Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Алгоритми та складність

Варіант №9

“ Дерево відрізків ”

Виконав студент 2-го курсу

Групи ІПС-21

Тесленко Назар Олександрович

Київ – 2025

Завдання:

Реалізувати дерево відрізків (реалізація на основі червоно чорного дерева)

Теорія:

Дерево відрізків (Segment Tree) - структура даних, що дозволяє ефективно виконувати операції на діапазонах (відрізках) елементів, такі як знаходження суми, мінімуму, максимуму тощо, а також оновлення значень, за логарифмічний час.

Червоно-чорне дерево (Red-Black Tree) - вид самобалансованого бінарного дерева пошуку, де кожен вузол має додатковий атрибут - колір (червоний або чорний), що забезпечує приблизно рівномірну глибину всіх гілок дерева.

Дерево відрізків на основі червоно-чорного дерева - гібридна структура даних, що поєднує функціональність дерева відрізків з балансуванням червоно-чорного дерева, забезпечуючи ефективне виконання операцій на діапазонах з гарантованою логарифмічною складністю.

Вузол (Node) - елемент дерева, що містить:

- **left, right** - ліва та права границі відрізка, який представляє вузол
- **sum** - агреговане значення (сума) для відповідного відрізка
- **color** - колір вузла (червоний або чорний)
- **parent** - вказівник на батьківський вузол
- **leftChild, rightChild** - вказівники на лівого та правого нащадків

NIL-вузол (NIL Node) – спеціальний нуль-вузол, що використовується замість nullptr у червоно-чорному дереві, завжди має чорний колір.

Відрізок (Range) - діапазон [left, right], який представляє певний вузол дерева.

Обертання (Rotation) - операція зміни структури дерева, що зберігає порядок обходу (інфіксний порядок), використовується для відновлення балансу:

- **Ліве обертання (Left Rotation)** - обертання, при якому правий нащадок вузла стає його батьком
- **Праве обертання (Right Rotation)** - обертання, при якому лівий нащадок вузла стає його батьком

Колір вузла (Node Color) - атрибут вузла в червоно-чорному дереві:

- **Червоний (Red)** - позначає вузли, які можуть порушувати баланс дерева
- **Чорний (Black)** - позначає вузли, що забезпечують баланс дерева

Властивості червоно-чорного дерева (Red-Black Tree Properties):

1. Кожен вузол або червоний, або чорний

2. Корінь завжди чорний
3. Листя (NIL-вузли) завжди чорні
4. Якщо вузол червоний, обидва його нащадки чорні
5. Для кожного вузла всі прості шляхи від нього до листя містять однакову кількість чорних вузлів

Алгоритм

Побудова структури дерева

- Ініціалізується корінь дерева як NIL (порожній вузол).

Додавання вузлів (інтервалів) до дерева

- Створюється новий вузол для збереження інтервалу та його суми.
- Якщо дерево порожнє, новий вузол стає коренем і позначається чорним.
- В іншому випадку, рекурсивно шукається місце для вставки новго вузла та виконується саме додавання.
- Виконується виправлення можливих порушень властивостей червоно-чорного дерева (fixInsert).
- Оновлюються суми значень у вузлах від нового вузла вгору по дереву (updateSum)

Вивід дерева

- Друкується вміст дерева у вигляді вкладеної структури за допомогою printTreePriv.

Запит суми на інтервалі

- Виконується рекурсивний пошук підінтервалів, що перетинаються із запитом.
- Якщо поточний вузол повністю входить у запитуваний інтервал, його сума додається до результату.
- Якщо інтервал вузла не перетинається із запитом, повертається 0.
- Якщо є часткове перекриття, обчислюється сума значень лівої та правої гілок.

Видалення інтервалу

- Виконується рекурсивний пошук вузлів, що перетинаються із заданим інтервалом.
- Якщо інтервал повністю включає вузол, спочатку рекурсивно видаляються його підвузли.
- Визначається вузол-наступник у разі наявності обох дітей (minimum).
- Виконується заміна вузла (transplant) і коригування властивостей червоно-чорного дерева (fixDelete).

- Після видалення оновлюються суми в батьківських вузлах (updateSum)

Алгоритми методів клас можна реалізувати наступним псевдокодом:

void insert(l, r, sum)

```
створити newNode(l, r, sum), позначити червоним
встановити newNode.leftChild = NIL, newNode.rightChild = NIL
ЯКЩО root == NIL → root = newNode, позначити чорним, ВИХІД

Встановити parent = NIL, current = root
ПОКИ current != NIL
    parent = current
    ВИЗНАЧИТИ напрямок руху (ліво/право) на основі (l, r)
    current = відповідний нащадок

Встановити parent як батька newNode
Додати newNode як лівого або правого нащадка parent
викликати fixInsert(newNode)
викликати updateSum(newNode)
```

double queryPrivate(node, l, r)

```
ЯКЩО node == NIL → ПОВЕРНУТИ 0
ЯКЩО [l, r] повністю включає node → ПОВЕРНУТИ node.sum
ЯКЩО [l, r] не перетинається з node → ПОВЕРНУТИ 0

Встановити totalSum = 0
ДОДАТИ sum(queryPrivate(leftChild, l, r)) якщо leftChild не NIL
ДОДАТИ sum(queryPrivate(rightChild, l, r)) якщо rightChild не NIL

ПОВЕРНУТИ totalSum
```

void removeNode(node, l, r)

```
ЯКЩО node == NIL → ВИХІД
ЯКЩО [l, r] повністю включає node:
    Видалити нащадків у [l, r]
    Видалити node, викликати transplant() для оновлення батьківських
зв’язків
    викликати fixDelete(x) якщо видалено чорний вузол
```

Оновити суму вузлів

ІНАКШЕ:

Рекурсивно викликати removeNode(leftChild, l, r) якщо лівий нащадок перетинається

Рекурсивно викликати removeNode(rightChild, l, r) якщо правий нащадок перетинається

викликати updateSum(node)

void fixInsert(node)

ПОКИ parent червоний:
ВИЗНАЧИТИ grandparent і uncle
ЯКЩО uncle червоний:
recolor parent, uncle, grandparent
перейти до grandparent
ІНАКШЕ:
ЯКЩО node не на "правильному" боці → ротація (leftRotate/rightRotate)
recolor parent і grandparent
ротація grandparent (rightRotate/leftRotate)
root позначити чорним

void fixDelete(x)

ПОКИ x не root і чорний:
ВИЗНАЧИТИ sibling
ЯКЩО sibling червоний:
recolor sibling і parent
ротація parent (leftRotate/rightRotate)
оновити sibling

ЯКЩО обидва нащадки sibling чорні:
recolor sibling
x = parent
ІНАКШЕ:
ЯКЩО sibling має "правильного" червоного нащадка → ротація sibling (leftRotate/rightRotate)
recolor sibling і parent
ротація parent (rightRotate/leftRotate)
x = root

x позначити чорним

Мова реалізації: C++

Модулі програми

- **class SegmentTree**

Реалізує структуру даних для зберігання та обробки інтервалів із можливістю додавання, видалення та запиту сум. Використовує червоно-чорне дерево для підтримки збалансованості.

Головні методи:

- **void insert(l, r, sum)** – додає новий інтервал [l, r] із заданою сумою sum у дерево. Виконує балансування червоно-чорного дерева та оновлення сум.
- **void remove(l, r)** – видаляє інтервали, які перетинаються з [l, r], коригує структуру дерева.
- **int query(l, r)** – повертає суму значень у дереві для заданого інтервалу [l, r].
- **void printTree()** – виводить дерево у вигляді вкладеної структури, відображаючи його поточний стан.

Допоміжні приватні методи:

- **queryPrivate(node, l, r)** – рекурсивний метод для пошуку суми в дереві.
- **removeNode(node, l, r)** – рекурсивне видалення підінтервалів із дерева.
- **fixInsert(node)** – виправляє можливі порушення правил червоно-чорного дерева після вставки.
- **fixDelete(node)** – коригує баланс дерева після видалення вузла.
- **updateSum(node)** – оновлює значення суми у вузлах після змін у дереві.
- **leftRotate(node) / rightRotate(node)** – виконують ротацію для перебалансування дерева.
- **transplant(u, v)** – замінює піддерево u піддеревом v, використовується при видаленні вузлів.
- **minimum(node)** – знаходить вузол із мінімальним значенням у піддереві node.

Складність алгоритму Segment Tree (Red-Black Tree) та залжених методів

Головні методи:

Вставка інтервалу insert(l, r, sum) – O(log n)

створити новий вузол newNode – O(1)
знайти місце вставки в дереві – O(log n)

додати newNode як лівого або правого нащадка – O(1)
виправити порушення червоно-чорного дерева (fixInsert) – O(log n)
оновити суми в вузлах (updateSum) – O(log n)

Запит суми (query(l, r)) – O(log n)

Int queryPrivate(node, l, r)
якщо поточний вузол повністю входить у [l, r] → повернути його суму – O(1)
якщо поточний вузол не перетинається з [l, r] → повернути 0 – O(1)
обчислити суму у лівому піддереві (queryPrivate) – O(log n)
обчислити суму у правому піддереві (queryPrivate) – O(log n)
повернути загальну суму – O(1)

Видалення інтервалу (remove(l, r)) – O(log n)

void removeNode(node, l, r)

знайти вузол для видалення – O(log n)

якщо вузол має двох дітей:

знайти наступника (minimum) – O(log n)

замінити вузол на наступника (transplant) – O(1)

якщо вузол має одного або жодного нащадка:

видалити вузол (transplant) – O(1)

виправити баланс дерева (fixDelete) – O(log n)

оновити суми в батьківських вузлах (updateSum) – O(log n)

Фіксація порушень після вставки (fixInsert(node)) – O(log n)

void fixInsert(node)

ПОКИ parent червоний:

знайти grandparent і uncle – O(1)

якщо uncle червоний → recolor – O(1)

якщо uncle чорний → виконати ротацію (leftRotate/rightRotate) – O(1)

перейти до grandparent – O(1)

root позначити чорним – O(1)

Фіксація порушень після видалення (fixDelete(node)) – O(log n)

void fixDelete(x)

ПОКИ x не root і чорний:

знайти sibling – O(1)

якщо sibling червоний → recolor + ротація – O(1)

якщо обидва нащадки sibling чорні → recolor – O(1)

якщо є хоча б один червоний нащадок → ротація (leftRotate/rightRotate) – O(1)

перейти до батька – O(1)

позначити x чорним – O(1)

Ротації (leftRotate, rightRotate) – O(1)

void leftRotate(x)

змінити батьківські посилання та оновити зв'язки – O(1)

void rightRotate(x)

змінити батьківські посилання та оновити зв'язки – O(1)

Оновлення сум (updateSum(node)) – O(log n)

void updateSum(node)

ПОКИ node не NIL:

оновити sum як сума дітей – O(1)

перейти до батька – O(1)

Знаходження мінімального вузла (minimum(node)) – O(log n)

Node*minimum(node)

ПОКИ node.leftChild != NIL:

перейти до leftChild – O(1)

ПОВЕРНУТИ node

Заміна вузлів (transplant(u, v)) – O(1)

void transplant(u, v)

якщо u – корінь → root = v – O(1)

якщо u – лівий нащадок → parent.leftChild = v – O(1)

якщо u – правий нащадок → parent.rightChild = v – O(1)

встановити parent для v – O(1)

Тестові приклади:

```cpp
SegmentTree tree;

// Adding intervals (correct sum values)
tree.insert(1, 10, 0);
tree.insert(1, 5, 15);
tree.insert(1, 3, 6);
tree.insert(4, 5, 9);
tree.insert(6, 10, 40);
tree.insert(6, 8, 21);
tree.insert(9, 10, 19);

std::cout << "\n========================================";
std::cout << "\nBegin tree:\n";
tree.printTree();

// Query sum tests
std::cout << "\nQuery SUM [1, 3]: " << tree.query(1, 3) << "\n";
std::cout << "Query SUM [6, 10]: " << tree.query(6, 10) << "\n";
std::cout << "Query SUM [4, 5]: " << tree.query(4, 5) << "\n";
std::cout << "Query SUM [6, 8]: " << tree.query(6, 8) << "\n";
std::cout << "Query SUM [1, 8]: " << tree.query(1, 8) << "\n";
std::cout << "========================================\n";
```

```cpp
 //Removing an interval
std::cout << "\n===========Deleting [6, 8]===========\n\n";
tree.remove(6,8);
tree.printTree();

// Query sum after deletion
std::cout << "\nQuery SUM [1, 5] after deletion: " << tree.query(1, 5) << "\n";
std::cout << "Query SUM [6, 10] after deletion: " << tree.query(6, 10) << "\n\n";
```

Тест 1: (цілі числа)

Вхідне дерево:

```
=====================================
Begin tree:
        [9, 10] sum: 19 (Red)
    [6, 10] sum: 40 (Black)
        [6, 8] sum: 21 (Red)
[1, 10] sum: 55 (Black)
        [4, 5] sum: 9 (Red)
    [1, 5] sum: 15 (Black)
        [1, 3] sum: 6 (Red)

Query SUM [1, 3]: 6
Query SUM [6, 10]: 40
Query SUM [4, 5]: 9
Query SUM [6, 8]: 21
Query SUM [1, 8]: 36
=====================================
```

Як можемо бачити в дерево ієрархічно за правилами червоно чорного дерева додаються вузли, коректно оновлюються суми для батьківських вузлів а також вірно знаходяться суми на певних інтервалах.

Випадки видалення вузлів:

- Випадок 1:

```
===========Deleting [6, 8]===========

        [9, 10] sum: 19 (Red)
    [6, 10] sum: 19 (Black)
[1, 10] sum: 34 (Black)
        [4, 5] sum: 9 (Red)
    [1, 5] sum: 15 (Black)
        [1, 3] sum: 6 (Red)

Query SUM [1, 5] after deletion: 15
Query SUM [6, 10] after deletion: 19
```

  У батькківського вузлі з двома синами видаляжом одного сина. Видалення спрацювало коректно, і суми перахувались відповідно

- Випадок 2:

```
===========Deleting [6, 10]===========

[1, 10] sum: 15 (Black)
        [4, 5] sum: 9 (Red)
    [1, 5] sum: 15 (Black)
        [1, 3] sum: 6 (Red)

Query SUM [1, 5] after deletion: 15
Query SUM [6, 10] after deletion: 0
```

  Видаляємо батька двох синів. Видалення спрацювало коретно, тобто спочатку рекурсвино видаляються обидва сини, а потім сам батьківський вузол. Суми перерахувались відповідно.

- Випадок 3:

```
===========Deleting [1, 3]===========

[1, 10] sum: 9 (Black)
        [4, 5] sum: 9 (Red)
    [1, 5] sum: 9 (Black)

Query SUM [1, 5] after deletion: 9
Query SUM [6, 10] after deletion: 0
```

  Робота з іншим піддеревом, після того як одне з них було видалене. Видалення відбулось коректно з перерахованими сумами.

Тест 2 (раціональні числа)

Вхідне дерево:

```
=======================================
Begin tree:
        [9.5, 10.7] sum: 19.6 (Red)
    [6.2, 10.8] sum: 41.4 (Black)
        [6.2, 8.9] sum: 21.8 (Red)
[1.1, 20.9] sum: 57.3 (Black)
        [3.7, 5.8] sum: 9.2 (Red)
    [1.1, 5.9] sum: 15.9 (Black)
        [1.1, 3.7] sum: 6.7 (Red)

Query SUM [1.1, 3.7]: 6.7
Query SUM [6.2, 10.8]: 41.4
Query SUM [4.6, 5.7]: 0
Query SUM [6.2, 8.9]: 21.8
Query SUM [1.2, 8.9]: 31
=======================================
```

Як і для цілих чисел, у побудові дерева з раціональними даним ієрархічно втсавлені вузли, а також коректно обраховані суми.

Видалення вузлів:

- Як і для попереднього тексту було провдеено три теста для видалення, а саме
  - Сина батька
  - Батька двох синів
  - Видалення елемнту з іншого піддерева після того, як одне з них було повністю видалене

```
===========Deleting [9.5, 10.7]===========

    [6.2, 10.8] sum: 21.8 (Black)
        [6.2, 8.9] sum: 21.8 (Red)
[1.1, 20.9] sum: 37.7 (Black)
        [3.7, 5.8] sum: 9.2 (Red)
    [1.1, 5.9] sum: 15.9 (Black)
        [1.1, 3.7] sum: 6.7 (Red)

Query SUM [1.1, 5.9] after deletion: 15.9
Query SUM [6.2, 10.8] after deletion: 21.8
```

```
===========Deleting [6.2, 10.8]===========

[1.1, 20.9] sum: 15.9 (Black)
        [3.7, 5.8] sum: 9.2 (Red)
    [1.1, 5.9] sum: 15.9 (Black)
        [1.1, 3.7] sum: 6.7 (Red)

Query SUM [1.1, 5.9] after deletion: 15.9
Query SUM [6.2, 10.8] after deletion: 0
```

```
===========Deleting [1.1, 3.7]===========

[1.1, 20.9] sum: 9.2 (Black)
        [3.7, 5.8] sum: 9.2 (Red)
    [1.1, 5.9] sum: 9.2 (Black)

Query SUM [1.1, 5.9] after deletion: 9.2
Query SUM [6.2, 10.8] after deletion: 0
```

Як можемо бачити, для всіх трьох випадків елементи були коректно видалені та суми задітих вузлів були відповідно коректно перераховані.

Висновок:

У даній роботі було реалізовано дерево відрізків на основі червоно-чорного дерева, що поєднує ефективність операцій на інтервалах із балансуванням червоно-чорного дерева. Реалізовані алгоритми вставки, запиту суми на інтервалі та видалення забезпечують логарифмічну складність основних операцій, що є важливим для роботи з великими наборами даних.

Реалізація включає механізми балансування дерева за допомогою ротацій та коригування кольорів вузлів, що гарантує дотримання властивостей червоно-чорного дерева. Це дозволяє зберігати високу швидкодію при оновленні та обробці діапазонів значень.

Використані джерела:

- Лекція 2 з АіС
- https://www.geeksforgeeks.org/segment-tree-data-structure/
- https://en.wikipedia.org/wiki/Red%E2%80%93black_tree