КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра

спеціальності 122 «Комп'ютерні науки»

на тему:

ПЕРСИСТЕНТНІ СТРУКТУРИ ДАНИХ

Виконав студент 4-го курсу
Філіпенков Ілля Григорович _____

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Зубенко Віталій Володимирович _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань
Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«___»_____2021 р.
Протокол №___
Завідувач кафедри
Нікітченко М. С. _____

Київ – 2021

ЗМІСТ

РЕФЕРАТ

Обсяг роботи: 48 сторінок, 4 розділи, 5 додатків, 10 ілюстрацій, 4 таблиці та 14 джерел посилань.

ДОСЛІДЖЕННЯ, АЛГОРИТМ, МАСИВ, ГРАФ, СТЕК, ЧЕРГА, ДЕРЕВО, БІНАРНЕ ДЕРЕВО, ДЕРЕВО ВІДРІЗКІВ, СТРУКТУРА ДАНИХ, ПЕРСИСТЕНТНА СТРУКТУРА ДАНИХ, ШВИДКОДІЯ

Об’єктом дослідження є структури зберігання даних та методи їх обробки та реалізації. Предметом дослідження є реалізація персистентних структур даних та дослідження витрат ресурсів на їх обробку.

Метою роботи є дослідження персистентних структур даних, а саме стеку, черги та дерева відрізків, ефективних алгоритмів їх побудови, їх програмної реалізації та витрат часу на обробку різних за розміром даних за допомогою них.

Методи дослідження: алгоритмічне програмування, алгоритми роботи з бінарними деревами. Інструменти дослідження: мова програмування C++, середовище розробки Code::Blocks.

Результати роботи: досліджено персистентні структури даних, а саме стек, чергу і дерево відрізків, вирахувано асимптотичну складність роботи, вказані переваги та недоліки. Отримано нові знання про швидкодію досліджуваних структур даних та обсяги пам’яті, які вони використовують. Наведено приклади програмної реалізації персистентних структур даних та використання їх у задачах.

ВСТУП

Оцінка сучасного стану об’єкту дослідження. Для вирішення певного класу геометричних задач, а також при розробці деяких прикладних інструментів, виникла необхідність в структурах даних, які можуть запам’ятовувати свої попередні стани і, за необхідності, повертатися до них. Хорошими прикладами використання таких структур є сервіси геопозиціонування, системи управління версіями (git) або редактори документів Microsoft Office, де в процесі створення документу доступна будь-яка його попередня версія. Структури даних, які підтримують таку функціональність, називаються персистентними (англ. persistent data structures). Вперше вони були представлені у 1989 році [1].

Оскільки мова C++ не має збирача сміття, який знадобився би у випадку створення бібліотеки для персистентних структур даних, хорошої бібліотеки для таких цілей наразі не існує. І навіть попри те, що існують бібліотеки для Java, Python, C# та інших мов, у яких ці структури є уже реалізованими, часто виникають ситуації, коли необхідно їх модифікувати, що не є передбаченим у більшості реалізацій.

Актуальність роботи. Сучасні підходи до конструювання та розробки програмного забезпечення вимагають все ширшого застосування нових перспективних способів зберігання і обробки інформації. Якщо стоїть завдання мати доступ до усіх попередніх версій структури за той самий асимптотичний час, то персистентні структури даних є практично незамінними у алгоритмічному програмуванні.

Об’єкт дослідження. Структури зберігання даних та методи їх обробки та реалізації.

Предмет дослідження. Реалізація персистентних структур даних та дослідження витрат ресурсів на їх обробку.

Методи та засоби розробки. Алгоритмічне програмування, алгоритми роботи з бінарними деревами. Інструменти дослідження: мова програмування C++, середовище розробки Code::Blocks.

Мета роботи. Дослідження персистентних структур даних, а саме стеку, черги та дерева відрізків, ефективних алгоритмів їх побудови, їх програмної реалізації та витрат часу на обробку різних за розміром даних за допомогою них.

Для досягнення поставленої мети в роботі визначені наступні **завдання**:

- окреслити теоретичні відомості застосування персистентних структур для обробки даних;
- розглянути методи перетворення структур даних на персистентні;
- описати алгоритми реалізації та програмно реалізувати персистентні версії поширених структур даних;
- здійснити оцінку ресурсів часу та пам’яті, використовуваних описаними структурами;
- навести приклад розв’язання задач за допомогою персистентного дерева відрізків.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ПЕРСИСТЕНТНІ СТРУКТУРИ ДАНИХ

Персистентна структура даних – це структура, що зберігає свої попередні стани при кожній модифікації, таким чином, забезпечуючи можливість роботи з її станом в будь-який відрізок часу [3]. Через те, що при кожній модифікації структура зберігає свою минулу версію, ми можемо працювати не тільки з поточними, але і з даними із попередніх станів структури, тобто її «минулим». Щоб розрізняти одну й ту саму структуру з різних періодів часу, кожен її стан ідентифікується особливим чином (в вигляді числа, вектора чисел чи хеша), даний ідентифікатор для персистентних структур називається її версією.

Класифікують два типи таких структур за функціональністю: персистентні і ретроактивні. У випадку модифікації однієї з попередніх версій персистентної структури даних, зберігається початковий ланцюжок змін, а від старої версії структури відгалужується новий. У той же час, усі наступні версії залишаються незмінними. Отже, зміни минулих версій не вливають на дані у наступних версіях.

Ретроактивні структури даних надають іншу функціональність. При модифікації минулих версій, ці зміни вносяться у всі наступні. Таким чином, після змін стан структури у «теперішньому» буде іншим.

Самі ж персистентні структури даних поділяються на:

- Частково персистентні
- Повністю персистентні
- Конфлюентні
- Функціональні

В частково персистентних структурах даних усі попередні версії доступні лише для читання. Змінюваною ж є лише остання версія. Ланцюжком змін у такому випадку буде звичайна лінія. У якості ідентифікаторів версій можуть бути натуральні числа.

Повністю персистентні структури даних, окрім можливостей частково персистентних, мають іще здатність вносити зміни у будь-яку минулу версію структури. Таким чином, нова версія минулого стану створить нову гілку у графі версій. У роботі розглядаються саме повністю персистентні структури даних.

Конфлюентні структури даних дозволяють об’єднувати дві структури даних в одну (дерева пошуку, які можна зливати воєдино).

Функціональні структури даних повністю персистентні за означенням, оскільки в них неможливо зробити присвоювання, що знищує дані. Тобто змінювати значення змінних після першого присвоєння не можна.

Якщо структура даних функціональна, то вона і конфлюентна, якщо конфлюентна, то і повністю персистентна, якщо повністю персистентна, то і частково персистентна. Однак, бувають структури даних не функціональні, але конфлюентні [4].

РОЗДІЛ 2
МЕТОДИ ДЛЯ ПЕРЕТВОРЕННЯ СТРУКТУРИ ДАНИХ НА ПЕРСИСТЕНТНУ

Існує три основних підходи до того, щоб зробити структуру даних персистетною:

- Повне копіювання
- Копіювання шляху
- Метод «товстих» вузлів

Метод повного копіювання є найбільш тривіальним. При будь-якій модифікації структури даних створюється повна її копія і додається вказівник на неї в таблиці версій. Очевидно, що цей метод є занадто затратним у плані споживання потужності процесора та пам’яті. При цьому, такі витрати пам’яті абсолютно не виправдані, оскільки більшість елементів у структурі залишаються незмінними, але все одно дублюються і зберігаються у пам’яті.

Метод копіювання шляху зменшує кількість елементів, що копіюються. Використовуючи цей підхід, робиться копія лише тих елементів, які є сильно зв’язаними з тим, що модифікується, а не усіх, що є у структурі. Наприклад, для бінарного дерева копіюватимуться всі батьківські елементи по відношенню до того, що зазнав змін. Таким чином, в порівнянні з попереднім методом, зберігається значна кількість пам’яті, однак, частина не змінених компонентів все одно дублюється.

Метод «товстих» вузлів вирішує дану проблему, зводячи витрати пам’яті до мінімуму. Але він є застосовним лише для тих структур даних, які можна представити у вигляді «машини вказівників». «Машина вказівників» – структура даних у якій виконуються такі умови:

- вся структура даних складається тільки з вказівників і вузлів з даними;

- всі вузли структури фіксованого розміру, тобто складаються з константної кількості полів;
- для кожного вказівника є зворотній вказівник.

Отже, якщо структура даних є «машиною вказівників», то її можна представити так, що у кожному її вузлі зберігатиметься лог змін. Тоді при внесенні змін до структури у лог вузла буде вноситися новий запис, а значення поля у самому «товстому» вузлі змінюватися не буде. Запис у логу міститиме версію, коли була здійснена модифікація і нове значення. При зверненні до конкретної версії структури, у вузлі шукається підходяща версію і відповідне значення повертається з лога.

Головним недолік такого методу виникає, коли створюється дуже багато версій. Якщо лог вузла переповнюється, створюється новий вузол, поточним значенням якого є значення максимальної версії старого вузла. У такому випадку може виникнути необхідність обійти всю структуру. Але при більшості модифікацій всі операції над вузлом будуть відбуватися за O(1). Отже, амортизована складність будь-яких операцій над «товстими» вузлами буде O(1).

Розглянемо два останніх методи детальніше.

2.1 Метод копіювання шляху

Розглянемо цей метод на збалансованому дереві пошуку. Всі операції в ньому робляться за O(h), де h – висота дерева, а висота дерева O(log(n)), де n – кількість вершин. Нехай необхідно зробити якесь оновлення в цьому збалансованому дереві, наприклад, додати черговий елемент, але при цьому необхідно не втратити старе дерево. Замість того, щоб додавати нового сина у якийсь вузол, необхідно зробити копію цього вузла, і вже до неї додати нового сина. Окрім цього необхідно скопіювати усі вузли, з яких є досяжним перший скопійований вузол, починаючи з кореня, разом з усіма вказівниками. Таким

чином, вершини, з яких модифікований вузол є недосяжним, залишаються незміненими. Кількість нових вершин завжди буде рівна висоті дерева, тобто рівна логарифму від кількості листків. В результаті маємо доступ до обох версій дерева.

Оскільки розглядається збалансоване дерево пошуку, то, піднімаючи вершину вверх при балансуванні, необхідно робити копії всіх вершин, що беруть участь у обертаннях, у яких змінилися посилання на дітей. Оскільки таких вершин завжди не більше трьох, асимптотика O(log(n)) не постраждає. Коли балансування закінчиться, необхідно дійти вверх до кореня, копіюючи вершини на шляху.

Метод копіювання шляху гарно працює у тих структурах даних, де до модифікованого елемента прив’язано відносно небагато елементів (порядка логарифма). Наприклад, на стеку або двійкових деревах. Але у випадку перетворення черги в персистентну, операція додавання буде дуже дорогою, оскільки з хвостом черги зв’язані усі елементи, тобто змінювати треба буде всю структуру. Також доволі затратно застосовувати цей метод і у випадку, коли в структурі даних є посилання на предка.

2.2 Метод «товстих» вузлів

Нехай стоїть задача змінити значення у певному вузлі структури даних (наприклад, на рисунку 2.1 в першій версії структури даних у вузлі X є поле b = 3, а у другій версії це поле повинне бути рівним 4), але при цьому необхідно зберегти доступ до старої версії вузла. Витратами часу можна знехтувати. Тоді можна зберігати обидві версії вузла в одному великому комбінованому «товстому» вузлі.

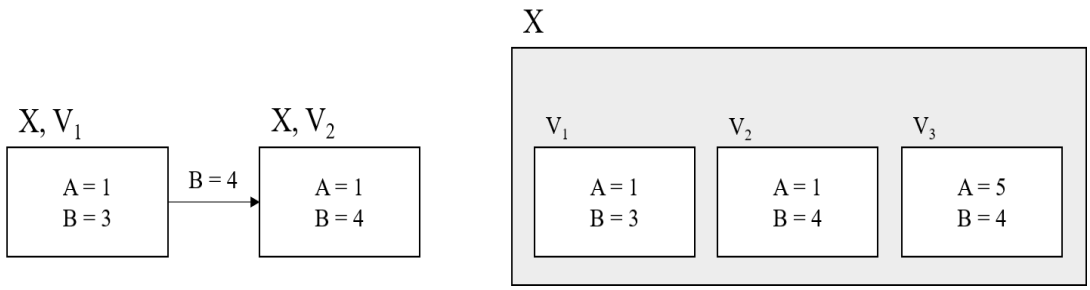

Рис. 2.1. Зміна $b = 4$

На рисунку 2.1 у «товстому» вузлі X зберігатимуться обидві версії V_1 і V_2, у яких $b = 3$ і $b = 4$ відповідно. При подальшому внесенні змін у «товстий» вузол X додається ще одна версія - V_3.

Нехай треба зробити запит до деякої версії структури даних. В такому випадку треба у самому вузлі здійснити пошук по списку версій і знайти максимальну версію, яка менша або дорівнює версії запиту. Коли вона знайдена, можна виконувати заплановані операції зміни або читання. Для швидкого пошуку по версіям, їх треба зберігати у вигляді дерева. Тоді пошук необхідної версії дасть погіршення асимптотики будь-якої операції на структурі даних на логарифм від кількості версій.

Інший варіант реалізації «товстого» вузла зберігає лог змін для кожної версії. Один запис у логу містить ідентифікатор версії, у якій відбулася зміна і саму зміну. Організація логу вершини може варіюватися. У більшості випадків зберігається окремий лог для кожного поля вершини. При модифікації вузла лог доповняється записом про номер версії та нове значення поля. Записи у логу так само варто зберігати у вигляді дерева для швидкого пошуку потрібної версії за логарифм.

Отже, ці варіанти не сильно відрізняються одне від одного, і асимптотика у них однакова. Метод «товстого» вузла працює за $O(\log(t))$, де t – кількість змін структури даних; пам’яті необхідно $O(n + t)$, де n – число вершин у структурі даних [3].

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПЕРСИСТЕНТНИХ ВЕРСІЙ ПОШИРЕНИХ СТРУКТУР ДАНИХ

3.1 Персистентний стек

Стек - послідовний список зі змінною довжиною, включення і виключення елементів з якого виконуються з одного боку списку, званого вершиною. Основні операції над стеком - включення нового елемента і виключення елемента з стека. Корисними можуть бути також такі операції, як визначення поточного числа елементів у стеку і очищення стека [5].

3.1.1 Постановка задачі

Спочатку існує один пустий стек з номером 0, n (кількість стеків) рівна одному. Необхідно реалізувати наступні операції:

- push(i, x) – додати елемент x в стек номер i. Результуючий стек буде мати номер $n + 1$.
- pop(i) – повернути останній елемент стека номер i і «викинути» його із стека. Результуючий стек буде мати номер $n + 1$.

Якщо узагальнити, то після кожної операції треба створити «новий» стек, не видаляючи старий.

3.1.2 Розв’язок

Наївний розв'язок даної задачі – копіювання стека при кожній операції, тобто симулювання описаного процесу. Його очевидним недоліком є неефективність. Таке рішення буде використовувати $O(n \cdot n)$ пам’яті та $O(n)$ часу.

Щоб оптимізувати розв’язок, необхідно представити стек у вигляді графа. Нехай кожна вершина графа – це елемент стеку. Тоді від усіх вершин, окрім хвоста пустимо ребро в попередній елемент стека, що і буде посиланням

на нього. На рисунку 3.1 наведено приклад стеку, у який послідовно додали числа від одного до п’яти.

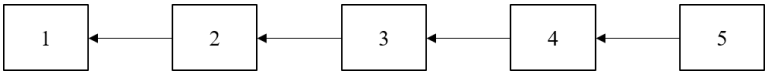

Рис. 3.1. Приклад стеку

Таким чином, для відновлення стеку необхідно знати лише його перший елемент, тобто «голову». Тоді було б набагато оптимальніше зберігати тільки *n* перших елементів. Операції push і pop можна описати наступним чином:

- push(*x*, *i*) – додає новий елемент у «голову» зі значенням *x*, який посилається на елемент з номером *i*, який у свою чергу є «головою» якоїсь версії, як на попередній елемент у стеку.
- pop(*i*) – повертає значення, що зберігається в елементі з номером *i* і робить копію попереднього елементу.

3.1.3 Приклад

Для кращого розуміння, треба розглянути алгоритм на прикладі [6]. Нехай спочатку існує один пустий стек. Збережемо його як «голову» з поміткою, що стек пустий (рис. 3.2).

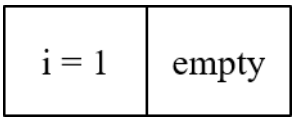

Рис. 3.2. Пустий стек

Виконаємо операцію push(1, 3). Додається нова вершина зі значенням 3, попередньою до неї є перша (рис.3.3).

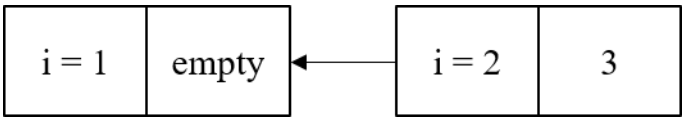

Рис. 3.3. Стек з одного елемента

На рисунку 3.4 зображений граф після операцій push(2, 7) і push(1, 4).

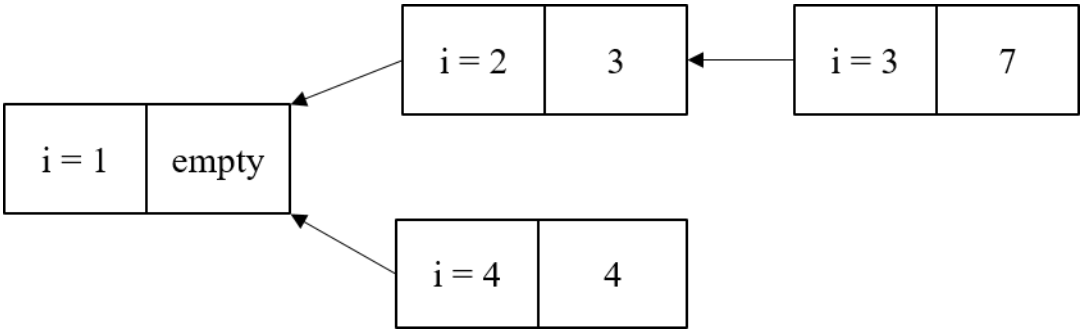

Рис. 3.4. Додано 3 елементи

Зберігаючи голови кожного зі стеків, легко можемо відновити будь-який з чотирьох уже створених. Виконуючи послідовно pop(2) і pop(3), отримуємо новий пустий стек, оскільки pop(2) повертає 3 і копіює першу вершину.

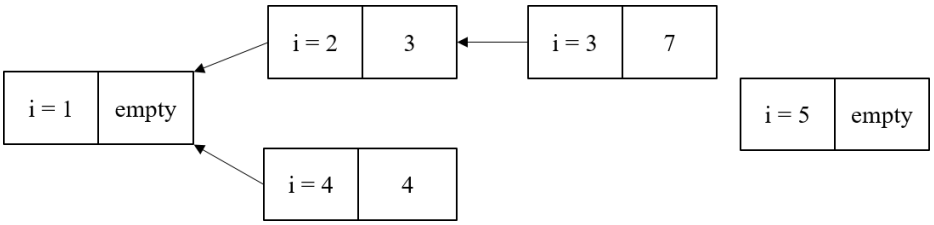

Рис. 3.5. Новий пустий стек

На рисунку 3.6 отримали шостий стек, виконавши операцію pop(3), яка повернула 7 і скопіювала другу вершину.

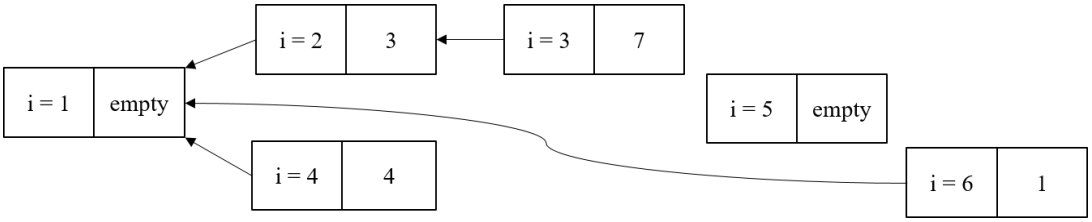

Рис. 3.6. Отримуємо шостий стек

3.1.4 Програмна реалізація

Зберігатимемо масив елементів стеку у вигляді структури, що містить значення та посилання на попередній елемент.

```
struct Node
{
    int value;
    int previous;
```

```
};
```

Функція додавання елементу в стек:

```
void push(int x, int y)
{
    ++all;
    z = last[x];
    last[i] = all;
    nodes[all].value = y;
    nodes[all].previous = z;
}
```

Функція вилучення елементу зі стеку:

```
int pop(int x)
{
    z = last[x];
    last[i] = nodes[z].previous;
    return nodes[z].value;
}
```

З повним кодом реалізації можна ознайомитися у Додатку А.

3.1.5 Оцінка використовуваних ресурсів часу та пам’яті

Оскільки ми не зберігаємо ніяких додаткових елементів, то обсяг пам’яті, необхідний для структури – O(n), де n – кількість операцій додавання елемента. Один запит на додавання або вилучення елемента працюватиме за O(1).

Для перевірки точного часу роботи програми було згенеровано файли з різною кількістю вхідних даних. Файли з вхідними даними включають в себе кількість операцій (n) та n рядків з власне операціями. Опис операції складається з двох чисел: номер версії, яка буде модифікуватися і значення, яке буде додаватися у стек (для операції push) або 0, якщо виконується видалення «голови» стеку (операція pop).

Під час самого генерування спочатку випадковим чином обиралася версія для операції, яка обов’язково мала не перевищувати кількість уже згенерованих операцій. Потім випадково обиралася одна з двох операцій і, якщо це операція push, додатково генерувалося випадкове число від 1 до 10 для додавання у стек.

Перевірка точного часу виконання програми проводилася на двох комп’ютерних конфігураціях:

- процесор - Intel(R) Core(TM) i7-3537U CPU 2.5 GHz, оперативна пам’ять – 8 GB (далі – Конфігурація 1)
- процесор - Intel(R) Core(TM) i3-4000M CPU 2.4 GHz, оперативна пам’ять – 8 GB (далі – Конфігурація 2)

За допомогою наперед генерованих тестів було перевірено точний час роботи програми для різних вхідних даних (таблиця 3.1).

Таблиця 3.1. Час роботи персистентного стека.

Кількість операцій (n)	$2 \cdot 10^5$	$2 \cdot 10^6$	$2 \cdot 10^7$	10^8
Конфігурація 1	420 мс	4315 мс	47691 мс	247965 мс
Конфігурація 2	469 мс	5219 мс	56059 мс	301080 мс
Різниця	12%	21%	18%	21%

3.2 Персистентна черга

Чергою є послідовний список зі змінною довжиною, в якому включення елементів виконується тільки з одного боку списку (кінець або хвіст черги), а виключення - з іншого боку (початок або голова черги). Основні операції над чергою - ті ж, що і над стеком - включення, виключення, визначення розміру, очищення, неруйнуюче читання [5].

3.2.1 Постановка задачі

Спочатку існує пуста черга з номером 0, *n* (кількість черг) рівна одному. Необхідно реалізувати наступні операції:

- push(*i*, *x*) – додати елемент *x* в голову черги номер *i*. Результуюча черга буде мати номер *n* + 1.
- pop(*i*) – повернути останній елемент черги номер *i* і «викинути» його із черги. Результуюча черга буде мати номер *n* + 1.

Якщо узагальнити, то після кожної операції треба створити «нову» чергу, не видаляючи стару.

3.2.2 Розв'язок

Найпростіший розв'язок – використання двох стеків: з першого вилучаємо елементи, у другий додаємо і, коли необхідно, перекидаємо всі елементи з другого в перший. Якщо обидва стеки будуть персистентними, то і черга в цілому стане такою. Незважаючи на те, що асимптотична оцінка складності такого рішення рівна O(1) на операцію, це рішення не є оптимальним. Дана оцінка є амортизованою, а при перетворенні структури даних на персистентну, амортизовані оцінки не зберігаються. Тому побудуємо персистентну чергу за допомогою чотирьох стеків.

У першому стеку (*first*) зберігатимемо усі елементи, які коли-небудь додавалися в чергу. Операція push буде додавати елемент в *first*, а потім перераховувати лічильники і заповнювати стек *forth*.

Другий стек (*second*) необхідний, щоб з нього завжди можна було вилучати елементи під час операції pop. Тобто, операція pop буде брати елемент з *second*, перераховувати лічильники і заповнювати стек *forth*.

Третій стек (*third*) буде персистентною копією стека *first*, з якої будуть братися елементи і додаватися в *forth*.

Четвертий стек (*forth*) використовуватимемо для того, щоб поповнювати другий стек, щоб у ньому не закінчилися елементи. Тобто, він буде містити деякі елементи черги, буде зростати, і коли *second* закінчиться, прийде на заміну.

Схематично реалізація представлена на рисунку 3.7.

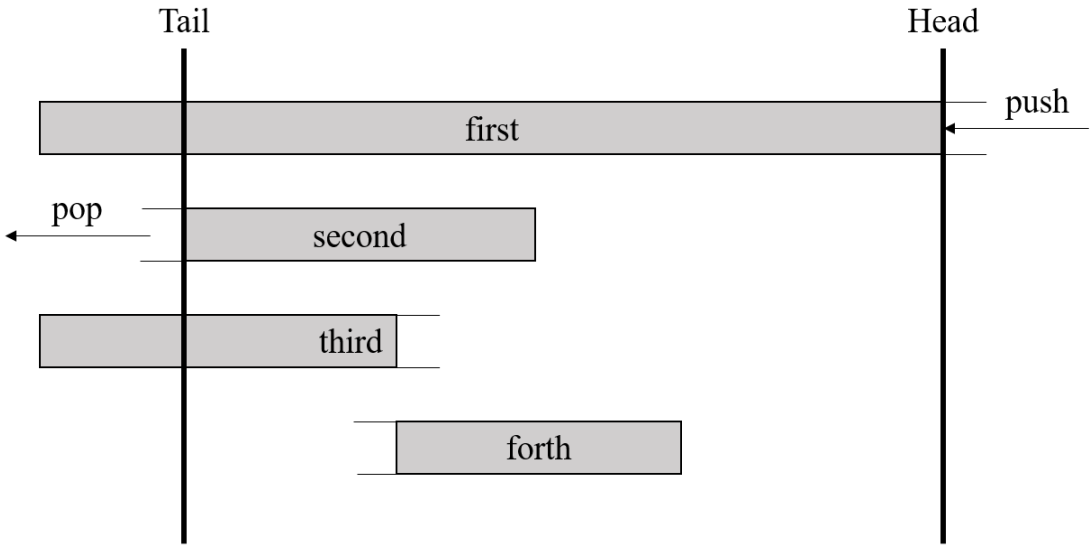

Рис. 3.7. Персистентна черга на чотирьох стеках

Заповнюючи стек *forth*, будемо підтримувати два лічильники. Необхідно знати кількість елементів в черзі (*first_size*) і кількість елементів, які треба перекинути в додатковий стек (*forth_size*). Відповідно, під час операції push збільшуємо *first_size* на одиницю, а під час pop зменшуємо обидва лічильники на один.

Алгоритм заповнення стека *forth*:

1. Якщо *forth_size* > 0, то переміщаємо один елемент з *third* в *forth* і зменшуємо *third_size* на одиницю.
2. Якщо *forth_size* = 0, то заміняємо *second* на *forth*, а *forth* і *third* заміняємо на порожні стеки.
3. Якщо *forth* порожній, то копіюємо *first* в *third* (за O(1)) і присвоюємо *forth_size* = *first_size*.

У порівнянні з реалізаціями персистентної черги на п’яти або шести стеках, можна помітити два недоліки. Вони виникли через те, що в стеку *first* зберігаються взагалі всі елементи, які були додані в чергу, а не тільки ті, які залишились в черзі на даний момент.

По-перше, використання пам’яті. Якщо є умова, що використовувати треба не всі версії, а тільки якісь конкретні, то у якості оптимізації алгоритму можна збирати сміття і таким чином економити пам’ять. У іншому випадку, старі елементи так і будуть висіти у хвості *first*.

По-друге, якщо необхідно реалізувати якусь додаткову функцію, наприклад, підрахунок суми поточних елементів черги, треба буде використовувати додаткові стеки [7].

3.2.3 Програмна реалізація

Клас для зберігання черги має у якості полів розмір черги та чотири стеки. Метод додавання елементу у кінець черги:

```
const Queue* push( int value ) const {
      return Queue(first_size + 1, forth_size, first->push(value), 
second, third, forth).morph();
    }
```

Метод вилучення елементу з голови черги:

```
const Queue* pop() const {
      assert(first_size);
      return  Queue(first_size  -  1,  forth_size  ==  -1  ?  -1  : 
forth_size - 1,
            first, second->pop(), third, forth).morph();
    }
```

З повним кодом реалізації можна ознайомитися у Додатку Б.

3.1.4 Оцінка використовуваних ресурсів часу та пам’яті

Оскільки ми не зберігаємо ніяких додаткових елементів, то обсяг пам’яті, необхідний для структури – O(n), де n – кількість операцій додавання

елемента. Один запит на додавання або вилучення елемента працюватиме за O(1).

Для перевірки точного часу роботи програми було згенеровано файли з різною кількістю вхідних даних. Файли з вхідними даними включають в себе кількість операцій (*n*) та *n* рядків з власне операціями. Опис операції складається з двох чисел: номер версії, яка буде модифікуватися і значення, яке буде додаватися у кінець черги (для операції push) або -1, якщо виконується видалення з початку черги (операція pop).

Під час самого генерування спочатку випадковим чином обиралася версія для операції, яка обов'язково мала не перевищувати кількість уже згенерованих операцій. Потім випадково обиралася одна з двох операцій і, якщо це операція push, додатково генерувалося випадкове число від 1 до 10 для додавання у чергу.

За допомогою наперед генерованих тестів було перевірено точний час роботи програми для різних вхідних даних на двох конфігураціях комп'ютера, що були вказані у підрозділі 3.1 (таблиця 3.2).

Таблиця 3.2. Час роботи персистентної черги.

Кількість операцій (*n*)	10^3	10^4	10^5	10^6
Конфігурація 1	3 мс	22 мс	132 мс	2532 мс
Конфігурація 2	3 мс	26 мс	156 мс	3015 мс
Різниця	0%	18%	18%	19%

3.3 Персистентне дерево відрізків

Деревом відрізків називається структура даних, яка дозволяє для даного масиву A швидко виконувати наступні операції:

- change(*i*, *x*) – змінити значення A[*i*] на *x*

- F(i, j) – порахувати f(A[i], A[i + 1],…, A[j])

Обидві операції виконуються за O(log(n)). В якості функції f зазвичай беруть суму, мінімум або максимум.

3.3.1 Постановка задачі

Нехай існує дерево відрізків з усіма елементами рівними нулю. Спочатку існує лише нульова версія дерева (n = 0). Необхідно реалізувати наступні операції:

- get(x, i) – отримати елемент під номером x з версії номер i. Нова версія не додається.
- change(x, i, y) – змінити елемент під номером x з версії номер i на нове значення y. Додається нова версія дерева під номером n + 1.
- get_sum(i, l, r) – отримати суму значень елементів з версії номер i, початковий елемент - l, кінцевий – r. Нова версія не додається.

3.3.2 Розв’язок

Найпростіший з багатьох способів перетворення дерева відрізків на персистентне – при кожній модифікації вносити зміни у повну копію попередньої версії так само, як у звичайне дерево відрізків. Очевидно, що цей спосіб є дуже затратним, в першу чергу, по пам’яті, а також погіршує асимптотику операції change до O(n).

Використовуючи метод копіювання шляху, можна оптимально розв’язати дану задачу. Для цього, в порівнянні зі звичайним деревом відрізків, додатково будемо зберігати посилання на дочірні вершини. Також додатково знадобиться зберігати масив вершин, які є коренями у відповідних версіях дерева. При побудові до нього додаватиметься єдина вершина, яка і буде коренем отриманого дерева. На рисунку 3.8 зображено побудову першої версії персистентного дерева відрізків.

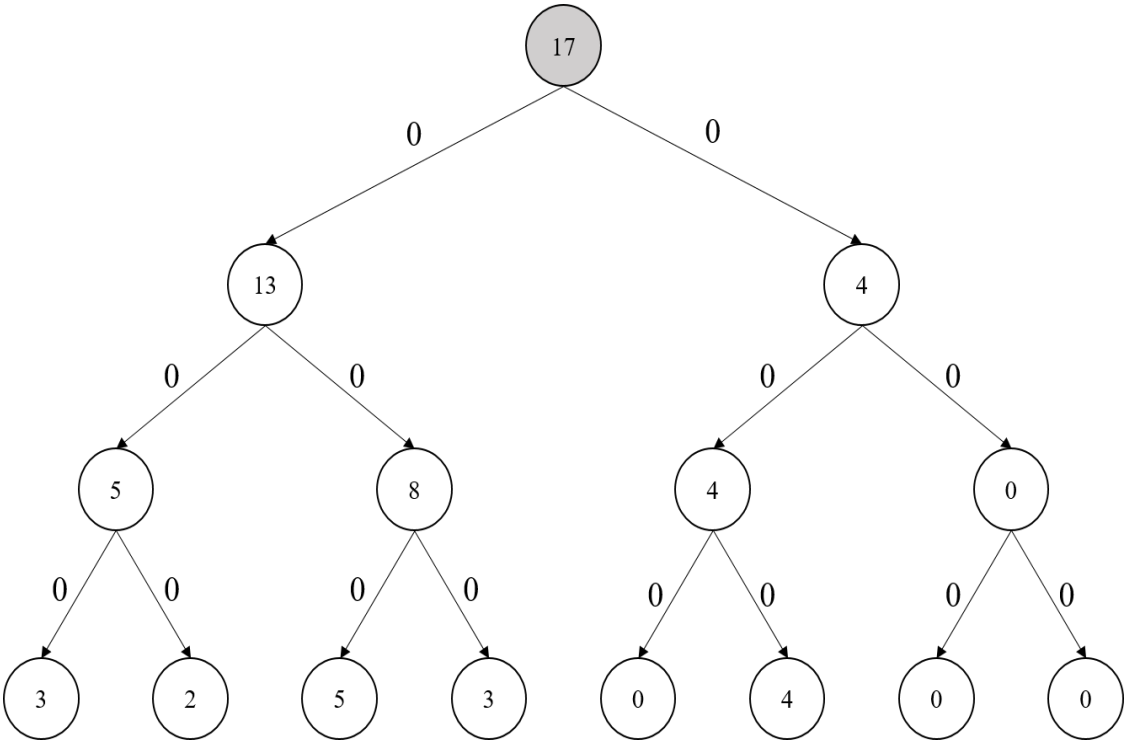

Рис. 3.8. Побудова персистентного дерева відрізків

При зміні значення у деякій вершині, згідно з методом копіювання шляху, до дерева будуть додаватися тільки ті вершини, які мали змінитися. Перераховані значення зберігатимуться у нових вершинах, замість старих. Всі нові вершини будуть посилатися на одну вершину із дерева старої версії і на одну з тільки що доданих в якості лівого і правого синів.

Додавання нової гілки (рис. 3.9) відбувається аналогічно побудові всього дерева з тою відмінністю, що замість побудови двох дочірніх вершин від кожної, будується лише одна в тому напрямі, в якому лежить змінена вершина. Після цього додаємо новий корінь у масив кореневих вершин.

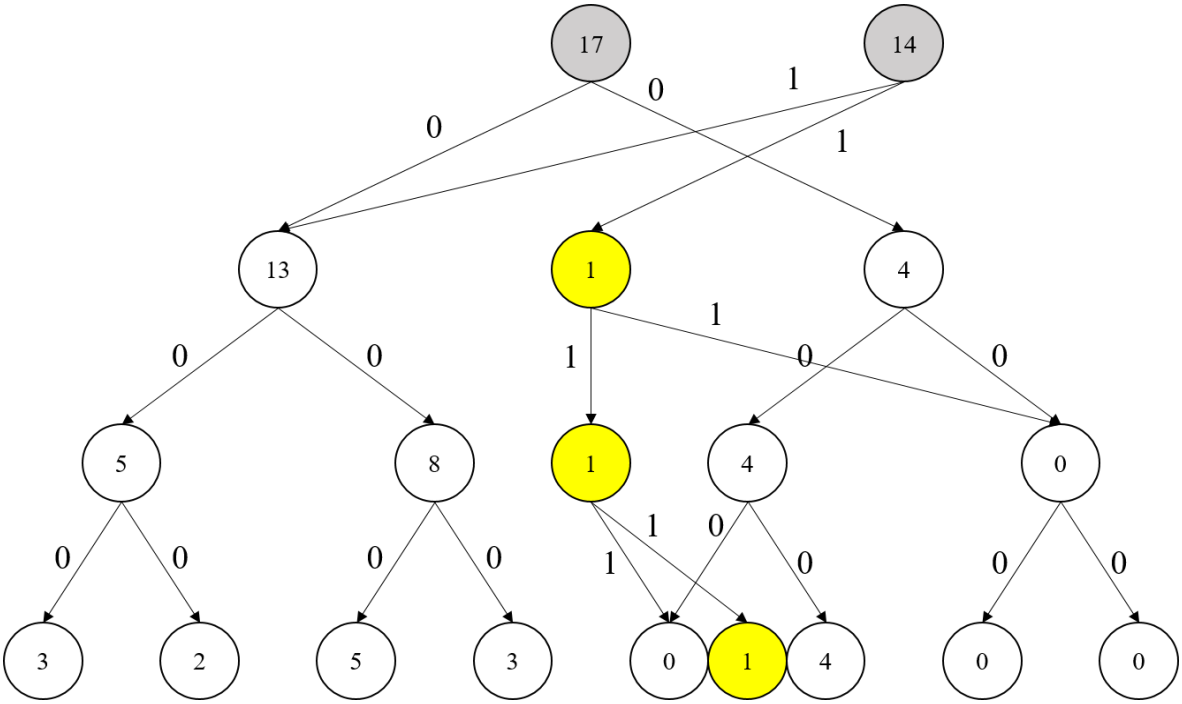

Рис. 3.9. Додавання нової гілки

Відмінності в обчисленні F в порівнянні з ефемерним деревом полягають в переходах до дочірніх вершин і запуску з різних кореневих вершин [8].

3.3.3 Програмна реалізація

Структура для зберігання вузла дерева:

```
struct Node
{
    int left, right, val;
    Node()
    {
        left = 0;
        right = 0;
        val = 0;
    }
};
```

Функція get(*x*, *i*):

```
int get(int i, int l, int r, int k)
{
    if (l == r)
    {
        return(l);
    }

    int d = (l + r) / 2;

    if (v[v[i].left].val >= k)
        return(get(v[i].left, l, d, k));
    else
        return(get(v[i].right, d + 1, r, k - v[v[i].left].val));
}
```

Функція change(*x*, *i*, *y*):

```
void change(int oldRoot, int newRoot ,int segmentLeft, int
segmentRight, int index, int newValue)
{
    if (segmentLeft == segmentRight)
    {
        v[newRoot].val = newValue;
        return;
    }

    int d = (segmentLeft + segmentRight) / 2;
    if (index <= d)
    {
        v[newRoot].right = v[oldRoot].right;
        v[newRoot].left = ++cnt;
        change(v[oldRoot].left, v[newRoot].left, segmentLeft, d,
index, newValue);
        v[newRoot].val = v[v[newRoot].left].val +
v[v[newRoot].right].val;
```

```
        } else
        {
            v[newRoot].left = v[oldRoot].left;
            v[newRoot].right = ++cnt;
            change(v[oldRoot].right, v[newRoot].right, d + 1,
segmentRight, index, newValue);
            v[newRoot].val = v[v[newRoot].left].val +
v[v[newRoot].right].val;
        }
    }
```

Функція get_sum(*i*, *l*, *r*):

```
    int get_sum(int oldRoot,int l,int r,int tl,int tr)
    {
        if (l > tr || r < tl)
            return 0;

        if (l >= tl && r <= tr){
            return v[oldRoot].val;
        }
        int m = (l + r) / 2;
        return get_sum(v[oldRoot].left, l, m, tl, tr) +
get_sum(v[oldRoot].right, m + 1, r, tl, tr);
    }
```

З повним кодом реалізації можна ознайомитися у Додатку В. На основі поданих у розділі реалізацій структур даних було сформовано бібліотеку, з заголовним файлом якої можна ознайомитися у Додатку Г.

3.3.4 Оцінка використовуваних ресурсів часу та пам’яті

Оскільки єдина зміна в порівнянні з ефемерним деревом відрізків – це додавання інформації про ліву і праву дочірні вершини, то складність побудови залишилась незмінною, тобто O(*n*).

Оскільки при зміні тільки додається log(*n*) вершин, то асимптотика операції зміни елемента – O(log(*n*)).

Складність обчислення суми на відрізку - $O(\log(n))$.

Для перевірки точного часу роботи програми було згенеровано файли з різною кількістю вхідних даних. Файли з вхідними даними включають в себе кількість елементів у масиві (n), кількість операцій (m) та m рядків з власне операціями. Опис операції складається з чотирьох чисел: номер версії, яка буде модифікуватися, тип операції (0 – get_sum, 1 – change), якщо операція зміни елементу, то вказується порядковий номер елементу і його нове значення, якщо отримання суми – початок та кінець відрізка, на якому вона шукається. З метою зменшення вхідних файлів, усі елементи масиву на початку рівні 0.

Під час самого генерування спочатку випадковим чином обиралася версія для операції, яка обов’язково мала не перевищувати кількість уже згенерованих операцій зміни елемента. Потім випадково обиралася одна з двох операцій. Якщо це операція change, випадково генерувалося число від 1 до n і значення від 1 до 10, яке стане новим. Якщо це операція обрахунку суми на відрізку, генерувалося випадкове число початку відрізка і кінця такого, щоб відрізок був додатнім.

За допомогою наперед генерованих тестів було перевірено точний час роботи програми для різних вхідних даних на двох конфігураціях комп’ютера, що були вказані у підрозділі 3.1 (таблиця 3.3 і таблиця 3.4).

Таблиця 3.3. Час роботи персистентного дерева відрізків на Конфігурації 1.

	$m = 10^3$	$m = 10^4$	$m = 10^5$	$m = 5 \cdot 10^5$	$m = 10^6$
$n = 10^3$	4 мс	42 мс	406 мс	2500 мс	4359 мс
$n = 10^4$	6 мс	47 мс	448 мс	2377 мс	4584 мс
$n = 10^5$	5 мс	49 мс	558 мс	2431 мс	4956 мс
$n = 10^6$	4 мс	66 мс	521 мс	2605 мс	5162 мс

Таблиця 3.4. Час роботи персистентного дерева відрізків на Конфігурації 2.

	$m = 10^3$	$m = 10^4$	$m = 10^5$	$m = 5 \cdot 10^5$	$m = 10^6$
$n = 10^3$	16 мс	62 мс	440 мс	2498 мс	4796 мс
$n = 10^4$	15 мс	63 мс	484 мс	2531 мс	5172 мс
$n = 10^5$	16 мс	78 мс	579 мс	2734 мс	5515 мс
$n = 10^6$	15 мс	78 мс	609 мс	2906 мс	5874 мс

РОЗДІЛ 4
ПРИКЛАДИ ВИКОРИСТАННЯ ПЕРСИСТЕНТНИХ СТРУКТУР ДАНИХ ДЛЯ РОЗВ'ЯЗУВАННЯ ОЛІМПІАДНИХ ЗАДАЧ

За допомогою персистентного дерева відрізків можна вирішити типову задачу для системного адміністратора, що була представлена на XI Всеросійській учнівській командній олімпіаді, авторство якої належить Антону Банних [9].

4.1 Постановка задачі

Відкат

У обов'язки системного адміністратора входить резервне копіювання інформації, що зберігається на різних серверах і «відкат» до попередньої версії у випадку виникнення проблем.

На даний момент у системний адміністратор бореться з проблемою недостачі місця для зберігання інформації для відновлення. Він вирішив перенести частину інформації на нові сервери. На жаль, якщо щось станеться під час переносу, він не зможе здійснити відкат, тому процедура переносу повинна бути детально спланована.

На даний момент у системного адміністратора зберігається n точок відновлення різних серверів, що пронумеровані від 1 до n. Точка відновлення з номером i дозволяє зробити відкат для сервера a_i. Він вирішив розбити перенесення на етапи, при цьому на кожному етапі у випадку виникнення проблем будуть доступні точки відновлення з номерами $l, l + 1, \ldots, r$ для деяких l і r.

Для того, щоб спланувати перенесення даних оптимальним способом, системному адміністратору необхідно навчитися відповідати на запити: для заданого l, при якому мінімальному r у процесі перенесення будуть доступні точки відновлення не менше, ніж k різних серверів.

Формат вхідного файлу

Перший рядок вхідного файлу містить два цілих числа n і m, що розділені пробілами – кількість точок відновлення і кількість серверів ($1 \le n, m \le 100\ 000$). Другий рядок містить n цілих чисел $a_1, a_2, \dots, a_n$ – номери серверів, яким відповідають точки відновлення ($1 \le a_i \le m$).

Третій рядок вхідного файлу містить q – кількість запитів, які необхідно обробити ($1 \le q \le 100\ 000$). В процесі обробки запитів необхідно підтримувати число p, на початку воно рівне 0. Кожен запит задається парою чисел x_i і y_i, використовуйте їх для отримання даних запиту наступним чином:

$$l_i=((x_i+p) \bmod n)+1,\ k_i=\left(\left(y_i+p\right) \bmod m\right)+1,\ \left(1\le l_i, x_i \le n,\ 1 \le k_i, y_i \le m\right)$$

Нехай відповідь на i-ий запит рівна r. Після виконання цього запиту, слід присвоїти p значення r.

Формат вихідного файлу

На кожен запит виведіть одне число – шукане мінімальне r, або 0, якщо такого не існує.

Обмеження по часу: 2 секунди.

Обмеження по пам'яті: 256 мегабайт.

Приклади

Таблиця 4.1. Приклад вхідних та вихідних даних.

Вхідні дані	Вихідні дані
7 3 1 2 1 3 1 2 1 4 7 3 7 1	1 4 0 6

7 1 2 2	

4.2 Розв’язання

Для початку, сформулюємо умову задачі більш коротко:

- Дано n цілих чисел $a_1, a_2,\ldots, a_n$ і q запитів
- Запит – числа l і k
- Відповідь на запит – мінімальне число r таке, що $a[l..r]$ містить k різних чисел
- Онлайн-відповіді на запити

Розглянемо спрощену задачу. Нехай $l = 1$. Введемо масив b такий, що $b_i = 1$, якщо число a_i не зустрічалося раніше, інакше – $b_i = 0$. Тоді відповіддю на запит буде мінімальне r таке, що $\sum_{i=1}^{r} b_i = k$. Побудуємо дерево відрізків на суму на масиві b. Тоді відповідь на запит буде здійснюватися за $O(\log(n))$ за допомогою «спуску по дереву», оскільки у вершині зберігається сума на відрізку $[l..r]$, а в кожній вершині відомо, куди йти (у лівого сина чи в правого).

Перейдемо до $l = 2$. Тоді $b_1 = 0$. Нехай j – перше входження a_1 в $a[2..n]$. Покладемо $b_j = 1$. Тоді в масиві b змінилось не більше двох елементів. Змінимо дерево відрізків відповідним чином за $O(\log(n))$. Отже, зрозуміло, як відповідати на запит і переходити до наступного суфікса за $O(\log(n))$.

Припустимо, що на запити можна відповідати у довільному порядку. Відсортуємо запити по l і будемо перебирати суфікси у порядку зменшення довжини і відповідати на всі запити для даного суфікса. Тоді час роботи $O(n \cdot \log(n))$, а використана пам’ять - $O(n)$.

Варто зазначити, що під час реалізації, для ефективного переходу, необхідно для i-го символу знати перше входження числа a_i в $a[i..n]$. Тому йдемо справа наліво і для кожного числа c запам’ятовуємо його останнє

входження $d[c]$. При обробці a_i записуємо його перше входження в $a[i+1..n]$ і оновлюємо $d[a_i] = i$. Час роботи – O(n), використана пам'ять - O($n + m$). Якщо перебирати суфікси в порядку збільшення довжини, то можна сумістити це з попередньою частиною алгоритму.

Отже, основна проблема – відповідь на запит у режимі онлайн. Можна помітити, що при переході до наступного суфікса змінюється порядка log(n) вершин дерева відрізків. Але на зберігання всіх копій дерева знадобиться занадто багато пам’яті. Саме тому рішенням для даної задачі слугуватиме персистентне дерево відрізків.

Замість зміни значення у вершині створюватимемо нову. Один запит типу set створить log(n) вершин на шляху від кореня до листка. Старе дерево відрізків не змінюється, а новий корінь відповідає зміненому дереву відрізків. Виклик get від різних коренів – все одно, що від різних дерев.

Тоді можна сформулювати остаточний розв’язок основної задачі. Перебираючи усі суфікси, для кожного запам’ятаємо відповідний корінь дерева відрізків. Відповіддю на запит буде виклик get від відповідного кореня. Час роботи алгоритму – O($(n+q) \cdot \log n$), витрачена пам’ять - O($n \cdot \log(n)$).

Програмна реалізація даного алгоритму рішення за допомогою засобів мови C++ наведена у Додатку Д.

ВИСНОВКИ

В даній роботі розглянуто персистентні структури даних, типи персистентності і методи перетворення структури даних на персистентну. Було реалізовано персистентні версії стека, черги і дерева відрізків, а також вирахуваний час роботи програм на різних вхідних даних. Наведено розв’язок олімпіадної задачі за допомогою персистентного дерева відрізків.

Персистентні структури даних – доволі нова і недосліджена тема, тому існує ще велика кількість відкритих питань та теорій, на які ще не було надано відповіді чи які не були доведені. Незважаючи на відносну складність, подібні структури активно використовуються при написанні геометричних алгоритмів а також при побудові різноманітних систем, здатних самовідновлюватися, адже вони можуть зробити «дешеву» копію самих себе.

Оскільки на даному етапі не існує надійних бібліотек на C++ для роботи з персистентними структурами даних, у випадку необхідності їх використання, розробнику доведеться самотужки реалізовувати їх. Саме для цього і можуть бути корисними матеріали даної роботи, оскільки у ній наведено програмну реалізацію цих структур за допомогою алгоритмічного програмування.

Застосування персистентним структурам даних можна знайти у багатьох напрямках. Наприклад, геометричні задачі (місцезнаходження точки), менеджмент стану в UI (персистентний DOM у веб-програмуванні), паралельні алгоритми (заміна для м’ютексів), а також у системах контролю версіями (git).

Серед недоліків персистентних структур даних можна виділити те, що вони не дають можливості виконати операції, які комбінують декілька версій структури. Також слід зазначити, що вдосконаленням таких структур була б можливість змінювати дані у декількох версіях одночасно.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. James R. Driscoll Making Data Structures Persistent / James R. Driscoll, Neil Sarnak, Daniel D. Sleator, Robert E. Tarjan // Journal of computer and system sciences. – 1989. - №38. – p. 86-124.
2. Маврин П. Ю. Дополнительные главы алгоритмов, часть 1: лекция №13 / Маврин П. Ю. // Computer Science Center. – 2019.
3. Соколов Б. С. Персистентные структуры данных и их эффективная реализация [Текст] / Соколов Б. С. – М.: Изд-во. Мир, - 1990 – С. 235.
4. Персистентные структуры данных [Електронний ресурс]. – Режим доступу: https://neerc.ifmo.ru/wiki/index.php?title=Персистентные_структуры_данных.
5. Алгоритми і структури даних [Електронний ресурс] : опорн. консп. лекцій / уклад. В. І. Манжула. - Тернопіль, 2015. - 63 с.
6. Habr [Електронний ресурс]. - Персистентные структуры, часть 1: персистентный стек. – Режим доступу: https://habr.com/ru/post/113585/.
7. Codeforces [Електронний ресурс]. - Персистентная очередь и её друзья. – Режим доступу: https://codeforces.com/blog/entry/15685?locale=ru.
8. Habr [Електронний ресурс]. - Персистентные деревья отрезков. – Режим доступу: https://habr.com/ru/post/142572/.
9. Codeforces [Електронний ресурс]. - 2010-2011 Всероссийская командная олимпиада школьников по программированию (ВКОШП 10). – Режим доступу: https://codeforces.com/gym/100043.
10. David Karger Advanced Algorithms Lecture 2: September 9, 2005 / David Karger. – 2005.
11. Персистентный стек [Електронний ресурс]. – Режим доступу: https://neerc.ifmo.ru/wiki/index.php?title=Персистентный_стек.

12. Персистентная очередь [Електронний ресурс]. – Режим доступу: https://neerc.ifmo.ru/wiki/index.php?title=Персистентная_очередь.
13. Персистентный дек [Електронний ресурс]. – Режим доступу: https://neerc.ifmo.ru/wiki/index.php?title=Персистентный_дек.
14. Персистентні структури даних [Електронний ресурс]. – Режим доступу до ресурсу: https://int-konf.org/ru/2015/pitannya-suchasnoji-nauki-i-osviti-14-16-07-2015-r/1101-russin-o-s-persistentni-strukturi-danikh.

ДОДАТОК А

Код реалізації персистентного стека

```
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    int value;
    int previous;
};

int n, x, y;
int last[200005];
int all;
int z;
int i;
Node nodes[200005];

void push(int x, int y)
{
    ++all;
    z = last[x];
    last[i] = all;
    nodes[all].value = y;
    nodes[all].previous = z;
}

int pop(int x)
{
    z = last[x];
    last[i] = nodes[z].previous;
    return nodes[z].value;
}

int main(int argc, const char * argv[]) {
    cin >> n;
    i = 1;
    while(i <= n)
    {
        cin >> x >> y;
        if (y == 0) {
```

```
            cout << pop(x) << '\n';
        } else {
            push(x, y);
        }
        i++;
    }
}
```

ДОДАТОК Б

Код реалізації персистентної черги

```
#include <bits/stdc++.h>

using namespace std;

#define files(s) freopen(s".in","r",stdin); freopen(s".out","w",stdout);

class Stack {
  private:
    int value;
    const Stack *next;
    Stack( int _value, const Stack *_next = 0 ) : value(_value), next(_next) { }
  public:
    const Stack* push( int _value ) const {
      return new Stack(_value, this);
    }
    bool empty() const {
      return !this;
    }
    int top() const {
      assert(this);
      return value;
    }
    const Stack* pop() const {
      assert(this);
      return next;
    }
    static const Stack *null;
};
const Stack *Stack::null = 0;

class Queue {
  private:
    int first_size, forth_size;
    const Stack *first, *second, *third, *forth;
    Queue( int _first_size, int _forth_size, const Stack *_first, const Stack
*_second,
       const Stack *_third, const Stack *_forth )
      : first_size(_first_size), forth_size(_forth_size), first(_first),
second(_second),
```

```
        third(_third), forth(_forth) {}
    Queue* morph() {
      if (forth_size == -1) {
        forth_size = first_size;
        forth = Stack::null;
        third = first;
      }
      if (forth_size > 0) {
        forth_size--;
        forth = forth->push(third->top());
        third = third->pop();
      }
      if (forth_size == 0) {
        second = forth;
        forth_size = -1;
        forth = third = Stack::null;
      }
      return new Queue(first_size, forth_size, first, second, third, forth);
    }
  public:
    Queue() : first_size(0), forth_size(-1) {
      first = second = third = forth = Stack::null;
    }
    const Queue* push( int value ) const {
      return Queue(first_size + 1, forth_size, first->push(value), second, third,
forth).morph();
    }
    bool empty() const {
      return first_size == 0;
    }
    int top() const {
      return second->top();
    }
    const Queue* pop() const {
      assert(first_size);
      return Queue(first_size - 1, forth_size == -1 ? -1 : forth_size - 1,
           first, second->pop(), third, forth).morph();
    }
};

const Queue* arr[1000001];
```

```
int main()
{
    int cnt = 1;

    files("queue16");
    int n;

    cin >> n;
    int x, y;

    arr[0] = new Queue();
    for(int i = 1; i <= n; i++)
    {
        cin >> x >> y;

        auto currentQueue = arr[x];
        if(y == -1 && currentQueue -> empty())
        {
            arr[cnt++] = currentQueue -> push(x);
            continue;
        }
        if(y == -1)
        {
            arr[cnt++] = currentQueue -> pop();
        }
        else
        {
            arr[cnt++] = currentQueue -> push(y);
        }
    }
    return 0;
}
```

ДОДАТОК В

Код реалізації персистентного дерева відрізків

```
#include <bits/stdc++.h>

using namespace std;

#define files(s) freopen(s".in","r",stdin); freopen(s".out","w",stdout);
#define int ll

typedef long long ll;

struct Node
{
    int left, right, val;
    Node()
    {
        left = 0;
        right = 0;
        val = 0;
    }
};

Node v[4000005];
int cnt;

int a[4000005], last[4000005], root[4000005];

void change(int oldRoot, int newRoot ,int segmentLeft, int segmentRight, int index,
int newValue)
{
    if (segmentLeft == segmentRight)
    {
        v[newRoot].val = newValue;
        return;
    }

    int d = (segmentLeft + segmentRight) / 2;
    if (index <= d)
    {
        v[newRoot].right = v[oldRoot].right;
        v[newRoot].left = ++cnt;
```

```
        change(v[oldRoot].left, v[newRoot].left, segmentLeft, d, index, newValue);
        v[newRoot].val = v[v[newRoot].left].val + v[v[newRoot].right].val;
    } else
    {
        v[newRoot].left = v[oldRoot].left;
        v[newRoot].right = ++cnt;
        change(v[oldRoot].right, v[newRoot].right, d + 1, segmentRight, index,
newValue);
        v[newRoot].val = v[v[newRoot].left].val + v[v[newRoot].right].val;
    }
}

int get(int i, int l, int r, int k)
{
    if (l == r)
    {
        return(l);
    }

    int d = (l + r) / 2;

    if (v[v[i].left].val >= k)
        return(get(v[i].left, l, d, k));
    else
        return(get(v[i].right, d + 1, r, k - v[v[i].left].val));
}

int get_sum(int oldRoot, int l, int r, int tl, int tr)
{
    if (l > tr || r < tl)
        return 0;

    if (l >= tl && r <= tr){
        return v[oldRoot].val;
    }
    int m = (l + r) / 2;
    return get_sum(v[oldRoot].left, l, m, tl, tr) + get_sum(v[oldRoot].right, m + 1,
r, tl, tr);
}

void build(int ver, int l, int r)
{
```

```
    if (l == r){
        return;
    }
    v[ver].left = ++cnt;
    v[ver].right = ++cnt;
    int m = (l + r) / 2;
    build(v[ver].left, l, m);
    build(v[ver].right, m + 1, r);
}

signed main()
{
    files("tree1313");
    int n;
    cin >> n;

    int q;
    cin >> q;
    int p = 0;
    cnt = 1;

    root[0] = ++cnt;
    build(0, 0, n - 1);

    for(int i = 1; i <= q; i++)
    {

        int x, y, l, k;
        cin >> x >> y >> l >> k;

        if(y == 1)
        {
            root[i] = ++cnt;
            change(root[x], root[i], 0, n - 1, l, k);
        }
        else
        {
            root[i] = root[i - 1];
            get_sum(root[x], 0, n - 1, l, k);
        }
    }
}
```

ДОДАТОК Г

Код заголовного файлу бібліотеки

```
#pragma once

#ifdef PERSISTENTLIBRARY_EXPORTS
#define PERSISTENTLIBRARY_API __declspec(dllexport)
#else
#define PERSISTENTLIBRARY_API __declspec(dllimport)
#endif

class PersistentStack
{
private:
	int size;
	int headNumber;
	PersistentStack(int size, int headNumber);
public:

	struct Node
	{
		int value;
		int previous;
	};

	static int* heads;

	static Node* elements;

	PersistentStack();

	bool isEmpty();

	const PersistentStack* push(int value);

	const PersistentStack* pop();
};

class Stack {
private:
    int value;
```

```
    const Stack* next;
    Stack(int _value, const Stack* _next = 0);
public:
    const Stack* push(int _value) const;

    bool empty() const;

       int top() const;

       const Stack* pop() const;

    static const Stack* null;
};

class PersistentQueue
{
private:
       int first_size, forth_size;
       const Stack* first, * second, * third, * forth;

       PersistentQueue(int _first_size, int _forth_size, const Stack* _first,
              const Stack* _second, const Stack* _third, const Stack* _forth);

       PersistentQueue* morph();

public:

       PersistentQueue();

       bool isEmpty();

       const PersistentQueue* push(int value);

       const PersistentQueue* pop();
};

class PersistentSegmentTree
{
private:
       int size;
       int root;
       PersistentSegmentTree(int size, int root);
```

```
	void build(int root, int l, int r);
public:

	struct Node
	{
		int left, right, val;
		Node()
		{
			left = 0;
			right = 0;
			val = 0;
		}
	};

	static int* roots;

	static Node* elements;

	PersistentSegmentTree();

	const PersistentSegmentTree* change(int index, int value);

	const PersistentSegmentTree* get(int index);

	const PersistentSegmentTree* getSum(int leftIndex, int rightIndex);
};
```

ДОДАТОК Д

Код реалізації розв'язку задачі «Відкат»

```
#include <bits/stdc++.h>

using namespace std;

#define files(s) freopen(s".in","r",stdin); freopen(s".out","w",stdout);
#define int ll

typedef long long ll;

struct Node
{
    int left, right, val;
    Node()
    {
        left = 0;
        right = 0;
        val = 0;
    }
};

Node v[6000000];
int cnt;

int a[200000], last[200000], root[200000];

void change(int oldRoot, int newRoot ,int segmentLeft, int segmentRight, int index, int
newValue)
{
    if (segmentLeft == segmentRight)
    {
        v[newRoot].val = newValue;
        return;
    }

    int d = (segmentLeft + segmentRight) / 2;
    if (index <= d)
    {
        v[newRoot].right = v[oldRoot].right;
        v[newRoot].left = ++cnt;
```

```
        change(v[oldRoot].left, v[newRoot].left, segmentLeft, d, index, newValue);
        v[newRoot].val = v[v[newRoot].left].val + v[v[newRoot].right].val;
    } else
    {
        v[newRoot].left = v[oldRoot].left;
        v[newRoot].right = ++cnt;
        change(v[oldRoot].right,  v[newRoot].right,  d  +  1,  segmentRight,  index,
newValue);
        v[newRoot].val = v[v[newRoot].left].val + v[v[newRoot].right].val;
    }
}

int get(int i, int l, int r, int k)
{
    if (l == r)
    {
        return(l);
    }

    int d = (l + r) / 2;

    if (v[v[i].left].val >= k)
        return(get(v[i].left, l, d, k));
    else
        return(get(v[i].right, d + 1, r, k - v[v[i].left].val));
}

signed main()
{
    files("rollback");
    int n, m;
    cin >> n >> m;

    for (int i = 1;i <= n; i++)
        cin >> a[i];

    for (int i = 1;i <= m; i++)
        last[i] = n + 1;

    cnt = 1;
    int last_root = 1;
```

```
    for (int i = n; i >= 1; i--)
    {
        int new_root = ++cnt;
        change(last_root, new_root, 1, n + 1, last[a[i]], 0);
        last_root = new_root;

        new_root = ++cnt;
        change(last_root, new_root, 1, n + 1, i, 1);
        last_root = new_root;

        last[a[i]] = i;
        root[i] = last_root;
    }

    int q;
    cin >> q;
    int p = 0;
    while(q--)
    {
        int l, k;
        cin >> l >> k;
        l += p;
        l %= n;
        l++;
        k += p;
        k %= m;
        k++;
        if(v[root[l]].val < k)
        {
            cout << 0 << '\n';
            p = 0;
            continue;
        }

        p = get(root[l], 1, n + 1, k);
        cout << p << '\n';
    }
}
```