Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук та кібернетики

Кафедра інтелектуальних інформацій систем

Алгоритми та складність

Лабораторна робота №2

Завдання №2 – АА-дерева

Тип даних – комплексні числа

Виконав студент 2-го курсу

Групи ІПС-21

Ольховатий Ігор

2023

Зміст

Теоретичні відомості

АА-дерево — це форма збалансованого дерева, яке використовується для ефективного зберігання та отримання впорядкованих даних. АА-дерева названі на честь Арне Андерссона, того, хто створив їх теорію.

АА-дерева є різновидом червоно-чорного дерева, форми бінарного дерева пошуку, яке підтримує ефективне додавання та видалення записів. На відміну від червоно-чорних дерев, червоні вузли в АА-дереві можна додати лише як правий дочірній елемент. Іншими словами, жоден червоний вузол не може бути лівим дочірнім елементом. Це призводить до симуляції дерева 2–3 замість дерева 2–3–4, що значно спрощує операції обслуговування. Алгоритми обслуговування червоно-чорного дерева повинні враховувати сім різних форм, щоб правильно збалансувати дерево.

З іншого боку, АА-дереву потрібно враховувати лише дві форми через сувору вимогу, згідно з якою лише праві посилання можуть бути червоними.

Оберти для підтримки балансу:

Тоді як для червоно-чорних дерев потрібен один біт балансуючих метаданих на вузол (колір), АА-дерева потребують $O(log(log(N)))$ бітів метаданих на вузол у формі цілого «рівня». Наступні інваріанти справедливі для АА-дерев:

1. Рівень кожного листкового вузла один.
2. Рівень кожного лівого дочірнього елемента рівно на одиницю менший, ніж рівень його батька.
3. Рівень кожної правої дитини дорівнює або на одиницю менше рівня її батька.
4. Рівень кожного правильного онука строго нижчий, ніж рівень його бабусі та дідуся.
5. Кожен вузол рівня вище одного має двох дочірніх вузлів.

Посилання, у якому рівень дочірнього елемента дорівнює рівню його батька, називається горизонтальним посиланням і є аналогом червоного посилання в червоно-чорному дереві. Окремі праві горизонтальні посилання дозволені, але послідовні заборонені; всі ліві горизонтальні посилання заборонені. Це більш обмежувальні обмеження, ніж аналогічні для червоно-чорних дерев, у

результаті чого повторне балансування дерева АА процедурно набагато простіше, ніж повторне балансування червоно-чорного дерева.

Вставки та видалення можуть тимчасово призвести до незбалансованості АА-дерева (тобто до порушення інваріантів АА-дерева) Для відновлення балансу потрібні лише дві різні операції: «перекіс» і «розбиття». Нахил — це поворот праворуч для заміни піддерева, що містить ліве горизонтальне посилання, на піддерево, що містить замість нього праве горизонтальне посилання. Розділення — це обертання вліво та підвищення рівня для заміни піддерева, що містить два або більше послідовних правих горизонтальних посилань, на піддерево, що містить два менших послідовних правих горизонтальних посилання. Реалізація вставки та видалення зі збереженням балансу спрощується завдяки використанню операцій нахилу та розбиття для зміни дерева лише за потреби, замість того, щоб змушувати їх викликаючих вирішувати, нахиляти чи розбивати.

Алгоритми

1. Перекіс

Перекіс полягає в тому, що вузол, що має більший рівень, повинен бути переставлений вправо. Для цього потрібно зробити так, щоб він став дочірнім вузлом правого нащадка свого батьківського вузла.

Опис:

1) Перевірка, чи є вузол пустим або має лівого нащадка.
2) Перевірка, чи вузол має той самий рівень, що його лівий нащадок. Якщо ні, повертаємо вузол без змін.
3) Якщо вузол має той самий рівень, що його лівий нащадок, виконуємо перестановку:
    - Зберігаємо посилання на лівого нащадка вузла у змінній **L**.
    - Змінюємо посилання лівого нащадка вузла на його правого нащадка.

- Якщо правий нащадок існує, змінюємо його батьківський вузол та напрямок на лівий вузол.
- Змінюємо батьківський вузол правого нащадка на вузол, що мав батьківський вузол першим.
- Змінюємо напрямок правого нащадка на "L".
- Змінюємо батьківський вузол вузла на батьківський вузол лівого нащадка.
- Якщо вузол мав батьківський вузол, змінюємо напрямок нащадка від батьківського вузла на "R".
- Змінюємо батьківський вузол лівого нащадка на вузол, що мав батьківський вуз

2. Розбиття

Даний алгоритм виконує перебудову AA-дерева під час вставки елемента або видалення елемента, якщо у дерева сталася ситуація, коли права гілка дерева має на 2 рівня вище, ніж ліва гілка.

Опис:

1) Перевіряємо чи існує даний вузол і чи існують його права і права права гілки (якщо хочаб одного з цих вузлів немає, то повертаємо сам вузол).
2) Перевіряємо чи рівні рівні рівні даного вузла та його правої правої гілки. Якщо так, то проводимо перебудову дерева
    - Створюємо змінну R, яка містить правий вузол даного вузла.
    - Правий вузол даного вузла стає лівим вузлом R
    - Якщо ліва гілка R існує, то встановлюємо батьківський вузол та напрямок лівої гілки R на поточний вузол.
    - Лівий вузол R стає батьківським вузлом поточного вузла.
    - Напрямок вузлів змінюється на 'L' для поточного вузла та на напрямок поточного вузла для R.
    - Рівень R збільшується на 1.
    - Повертаємо R
3) Якщо рівність не виконується, повертаємо вузол.

3. Вставка

Опис:

1) Функція отримує вузол для вставки нового значення, батьківський вузол та напрямок відносно батьківського вузла (**'L'** для лівого нащадка та **'R'** для правого). Відповідно до напрямку відносно батьківського вузла, вузол додається до відповідного поля **left** або **right** батьківського вузла. Якщо вузол є порожнім, створюється новий вузол зі значенням, який стає коренем дерева, а довжина дерева збільшується на 1.
2) Інакше порівнюється значення зі значенням поточного вузла. Якщо воно менше значення у вузлі, нове значення додається до лівого піддерева вузла. Якщо воно більше, додається до правого піддерева вузла. Якщо значення дорівнює, додається до вузла, а довжина збільшується на 1.
3) Після вставки нового вузла перевіряється на виконання правил AA-дерева та проводяться перекос та розбиття.

4. Вилучення

Опис:

1) Перевіряється, чи не є задане значення пустим.
2) Визначається порівняння між заданим значенням і значенням поточного вузла. Якщо перше більше, запускається алгоритм вилучення для правого дочірнього вузла, якщо менше - для лівого. Якщо значення співпадають, вузол видаляється іншим методом, який відповідає за видалення значення з вузла.
3) Розраховується довжина вузла до та після видалення значення. Якщо довжина стала меншою, відповідно зменшується лічильник довжини дерева.
4) Якщо після видалення значення довжина вузла стала рівною 0, він видаляється з дерева. Якщо вузол мав два дочірніх вузли, вибирається новий вузол з лівого піддерева з найбільшим значенням або з правого піддерева з найменшим значенням, залежно від випадку. Цей вузол підставляється на місце видаленого вузла.

Складність

1. Перекіс та розбиття

Часова складність операції перекосу та розбиття становить *O(1)* у найгіршому випадку. Це пояснюється тим, що функції виконують лише постійну кількість операцій для будь-якого дерева вхідних даних і не залежить від розміру дерева.

2. Вилучення

Часова складність операції видалення найгіршому випадку в дереві AA становить *O(log n)*, де *n* – кількість вузлів у дереві. Це пояснюється тим, що для видалення може знадобитися перебалансування дерева, що в гіршому випадку може зайняти *O(log n)* часу. Зокрема, після видалення вузла його батьківський вузол може знадобитися повторно збалансувати за допомогою операцій нахилу та розділення.

3. Вставка

Часова складність операції вставки найгіршому випадку в дерево AA також дорівнює *O(log n)*, де *n* – кількість вузлів у дереві. Це пояснюється тим, що для вставки також може знадобитися перебалансування дерева, що може зайняти *O(log n)* часу в гіршому випадку. Зокрема, після того, як вузол вставлено, його батьківський вузол може знадобитися повторно збалансувати за допомогою операцій нахилу та розділення.

Мова програмування

Python

Модулі програми

Функції, класи та їх методи:

```
class Complex:
        Представляє комплексне число з дійсною та уявною складовою.
    def __init__(self, real: int = 0, imag: int = 0, rand: bool = False) -> None:
        Ініціалізує новий об'єкт типу Complex.
```

```
    def __hash__(self) -> int:
        Повертає хеш-значення для цього комплексного об'єкта.

    def __neg__(self):
        Повертає від'ємний до цього комплексного об'єкта.

    def __str__(self) -> str:
        Повертає рядкове представлення цього комплексного об'єкта.

    def __repr__(self) -> str:
        Повертає рядкове представлення цього об'єкта Complex, яке можна використовувати для
відтворення об'єкта.

    def __add__(self, other: Union[int, float, 'Complex']) -> 'Complex':
        Додає заданий об'єкт до цього комплексного об'єкта та повертає новий комплексний об'єкт,
що представляє суму.

    def __sub__(self, other: Union[int, float, 'Complex']) -> 'Complex':
        Віднімає заданий об'єкт із цього комплексного об'єкта та повертає новий складний об'єкт,
що представляє різниця.

    def __mul__(self, other: Union[int, float, 'Complex']) -> 'Complex':
        Помножує заданий об'єкт на цей комплексний об'єкт і повертає новий комплексний об'єкт,
що представляє добуток.

    def __truediv__(self, other: Union[int, float, 'Complex']) -> 'Complex':
        Ділить цей комплексний об'єкт на даний об'єкт і повертає новий комплексний об'єкт, що
представляє приватне.

    def __abs__(self) -> float:
        Повертає величину комплексного числа.

    def __eq__(self, other) -> bool:
        Визначає, чи рівні два комплексні числа.

    def __lt__(self, other) -> bool:
        Визначає, чи комплексне число менше іншого комплексного числа.
```

```
    def __le__(self, other) -> bool:
            Визначає, чи комплексне число менше або дорівнює іншому комплексному числу.

    def __gt__(self, other) -> bool:
            Визначає, чи комплексне число більше за інше комплексне число.

    def __ge__(self, other) -> bool:
            Визначає, чи комплексне число більше або дорівнює іншому комплексному числу.

def _nodes_simple_comparison(v1, v2):
    """ Функція порівняння """

def make_compare_function_by_key(key):
    """ Бере функцію вилучення ключів і виконує функцію порівняння """
    return lambda v1, v2: _nodes_simple_comparison(key(v1), key(v2))


class _AATreeNode:
    """ Представляє вузол дерева """
    def __init__(self, val, parent, direction=None):
            """ Ініціалізує вузол """

    def getval(self):
            """ Повертає значення у вузлі """

    def addval(self, val):
            """ Додає значення до вузла """

    def drop_value(self, val, all_copies):
            """ Вилучає значення з вузла, використовується лише якщо є метод для вилучення """


class AATree:
    """ AA-дерево """

    def __init__(self, comparison_func=_nodes_simple_comparison, source=None):
    """ Ініціалізація дерева """
```

```
def __len__(self):
        """ Повертає довжину дерева """

def insert(self, val):
        """ Вставка """

def insert_from(self, source):
        """ Вставка з ітерованого об'єкту """


@staticmethod
def _skew(node):
        """ Перекіс """

@staticmethod
def _split(node):
        """ Розбиття """

def _insert_into_node(self, node, val, parent, direction=None):
        """ Рекурсивна функція вставки """

def forward_from(self, start=None, inclusive=True,
                 stop=None, stop_incl=False):
    """ Створює та повертає об'єкт генератора, який можна використовувати для
  обхід дерева в напрямку «вперед». """

def backward_from(self, start=None, inclusive=True,
                  stop=None, stop_incl=False):
        """ Створює та повертає об'єкт генератора, який можна використовувати для
  обхід дерева в напрямку «назад». """

def min(self, limit=None, inclusive=True):
    """ Повертає мінімальне значення, яке не менше (якщо "включно"
параметр має значення True) або більше (включно=False)
визначеного ліміту."""

def max(self, limit=None, inclusive=True):
```

```
        """ Повертає максимальне значення, яке не більше (якщо "включно"
    параметр має значення True) або менше (включно=False) визначеного ліміту."""


    def nodes_list(self):
        """ Повертає список об'єктів типу _AATreeNode """

    def _nodes_list(self, node):
        """ Рекурсивна функція для повернення списку вузлів """

    def remove(self, val, all_copies=False):
        """ Вилучення """

    def _delete(self, val, node, all_copies):
            """ Рекурсивна функція для вилучення """

    @staticmethod
    def _decrease_level(node):
        """ Знижує рівні для вузла та його нащадків """
```

Інтерфейс користувача

Інтерфейсом користувача для взаємодії з програмою є консоль.

Демонстрація роботи

Вхідні дані і код програми:

```
input_data = [Complex(2, 2), Complex(4, 2), Complex(3, 2), Complex(4, 5), Complex(5, 1)]
tree = AATree()
tree.insert_from(input_data)
print(*[v for v in tree.forward_from()])
a = tree.max(Complex(5, 0))
print(f"Max values not greater than 5 + 0i: {a}")
b = tree.min(Complex(4, 2))
print(f"Min values not less than 4 + 2i: {b}")
```

Результат роботи:

```
2 + 2i 3 + 2i 4 + 2i 5 + 1i 4 + 5i
Max values not greater than 5 + 0i: 4 + 2i
Min values not less than 4 + 2i: 4 + 2i
```

Детальніше опишемо роботу програми:

На вхід подається набір комплексних чисел: 2 + 2i, 4 + 2i, 3 + 2i, 4 + 5i, 5 + i. Далі будується пусте дерево, до нього вставляють всі елементи зі списку вхідних даних. Значення побудовного дерева виводяться на екран у порядку послідовного обходу, тобто у порядку зростання (операція порівняння для комплексних чисел визначена як порівняння їх модулів). Бачимо, що саме у такому порядку й показані значення, тобто дерево побудовано коректно. Далі покажемо, як працюють деякі додаткові функції, а саме пошук максимальних/мінімальних елементів але більших/менших за заданий. Знайдемо максимальний елемент менше за 5 + 0i: це 4 + 2i. Знайдемо мінімальний елемент не менший за 4 + 2i: це 4 + 2i.

Висновок

AA-дерева можуть підтримувати збалансованість, причому як вставка так і видалення працюють за логарифмічний час у найгіршому випадку. Окрім того, під час балансування дерева нам потрібно розглядати лише 2 випадки, а 7, як для червоно-чорних дерев. Продуктивність AA-дерева еквівалентна продуктивності червоно-чорного дерева. У той час як AA-дерево робить більше обертів, ніж червоно-чорне дерево, простіші алгоритми, як правило, швидші, і все це врівноважується, щоб отримати однакову продуктивність. Червоно-чорне дерево є більш послідовним у своїй продуктивності, ніж дерево AA, але дерево AA має тенденцію бути більш плоским, що призводить до трохи швидшого часу пошуку.

Література

1. https://en.wikipedia.org/wiki/AA_tree