**Вступ**

Дерева є структури даних, в яких реалізовані операції над динамічними множинами. З таких операцій хотілося б виділити - пошук елемента, пошук мінімального (максимального) елемента, вставка, видалення, перехід до батькові чи матері, перехід до дитини. Таким чином, дерево може використовуватися і як звичайний словник, і як черга з пріоритетами.

Основні операції в деревах виконуються за час пропорційне його висоті. Збалансовані дерева мінімізують свою висоту (наприклад, висота бінарного збалансованого дерева з n вузлами дорівнює log n). Більшість знайоме з такими збалансованими деревами, як «червоно-чорне дерево», «AVL-дерево», «Декартово дерево», тому не будемо заглиблюватися.

У чому ж проблема цих стандартних дерев пошуку? Розглянемо величезну базу даних, яка представлена ​​у вигляді одного зі згаданих дерев. Очевидно, що ми не можемо зберігати все це дерево в оперативній пам'яті => в ній зберігаємо лише частина інформації, решта ж зберігається на сторонньому носії (припустимо, на жорсткому диску, швидкість доступу до якого набагато повільніше). Такі дерева як червоно-чорне або Декартово вимагатимуть від нас log n звернень до стороннього носія. При великих n це дуже багато. Якраз цю проблему і покликані вирішити B-дерева!

B-дерева також є збалансовані дерева, тому час виконання стандартних операцій в них пропорційно висоті. Але, на відміну від інших дерев, вони створені спеціально для ефективної роботи з дисковою пам'яттю (в попередньому прикладі - стороннім носієм), а точніше - вони мінімізують звернення типу введення-виведення.

**Структура**

При побудові B-дерева застосовується фактор t, який називається мінімальним ступенем. Кожен вузол, крім кореневого, повинен мати, як мінімум t - 1, і не більше 2t - 1 ключів. Позначається n [x] - кількість ключів у вузлі x.

Ключі в вузлі зберігаються в неубутних порядку. Якщо x не є листом, то він має n [x] + 1 дітей. Якщо занумерувати ключі в вузлі x, як k [i], а дітей c [i], то для будь-якого ключа в поддереве з коренем c [i] (нехай k1), виконується наступна нерівність - k [i-1] ≤k1≤ k [i] (для c [0]: k [i-1] = -∞, а для c [n [x]]: k [i] = + ∞). Таким чином, ключі вузла задають діапазон для ключів їхніх дітей.

Все листя B-дерева повинні бути розташовані на одній висоті, яка і є висотою дерева. Висота B-дерева з n ≥ 1 вузлами і мінімальним ступенем t ≥ 2 не перевищує logt (n + 1). Це дуже важливе твердження (чому - ми зрозуміємо трохи пізніше)!

h ≤ logt ((n + 1) / 2) - логарифм за основою t.

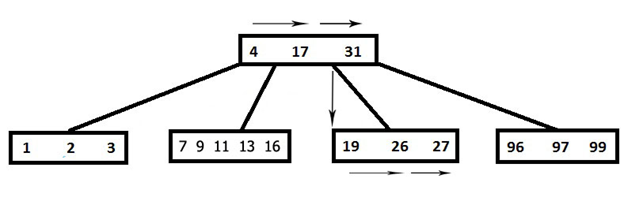
Операції, здійсненні з B-деревом

Як згадувалося вище, в B-дереві виконуються всі стандартні операції з пошуку, вставці, видалення і т.д.

**Пошук**

Пошук в B-дереві дуже схожий з пошуком в бінарному дереві, тільки тут ми повинні зробити вибір шляху до нащадка не з 2 варіантів, а з декількох. В іншому - ніяких відмінностей. На малюнку нижче показаний пошук ключа 27. Пояснимо ілюстрацію (і відповідно стандартний алгоритм пошуку):

* Йдемо по ключам кореня, поки менше необхідного. В даному випадку дійшли до 31.
* Спускаємося до дитини, який знаходиться лівіше цього ключа.
* Йдемо по ключам нового вузла, поки менше 27. В даному випадку - знайшли 27 і зупинилися.

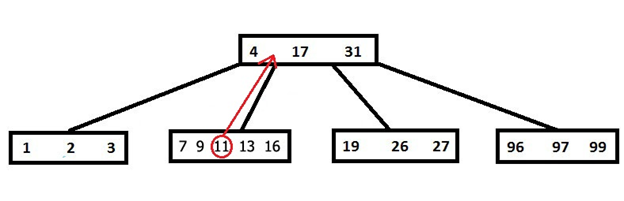


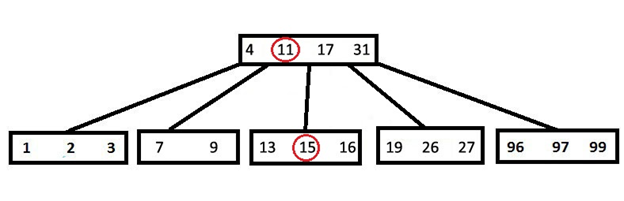
Операція пошуку виконується за час O (t logt n), де t - мінімальна ступінь. Важливо тут, що дискових операцій ми здійснюємо лише O (logt n)!

**Додавання**

На відміну від пошуку, операція додавання істотно складніше, ніж в бінарному дереві, так як просто створити новий лист і вставити туди ключ не можна, оскільки будуть порушуватися властивості B-дерева. Також вставити ключ в уже заповнений лист неможливо => необхідна операція розбиття вузла на 2. Якщо лист був заповнений, то в ньому знаходилося 2t-1 ключів => розбиваємо на 2 по t-1, а середній елемент (для якого t-1 перших ключів менше його, а t-1 останніх більше) переміщається в батьківський вузол. Відповідно, якщо батьківський вузол також був заповнений - то нам знову доводиться розбивати. І так далі до кореня (якщо розбивається корінь - то з'являється новий корінь і глибина дерева збільшується). Як і у випадку звичайних бінарних дерев, вставка здійснюється за один прохід від кореня до листа. На кожній ітерації (в пошуках позиції для нового ключа - від кореня до листа) ми розбиваємо всі заповнені вузли, через які проходимо (в тому числі лист). Таким чином, якщо в результаті для вставки потрібно розбити якийсь вузол - ми впевнені в тому, що його батько не заповнений!

На малюнку нижче проілюстровано той же дерево, що і в пошуку (t = 3). Тільки тепер додаємо ключ "15". У пошуках позиції для нового ключа ми натикаємося на заповнений вузол (7, 9, 11, 13, 16). Дотримуючись алгоритму, розбиваємо його - при цьому «11» переходить в батьківський вузол, а вихідний розбивається на 2. Далі ключ "15" вставляється в другій «відколовся» вузол. Всі властивості B-дерева зберігаються!





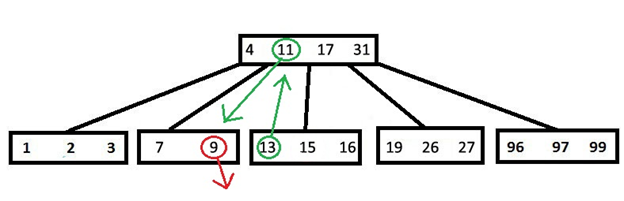
Операція додавання відбувається також за час O (t logt n). Важливо знову ж, що дискових операцій ми виконуємо лише O (h), де h - висота дерева.

**Bидалення**

Видалення ключа з B-дерева ще більш громіздкий і складний процес, ніж вставка. Це пов'язано з тим, що видалення з внутрішнього вузла вимагає перебудови дерева в цілому. Аналогічно вставці необхідно перевіряти, що ми зберігаємо властивості B-дерева, тільки в даному випадку потрібно відстежувати, коли ключів t-1 (тобто, якщо з цього вузла видалити ключ - то вузол не зможе існувати). Розглянемо алгоритм видалення:

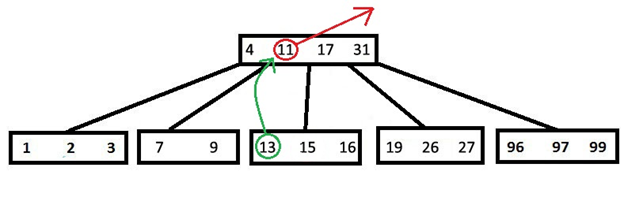
1) Якщо видалення відбувається з листа, то необхідно перевірити, скільки ключів знаходиться в ньому. Якщо більше t-1, то просто видаляємо і більше нічого робити не потрібно. Інакше, якщо існує сусідній лист (що знаходиться поруч з ним і має такого ж батька), який містить більше t-1 ключа, то виберемо ключ з цього сусіда, який є роздільником між рештою ключами вузла-сусіда і вихідного вузла (тобто не більше всіх з однієї групи і не найменше з іншого). Нехай це ключ k1. Виберемо ключ k2 з вузла-батька, який є роздільником вихідного вузла та його сусіда, який ми вибрали раніше. Видалимо з вихідного вузла потрібний ключ (який необхідно було видалити), спустимо k2 в цей вузол, а замість k2 в вузлі-батьку поставимо k1. Щоб було зрозуміліше нижче представлений малюнок (рис.1), де видаляється ключ "9". Якщо ж все сусіди нашого вузла мають по t-1 ключу. То ми об'єднуємо його з будь-яким сусідом, видаляємо потрібний ключ. І той ключ з вузла-батька, який був роздільником для цих двох «колишніх» сусідів, перемістимо в наш новоутворений вузол (очевидно, він буде в ньому медианой).

Мал. 1.



2) Тепер розглянемо видалення з внутрішнього вузла x ключа k. Якщо дочірній вузол, що передує ключу k містить більше t-1 ключа, то знаходимо k1 - попередника k в поддереве цього вузла. Видаляємо його (рекурсивно запускаємо наш алгоритм). Замінюємо k у вихідному вузлі на k1. Проробляємо аналогічну роботу, якщо дочірній вузол, наступний за ключем k, має більше t-1 ключа. Якщо обидва (наступний і попередній дочірні вузли) мають по t-1 ключу, то об'єднуємо цих дітей, переносимо в них k, а далі видаляємо k з нового вузла (рекурсивно запускаємо наш алгоритм). Якщо зливаються 2 останніх нащадка кореня - то вони стають коренем, а попередній корінь звільняється. Нижче представлений малюнок (рис.2), де з кореня віддаляється «11» (випадок, коли у наступного вузла більше t-1 дитини).

Рис.2.



Операція видалення відбувається за такий же час, що і вставка O (t logt n). Та й дискових операцій потрібно всього лише O (h), де h - висота дерева.

Отже, ми переконалися в тому, що B-дерево є швидкою структурою даних (поряд з такими, як червоно-чорне, АВЛ). І ще одна важлива властивість, яке ми отримали, розглянувши стандартні операції, - автоматична підтримка властивості збалансованості - зауважимо, що ми ніде не балансуємо його спеціально.

**Бази даних**

Проаналізувавши, разом зі швидкістю виконання, кількість проведених операцій з дисковою пам'яттю, ми можемо сказати, що B-дерево безсумнівно є більш вигідною структурою даних для випадків, коли ми маємо великий обсяг інформації.

Очевидно, збільшуючи t (мінімальний ступінь), ми збільшуємо розгалуження нашого дерева, а отже зменшуємо висоту! Яке ж t вибрати? - Вибираємо відповідно до розміру оперативної пам'яті, доступної нам (тобто скільки ключів ми можемо одноразово переглядати). Зазвичай це число знаходиться в межах від 50 до 2000. Розберемося, що ж дає нам гіллястість дерева на стандартному прикладі, який використовується у всіх статтях про B-tree. Нехай у нас є мільярд ключів, і t = 1001. Тоді нам буде потрібно всього лише 3 дискові операції для пошуку будь-якого ключа! При цьому враховуємо, що корінь ми можемо зберігати постійно. Тепер видно, на скільки це мало!

Також, ми читаємо не окремі дані з різних місць, а цілими блоками. Переміщаючи вузол дерева в оперативну пам'ять, ми переміщаємо виділений блок послідовної пам'яті, тому ця операція досить швидко працює.

Відповідно, ми маємо мінімальне навантаження на сервер, і при цьому малий час очікування. Ці та інші описані переваги дозволили B-деревам стати основою для індексів, які базуються на деревах в СУБД.