

Київський національний університет імені Тараса
Шевченка Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних інформаційних систем Алгоритми та
складність

Лабораторна робота №2.2

“Реалізація В+ дерева”

Виконав студент 2-го курсу

Групи ІПС-22

Гончаренко Ілля Сергійович

Завдання:

Реалізувати B+ - дерево для комплексних чисел.

Предметна область: комп'ютерна наука та програмування.

Теорія

B+ дерево є видом самобалансуючого дерева пошуку, яке використовується для упорядкування та швидкого пошуку даних у пам'яті комп'ютера. Воно складається з вузлів, які можуть мати декілька дочірніх вузлів та значення ключів. Всі листя розташовані на одному рівні, що полегшує пошук. В B+ дереві, на відміну від B-дерева, всі записи зберігаються на рівні листових вузлів дерева; у внутрішніх вузлах зберігаються лише ключі.

Реалізація B+ дерева для комплексних чисел полягає в тому, що вузли дерева містять комплексні числа як ключі та, можливо, іншу додаткову інформацію. Вставка, видалення та пошук ефективно виконуються відповідно до правил B+ дерева, з урахуванням порядку, який визначається за модулем комплексного числа. Такий підхід дозволяє ефективно виконувати операції над відрізками комплексних чисел та швидко здійснювати пошук та зміну цих відрізків у структурі даних.

Операції B+ дерева:

- Вставка: Додає новий ключ у дерево. Після вставки дерево може потребувати перебалансування.
- Видалення: Видаляє ключ із дерева. Після видалення дерево також може потребувати перебалансування.
- Пошук: Шукає ключ у дереві та повертає відповідне значення.

Структура B+ дерева:

- Вузол: Містить ключі та вказівники на дочірні вузли або листки.
- Листок: Містить пари ключ-значення. Всі листя зв'язані у впорядкований список, що дозволяє ефективно виконувати діапазонні запити.

Опис алгоритму:

- Вставка:
 - Знаходить місце для вставки ключа.
 - Якщо вузол переповнений, він розщеплюється, і ключі розподіляються між двома новими вузлами.
- Видалення:
 - Знаходить ключ для видалення.
 - Якщо вузол стає надто маленьким, він злитий з іншим вузлом або ключі перерозподіляються.
- Пошук:
 - Починає з кореневого вузла.
 - Порівнює ключ пошуку з ключами в поточному вузлі та переходить до відповідного дочірнього вузла.

- Повторює цей процес до досягнення листа.

Опис алгоритмів функцій:

1. **insert(complex<double> key):**

- Початок вставки нового ключа у B+-дерево.
- Якщо дерево порожнє, створюється кореневий вузол з одним ключем.
- Якщо кореневий вузол заповнений, виконується розщеплення кореневого вузла.
- Якщо кореневий вузол не заповнений, викликається допоміжна функція `insertNonFull`, яка вставляє ключ у листовий вузол або викликає рекурсивно для відповідного дочірнього вузла.

2. **insertNonFull(node currentNode, complex<double> key):**

- Допоміжна функція для вставки ключа у неповний вузол (вузол, який не переповнений).
- Якщо поточний вузол є листовим, вставляється ключ у відсортований масив ключів вузла.
- Якщо поточний вузол не є листовим, шукається відповідний дочірній вузол для вставки ключа за допомогою рекурсії.

3. **display(node tNode):**

- Функція виводить ключі B-дерева у вигляді табличної структури.
- Починаємо з кореневого вузла і виводимо його ключі.
- Якщо вузол не є листовим, додаємо всі його дочірні вузли в чергу для подальшого виводу.
- Повторюємо цей процес для кожного вузла у черзі, доки черга не стане порожньою.

4. **findParent(node current, node child):**

- Функція знаходить батьківський вузол для вказаного дочірнього вузла.
- Починаємо з кореневого вузла та рекурсивно переходимо по дереву, шукаючи дочірній вузол.
- Якщо знайдено батьківський вузол, повертається його вказівник.

5. **search(const T& key, Node* node)**

- Для пошуку елемента в B+ дереві ми можемо використовувати алгоритм, подібний до пошуку в B-дереві. Ми рухаємося вниз по дереву, спускаючись через вузли за допомогою ключів досягнення листа, де знаходимо або не знаходимо елемент.

6. **remove(const T& key, Node* node)**

- Пошук відповідного вузла:
 - Починаємо пошук з кореня дерева.
 - Шукаємо відповідний вузол, що містить шуканий ключ.
 - Якщо ключ не знайдено, повертаємо помилку, оскільки елемент не існує у дереві.
- Видалення ключа з листового вузла:

Якщо знайдений вузол є листовим, просто видаляємо ключ зі списку ключів цього вузла.

Якщо після видалення кількість ключів у вузлі стала меншою за мінімально допустиму, можливе злиття з сусідніми вузлами або перерозподіл ключів.

- Видалення ключа з внутрішнього вузла:

Якщо знайдений вузол є внутрішнім, ми знаходимо наступний за ним ключ для заміни.

Замінюємо видаляний ключ цим наступним ключем.

Повторюємо операцію видалення заміненого ключа у відповідному піддереві.

- Оновлення структури дерева:

Після видалення ключа може виникнути ситуація, коли кількість ключів у вузлі стає меншою за мінімально допустиму.

У такому випадку ми можемо виконати злиття або перерозподіл ключів між вузлами, щоб забезпечити збалансованість дерева.

Складність

n - кількість вузлів

insert(complex<double> key):

1. Звичайний випадок $O(\log n)$
2. У найгіршому випадку, коли доведеться розщепити всі вузли до кореневого, складність може бути $O(\log n)$

insertNonFull(node currentNode, complex<double> key)

1. У найгіршому випадку, коли дерево має високу глибину і доведеться рекурсивно спускатися до листового вузла, складність також може бути $O(\log n)$
2. У кожному рекурсивному виклику відбувається операція перебудови частини дерева, що має амортизовану складність $O(1)$, оскільки розщеплення вузлів рідко виникає.

display(node tNode)

1. Звичайний випадок $O(1)$
2. У найгіршому випадку, коли кожен вузол має $O(1)$ дочірніх вузлів та дерево має n ключів, загальна складність може бути $O(n)$

findParent(node current, node child)

1. $O(\log n)$, оскільки вона рекурсивно переходить по дереву вгору від дочірнього вузла до його батьківського вузла.

search(const T& key, Node* node)

1. В+ дереві функція пошуку має складність $O(\log_B(n))$, де n - кількість ключів у дереві, а B - максимальна кількість ключів у вузлі (тобто, "ступінь" дерева).

remove(const T& key, Node* node)

1. Складність функції видалення в B+ дереві також $O(\log B(n))$. Однак слід врахувати, що видалення ключа може призвести до необхідності перерозподілу ключів або злиття вузлів, що може вимагати додаткового часу, особливо в разі, якщо ці операції включають переміщення ключів між вузлами.

Тестові приклади

1. Вставка одного ключа

Початковий стан B+-дерева:

root

Вставимо комплексне число $2 + 3i$:

root

|

$2 + 3i$

2. Тестовий приклад 2: Вставка кількох ключів без розщеплення

Початковий стан B+-дерева:

root

Вставимо комплексні числа $1 + 1i$, $3 + 2i$, $4 - 1i$:

root

| |

$1 + 1i$ $3 + 2i$ $4 - 1i$

3. Вставка ключів з розщепленням листя

Початковий стан B+-дерева:

root

| |

$1 + 1i$ $3 + 2i$ $4 - 1i$

Вставимо комплексне число $5 + 4i$:

```

root
| | |
1 + 1i 3 + 2i 4 - 1i 5 + 4i

```

Оскільки лист `4 - 1i` вже має максимальну кількість ключів, ми розщеплюємо його:

```

root
| | | |
1 + 1i 3 + 2i 4 - 1i 5 + 4i
           |
          4 - 1i

```

Після розщеплення листа `4 - 1i` ми вставляємо `5 + 4i` у відповідний підвузол.
Звичайно, ось детальне описання тестових прикладів для пошуку та видалення елементів у B+ дереві:

4. Тестовий приклад для пошуку елемента:

Вихідний стан B+ дерева:

```

root
| | | |
1 + 1i 3 + 2i 4 - 1i 5 + 4i
           |       |
        4 - 1i   5 + 4i

```

Пошук елемента `4 - 1i`:

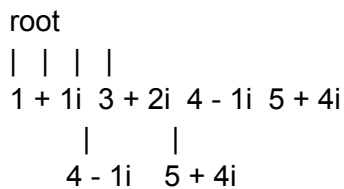
Опис кроків:

- Починаємо пошук з кореня дерева.
- Перевіряємо ключі в кореневому вузлі, обираємо відповідне піддерево для продовження пошуку.
- Переходимо до вузла, що містить ключ `4 - 1i`, та знаходимо цей ключ у його листовому вузлі.

Очікуваний результат: Під час пошуку елемента `4 - 1i` він буде знайдено у дереві.

5. Тестовий приклад для видалення елемента:

Вихідний стан B+ дерева:

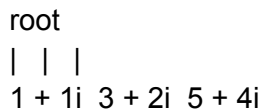


Видалення елемента `4 - 1i`:

Опис кроків:

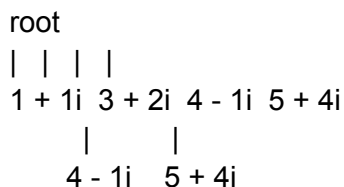
- Знаходимо вузол, що містить ключ `4 - 1i`.
- Видаляємо ключ `4 - 1i` з вузла, після чого перенумеруємо ключі та оновлюємо вузол.
- Якщо після видалення листового ключа вузол стає порожнім, можливе злиття з сусідніми вузлами.

Очікуваний результат: Після видалення елемента `4 - 1i` дерево буде оновлено і матиме такий вигляд:



6. Тестовий приклад для видалення неіснуючого елемента:

Вихідний стан B+ дерева:



4. Видалення елемента `2 + 3i`:

Опис кроків:



- Пошук елемента `2 + 3i`.
- Виявлення відсутності цього елемента у дереві.

Очікуваний результат: Елемент `2 + 3i` відсутній у дереві, тому жодних змін у структурі дерева не відбулося.

Висновок

B+ дерево є потужним і ефективним інструментом для організації та управління великими обсягами даних. Його унікальна структура дозволяє ефективно виконувати операції вставки, видалення та пошуку в середовищі з обмеженими ресурсами, такими як пам'ять та швидкість доступу до диска. B+ дерево часто використовується у базах даних для індексування, що полегшує пошук та фільтрацію даних у великих таблицях. Крім того, його широкий спектр застосувань включає в себе файлові системи, кешування та будь-яку ситуацію, де необхідно швидко та ефективно виконувати операції з даними, які відображаються в структурі дерева.

Література

- [B+ дерево — Вікіпедія](#)
-  **B+Tree Basics**
-  **10.2 B Trees and B+ Trees. How they are useful in Databases**