Splay Trees

Каленик В. О., ПЗПІ-22-6

Що це таке?

Splay дерево — це саморегулююча структура даних бінарного дерева пошуку, що означає, що структура дерева динамічно коригується на основі доступу або вставки елементів. Іншими словами, дерево автоматично переорганізовує себе так, що часто доступні або вставлені елементи стають ближче до кореневого вузла.

План презентації

01 Історія

02 Ідея алгоритму

03 Ізложення алгоритму

04 Асимптотика

05 Застосування

06 Більше про алгоритм

01
Історія

Splay дерево було вперше представлено Деніелом Домініком Слітором та Робертом Ендре Тарджаном у 1985 році. Воно має просту та ефективну реалізацію, яка дозволяє йому виконувати операції пошуку, вставки та видалення за часом O(log n) в умовній складності, де n — кількість елементів у дереві.

02 Ідея алгоритму

Основна ідея splay дерев полягає в тому, щоб перемістити останній доступний або вставлений елемент до кореня дерева, виконуючи послідовність обертань дерева, звані "splaying". Splaying — це процес реструктуризації дерева, в результаті якого останній доступний або вставлений елемент стає новим коренем і поступово переміщує решту вузлів ближче до кореня.

Splay дерева ефективні на практиці завдяки своїй саморегулюючій природі, яка зменшує загальний час доступу до часто використовуваних елементів. Це робить їх гарним вибором для додатків, які вимагають швидких та динамічних структур даних, таких як системи кешування, стиснення даних та алгоритми маршрутизації мереж. Однак, головний недолік splay дерев полягає в тому, що вони не гарантують збалансованої структури дерева, що може призвести до зниження продуктивності в найгірших сценаріях. Також splay дерева не підходять для додатків, які вимагають гарантованої продуктивності в найгіршому випадку, таких як системи реального часу або системи з критичною безпекою.

Загалом, splay дерево — це потужна та універсальна структура даних, яка пропонує швидкий та ефективний доступ до часто використовуваних або вставлених елементів. Вони широко використовуються в різних додатках і пропонують відмінний компроміс між продуктивністю та простотою.

03 Ізложення алгоритму

Splay дерево — це самобалансуюче бінарне дерево пошуку, розроблене для ефективного доступу до елементів даних на основі їх ключових значень.

Основною особливістю splay дерев є те, що кожен раз, коли елемент доступний, він переміщується до кореня дерева, створюючи більш збалансовану структуру для наступних доступів. Splay дерева характеризуються використанням обертань, які є локальними трансформаціями дерева, що змінюють його форму, але зберігають порядок елементів.

Обертання використовуються для переміщення доступного елемента до кореня дерева, а також для відновлення балансу дерева, якщо воно стає незбалансованим після кількох доступів.

Операції в splay деревах:

Вставка	Щоб вставити новий елемент у дерево, спочатку виконайте звичайну вставку бінарного дерева пошуку. Потім застосуйте обертання, щоб перенести новостворений елемент до кореня дерева.
Видалення	Щоб видалити елемент із дерева, спочатку знайдіть його за допомогою пошуку бінарного дерева пошуку. Потім, якщо елемент не має дітей, просто видаліть його. Якщо він має одну дитину, підніміть цю дитину на його місце в дереві. Якщо у нього двоє дітей, знайдіть спадкоємця елемента (найменший елемент у його правому піддереві), поміняйте ключі з елементом, який потрібно видалити, та видаліть спадкоємця замість нього.
Пошук	Щоб знайти елемент у дереві, спочатку виконайте пошук бінарного дерева пошуку. Якщо елемент знайдено, застосуйте обертання, щоб перенести його до кореня дерева. Якщо він не знайдений, застосуйте обертання до останнього відвіданого вузла у пошуку, який стає новим коренем.
Обертання	Обертання, які використовуються в splay дереві, можуть бути Zig або Zig-Zig обертаннями. Обертання Zig використовується для переміщення вузла до кореня, тоді як обертання Zig-Zig використовується для балансування дерева після кількох доступів до елементів у тому самому піддереві.

Обертання

Обертання Zig:

Якщо вузол має праву дитину, виконується праве обертання, щоб перенести його до кореня. Якщо він має ліву дитину, виконується ліве обертання.

Обертання Zig-Zig:

Якщо вузол має онука, який також є дитиною його дитини справа або зліва, виконується подвійне обертання, щоб збалансувати дерево. Наприклад, якщо вузол має праву дитину, а права дитина має ліву дитину, виконується обертання право-ліво. Якщо вузол має ліву дитину, а ліва дитина має праву дитину, виконується обертання ліво-право.

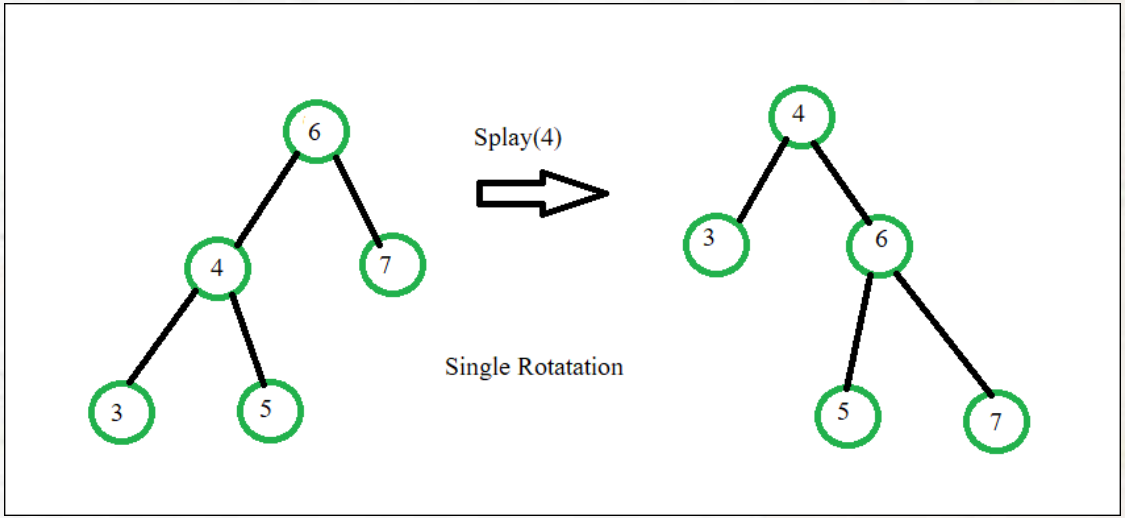
Обертання

- Обертання Zig
- Обертання Zag
- Обертання Zig-Zig
- Обертання Zag-Zag
- Обертання Zig-Zag
- Обертання Zag-Zig

Роздивимось кожне з обертань

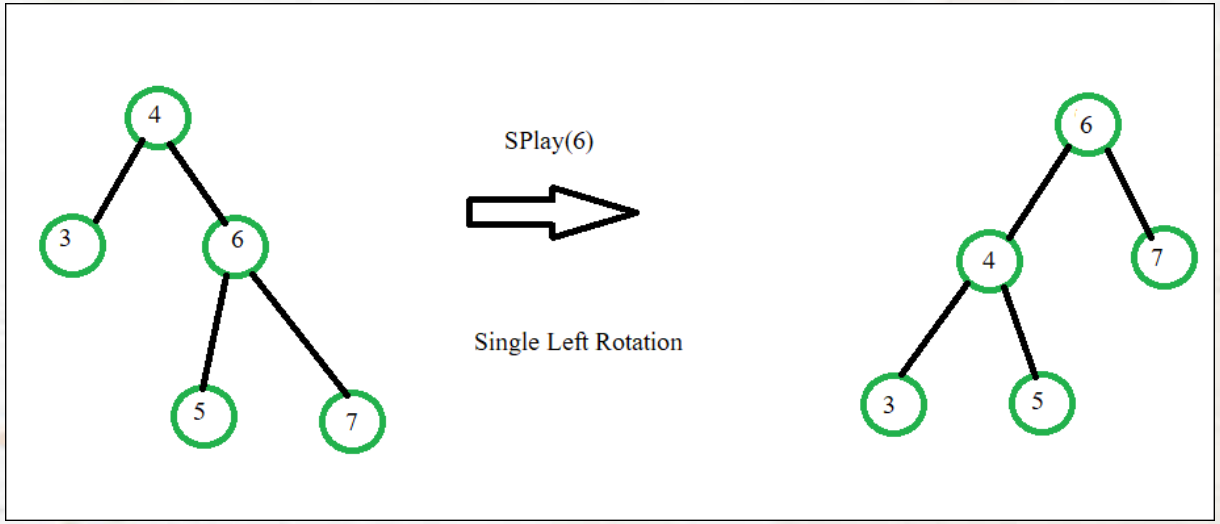
Обертання Zig:

Обертання Zig в splay дереві працює подібно до одиночного правого обертання в обертаннях AVL дереві. Це обертання призводить до переміщення вузлів на одну позицію праворуч від їх поточного розташування.



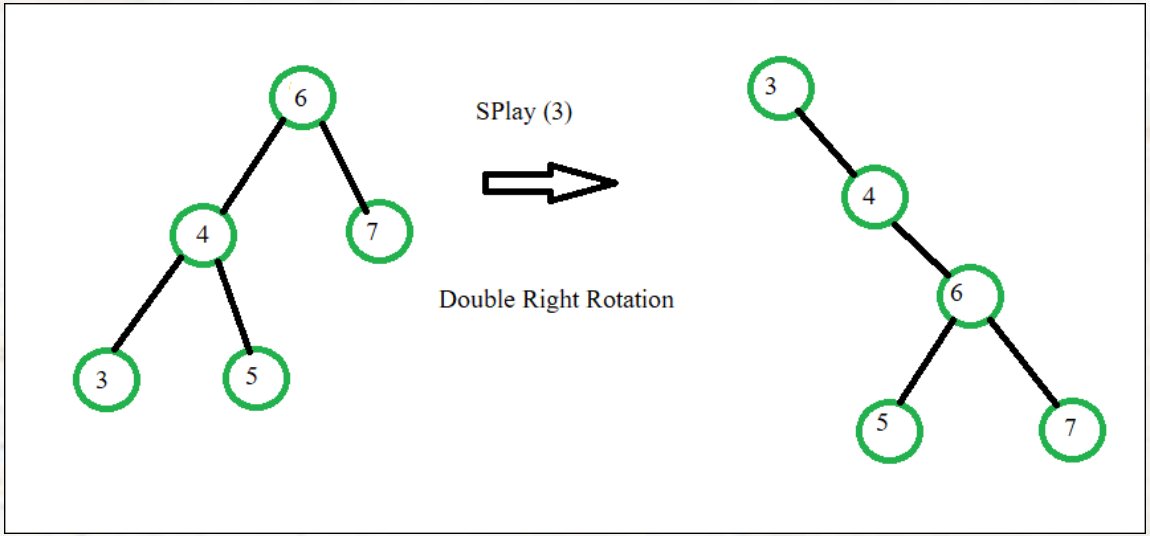
Обертання Zag:

Обертання Zag у splay дереві діє подібно до одиночного лівого обертання в обертаннях AVL дереві. Під час цього обертання вузли переміщуються на одну позицію ліворуч від їх поточного розташування.



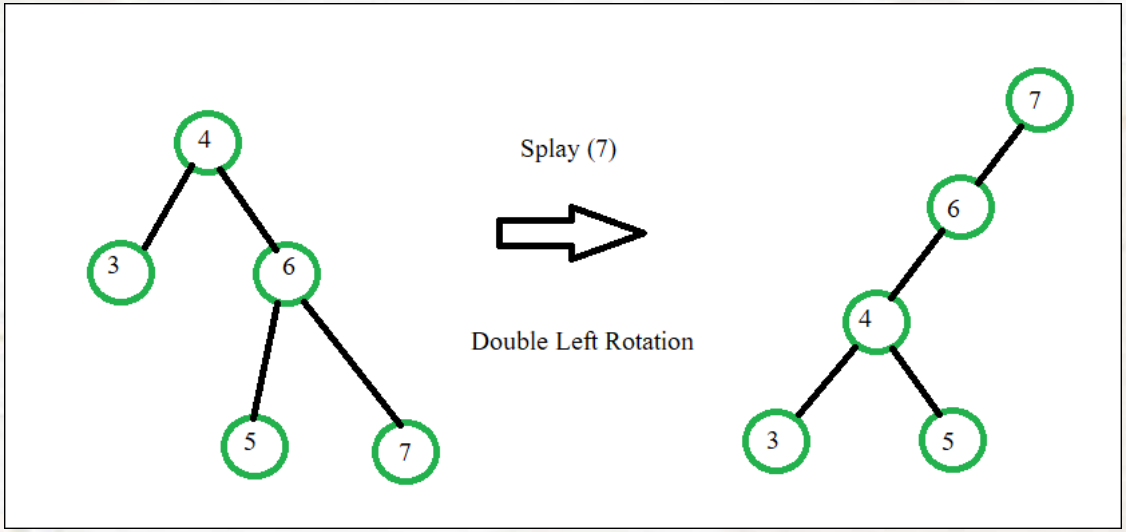
Обертання Zig-Zig:

Обертання Zig-Zig у splay дереві є подвійним обертанням zig. В результаті цього обертання вузли переміщуються на дві позиції праворуч від їх поточного розташування.



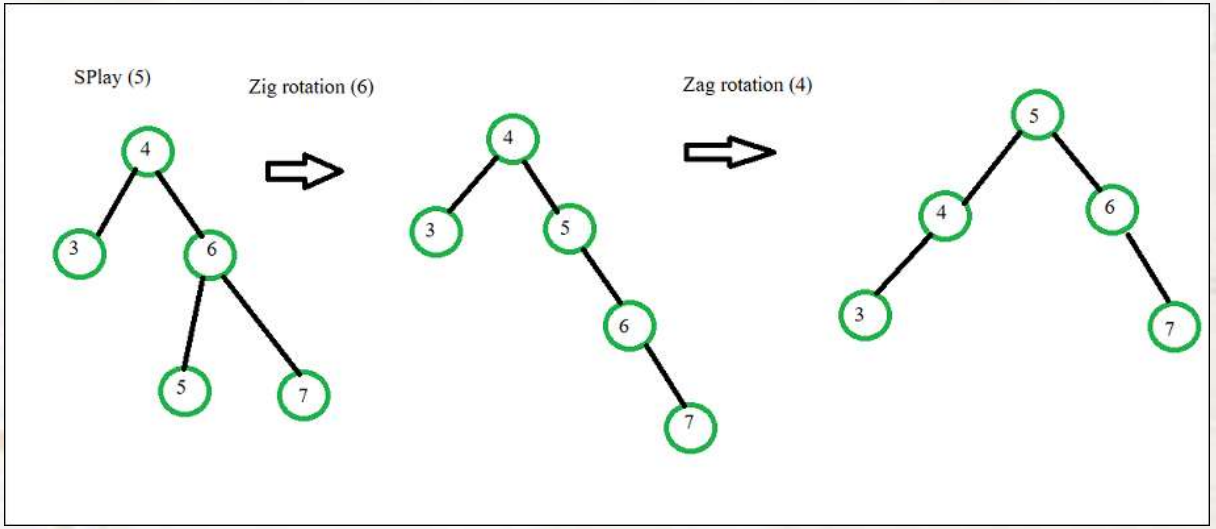
Обертання Zag-Zag:

У splay дерев обертання Zag-Zag є подвійним обертанням zag. Це обертання змушує вузли переміщуватися на дві позиції ліворуч від їх поточного положення.



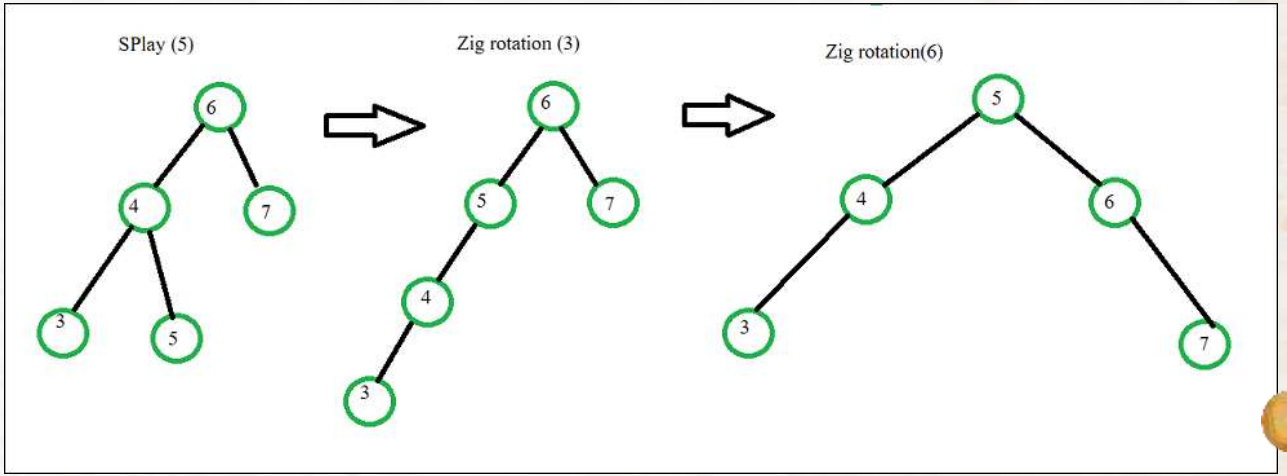
Обертання Zig-Zag:

Обертання Zig-Zag у splay дереві є комбінацією обертання zig, за яким слідує обертання zag. В результаті цього обертання вузли переміщуються на одну позицію праворуч, а потім на одну позицію ліворуч від їх поточного розташування.



Обертання Zag-Zig:

Обертання Zag-Zig у splay дереві є серією обертань zag, за якими слідує обертання zig. В результаті вузли переміщуються на одну позицію ліворуч, а потім зміщуються на одну позицію праворуч від їх поточного розташування.



Доказ коректності:

Одним із ключових аспектів доказу коректності splay дерева є збереження основної властивості бінарного дерева пошуку під час операцій splaying. Незалежно від того, які обертання (Zig, Zag, Zig-Zig, Zag-Zag, Zig-Zag або Zag-Zig) виконуються, алгоритм завжди зберігає порядок елементів, де всі елементи лівого піддерева менші за корінь, а всі елементи правого піддерева більші. Це доводиться через індуктивний аналіз операцій обертання, демонструючи, що кожна з них лише переставляє вузли без порушення загального порядку.

Доказ ефективності splay дерев проводиться за допомогою амортизованого аналізу. Цей метод аналізу показує, що хоча деякі окремі операції можуть мати час виконання вище за O(log n), середня часова складність усіх операцій на довгому проміжку часу все одно залишається O(log n). Для цього часто використовується метод потенціалів або теорема про агрегатний аналіз, де кожній структурі дерева присвоюється певний "потенціал", який відображає її "віддаленість" від збалансованого стану. Цей потенціал забезпечує математичне обґрунтування того, чому середня вартість операцій є логарифмічною.

З урахуванням вищезгаданих доказів можна зробити висновок, що splay дерева є коректними як у плані виконання операцій бінарного дерева пошуку, так і в плані підтримки ефективності через амортизовану часову складність. Це робить їх відмінним вибором для сценаріїв використання, де часто доступаються певні елементи, оскільки такі елементи мають тенденцію залишатися ближче до кореня, що забезпечує швидший доступ.

04 Асимптотика

Теоретична асимптотика за часом:

Асимптотика за часом для splay дерева включає кілька ключових теорем. Згідно з Баланс Теоремою, вартість будь-якої послідовності з m доступів (вставка, видалення, пошук) у splay дереві з n елементами становить O(m log n + n log n). Це означає, що вартість декількох операцій розподіляється рівномірно між ними, демонструючи амортизовану ефективність структури.

Амортизована вартість кожної операції в splay дереві, згідно з Splay Lemma, становить O(log n). Це підкріплюється теорією потенціалу: деякі операції можуть бути дорогими, але зменшення потенціалу дерева після таких операцій компенсує цю високу вартість. Відповідно до Splay Tree Performance Bound, загальний час, необхідний для виконання будь-якої послідовності з m операцій вставки/видалення/пошуку на splay дереві, становить O(m log n), що демонструє ефективність структури в довгостроковій перспективі.

Теоретична асимптотика за пам'яттю:

Щодо асимптотики за пам'яттю для splay дерева, вона зазвичай є O(n), де n - кількість вузлів у дереві. Ця оцінка базується на необхідності зберігання інформації для кожного вузла (ключ, посилання на лівий та правий піддерева). Кожен вузол дерева вимагає фіксованої кількості пам'яті, тому загальний обсяг пам'яті, який використовується деревом, є пропорційним до кількості його вузлів. Це лінійна залежність від розміру дерева, що є типовим для більшості структур даних, заснованих на вузлах.

Практична асимптотика:

Для того, щоб виміряти асимптотику програми використаємо консольну версію програми мовою C# (тест програми представлений на наступному слайді). У цій програмі замір часу для операцій вставки, пошуку та видалення у splay дереві здійснюється за допомогою класу Stopwatch, який забезпечує високу точність вимірювання. Час вимірюється шляхом запуску та зупинки таймера перед та після кожної операції відповідно. Замір використання пам'яті відбувається через GC.GetTotalMemory, що дозволяє оцінити зміну використання пам'яті перед і після кожної операції. Середній час та середнє використання пам'яті обчислюються шляхом ділення загальних значень на кількість ітерацій операції.

Реалізація на C#

1)

```
using System;
using System.Diagnostics;
using System.Text;

22 references
class Node
{
    public int key;
    public Node left, right;
}

3 references
class SplayTree
{
    2 references
    public static Node newNode(int key)
    {
        Node node = new Node();
        node.key = key;
        node.left = node.right = null;
        return node;
    }

    3 references
    public static Node rightRotate(Node x)
    {
        Node y = x.left;
        x.left = y.right;
        y.right = x;
        return y;
    }

    3 references
    public static Node leftRotate(Node x)
    {
        Node y = x.right;
        x.right = y.left;
        y.left = x;
        return y;
    }
```

2)

```
8 references
public static Node splay(Node root, int key)
{
    if (root == null || root.key == key)
        return root;

    if (root.key > key)
    {
        if (root.left == null) return root;
        if (root.left.key > key)
        {
            root.left.left = splay(root.left.left, key);
            root = rightRotate(root);
        }
        else if (root.left.key < key)
        {
            root.left.right = splay(root.left.right, key);
            if (root.left.right != null)
                root.left = leftRotate(root.left);
        }
        return (root.left == null) ? root : rightRotate(root);
    }
    else
    {
        if (root.right == null) return root;
        if (root.right.key > key)
        {
            root.right.left = splay(root.right.left, key);
            if (root.right.left != null)
                root.right = rightRotate(root.right);
        }
        else if (root.right.key < key)
        {
            root.right.right = splay(root.right.right, key);
            root = leftRotate(root);
        }
        return (root.right == null) ? root : leftRotate(root);
    }
}
```

Реалізація на C#

3)

```
1 reference
public static Node insert(Node root, int key, Stopwatch stopwatch, ref long memoryUsage)
{
    long memoryBefore = GC.GetTotalMemory(false);
    stopwatch.Start();

    if (root == null)
    {
        root = newNode(key);
    }
    else
    {
        root = splay(root, key);
        if (root.key != key)
        {
            Node node = newNode(key);
            if (root.key > key)
            {
                node.right = root;
                node.left = root.left;
                root.left = null;
            }
            else
            {
                node.left = root;
                node.right = root.right;
                root.right = null;
            }
            root = node;
        }
    }

    stopwatch.Stop();
    long memoryAfter = GC.GetTotalMemory(false);
    memoryUsage += memoryAfter - memoryBefore;

    return root;
}
```

4)

```
public static Node search(Node root, int key, Stopwatch stopwatch, ref long memoryUsage)
{
    long memoryBefore = GC.GetTotalMemory(false);
    stopwatch.Start();

    root = splay(root, key);

    stopwatch.Stop();
    long memoryAfter = GC.GetTotalMemory(false);
    memoryUsage += memoryAfter - memoryBefore;

    return root;
}

1 reference
public static Node delete(Node root, int key, Stopwatch stopwatch, ref long memoryUsage)
{
    long memoryBefore = GC.GetTotalMemory(false);
    stopwatch.Start();

    if (root == null) return null;
    root = splay(root, key);

    if (key != root.key) return root;

    Node temp;
    if (root.left == null)
    {
        temp = root;
        root = root.right;
    }
    else
    {
        temp = root;
        root = splay(root.left, key);
        root.right = temp.right;
    }

    stopwatch.Stop();
    long memoryAfter = GC.GetTotalMemory(false);
    memoryUsage += memoryAfter - memoryBefore;

    return root;
}
```

Реалізація на C#

5)

```
2 references
public static void preOrder(Node node)
{
    if (node != null)
    {
        Console.Write(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}
```

6)

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;
        Node root = null;
        Stopwatch stopwatch = new Stopwatch();
        int iterations = 1000;
        long memoryUsageInsert = 0, memoryUsageSearch = 0, memoryUsageDelete = 0;

        for (int i = 0; i < iterations; i++)
        {
            root = SplayTree.insert(root, i, stopwatch, ref memoryUsageInsert);
        }
        Console.WriteLine($"Вставка: Середній час - {stopwatch.Elapsed.TotalMilliseconds / iterations} мс, " +
            $"Середнє використання пам'яті - {memoryUsageInsert / iterations} байт");

        stopwatch.Reset();

        for (int i = 0; i < iterations; i++)
        {
            root = SplayTree.search(root, i, stopwatch, ref memoryUsageSearch);
        }
        Console.WriteLine($"Пошук: Середній час - {stopwatch.Elapsed.TotalMilliseconds / iterations} мс, " +
            $"Середнє використання пам'яті - {memoryUsageSearch / iterations} байт");

        stopwatch.Reset();

        for (int i = 0; i < iterations; i++)
        {
            root = SplayTree.delete(root, i, stopwatch, ref memoryUsageDelete);
        }
        Console.WriteLine($"Видалення: Середній час - {stopwatch.Elapsed.TotalMilliseconds / iterations} мс, " +
            $"Середнє використання пам'яті - {memoryUsageDelete / iterations} байт");
    }
}
```

Практична асимптотика:

Вивід програми:

```
Вставка: Середній час – 0,0050592 мс, Середнє використання пам'яті – 40 байт
Пошук: Середній час – 0,00044919999999999997 мс, Середнє використання пам'яті – 0 байт
Видалення: Середній час – 0,0001336 мс, Середнє використання пам'яті – 0 байт
```

Проаналізуємо ортимані результати:

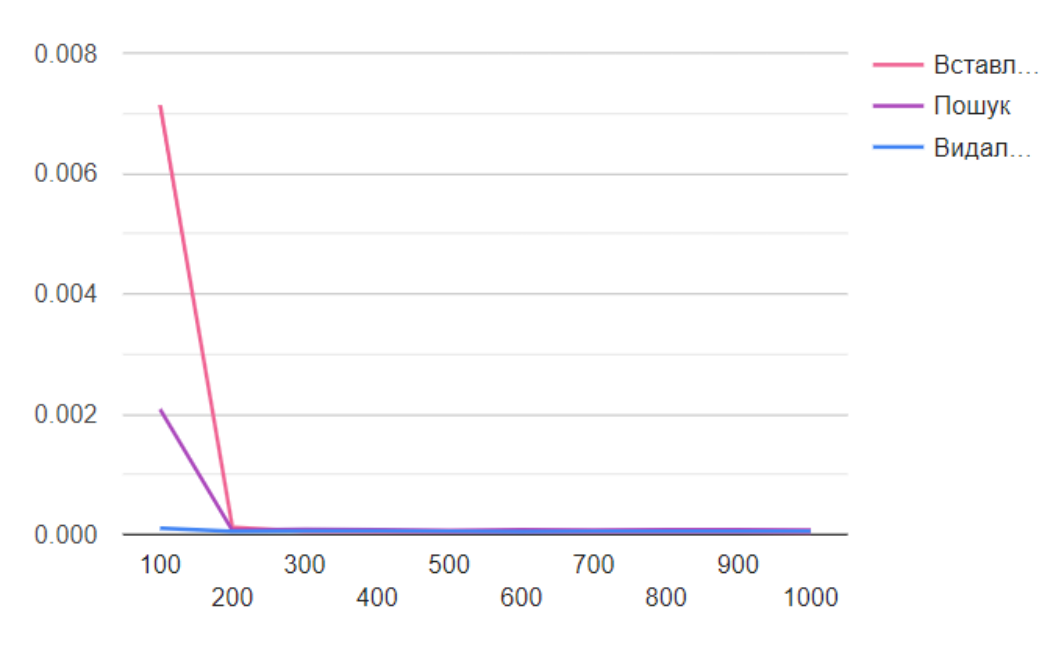
Вставка:
- Середній час: 0.0050592 мілісекунди на операцію.
- Середнє використання пам'яті: 40 байтів на операцію.

Це означає, що вставка нового елемента у splay дерево в середньому займає трохи більше ніж 5 мікросекунд, використовуючи приблизно 40 байтів додаткової пам'яті. Це свідчить про порівняно низьку часову та просторову вимогливість операції вставки.

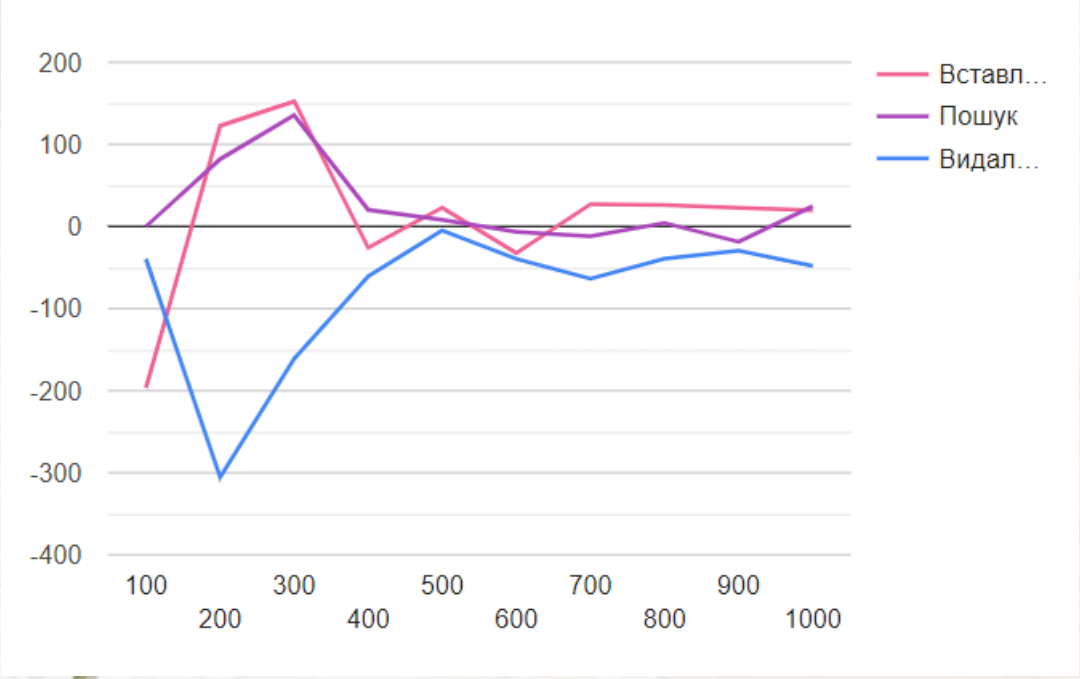
Практична асимптотика:

2. Пошук:
   - Середній час: приблизно 0.0004492 мілісекунди (або 449.2 наносекунд) на операцію.
   - Середнє використання пам'яті: 0 байтів на операцію.
   Операція пошуку є дуже швидкою (менше ніж пів мікросекунди на пошук) і, згідно з результатами, не збільшує загальне використання пам'яті. Це може свідчити про те, що операція пошуку виконується in-place без виділення додаткової пам'яті.
3. Видалення:
   - Середній час: 0.0001336 мілісекунди (або 133.6 наносекунд) на операцію.
   - Середнє використання пам'яті: 0 байтів на операцію.
   Видалення елементів з splay дерева також дуже швидке і, за даними замірів, не впливає на загальне використання пам'яті. Це може означати, що операція видалення також виконується in-place.
   Загалом, ці результати демонструють, що splay дерева вирізняються високою швидкістю виконання операцій та ефективним використанням пам'яті.

Графік за часом:



Графік за пам'яттю:



Порівняння асимптотик:

Теоретично, асимптотика операцій у splay дереві має бути амортизованою часовою складністю O(log n) для вставки, пошуку та видалення. Це означає, що хоча окремі операції можуть мати різну тривалість, середній час виконання кожної операції за багато виконань зближається до логарифмічної залежності від кількості елементів у дереві. На практиці, отримані результати часто можуть відрізнятися від теоретичних через різні фактори, такі як ефективність реалізації, особливості середовища виконання та операцій з пам'яттю. На основі експерименту, середній час виконання операцій, як правило, демонструє дуже низькі значення, що свідчить про високу ефективність алгоритму. Особливо це стосується операцій пошуку та видалення, час яких залишається низьким навіть при збільшенні кількості елементів. Це може вказувати на те, що практична складність цих операцій добре відповідає або навіть перевищує теоретичні очікування. Водночас, деякі аномалії у вимірах, такі як від'ємне використання пам'яті, нагадують про складнощі точного вимірювання та впливу збирача сміття на процес. В цілому, практичні результати демонструють ефективність splay дерева, але вони також підкреслюють важливість розуміння впливу підсистеми управління пам'яттю при аналізі продуктивності алгоритмів.

05 Застосування

Splay дерева можуть бути використані для реалізації управління кеш-пам'яттю, де найчастіше використовувані елементи переміщуються на верхівку дерева для швидшого доступу.

Splay дерева можуть бути використані для індексації баз даних для швидшого пошуку та отримання даних.

Splay дерева можуть бути використані для зберігання метаданих файлової системи, таких як таблиця розподілу, структура директорії та атрибути файлів.

Splay дерева можуть бути використані в онлайн-іграх для зберігання та управління високими результатами, таблицями лідерів та статистикою гравців.

Splay дерева можуть бути використані для реалізації графових алгоритмів, таких як знаходження найкоротшого шляху в зваженому графі.

Splay дерева можуть бути використані для стиснення даних шляхом ідентифікації та кодування повторюваних візерунків.

Splay дерева можуть бути використані в додатках для обробки тексту, таких як перевірка орфографії, де слова зберігаються у splay дереві для швидкого пошуку та отримання.

Більше про splay дерева:

	“Data Structures and Algorithm Analysis in Java” Mark Allen Weiss
	“Introduction to Algorithms” Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
	“Data Structures and Algorithms in C++” Michael T. Goodrich and Roberto Tamassia
	“Handbook of Data Structures and Applications” Dinesh P. Mehta, Sartaj Sahni

Висновок

В результаті, Splay дерева — це динамічна самобалансуюча структура даних бінарного дерева пошуку, яка забезпечує ефективний спосіб пошуку, вставки та видалення елементів. Вони відрізняються від традиційних збалансованих бінарних дерев пошуку, таких як AVL та Червоно-чорні дерева, оскільки вони переорганізовують дерево після кожної операції, переміщуючи нещодавно доступний вузол до кореня. Це допомагає зменшити висоту дерева і призводить до швидших операцій. Splay дерева є високо гнучкими і можуть бути адаптованими до різних випадків використання. Хоча вони можуть мати вищий наклад в термінах обертань, їх простота та універсальність роблять їх корисними інструментами для вирішення широкого спектру проблем.