “File searching using variable length keys,” in western joint computer conference, 1959.

[15] Hans-Peter leis, Armin Kemper, Thomas Grust, ART: An Adaptive Real-time Tree Index, 2008.

[16] D. R. Morrison, “PATRICIA-practical algorithm to retrieve information coded in alphanumeric,” J. ACM, vol. 15, no. 4, 1968.

[17] D. E. Knuth, The art of computer programming, volume 3: (2nd ed.) sorting and searching, 1998.

[18] N. Askitis and R. Sinha, “Engineering scalable, cache and space efficient tries for strings,” The VLDB Journal, vol. 19, no. 5, 2010.

[19] M. Bohm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, ¨ “Efficient in-memory indexing with generalized prefix trees,” in BTW, 2011.

1. **Приложения**

**Приложение 1.**

std::string generate\_random\_word(size\_t length)

{

std::random\_device rd;

std::mt19937 generator(rd());

std::uniform\_int\_distribution<> distribution(0, 25);

std::string word;

for (int i = 0; i < length; ++i)

{

word += static\_cast<char>(distribution(generator) + 'a');

}

return word;

}

**Приложение 2.**

const int dictionary\_size = 100000;

const int search\_keys\_count = 10000;

const int word\_length = 10;

int main()

{

std::vector<std::string> words;

for (int i = 0; i < dictionary\_size; ++i)

{

words.emplace\_back(generate\_random\_word(word\_length));

}

Trie trie;

std::set<std::string> set;

std::multiset<std::string> multi\_set;

std::unordered\_set<std::string> un\_set;

absl::btree\_set<std::string> btree;

auto start\_trie = std::chrono::high\_resolution\_clock::now();

for (const auto& word : words)

{

trie.insert(word);

}

auto end\_trie = std::chrono::high\_resolution\_clock::now();

auto duration\_trie = std::chrono::duration\_cast<std::chrono::microseconds>(end\_trie - start\_trie);

auto start\_set = std::chrono::high\_resolution\_clock::now();

for (const auto& word : words)

{

set.insert(word);

}

auto end\_set = std::chrono::high\_resolution\_clock::now();

auto duration\_set = std::chrono::duration\_cast<std::chrono::microseconds>(end\_set - start\_set);

auto start\_mset = std::chrono::high\_resolution\_clock::now();

for (const auto& word : words)

{

multi\_set.insert(word);

}

auto end\_mset = std::chrono::high\_resolution\_clock::now();

auto duration\_mset = std::chrono::duration\_cast<std::chrono::microseconds>(end\_mset - start\_mset);

auto start\_unset = std::chrono::high\_resolution\_clock::now();

for (const auto& word : words)

{

un\_set.insert(word);

}

auto end\_unset = std::chrono::high\_resolution\_clock::now();

auto duration\_unset = std::chrono::duration\_cast<std::chrono::microseconds>(end\_unset - start\_unset);

auto start\_btree = std::chrono::high\_resolution\_clock::now();

for (const auto& word : words)

{

btree.insert(word);

}

auto end\_btree = std::chrono::high\_resolution\_clock::now();

auto duration\_btree = std::chrono::duration\_cast<std::chrono::microseconds>(end\_btree - start\_btree);

std::cout << "Time in prefix " << duration\_trie.count() << std::endl;

std::cout << "Time in RB-balance-tree with not dublicate " << duration\_set.count() << std::endl;

std::cout << "Tine in RB-balance-tree " << duration\_mset.count() << std::endl;

std::cout << "Time in hash " << duration\_unset.count() << std::endl;

std::cout << "Time in B-tree " << duration\_btree.count() << std::endl;

return 0;

}

**Приложение 3.**

const int dictionary\_size = 100000;

const int search\_keys\_count = 10000;

const int word\_length = 10;

int main()

{

std::vector<std::string> words;

for (int i = 0; i < dictionary\_size; ++i)

{

words.emplace\_back(generate\_random\_word(word\_length));

}

Trie trie;

std::set<std::string> set;

std::multiset<std::string> multi\_set;

std::unordered\_set<std::string> un\_set;

absl::btree\_set<std::string> btree;

for (const auto& word : words)

{

trie.insert(word);

set.insert(word);

un\_set.insert(word);

multi\_set.insert(word);

btree.insert(word);

}

std::vector<std::string> search\_keys;

for (int i = 0; i < search\_keys\_count; ++i)

{

search\_keys.emplace\_back(generate\_random\_word(word\_length));

}

auto start\_trie = std::chrono::high\_resolution\_clock::now();

for (const auto& key : search\_keys)

{

trie.find(key);

}

auto end\_trie = std::chrono::high\_resolution\_clock::now();

auto duration\_trie = std::chrono::duration\_cast<std::chrono::microseconds>(end\_trie - start\_trie);

auto start\_set = std::chrono::high\_resolution\_clock::now();

for (const auto& key : search\_keys)

{

set.find(key);

}

auto end\_set = std::chrono::high\_resolution\_clock::now();

auto duration\_set = std::chrono::duration\_cast<std::chrono::microseconds>(end\_set - start\_set);

auto start\_un\_set = std::chrono::high\_resolution\_clock::now();

for (const auto& key : search\_keys)

{

un\_set.find(key);

}

auto end\_un\_set = std::chrono::high\_resolution\_clock::now();

auto duration\_un\_set = std::chrono::duration\_cast<std::chrono::microseconds>(end\_un\_set - start\_un\_set);

auto start\_mset = std::chrono::high\_resolution\_clock::now();

for (const auto& key : search\_keys)

{

trie.find(key);

}

auto end\_mset = std::chrono::high\_resolution\_clock::now();

auto duration\_mset = std::chrono::duration\_cast<std::chrono::microseconds>(end\_mset - start\_mset);

auto start\_btree = std::chrono::high\_resolution\_clock::now();

for (const auto& key : search\_keys)

{

trie.find(key);

}

auto end\_btree = std::chrono::high\_resolution\_clock::now();

auto duration\_btree = std::chrono::duration\_cast<std::chrono::microseconds>(end\_btree - start\_btree);

std::cout << "Time in prefix " << duration\_trie.count() << std::endl;

std::cout << "Time in RB-balance-tree with not dublicate " << duration\_set.count() << std::endl;

std::cout << "Tine in RB-balance-tree " << duration\_mset.count() << std::endl;

std::cout << "Time in hash " << duration\_un\_set.count() << std::endl;

std::cout << "Time in B-tree " << duration\_btree.count() << std::endl;

return 0;

}