Варианты представления таблиц классификации в SRAM-памяти (для поиска по полному соответствию)

Содержание

Введение

Таблицы классификации можно разделить на два типа. Первый тип таблиц используется для поиска по полному соответствию, второй - для поиска префикса наибольшей длины (longest prefix match). В данном разделе проводится обзор известных программных реализаций таблиц классификации первого типа. Реализация должна обеспечивать возможность быстрого поиска и быстрой модификации таблицы, причем время поиска, количество обращений к памяти при поиске и возможность разбиения памяти, использующейся для хранения таблицы классификации, на несколько независимых областей являются ключевыми критериями при сравнении различных реализаций.

Таблица классификации состоит из **записей** вида (***ключ***, ***значение***), причем все ключи таблицы должны быть попарно различными. И ключи, и значения имеют известный формат и представляются в виде чисел двоичной системы счисления фиксированной длины.

Реализация таблицы классификации должна поддерживать две операции: **поиск** и **модификацию**. При поиске задается ключ, называемый **искомым**. Алгоритм поиска должен определить, содержит ли таблица классификации запись с данным ключом, и в случае наличия такой записи вернуть значение этой записи, т.н. **результат поиска**. Поиск называется **успешным**, если искомый ключ содержится в таблице классификации, и **неуспешным** в противном случае. Ключ, содержащийся в таблице классификации, называется **присутствующим**, а не содержащийся - **отсутствующим**.

Операция модификации может быть двух типов: **добавление** или **удаление**. Для обеих операций задается **целевой ключ** и **целевое значение**. В случае операции добавления алгоритм модификации должен найти в таблице запись с целевым ключом и заменить ее значение на целевое, а в случае отсутствия в таблице такой записи добавить новую запись в таблицу, содержащую целевые ключ и значение. В случае операции удаления алгоритм должен в случае наличия в таблице классификации записи с целевым ключом удалить ее из таблицы.

Введем следующие обозначения:

- K - размер ключа в битах;
- V - размер значения в битах;
- N - максимальное количество хранимых в таблице классификации записей;
- P - размер указателя на ключ или значение в битах; данная величина равна $\lceil log_2(N+1) \rceil$;
- n - текущее количество хранимых в таблице классификации записей.

В данном разделе принимаются следующие допущения:

1) на множестве всевозможных ключей таблицы введена операция порядка;
2) мощность множества всевозможных ключей намного больше, чем максимальное число хранимых в таблице записей, поэтому возможность использования массивов с индексацией по ключам неэффективна по памяти и поэтому не рассматривается;
3) для оценки средних значений критериев (таких как среднее время поиска, среднее количество обращений к памяти и др.) используется предположение о том, что таблица построена из записей со случайно сгенерированными ключами, причем ключи представляют собой последовательность независимых равномерно распределенных случайных величин;
4) операция поиска/модификации называется эффективной, если среднее время ее выполнения не превосходит $O(log_2(n))$ либо $O(K)$;

5) не учитываются возможные проблемы, связанные с невыровненностью адресов при обращении к памяти, а также с невозможностью загрузки/сохранения нецелого числа байт;
6) для хранения указателей на объекты в памяти достаточно использовать $\lceil log_2(p+1) \rceil$ бит, где p - максимально возможное количество объектов данного типа;
7) при оценке размера памяти, количества обращений к памяти, размеров считываемой и обновляемой областей памяти могут не учитываться изменения, связанные с выделением/освобождением блоков памяти, а также блоки памяти постоянной длины, не зависящие от количества записей в таблице и использующиеся для хранения таблицы (например, указатель на корневой узел дерева, количество элементов в массиве и т.д.).

1. Структуры с неэффективным поиском

1.1. Неупорядоченный массив

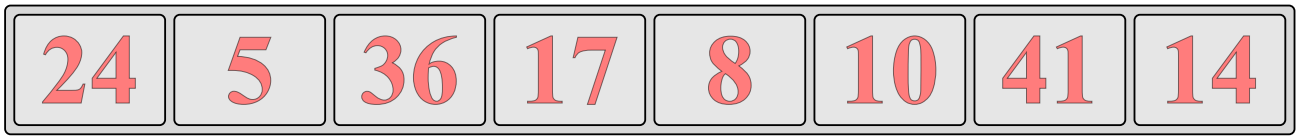

Рис. 1. Пример неупорядоченного массива; красным цветом представлены хранимые ключи; для компактности значения не представлены.

Описание.

Записи хранятся в неупорядоченном по ключам массиве (рис. 1).

Алгоритм поиска.

Последовательно просматриваются записи массива, начиная с нулевой. При каждом просмотре ключ записи сравнивается с искомым. Поиск завершается либо при обнаружении совпадающего с искомым ключа (успешный поиск, в таком случае значение, содержащееся в текущей записи, является результатом поиска), либо при достижении конца массива (неуспешный поиск).

Алгоритм модификации.

Проводится поиск записи с целевым ключом. Для операции удаления в случае успешного поиска найденная запись удаляется, а каждая из последующих записей в массиве сдвигается на одну позицию влево. Для операции добавления в случае успешного поиска значение найденной записи изменяется на целевое, а в случае неуспешного - новая запись добавляется в конец массива.

Размер памяти (в битах): $(K+V) \cdot N$.

Время поиска: $O(n)$ в среднем и в худшем.

Время модификации: $O(n)$ в среднем и в худшем.

Размер считываемой области памяти при поиске (в битах).

При успешном поиске - в среднем $\frac{1}{2}K \cdot (n+1)+V$, в худшем $K \cdot n+V$. При неуспешном поиске - $K \cdot n$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с отсутствующим ключом - $K+V$. При добавлении записи с присутствующим ключом - V. При удалении записи с присутствующим ключом - в среднем $\frac{1}{2}(K+V) \cdot (n+1)$, в худшем $(K+V) \cdot n$. При удалении записи с отсутствующим ключом - 0.

Возможность разбиения памяти: можно хранить ключи и значения в отдельных массивах.

1.2. Неупорядоченный связный список

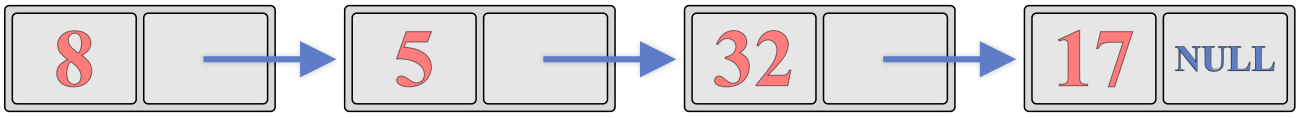

Рис. 2. Пример неупорядоченного связного списка; красным цветом представлены хранимые ключи, синим - указатели; для компактности значения в узлах не представлены.

Описание.

Записи хранятся в неупорядоченном связном списке, каждый узел которого содержит ключ, значение и указатель на следующий узел либо значение NULL, если этот узел - последний (рис. 2).

Алгоритм поиска.

Последовательно просматриваются узлы списка, начиная с корневого. При каждом просмотре ключ узла сравнивается с искомым. Поиск завершается либо при обнаружении совпадающего с искомым ключа (успешный поиск, в таком случае значение, содержащееся в текущем узле, является результатом поиска), либо при достижении конца списка (неуспешный поиск).

Алгоритм модификации.

Проводится поиск узла с целевым ключом. Для операции удаления в случае успешного поиска найденный узел удаляется, а указателю предшествующего узла присваивается ссылка на следующий за удаляемым узел. Для операции добавления в случае успешного поиска значение найденного узла изменяется на целевое, а в случае неуспешного - новый узел добавляется в конец списка.

Размер памяти (в битах): $(K + V + P) \cdot N$.

Время поиска: $O(n)$ в среднем и в худшем.

Время модификации: $O(n)$ в среднем и в худшем.

Размер считываемой области памяти при поиске (в битах).

При успешном поиске - в среднем $\frac{1}{2}(K + P) \cdot (n + 1) + V$, в худшем $(K + P) \cdot n + V$. При неуспешном поиске - $(K + P) \cdot n$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с отсутствующим ключом - $K + P + V$. При добавлении записи с присутствующим ключом - V. При удалении записи с присутствующим ключом - P. При удалении записи с отсутствующим ключом - 0.

Возможность разбиения памяти.

Значения можно хранить в отдельной области памяти, для этого узлы списка должны содержать только ключ и указатель на следующий узел, а указатель на сопоставленное ключу значение должен соответствовать указателю на узел, хранящий данный ключ, поэтому дополнительная память при таком разбиении не требуется.

Резюме.

По сравнению с неупорядоченным массивом данная реализация требует дополнительной памяти для хранения указателей, однако операция модификация не приводит к обновлению значительной части содержимого памяти.

2. Структуры с эффективным поиском, но неэффективной модификацией

2.1. Упорядоченный массив

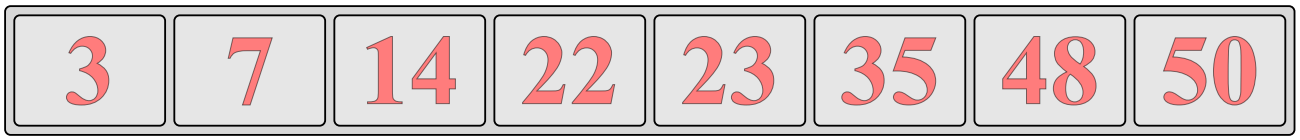

Рис. 3. Пример упорядоченного массива; красным цветом представлены хранимые ключи; для компактности значения не представлены.

Описание.

Записи хранятся в упорядоченном по ключам массиве (рис. 3).

Алгоритм поиска.

Используется алгоритм бинарного поиска, работающий с подмассивом исходного массива. В начале алгоритма подмассив инициализируется всем массивом. На каждой итерации алгоритма сначала проверяется, содержится ли искомый ключ в промежутке, задаваемом ключами двух крайних записей подмассива. Невыполнение этого условия означает отсутствие искомого ключа в таблице. В противном случае подмассив делится на 2 равные части (число записей в полученных частях должно отличаться максимум на один), после чего в результате сравнения искомого ключа с ключом центральной записи подмассива определяется часть, в которой потенциально может находиться запись с искомым ключом. В конце итерации подмассив заменяется на найденную часть. Алгоритм работает до тех пор, пока выполняются проверки, а количество записей в подмассиве больше одного. Когда в подмассиве остается одна запись, проверяется ее ключ на равенство искомому. В случае совпадения поиск успешен, а результатом поиска является значение, содержащееся в этой записи, в противном случае поиск неуспешен.

Алгоритм модификации.

Проводится поиск записи с целевым ключом. Для операции удаления в случае успешного поиска найденная запись удаляется, а каждая из последующих записей в массиве сдвигается на одну позицию влево. Для операции добавления в случае успешного поиска значение найденной записи изменяется на целевое. Для операции добавления в случае неуспешного поиска сначала определяется позиция для вставки новой записи на основе результата поиска, после чего все записи массива, начиная с найденной позиции, сдвигаются на одну позицию вправо, а на освобожденную позицию вставляется новая запись.

Размер памяти (в битах): $(K + V) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем и в худшем.

Время модификации: $O(n)$ в среднем и в худшем.

Количество обращений к памяти при поиске: $\lceil log_2(n-1) \rceil + 3$ в худшем при $n \geq 2$ (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, V)$ бит).

Размер считываемой области памяти при поиске (в битах).

При $n \geq 2$ при успешном поиске - в худшем $K \cdot (\lceil log_2(n-1) \rceil + 2) + V$, при неуспешном поиске - в худшем $K \cdot (\lceil log_2(n-1) \rceil + 2)$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - в среднем $\frac{1}{2}(K+V)\cdot n+1$, в худшем $(K+V)\cdot(n+1)$. При удалении записи с присутствующим ключом - в среднем $\frac{1}{2}(K+V)\cdot(n+1)$, в худшем $(K+V)\cdot n$. При удалении записи с отсутствующим ключом - 0.

Возможность разбиения памяти: можно хранить ключи и значения в отдельных массивах.

Резюме.

По сравнению с неупорядоченным массивом и неупорядоченным связным списком данная реализация позволяет проводить быстрый поиск. Недостаток - необходимость обновления значительной части содержимого памяти при модификации, меняющей множество хранимых ключей, в среднем и в худшем.

2.2. Многоуровневый упорядоченный массив

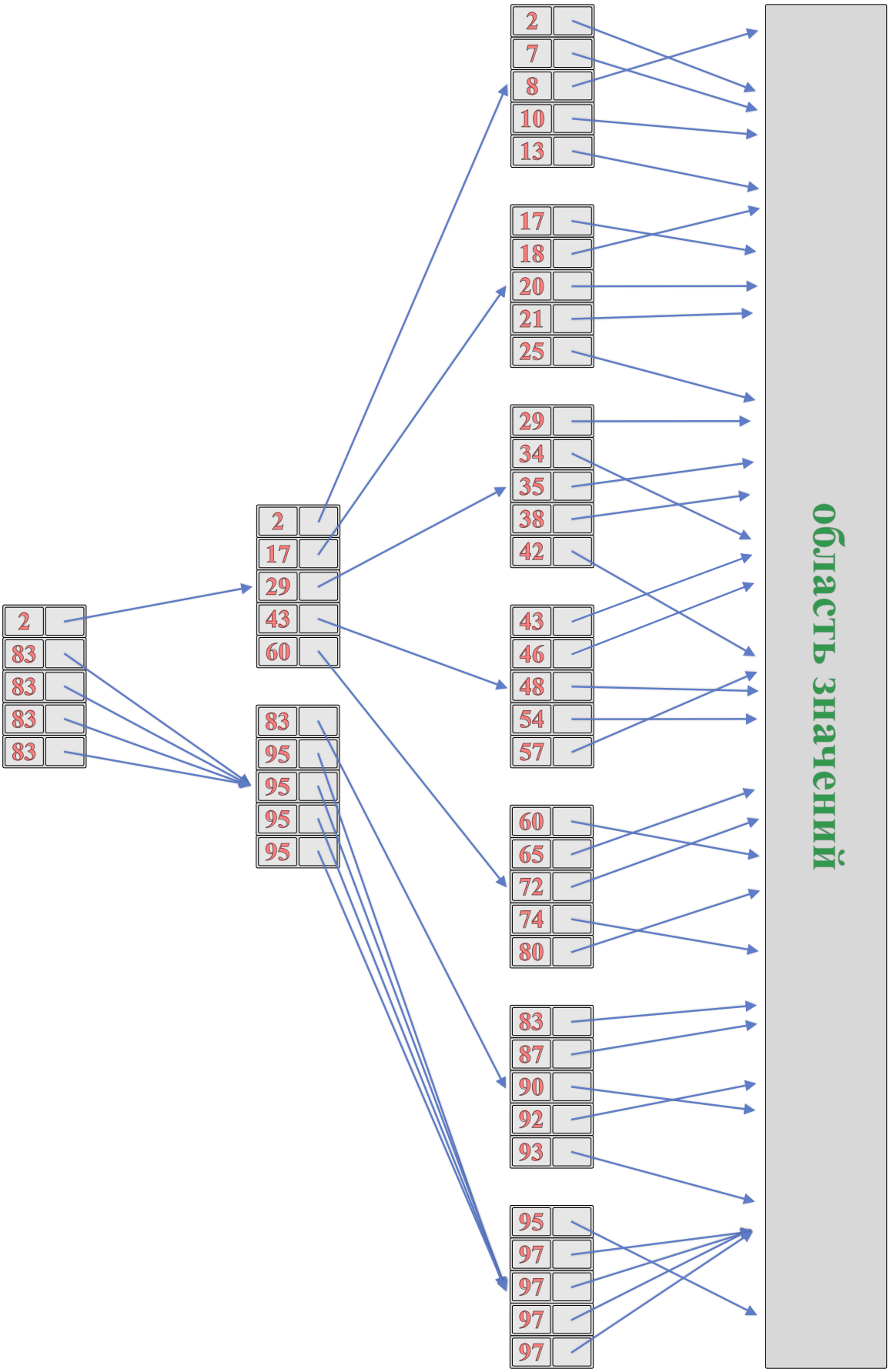

Рис. 4. Пример многоуровневого упорядоченного массива с $M = 5$; красным цветом представлены хранимые ключи, синим - указатели.

Описание.

Данная структура предназначена для случаев, когда при чтении из памяти загрузка данных происходит крупными блоками фиксированного размера, а само обращение к памяти является критической с точки зрения времени выполнения операцией. Многоуровневый упорядоченный массив позволяет минимизировать количество обращений к памяти.

Память разбивается на две области: для хранения ключей и значений. Область ключей состоит из блоков фиксированного размера. Каждый блок является массивом, состоящим из узлов вида (*ключ*, *указатель*). Количество узлов в массиве, одинаковое для всех блоков, подбирается таким образом, чтобы блок мог быть считан за одно обращение к памяти. Пусть это количество равно $M \geq 2$.

Опишем формат хранения таблицы классификации в виде правила построения блоков. На первом шаге записи таблицы сортируются по возрастанию ключей, после чего разбиваются на группы по M записей (в последней группе может быть меньше M записей) с сохранением построенного порядка. Таким образом, запись с наименьшим ключом является первой записью первой группы. Для каждой пары (*ключ*, *значение*) значение располагается в произвольном незанятом месте памяти в области значений, после чего формируется узел, состоящий из ключа и указателя на место хранения соответствующего ему значения. После этого для каждой группы записей формируется массив в памяти, состоящий из узлов, соответствующих записям данной группы, с сохранением порядка узлов относительно исходного порядка записей. Если количество записей в последней группе меньше M, последний узел дублируется в свободные правые ячейки массива. Полученные массивы являются листовыми блоками и сохраняются в произвольном свободном месте памяти в области ключей.

На втором шаге рассматриваются первые узлы из каждого блока листового уровня. Для каждого такого узла v формируется узел, состоящий из ключа узла v и указателя на соответствующий узлу v блок в памяти. Полученные узлы разбиваются на группы по M узлов с сохранением порядка соответствующих узлам листовых блоков. Каждая сформированная группа объединяется в массив длины M, для последней группы ее последний узел дублируется в свободные правые ячейки массива. Полученные массивы являются предлистовыми блоками и сохраняются в произвольном свободном месте памяти в области ключей.

Продолжая аналогичные шаги, в итоге на некотором шаге формируется только один блок. Данный блок является корневым и также сохраняется в память. Высота дерева, формируемого связями между построенными блоками (с учетом указателей на значения), равна $\lceil log_M(n) \rceil$. Таким образом, для поиска по ключу необходимо не более $\lceil log_M(n) \rceil + 1$ обращений к памяти. На рис. 4 приводится пример данной структуры.

Алгоритм поиска.

Проводится спуск по соответствующему построенным связям дереву от корня к листьям. На каждом шаге проводится одно обращение к памяти, в результате которого считывается один блок (на первом шаге считывается корневой блок). Затем, если считываемый блок не является финальным, то среди хранящихся в блоке пар вида (*ключ*, *указатель*) находится наибольшая по порядку пара, для которой *ключ* $\leq$ *искомый ключ*. Если такой пары не существует (т.е. на данном шаге рассматривается корневой блок и для него ключ первого по порядку узла больше искомого ключа), то поиск завершается неуспехом. В противном случае происходит переход к следующему шагу с сохранением найденного указателя для считывания блока на следующем шаге.

Если же считываемый блок является финальным, то среди хранящихся в блоке пар ищется пара с ключом, равным искомому. Если такой пары не существует, то поиск завершается неуспехом. В противном случае происходит финальная загрузка значения по указателю в найденной паре.

Алгоритм модификации.

Любая модификация, меняющая множество хранимых ключей, предполагает практически полную перестройку хранящихся в памяти блоков, поэтому алгоритм модификации состоит в поиске по искомому ключу для выявления необходимости перестройки и последующей перестройки в случае необходимости. Перестройка может быть выполнена за время $O(n)$.

Размер памяти (в битах): верхняя оценка - $(K+P)\cdot\left(\frac{M}{M-1}\cdot N + log_M(N)+1\right)+V\cdot N$, приблизительная оценка при достаточно больших N - $\left((K+P)\cdot\frac{M}{M-1}+V\right)\cdot N$.

Время поиска: $O(log_2(n))$ в среднем и в худшем при использовании бинарного поиска на каждом шаге алгоритма поиска.

Время модификации: $O(n)$ в среднем и в худшем.

Количество обращений к памяти при поиске: $\lceil log_M(n)\rceil+1$ в худшем (при условии, что каждый блок может быть считан за одно обращение).

Размер считываемой области памяти при поиске (в битах).

При успешном поиске - в худшем $(K+P)\cdot M\cdot(\lceil log_M(n)+1\rceil)+V$, при неуспешном поиске - в худшем $(K+P)\cdot M\cdot(\lceil log_M(n)+1\rceil)$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При удалении записи с отсутствующим ключом - 0. В остальных случаях - в худшем $(K+P)\cdot\left(\frac{M}{M-1}\cdot n + log_M(n)+1\right)+V\cdot n$ (практически полная перестройка).

Возможность разбиения памяти.

Каждый уровень соответствующего построенным связям дерева и область значений можно хранить в отдельных областях памяти. Две наибольшие области следующие: область листовых блоков ($(K+P)\cdot\left\lceil\frac{N}{M}\right\rceil\cdot M$ бит) и область значений ($V\cdot N$ бит). Корневой блок можно хранить в быстрой памяти, в этом случае число обращений к медленной памяти при поиске сокращается на один.

Резюме.

Даная реализация при достаточно большом размере блока, который может быть считан из памяти за одно обращение, позволяет организовать поиск по таблицам классификации, содержащим большое количество записей, при котором результат возвращается всего за несколько обращений к памяти. Эту реализацию имеет смысл использовать только в том случае, если обращение к памяти является критической операцией с точки зрения времени выполнения, в противном случае эффективнее использовать упорядоченный массив, занимающий меньше памяти. Однако и упорядоченный массив, и многоуровневый упорядоченный массив имеют существенный недостаток - отсутствие поддержки эффективной модификации. Далее в разделе рассматриваются структуры, не обладающие этим недостатком. Аналогом многоуровневого упорядоченного массива, минимизирующим количество обращений к памяти и поддерживающим эффективную модификацию, является B-дерево.

3. Структуры с эффективным поиском и эффективной модификацией

3.1. Деревья поиска

В данном параграфе рассматриваются структуры данных на основе деревьев поиска. При использовании таких структур каждой записи таблицы классификации сопоставляется некоторый узел дерева. Одному и тому же узлу в общем случае может быть сопоставлено несколько пар. Быстрота выполнения операции поиска достигается за счет того, что для нахождения узла с заданным ключом достаточно один раз пройти от корня дерева вниз, при каждом разветвлении однозначно определяя нужную ветвь исходя из сопоставленных текущему узлу ключей. Это обеспечивает время выполнения, пропорциональное длине пройденного пути. Деревья поиска разделяются на два основных класса: несбалансированные и сбалансированные. Для сбалансированных деревьев гарантируется логарифмическая (относительно общего количества узлов) высота, что обеспечивает в наихудшем случае логарифмическое время поиска. Для несбалансированных деревьев в наихудшем случае поиск выполняется за линейное время, однако обычно это время также логарифмическое.

3.1.1. Несбалансированное дерево бинарного поиска

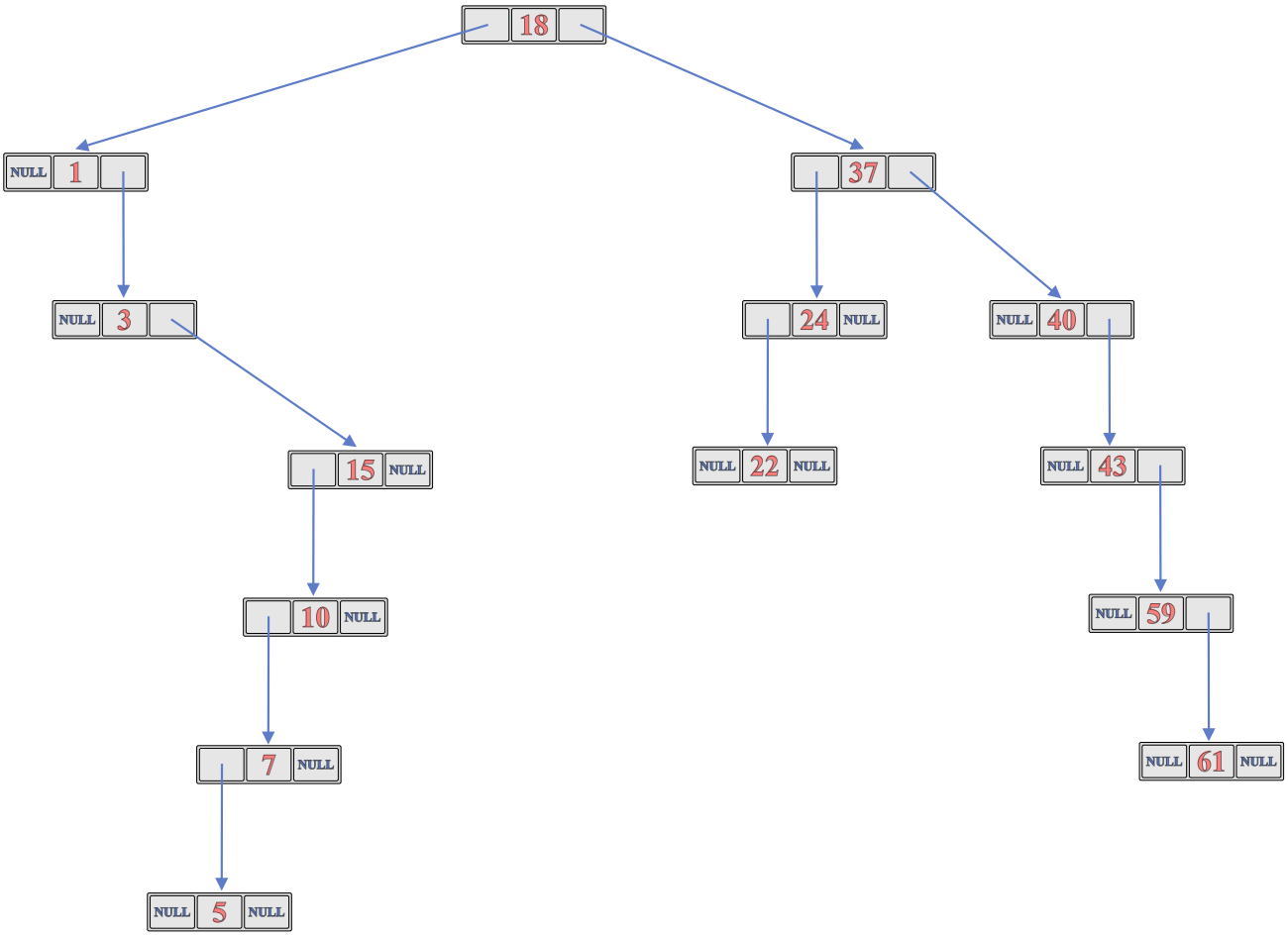

Рис. 5. Пример дерева бинарного поиска; красным цветом представлены хранимые ключи, синим - указатели; для компактности значения в узлах не представлены.

Описание.

Каждой хранящейся в таблице паре (*ключ*, *значение*) ставится в соответствие узел, содержащий данную пару и два указателя: на левый дочерний узел и на правый дочерний узел. В частном случае указатели могут быть равны NULL. Все узлы объединяются в корневое дерево, в котором ключ в любом узле больше ключей левого поддерева данного узла и меньше ключей правого поддерева данного узла (рис. 5).

В несбалансированном дереве бинарного поиска сложность операций поиска и модификации, а также размер обновляемой памяти при модификации зависят от порядка добавления записей в дерево при его формировании. Порядок добавления записей влияет на сбалансированность и высоту дерева: при неудачном порядке в наихудшем случае дерево будет иметь вид упорядоченного по ключам связного списка, для которого сложность операций поиска и модификации равна $O(n)$ в среднем и в худшем.

Алгоритм поиска.

Проводится спуск по дереву от корневого узла к листьям. На каждом шаге рассматривается очередной узел и проводится сравнение искомого ключа с ключом узла. Если ключи совпадают, то поиск успешен и хранящееся в текущем узле значение является результатом поиска. Если искомый ключ меньше (больше) ключа узла, то происходит переход к следующему шагу с рассмотрением левого (правого) дочернего узла в качестве текущего. Если на каком-то шаге ключи не совпадают, а нужного дочернего узла не существует (соответствующий указатель равен NULL), то дерево поиска не содержит искомый ключ (поиск неуспешен).

Алгоритм модификации.

Проводится поиск по целевому ключу. Для операции добавления в случае успешного поиска значение найденного узла, содержащего целевой ключ, изменяется на целевое. Для операции добавления в случае неуспешного поиска для целевого ключа и целевого значения создается новый узел дерева, указатель на который затем добавляется в последний просмотренный при поиске узел (таким образом, новый узел становится левым или правым дочерним узлом для данного узла, в зависимости от результата сравнения целевого ключа с ключом данного узла).

Для операции удаления в случае успешного поиска необходимо предварительно выполнить цепочку операций ротации (вращения) для поддерева, корень которого содержит удаляемый ключ, сложность данной цепочки операций равна $O(h_s)$, где h_s - высота вращаемого поддерева.

Высота дерева: $O(log_2(n))$ в среднем, $n-1$ в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K+V+2P$.

Размер памяти для хранения всей структуры (в битах): $(K+V+2P) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем, $O(n)$ в худшем.

Время модификации: $O(log_2(n))$ в среднем, $O(n)$ в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем, $2n$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K,P,V)$ бит).

Размер считываемой области памяти при поиске (в битах): $(K+P) \cdot O(log_2(n))$ в среднем, $(K+P) \cdot (n-1) + K + V$ в худшем.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - $K+V+3P$. При удалении записи с отсутствующим ключом - 0. При удалении записи

с присутствующим ключом - в худшем $P \cdot (2n - 3)$ (в стандартной реализации).

Возможность разбиения памяти.

Значения можно хранить в отдельной области памяти, для этого узлы дерева должны содержать только ключ и два указателя на дочерние узлы, а указатель на сопоставленное ключу значение должен соответствовать указателю на узел, хранящий данный ключ, поэтому дополнительная память при таком разбиении не требуется.

Резюме.

Данная реализация в среднем обеспечивает эффективный поиск и модификацию, однако в наихудшем случае (например, при добавлении записей в таблицу классификации в порядке возрастания либо убывания ключей) сложность данных операций может возрасти до $O(n)$. Если же порядок добавления записей случайный с равномерным распределением, то средняя сложность данных операций равна $O(log_2(n))$, причем, согласно закону больших чисел, вероятность существенного отклонения от среднего значения падает с ростом n.

В следующем пункте описывается модифицированная структура, в которой вне зависимости от порядка добавления/удаления записей демонстрируются такие же вероятностные свойства, как и в случае случайного равномерно распределенного добавления записей.

3.1.2. Рандомизированное дерево бинарного поиска

Описание.

Данная структура полностью идентична несбалансированному дереву бинарного поиска. Разница заключается лишь в алгоритме добавления и удаления записей, в ходе которых применяются случайные решения о месте вставки очередного узла (в случае добавления) и узле, замещаемом удаляемый (в случае удаления). Данный случайный фактор защищает степень сбалансированности и высоту дерева от порядка добавления/удаления записей. При применении данных модификаций создание дерева с помощью произвольной неслучайной последовательности операций вставки и удаления эквивалентно построению дерева из случайной перестановки записей.

Алгоритм поиска полностью совпадает с алгоритмом для несбалансированного дерева бинарного поиска.

Алгоритм модификации.

В случае операции добавления при спуске вниз по дереву на каждом шаге принимается случайное решение, на основании которого определяется, произвести ли вставку в корень текущего поддерева или продолжить спуск по дереву. В случае операции удаления принимается случайное решение, на основании которого выбирается узел, замещающий удаляемый.

Высота дерева: $O(log_2(n))$ в среднем, $n - 1$ в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K + V + 2P$.

Размер памяти (в битах): $(K + V + 2P) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем, $O(n)$ в худшем.

Время модификации: $O(log_2(n))$ в среднем, $O(n)$ в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем, $2n$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, P, V)$ бит).

Размер считываемой области памяти при поиске (в битах): $(K + P) \cdot O(log_2(n))$ в среднем, $(K + P) \cdot (n - 1) + K + V$ в худшем.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - в худшем $K + V + P \cdot (2n + 3)$ (в стандартной реализации). При удалении записи с отсутствующим ключом - 0. При удалении записи с присутствующим ключом - в худшем $P \cdot (2n - 3)$ (в стандартной реализации).

Возможность разбиения памяти: значения можно хранить в отдельной области памяти.

Резюме.

Все приведенные оценки для данной реализации совпадают с оценками для несбалансированного дерева бинарного поиска, однако стоит отметить следующие особенности рандомизированного дерева:

- для рандомизированного дерева средняя сложность операции модификации выше за счет случайности и связанной с ней дополнительной логикой;
- для рандомизированного дерева указанные средние оценки применимы вне зависимости от порядка добавления/удаления записей, при этом порядок добавления/удаления записей не обязан быть случайным;
- как правило, при достаточно большом n отношение высоты рандомизированного дерева к высоте полностью сбалансированного дерева, состоящего из тех же самых записей, отличается на небольшую константу, вне зависимости от порядка добавления/удаления записей; из-за этого свойства рандомизированные деревья часто называют сбалансированными.

3.1.3. АВЛ-дерево

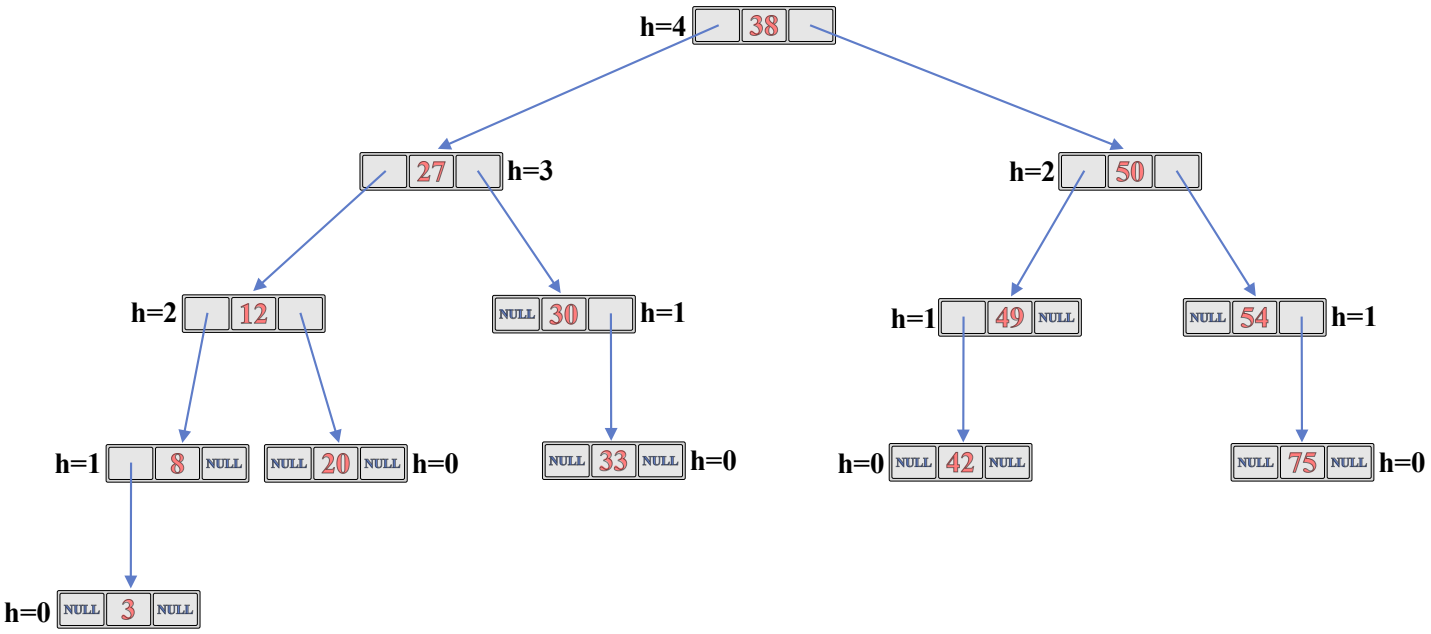

Рис. 6. Пример АВЛ-дерева; красным цветом представлены хранимые ключи, синим - указатели, h - высота соответствующего узлу поддерева; для компактности значения в узлах не представлены.

Описание.

АВЛ-дерево - сбалансированное дерево бинарного поиска, для каждого узла которого высоты двух его поддеревьев отличаются не более чем на один (рис. 6). При этом если узел не содержит поддерево, высота данного поддерева полагается равной -1. Каждый узел АВЛ-дерева дополнительно хранит показатель разбалансировки (для его хранения используются 2 бита), который равен разности высот правого и левого поддеревьев данного узла. Данная структура из-за хорошей сбалансированности позволяет быстро проводить поиск в худшем случае. Высота АВЛ-дерева не превосходит $\lfloor 1.45 \cdot log_2(n+2) \rfloor$.

Алгоритм поиска полностью совпадает с алгоритмом для несбалансированного дерева бинарного поиска.

Алгоритм модификации.

Проводится поиск по целевому ключу. Для операции добавления в случае успешного поиска значение найденного узла, содержащего целевой ключ, изменяется на целевое. Для операции добавления в случае неуспешного поиска новый узел, содержащий целевые ключ и значение, добавляется в дерево в качестве листа с показателем разбалансировки 0. Затем осуществляется проход по дереву от добавленного узла к корню с пересчетом показателя балансировки узлов и выполнением процедуры балансировки в случае, если данный показатель для текущего узла равен 2 или -2. Если же показатель балансировки текущего узла остается неизменным, проход завершается. Процедура балансировки узла заключается в выполнении одного из четырех возможных видов локальных ротаций.

Операция удаления реализуется чуть сложнее и для краткости не описывается.

Высота дерева: $\lfloor 1.45 \cdot log_2(n+2) \rfloor$ в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K + V + 2P + 2$.

Размер памяти для хранения всей структуры (в битах): $(K + V + 2P + 2) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем и в худшем.

Время модификации: $O(log_2(n))$ в среднем и в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем и в худшем, верхняя оценка - $2.9 \cdot log_2(n+2)$ (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, P, V)$ бит).

Размер считываемой области памяти при поиске (в битах): $O(log_2(n))$ в среднем и в худшем, верхняя оценка - $(K + P) \cdot \lfloor 1.45 \cdot log_2(n+2) \rfloor + V$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При удалении записи с отсутствующим ключом - 0. Для остальных случаев $P \cdot O(log_2(n))$ в худшем.

Возможность разбиения памяти: значения можно хранить в отдельной области памяти.

Резюме.

Данная реализация в худшем обеспечивает эффективный поиск и модификацию.

3.1.4. Красно-черное дерево

Описание.

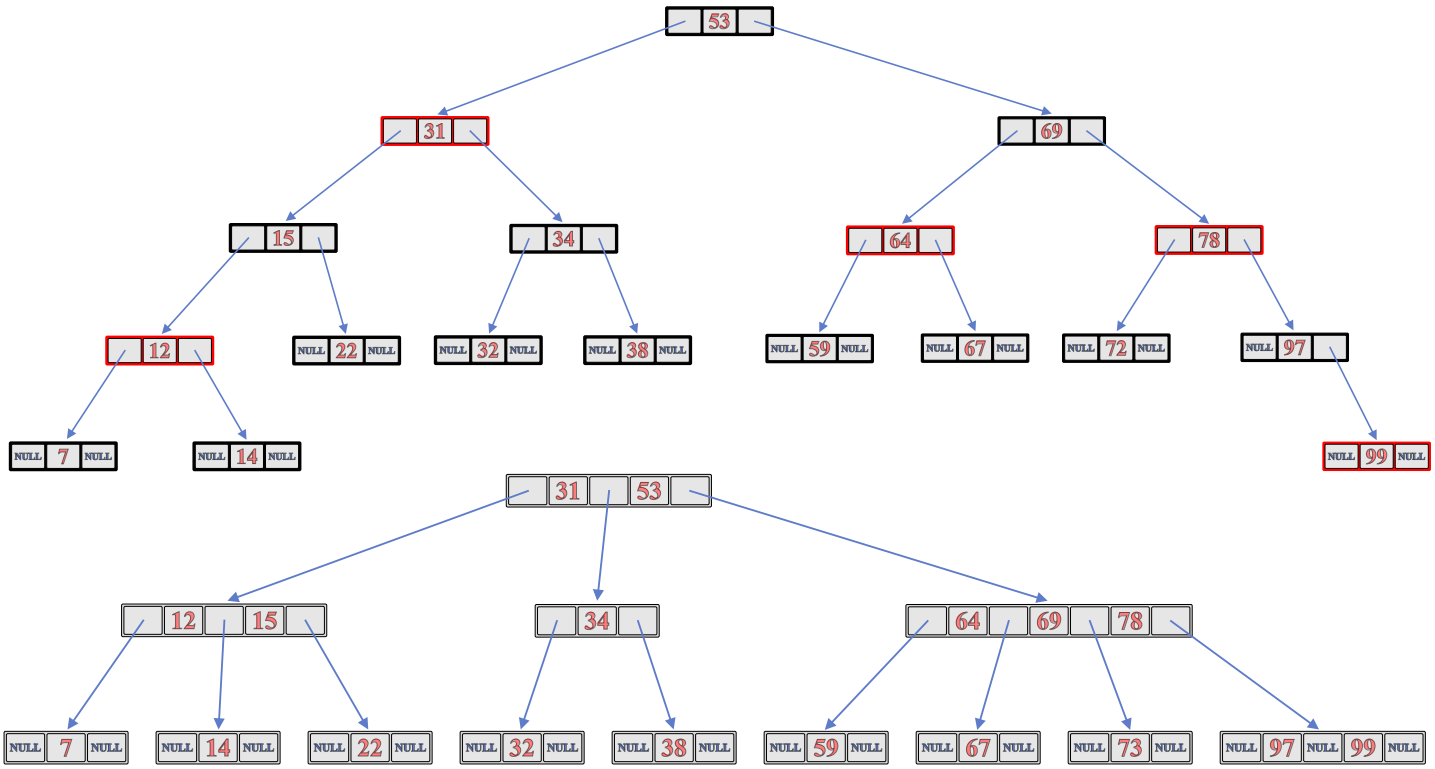

Рис. 7. Пример красно-черного дерева (сверху) и соответствующего ему сбалансированного 2-3-4-дерева (снизу); красным цветом представлены хранимые ключи, синим - указатели; для компактности значения в узлах не представлены.

Красно-черное дерево является деревом бинарного поиска, в котором каждому узлу присваивается цвет (красный или черный) и выполняются следующие свойства:

- корень дерева - черный;
- каждый нелистовой узел содержит два дочерних узла;
- дочерним узлом красного узла не может быть красный узел;
- любой пусть от корня до листового узла дерева содержит одинаковое число черных узлов.

Благодаря выполнению данных свойств красно-черное дерево является сбалансированным с высотой, не превышающей $2\,log_2(n)$, что позволяет проводить операции поиска и модификации за время $O(log_2(n))$ в худшем случае.

Красно-черному дереву можно поставить в соответствие сбалансированное 2-3-4-дерево. Сбалансированное 2-3-4-дерево - это дерево поиска, удовлетворяющее следующим свойствам:

- узлы дерева делятся на три типа: 2-узлы - с одним ключом и двумя указателями, 3-узлы - с двумя ключами и тремя указателями, 4-узлы - с тремя ключами и четырьмя указателями; упорядоченные по возрастанию ключи k-узла делят множество всевозможных ключей на k промежутков, при этом k-й указатель узла должен либо быть равен NULL, либо указывать на корень поддерева, содержащего только ключи из k-го промежутка; также каждый узел содержит по одному значению для каждого ключа;
- все указатели со значением NULL расположены на одинаковом расстоянии от корня.

Каждый черный узел красно-черного дерева может содержать от 0 до 2 дочерних красных узлов. В таком случае каждой группе в красно-черном дереве, состоящей из одного черного узла и p дочерних красных узлов можно сопоставить (p+2)-узел, содержащий упорядоченные по возрастанию

ключи исходной группы и соответствующие им значения. Далее полученные k-узлы объединяются в сбалансированное 2-3-4-дерево с помощью указателей. На рис. 7 показан пример красно-черного дерева вместе с соответствующим ему сбалансированным 2-3-4-деревом.

Представление красно-черного дерева используется для хранения в памяти, в то время как вид сбалансированного 2-3-4-дерева удобен для описания алгоритма модификации.

Алгоритм поиска по красно-черному дереву полностью совпадает с алгоритмом для несбалансированного дерева бинарного поиска.

Алгоритм модификации.

Опишем один из возможных алгоритмов добавления новой записи в соответствующее красно-черному дереву 2-3-4-дерево. Проводится поиск по дереву целевого ключа, при котором осуществляется проход по дереву от корня к листьям. На каждом шаге проводится сравнение ключей текущего узла с целевым ключом, по результатам которого либо проводится модификация значения (если узел содержит ключ, равный целевому), либо определяется один из дочерних узлов для обработки на следующем шаге (если текущий узел не является листовым), либо в узел вставляется целевая запись, тем самым превращая листовой k-узел в (k+1)-узел (если текущий узел является листовым).

Чтобы после вставки в листовой узел целевой записи он остался k-узлом при $k \leq 4$, при спуске вниз по дереву дополнительно проводятся следующие действия:

- если текущий узел - 4-узел, а его родитель - 2-узел, то данная пара узлов преобразуется в 3-узел с двумя дочерними 2-узлами;
- если текущий узел - 4-узел, а его родитель - 3-узел, то данная пара узлов преобразуется в 4-узел с двумя дочерними 2-узлами;
- если текущий узел - корневой 4-узел, то он преобразуется в три 2-узла.

Данный алгоритм приводится для простоты описания, в реальности работа происходит с красно-черным деревом, а вместо описанных выше дополнительных действий по разбиению 4-узлов проводятся ротации (повороты) красно-черного дерева при обработке черных узлов, содержащих двух потомков красного цвета (такие черные узлы соответствуют 4-узлам в 2-3-4-дереве). Эти повороты позволяют сохранять свойства красно-черных деревьев при добавлении новых узлов, тем самым поддерживается их сбалансированность.

Стоит отметить, что разбиение 4-узла в 2-3-4-дереве является нечастой ситуацией, в среднем при добавлении новой записи требуется менее одного такого разбиения.

Операция удаления реализуется чуть сложнее и для краткости не описывается.

Высота дерева: $2\,log_2(n)$ в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K + V + 2P + 1$.

Размер памяти для хранения всей структуры (в битах): $(K + V + 2P + 1) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем и в худшем.

Время модификации: $O(log_2(n))$ в среднем и в худшем.

Количество обращений к памяти при поиске: верхняя оценка - $4\,log_2(n) + 2$ (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, P, V)$ бит).

Размер считываемой области памяти при поиске (в битах): верхняя оценка - $2 \cdot (K + P) \cdot (log_2(n) + 1) + V$.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При удалении записи с отсутствующим ключом - 0. Для остальных случаев $P \cdot O(log_2(n))$ в худшем.

Возможность разбиения памяти.

Значения можно хранить в отдельной области памяти. Также, в отличие от других рассмотренных видов деревьев поиска, красно-черные деревья предоставляют возможность выделения для каждого уровня соответствующего сбалансированного 2-3-4-дерева собственной независимой области памяти, в которой хранятся узлы красно-черного дерева, соответствующие данному уровню, с сохранением эффективности операции модификации и без необходимости многократного увеличения суммарного размера используемой памяти. Это возможно благодаря двум причинам. Во-первых, количество уровней в соответствующем 2-3-4-дереве, а значит и количество областей памяти, не превосходит $\lceil log_2(N+1) \rceil$. Во-вторых, при выполнении каждой операции добавления/удаления узла в 2-3-4-дереве число узлов, уровень которых изменяется, не превосходит $O(log_2(n))$. Таким образом, операция модификации не приводит к существенному перестроению таблицы классификации.

Для реализации описанного разбиения памяти на независимые области размер каждой такой области должен быть достаточным для хранения максимально возможного количества узлов в соответствующем уровне, что приводит к увеличению суммарного объема памяти, необходимого для хранения таблицы классификации. Вопрос оценки необходимого размера памяти, выделяемого для каждого уровня, и возможных улучшений описанной структуры для более эффективной поддержки разбиения памяти на независимые области требует дополнительного анализа.

Резюме.

Данная реализация в худшем обеспечивает эффективный поиск и модификацию. Преимущество красно-черного дерева перед АВЛ-деревом заключается в более быстрых операциях модификации (как правило) и в меньшем размере памяти за счет дополнительного хранения одного бита (для цвета) на каждый узел дерева вместо двух бит в случае АВЛ-дерева (для показателя разбалансировки). Недостаток - большая высота дерева, а значит более медленный поиск. При одном и том же количестве листовых узлов высота красно-черного дерева превосходит высоту АВЛ-дерева не более, чем на 39%.

Дополнительным преимуществом красно-черного дерева перед другими рассмотренными деревьями поиска является возможность эффективного хранения узлов дерева в нескольких независимых областях памяти в соответствии с их уровнем в соответствующем 2-3-4-дереве, что, однако, приводит к увеличению суммарного размера памяти, необходимого для хранения таблицы.

3.1.5. B-дерево

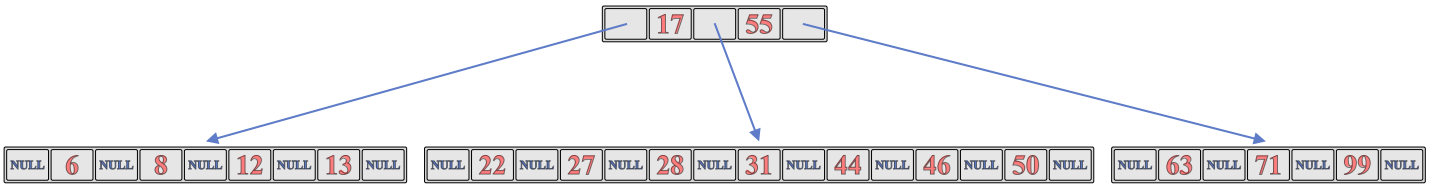

Рис. 8. Пример B-дерева для $M = 8$; красным цветом представлены хранимые ключи, синим - указатели; для компактности значения в узлах не представлены.

Описание.

Данная структура предназначена для случаев, когда при чтении из памяти загрузка данных происходит крупными блоками фиксированного размера, а само обращение к памяти является критической с точки зрения времени выполнения операцией. B-дерево позволяет минимизировать количество обращений к памяти.

Назовем k-узлом ($k \geq 2$) узел, состоящий из $k-1$ ключей, $k-1$ значений и k указателей на другие узлы; упорядоченные по возрастанию ключи k-узла делят множество всевозможных ключей на k промежутков, при этом k-й указатель узла должен либо быть равен NULL, либо указывать на корень поддерева, содержащего только ключи из k-го промежутка.

B-дерево порядка M (M четное и не меньше четырех) - сбалансированное дерево поиска, каждый узел которого является k-узлом, причем для каждого узла $k \in [2, M]$, для некорневых узлов $k \geq M/2$, а нелистовые узлы не содержат NULL в указателях (рис. 8). Предполагается, что величина M выбрана таким образом, чтобы M-узел мог быть считан за одно обращение к памяти. B-дерево является обобщением описанного в предыдущем пункте сбалансированного 2-3-4-дерева. Высота B-дерева не превосходит $log_{M/2}(n)$.

Существуют различные способы представления k-узлов дерева в памяти. Одним из таких способов является их представление в виде массива длины не более M, упорядоченного по возрастанию ключей. В таком случае для каждого узла также хранится длина массива ($\lceil log_2(M+1) \rceil$ бит).

Алгоритм поиска.

Проводится спуск по дереву от корневого узла к листьям. На каждом шаге за одно обращение к памяти загружается очередной узел (начиная с корневого) и проводится сравнение искомого ключа с ключами загруженного узла. Если среди ключей узла имеется совпадающий с искомым, то поиск успешен и соответствующее данному ключу значение является результатом поиска. В противном случае среди k промежутков, на которые делят множество всевозможных ключей ключи текущего k-узла, выбирается промежуток, которому принадлежит искомый ключ; если соответствующий данному промежутку указатель равен NULL, то поиск неуспешен, в противном случае происходит переход к следующему шагу, на котором найденный указатель используется для загрузки очередного узла.

Алгоритм модификации.

Один из возможных алгоритмов добавления новой записи обобщает алгоритм для 2-3-4-дерева, описанный в предыдущем пункте. Проводится поиск по дереву целевого ключа, при котором осуществляется проход по дереву от корня к листьям. На каждом шаге проводится сравнение ключей текущего узла с целевым ключом, по результатам которого либо проводится модификация значения (если узел содержит ключ, равный целевому), либо определяется один из дочерних узлов для обработки на следующем шаге (если текущий узел не является листовым), либо в узел вставляется целевая запись, тем самым превращая листовой k-узел в (k+1)-узел (если текущий узел является листовым).

Чтобы после вставки в листовой узел целевой записи он остался k-узлом при $k \leq M$, при спуске вниз по дереву дополнительно проводятся следующие действия:

- если текущий узел - M-узел, а его родитель - k-узел, то данная пара узлов преобразуется в (k+1)-узел с двумя дочерними (M/2)-узлами;
- если текущий узел - корневой M-узел, то он преобразуется в 2-узел, связанный с двумя (M/2)-узлами.

Операция удаления реализуется чуть сложнее и для краткости не описывается.

Высота дерева: $log_{M/2}(n)$ в худшем.

Количество узлов: верхняя оценка - $\left\lceil \frac{2N}{M} \right\rceil$.

Размер памяти для хранения узла (в битах): $(K + V + P) \cdot (M - 1) + P$.

Размер памяти для хранения всей структуры (в битах): верхняя оценка - $(K + V + P) \cdot \left(\frac{2N}{1 - 2/M} + M \right)$.

Время поиска: $O(log_2(n))$ в среднем и в худшем, если время загрузки из памяти k-узла не зависит от k; $M \cdot O(log_M(n))$ в среднем и в худшем, если время загрузки из памяти пропорционально k.

Время модификации: $M \cdot O(log_M(n))$ в среднем и в худшем.

Количество обращений к памяти при поиске: $h + 1$, где h - высота B-дерева; верхняя оценка - $log_{M/2}(n) + 1$ (при условии, что каждый узел может быть считан за одно обращение).

Размер считываемой области памяти при поиске (в битах): $(K + V + P) \cdot M \cdot O(log_2(n))$ в среднем и в худшем, верхняя оценка - $(K + V + P) \cdot M \cdot (log_{M/2}(n) + 1)$ (при условии, что узлы полностью загружаются из памяти).

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При удалении записи с отсутствующим ключом - 0. Для остальных случаев верхняя оценка - $(K + V + P) \cdot M \cdot O(log_M(n))$ (при условии, что узлы полностью загружаются из памяти).

Возможность разбиения памяти.

Значения можно хранить в отдельной области памяти, в этом случае размер узлов уменьшается, а значит число M увеличивается, что может привести к уменьшению высоты дерева. Каждый уровень B-дерева можно хранить в независимой памяти, однако это приводит к увеличению суммарного объема памяти, необходимого для хранения таблицы классификации.

Резюме.

Данную реализацию имеет смысл использовать в случаях, когда обращение к памяти является критической с точки зрения времени выполнения операцией, при этом существует возможность за одно обращение загружать крупные блоки данных. При выполнении этих условий B-дерево предоставляет возможность сократить количество обращений к памяти при поиске по сравнению с бинарными деревьями. При этом количество обращений при поиске не превосходит $h + 1$, где h - высота B-дерева.

3.1.6. Сравнение различных деревьев поиска

В данном параграфе был проведен обзор наиболее распространенных деревьев поиска. Несбалансированное ДБП в типичных случаях обладает логарифмическим временем поиска и записи, однако при неудачных входных данных (например, при добавлении в дерево записей в порядке возрастания или убывания ключей) высота дерева может стать соразмерной количеству записей, что приведет к линейному поиску и вставке. Рандомизированное ДБП устраняет данный недостаток путем случайного выбора места вставки и замещаемого узла, но тем не менее не гарантирует логарифмического времени поиска и модификации, поскольку вероятность того, что высота дерева будет соразмерна количеству записей, остается ненулевой.

АВЛ-дерево и красно-черное дерево являются сбалансированными и гарантируют логарифмическую высоту и время поиска/модификации за счет проведения дополнительной реструктуризации на основе вращений при выполнении операций модификации. АВЛ-дерево является более сбалансированным (его высота не превосходит $\lfloor 1.45 \cdot log_2(n+2) \rfloor$), поэтому обеспечивает более быстрый поиск. Однако красно-черные деревья обычно характеризуются более быстрыми операциями модификации за счет более гибкой структуры и, как следствие, отсутствия необходимости частых вращений. И АВЛ-деревья, и красно-черные деревья используют дополнительную память: 2 бита на узел в случае АВЛ-дерева (для хранения показателя разбалансировки) и 1 бит на узел в случае красно-черного дерева (для хранения цвета).

B-дерево имеет смысл использовать в случаях, когда при чтении из памяти загрузка данных происходит крупными блоками фиксированного размера, а само обращение к памяти является критической с точки зрения времени выполнения операцией. B-дерево позволяет минимизировать количество обращений к памяти за счет загрузки из памяти каждого узла за одно обращение. Так как высота B-дерева уменьшается при увеличении количества хранимых ключей в узле, это позволяет сократить количество обращений к памяти при поиске. Если же обращение к памяти не является критической операцией, имеет смысл использовать бинарные деревья поиска, так как B-дерево характеризуется более сложной и долгой операцией модификации при большом M.

Несбалансированное и рандомизированное ДБП не предоставляют возможности эффективной по памяти реализации хранения узлов дерева в различных независимых областях памяти в соответствии с их уровнем, поскольку для данных структур и количество различных уровней, и количество узлов на одном уровне для большинства уровней в наихудшем случае соразмерны количеству хранимых записей n, поэтому подобная реализация требовала бы суммарной памяти размера $O(N^2)$. Для АВЛ-дерева такая реализация также была бы неэффективна, так как при выполнении одной процедуры балансировки количество узлов, уровень которых меняется, в наихудшем случае равно $O(n)$. Возможность такой реализации предоставляют красно-черное дерево и B-дерево, однако это требует увеличения суммарного объема памяти, необходимого для хранения таблицы классификации.

В таблице 1 проводится сравнение основных характеристик рассмотренных реализаций деревьев поиска.

Таблица 1. Основные характеристики различных деревьев поиска.

Структура	**Размер памяти (в битах)**	**Время поиска/модификации**
Несбалансированное ДБП	$(K + V + 2P) \cdot N$	$O(log_2(n))$ в среднем, $O(n)$ в худшем
Рандомизированное ДБП	$(K + V + 2P) \cdot N$	$O(log_2(n))$ в среднем, $O(n)$ в худшем
АВЛ-дерево	$(K + V + 2P + 2) \cdot N$	$O(log_2(n))$
Красно-черное дерево	$(K + V + 2P + 1) \cdot N$	$O(log_2(n))$
B-дерево порядка M	$\leq (K + V + P) \cdot \left(\frac{2N}{1 - 2/M} + M \right)$	поиск: $O(log_2(n))$ при быстрой загрузке узла, иначе $M \cdot O(log_M(n))$; модификация: $M \cdot O(log_M(n))$

3.2. Хеш-структуры

3.2.1. Многоуровневое хеширование

Общее описание.

В многоуровневом[1] хешировании используются $L > 1$ независимых уровней SRAM-памяти Mem_1, ..., Mem_L и одна CAM-память CAM (память $(L+1)$-го уровня). Первый уровень также называется верхним, а $(L+1)$-й - нижним. Каждый уровень SRAM-памяти состоит из ячеек размера $K+V$ бит, любая ячейка может быть считана за одно обращение к памяти по ее номеру (ячейки нумеруются с нуля), N_l - количество ячеек в памяти Mem_l, $l \in \{1, ..., L\}$. Предполагается, что $N_1 \geq N_2 \geq ... \geq N_L$, а также что $\sum_{l=1}^{L} N_l > N$. Для каждого уровня l задана хеш-функция $h_l(key)$, отображающая произвольный ключ key размера K бит на номер из диапазона $[0, N_l - 1]$. Таким образом, любому ключу key сопоставляется по одной ячейке из каждого уровня SRAM-памяти.

Память CAM является ассоциативной памятью, хранящей записи вида (*ключ*, *значение*) в ячейках размера $K+V$ бит. Памяти CAM на вход подается ключ key, на выход она передает флаг b_{CAM} размера 1 бит и значение val. Если память содержит запись с ключом key, то флаг b_{CAM} равен 1, а значение val совпадает со значением соответствующей ключу key записи. Если память не содержит запись с ключом key, то флаг b_{CAM} равен 0, а значение val не определено. Ячейки в памяти CAM проверяются параллельно, тем самым обеспечивается быстрый поиск. Через N_{CAM} обозначается количество ячеек в памяти CAM. Предполагается, что $N_{CAM} \leq N_L$.

Инициализация и добавление записей.

При инициализации таблицы всем ключам ячеек памятей Mem_1, ..., Mem_L, CAM присваивается выделенное значение key_0, заведомо не использующееся в таблице (если в таблице в качестве ключей могут использоваться все 2^K двоичных чисел длины K, то размер ключа K для данной хеш-структуры увеличивается на один). Ячейки с ключом key_0 называются свободными. Ячейка l-го уровня памяти под номером i обозначается через (l, i). При добавлении в таблицу новой записи (key, val) проверяются ячейки памяти SRAM из множества соответствующих ключу key ячеек $\{(l, h_l(key)) \mid l \in \{1, ..., L\}\}$, для размещения записи выбирается ячейка минимального (наиболее высокого) уровня, являющаяся либо свободной, либо содержащая ключ key. Если такой ячейки нет, запись размещается в памяти CAM, при этом если в данной памяти уже хранится ключ key в некоторой ячейке, то для размещения используется эта ячейка, иначе любая свободная ячейка. Если же память CAM не содержит таких ячеек, диагностируется переполнение хеш-структуры.

На рис. 9 приведен пример добавления трех новых записей в хеш-структуру, состоящую из 4 уровней памяти, в которой первый уровень содержит 8 ячеек, второй - 4 ячейки, третий и четвертый - по 2 ячейки, красными крестиками отмечены занятые ячейки. Первой записи (key^1, val^1) соответствует 0-я ячейка памяти Mem_1, 1-я ячейка памяти Mem_2 и 0-я ячейка памяти Mem_3. Среди данных трех ячеек единственной свободной является ячейка памяти Mem_2, в которую и сохраняется запись. Второй записи (key^2, val^2) соответствует 0-я ячейка памяти Mem_1, 3-я ячейка памяти Mem_2 и 1-я ячейка памяти Mem_3. Запись сохраняется в последнюю ячейку, поскольку первые две заняты. Третьей записи (key^3, val^3) соответствуют две свободные ячейки: 4-я ячейка памяти Mem_1 и 2-я ячейка

[1] Название хеш-структуры выбрано авторами раздела и используется исключительно внутри данного документа. Похожая реализация описывается в работе [2].

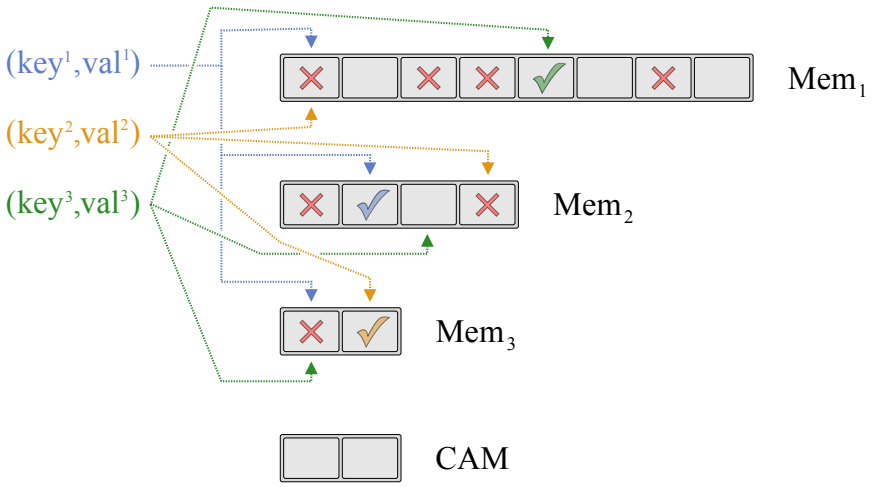

Рис. 9. Пример добавления трех новых записей в хеш-структуру, состоящую из 4 уровней памяти.

памяти Mem_2. Запись сохраняется в ячейку памяти более высокого уровня, то есть памяти Mem_1. Если бы для какой-то из записей отсутствовали свободные ячейки памяти SRAM, такая запись была бы сохранена в памяти CAM, содержащей две свободные ячейки.

Поиск.

Поиск по искомому ключу key реализуется с помощью последовательного или параллельного поиска в каждой используемой памяти (включая память CAM) и последующего объединения их результатов. Результатом поиска по отдельной памяти является флаг успеха размера 1 бит и значение размера V бит. Если память содержит запись с искомым ключом key, то флаг успеха равен 1, а значение равно значению такой записи, в противном случае флаг успеха равен 0, а значение не определено. Поиск по памяти Mem_l ($l \in \{1, ..., L\}$) предполагает считывание пары (key_l, val_l) из ячейки с номером $h_l(key)$ и проверки на равенство ключей key_l и key, в случае его выполнения возвращается флаг успеха 1 и значение val_l, иначе возвращается флаг успеха 0 и произвольное значение. При объединении результатов локальных поисков используются полученные флаги успеха и значения для каждой памяти. Если все флаги успеха равны нулю, то искомый ключ отсутствует в таблице, в таком случае возвращается флаг успеха 0 и произвольное значение. Если для некоторой памяти флаг успеха равен единице, то возвращается флаг успеха 1 и найденное в данной памяти значение.

Поиск по таблице может быть реализован аппаратно. На рис. 10 представлена схема такой реализации. Схема состоит из функциональных узлов (блоки серого цвета) и соединений (линии черного цвета). Каждое соединение характеризуется шириной (подписи синего цвета) и соответствует некоторой логической переменной (подписи черного цвета). Схема имеет один вход ширины K (искомый ключ key) и два выхода ширины 1 (флаг успеха $success$) и V (значение val). Функциональные узлы имеют следующие обозначения:

- «$h_l(\cdot)$» - блок, реализующий хеш-функцию $h_l(key)$, $l \in \{1, ..., L\}$; вход - ключ key размера K бит; выход - значение хеш-функции $h_l(key)$ размера $\lceil \log_2(N_l) \rceil$ бит;
- «Mem_l» - SRAM-память Mem_l уровня l, $l \in \{1, ..., L\}$; вход - номер ячейки $ind_l \in \{0, ..., N_l - 1\}$ размера $\lceil \log_2(N_l) \rceil$ бит, выход - хранящаяся в данной ячейке пара (key_l, val_l) размера $K + V$ бит;

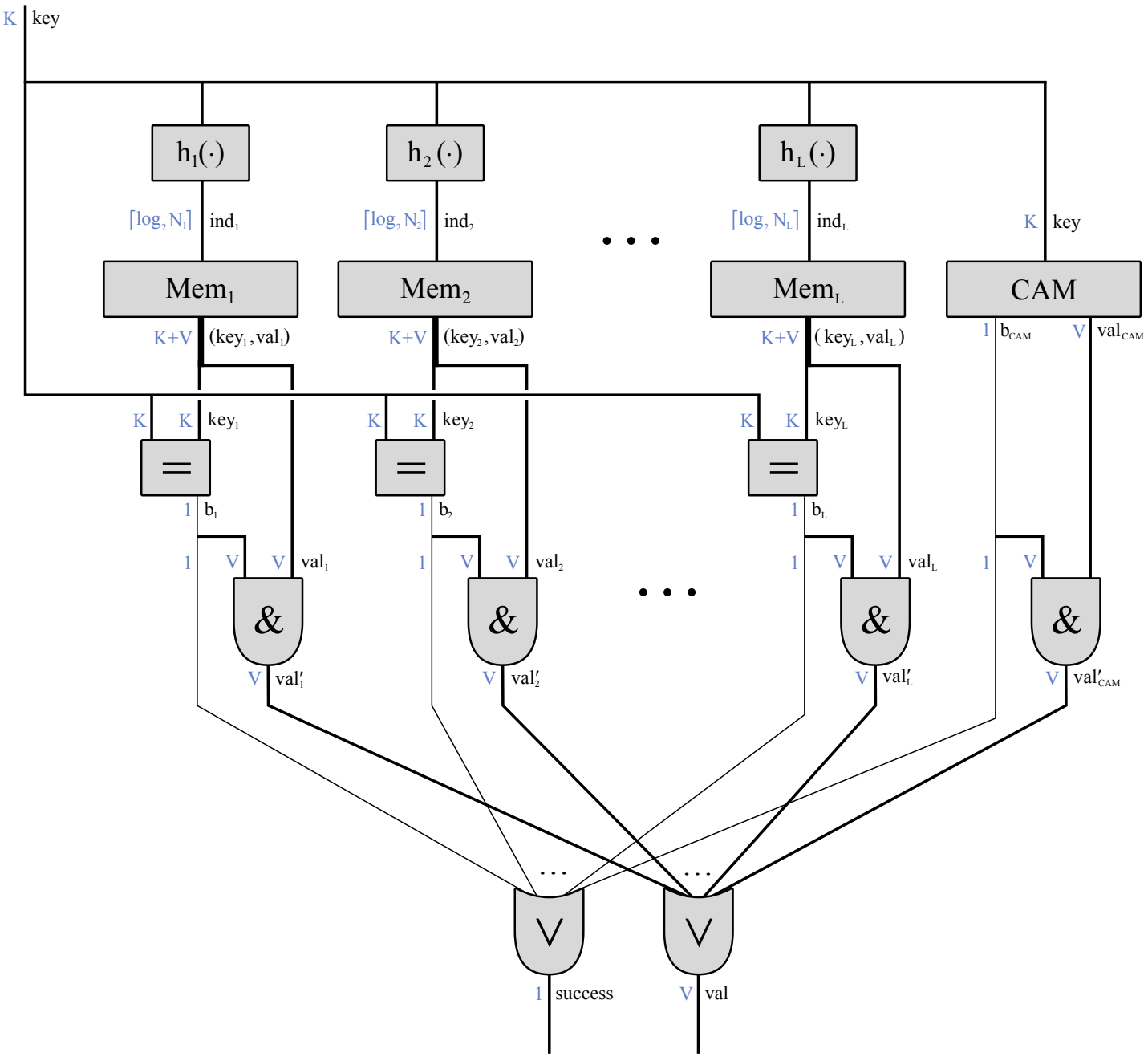

Рис. 10. Схема аппаратной реализации многоуровневого хеширования; серым цветом обозначены функциональные узлы, черным - шины (соединения) и соответствующие им переменные, синим - ширина (количество сигналов) шин.

- «CAM» - память CAM; вход - ключ key размера K бит; выход - пара (b_{CAM}, val_{CAM}), где b_{CAM} - флаг успеха размера 1 бит, val_{CAM} - значение размера V бит; если память содержит запись с ключом key, то $b_{CAM} = 1$, val_{CAM} равен значению этой записи; если память не содержит запись с ключом key, то $b_{CAM} = 0$, val_{CAM} не определено;
- «=» - компаратор с $2K$ входами; вход - пара ключей (key, key') размера K бит каждый; выход - флаг равенства b размера 1 бит; если $key = key'$, то $b = 1$, иначе $b = 0$;
- «&» - конъюнктор с $2V$ входами и V выходами; вход - пара значений $((x_1^1, ..., x_V^1), (x_1^2, ..., x_V^2))$ размера V бит каждое; выход - значение $(x_1^1 \wedge x_1^2, ..., x_V^1 \wedge x_V^2)$ размера V бит;
- «$\vee$» - дизъюнктор с L $(V \cdot L)$ входами и 1 (V) выходом; вход - L величин $(x^1, ..., x^L)$

$(((x_1^1, ..., x_V^1), ..., (x_1^L, ..., x_V^L)))$ размера 1 (V) бит каждая; выход - значение $x^1 \vee ... \vee x^L$ $((x_1^1 \vee ... \vee x_1^L, ..., x_V^1 \vee ... \vee x_V^L))$ размера 1 (V) бит.

Логические переменные обозначаются следующим образом:

- key - искомый ключ;
- ind_l - номер ячейки памяти Mem_l, соответствующей искомому ключу, $l \in \{1, ..., L\}$;
- (key_l, val_l) - пара (*ключ*, *значение*), считанная из памяти Mem_l при поиске, $l \in \{1, ..., L\}$;
- b_{CAM} - флаг успеха при поиске по памяти CAM;
- val_{CAM} - значение, считанное из памяти CAM при поиске;
- b_l - флаг успеха при поиске по памяти Mem_l, $l \in \{1, ..., L\}$;
- val'_l - если поиск по памяти Mem_l успешен, то значение, считанное из памяти, иначе $(0, ..., 0)$, $l \in \{1, ..., L\}$;
- val'_{CAM} - если поиск по памяти CAM успешен, то значение, считанное из памяти, иначе $(0, ..., 0)$;
- $success$ - флаг успеха поиска;
- val - если таблица содержит запись с искомым ключом, то значение такой записи, иначе результат не определен.

В схему на рис. 10 при необходимости могут быть добавлены внутренние регистры (например, перед блоками памяти) для уменьшения минимальной тактовой частоты (следовательно, увеличения максимальной пропускной способности), необходимой для корректного функционирования схемы.

Удаление и балансировка.

При удалении из таблицы записи с ключом key происходит последовательный поиск по ключу key в каждой используемой памяти в порядке увеличения (от верхнего к нижнему) уровня памяти. Если рассматриваемая на текущем шаге память содержит ячейку с данным ключом, то ключ в найденной ячейке заменяется на key_0, после чего процедура завершается без продолжения поиска на нижших уровнях.

Содержимое хеш-структуры называется *сбалансированным*, если для любой хранящейся в таблице записи при ее удалении и немедленном повторном добавлении не меняется уровень памяти, в котором хранится данная запись. При добавлении в таблицу произвольной записи гарантируется сохранение условия сбалансированности, однако при удалении записи возможна разбалансировка, возникающая из-за того, что при удалении записи, хранящейся в памяти уровня l, освободившаяся ячейка потенциально может подходить для записей уровней $l+1$, ..., $L+1$, которые были добавлены в таблицу позже удаленной записи.

При многократном повторении операций удаления и добавления записей из-за разбалансировки распределение занятых ячеек в памяти смещается в сторону низших уровней, несмотря на то, что количество занятых ячеек остается неизменным. Это может привести к быстрому переполнению хеш-структуры. Чтобы хеш-структура оставалась сбалансированной, необходимо проводить *процедуру балансировки*. Возможны две стратегии балансировки: пассивная и активная. В случае пассивной балансировки непосредственно при добавлении или удалении записей не проводятся какие-либо дополнительные преобразования, однако время от времени происходит проверка занятых ячеек в порядке уменьшения уровня памяти на возможность перемещения хранящихся в них записей в верхние уровни. Такая проверка инициируется при достижении некоторого порога специального критерия, такого как число удалений после предыдущей балансировки или суммарное количество занятых ячеек в нескольких низших уровнях.

При активной балансировке для каждой ячейки памяти (l, i) поддерживается FIFO-очередь в виде двусвязного списка, хранящая информацию о ячейках памяти, в которых в данный момент содержатся записи, не попавшие в ячейку (l, i) из-за ее занятости. При добавлении в таблицу новой записи в ячейку (l', i') для соответствующих данной записи занятых ячеек уровней памяти от 1 до $l-1$ пара (l', i') добавляется в конец очереди для каждой такой ячейки. Данные очереди представляют собой связи между ячейками «сверху вниз» и используются при освобождении ячеек: при освобождении ячейки (l, i) из соответствующей ей очереди (если она непуста) извлекается пара (l', i'), после чего 1) запись из ячейки (l', i') перемещается в ячейку (l, i) и 2) освобождается ячейка (l', i'). Чтобы связи «сверху вниз» оставались корректными, при удалении или перемещении записи (key, val) из ячейки (l', i') необходимо исключить пару (l', i') из очередей уровней памяти от $l+1$ до $l'-1$, где l - уровень памяти, куда перемещается запись (key, val), либо 0, если запись удаляется. Помимо этого, в очередях уровней памяти от 1 до $l-1$ пара (l', i') заменяется на новое местоположение записи (key, val). Для эффективного выполнения подобных модификаций для каждой занятой ячейки памяти (l, i) также хранится массив, состоящий из $l-1$ указателей на узлы FIFO-очередей, хранящих пару (l, i). Элементы массива упорядочены по возрастанию уровней памяти (от 1 до $l-1$), в очередях которых содержатся данные указатели. При удалении или перемещении записей содержимое массивов также корректируется. Такие массивы представляют собой связи между ячейками «снизу вверх».

Хотя активная балансировка требует хранения дополнительных данных, эти данные могут храниться в медленной памяти, поскольку они не используются непосредственно при поиске. При этом предполагается, что поиски в таблице проводятся на несколько порядков чаще, чем ее модификации. При активной балансировке сложность одной операции модификации в наихудшем случае равна $O(L)$, а размер дополнительной памяти не превосходит $C \cdot \left(\sum_{l=1}^{L-1} l \cdot N_{l+1} + L \cdot N_{CAM} \right)$, где C - размер памяти, необходимый для хранения информации о местоположении произвольной ячейки памяти (l, i) и трех указателей на узлы с такой информацией.

При использовании активной балансировки выполняется следующее свойство: пока в хеш-структуре ни разу не диагностировалось переполнение, в каждый момент времени распределение занятых ячеек среди различных уровней памяти зависит лишь от порядка добавления в таблицу записей, которые в данный момент хранятся в таблице, но не зависит от порядка добавления и удаления записей, которые к данному моменту уже не хранятся в таблице. Это свойство удобно для оценки вероятности переполнения таблицы, поскольку позволяет не учитывать удаленные записи.

Память.

Размер быстрой памяти для обеспечения поиска равен $(K+V) \cdot \left(\sum_{l=1}^{L} N_l + N_{CAM} \right)$ бит (включая $(K+V) \cdot N_{CAM}$ бит CAM-памяти). Для поддержки эффективной модификации требуется дополнительная медленная память, включающая память для хранения записей и память для поддержки активной балансировки.

Пропускная способность.

Пропускная способность аппаратной реализации поиска обратно пропорциональна минимальной тактовой частоте, необходимой для корректного функционирования схемы на рис. 10. Минимальная тактовая частота пропорциональна задержке критического пути схемы и может быть уменьшена за счет введения в схему дополнительных внутренних регистров. Таким образом, верхний предел возможной пропускной способности ограничен лишь пропускной способностью одного обращения к

блоку памяти максимального размера.

Задержка.

Задержка аппаратной реализации поиска примерно равна задержке критического пути схемы на рис. 10 (задержка увеличивается при добавлении в схему внутренних регистров). Критический путь включает в себя вычисление хеш-функции $h_1(key)$, обращение к памяти Mem_1 и прохождение через компаратор, конъюнктор и дизъюнктор.

Площадь на кристалле.

Площадь на кристалле, занимаемая аппаратной реализацией поиска, по большей части определяется площадью, отводимой под блоки памяти Mem_1 (SRAM), ..., Mem_L (SRAM), CAM (CAM) размеров в битах $(K+V)\cdot N_1$, ..., $(K+V)\cdot N_L$, $(K+V)\cdot N_{CAM}$ соответственно и площадью, занимаемой блоками вычисления хеш-функций $h_1(\cdot)$, ..., $h_L(\cdot)$.

Энергопотребление.

Несмотря на то, что побитовое энергопотребление памяти CAM намного выше, чем памяти SRAM, в схеме аппаратной реализации поиска доля энергопотребления памяти CAM в суммарном энергопотреблении невелика, поскольку предполагается, что размер памяти CAM небольшой по сравнению с общим размером используемой памяти.

Вероятность переполнения.

Вероятность переполнения хеш-структуры при добавлении в пустую таблицу N случайных записей с попарно различными ключами key^1, ..., key^N зависит от следующих величин:

- параметров L и N_1, ..., N_L, N_{CAM}, определяющих количество и размеры уровней памяти;
- хеш-функций $h_1(\cdot)$, ..., $h_L(\cdot)$;
- вероятностного распределения вектора $(key^1, ..., key^N)$.

Для оценки вероятности переполнения необходимо иметь представление либо о вероятностном распределении добавляемых/удаляемых ключах и их зависимостях друг между другом, либо хотя бы о «типовом» векторе $(key^1, ..., key^N)$. Например, если ключи key^1, ..., key^N независимы и равномерно распределены на множестве всевозможных битовых строк длины K, а $N = 64\,000$ то при правильном подборе параметров и хеш-функций можно гарантировать практически нулевую вероятность ($\leq 10^{-20}$) переполнения при суммарном размере памяти, превышающем $(K+V)\cdot N$ бит (минимальный размер памяти для хранения произвольных N записей) всего на 10%, и используя не более 25 уровней памяти и 50-100 ячеек CAM-памяти. Аналогичные результаты достигаются, если суффиксы ключей key^1, ..., key^N, состоящие из младших $\lceil log_2(N) \rceil$ разрядов ключей, независимы и равномерно распределены на множестве всевозможных битовых строк длины $\lceil log_2(N) \rceil$.

Подбор параметров.

Параметры хеш-структуры L, N_1, ..., N_L, N_{CAM}, $h_1(\cdot)$, ..., $h_L(\cdot)$ следует подбирать, имея представление о распределении добавляемых ключей. Варьируя параметры, можно достичь различных отношений трех основных настраиваемых критериев: память, сложность хеш-функций и вероятность переполнения. Чтобы сохранить суммарный размер памяти всего лишь немного превышающим $(K+V)\cdot N$ бит и при этом обеспечить довольно низкую вероятность переполнения, несколько верхних уровней памяти должны иметь высокую долю заполненности и в то же время хранить бóльшую часть записей, в то время как нижние уровни используются в основном как резервные для «захвата» записей, которым не хватило места в верхних уровнях. Чтобы этого достичь, количество ячеек в первом уровне можно взять примерно $N_1 = N/4$ и затем уменьшать размеры уровней в геометрической

прогрессии с некоторым множителем, не обязательно равным 2. Чтобы количество уровней не стало слишком большим, вместо использования нескольких нижних уровней добавляется память CAM в качестве последнего рубежа.

Предположим, что $K = 20$, $N = 128\,000$, а добавляемые ключи являются независимыми и имеют равномерное распределение на множестве всевозможных битовых строк длины 20. Тогда можно использовать параметры $L = 24$, $N_1 = N_2 = 32\,768$, $N_3 = N_4 = 16\,384$, $N_5 = N_6 = 8\,192$, $N_7 = N_8 = N_9 = 4\,096$, $N_{10} = N_{11} = 2\,048$, $N_{12} = ... = N_{17} = 1\,024$, $N_{18} = ... = N_{24} = 512$, $N_{CAM} = 50$, а в качестве хеш-функций использовать случайные[2] хеш-функции из [3]. В таком случае суммарно требуется 140 800 ячеек SRAM-памяти, а типичное состояние заполненности уровней представлено в табл. 2. Как видно из таблицы, состояние хеш-структуры имеет довольно большой запас незанятых ячеек нижних уровней (включая CAM-память). Среди 26 000 случайных генераций хеш-структуры ни одна запись не была добавлена в ячейку ниже 17-го уровня.

Таблица 2. Типичное состояние заполненности уровней описанной хеш-структуры в случае равномерно распределенных ключей и случайно сгенерированных хеш-функций согласно [3].

Уровень	**Число занятых ячеек**	**Общее число ячеек**
1	32 248	32 768
2	31 281	32 768
3	16 103	16 384
4	15 586	16 384
5	8 056	8 192
6	7 802	8 192
7	4 029	4 096
8	3 944	4 096
9	3 620	4 096
10	1 897	2 048
11	1 678	2 048
12	850	1 024
13	623	1 024
14	244	1 024
15	38	1 024
16	1	1 024
17	0	1 024
18	0	512
19	0	512
20	0	512
21	0	512
22	0	512
23	0	512
24	0	512
CAM	0	50
Сумма	**128 000**	**140 850**

[2] При случайном генерировании хеш-функции дополнительно следует обеспечивать условие невырожденности правой части матрицы параметров хеш-функции, что исключает «неудачно сгенерированные» хеш-функции с уменьшенной мощностью области значений.

Преимущества и недостатки.

Описанная хеш-структура при использовании активной балансировки обладает следующими достоинствами и недостатками по сравнению с древовидными структурами, trie-структурами и другими хеш-структурами. Достоинства:

- небольшой размер требуемой для поиска памяти, который может превышать минимальный размер (в случае хранения структуры в виде простого массива записей) лишь на 10%;
- минимальная задержка среди всех рассмотренных структур в случае аппаратной реализации; задержка приблизительно равна сумме задержек одного вычисления хеш-функции и одного обращения к SRAM-памяти;
- потенциально высокая пропускная способность аппаратной схемы, поскольку к каждой области памяти требуется всего лишь 1 обращение при поиске;
- относительно недорогие операции добавления и удаления с временем, в наихудшем случае пропорциональным количеству используемых уровней; в случае аппаратной реализации поиска и использования независимого компонента для обновления нет необходимости остановки поиска при обновлении;
- при известном распределении добавляемых в таблицу ключей и правильном подборе размеров уровней и хеш-функций может быть гарантирована практически нулевая (близкая к нулю) вероятность переполнения;
- детерминированность при поиске (в отличие от других хеш-структур с неизвестным заранее количеством обращений к памяти).

Недостатки:

- при неизвестном характере распределения добавляемых в таблицу ключей или плохом подборе размеров уровней и хеш-функций может быть переполнение хеш-структуры;
- для поддержки низкой вероятности переполнения может потребоваться значительное число (совпадающее с количеством уровней) хеш-функций; реализация одной хеш-функции с хорошими характеристиками может являться аппаратно-затратной;
- нет гарантий отсутствия переполнения хеш-структуры; при известных размерах уровней и хеш-функциях может быть вычислена «наихудшая» последовательность ключей размера, намного меньшего N, приводящая к переполнению структуры, что может являться потенциальной уязвимостью;
- необходимость хранения вспомогательных данных для обеспечения сбалансированности хеш-структуры при модификации; размер вспомогательных данных в худшем случае равен $C \cdot \left(\sum_{l=1}^{L-1} l \cdot N_{l+1} + L \cdot N_{CAM} \right)$, где C - размер памяти, необходимый для хранения информации о местоположении произвольной ячейки памяти (l, i) и трех указателей;
- использование CAM-памяти может привести к незначительному росту энергопотребления.

3.2.2. Линейное зондирование

Описание.

В линейном зондировании порядка H записи хранятся в массиве длины H с ячейками размера $K + V$ бит, нумеруемыми от нуля, $H > N$. Каждая ячейка содержит либо хранящуюся в таблице запись (*ключ*, *значение*), либо величину NULL, являющуюся индикатором пустоты ячейки (предпо-

лагается, что количество всевозможных записей, которые могут храниться в таблице, меньше 2^{K+V}, что позволяет выделить отдельное значение для хранения величины NULL). Заданная хеш-функция $h(\cdot)$ отображает произвольное двоичное число длины K в число из отрезка $[0, H-1]$. Содержимое массива должно удовлетворять следующим условиям:

- число занятых ячеек совпадает с количеством хранимых в таблице записей;
- для каждой хранимой в таблице записи существует ячейка с данной записью;
- для любой хранимой в таблице записи (*ключ*, *значение*) для номера $k_1 = h(\textit{ключ})$ и номера содержащей данную запись ячейки k_2 в случае $k_1 \neq k_2$ ячейки с номерами k_1, $(k_1 + 1)\ mod\ H$, $(k_1 + 2)\ mod\ H$, ... , k_2 являются занятыми.

Таким образом, хеш-функция ставит в соответствие произвольному ключу номер базовой ячейки, предназначенной для хранения записи с данным ключом, при этом в действительности данная запись может располагаться в ячейке, находящейся правее базовой, но не отделяемой от нее пустыми ячейками. При определении правой ячейки массив считается циклическим, т.е. правой ячейкой ячейки с номером $H-1$ является ячейка с номером 0.

...

3.3. Поразрядный поиск

В данном параграфе рассматриваются структуры данных на основе поразрядного поиска. Такие структуры похожи на рассмотренные выше деревья поиска. Отличие от последних заключается в том, что на каждом шаге алгоритма поиска вместо полного сравнения ключей происходит лишь сравнения небольших фрагментов, т.н. разрядов.

3.3.1. Дерево цифрового поиска

Описание.

Каждой хранящейся в таблице паре (*ключ*, *значение*) ставится в соответствие узел, содержащий данную пару и два указателя: на левый дочерний узел и на правый дочерний узел. В частном случае указатели могут быть равны NULL. Все узлы объединяются в корневое дерево, каждому узлу которого соответствует двоичный префикс, однозначно определяемый местоположением узла в дереве следующим образом:

- префикс корневого узла равен пустой строке;
- если узел является левым дочерним узлом своего родителя, то его префикс получается из родительского добавлением справа нуля;
- если узел является правым дочерним узлом своего родителя, то его префикс получается из родительского добавлением справа единицы.

Для каждого поддерева с корнем *root* должно выполняться следующее свойство: ключи всех узлов данного поддерева начинаются с префикса, соответствующего *root* (рис. 11).

Высота описанного корневого дерева не превышает размера ключа K, а каждый хранимый в таблице классификации ключ находится где-то на пути от корня до листа, однозначно определяемым левыми разрядами данного ключа и структурой дерева.

Алгоритм поиска.

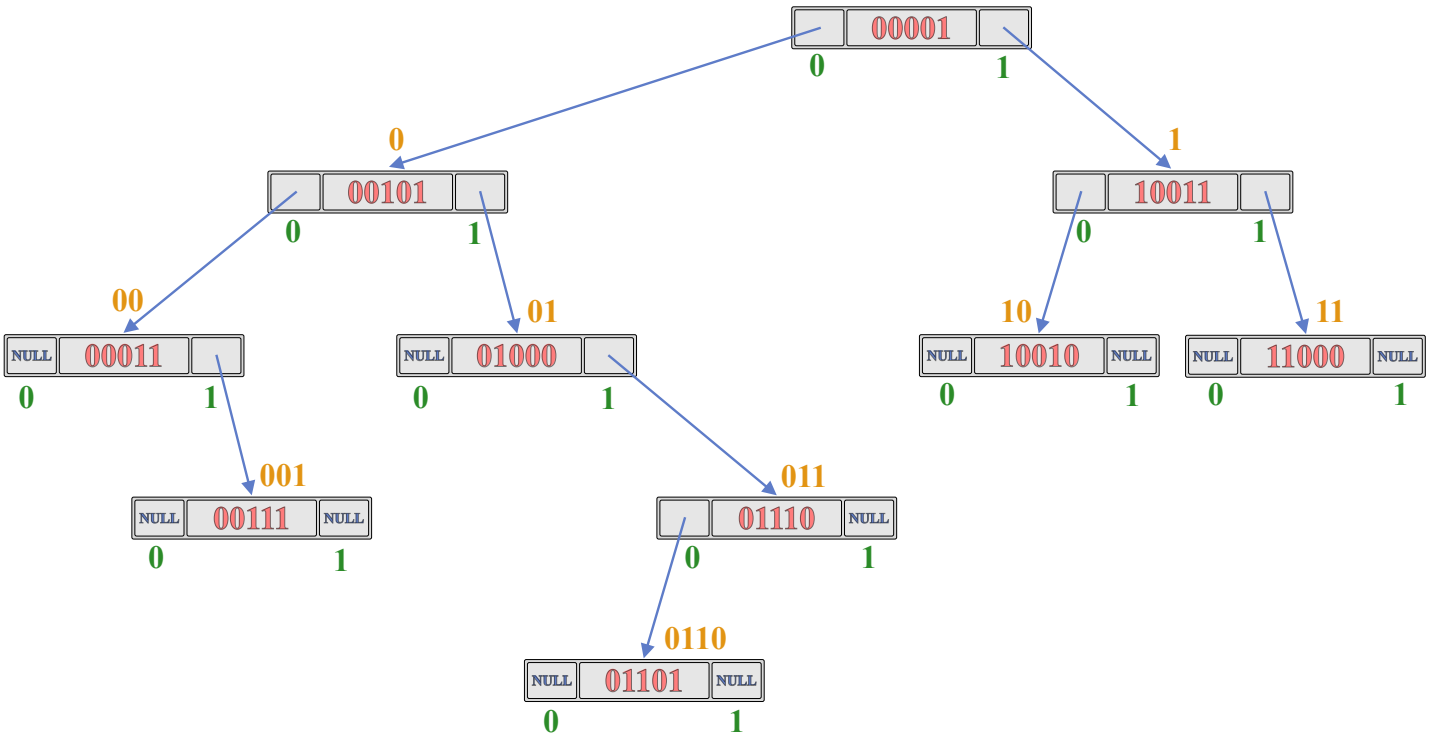

Рис. 11. Пример дерева цифрового поиска с $K = 5$, полученного в результате последовательного добавления в первоначально пустое дерево записей с ключами 00001, 10011, 00101, 10010, 00011, 01000, 01110, 00111, 11000, 01101; красным цветом представлены хранимые ключи, синим - указатели, оранжевым - префиксы узлов; для компактности значения в узлах не представлены.

Проводится спуск по дереву от корневого узла к листьям. При спуске учитывается номер текущего проверяемого разряда, вначале он равен нулю. На каждом шаге рассматривается очередной узел и проводится сравнение искомого ключа с ключом узла. Если ключи совпадают, то поиск успешен и хранящееся в текущем узле значение является результатом поиска. В противном случае проверяется значение текущего разряда искомого ключа: если оно равно нулю (единице), происходит переход к следующему шагу с рассмотрением левого (правого) дочернего узла в качестве текущего с предварительным увеличением номера текущего проверяемого разряда на один. Если на каком-то шаге ключи не совпадают, а нужного дочернего узла не существует (соответствующий указатель равен NULL), то дерево не содержит искомый ключ (поиск неуспешен).

Алгоритм модификации.

Проводится поиск по целевому ключу. Для операции добавления в случае успешного поиска значение найденного узла, содержащего целевой ключ, изменяется на целевое. Для операции добавления в случае неуспешного поиска для целевого ключа и целевого значения создается новый узел дерева, указатель на который затем добавляется в последний просмотренный при поиске узел, при этом новый узел становится левым (правым) дочерним узлом, если значение последнего проверяемого при поиске разряда целевого ключа равно нулю (единице).

Для операции удаления в случае успешного поиска выполняются следующие действия:

- если узел, содержащий целевой ключ, является листом дерева, данный узел удаляется из дерева;
- если узел v, содержащий целевой ключ, содержит дочерние узлы, то в поддереве с корнем v произвольно выбирается листовой узел, ключ и значение выбранного листового узла записываются

в v, после чего листовой узел удаляется из дерева.

Высота дерева: K в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K + V + 2P$.

Размер памяти для хранения всей структуры (в битах): $(K + V + 2P) \cdot N$.

Время поиска: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Время модификации: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем, $2(K + 1)$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, P, V)$ бит), $K+2$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K + P, V)$ бит).

Размер считываемой области памяти при поиске (в битах): $(K + P) \cdot O(log_2(n))$ в среднем, $(K + P) \cdot K + K + V$ в худшем.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - $K+V+3P$. При удалении записи с отсутствующим ключом - 0. При удалении записи с присутствующим ключом - $K + V + P$.

Возможность разбиения памяти.

Значения можно хранить в отдельной области памяти. Также возможно разбиение множества всех узлов на блоки в соответствием с уровнем узла (в данном случае под уровнем понимается расстояние от корня) с хранением различных блоков в независимых областях памяти. Это возможно, поскольку при выполнении любой операции модификации изменению подлежат не более двух узлов, уровень остальных узлов при этом не меняется.

Если количество блоков совпадает с высотой дерева, т.е. различным уровням соответствуют различные блоки, то для хранения l-го блока, состоящего из узлов, расположенных на расстоянии l от корня ($l \in \{0, ..., K\}$), достаточно $(K + V + 2P) \cdot \min\left(2^l, \left\lceil \frac{N}{2} \right\rceil\right)$ бит. Для хранения всего дерева необходимо примерно $(K + V + 2P) \cdot \left(\frac{K - log_2(N) + 3}{2}\right) \cdot N$ бит. Таким образом, при таком разбиении размер памяти увеличивается примерно в $\frac{K - log_2(N) + 3}{2}$ раз. Размер памяти можно уменьшить, если в одном блоке хранить узлы нескольких соседних уровней. Также размер памяти можно сократить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив его требованием вмещения N записей в типичных случаях или с большой вероятностью, но без строгой гарантии.

Резюме.

Данная реализация по своим характеристикам похожа на сбалансированные деревья бинарного поиска. Как и в сбалансированных деревьях бинарного поиска, количество обращений к памяти при поиске, время поиска и время модификации пропорциональны высоте дерева. Если в сбалансированных деревьях бинарного поиска в худшем случае высота дерева соразмерна $log_2(n)$ ($1.45\,log_2(n + 2)$ для АВЛ-дерева и $2\,log_2(n)$ для красно-черного дерева), то в данной реализации высота дерева не превосходит размера ключа K. Если K соразмерно величине $2\,log_2(n)$, деревья цифрового поиска представляют хорошую альтернативу деревьям бинарного поиска.

Первым преимуществом дерева цифрового поиска перед деревьями бинарного поиска является

отсутствие сравнений при поиске и модификации, используются лишь проверки на равенство. Второе преимущество - простой алгоритм модификации, не требующий проведения операций балансировки дерева. Это приводит к более легкой аппаратной реализации алгоритмов поиска и модификации.

Данная реализация поддерживает возможность разбиения памяти на несколько независимых областей, однако это обычно требует кратного увеличения суммарного размера памяти. Размер памяти можно понизить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным.

3.3.2. Trie-дерево

Описание.

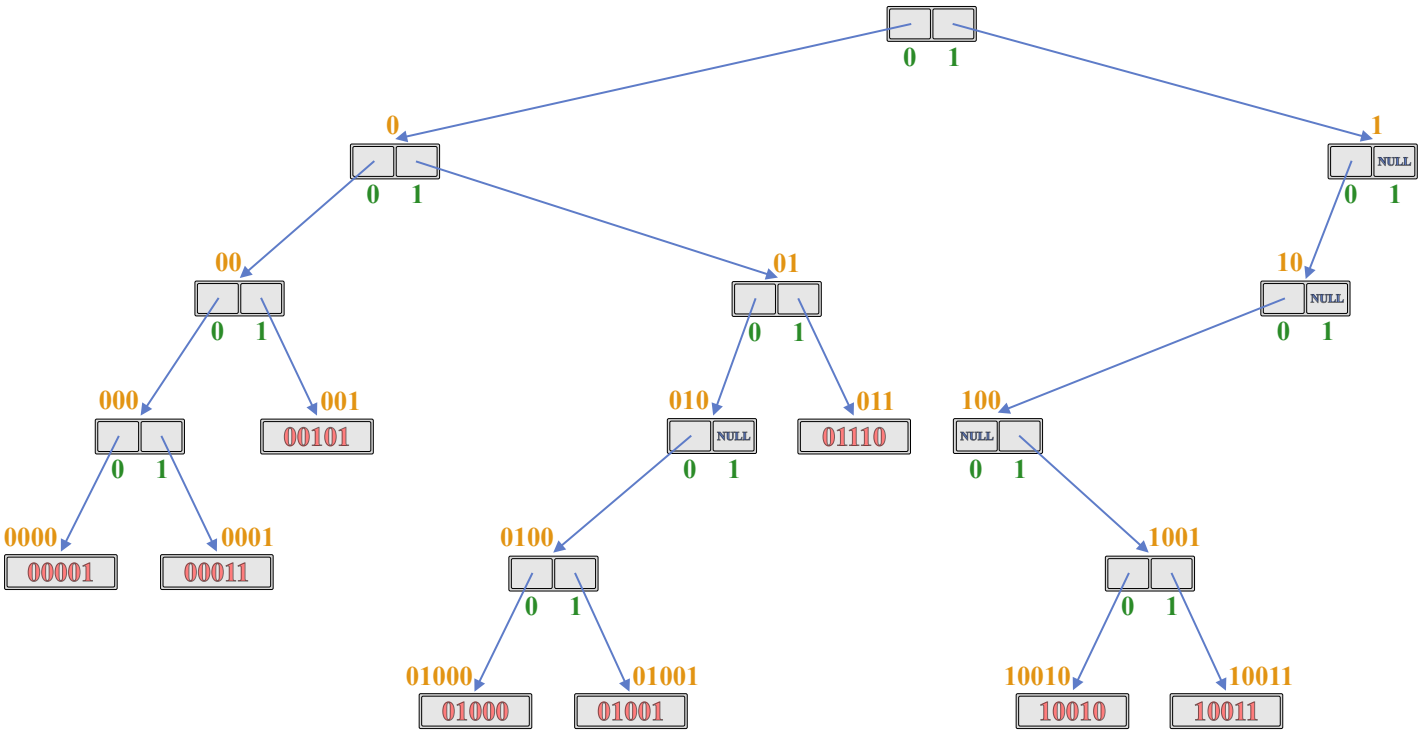

Рис. 12. Пример trie-дерева с $K = 5$, содержащего ключи 00001, 10011, 00101, 10010, 00011, 01000, 01001, 01110; красным цветом представлены хранимые ключи, синим - указатели, оранжевым - префиксы узлов; для компактности значения в листовых узлах не представлены.

Trie-дерево является корневым деревом, в котором, в отличие от дерева цифрового поиска, ключи и значения могут храниться только в листьях дерева. Все узлы дерева делятся на два типа: внутренние и листья. Каждый внутренний узел содержит два указателя: на левый дочерний узел и на правый дочерний узел. В частном случае один из указателей может быть равен NULL. Лист содержит пару (*ключ*, *значение*) и не содержит указателей. Каждому узлу дерева соответствует двоичный префикс, однозначно определяемый местоположением узла в дереве следующим образом:

- префикс корневого узла равен пустой строке;
- если узел является левым дочерним узлом своего родителя, то его префикс получается из родительского добавлением справа нуля;
- если узел является правым дочерним узлом своего родителя, то его префикс получается из родительского добавлением справа единицы.

Для каждого поддерева с корнем *root* должно выполняться следующее свойство: ключи всех листовых узлов данного поддерева начинаются с префикса, соответствующего *root*. При этом родитель любого листового узла должен иметь два дочерних узла (рис. 12).

Структура trie-дерева не зависит от порядка вставки ключей, его высота не превышает размера ключа K, а каждый хранимый в таблице классификации ключ хранится в листе, путь от корня до которого однозначно определяется левыми разрядами данного ключа и структурой дерева. Предполагается, что внутренние узлы и листовые узлы хранятся в отдельных областях памяти, поэтому по адресу узла в памяти однозначно определяется тип узла.

Алгоритм поиска.

Проводится спуск по дереву от корневого узла к листьям. При спуске учитывается номер текущего проверяемого разряда, вначале он равен нулю. На каждом шаге рассматривается очередной узел. Если узел является листовым, проводится сравнение искомого ключа с ключом узла: если ключи совпадают, то поиск успешен и хранящееся в текущем узле значение является результатом поиска, в противном случае дерево не содержит искомый ключ (поиск неуспешен). Если узел является внутренним, то проверяется значение текущего разряда искомого ключа: если оно равно нулю (единице), происходит переход к следующему шагу с рассмотрением левого (правого) дочернего узла в качестве текущего с предварительным увеличением номера текущего проверяемого разряда на один. Если на каком-то шаге при рассмотрении внутреннего узла нужного дочернего узла не существует (соответствующий указатель равен NULL), то дерево не содержит искомый ключ (поиск неуспешен).

Алгоритм модификации.

Проводится поиск по целевому ключу. Для операции добавления в случае успешного поиска значение найденного узла, содержащего целевой ключ, изменяется на целевое.

Для операции добавления в случае неуспешного поиска выполняются следующие действия:

- если поиск завершился на внутреннем узле, то соответствующему указателю на дочерний узел, равному NULL, присваивается ссылка на новый листовой узел, содержащий целевые ключ и значение;
- если поиск завершился на листовом узле u, то
  1) для родителя w узла u указатель на u заменяется указателем на новый внутренний узел r с равными NULL указателями (если узел u является корнем дерева, то узел r становится новым корнем дерева);
  2) вводится число k, равное длине пути от корня до узла r;
  3) для узла r указатель на дочерний узел (левый или правый, в зависимости от k-го разряда ключа узла u) заменяется указателем на узел u;
  4) проверяется k-й разряд целевого ключа; если значение разряда равно нулю (единице) и отличается от значения этого же разряда ключа узла u, то указателю на левый (правый) дочерний узел узла r присваивается ссылка на новый листовой узел, содержащий целевые ключ и значение, после чего происходит завершение операции модификации;
  5) происходит переход к п. 1.

Для операции удаления в случае успешного поиска выполняются следующие действия:

1) листовой узел, содержащий целевой ключ, удаляется из дерева; если удаленный узел являлся корневым, происходит завершение операции модификации;

2) если единственным оставшимся дочерним узлом родителя удаленного узла является листовой узел, то происходит процесс поднятия данного листового узла вверх по дереву с замещением им родителей до тех пор, пока либо родитель этого листового узла не будет иметь двух дочерних узлов, либо поднимаемый листовой узел не станет корнем дерева.

Высота дерева: K в худшем.

Количество внутренних узлов: верхние оценки - $\frac{(K - \lfloor log_2(n) - 1 \rfloor - 1) \cdot n}{2} + 2^{\lfloor log_2(n) - 1 \rfloor + 1} - 1$, $\frac{(K - log_2(n) + 3) \cdot n}{2}$, $\frac{K \cdot n}{2}$, в среднем - $1.44 \cdot n$.

Количество листовых узлов: n.

Размер памяти для хранения внутреннего узла (в битах): $2\hat{P}$, где $\hat{P} = \left\lceil log_2 \left(\frac{(K - \lfloor log_2(N) - 1 \rfloor + 1) \cdot N}{2} + 2^{\lfloor log_2(N) - 1 \rfloor + 1} \right) \right\rceil \leq \left\lceil log_2 \left(\frac{(K+2) \cdot N}{2} + 1 \right) \right\rceil$ - размер указателя на узел (в битах).

Размер памяти для хранения листового узла (в битах): $K + V$.

Размер памяти для хранения всей структуры (в битах).

Точная величина - $(K + V) \cdot N + \hat{P} \cdot \left((K - \lfloor log_2(N) - 1 \rfloor - 1) \cdot N + 2^{\lfloor log_2(N) - 1 \rfloor + 2} - 2 \right)$. Верхние оценки - $(K + V + \hat{P} \cdot (K - log_2(N) + 3)) \cdot N$, $\left(K + V + K \cdot \hat{P} \right) \cdot N$. Размер может быть уменьшен до $(K + V + C \cdot 2P) \cdot N$, где C - небольшое число, соразмерное 2, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным.

Время поиска: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Время модификации: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем, $K + 2$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(\hat{P}, K, V)$ бит).

Размер считываемой области памяти при поиске (в битах): $\hat{P} \cdot O(log_2(n))$ в среднем, $(\hat{P} + 1) \cdot K + V$ в худшем.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - $(2\hat{P} + 1) \cdot K + V$ в худшем. При удалении записи с отсутствующим ключом - 0. При удалении записи с присутствующим ключом - $\hat{P}$.

Возможность разбиения памяти.

Внутренние узлы и листовые узлы, как следует из описания структуры, хранятся в отдельных областях памяти. Хранение ключей и значений для листовых узлов также можно разделить. Помимо этого, аналогично дереву цифрового поиска, возможно разбиение множества всех внутренних узлов на блоки в соответствием с уровнем узла (в данном случае под уровнем понимается расстояние от корня) с хранением различных блоков в независимых областях памяти.

Если количество блоков совпадает с высотой дерева, т.е. различным уровням соответствуют различные блоки, то для хранения l-го блока, состоящего из внутренних узлов, расположенных на расстоянии l от корня ($l \in \{0, ..., K - 1\}$), достаточно $\hat{P} \cdot \min\left(2^{l+1}, N\right)$ бит. Для хранения всех блоков достаточно $\hat{P} \cdot \left((K - \lfloor log_2(N) - 1 \rfloor - 1) \cdot N + 2^{\lfloor log_2(N) - 1 \rfloor + 2} - 2 \right) \leq \hat{P} \cdot (K - log_2(N) + 3) \cdot N$ бит, а всей структуры - $(K + V) \cdot N + \hat{P} \cdot \left((K - \lfloor log_2(N) - 1 \rfloor - 1) \cdot N + 2^{\lfloor log_2(N) - 1 \rfloor + 2} - 2 \right) \leq (K + V + \hat{P} \cdot (K - log_2(N) + 3)) \cdot N$ бит, при этом данная оценка совпадает с верхней оценкой в случае отсутствия разбиения.

Размер памяти не получится уменьшить при хранении в одном блоке промежуточных узлов нескольких соседних уровней. Однако, как и в случае дерева цифрового поиска, размер памяти можно сократить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив его требованием вмещения N записей в типичных случаях или с большой вероятностью, но без строгой гарантии.

Резюме.

В trie-дереве ключи и значения таблицы классификации хранятся в листьях дерева. В связи с этой особенностью данная реализация имеет следующие основные отличия от дерева цифрового поиска:

- увеличение верхней оценки количества узлов и размера памяти, гарантирующего хранение N произвольных записей;
- увеличение размера памяти для хранения указателя на узел;
- уменьшение гарантированной оценки размера считываемой области памяти при поиске;
- уменьшение количества сравнений ключей при поиске до одного.

Как и в случае дерева цифрового поиска, данная реализация поддерживает возможность разбиения памяти на несколько независимых областей. Необходимый суммарный размер памяти как в случае разбиения, так и без него, одинаков. Его можно понизить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным. Без данного упрощения реализация является заведомо неэффективной по памяти при больших N и K.

3.3.3. Patricia-дерево

Описание.

Patricia-дерево задумывалось как реализация, устраняющая следующие недостатки trie-дерева:

1) наличие двух различных типов узлов: внутренних и листовых;
2) проблема длинных цепочек без ветвления, заключающаяся в возможном наличии большого числа следующих друг за другом внутренних узлов, каждый из которых имеет только один дочерний узел; такие цепочки могут занимать значительную часть памяти и увеличивать как среднее количество обращений к памяти при поиске, так и само время поиска.

Patricia-дерево решает вторую проблему за счет замены каждой такой цепочки одним узлом, в котором дополнительно хранится номер разряда, который следует проверять в данном узле для принятия решения о переходе в правильный дочерний узел при поиске. Первая проблема решается с помощью устранения листовых узлов с хранением их ключей и значений в одном из узлов дерева, при этом для каждого узла, который в соответствующем trie-дереве является предлистовым, вместо указателя на листовой узел используется указатель на один из предшествующих узлов либо на этот же узел (обратный указатель). Индикатором того, что узел в patricia-дереве следует рассматривать как листовой при поиске, является отсутствие увеличения номера разряда данного узла по сравнению с пройденным на предыдущем шаге.

Таким образом, каждый узел в patricia-дереве хранит ключ, значение, два указателя на дочерние узлы и номер разряда, при этом количество узлов совпадает с количеством хранимых ключей, а указатели со значением NULL отсутствуют (рис. 13).

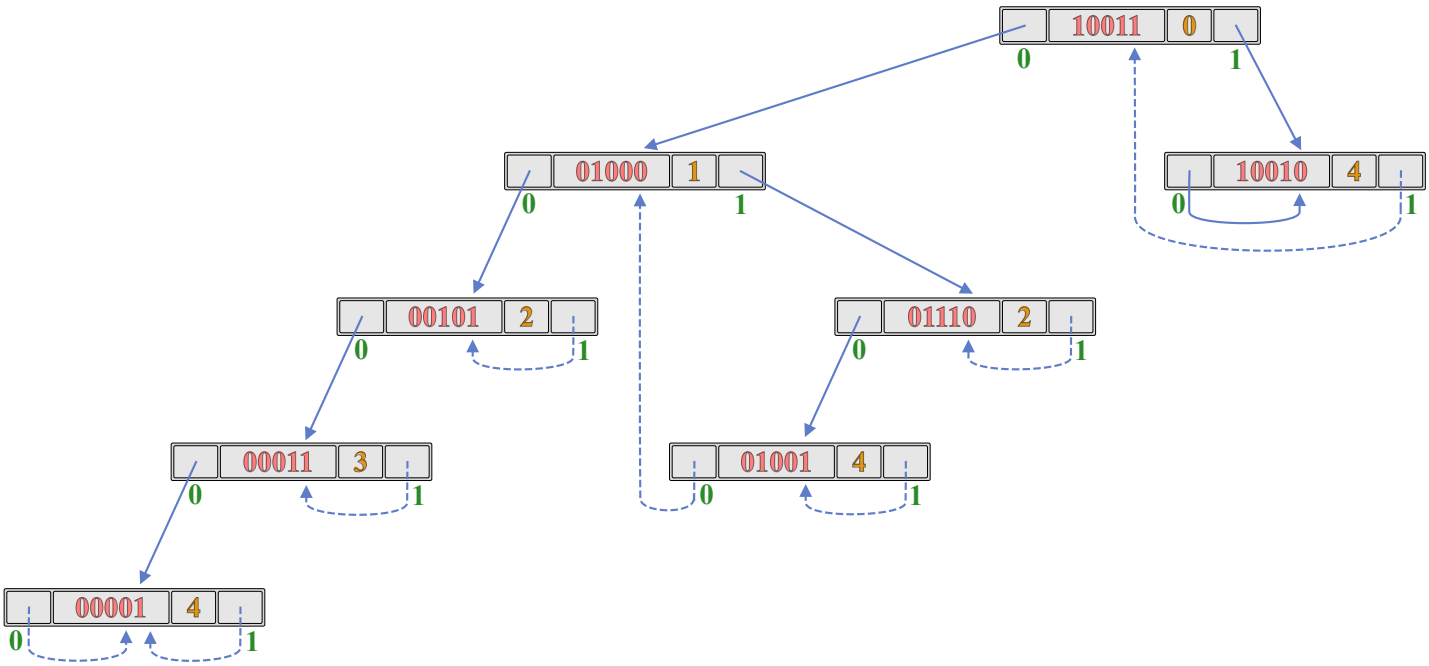

Рис. 13. Пример Patricia-дерева с $K = 5$, полученного в результате последовательного добавления в первоначально пустое дерево записей с ключами 00001, 10011, 00101, 10010, 00011, 01000, 01001, 01110; красным цветом представлены хранимые ключи, синим - указатели, синим пунктиром - обратные указатели, оранжевым - номера проверяемых разрядов узлов при поиске; для компактности значения в узлах не представлены.

Алгоритм поиска.

Проводится спуск по дереву, начиная с корневого узла. При спуске учитывается номер последнего проверенного разряда, вначале он равен -1. На каждом шаге рассматривается очередной узел. Если номер разряда текущего узла больше номера последнего проверенного разряда, то происходит проверка разряда с данным номером у искомого ключа: если он равен нулю (единице), то происходит переход к левому (правому) дочернему узлу текущего узла с предварительным обновлением номера последнего проверенного разряда. Если же номер разряда текущего узла меньше либо равен номеру последнего проверенного разряда, то проводится сравнение искомого ключа и ключа текущего узла: если они равны, то поиск успешен и хранящееся в текущем узле значение является результатом поиска, в противном случае дерево не содержит искомый ключ (поиск неуспешен).

Алгоритм модификации.

Проводится поиск по целевому ключу, который завершается либо узлом, содержащим целевой ключ (успешный поиск), либо узлом с ключом, не равным целевому (неуспешный поиск). Для операции добавления в случае успешного поиска значение найденного узла изменяется на целевое.

Для операции добавления в случае неуспешного поиска выполняются следующие действия:

1) находится самый левый разряд, которым отличаются целевой и найденный ключи (данный разряд при поиске по целевому ключу не проверялся);
2) проводится повторный поиск по целевому ключу, целью которого является нахождение места вставки нового узла с целевыми ключом и значением, одним из указателей, ссылающимся на этот же вставляемый узел, и найденным номером разряда; если в результате поиска попадается узел с номером разряда, большим найденного, то новый узел вставляется перед этим узлом; в противном случае новый узел вставляется после последнего пройденного узла (дочерним узлом

которого является узел, содержащий найденный во время первого поиска ключ).

Для операции удаления в случае успешного поиска выполняется исключение из дерева узла, указывающего на содержащий целевой ключ узел. Пусть u - исключаемый узел, k и v - ключ и значение узла u, w - родитель узла u, r - узел, содержащий целевой ключ. В таком случае связи в дереве при исключении узла u восстанавливаются следующим образом:

- ключ k и значение v записываются в узел r;
- указателю узла w, ссылающемуся на узел u, присваивается ссылка на узел r.

Высота дерева: K в худшем.

Количество узлов: n.

Размер памяти для хранения узла (в битах): $K + V + 2P + Q$.

Размер памяти для хранения всей структуры (в битах): $(K+V+2P+Q) \cdot N$, где $Q = \lceil log_2(K) \rceil$ - размер поля, хранящего номер разряда ключа.

Время поиска: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Время модификации: $O(log_2(n))$ в среднем, $O(K)$ в худшем.

Количество обращений к памяти при поиске: $O(log_2(n))$ в среднем, $2(K + 1)$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, V, P, Q)$ бит), $K + 2$ (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, V, 2P + Q)$ бит).

Размер считываемой области памяти при поиске (в битах): $(P + Q) \cdot O(log_2(n))$ в среднем, $(P+Q+1) \cdot K+V$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, V, P, Q)$ бит); $(2P + Q) \cdot O(log_2(n))$ в среднем, $(2P + Q + 1) \cdot K + V$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(K, V, 2P + Q)$ бит).

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - $K + V + 3P + Q$ в худшем. При удалении записи с отсутствующим ключом - 0. При удалении записи с присутствующим ключом - $K + V + P$.

Возможность разбиения памяти.

Для patricia-дерева есть возможность хранения узлов в различных независимых областях памяти, при которой узлы хранятся в соответствии с их уровнем (расстоянием от корня) в соответствующем trie-дереве, а ключи и значения, хранящиеся в каждом узле patricia-дерева, перемещаются в отдельные листовые узлы. Однако в таком случае теряются преимущества patricia-дерева, фактически реализация превращается в trie-дерево. Разбиение узлов в соответствии с их уровнем в patricia-дереве неэффективно по времени модификации, так как при таком разбиении при добавлении/удалении узла количество узлов, уровень которых меняется, в наихудшем случае равно $O(n)$.

Резюме.

Patricia-дерево является улучшением trie-дерева, обеспечивающим хорошую гарантированную оценку по памяти, пропорциональную числу узлов, и более быстрый поиск за счет устранения длинных цепочек узлов без ветвления. Также данная реализация не требует отдельного типа узлов для хранения листьев дерева, содержащих ключи и значения. Однако сама по себе данная структура не позволяет эффективно реализовать хранение узлов дерева в различных независимых областях памяти, так как при выполнении одной операции модификации уровень $O(n)$ узлов может измениться (как в случае стандартного определения уровня для дерева, так и в случае определения уровня как расстояния от корня). Эффективная по времени модификации реализация хранения узлов в

различных независимых областях памяти совпадает с реализацией для trie-дерева с присущей ей недостатками.

3.3.4. Многопутевое trie-дерево

Описание.

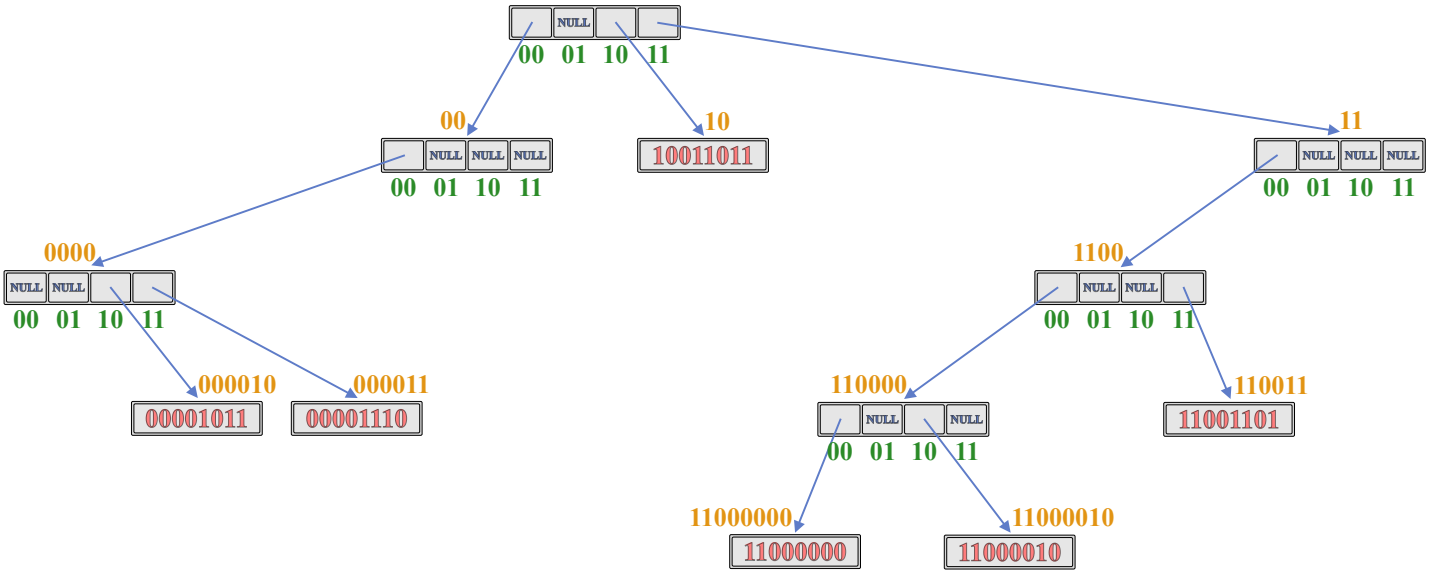

Рис. 14. Пример многопутевого trie-дерева с $K = 8$, $R = 4$, содержащего ключи 00001011, 00001110, 10011011, 11000000, 11000010, 11001101; красным цветом представлены хранимые ключи, синим - указатели, оранжевым - префиксы узлов; для компактности значения в листовых узлах не представлены.

Многопутевое trie-дерево порядка R является обобщением trie-дерева, в котором каждый внутренний узел содержит $R = 2^L$ указателей на дочерние узлы ($L \geq 1$). Каждый листовой узел, как и в случае trie-дерева, содержит только пару (*ключ*, *значение*). Каждому узлу дерева соответствует двоичный префикс, однозначно определяемый местоположением узла в дереве следующим образом:

- префикс корневого узла равен пустой строке;
- если узел является k-м дочерним узлом своего родителя ($k \in \{0, ..., R-1\}$), то его префикс получается из родительского добавлением справа двоичной строки длины L, являющейся представлением десятичного числа k.

Для каждого поддерева с корнем *root* должно выполняться следующее свойство: ключи всех листовых узлов данного поддерева начинаются с префикса, соответствующего *root*. При этом родитель любого листового узла должен иметь как минимум два дочерних узла (рис. 14).

Структура многопутевого trie-дерева не зависит от порядка вставки ключей, его высота не превышает $\left\lceil \frac{K}{L} \right\rceil$, а каждый хранимый в таблице классификации ключ хранится в листе, путь от корня до которого однозначно определяется левыми разрядами данного ключа и структурой дерева. Предполагается, что внутренние узлы и листовые узлы хранятся в отдельных областях памяти, поэтому по адресу узла в памяти однозначно определяется тип узла.

Алгоритм поиска.

Проводится спуск по дереву от корневого узла к листьям. При спуске учитывается текущий диапазон номеров проверяемых разрядов, состоящий из L номеров, вначале он равен $[0, L-1]$. На

каждом шаге рассматривается очередной узел. Если узел является листовым, проводится сравнение искомого ключа с ключом узла: если ключи совпадают, то поиск успешен и хранящееся в текущем узле значение является результатом поиска, в противном случае дерево не содержит искомый ключ (поиск неуспешен). Если узел является внутренним, то

1) вводится число k, образованное разрядами искомого ключа из текущего диапазона номеров проверяемых разрядов;
2) границы текущего диапазона номеров проверяемых разрядов увеличиваются на L;
3) если указатель на k-й дочерний узел не равен NULL, то происходит переход к следующему шагу с рассмотрением данного дочернего узла в качестве текущего; в противном случае дерево не содержит искомый ключ (поиск неуспешен).

Алгоритм модификации.

Проводится поиск по целевому ключу. Для операции добавления в случае успешного поиска значение найденного узла, содержащего целевой ключ, изменяется на целевое.

Для операции добавления в случае неуспешного поиска выполняются следующие действия:

- если поиск завершился на внутреннем узле, то соответствующему указателю на дочерний узел, равному NULL, присваивается ссылка на новый листовой узел, содержащий целевые ключ и значение;
- если поиск завершился на листовом узле w, то
  1) для родителя узла w указатель на узел w заменяется указателем на новый внутренний узел u с равными NULL указателями (если узел w является корнем дерева, то узел u становится новым корнем дерева);
  2) рассматриваются числа k и $\hat{k}$, образованные разрядами ключа узла w и целевого ключа из диапазона $[i \cdot L, (i+1) \cdot L - 1]$ соответственно, где i - длина пути от корня до узла u;
  3) для узла u указатель на k-й дочерний узел заменяется указателем на узел w;
  4) если $\hat{k} \neq k$, то
     4.1) создается новый листовой узел r, содержащий целевые ключ и значение;
     4.2) для узла u указатель на $\hat{k}$-й дочерний узел заменяется указателем на узел r;
     4.3) происходит завершение операции модификации (выход из цикла);
  5) происходит переход к п. 1.

Для операции удаления в случае успешного поиска выполняются следующие действия:

1) листовой узел, содержащий целевой ключ, удаляется из дерева; если удаленный узел являлся корневым, происходит завершение операции модификации;
2) если родитель удаленного узла содержит один дочерний узел, и этот узел является листовым, то происходит процесс поднятия данного листового узла вверх по дереву с замещением им родителей до тех пор, пока либо родитель этого листового узла не будет иметь не менее двух дочерних узлов, либо поднимаемый листовой узел не станет корнем дерева.

Высота дерева: $\left\lceil \frac{K}{L} \right\rceil$ в худшем.

Количество внутренних узлов: верхние оценки - $\left(\left\lceil \frac{K}{L} \right\rceil - \left\lfloor log_R \left(\frac{n}{2} \right) \right\rfloor + 1 \right) \cdot \frac{n}{2}$, $\left\lceil \frac{K}{L} \right\rceil \cdot \frac{n}{2}$, в среднем - $\frac{1.44}{L} \cdot n$.

Количество листовых узлов: n.

Размер памяти для хранения внутреннего узла (в битах): верхняя оценка - $R \cdot \hat{P}$, где $\hat{P} = \left\lceil log_2 \left(\left\lceil \frac{K}{L} \right\rceil \cdot \frac{N}{2} + N + 1 \right) \right\rceil$ - размер указателя на узел (в битах).

Размер памяти для хранения листового узла (в битах): $K + V$.

Размер памяти для хранения всей структуры (в битах).

Верхние оценки - $\left(K + V + \frac{R \cdot \hat{P}}{2} \cdot \left(\left\lceil \frac{K}{L} \right\rceil - \left\lfloor log_R \left(\frac{N}{2} \right) \right\rfloor + 1 \right) \right) \cdot N$, $\left(K + V + \frac{R \cdot \hat{P}}{2} \cdot \left\lceil \frac{K}{L} \right\rceil \right) \cdot N$. Размер может быть уменьшен до $\left(K + V + C \cdot \frac{R \cdot P}{L} \right) \cdot N$, где C - небольшое число, соразмерное 2, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным.

Размер памяти, необходимый для хранения внутренних узлов многопутевого trie-дерева, примерно в $\frac{R}{2 \cdot L}$ раз превышает аналогичный размер для обычного trie-дерева с тем же числом узлов.

Время поиска: $O(log_R(n))$ в среднем, $O\left(\left\lceil \frac{K}{L} \right\rceil \right)$ в худшем.

Время модификации: $O(R \cdot log_R(n))$ в среднем, $O\left(R \cdot \left\lceil \frac{K}{L} \right\rceil \right)$ в худшем.

Количество обращений к памяти при поиске: $O(log_R(n))$ в среднем, $\left\lceil \frac{K}{L} \right\rceil + 2$ в худшем (при условии, что за 1 обращение может быть считан фрагмент размера $max(\hat{P}, K, V)$ бит).

Размер считываемой области памяти при поиске (в битах): $\hat{P} \cdot O(log_R(n))$ в среднем, $\hat{P} \cdot \left\lceil \frac{K}{L} \right\rceil + K + V$ в худшем.

Размер обновляемой области памяти при модификации (в битах).

При добавлении записи с присутствующим ключом - V. При добавлении записи с отсутствующим ключом - $R \cdot \hat{P} \cdot \left\lceil \frac{K}{L} \right\rceil + K + V$ в худшем. При удалении записи с отсутствующим ключом - 0. При удалении записи с присутствующим ключом - $\hat{P}$.

Возможность разбиения памяти.

Внутренние узлы и листовые узлы, как следует из описания структуры, хранятся в отдельных областях памяти. Хранение ключей и значений для листовых узлов также можно разделить. Помимо этого, аналогично дереву цифрового поиска и trie-дереву, возможно разбиение множества всех внутренних узлов на блоки в соответствием с уровнем узла (в данном случае под уровнем понимается расстояние от корня) с хранением различных блоков в независимых областях памяти.

Если количество блоков совпадает с высотой дерева, т.е. различным уровням соответствуют различные блоки, то для хранения l-го блока, состоящего из внутренних узлов, расположенных на расстоянии l от корня ($l \in \{0, ..., \left\lceil \frac{K}{L} \right\rceil - 1\}$), достаточно $R \cdot \hat{P} \cdot \min\left(R^l, \frac{N}{2} \right)$ бит. Для хранения всех блоков достаточно $\frac{R \cdot \hat{P}}{2} \cdot \left(\left\lceil \frac{K}{L} \right\rceil - \left\lfloor log_R \left(\frac{N}{2} \right) \right\rfloor + 1 \right) \cdot N$ бит, а всей структуры - $\left(K + V + \frac{R \cdot \hat{P}}{2} \cdot \left(\left\lceil \frac{K}{L} \right\rceil - \left\lfloor log_R \left(\frac{N}{2} \right) \right\rfloor + 1 \right) \right) \cdot N$ бит, при этом данная оценка совпадает с верхней оценкой в случае отсутствия разбиения. Размер памяти не получится уменьшить при хранении в одном блоке промежуточных узлов нескольких соседних уровней. Однако, как и в случаях дерева цифрового поиска и trie-дерева, размер памяти можно сократить, если отказаться от требования га-

рантированного вмещения в таблицу классификации N записей, заменив его требованием вмещения N записей в типичных случаях или с большой вероятностью, но без строгой гарантии.

Резюме.

Многопутевое trie-дерево по сравнению с обычным trie-деревом обладает меньшей высотой и обеспечивает более быстрый поиск. Это достигается за счет большего размера памяти, отводимого под хранение внутренних узлов. Основной проблемой данной реализации, которая может привести к существенным потерям памяти, является проблема пустых ссылок - возможное наличие значительного числа узлов, в которых почти все указатели равны NULL. Размер памяти, необходимый для хранения внутренних узлов многопутевого trie-дерева, примерно в $\frac{R}{2 \cdot L}$ раз превышает аналогичный размер для обычного trie-дерева с тем же числом узлов.

Как и в случае trie-дерева, данная реализация поддерживает возможность разбиения памяти на несколько независимых областей. Необходимый суммарный размер памяти как в случае разбиения, так и без него, одинаков. Его можно понизить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным. Без данного упрощения реализация является заведомо неэффективной по памяти при больших N и K.

3.3.5. Сравнение различных структур поразрядного поиска

В данном параграфе был проведен обзор наиболее распространенных структур поразрядного поиска (далее - СПП): дерева цифрового поиска, trie-дерева, Patricia-дерева и многопутевого trie-дерева. Рассмотренные структуры поразрядного поиска похожи на сбалансированные деревья бинарного поиска (далее - СДБП), поскольку для обоих классов структур

- реализация представляется в виде дерева;
- операция поиска по ключу предполагает один проход по дереву от корня к листьям;
- наихудшие оценки количества обращений к памяти при поиске/модификации и времени поиска/модификации пропорциональны высоте дерева;
- в наихудшем случае высота дерева пропорциональна $log_2(N)$ (для СДБП) или K (для СПП), и часто эти величины являются соразмерными.

Однако данные классы структур обладают следующими существенными отличиями:

- при поиске в СДБП на каждом шаге проводится операция полного сравнения ключей, а в СПП - лишь операция сравнение на равенство одного или нескольких разрядов ключей (за исключением дерева цифрового поиска, в котором на каждом шаге дополнительно проводится операция полного сравнения ключей на равенство), таким образом СПП можно считать менее требовательными к вычислительным ресурсам;
- если величина $2 \cdot log_2(N)$ сильно меньше K, то СДБП обладают лучшими гарантированными оценками по высоте дерева и времени поиска по сравнению с первыми тремя структурами СПП (без учета многопутевого trie-дерева); в случае $K < 1.45 \cdot log_2(N)$ аналогичными преимуществами уже обладают СПП.

В таблице 3 проводится сравнение основных характеристик рассмотренных реализаций СПП. Дерево цифрового поиска и Patricia-дерево требуют небольшого размера памяти, соразмерного СДБП. Преимущество Patricia-дерева перед деревом цифрового поиска заключается в отсутствии

необходимости полного сравнения ключей на каждом шаге алгоритма поиска, недостаток - необходимость хранения в узлах номера проверяемого при поиске разряда и более сложная реализация операции модификации.

Trie-дерево является базовой структурой и не предоставляет каких-либо существенных преимуществ перед своей улучшенной версией в виде Patricia-дерева. Основная проблема trie-дерева - большой размер памяти, необходимый для гарантированного хранения N записей. Этим же недостатком обладает многопутевое trie-дерево порядка R, для которого размер необходимой памяти еще дополнительно увеличивается за счет хранения R указателей в каждом внутреннем узле. Размер памяти для данных двух структур можно понизить, если отказаться от требования гарантированного вмещения в таблицу классификации N записей, заменив это требование более мягким вероятностным. Без данного упрощения эти реализации является заведомо неэффективными по памяти при больших N и K. Преимуществом многопутевого trie-дерева перед остальными тремя СПП является меньшая высота, а значит меньшее количество обращений к памяти при поиске и более быстрый поиск.

СПП предоставляют возможность разбиения множества всех узлов на блоки в соответствием с уровнем узла (в данном случае под уровнем понимается расстояние от корня) с хранением различных блоков в независимых областях памяти, однако для дерева цифрового поиска и Patricia-дерева это достигается путем кратного увеличения суммарного размера памяти. Избежать значительного увеличения данного размера можно либо за счет хранения в одном блоке нескольких соседних уровней, либо за счет отказа от требования гарантированного вмещения в таблицу классификации N записей с заменой его на требование вмещения N записей в типичных случаях или с большой вероятностью, но без строгой гарантии.

Таблица 3. Основные характеристики различных структур поразрядного поиска

Структура	**Высота**	**Размер памяти (в битах)**	**Время поиска/модификации**
Дерево цифрового поиска	$\leq K$	$(K+V+2P)\cdot N$	$O(log_2(n))$ в среднем, $O(K)$ в худшем
Trie-дерево	$\leq K$	$\leq (K+V+\hat{P}\cdot(K-log_2(N)+3))\cdot N$, $\leq \left(K+V+K\cdot\hat{P}\right)\cdot N$, где $\hat{P}=\left\lceil log_2\left(\frac{K\cdot N}{2}+N+1\right)\right\rceil$	$O(log_2(n))$ в среднем, $O(K)$ в худшем
Patricia-дерево	$\leq K$	$(K+V+2P+Q)\cdot N$, где $Q=\lceil log_2(K)\rceil$	$O(log_2(n))$ в среднем, $O(K)$ в худшем
Многопутевое trie-дерево порядка $R=2^L$	$\leq \left\lceil\frac{K}{L}\right\rceil$	$\leq \left(K+V+\frac{R\cdot\hat{P}}{2}\cdot\left(\left\lceil\frac{K}{L}\right\rceil-\left\lfloor log_R\left(\frac{N}{2}\right)\right\rfloor+1\right)\right)\cdot N$, $\leq \left(K+V+\frac{R\cdot\hat{P}}{2}\cdot\left\lceil\frac{K}{L}\right\rceil\right)\cdot N$, где $\hat{P}=\left\lceil log_2\left(\left\lceil\frac{K}{L}\right\rceil\cdot\frac{N}{2}+N+1\right)\right\rceil$	поиск: $O(log_R(n))$ в среднем, $O\left(\left\lceil\frac{K}{L}\right\rceil\right)$ в худшем; модификация: $O(R\cdot log_R(n))$ в среднем, $O\left(R\cdot\left\lceil\frac{K}{L}\right\rceil\right)$ в худшем

Список литературы

[1] Седжвик Р. Алгоритмы на C++.: Пер. с англ. - СПб.: ООО «Диалектика», 2019. - 1056 с.: ил. - Парал. тит. англ. ISBN: 978-5-907144-21-7 (рус.)

[2] S. Kumar, J. Turner and P. Crowley, «Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking», IEEE INFOCOM 2008 - The 27th Conference on Computer Communications, Phoenix, AZ, USA, 2008, pp. 101-105, doi: 10.1109/INFOCOM.2008.29.

[3] M.V. Ramakrishna, E. Fu and E. Bahcekapili, A Performance Study of Hashing Functions for Hardware Applications, East Lansing, MI: Michigan State University, Department of Computer Science. 1994.