Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Факультет комп’ютерних наук

КУРСОВА РОБОТА

з дисципліни «Теорія алгоритмів»

Тема «Splay tree (Скошене дерево)»

Виконав студент 2 курсу
групи КС-21
Безрук Юрій Русланович
Перевірив:

доц. Щебенюк В.С. .....
доц. Олешко О.І. .....
ст. викл. Лисицький К.Є. .....

Харків – 2019

ЗМІСТ

ВСТУП

У сучасних мережах даних та пошукових пристроях ми часто стикаємося з проблемами пошуку серед великих об’ємів інформації. Для повного розуміння проблеми досить уявити, яка кількість різноманітної інформації зберігається, скажімо, на серверах пошукової системи Google. Звичайні методи пошуку, такі як лінійний, або навіть, здавалось би, більш досконалий бінарний пошук, лише на перший погляд можуть здаватися корисними, але при пошукові серед великих об’ємів даних (а мова йде по всесвітню інформацію, яка взагалі на сьогоднішній день наближається до 40 зеттабайт) вони працюють дуже і дуже повільно. Проблемою, що розглядається у цій роботі є повільний доступ до елементів баз даних. Можна поглянути на питання з іншого ракурсу і сформулювати проблему інакше, адже на практиці, у більшості мереж частіше за інші використовується невелика кількість ключів проти великого числа нечасто запитуваних. Уявіть ситуацію, коли у нас є мільйони чи мільярди ключів, і лише до деяких з них ми звертаємося часто, що дуже ймовірно у багатьох практичних програмах. Зрозуміло, що проблемою виступає потреба кожного разу витрачати один і той же час на пошук елементу незважаючи на те, що доступ до нього виконується набагато частіше за інші.

Об’єктом даної роботи є повільна процес обробки систем пошуку, зокрема й часто запитуваних елементів, яка і створює проблему швидкого пошуку елементів.

Предметом даної роботи виступає аспект вирішення проблеми, зазначеної вище, за допомогою однієї з специфічних структур даних.

Метою даної роботи є огляд проблеми швидкого пошуку, ознайомлення з структурами даних, що можуть допомогти у розв’язанні поставленої задачі та пошук *алгоритму* прискорення повторного доступу до елементів серед об’ємних мереж даних.

Алгоритм, з точки зору *теорії алгоритмів* — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій, або система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час [1].

Теорія алгоритмів – наука, що вивчає загальні властивості й закономірності алгоритмів та різноманітні формальні моделі їх подання. До завдань теорії алгоритмів відносяться формальний доказ алгоритмічної нерозв'язності завдань, асимптотичний аналіз складності алгоритмів, класифікація алгоритмів відповідно до класів складності, розробка критеріїв порівняльної оцінки якості алгоритмів та ін. [2].

Одним з напрямків вирішення проблеми може слугувати вдосконалення методу бінарного пошуку, використовуваного специфічними структурами даних – *деревами*. Для більш детального розуміння об’єкту та предмету курсової роботи слід ввести декілька термінів.

Дерево (англ. Tree) – це тип даних, що представляє собою ієрархічну деревоподібну структуру у вигляді набору пов'язаних вузлів, кожен із яких має посилання на декілька інших, що називають нащадками або дітьми. До основних понять, пов’язаних з деревом відносять:

- **Вузол** — частина дерева, у якій зберігаються дані та/або ключ, а також зв’язки (зазвичай адреси чи посилання) з дітьми (іноді, також із батьком). Кожен окремий вузол разом з його дітьми утворю *піддерево*, для якого зберігаються усі основні властивості дерева.
- **Корінь** — верхній вузол в дереві, його «початок», не має батьків, може мати або не мати дітей (у випадку їх відсутності використовують нульове посилання)
- **Дитина** (також **Нащадок**) — вузол, безпосередньо приєднаний до іншого на шляху від кореня.
- **Батько** — зворотне поняття до дитини.
- **Брати, сестри** — вузли з того ж батька.

- **Лист** (також **Зовнішній вузол**) — вузол, який не має дітей, знаходиться на кінці уявної «гілки» дерева.

Фактично, дерево є окремим випадком графу, у якому відсутні цикли.

Бінарне, або Двійкове дерево (англ. Binary tree) – це дерево, кожен вузол якого має лише двох нащадків.

Бінарне дерево пошуку (англ. Binary search tree, далі - BST) – двійкове дерево, в якому значення вузлів повинні задовольняти *умові впорядкованості*:

Нехай x — довільна вершина двійкового дерева пошуку. Якщо вершина y знаходиться в лівому піддерева вершини x, то $y < x$. Якщо y знаходиться у правому піддереві x, то $y \geq x$ [3].

Тобто усі ліві нащадки(діти, діти дітей і т.д.) менші, ніж ключ вузла, а усі праві – більші або дорівнюють їм.

Збалансоване дерево в загальному розумінні цього слова — це такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем є мінімальною. Ця властивість є важливою тому, що час виконання більшості алгоритмів на бінарних деревах пошуку пропорційний до їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних ситуаціях. Процедура зменшення (балансування) висоти дерева виконується за допомогою трансформацій, відомих як поворот дерева, в певні моменти часу (переважно при видаленні або додаванні нових елементів). До збалансованих дерев відносять, наприклад, такі: AVL-дерева, B-дерева, червоно-чорні та скошені дерева.

Збалансовані дерева є добрим кроком на шляху до прискорення процесу пошуку, але вони ніяк не вирішують питання повторного виклику. Для вдосконалення цієї механіки у 1983-му році Робертом Тарьяном та Даніелем Слейтором була запропонована ідея так званого *скошеного дерева* [4].

РОЗДІЛ 1 ОСНОВНІ ВИЗНАЧЕННЯ

1.1 Бінарні та збалансовані дерева

Бінарні дерева пошуку набагато ефективніші в операціях пошуку, аніж лінійні структури, в яких витрати часу на пошук пропорційні $O(n)$, де n — розмір масиву даних, тоді як в повному бінарному дереві цей час пропорційний в середньому $O(log_2 n)$ або $O(h)$, де h — висота дерева (хоча гарантувати, що h не перевищує $\log_2 n$ можна лише для збалансованих дерев, які є ефективнішими в алгоритмах пошуку, аніж прості бінарні дерева пошуку) [3].

Бінарні дерева пошуку дозволяють зберігати впорядковану інформацію та отримувати доступ до неї, в середньому, за логарифмічний час, на відміну від лінійних структур. Збалансовані ж дерева вирішують проблему швидкого доступу до елементів ще краще, тому що, завдяки властивості саморегулювання, (яка не дозволяє дереву вироджуватися у зв'язний список, уповільнюючи доступ до елементів, що знаходяться далеко від кореню), дерево постійно підтримує рівень своєї висоти та розгалуженості. Тому збалансовані дерева добре виконують функцію доступу для мереж даних.

Але часто трапляються ситуації, коли один і той же ключ може запитуватись декілька разів. І це доволі незручно – кожного разу перебирати логарифмічний шлях до одного й того ж самого об’єкту. Задля вирішення цієї проблеми й було винайдено дерево Splay. Ця структура даних створена таким чином, що найчастіше використані елементи розташовані у дереві ближче до кореню, а той, що був використаний у попередньому запиті – у самому корені, тому маємо константну швидкість відгуку для нього при повторних викликах.

1.2 Скошене дерево

Скошене (англ. Splay tree) або косе дерево – це двійкове **дерево** пошуку, в якому підтримується властивість збалансованості. Це дерево належить класу «саморегулюючих дерев», які підтримують необхідний баланс розгалуження дерева, щоб забезпечити виконання операцій пошуку, додавання і видалення за логарифмічна час від числа збережених елементів. Дерево Splay реалізується без використання будь-яких додаткових полів в вузлах дерева (як, наприклад, в Червоно-чорних деревах або AVL-деревах, де в вершинах зберігається, відповідно, колір вершини і глибина піддерева). Замість цього використовують «розширюючі операції» (splay operation), частиною яких є обертання, що виконуються при кожному зверненні до дерева [4].

Дерево splay - це бінарне дерево пошуку. Однак є одна цікава відмінність: кожного разу, коли елемент переглядається на дереві, воно реорганізується для переміщення цього елемента до кореня, не порушуючи інваріант бінарного дерева пошуку. Якщо наступний запит на пошук призначений для того ж елемента, його можна негайно повернути. Взагалі, якщо невелика кількість елементів активно використовується, вони, як правило, знаходяться біля верхівки дерева і, таким чином, швидко знаходяться [5].

Найгірша складність часу операцій BST, таких як пошук, видалення, вставка, є O (n). Найгірший випадок трапляється при перекосі дерева у один бік, тобто коли усі елементи, що додаються у дерево, більші від кореня, або навпаки. Важливо те, що гнучкі дерева пропонують амортизовану продуктивність O (log n); послідовність операцій M на дереві відтворення n-вузла займає час O (M log n).

Ми можемо отримати найгіршу часову складність як O (log n) з AVL та Red-Black дерев, завдяки саморегулюванню кількості рівнів дерева(різниця між рівнями кожного піддерева не більше одиниці) [6].

РОЗДІЛ 2 ОСНОВНІ ОПЕРАЦІЇ

2.1 Повороти. Операція *Splay*()

Як і AVL та червоно-чорні дерева, дерево Splay також самоврівноважує BST. Основна ідея дерева splay полягає в тому, щоб повернути нещодавно доступний елемент до кореня дерева, завдяки чому нещодавно шуканий елемент стане доступним в O (1) час, якщо знову доступ до нього. Ідея полягає у використанні локальності звернення (у типовій програмі 80% доступу становлять 20% елементів) [6].

Скошене дерево використовує вдосконалені різновиди методу доступу до елементу в ньому деяких збалансованих дерев – обертання дерева. При доступі до елемента в дереві Splay, обертання дерев використовуються для переміщення його до вершини. Цей простий алгоритм може призвести до надзвичайно хороших показників на практиці.

***Splay*() (скіс/розширення)** – Основна операція скошеного дерева. Переміщує вузол в корінь за допомогою послідовного виконання деяких з шести можливих операцій: Zig, Zag, Zig-Zig, Zag-Zag, Zig-Zag та Zag-zig. Ці обертання мають два важливі наслідки: вони переміщують вузол, який рухається вгору на дереві, а також скорочують шлях до будь-яких вузлів уздовж шляху до заплетеного вузла. Останній ефект означає, що операції плетіння, як правило, роблять дерево більш врівноваженим.

2.1.1 Повороти Zig та Zag

Просте обертання дерева, яке також використовується в AVL-деревах. Виконується, коли p є коренем. Дерево повертається по ребру між *x* і *p*, коли *x* - лівий нащадок *p*. Існує лише для розбору крайнього випадку і виконується тільки один раз в кінці, коли початкова глибина вузла *x* була непарна.

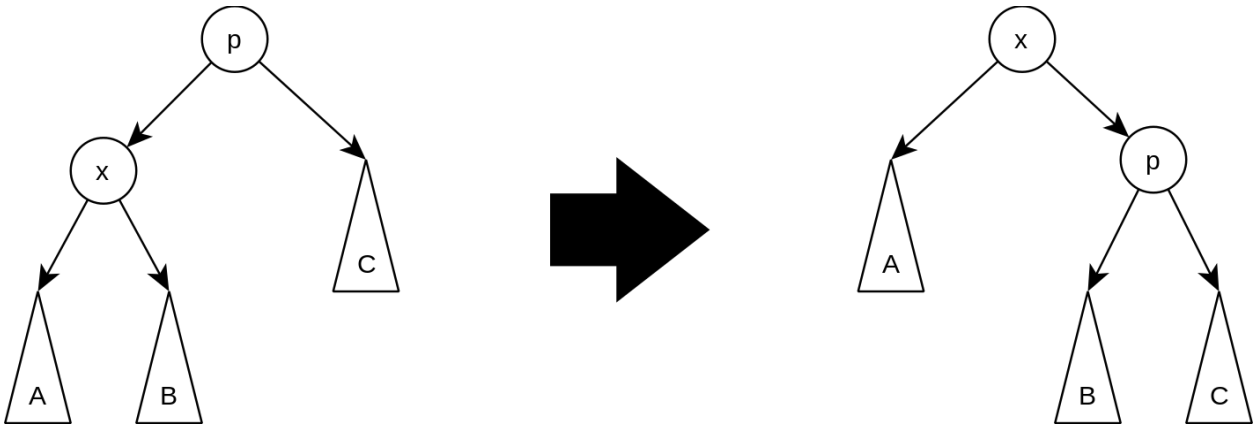

Рисунок 2.1 - Zig-обертання [4]

Zag-обертання виконується аналогічно для ситуації коли *x* - правий нащадок *p*.

2.1.2 Повороти Zig-Zig та Zag-Zag

Основна частина операції *Splay*(). На відміну від перших двох обертів, які існують лише для випадків, коли повороти закінчились, але вузол не став коренем, а є нащадком кореня, ці обертання зазвичай виконуються багато разів, наближуючи вузол до кореня. Як видно з назви, вони виконуються парами, тобто маємо подвійні оберти Zig або Zag для елемента *x* та його батька *p*. Вузли на шляху від заплетеного вузла до кореня в середньому зміщаються ближче до кореня. У випадку Zig-Zig викликаний вузол - це ліва дитина лівої дитини або права дитина дитини-правої (Zag-Zag).

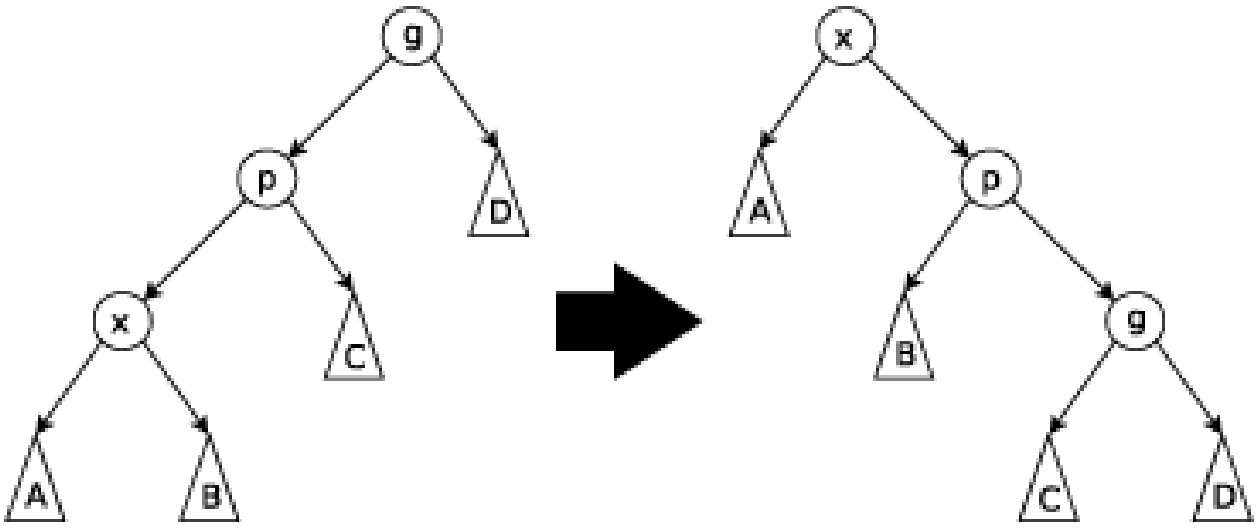

Рисунок 2.2 - Zig-Zig-обертання [4]

2.1.3 Повороти Zig-Zag та Zag-Zig

Як і попередні оберти, Zig-Zag і Zag-Zig виконуються у більшості звернень дерева, також виконуючи подвійний оберт, але тепер – різнойменний. У випадку Zig-Zag вузол x - це ліва дитина p, який є правою дитиною g, а при оберті Zag-Zig - навпаки. Обертання створюють піддерево, висота якого менша, ніж у вихідного дерева. Таким чином, це обертання покращує рівновагу дерева.

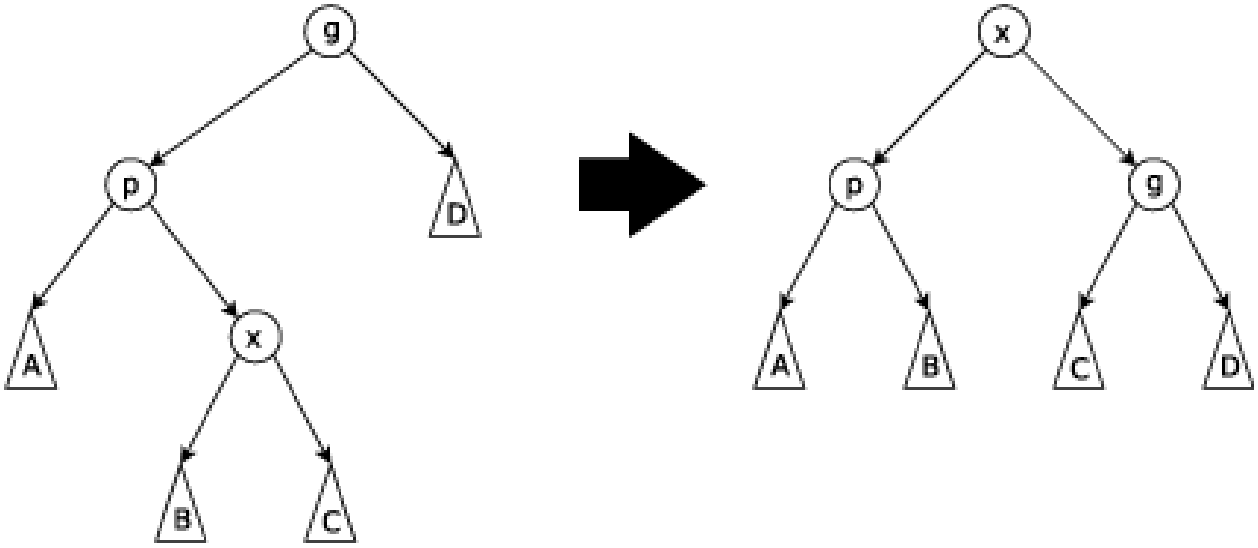

Рисунок 2.3 - Zag-Zig-обертання [4]

Тож, при звертанні до потрібного вузла, операція *Splay()*, в залежності від виникаючих ситуацій, циклічно/рекурсивно викликає повороти Zig-Zig, Zag-Zag, Zig-Zag та Zag-Zig для нього, поки він не стане коренем дерева, або нащадком кореню (така ситуація можлива тому, що повороти подвійні, а глибина дерева може бути непарною). У випадку непарної глибини, виконується додатково поворот Zig або Zag (в залежності від того, яким нащадком кореню тепер є вузол – лівим чи правим).

2.2 Структурні операції

Функція *Splay()* є вбудованою прихованою операцією скошеного дерева, що виконується на елементі при будь-якій зі звичних операцій над структурою даних типу BST, які реалізуються таким чином:

2.2.1 Додавання нового елементу

***Insert*()** – додавання елементу до дерева. Спочатку виконується звичайне додавання для бінарного дерева пошуку, а саме: починаючи з кореня, порівнюємо елемент/ключ(в залежності від програмної реалізації) з вузлом дерева. Якщо елемент, який ми бажаємо додати, більший від кореня, переходимо до правого нащадка, якщо навпаки – до лівого. Повторюємо цей виклик(тобто продовжуємо порівнювати елемент за вузлами, рухаючись у напрямку сортування дерева) доки не знайдемо вільного місця для нього (вузла, у якого не буде нащадка за потрібним нам напрямком), та встановлюємо елемент на це місце. Таким чином дерево залишається впорядкованим. Після цього виконуємо до цього елементу функцію *Splay*(), виштовхуючи його до кореню. Далі розглянемо візуалізацію прикладу. Маємо дерево з вузлами 100, 110, 200, 1000:

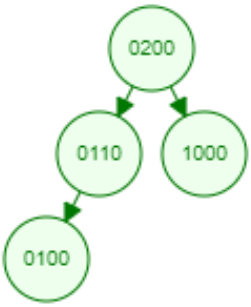

Рисунок 2.4 - Приклад дерева

Додамо до нього елемент 150. Для цього покроково порівнюємо його з вузлами дерева, починаючи від кореню:

1. 150<200: йдемо до лівого нащадка вузла 200
2. 150>110: йдемо до правого нащадка вузла 110
3. Вузол 110 не має правого нащадка: 150 стає правим нащадком вузла 110.

Елемент 150 доданий до дерева, ієрархічність та впорядкованість збережена. Візуалізація процесу наведена на рис. 2.5.

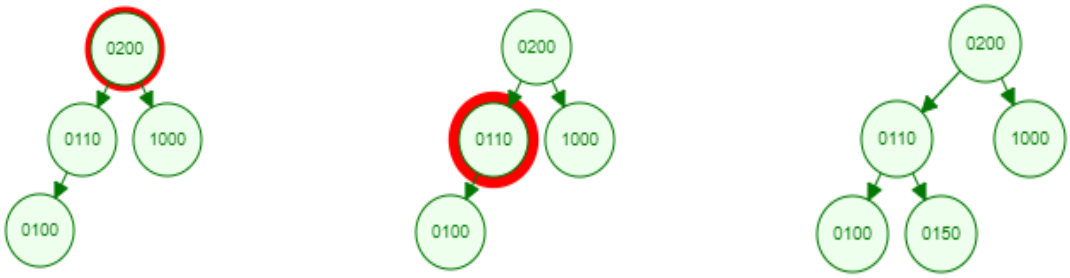

Рисунок 2.5 - Додавання елемента до дерева

Після цього застосовується операція *Splay(150)*. Оскільки 150 є правим нащадком лівого нащадка, виконується поворот Zag-Zig:

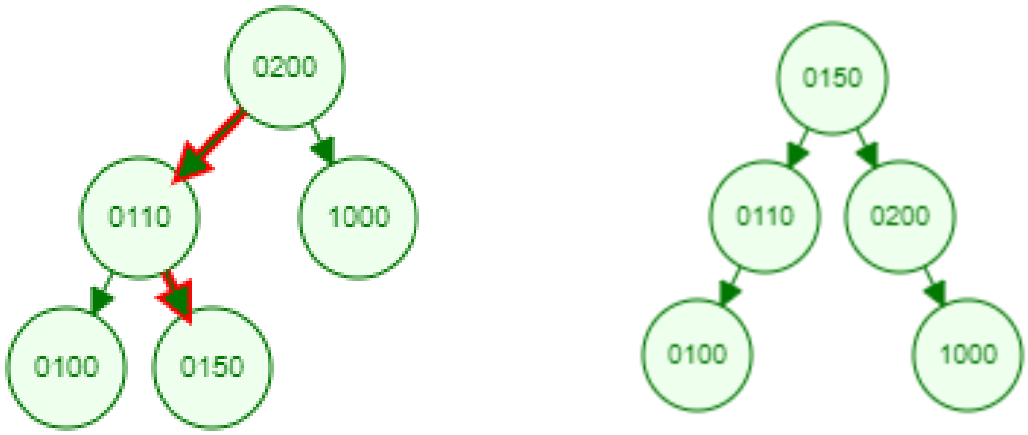

Рисунок 2.6 - Виконання операції *Splay(150)*

Оскільки 150 стало коренем дерева, додатковий поворот Zig чи Zag не потрібен (глибина дерева = 2), елемент додано до дерева, виконаний скіс до нього, операція додавання завершена.

2.2.2 Пошук елементу

***Search*()** – операція пошуку елементу у дереві. Теж спочатку виконується звичайний бінарний пошук, тобто перевірка для вузлів, починаючи від кореню, шуканий елемент більший, менший чи дорівнює йому. У разі що він більший – перевіряємо правий елемент, менший – лівий. Якщо поточний вузол дорівнює шуканому – запускаємо *Splay*() на нього. Розглянемо на прикладі такого дерева:

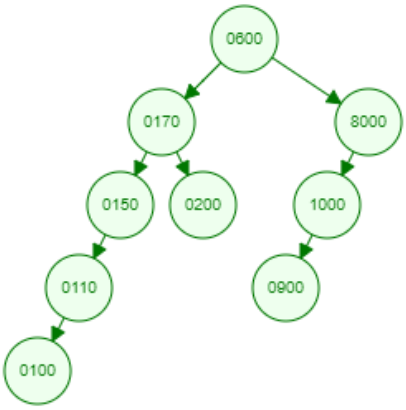

Рисунок 2.7 - Приклад дерева 2

Знайдімо вузол 100 у ньому. Пошук починається з кореню:

1. 100<600: перевіряємо лівого нащадка
2. 100<170: перевіряємо лівого нащадка
3. 100<150: перевіряємо лівого нащадка
4. 100<110: перевіряємо лівого нащадка
5. 100=100: вузол знайдено, запускаємо процедуру *Splay*(100)

Наведемо візуалізацію:

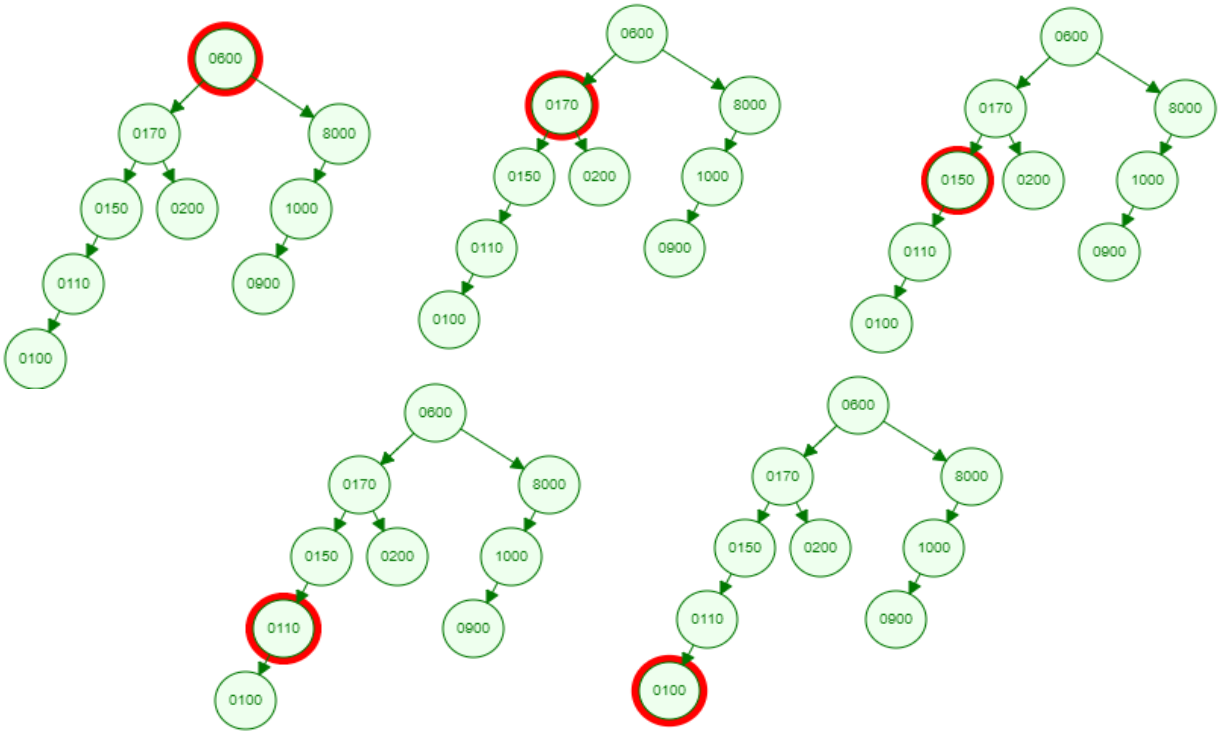

Рисунок 2.8 - Пошук елементу

Далі процедура *Splay*(): виклики подвійних обертів:

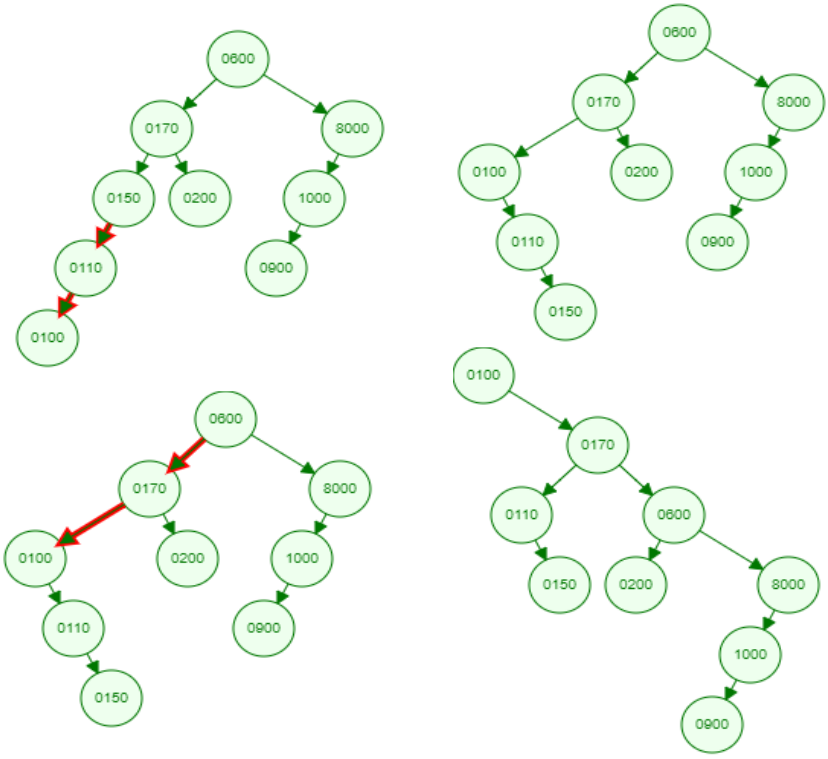

Рисунок 2.9 - операція *Splay*() при пошукові

У результаті скосу дерева, шуканий елемент став коренем дерева, завдяки чому доступ до нього при повторних викликах буде лінійно швидким.

Іноді при пошукові також використовують обхід дерева замість бінарного пошуку.

2.2.3 Видалення вузла за значенням

***Delete*() –** операція видалення вузла з дерева. А ось видалення в скошеному дереві працює дещо простіше, ніж у бінарному дереві пошуку, адже тут завдяки скосу виключаються усі складні ситуації, пов'язані з елементами, що знаходяться всередині дерева. Алгоритм такий: спочатку знаходимо вузол у дереві та виштовхуємо його до кореню (тобто операція *Search*()), після цього видаляємо елемент, маючи два дерева – піддерево лівого нащадка та піддерево правого нащадка. Застосовуємо до них злиття дерев функцією *Merge*() (див. пункт 2.2.4), у результаті якого максимальний елемент лівого піддерева стає новим коренем, а праве піддерево – його правим нащадком, оскільки усі елементи правого піддерева більші за будь які ліві. Візуалізуємо це так: операція пошуку на елемент, який ми бажаємо видалити:

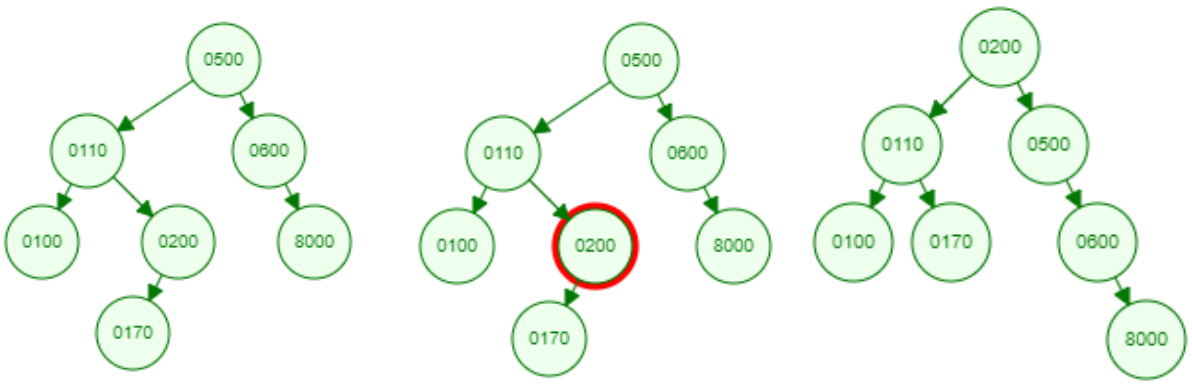

Рисунок 2.10 - Пошук при видаленні

Далі по алгоритму – видалення кореню, що утворює два піддерева:

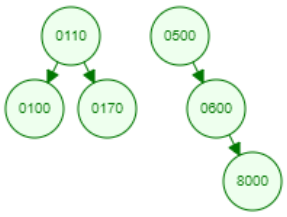

Рисунок 2.11 - Видалення елементу

А потім – операція злиття *Merge*() (більш детально про неї у наступному підрозділі), що поєднує два отриманих піддерева в одне.

Рисунок 2.12 - злиття дерев після видалення

2.2.4 Злиття дерев

***Merge*()** – операція, що виконує злиття двох дерев *T1* (виконуючого виклик функції) і *T2* (ціль приєднання), таких, що всі елементи *T1* менші за будь-який елемент *T2*. Для злиття потрібно, щоб у кореню дерева *T1* не було правого нащадка. Виконується це таким чином: у дерева *T1* знаходиться максимальний елемент (у крайній правій позиції, отримуємо шляхом постійного звернення до правих нащадків), після чого до нього виконується *Splay*(). Оскільки елемент максимальний, правих нащадків у нього не може бути, тому скосивши до нього дерево, отримаємо корінь з вільним правим нащадком, куди і приєднуємо дерево *T2*. На минулому прикладі з видаленням це мало вигляд:

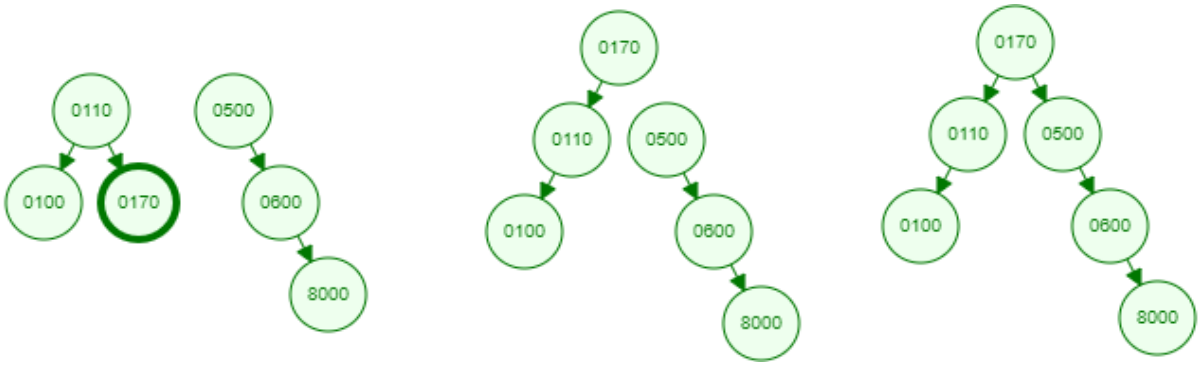

Рисунок 2.13 - Злиття дерев

РОЗДІЛ 3 РЕАЛІЗАЦІЯ

3.1 Огляд класу

Перейдемо до реалізації дерева. Розглянемо її на мові програмування Java. Дерево буде розраховане на зберігання інформації у вигляді цілих чисел, без використання поширеного методу «ключ-значення», задля полегшення сприйняття. Кожен вузол дерева представимо у вигляді повноцінного об’єкту класу, з повним набором можливостей усього дерева, тобто кожен вузол – це самостійне дерево. Каркас класу з початковими необхідними полями та шаблонними методами, що подаються без роз’яснень зображено на рис. 3.1.

```
public class SplayTree {
    Integer data=null;
    SplayTree left=null;
    SplayTree right=null;
    SplayTree parent=null;

    SplayTree() {}
    SplayTree(Integer elem){
        data=elem;
    }
    public void print() {           //друк
    public void printAsTree() {         //ієрархічний друк
    private int depth(SplayTree tr, int dep_count){         //глибина дерева
    private void print_symbol(int n) {      //друк заповнювачу
}
```

Рисунок 3.1 - Каркас класу

Детальніше про поля та методи:

- *data* – поле зберігання інформації.
- *left* – посилання на лівого нащадка.
- *right* – посилання на правого нащадка.
- *parent* – посилання на батька. Дорівнює null лише у кореня.
- Конструктори: за замовчуванням – пустий (залишаємо усі поля null), та перевантажений – з ініціалізацією першого елемента.

- *print*() – метод, що лінійно друкує елементи дерева за допомогою прямого обходу.
- *printAsTree*() – метод ієрархічного друку, тобто виведення всіх елементів дерева саме у вигляді дерева, рівень за рівнем.
- *depth*() – приватний службовий метод, підраховуюючий глибину дерева. Використовується у ієрархічному друці.
- *print_symbol*() – приватний службовий метод, працюючий як заповнювач при друці дерева.

  Повний лістинг класу SplayTree наведено у додатку А.

3.2 Повороти та скіс дерева

Основною внутрішньою операцією дерева є *Splay*(). Для її втілення необхідно реалізувати можливість поворотів дерева. Почнемо зі звичайного одиничного повороту – Zig. На відміну від звичних реалізацій, додамо цій процедурі можливість повороту не лише нащадка кореню, а й будь-якого вузла у дереві. Це потребує більше операцій пере адресації покажчиків, але, натомість, набагато полегшить реалізацію подвійних обертів.

```
/**функція, що виконує одиничний лівий
 * оберт вузла x щодо його батька
 * @param x - посилання на вузол,
 * до якого необхідно застосувати оберт
 * @return x - вузол після повороту
 */
private SplayTree Zig(SplayTree x) {
    SplayTree p=x.parent;       //збереження батька вузла
    if(p.parent!=null) {        //перевизначення посилання з дідуся
        if(p==p.parent.left)p.parent.left=x;
        else p.parent.right=x;
    }
    x.parent=p.parent;      //перевизначення покажчиків
    p.left=x.right;
    if(p.left!=null)p.left.parent=p;
    x.right=p;
    p.parent=x;
    return x;
}
```

Рисунок 3.2 - Функція *Zig*(*x*)

Через те, що розглядається найгірший випадок – вузол у середині дерева, потрібно пре направити 6 покажчиків: згори (від діда до батька, після повороту – до самого вузла), відповідно, з вузла до батька, зробити праве піддерево лівим нащадком колишнього батька, відповідно змінити його батька, та виконати сам поворот: замінити покажчики від сина до батька на від батька до сина (в обидві сторони). Функція повертає адресу елемента, до якого застосували оберт. Аналогічно реалізується і правий поворот – функція *Zag*(*x*) (без коментарів):

```
private SplayTree Zag(SplayTree x) {
    SplayTree p=x.parent;
    if(p.parent!=null) {
        if(p==p.parent.left)p.parent.left=x;
        else p.parent.right=x;
    }
    x.parent=p.parent;
    p.right=x.left;
    if(p.right!=null)p.right.parent=p;
    x.left=p;
    p.parent=x;
    return x;
}
```

Рисунок 3.3 - Функція *Zag*(*x*)

Тепер, так як методи *Zig*() та *Zag*() детерміновані для будь-якого вузла дерева, для виконання подвійних обертів достатньо лише двічі викликати потрібні одиничні оберти:

```
private SplayTree ZigZig(SplayTree x){
    x=Zig(x);
    return Zig(x);
}
private SplayTree ZagZag(SplayTree x){
    x=Zag(x);
    return Zag(x);
}
private SplayTree ZigZag(SplayTree x){
    x=Zig(x);
    return Zag(x);
}
private SplayTree ZagZig(SplayTree x){
    x=Zag(x);
    return Zig(x);
}
```

Рисунок 3.4 - Функції подвійних обертів

Так як усі повороти реалізовані, можемо перейти до опису операції *Splay*(). Напишемо її на базі рекурсії, через перевірку умов, чи є даний вузол певною дитиною або внуком, від чого й буде залежати, який вид повороту виконати.

```
/**функція, що скошує дерево до заданого
 * елементу, рекурсивно виконуючи оберти
 * доки елемент не стане коренем
 * @param x - посилання на вузол для скосу
 * @return - усе те ж посилання
 */
private SplayTree Splay(SplayTree x) {
    if(x.parent==null)
        return x;
    if(x.parent.parent==null) {
        if(x==x.parent.left)
            return Zig(x);
        else
            return Zag(x);
    }
    if(x==x.parent.left) {
        if(x.parent==x.parent.parent.left)
            return Splay(ZigZig(x));
        else
            return Splay(ZigZag(x));
    }
    else {
        if(x.parent==x.parent.parent.right)
            return Splay(ZagZag(x));
        else
            return Splay(ZagZig(x));
    }
}
```

Рисунок 3.5 - Функція *Splay*()

Тобто, у випадку, коли вузол x вже є коренем, повертаємо посилання на нього. Якщо вузол є лівим або правим нащадком кореню, виконуємо відповідний одиничний поворот. Слід розуміти, що ця умова виконується як при викликанні скосу на нащадка кореню, там і при виклику *Splay*() на елементі, що має непарну довжину шляху до кореня (оскільки основні повороти є подвійними, у такому випадку виникне ситуація, коли подвійний оберт вже виконати неможливо, але вузол ще не є коренем. У цьому разі і використовуються звичайні одиничні повороти).

Якщо ж вузол знаходиться у глибині дерева, потрібно звернути увагу на дві речі: яким нащадком батька p є дитина x, та яким нащадком дідуся g є

вузол *p*, і, відповідно до цієї інформації, виконати потрібний подвійний поворот.

3.3 Реалізація основних операцій

Тепер дерево може виконувати свою основну операцію – скіс, а отже, можна приступити до написання загальноприйнятих операцій. Почнемо з додавання елементу (метод *Insert*()).

```
/**метод додавання елементу до
 * дерева
 * @param elem - елемент,
 * який необхідно додати
 * @return - посилання на новий корінь
 */
public SplayTree Insert(Integer elem){
    if(elem<data){
        if(left==null){
            left=new SplayTree(elem);
            left.parent=this;
            return Splay(left);
        }
        else
            return left.Insert(elem);
    }
    else {
        if(right==null){
            right=new SplayTree(elem);
            right.parent=this;
            return Splay(right);
        }
        else
            return right.Insert(elem);
    }
}
```

Рисунок 3.6 - Метод *Insert*()

Він виконує звичайний пошук, звичний для типу BST, а саме порівняння елементу з поточним вузлом та рекурсивні виклики такого ж методу у лівого нащадка, якщо додаваний елемент менший за нього, або правого, якщо елемент, відповідно, більший (або дорівнює) поточному вузлу. Рекурсивні виклики продовжуються доти, доки не буде знайдений елемент без відповідного нащадку. Тоді додаваний елемент і стає цим нащадком, після чого до нього виконується операція *Splay*().

Операція пошуку виконується із застосуванням бінарного пошуку у дереві, який також рекурсивно порівнює елемент з вузлами, за потреби

звертаючись до нащадків. У випадку, коли елемент не знайдено, метод повертає нульове посилання (null).

```
/**метод пошуку у дереві.
 * виконує рекусивний бінарний пошук елементу,
 * після чого скошує дерево до нього
 * @param x - елемент, який потрібно
 * @return посилання на знайдений вузол,
 * або null, якщо такий елемент відсутній
 */
public SplayTree Search(Integer x) {
    if(x<data)
        if(left!=null)
            return left.Search(x);
    if(x>data)
        if(right!=null)
            return right.Search(x);
    if(x==data)
        return Splay(this);
    return null;
}
```

Рисунок 3.7 - Метод *Search*()

Для того, щоб реалізувати видалення елементу в скошеному дереві, потрібно написати метод злиття *Merge*() для двох дерев, таких що усі елементи другого більші за усі елементи першого. Працювати він буде таким чином: у викликаючого дерева знаходитиметься максимальний елемент (крайній правий), і дерево скошується до нього, у результаті чого воно перебудується таким чином, що корінь матиме лише лівого нащадка. А правим нащадком йому робимо друге дерево, оскільки всі його елементи за умовою більші за корінь. Реалізацію наведено на рис. 3.8.

Для видалення елементу, застосовуємо до нього метод *Search*(), виштовхуючи його до кореню. У разі якщо елемент не було знайдено – повертаємо null. В іншому випадку – зберігаємо його ліве та праве піддерево *T1* та *T2*, після чого видаляємо корінь. У мові програмування Java для цього достатньо лише стерти усі посилання на цей об’єкт, після чого він буде автоматично видалений через деякий час збирачем сміття. Зсередини дерева

на нього є посилання лише у його нащадків – піддерев ((*T1, T2*).*parent*), які і зануляємо, після чого використовуємо злиття піддерев методом *Merge*().

```
/**метод виконуючий видалення елементу
 * з дерева за допомогою вбудованих -
 * Search() & Merge()
 * @param x - елемент, який потрібно видалити
 * @return - посилання на новий корінь
 *       після видалення або null якщо елемент
 *       не знайдено
 */
public SplayTree Delete(Integer x) {
    SplayTree T1=Search(x);
    if (T1==null) return T1;
    SplayTree T2=T1.right;
    T1=T1.left;
    T1.parent=null;
    T2.parent=null;
    return T1.Merge(T2);
}
/**метод злиття дерева з іншим
 * @param T2 - дерево, яке потрібно
 * приєднати до поточного
 * @return посилання на корінь нового дерева
 */
public SplayTree Merge(SplayTree T2) {
    SplayTree T1=this;
    while(T1.right!=null)
        T1=T1.right;
    Splay(T1);
    T1.right=T2;
    return T1;
}
```

Рисунок 3.8 - Методи *Delete*() та *Merge*()

Тепер маємо повноцінну реалізацію скошеного дерева з усіма необхідними методами та полями. Повний код класу наведено у додатку А.

РОЗДІЛ 4 ПРАКТИЧНІ ЗАСТОСУВАННЯ

Дерево splay - це ефективна реалізація збалансованого дерева бінарного пошуку, що використовує перевагу локальності в ключах, які використовуються в вхідних запитах пошуку. Для багатьох застосувань є чудова локалізація ключів. Хороший приклад - мережевий маршрутизатор. Мережевий маршрутизатор отримує мережеві пакети з високою швидкістю від вхідних з'єднань і повинен швидко вирішити, яким вихідним проводом надсилати кожен пакет, виходячи з IP-адреси в пакеті. Роутеру потрібна велика таблиця (карта), за допомогою якої можна шукати IP-адресу та з’ясовувати, яке вихідне з'єднання використовувати. Якщо IP-адресу було використано один раз, вона, ймовірно, буде використана знову, можливо, багато разів. Скошені дерева можуть забезпечити хороші показники в цій ситуації [5]. У проекті кожен вузол дерева Splay зберігає IP-адресу веб-сайту, відповідну назву веб-сайту та кількість відвідувань, зроблених на веб-сайті. Кількість відвідувань веб-сайту збільшується щоразу, коли викликається функція пошуку для пошуку веб-сайту із вказаною IP-адресою [7].

Дерева Splay стали найбільш широко використовуваною базовою структурою даних, винайденою за останні 30 років, оскільки вони є найшвидшим типом збалансованого дерева пошуку для багатьох застосувань. Дерева Splay використовуються у Windows NT (у віртуальній пам'яті, мережах та файловій системній коді), компіляторі gcc та бібліотеці GNU C++, редакторі рядків sed, мережевих маршрутизаторах Fore Systems, найпопулярнішій реалізації Unix malloc, Linux завантажуваному ядрі модулі та в безлічі іншого програмного забезпечення [6].

ВИСНОВКИ

Таким чином, структура даних Splay Tree є прекрасним варіантом для збереження відсортованої інформації, вирішуючим проблему повторних викликів елементів. У процесі огляду одного з можливих варіантів реалізації було створено дерево, яке здатне зберігати лише цілочисельний тип даних, у цілях підвищення зрозумілості коду та описів, але, зазвичай, такі структури роблять з можливістю зберігати об’єкти будь яких порівнюваних типів, або, навіть неповрівнюваних, якщо використовувати реалізацію методом зберігання «ключ-дані». Тоді сортування даних буде відбуватися за ключем, що надає змогу поєднати дерево з процесом хешування та кодування даних, створюючи менш ресурсозатратні та захищені системи пошуку.

Однак і у цієї структури є недоліки. Перший з них – постійна витрата ресурсів пам’яті та швидкості на будь – яку дію, окрім кореню. Це виникає через те, що для навіть, здавалося б, елементарних операцій виконується велика кількість процедур, включаючи скіс з усіма його поворотами. Остаточно вирішити цю проблему для дерев неможливо, адже тільки таким чином вони досягають подібних результатів. Цю проблему можна вирішити лише використовуючи інші типи структур, або дещо вдосконалюючи алгоритми, наприклад, використовуючи ітерацію замість рекурсії.

Іншим важливим спостереженням є те, що насправді скошене дерево не є повноцінним збалансованим деревом, оскільки, на відміну від згадуваних раніше AVL та Червоно-чорних, у Splay tree немає ніяких спеціальних механізмів впорядкування вузлів або спостереження за глибиною піддерев. Збалансованість у ньому – це побічний фактор постійних поворотів, через які елементи, в основному, згуртовуються ближче до кореню. Але іноді повороти не працюють на користь балансу – наприклад, якщо постійно додавати елементи, що перевищують попередні. У цьому випадку постійний Zag-оберт буде вироджувати дерево у лінійний список, з швидкістю пошуку

неповторюваних елементів O (n). Таким чином, «збалансоване» скошене дерево у окремих ситуаціях може не виконувати взагалі ніяких балансуючих аспектів, а отже, втрачає свою цінність.

На мою думку, алгоритми балансування дерева потребують деякого вдосконалення задля того, щоб мінімізувати вплив виняткових ситуацій на швидкість пошуку та збільшити амортизаційну властивість загалом. Однією з можливих модифікацій я бачу інтегрування у дерево властивостей AVL-дерев, а саме збереження інформації про глибину піддерев та автоматичне стягування дерева, яке, з одного боку, анулює можливість виродження дерева в список, а з іншого – прискорить швидкість доступу до елементів дерева, скіс до яких давно не застосовувався.

Говорячи про великі об’єми інформації, щоб компенсувати витрати пам’яті і швидкості я пропоную підключити можливість багатопоточних звернень до структури, аби процеси балансування виконувалися у фоновому для основних операцій режимі. Багатопоточність, звичайно має бути синхронізованою, для безпечного регулювання вузлами дерева без втрат даних.

Я вважаю, що, втіливши подібні вимоги до вдосконалення, Splay Tree має перспективу стати дуже корисним об’єктом для зберігання різноманітних відсортованих пошукових даних і знадобитися у великій кількості досі не вирішених питань швидкодії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Алгоритм – Вікіпедія. // https://uk.wikipedia.org/wiki/Алгоритм. Дата звернення: 14.12.2019.
2. Теорія Алгоритмів – Знаймо. // http://znaimo.com.ua/Теорія_алгоритмів. Дата звернення: 14.12.2019.
3. Двійкове дерево пошуку – Вікіпедія. // https://uk.wikipedia.org/wiki/Двійкове_дерево_пошуку. Дата звернення: 06.12.2019.
4. Расширяющееся дерево – Википедия. // https://ru.wikipedia.org/wiki/Расширяющееся_дерево. Дата звернення: 06.12.2019.
5. Splay Trees – Cornell University – Department of Computer Science. // https://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec08-splay/rec08.html. Дата звернення: 08.12.2019.
6. Splay Tree Set 1 (Search) – GeeksforGeeks. // https://www.geeksforgeeks.org/splay-tree-set-1-insert. Дата звернення: 08.12.2019.
7. Splay-Trees - GitHub // https://github.com/srth21/Splay-Trees/blob/master/README.md. Дата звернення: 14.12.2019.

ДОДАТОК А
ПРИКЛАД КОДУ РЕАЛІЗАЦІЇ

Лістинг 1 – повна реалізація класу SplayTree

```
import java.util.ArrayDeque;
import java.util.Queue;

public class SplayTree {
	Integer data=null;
	SplayTree left=null;
	SplayTree right=null;
	SplayTree parent=null;

	SplayTree() {}
	SplayTree(Integer elem){
		data=elem;
	}
	public SplayTree Insert(Integer elem){
		if(elem<data){
		  if(left==null){
			left=new SplayTree(elem);
			left.parent=this;
			return Splay(left);
		  }
		  else
			return left.Insert(elem);
		}
		else {
		    if(right==null){
			right=new SplayTree(elem);
			right.parent=this;
			return Splay(right);
		    }
		    else
			return right.Insert(elem);
		}
	}
	public SplayTree Search(Integer x) {
		if(x<data)
			if(left!=null)
				return left.Search(x);
		if(x>data)
			if(right!=null)
				return right.Search(x);
		if(x==data)
			return Splay(this);
		return null;
	}
```

```
public SplayTree Delete(Integer x) {
      SplayTree T1=Search(x);
      if (T1==null) return T1;
      SplayTree T2=T1.right;
      T1=T1.left;
      T1.parent=null;
      T2.parent=null;
      return T1.Merge(T2);
}
public SplayTree Merge(SplayTree T2) {
      SplayTree T1=this;
      while(T1.right!=null)
            T1=T1.right;
      Splay(T1);
      T1.right=T2;
      return T1;
}
private SplayTree Zig(SplayTree x) {
      SplayTree p=x.parent;
      if(p.parent!=null) {
            if(p==p.parent.left)p.parent.left=x;
            else p.parent.right=x;
      }
      x.parent=p.parent;
      p.left=x.right;
      if(p.left!=null)p.left.parent=p;
      x.right=p;
      p.parent=x;
      return x;
}
private SplayTree Zag(SplayTree x) {
      SplayTree p=x.parent;
      if(p.parent!=null) {
            if(p==p.parent.left)p.parent.left=x;
            else p.parent.right=x;
      }
      x.parent=p.parent;
      p.right=x.left;
      if(p.right!=null)p.right.parent=p;
      x.left=p;
      p.parent=x;
      return x;
}
private SplayTree ZigZig(SplayTree x){
      x=Zig(x);
      return Zig(x);
}
private SplayTree ZagZag(SplayTree x){
      x=Zag(x);
      return Zag(x);
```

```
}
private SplayTree ZigZag(SplayTree x){
      x=Zig(x);
      return Zag(x);
}
private SplayTree ZagZig(SplayTree x){
      x=Zag(x);
      return Zig(x);
}
private SplayTree Splay(SplayTree x) {
      if(x.parent==null)
            return x;
      if(x.parent.parent==null) {
            if(x==x.parent.left)
                  return Zig(x);
            else
                  return Zag(x);
      }
      if(x==x.parent.left) {
            if(x.parent==x.parent.parent.left)
                  return Splay(ZigZig(x));
            else
                  return Splay(ZigZag(x));
      }
      else {
            if(x.parent==x.parent.parent.right)
                  return Splay(ZagZag(x));
            else
                  return Splay(ZagZig(x));
      }
}
public void print() {               //друк
      Splay(this);
      System.out.print(data+" ");
      if(left!=null)left.print();
      if(right!=null)right.print();
}
public void printAsTree() {   //ієрархічний друк
      SplayTree Null=new SplayTree();
      SplayTree Space=new SplayTree(Integer.MIN_VALUE);
      Queue<SplayTree> q= new ArrayDeque<>();
      int tabs=0;
      for(int i=0; i<depth(this,0); i++)
            tabs=2*tabs+1;
       print_symbol(tabs);
       q.add(this);
       q.add(Null);
       while(!q.isEmpty()) {
            SplayTree temp=q.poll();
            if(temp.data==null) {
                  if(q.isEmpty())
                        break;
```

```
                        else {
                              q.add(Null);
                              System.out.println();
                              tabs=(tabs-1)/2;
                                    print_symbol(tabs);
                        }
                  }
                  else {

            if(temp.data==null||temp.data==Integer.MIN_VALUE)
                  System.out.print("~");
                  else System.out.print(temp.data);

                  print_symbol(2*tabs+1);
                  if(temp.left!=null)q.add(temp.left);
                  else if(tabs>0) q.add(Space);
                  if(temp.right!=null)q.add(temp.right);
                  else if(tabs>0) q.add(Space);
                  }
            }
            System.out.println();
      }
      private int depth(SplayTree tr, int dep_count){
        if((tr.left==null)&&(tr.right==null))
            return dep_count;
        if(tr.left==null)
            return depth(tr.right, dep_count+1);
        if(tr.right==null)
            return depth(tr.left, dep_count+1);
        else return Math.max(depth(tr.left, dep_count+1),
depth(tr.right, dep_count+1));
      }
      private void print_symbol(int n) {
            for(int i=0; i<n; i++)
                  System.out.print(" ");
      }
}
```