Дослідження Self-Balancing Binary Search Trees

Кафедра комп'ютерних наук
Український католицький університет
Львів, Україна

9 травня 2025 р.

Зміст

Анотація—У цьому звіті наведено опис та реалізацію чотирьох видів самобалансувальних дерев пошуку: B-дерево, AVL-дерево, Червоно-чорне дерево та Splay-дерево.

I. Вступ

У даному проєкті здійснено дослідження самобалансованих бінарних дерев пошуку — фундаментальних структур даних, що гарантують $\mathcal{O}(\log n)$-час на операції пошуку, вставки, видалення й обходу. Окрім теоретичного аналізу, реалізовано примітивну SQL-подібну систему бази даних, в якій індексація та зберігання даних виконуються безпосередньо на основі власних реалізацій дерев.

II. Склад команди й розподіл задач

- Падучак Маргарита — B-дерево
- Шевчук Дарина — AVL-дерево, візуалізація
- Ягода Микита — Red-Black Tree, розробка бази даних
- Лещук Роман — Splay-дерево, 2-3-дерево, рандомізоване декартове дерево, тестування, допомога з базою даних

III. Мета та завдання

А. Мета

1) Вивчити принципи самобалансування дерев пошуку.
2) Розробити й оптимізувати п'ять різновидів таких дерев.
3) Створити на їхній основі просту СУБД із SQL-інтерфейсом.
4) Провести експерименти й порівняльний аналіз ефективності.

Б. Основні задачі

- Реалізувати структури: AVL-дерево, B-дерево, Red–Black Tree, Splay-дерево, 2-3-дерево.
- Забезпечити в кожній структурі: вставку, видалення, пошук, обходи (in-order, pre-order).
- Розробити консольний інтерфейс із командами INSERT, DELETE, SELECT, UPDATE.
- Зібрати статистику часу виконання операцій на різних обсягах даних.
- Порівняти показники з відкритими СУБД (MySQL, PostgreSQL).
- Побудувати графіки та таблиці для наочності результатів.

IV. B-дерево

А. Структура даних

B-дерево — це m-арне дерево пошуку, у якому:

- Кожен вузол (окрім кореня) має щонайменше $\lceil m/2 \rceil - 1$ та щонайбільше $m - 1$ ключів.
- Внутрішній вузол з k ключами має $k + 1$ дітей.
- Всі листи знаходяться на одній глибині.

Завдяки цьому B-дерево мінімізує кількість доступів до зовнішньої пам'яті і забезпечує $\mathcal{O}(\log n)$ час операцій.

Б. Опис реалізації функцій

1) Пошук: Функція виконує рекурсивний пошук ключа k у вузлі x B-дерева. Вона переглядає ключі у вузлі зліва направо, доки не знайде ключ або не визначить, що потрібно перейти до відповідного нащадка. Якщо вузол є листком і ключ не знайдено — повертається None.

Algorithm 1 BTree_Find(k, x)

if $x =$ None then
  $x \leftarrow$ root
end if
$i \leftarrow 0$
while $i < |x.keys|$ and $k > x.keys[i][0].columns$[key_col] do
  $i \leftarrow i + 1$
end while
if $i < |x.keys|$ and $k = x.keys[i][0].columns$[key_col] then
  return i
else if $x.leaf$ then
  return None
else
  return BTree_Find(k, x.children[i])
end if

2) Вставка: Вставка в B-дерево починається з пошуку позиції для нового елемента. Якщо ключ уже існує, нове значення додається до списку відповідного вузла. Якщо корінь повний, він розділяється, і дерево збільшується вгору. Далі вставка виконується у відповідний підвузол, гарантуючи, що новий ключ завжди потрапляє в неповний вузол.

Algorithm 2 BTree_Insert(k)

```
existing ← BTree_Find_For_Insert(k)
if existing ≠ None then
    existing[0].keys[existing[1]].append(k)
    return
end if
if |root.keys| = 2t − 1 then
    temp ← new BTreeNode(leaf = False)
    prev_root ← root
    root ← temp
    temp.children ← [prev_root]
    Split_Children(temp, 0)
    Insert_Non_Full(temp, k)
else
    Insert_Non_Full(root, k)
end if
```

Algorithm 3 Insert_Non_Full(x, k)

```
Визначити позицію i для ключа k у вузлі x
if x — лист then
    Вставити k у x.keys на позицію i
else
    if дитина x.children[i] повна then
        Розбити x.children[i]
        За потреби скоригувати i
    end if
    Рекурсивно викликати Insert_Non_Full(x.children[i], k)
end if
```

3) Розбиття вузла: Розбиття (розщеплення) вузла в B-дереві виконується тоді, коли вузол переповнений. Його середній ключ піднімається в батьківський вузол, а сам вузол розділяється на два: лівий з ключами до середнього та правий з ключами після. Якщо це не лист, відповідно перерозподіляються і нащадки.

Algorithm 4 Split_Children(x, i)

```
t ← self.t
y ← x.children[i]
z ← new BTreeNode(leaf = y.leaf)
insert z у x.children на позицію i + 1
insert y.keys[t − 1] у x.keys на позицію i
z.keys ← y.keys[t : 2t]
y.keys ← y.keys[0 : t − 1]
if ¬y.leaf then
    z.children ← y.children[t : 2t]
    y.children ← y.children[0 : t]
end if
```

4) Видалення: Видалення елемента з B-дерева виконується так, щоб зберегти баланс і властивості дерева. Якщо ключ знаходиться в листі — його просто видаляють. Якщо в внутрішньому вузлі — замінюють на попередник або наступник і рекурсивно видаляють. Якщо дочірній вузол має мінімальну кількість ключів, виконується злиття або позичання ключа у сусіда для підтримання мінімального розміру.

Algorithm 5 BTree_Delete(x, k)

```
Знайти позицію i для ключа k у вузлі x
if k знайдено в x then
    if x — лист then
        Видалити k з x
    else
        Вибрати між попередником, наступником або об’єднанням:
        if ліве піддерево має ≥ t ключів then
            Замінити k на попередника й рекурсивно видалити його
        else if праве піддерево має ≥ t ключів then
            Замінити k на наступника й рекурсивно видалити його
        else
            Об’єднати обидва піддерева й рекурсивно видалити k
        end if
    end if
else
    if x не є листом then
        Переконатися, що цільовий нащадок має ≥ t ключів:
        if сусід може віддати ключ then
            Позичити ключ у сусіда
        else
            Об’єднати з одним із сусідів
        end if
        Рекурсивно викликати BTree_Delete для цього нащадка
    end if
end if
```

V. Червно–чорне дерево

А. Теоретичні основи

Червоно-чорне дерево – різновид самозбалансованого бінарного дерева пошуку, вершини якого мають додаткову властивість – колір. В червоно-чорному дереві також є додаткові правила:

1) Кожен вузол або чорний, або червоний
2) Корінь – чорний
3) Кожний nil листок – чорний
4) Якщо вершина червона – то обидва її сини чорні
5) Усі прості шляхи від будь-якої вершини до будь-якого листка в її лівому та правому піддереві містять однакову кількість чорних вершин, не враховуючи саму вершину.

Б. Опис реалізації функцій RedBlackTree

1) Вставка: Інтуїція та загальна ідея

Вставляємо новий червоний вузол як в бінарному дереві пошуку. Якщо батько нової вершини чорного кольору – жодних правил червоно-чорного дерева не порушено, а отже вставка закінчена. Інакше – порушено правило, що сини червоного вузла чорні – виправляємо порушення в піддеревах, поки загальне дерево не виконуватиме всіх правил. В кожен момент може бути порушення лише одного правила.

Algorithm 6 Insert(x)

```
Вставити новий Node червоного кольору як в бінарному дереві пошуку
if parent.color == BLACK then
    return
else
    if uncle.color == RED then
        uncle.color = parent.color = BLACK
        grandparent.color = RED
        рекурсивно виправляємо можливі порушення правил з вузла grandparent
    else
        if parent is left child and node is right child then
            Rotate parent left
            рекурсивно виправляємо порушення правил з вузла parent(який був до повроту)
        end if
        if parent is left child and node is left child then
            Rotate grandparent right
            рекурсивно виправляємо порушення правил з вузла node(який і був)
        end if
        Аналогічні перевірки та дії з іншого боку
    end if
end if
```

2) Видалення: Інтуїція та загальна ідея

Видаляємо як в бінарному дереві пошуку: якщо вершина має синів, то беремо значення(колір незмінний) найменшого з правого піддерева, або найбільшого з лівого і тепер видаляємо того, чиє значення брали. Повторюємо до тих пір, поки не маємо видалити листок. Якщо листок червоний, то нічого не робимо, оскільки жодне правило не порушене. Якщо чорний, то позначаємо цю вершину як Двічічорного(Doubleblack | DB) і після балансування видаляємо. Виправляємо до тих пір, поки DB не зникне, або не стане коренем.

Case	Check condition	Action
1	If node to be delete is a red leaf node	Just remove it from the tree
2	If DB node is root	Remove the DB and root node becomes black.
3	(a) If DB's sibling is black, and (b) DB's sibling's children are black	(a) Remove the DB (if null DB then delete the node and for other nodes remove the DB sign) (b) Make DB's sibling red. (c) If DB's parent is black, make it DB, else make it black
4	If DB's sibling is red	(a) Swap color DB's parent with DB's sibling (b) Perform rotation at parent node in the direction of DB node (c) Check which case can be applied to this new tree and perform that action
5	(a) DB's sibling is black (b) DB's sibling's child which is far from DB is black (c) DB's sibling's child which is near to DB is red	(a) Swap color of sibling with sibling's red child (b) Perform rotation at sibling node in direction opposite of DB node (c) Apply case 6
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red (remember this node)	(a) Swap color of DB's parent with DB's sibling's color (b) Perform rotation at DB's parent in direction of DB (c) Remove DB sign and make the node normal black node (d) Change colour of DB's sibling's far red child to black.

Табл. I
Red-Black Tree Deletion Cases

VI. AVL-дерево

A. Теоретичні основи

AVL-дерево — це перше самобалансувальне двійкове дерево пошуку, запропоноване Георгієм Адельсоном-Вельським і Євгеном Ландісом у 1962 році. Його основною властивістю є збереження балансу: для кожного вузла висота лівого і правого піддерева відрізняється не більше ніж на 1.

Кожна операція (вставка, видалення) зберігає цю властивість через відповідне балансування — обертання вузлів. Усі основні операції виконуються за час $O(\log n)$ у найгіршому випадку.

Б. Структура даних

- Кожен вузол зберігає об'єкт типу DataEntry.
- Містить вказівники на лівого та правого нащадка.
- Додаткове поле height визначає висоту вузла, необхідну для обчислення коефіцієнта балансу.

В. Опис реалізації

Клас AVLTree реалізує базовий інтерфейс AbstractTree та містить основні методи:

- insert(entry) — вставка елемента з балансуванням.
- erase(key) — видалення елементів за ключем.
- find(key) — пошук усіх відповідників.
- inorder(), preorder(), postorder() — різні обходи дерева.

1) Обчислення висоти та балансу:

```
def get_height(self, node):
    if not node:
        return 0
    return node.height

def get_balance(self, node):
    if not node:
        return 0
    return self.get_height(node.left) - self.get_height(node.right)
```

2) Обертання вузлів:

```
def __rotate_left(self, node):
    new_root = node.right
    node.right = new_root.left
    new_root.left = node
    update heights...
    return new_root

def __rotate_right(self, node):
    new_root = node.left
    node.left = new_root.right
    new_root.right = node
    update heights...
    return new_root
```

3) Вставка елемента:

```
def __insert(self, node, data_entry):
    if node is None:
        return new AVLTreeNode(data_entry)

    if key < node_key:
        node.left = insert(node.left, data_entry)
    elif key > node_key:
        node.right = insert(node.right, data_entry)
    else:
        node.data.append(data_entry)
        return node

    update height and balance
    perform rotations if needed
    return node
```

4) Видалення елемента: Видалення враховує всі стандартні випадки: вузол без дітей, з одним або двома, з наступним балансуванням.

```
def __delete(self, node, key):
    if node is None:
        return None

    # locate and delete node
    # find successor if necessary
    # update height and balance
    # perform rotations
    return node
```

Г. Переваги

- Гарантована логарифмічна глибина.
- Підходить для частих вставок/видалень.

Д. Недоліки

- Ускладнена реалізація.
- Більше обчислень при вставці та видаленні в порівнянні з незбалансованими деревами.

VII. Splay-дерево

A. Теоретичні основи

Splay-дерево - вид самозбалансованого бінаргоно дерева, де вершина не тримає жодної додаткової інформації для балансування, натомість воно відбувається під час підняття елемента до кореня дерева під час пошуку. Тому це дерево не має строгих гарантій на одиночні операції вставки/пошуку/видалення, бо може вироджуватися в ланцюг, але гарантує, що в середньому операції виконуватимуться за O(log(N)). Це означає, що навіть якщо дерево виродилося в ланцюг, то потім за кілька лінійних за часом виконання операцій воно збалансується, а щоб виродити його в ланцюг, потрібно зробити стільки операцій вставки, якою є довжина ланцюга. Саме тому операції, які працюватимуть за лінійний час у такому дереві траплятимуться дуже рідко.

Б. Як саме дерево балансується

Основою балансування дерева є метод splay, який підіймає дану вершину до кореня дерева. Це підняття, на відміну від інших дерев, виконується в два етапи, залежно від позиції поточної вершини не лише відносно батька, але й відносно прабатька. Це дозволяє поособливому здійснювати підйом, коли і батько відносно прабатька, і вершина відносно батька знаходиться одночасно в лівому або правому піддереві: тоді можна здійснювати повороти не знизу вверх, а зверху вниз, що сприяє балансуванню дерева.

В. Вставка елемента

Спершу ми просто вставляємо елемент як у звичайне бінарне дерево, а тоді викликаємо на цьому елементі метод splay. Таким чином, щойно вставлений елемент опиняється в корені дерева.

Г. Пошук елемента

Спершу ми шукаємо елемент як у звичайному бінарному дерево, а тоді викликаємо на знайденому елементі метод splay, щойно знайдений елемент опиняється в корені дерева. Це також означає, що елементи, до яких часто зверталися, будуть в середньому на меншій відстані від кореня, ніж ті, до яких звертаються рідше.

Д. Видалення елемента

Спершу ми викликаємо метод пошуку, відповідно елемент, який треба видалити, опиняється у корені дерева. Тоді ми зберігаємо окремо ліве і праве піддерево (у них є всі необхідні елементи, корінь відкидаємо). Тоді здійснюємо підняття найбільшого елемента лівого піддерева: тоді найбільший елемент стане його коренем, і не матиме правого сина, бо за означенням правий син має бути більшим за поточну вершину, а поточна вершина найбільша зі всіх в лівому піддереві. Отже, можемо приєднати праве піддерево як правого сина до кореня лівого піддерева (бо кожен елемент правого піддерева більший за найбільший елемент лівого), і утвориться повноцінне дерево, в якому ми видалили елемент. Ідентично можна було б брати праве піддерево, шукати в ньому найменший елемент і приєднувати ліве, це також дозволило б коректне видалення.

VIII. Рандомізоване декартове дерево

A. Теоретичні основи

Рандомізоване декартове дерево (далі Treap) - це самозбалансоване бінарне дерево, яке водночас поєднує в собі властивості бінарного дерева та купи. В кожній вершині воно зберігає пріоритет, який визначається випадково під час вставки, і виконує ліві та праві повороти дерева (ідентичні до інших самозбалансованих бінарних дерев), аби вершини з меншим (або більшим, надалі розглядатиметься варіант саме з меншим) пріоритетом були ближче до кореня. Таким чином, складність кожної операції виходить O(log(N)) (це і є висота дерева) за рахунок рандомізованих пріоритетів, але її неможливо строго гарантувати через випадковість. Проте на практиці цей підхід виправданий, бо ймовірність того, що випадковий розподіл пріоритетів спричинить виродження дерева в ланцюг мізерна.

Б. Вставка елемента

Спершу ми просто вставляємо елемент з випадковим пріоритетом як у звичайне бінарне дерево, а тоді поступово підіймаємо його за потреби методами поворотів вліво/вправо.

В. Пошук елемента

Тут ми шукаємо елемент як у звичайному бінарному дерево, без жодниих змін.

Г. Видалення елемента

Спершу ми шукаємо необхідну вершину. Якщо в неї один син або вона взагалі листок, то видалення тривіальне - перепідвішуємо цього сина до батька поточної вершини. Інакше ми дивимось на пріоритети лівого і правого сина і виконуємо поворот так, щоб син з меншим пріоритетом став на місце поточної вершини. Після цього поточна вершина опуститься на один рівень нижче, і як максимум за log(N) таких операцій буде виконуватися попередня умова, яка робить видалення тривіальним. Це і гарантує, що видалення відбувається за log(N).

IX. Обходи дерев

У деревоподібних структурах даних існує три класичні способи обходу: прямий (preorder), симетричний (inorder) та обернений (postorder). Ці обходи важливі для аналізу структури, копіювання, серіалізації та візуалізації дерев.

A. Прямий обхід (Preorder)

Порядок відвідування: корінь → ліве піддерево → праве піддерево.

```
def preorder(self) -> list[DataEntry]:
    def __preorder_recursive(node, result):
        if node is None:
            return
        result.extend(node.data)
        __preorder_recursive(node.left, result)
        __preorder_recursive(node.right, result)
    ...
```

Приклад використання: копіювання структури дерева або побудова виразів.

Б. Симетричний обхід (Inorder)

Порядок відвідування: ліве піддерево → корінь → праве піддерево.

```
def inorder(self) -> list[DataEntry]:
    def __inorder_recursive(node, result):
        if node is None:
            return
        __inorder_recursive(node.left, result)
        result.extend(node.data)
        __inorder_recursive(node.right, result)
    ...
```

Приклад використання: виведення ключів дерева у відсортованому порядку (у випадку двійкового дерева пошуку).

B. Обернений обхід (Postorder)

Порядок відвідування: ліве піддерево → праве піддерево → корінь.

```
def postorder(self) -> list[DataEntry]:
    def __postorder_recursive(node, result):
        if node is None:
            return
        __postorder_recursive(node.left, result)
        __postorder_recursive(node.right, result)
        result.extend(node.data)
    ...
```

Приклад використання: звільнення пам'яті або видалення дерева.

Г. Порівняння

Усі три види обходу реалізуються рекурсивно та мають складність $O(n)$, де n — кількість вузлів у дереві.

Обхід	Порядок
Preorder	Корінь, Ліво, Право
Inorder	Ліво, Корінь, Право
Postorder	Ліво, Право, Корінь

Табл. II
Порівняння алгоритмів обходу дерева

X. Візуалізація продуктивності дерев

Для оцінки ефективності реалізованих дерев пошуку було розроблено скрипт тестування (visualization.py), який вимірює час виконання ключових операцій: вставка (insert), пошук (find) та видалення (erase) для різних розмірів даних.

A. Методологія тестування

Тестування проводилося на випадково згенерованих наборах даних, що складаються з об'єктів типу DataEntry. Для кожного з розмірів (від 100 до 10 000 записів) виконувались:

- серія вставок,
- серія пошуків випадкових ключів,
- серія видалень випадкових елементів (50% записів).

Кожен експеримент повторювався кілька разів для усереднення результатів.

Б. Результати візуалізації

Скрипт генерує три типи графіків:

1) Звичайні графіки: Відображають залежність часу виконання від кількості елементів:

- Insert Performance — загальний час на вставку всіх елементів.
- Find Performance — середній час на одну операцію пошуку.
- Erase Performance — середній час на одну операцію видалення.

2) Графіки з логарифмічною шкалою: Ці графіки (log-log) дозволяють краще побачити різницю в асимптотичних властивостях між реалізаціями, особливо для великих розмірів.

3) Стовпчикові діаграми: Для порівняння продуктивності різних дерев на фіксованому розмірі (наприклад, $n = 10\,000$), скрипт будує стовпчикові діаграми, де кожна група включає:

- час вставки,
- час пошуку,
- час видалення.

B. Інтерпретація результатів

- AVL-дерево демонструє стабільно хорошу продуктивність на всіх операціях завдяки суворому балансуванню.
- Splay-дерево швидке при повторному доступі до тих самих елементів, але може бути повільніше на випадкових наборах.

- Червно-чорне дерево трохи швидше на вставках і видаленнях у середньому, однак його ефективність сильно залежить від реалізації.

Г. Висновки

Візуалізація підтвердила теоретичні оцінки складності алгоритмів. Збалансовані дерева пошуку (AVL, Red-Black) забезпечують предсказувану та ефективну роботу, в той час як Splay-дерево краще підходить для сценаріїв з локальністю доступу.

XI. Розробка консольного інтерфейсу

A. Архітектура CLI

Модуль crud.py реалізує консольний інтерфейс командного рядка (CLI) для взаємодії з базою даних, що базується на самобалансувальних деревах. Він надає можливість виконувати SQL-подібні запити до даних, які зберігаються в структурах різних типів дерев.

1) Основні компоненти:

- Парсер запитів — інтерпретує SQL-подібні команди користувача.
- Модуль валідації — перевіряє коректність імен таблиць, стовпців та значень.
- Система обробки помилок — забезпечує інформативні повідомлення про помилки.
- Інтерактивний режим — дозволяє користувачу працювати через командний рядок.

Б. Підтримувані операції

Реалізовано базові SQL-операції, зокрема:

- SELECT — вибірка даних з колекцій за заданими критеріями.
- INSERT — додавання нових записів до таблиці.

В. Приклади використання

1) Вибірка даних:

```
python crud.py -t
python crud.py test_database rb -q select id name from another_table
```

2) Додавання запису:

```
python crud.py -t
python crud.py test_database avl -q insert into another_table values 5 name description
```

3) Інтерактивний режим:

```
python crud.py -t
python crud.py test_database splay -i
Query> select * from another_table
Query> insert into another_table values 10 new_item "sample descrip
```

Г. Реалізація парсера запитів

Функція parse_query аналізує аргументи командного рядка та перетворює їх у відповідні операції бази даних

Д. Вибір структури дерева

CLI надає можливість вибирати тип дерева для використання в базі даних через параметр командного рядка:

```
TreeType = get_tree_class(tree_name)
```

Підтримуються всі реалізовані типи дерев:

- AVL
- B-Tree
- Red-Black
- Splay
- Treap

E. Тестовий режим

Для спрощення тестування та демонстрації функціональності, модуль має вбудований тестовий режим, який створює приклад бази даних з тестовими таблицями та даними:

python crud.py -t

Модуль забезпечує надійну обробку помилок, що покращує досвід користувача при роботі з системою.