КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Курсова робота

за спеціальністю 122 Комп'ютерні науки
на тему:

ДЕРЕВО ВІДРІЗКІВ
ТА ЙОГО МОДИФІКАЦІЇ

Виконав студент 3 курсу
Фекете Іван Іванович _____
(підпис)

Науковий керівник:
кандидат фізико-математичних наук, доцент
Зубенко Віталій Володимирович _____
(підпис)

Засвідчую, що в цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ - 2020

ЗМІСТ

РЕФЕРАТ

Обсяг роботи: 41 сторінка, 9 ілюстрацій, 2 таблиці та 12 джерел посилань.

ДОСЛІДЖЕННЯ, АЛГОРИТМ, СТРУКТУРА ДАНИХ, ГРАФ, БІНАРНЕ ДЕРЕВО, МАСИВ, ДЕРЕВО ВІДРІЗКІВ, ПЕРСИСТЕНТНА СТРУКТУРА ДАНИХ, ПРИНЦИП “РОЗДІЛЯЙ І ВОЛОДАРЮЙ”, АСОЦІАТИВНА ОПЕРАЦІЯ, ОНОВЛЕННЯ ДАНИХ, НАЙБІЛЬША ПІДПОСЛІДОВНІСТЬ, ЩО ЗРОСТАЄ, ШВИДКОДІЯ, РОБОТА З ВЕЛИКИМИ ОБСЯГАМИ ДАНИХ

Об’єктом роботи є структура зберігання даних під назвою дерево відрізків. Предметом роботи є реалізація ключових робочих варіантів дерева відрізків та дослідження ресурсів, які витрачають такі реалізації.

Метою роботи є дослідження структури даних дерево відрізків та ефективних алгоритмів його побудови, визначення класу задач, для яких застосування даної структури даних є можливим та оптимальним, систематизація знань про можливі модифікації дерева відрізків для застосування на ширшому класі задач.

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

Результати роботи: досліджено структуру дерева відрізків, описано алгоритм його роботи, вирахувано асимптотичну складність роботи, досліджено різноманітні можливості структури, вказані переваги та недоліки. Отримано нові знання про швидкодію дерева відрізків та та обсяги пам’яті, які використовує ця структура даних, досліджено область задач, які є розв’язними за допомогою дерева відрізків та для яких дана структура становить теоретичний та практичний інтерес. Впроваджено класифікацію модифікацій дерева відрізків, нові підходи до використання структури даних, нові методи розробки персистентних структур даних.

ВСТУП

Оцінка сучасного стану об'єкта дослідження. Структури даних, що базуються на бінарних деревах не є чимось новим для науки. За всю історію існування структур даних було розроблено багато ефективних способів зберігати дані в бінарних деревах, як от бінарні купи та купи Фібоначчі, збалансовані бінарні дерева пошуку на кшталт червоно-чорного дерева, АВЛ-дерева, декартового дерева тощо. Попри те, що найпоширеніші з цих структур вже давно є частинами стандартних бібліотек мов C++, Java, Python та багатьох інших, часто виникає потреба в модифікації дерев пошуку для певних задач способом, що не є передбаченим в стандартних реалізаціях. І тут виникає проблема, адже реалізації більшості збалансованих бінарних дерев пошуку є досить складними для розуміння, що створює подальші проблеми при підтримці і адаптації такого коду для різних цілей.

Актуальність роботи та підстави для її виконання. Ефективна обробка даних являється однією з основних задач програмування. Для вирішення цієї проблеми часто використовують різноманітні структури даних, серед яких важливе місце посідають різні види дерев. Часто досягнути асимптотики $O(log(n))$ для відповіді на запити суми, максимуму, мінімуму серед певних елементів динамічного масиву є нетривіальною задачею. Єдиного підходу до вивчення структур даних для обробки масивів ще немає, тому дана проблема є актуальною та багатогранною для подальшого розвитку.

Метою й завданням роботи є проведення дослідження з питань, пов'язаних із можливостями швидкої обробки масивів за допомогою дерева відрізків.

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

- окреслити теоретичні відомості застосування структури дерева відрізків для обробки масивів;

- розглянути роботу з бінарними деревами пошуку;
- охарактеризувати базове завдання та побудову дерева відрізків;
- реалізувати запити зміни елемента, запити на знаходження суми елементів на відрізку за допомогою мови програмування С++;
- здійснити оцінку ресурсів часу та пам’яті, використовуваних деревом відрізків;
- дослідити модифікації дерева відрізків;
- дати оцінку використання дерева відрізків в поєднанні з іншими алгоритмами, окреслити його переваги та недоліки.

Об’єкт дослідження: методи швидкої обробки інформації.

Предмет досліджень: структура дерева відрізків.

Можливі сфери застосування. Практична цінність дослідження полягає в можливості використання одержаних результатів для більш глибокого розуміння сутності структури дерева відрізків, що сприятиме зростанню ефективності управління даними процесами через підвищення вірогідності та реальності оцінки ресурсів часу та пам’яті, використовуваних деревом відрізків. Сама структура даних може широко використовуватися в комерційній розробці у випадках, коли потрібна висока швидкодія роботи з великими інтервалами даних, впорядкованих за певним індексом. Матеріали наведеної роботи мають теоретичне та практичне значення для науковців, які цікавляться даною проблематикою.

РОЗДІЛ 1.ТЕОРЕТИЧНІ ВІДОМОСТІ ЗАСТОСУВАННЯ СТРУКТУРИ ДЕРЕВА ВІДРІЗКІВ ДЛЯ ОБРОБКИ МАСИВІВ

На практиці часто виникає необхідність в обробці даних у вигляді довільного набору значень, тобто масивів. **Масив —** це кінцева іменована послідовність величин одного типу, які розрізняються за порядковим номером [1].

Дерева пошуку представляють собою структури даних, які підтримують більшість операцій з динамічними множинами: пошук елементів, мінімального та максимального значення, попереднього та наступного елементу, вставку та видалення. Таким чином, дерево пошуку може використовуватись як словник, так і як черга з пріоритетами [6].

Дерево – в інформатиці та програмуванні одна з найпоширеніших структур даних [6]. Формально дерево визначається як скінченна множина T з однією або більше вершин (вузлів, nodes), яке задовольняє наступним вимогам:

- ✓ існує один відокремлений вузол – корінь (*root*) дерева;
- ✓ інші вузли (за виключенням кореня) розподілені серед m 0 непересічних множин $T_1, ..., T_m$ і кожна з цих множин в свою чергу є деревом. Дерева $T_1, ..., T_m$ мають назву піддерев (*subtrees*) даного кореня.

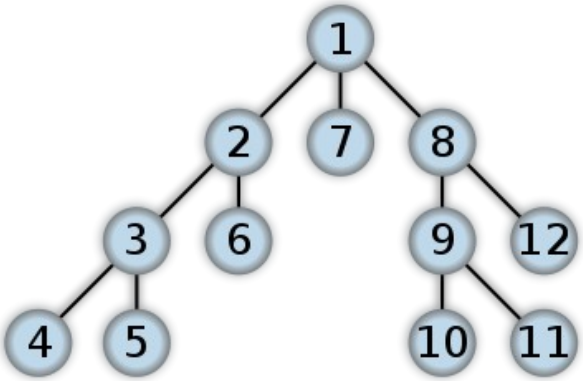

Рис. 1.1. Дерево

Найчастіше дерева в інформатиці зображують з коренем, який знаходиться зверху (говорять, що дерево в інформатиці "росте вниз").

1. Бінарне дерево. Важливим окремим випадком кореневих дерев є бінарні дерева, які широко застосовуються в програмуванні і визначаються як множина вершин, яка має відокремлений корінь та два піддерева (праве та ліве), що не перетинаються, або є пустою множиною вершин (на відміну від звичайного дерева, яке не може бути пустим).

Кожна вершина бінарного дерева, відмінна від кореня, може розглядатися як корінь бінарного ***піддерева*** з вершинами, які є досяжними з неї [2]. Зображення декількох бінарних дерев наведено на рис. 1.2.

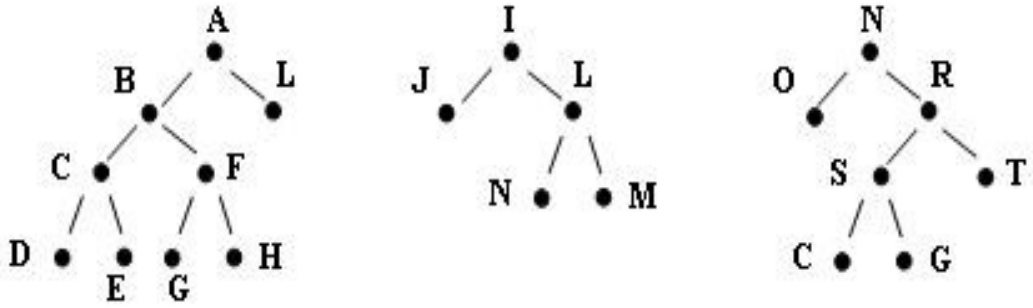

Рис.1.2. Приклади бінарних дерев

В програмуванні бінарне дерево – структура даних, в якому кожна вершина має не більше двох синів. Зазвичай такі сини називаються правим та лівим. На базі бінарних дерев будуються такі структури, як бінарні дерева пошуку та бінарні купи.

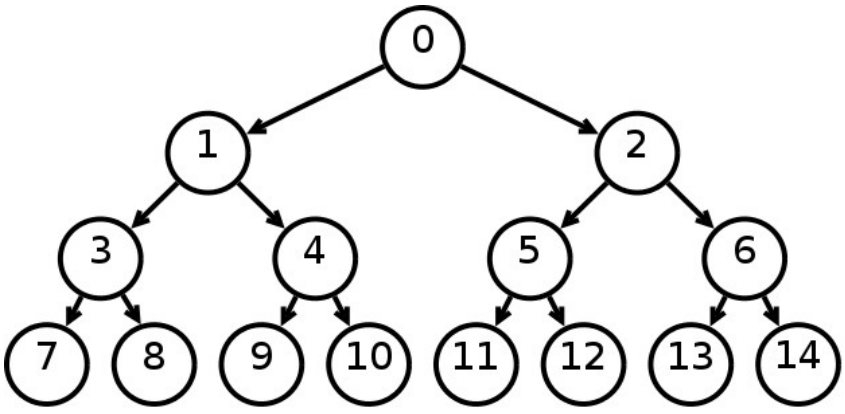

Рис.1.3. Бінарне дерево

✓ **Різновиди бінарних дерев**

Бінарне дерево – таке кореневе дерево, в якому кожна вершина має не більше двох синів.

Повне (закінчене) бінарне дерево – таке бінарне дерево, в якому кожна вершина має нуль або двох синів.

Ідеальне бінарне дерево – це таке повне бінарне дерево, в якому листя (вершини без синів) лежать на однаковій глибині (відстані від кореня).

Бінарне дерево на кожному n-му рівні має від *1* до $2n$ вершин.

✓ **Бінарне дерево на базі масиву**

Бінарні дерева також можуть бути побудовані на базі масивів. Такий метод набагато ефективніший щодо економії пам'яті. В такому представленні, якщо вершина має порядковий номер i, то її діти знаходяться за індексами $2i+1$ та $2i+2$, а батьківська вершина за індексом $\frac{i-1}{2}$ (за умов, що коренева вершина має індекс *0*).

2. Важливі операції на деревах:

- ✓ обхід вершин в різному порядку;
- ✓ перенумерація вершин;
- ✓ пошук елемента;
- ✓ додавання елемента у визначене місце в дереві;
- ✓ видалення елемента;
- ✓ видалення цілого фрагмента дерева;
- ✓ додавання цілого фрагмента дерева;
- ✓ трансформації (повороти) фрагментів дерева;
- ✓ знаходження кореня для будь-якої вершини.

Найбільшого розповсюдження ці структури даних набули в тих задачах, де необхідне маніпулювання з ієрархічними даними, ефективний пошук в даних, їхнє структуроване зберігання та модифікація.

РОЗДІЛ 2. РОБОТА З БІНАРНИМИ ДЕРЕВАМИ ПОШУКУ

2.1. Базове завдання дерева відрізків

Щоб розібратись у ідеї і особливостях роботи дерева відрізків найкраще перейти до конкретної задачі.

Розглянемо масив a із n елементів і m запитів. Запити поділяються на два типи:

- зміна значення певного елементу масиву
- виведення суми чисел в масиві з індексами від l до r

Формат вхідних даних:

Перший рядок містить два цілі числа: n і m ($1 \le n, m \le 10^5$). В наступному рядку задано n елементів масиву a ($1 \le a[i] \le 10^9$). В наступних m рядках задані запити у форматах:

- "1 x y " для запитів першого типу ($1 \le x \le n$, $1 \le y \le 10^9$)
- "2 l r" для запитів другого типу ($1 \le l \le r \le n$)

Формат вихідних даних:

Для кожного запиту другого типу потрібно з нового рядка вивести одне ціле число – відповідь на запит.

Обмеження часу роботи програми: 1 с.

Обмеження пам'яті: 256 МіБ.

Таблиця 1.1

Вхідні та вихідні дані

Вхідні дані	Вихідні дані
5 5 1 2 3 4 5 2 1 5 1 2 3 1 3 2 2 1 1 2 4 5	15 1 9

Реалізація поставленої задачі спонукає спробувати виконати всі запити на зміни, а для запитів другого типу проходитись по масиву від лівого елемента до правого. Однак, слід зауважити, що такий розв'язок не є

оптимальним, адже в гіршому випадку він зробить n^2 операцій (коли всі запити будуть на знаходження суми від першого до останнього елемента). Отже, його асимптотична складність $O(n^2)$.

Іншою ідеєю є порахувати префіксні суми. Таким чином, на запити другого типу можна буде відповідати одразу, але для запитів першого типу їх буде потрібно перераховувати, що знову сповільнює алгоритм.

В обох випадках, при найбільших тестах програма буде працювати значно довше однієї секунди.

Якщо будемо знати суму перших двох елементів, то зможемо відповідаючи на запити суми не додавати їх знову. Це пришвидшить роботу алгоритму, але не сильно. На цій ідеї базується структура дерева відрізків. Знаючи суми на певних великих частинах масиву можна швидше знаходити суму на відрізках. Тепер постає питання, як найоптимальніше порахувати ці суми. Вибравши кожні два елементи, що знаходяться поруч і знайшовши їхні суми алгоритм можна пришвидшити вдвічі. Далі, те саме можна повторити з новим масивом сум і так до тих пір, поки не залишиться один елемент. В результаті отримаємо суму на всьому масиві, суму на його правій і лівій половині і т. д.

Все це зручно представити у вигляді бінарного дерева, коренем якого буде вершина, в якій міститься сума на всьому масиві, а в листках окремі елементи масиву. При цьому кожен елемент буде дорівнювати сумі його синів.

Глибина дерева буде двійковим логарифмом з n (на кожному рівні ми ділимо n на 2).

Щоб змінити елемент в дереві достатньо спуститись по ньому вниз замінити значення однієї вершини, а потім підніматись і оновлювати його для інших. Щоб дізнатись суму на відрізку можна також рекурсивно спускатись по дереву.

Надалі буде розглянуто детальніший опис реалізації окремих процедур і функцій, потрібних для дерева відрізків, а також наведено приклади програм, написаних на C++.

1.2. Побудова дерева відрізків

Нехай кожна вершина нашого дерева буде мати 4 параметри: свій порядковий номер у масиві, значення елемента, що в ній знаходиться, правий та лівий кінці відрізків, що покриває. Значення буде зберігатись у масиві, а інші три параметри будуть передаватись у рекурсивній процедурі.

Важливим фактом у реалізації є те, що якщо наша вершина має номер i, то її сини будуть мати номери $2i$ та $2i + 1$. При цьому не будуть двох вершин з однаковим номером, а також невикористаних номерів (доводиться тим, що на кожному рівні вершин є вдвічі більше, ніж на попередньому). Це дозволяє легко зберігати дерево у звичайному масиві, не створюючи структур та економлячи пам’ять.

Реалізувати побудову такого графа неважко, за допомогою рекурсивної процедури, яка буде отримувати номер вершини, в якій ми зараз знаходимось та лівий і правий кінці відрізків, який вона покриває. Якщо лівий і правий кінці рівні, то ми знаходимось в листі і можем присвоїти йому значення відповідного елемента масиву. Інакше рекурсивно перейдемо до синів даної вершини і перерахуєм її значення, спираючись на значення синів.

Блок-схема алгоритму побудови дерева відрізків наведена на рис. 2.1.

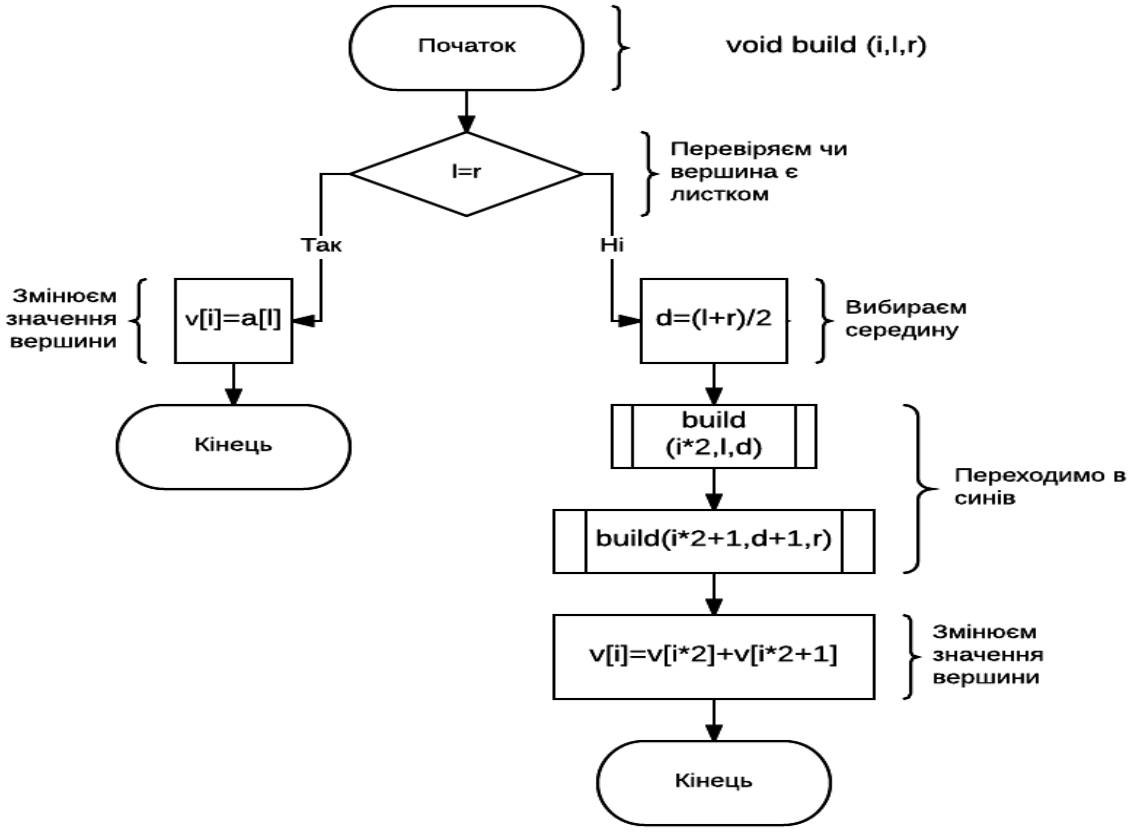

Рис. 2.1. Блок-схема алгоритму побудови дерева відрізків

Код реалізації побудови дерева відрізків на C++

```
void build (int i, int l, int r)
{
	if (l==r)
	{
		v[i]=a[l];
	} else
	{
		int d=(l+r)/2;
		build (i*2,l,d);
		build(i*2+1,d+1,r);
		v[i]=v[i*2]+v[i*2+1];
	}
}
```

Значення змінних:

- *i* – номер вершини, в якій ми зараз знаходимось;
- ***l*** – лівий кінець відрізка, який покриває наша вершина;

- *r* – правий кінець відрізка, який покриває наша вершина;
- *v[]* – масив значень вершин дерева;
- *d* – середина відрізка.

Починати запуск цієї процедури варто з кореня ***(****build(1,1,n)****)***.

2.3. Реалізація запиту зміни елемента

Щоб легко змінити елемент масиву достатньо рекурсивно пройтись від кореня дерева до листа, який відповідає цьому елементу, а на зворотному шляху перерахувати значення у всіх вершинах на цьому шляху. Якщо наша вершина не є потрібним листом, але покриває його, то однин із її синів обов'язково повинен покривати цей лист, а інший ні. Ми будемо завжди заходити лише в потрібного з них, таким чином постійно зменшуючи теперішній відрізок вдвічі.

У процедуру будемо передавати номер теперішнього елемента, кінці відрізка, номер елемента, який потрібно змінити та значення на яке його потрібно змінити.

Блок-схема алгоритму зміни елементу наведено на рис. 2.3.

Код реалізації зміни одного елемента у дереві відрізків на C++

```
void change(int i, int l, int r, int x, int y) {
	if (l==r) {
		v[i]=y;
	} else {
		int d=(l+r)/2;
		if (x<=d) change(i*2,l,d,x,y);
		else change(i*2+1,d+1,r,x,y);
		v[i]=v[i*2]+v[i*2+1];
	}
}
```

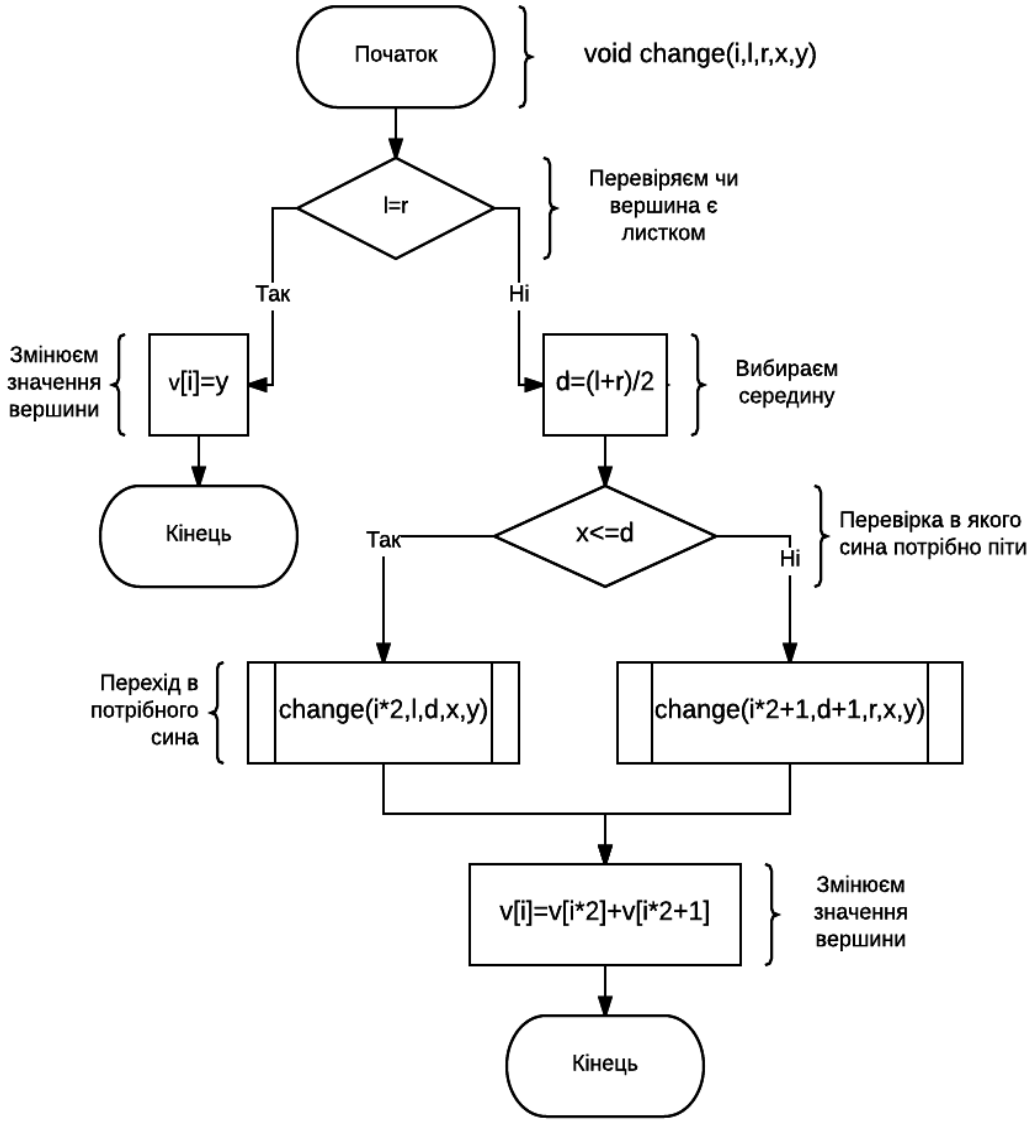

Рис. 2.3. Блок-схема алгоритму зміни елементу

Значення змінних:

- *i* – номер вершини, в якій ми зараз знаходимось;
- *l* – лівий кінець відрізка, який покриває наша вершина;
- *r* – правий кінець відрізка, який покриває наша вершина;
- *v []* – масив значень вершин дерева;
- *d* – середина відрізка;
- *x* – номер елемента масиву, який потрібно змінити;
- *y* – значення, на яке потрібно змінити.

Починати запуск цієї процедури варто з кореня дерева *(change(1,1,n,x,y))*.

2.4. Реалізація запиту на знаходження суми елементів на відрізку

Для цього запиту можна скористатися рекурсивною функцією. Вона буде отримувати номер теперішнього елемента, лівий і правий край відрізка, на якому ми знаходимось та лівий і правий край відрізка, на якому потрібно знайти суму. Якщо ці два відрізки не перетинаються, то ми можемо повернути 0. Якщо відрізок, в якому ми знаходимось повністю належить відрізку, на якому потрібно знайти суму, то ми можемо повернути значення елемента вершини, в якій знаходимось. Інакше рекурсивно запустимо функцію в обох синів і суму відповідей для них повернемо. Таким чином, кожен елемент відрізка масиву ми додаємо рівно один раз.

Блок-схема алгоритму реалізації запиту на знаходження суми елементів на відрізку наведено на рис. 2.4.

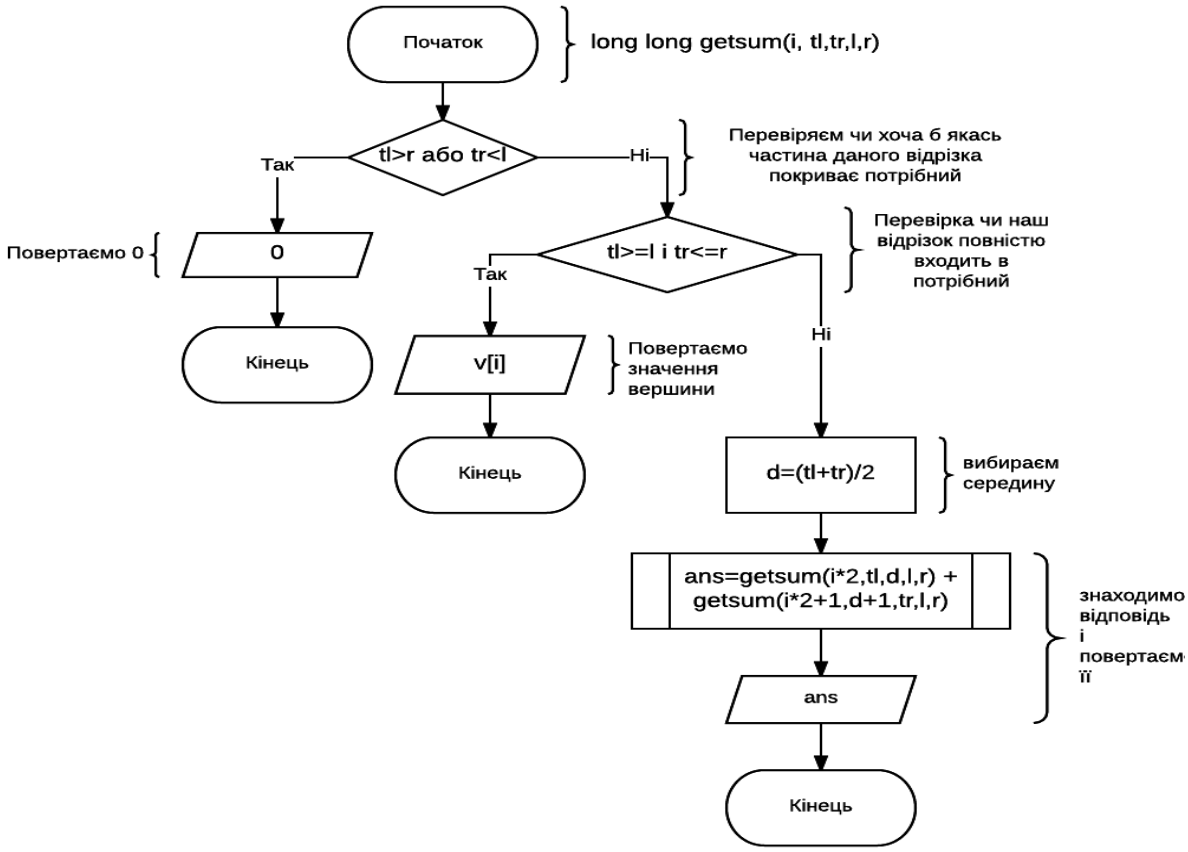

Рис. 2.4. Блок-схема алгоритму реалізації запиту на знаходження суми елементів на відрізку

Код реалізації запиту знаходження суми на відрізку на C++

```
long long getsum(int i, int tl, int tr, int l, int r) {
        if (tl>r || tr<l) {
                return 0;
        } else if (tl>=l && tr<=r) {
                return v[i];
        } else {
                int d=(tl+tr)/2;
                return getsum(i*2,tl,d,l,r) + getsum(i*2+1,d+1,tr,l,r);
        }
}
```

Значення змінних:

- *i* – номер вершини, в якій ми зараз знаходимось;
- *tl* – лівий кінець відрізка, який покриває наша вершина;
- *tr* – правий кінець відрізка, який покриває наша вершина;
- *v[]* – масив значень вершин дерева;
- *d* – середина відрізка;
- *l* – лівий кінець відрізка, суму на якому потрібно знайти.
- *r* – правий кінець відрізка, суму на якому потрібно знайти.

Починати запуск цієї процедури варто з кореня дерева *(getsum(1,1,n,l,r))*.

Для економії часу також можна переходити тільки в правий або лівий відрізок, якщо інший не належить до відрізку запиту. В такому випадку, перевірка на те чи належить наш відрізок запиту буде непотрібною.

У додатку 1 наведено повний код реалізації дерева відрізків для поставленої задачі.

2.5. Оцінка ресурсів часу та пам’яті, використовуваних деревом відрізків

Якщо кількість елементів в початковому масиві буде степенем двійки, то кількість вершин буде $2n$-1 (сума геометричної прогресії 1,2,4…n). Інакше, сума вершин не буде більша за $4n$, адже ми можемо доповнити наш масив нулями до степені двійки і тоді кількість елементів в ньому буде менша за $2n$. Отже, для побудови дерева достатньо масиву з $4n$ елементів. В умові задачі $n \leq 10^5$, отже масиву з $4 \cdot 10^5$ вистачить в будь-якому випадку. Максимальне значення у вершині може бути 10^{15}, це виходить за тип int у C++. Щоб виправити цю проблему потрібно використовувати тип long long. Один елемент цих типів займає 8 байт пам’яті (64 біти). Загалом з усіма допоміжними змінними та масивами програма не займе більше 50 мб.

При побудові дерева ми зайдемо в кожну вершину один раз, а оскільки вершин не більше, ніж $4n$, то складність побудови дерева $O(n)$.

При відповіді на запити ми будемо спускатись по дереву і не пройдемо більше його глибини, яка рівна двійковому логарифму з n. Отже, асимптотична складність обробки одного запиту $O(\log(n))$.

Асимптотичну складність алгоритму в цілому можна оцінити як $O(m \cdot \log(n))$. Мова програмування C++ виконує за секунду приблизно $2 \cdot 10^8$ арифметичних операцій, що набагато більше, ніж кількість операцій, яку зробить наша програма на максимальних тестах.

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

Таблиця 2.1

	n=100	n=1000	n=10000	n=100000
m=100	<1 мс	<1 мс	15 мс	36 мс
m=1000	<1 мс	15 мс	15 мс	39 мс
m=10000	31 мс	46 мс	61 мс	62 мс
m=100000	152 мс	162 мс	180 мс	220 мс

2.6 Узагальнення дерева відрізків для різних типів задач

У нашому конкретному випадку ми розглядали дерево відрізків для операції суми цілих числових значень, але ця структура даних підходить і для більш загальних задач. Дерево відрізків може використовуватися для зберігання довільних типів даних і результатів операцій над ними, для яких виконується умова асоціативності: $x*(y*z) = (x*y)*z$. Зокрема, найпошренішими з таких операцій є:

- сума значень на відрізку(зокрема сума за модулем, xor-сума)
- знаходження кількості значень на відрізку, що задовільняють певний критерій(зводиться до знаходження суми значень на відрізку)
- множення(цілих чисел за модулем, кватерніонів, матриць)
- знаходження найбільшого/найменшого значення в масиві
- сортування даних

Для можливості оновлення даних важливим є існування алгоритму швидкого обрахунку значення обраної операції. Не для всіх асоціативних операцій існує такий алгоритм, зокрема, для операції сортування об'єднання двох відсортованих послідовностей даних матиме складність $O(l)$, де l — це сумарна довжина двох послідовностей. Тому складність оновлення даних в дереві відрізків навіть на однії позиції матиме складність, що дорівнює сумарній довжині відрізків всіх вершин на шляху від заданої вершини до кореня, а саме $O(n)$. Проте це не означає, що для таких операцій немає сенсу використовувати дерево відрізків. Побудова дерева для заданих даних матиме складність $O(n \cdot log(n))$, а ми все одно зможемо виконувати запити на відрізку, інформація про які не зберігається явно, але може бути легко отримана із збережених даних. Зокрема, до таких запитів належить кількість чисел на відрізку, що дорівнюють, менші або більші за задане число. Результат такої операції для окремо взятої вершини легко обраховується бінпошуком, після чого, ми просто знаходимо суму потрібних нам значень за допомогою вже відомих нам алгоритмів.

РОЗДІЛ 3. МАСОВІ ЗМІНИ ЕЛЕМЕНТІВ У ДЕРЕВІ ВІДРІЗКІВ

3.1. Опис виконання запиту масових змін елементів на відрізку

Одним із найпопулярніших вдосконалень дерева є можливість зміни елементів на відрізку. Припустимо, що в нашій задачі з'явився третій тип запитів – замінити всі значення в масиві з *i*-тового елемента по *j*-товий на *x,* де *i, j, x* - натуральні числа ($0 \le x \le 10^9$, $1 \le i \le j \le n$).

Якщо користуватись тими методами, які були запропоновані раніше, то в гіршому випадку доведеться *m* разів змінювати *n* елементів. Очевидно, що це буде занадто довго.

Для вирішення проблеми введемо поняття проштовхування. Замість того, щоб кожного разу спускатись по дереву в листи і змінювати значення в них будемо спускатись до вершин які їх покривають і якщо потрапимо в таку вершину то перерахуємо її значення та запам'ятаємо, що маємо потім змінити її синів. Далі, щоразу коли будемо покидати вершину і переходити до її синів в усіх типах запитів, будемо проштовхувати зміни, які ми запам'ятали. А саме, змінювати значення двох її синів та запам'ятовувати це саме значення для них.

Яким чином можна реалізувати це все? Для цього знадобиться допоміжний масив *push[]*, в якому будуть зберігатися значення, які потрібно проштовхнути в синів для кожної вершини. Якщо проштовхування робити не потрібно, то значення елемента цього масиву буде рівне -1. Також, перед переходом в синів потрібно зробити саме проштовхування. Для цього, можна зробити окрему процедуру *make_push()***,** яка буде приймати номер вершини та відрізок, який вона покриває. Вона буде змінювати значення елемента та проштовхувань його синів. Знаючи, на скільки зміняться елементи відрізка та їх кількість, неважко визначити їхню суму. Запит третього типу реалізуємо за допомогою рекурсивної процедури з такими самими параметрами, як і для запитів другого типу. Єдиною відмінністю буде те, що замість повернення суми ми будемо перераховувати значення та робити запам'ятовування.

З першого погляду здається, що така оптимізація не мала б суттєво вплинути на час виконання програми, адже зміни в дереві потрібно буде зробити. Але, давайте підрахуємо складність алгоритму. Якщо не враховувати константу, то час виконання всіх інших запитів не зміниться, адже операція проштовхування виконується з асимптотичною складністю *O(1)*. Запити нового типу будуть працювати так само як запити на знаходження суми на відрізку, отже їхня складність буде однакова. В результаті цього, складність алгоритму така ж сама, як і без запиту третього типу.

Використана пам’ять теж суттєво не зміниться. Для реалізації знадобиться лише один новий масив з такими ж розмірами, як і масив значень елементів.

Вносячи невеликі зміни, можна також пристосувати програму до можливості віднімати або додавати певне число на відрізку.

3.2.Реалізація запиту зміни елементів на відрізку

Для зміни на відрізку можна використати рекурсивну процедуру, яка буде приймати номер вершини, в якій ми стоїмо, її лівий і правий кінець та лівий і правий кінці відрізка, на якому потрібно зробити зміну. Якщо останні два відрізки не дотикаються, то ми можемо закінчити роботу. Якщо ж наша вершина повністю лежить у потрібному відрізку, то ми можемо перерахувати відповідь для неї і змінити її ***push*** (пообіцяти, що при потребі потім змінимо значення і у її синах). Інакше, ми робимо проштовхування, рекурсивно переходимо до її синів і перераховуємо значення вершини.

Для кращого розуміння роботи алгоритму можна розглянути його на конкретному прикладі. Нехай в нас масив з 8 елементів. Дерево, побудоване на ньому, буде мати такий вигляд, як на рисунку **3.1**.

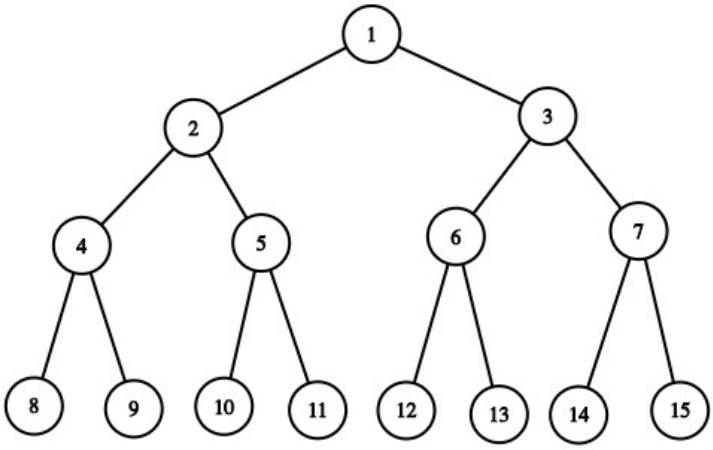

Рис. 3.1. Дерево відрізків, побудоване на масиві з 8-ми елементів.

Тепер нам приходить запит змінити всі елементи від 1 до 4 на 8. Ми перейдемо у вершину 1, яка покриває відрізок 1-8. Далі, у вершину 2 (1-4). Ця вершина повністю покривається потрібним відрізком. Отже, ми змінимо її значення і пообіцяємо при потребі змінити значення у її піддереві. На скільки ж потрібно змінити значення вершини? Очевидно, що після присвоєння всім елементам від ***tl*** до ***tr*** значення ***x*** їхня сума буде ***(tr-tl+1)*x***.

Далі, якщо нам прийде запит знайти суму на відрізку 1-2. Ми спустимось у вершину 2 (1-4), перед тим, як перейти до синів зробимо проштовхування, після якого змінимо значення синів і їхній push.

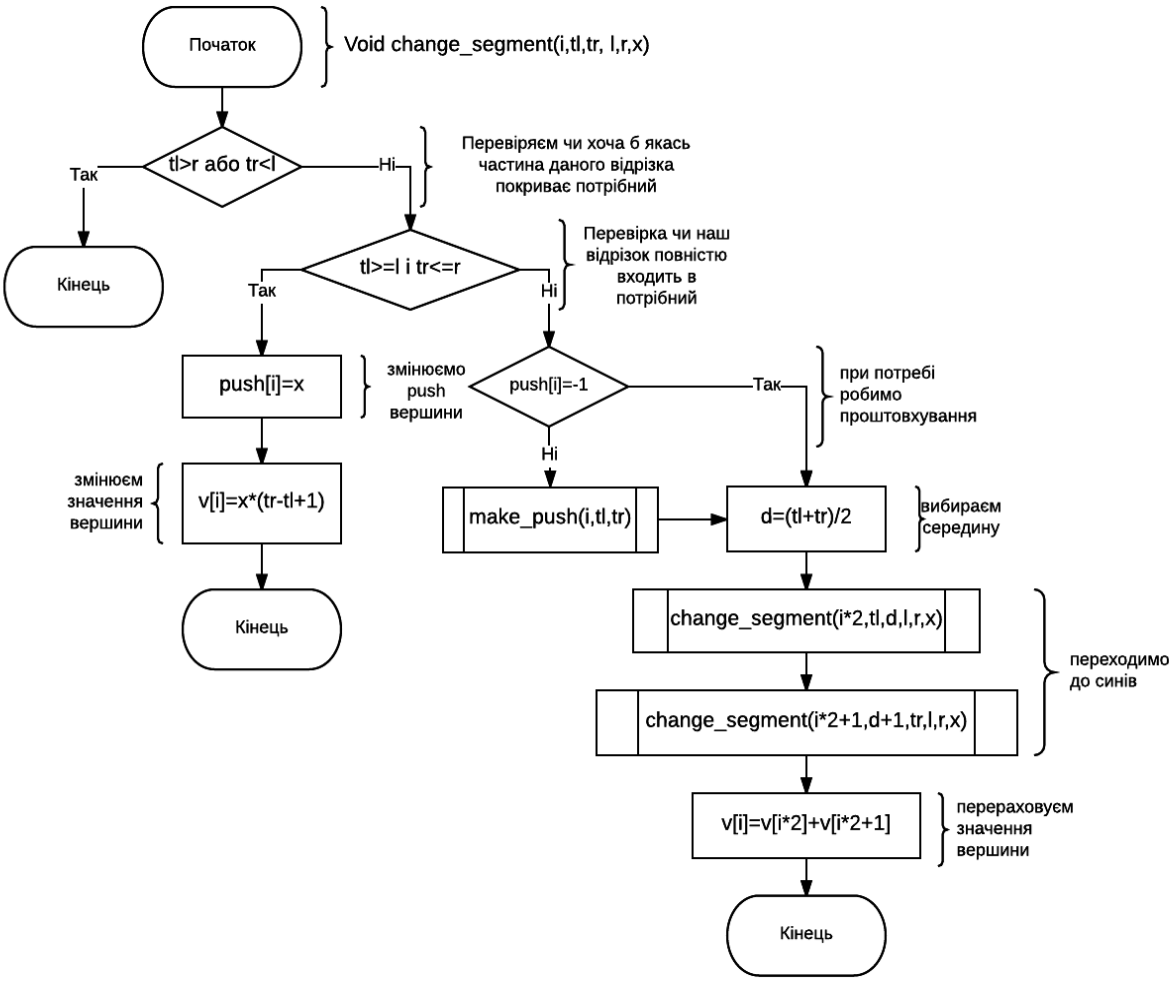

Рис. 3.2. Блок-схема алгоритму зміни елементів на відрізку

Код реалізації операції зміни на відрізку на C++

```
void change_segment(int i, int tl, int tr, int l, int r, long long x) {
        if (tl>r || tr<l) {
                return;
        } else if (tl>=l && tr<=r) {
                push[i]=x;
                v[i]=x*(tr-tl+1);
        } else {
                if(push[i]!=-1) {
                        make_push(i,tl,tr);
                }
                int d=(tl+tr)/2;
```

```
            change_segment(i*2,tl,d,l,r,x);
            change_segment(i*2+1,d+1,tr,l,r,x);
            v[i]=v[i*2]+v[i*2+1];
        }
}
```

Процедура проштовхування має змінити значення синів та їхній *push,* а також присвоїти *push* теперішньої вершини -1.

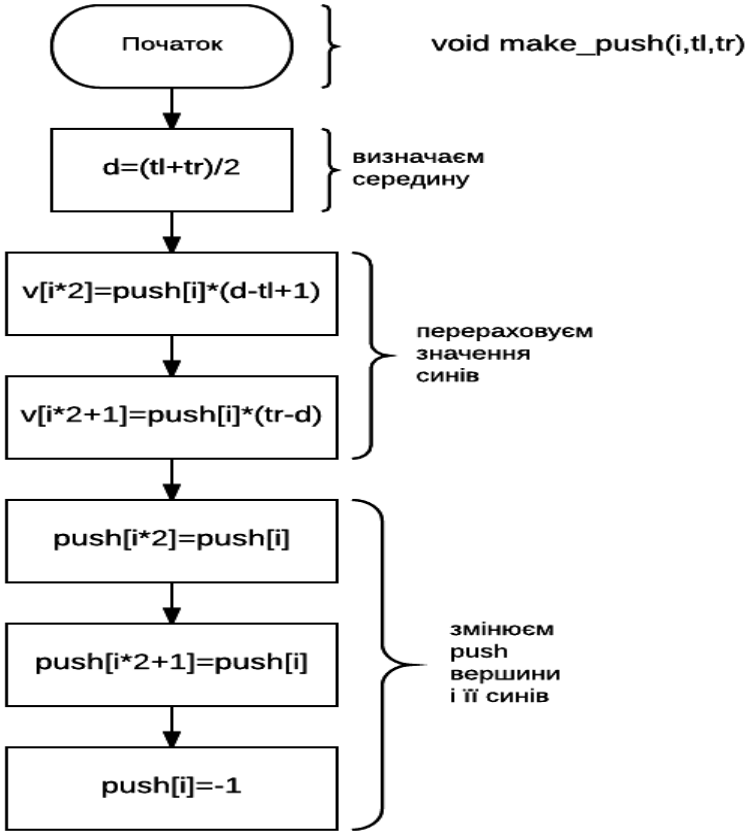

Рис. 3.3. Блок-схема процедури make_push

Код реалізації операції проштовхування на C++

```
void make_push(int i, int tl, int tr)
{
        int d=(tl+tr)/2;
        v[i*2]=push[i]*(d-tl+1);
        v[i*2+1]=push[i]*(tr-d);
        push[i*2]=push[i];
        push[i*2+1]=push[i];
        push[i]=-1;
}
```

Значення змінних:

- *i* – номер вершини, в якій ми зараз знаходимось;
- *tl* – лівий кінець відрізка, який покриває наша вершина;
- *tr* – правий кінець відрізка, який покриває наша вершина;
- *v[]* – масив значень вершин дерева;
- *d* – середина відрізка;
- *l* – лівий кінець відрізка, суму на якому потрібно знайти;
- *r* – правий кінець відрізка, суму на якому потрібно знайти;
- *push[]* – масив, який відповідає за збереження інформації про потребу зміни піддерева даної вершини.

Початкові значення елементів масиву *push[]* мають дорівнювати -1.

До всіх інших функцій і процедур потрібно додати проштовхування перед переходом до синів вершини.

У додатку 3 наведено повний код реалізації програми із запитом зміни на відрізку.

РОЗДІЛ 4. МОДИФІКАЦІЇ КЛАСИЧНОГО ДЕРЕВА ВІДРІЗКІВ

4.1. Неявне дерево відрізків та його застосування

В деяких задачах виникає ситуація, коли доводиться працювати з даними, об’єм яких перевищує допустимі об’єми оперативної пам’яті сучасних комп’ютерів. Класична реалізація дерева відрізків, описана у попередніх розділах, вимагає обсяги пам’яті, лінійні відносно кількості елементів у масиві. В той же час, якщо обсяги даних не дозволяють зберегти їх в оперативній пам’яті, а інформацію про ці дані ми отримуємо під час роботи з ними, то структуру дерева відрізків можна оптимізувати.

Основна ідея неявного дерева відрізків полягає в тому, щоб на самому початку не будувати бінарне дерево, а будувати його під час отримання нових даних. Тобто структура такого дерева буде нагадувати бінарне дерево, в якого може бути відсутній лівий або правий син. Тоді під час оновлення даних в такому дереві ми будемо або переходити в наявні вершини, або створювати нові, якщо дотепер ми не переходили у цей вузол дерева.

З алгоритму випливає, що під час кожного такого запиту ми створимо не більше ніж $O(\log(R))$ вершин, де R — довжина діапазону індексів, з яким ми працюємо. При обрахуванні результату запиту на відрізку алгоритм майже не відрізнятиметься від класичної версії: тепер замість того, щоб переходити в ліву і праву вершину одночасно, ми будемо переходити тільки в ті вершини, які на даний момент створені. Це дає нам складність часу роботи $O(\log(R))$ для обох запитів, і складність пам’яті $O(M \cdot \log(R))$, де M — загальна кількість запитів оновлення даних.

4.2. Розширення дерева відрізків до персистентної структури даних

Тепер припустимо, що перед нами виникає потреба не тільки виконувати стандартні запити оновлення інформації та обчислення значення на відрізку, а й зберігати проміжні копії дерева відрізків з ціллю “відкотитися” до однієї з попередніх версій чи використати її для обрахунку значення цільової функції. Очевидно, зберігання повної копії на кожному кроці є неефективним по пам’яті, а відтворення “з нуля” за допомогою списку запитів — за часом.

Ключ до ефективного алгоримту, як і в попередній задачі, полягає в модифікації алгоритму оновлення значення на певній позиції. Давайте розглянемо, які саме вершини будуть змінюватися при виконанні запиту оновлення. З алгоритму випливає, що це будуть вершини на шляху від кореня до листка. Ідея модифікації полягає в тому, щоб не змінювати значення в цих вершинах напряму, а створити копії тих вершин, значення яких змінилося, і розставити в них посилання на вершини за таким принципом: якщо в сина є оновлена копія, то наша вершина посилається на неї, якщо ні — то на версію цієї вершини зі старого дерева відрізків.

Структура даних, побудована за таким принципом, є цілісною і дозволяє зберігати обидві версії дерева відрізків, використовуючи $O(log(n))$ пам’яті. Для доступу до певної версії дерева відрізків достатньо зберігати її корінь, тож дана структура має незмінну складність часу роботи, і складність пам’яті $O(m \cdot log(n))$, де m — загальна кількість запитів оновлення даних.

Варто підмітити, що ця модифікація також не є надто чутливою до зміни діапазону, в якому зберігаються дані, тому, аналогічно до попередньої модифікації, вона може бути використана для роботи з великими діапазонами індексів.

РОЗДІЛ 5. ВИКОРИСТАННЯ СТРУКТУРИ ДЕРЕВА ВІДРІЗКІВ В ПОЄДНАННІ ІЗ ІНШИМИ АЛГОРИТМАМИ

Дана структура часто використовується для пришвидшення роботи інших алгоритмів. Одним з них є алгоритм пошуку довжини найбільшої підпослідовності, що зростає. В оригінальному вигляді він має асимптотичну складність $O(n^2)$, де n – довжина масиву. Якщо використати дерево, то складність зміниться на $O(n \cdot log(n))$.

Нехай, у нас є масив a з n елементів ($1 \le n \le 10^5$, $1 \le a[i] \le 10^5$). Потрібно знайти таку послідовність елементів масиву, щоб виконувались дві умови: $a[i] < a[j]$ та $i < j$, і вивести максимальну довжину такої послідовності.

Розв'язок за допомогою динамічного програмування

Ми будемо перераховувати динаміку *d[]*, в якій *d[i]* буде рівне максимальній довжині зростаючої послідовності, що закінчується і-товим елементом масиву. Ми будемо перераховувати її від *d[0]* до *d[n]*. Отже, коли ми дійдемо до *d[i]*, то нам вже будуть відомі *d[i-1], d[i-2], ..., d[0]*. Якщо цей елемент є найменшим на префіксі, то *d[i]* буде рівне 1, інакше ми можемо перебрати передостаній елемент нашої послідовності і вибрати найдовшу довжину, яку можемо взяти. Вибраний передостанній елемент має бути менший за теперішній.

$$d[i] = \max\left(1, \max_{\substack{j=0\dots i-1,\\ a[j]<a[i]}} (d[j]+1)\right). \tag{4.1}$$

Розв'язок з використанням дерева відрізків за O(n·log(max(a[i])))

Щоб кожного разу не перебирати всі попередні елементи масиву як передостанні, їхні значення можна зберігати в дереві відрізків, в якому значення в листку, який покриває відрізок *i-i* буде рівне максимальній довжині зростаючої послідовності, останній елемент якої дорівнює *i*.

В такому випадку ми зможемо перераховувати *d[i]* за *O(log(max(a[i])))* і воно буде рівне *get_max (1,1,n,1,a[i]-1)* + 1.

В додатку 5 наведено код реалізації розв’язку задачі. Слід зауважити, що існує розв’язок за допомогою бінарного пошуку, який має таку саму асимптотичну складність, але меншу константу, за рахунок чого на практиці працює приблизно в 4 рази швидше.

РОЗДІЛ 6. ПРИКЛАДИ ВИКОРИСТАННЯ ДЕРЕВА ВІДРІЗКІВ ПРИ РОЗВ'ЯЗУВАННІ ОЛІМПІАДНИХ ЗАДАЧ

Структуру дерева відрізків часто використовують при розв'язуванні олімпіадних задач через свою гнучкість. Розглянемо задачу із XXIV Всеукраїнської учнівської олімпіади з програмування за 2011 рік, авторство, якої належить Іллі Порубльову [12].

Турист

Турист подорожує пішки уздовж координатної вісі O_x. Йти можна в будь-якому з двох можливих напрямків та з будь-якою швидкістю, що не перевищує V, в тому числі знаходитись на місці. З газетних анонсів він знає, що у момент t_1 у точці з координатою x_1 відбудеться одна цікава подія, у момент t_2 у точці з координатою x_2 - ще одна, і т. д., до (x_N, t_N). Цікаві події достатньо короткотривалі, їх можна вважати миттєвими. Вважається, що турист відвідав подію i, якщо у момент t_i він знаходився у точці з координатою x_i.

Напишіть програму, яка знайде максимальну кількість подій, які зможе відвідати турист, для таких двох припущень:

- спочатку руху (у момент часу 0) турист знаходиться у точці 0;
- турист може обрати початкову точку, з якої він вирушить.

Вхідні дані

Перший рядок містить єдине натуральне число **N** ($1 \le N \le 100\ 000$) - кількість цікавих подій. Наступні **N** рядків містять по два цілих числа x_i та t_i - координату та момент часу події з номером і. Останній **(N+2)-**ий рядок файлу містить єдине ціле число **V** - значення максимальної швидкості руху туриста. Усі значення x_i належать діапазону $-10^8 \le x_i \le 10^8$, усі значення t_i належать діапазону $1 \le t_i \le 10^6$, значення **V** належить діапазону $1 \le V \le 1000$. У вхідних даних можливі різні події, що мають однакову координату **x** або однаковий час **t**, але неможливі різні події, що мають однакові **x** та **t** одночасно.

Вихідні дані

Єдиний рядок має містити два цілих числа - максимально можливу кількість подій, які турист може відвідати, якщо він розпочне рух у момент 0 з точки 0, потім максимально можливу кількість подій, які турист може відвідати, самостійно обравши точку старту.

Вхідні дані	Вихідні дані
3 –1 1 42 7 40 8 2	1 2

Неефективні способи розв'язку

Ідейно найпростішим розв'язком є повний перебір порядку, в якому турист буде відвідувати всі події. При n>30 даний спосіб буде працювати більше години. Розглядати його детальніше, не має сенсу, через неефективність. Перед розглядом наступного алгоритму потрібно довести один факт. Очевидно, що турист завжди може рухатись з максимальною швидкістю і вільний час чекати до початки події на місці, в якому вона відбудеться. Ми можемо відсортувати всі події за часом. Далі для кожної події рахувати елементарну динаміку, перебираючи всі події, що відбулися раніше. Якщо в подію з номером і ми могли перейти із події з номером j, то $dp_i=max(dp_i, dp_j + 1)$. Складність даного алгоритму квадратична. Хоч на максимальних тестах програма працює задовго, але вже набирає більше половини балів.

Ефективний розв'язок

Розглянемо ситуацію, коли ми хочемо перейти із події i в подію j. Для цього, відстань між подіями має бути менша, ніж час між ними, помножений на швидкість людини. Розіб'ємо цю умову на два випадки (коли наша подія знаходиться далі, ніж подія куди ми йдемо і навпаки).

Отримуємо дві нерівності *x[i]-x[j]>=v·(t[i]-t[j])* та *x[j]-x[i]>=v·(t[i]-t[j])*. Зведемо ці нерівності до системи і проведемо нескладні математичні перетворення. В результаті отримаємо

$$\begin{cases} X[i] - v*t[i] \geq x[j] - v*t[j] \\ X[i] + v*t[i] \geq x[j] + v*t[j] \end{cases} \quad \textbf{(5.1)}$$

Для кожної вершини порахуємо дві формули, які будуть дорівнювати *f1[i]=x[i]-v[i]·t[i]* і *f2[i]=x[i]+v[i]·t[i]*. Щоб встигнути перейти з однієї події на іншу, пораховані значення для початкової події мають бути не менші, ніж пораховані значення для другої. Можна відсортувати події за першим значенням, а при рівності перших значень – за другими. Наразі, задача зводиться до пошуку найбільшої неспадної підпослідовності серед значень другого масиву. Розв’язок цієї задачі вже був розглянутий у попередніх розділах. Щоб розібрати випадок, коли початкова координата рівна нулю, потрібно відкинути всі події, пораховані *f1* та *f2* яких від’ємні.

Задача чудово відображає, що для використання дерева відрізків в умові не обов’язково повинні бути введені запити на знаходження певних значень на відрізках динамічного масиву.

ВИСНОВКИ

В даній роботі нами розглянуто структуру дерева відрізків, описано алгоритм його роботи, вирахувано асимптотичну складність роботи, досліджено різноманітні можливості структури, вказані переваги та недоліки.

Дерево відрізків – дуже гнучка структура, і число завдань, що вирішуються нею, теоретично необмежені. Крім наведених вище видів операцій з деревами відрізків, також можливі і набагато більш складні операції.

Важливою особливістю дерев відрізків є те, що вони споживають лінійний обсяг пам'яті: стандартному дереву відрізків потрібно близько 4n елементів пам'яті для роботи над масивом розміру n.

Дерево відрізків дозволяє значно пришвидшити операції та запити на відрізку масиву. Воно є чудовою оптимізацією інших алгоритмів. Асимптотична складність побудови дерева *O(n)*, а відповіді на запити здійснюються за *O(log(n))*. Воно є зручним у використанні та нескладним у реалізації. За допомогою простоти ідеї розбиття масиву на відрізки його можна всебічно вдосконалювати та підлаштовувати для розв’язання поставленої задачі.

Ця структура має безліч переваг порівняно з іншими, хоча може програвати їм в окремих характеристиках (наприклад, дерево Фенвіка має меншу константу у часі роботи, а SQRT-декомпозиція використовує менше ресурсів пам’яті).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Акимов О.Е. Дискретная математика: логика, группы, графы. - М.: Лаборатория базовых знаний, 2001. - 376 с.
2. Бондаренко М.Ф. Комп'ютерна дискретна математика. - Харків: "Компанія СМІТ", 2004. - 480 с.
3. Дистанционная подготовка к информатике [Електронний ресурс]. – Режим доступу: http://informatics.mccme.ru
4. Караванова Т.П. Інформатика: методи побудови алгоритмів та їх аналіз. Обчислювальні алгоритми: Навч. посіб. для 9-10 кл. із поглибл. вивч. інформатики – К.: Генеза. – 2008.- 333 с.
5. Кузьменко В. В. Основи дискретної математики. Розділ "Елементи теорії графів": Конспект лекцій. / В.В. Кузьменко, Г. Г. Швачич, Г. І. Рижанкова, В.М. Пасинков. – Дніпропетровськ: НМетАУ, 2004. – 38с.
6. Методичні вказівки до лабораторної роботи "Структура даних бінарне дерево пошуку" для підготовки студентів напрямку 6.0915 "Комп'ютерна інженерія" / Укл. Т.А.Лисак – Львів: Видавництво НУ "Львівська політехніка", 2010 – 16 с.
7. Новиков Ф.А. Дискретная математика для программистов. - С.-Пб.: Питер, 2001. - 304 с.
8. Структури даних та алгоритми. Методичні вказівки до лабораторних занять для студентів спеціальності 125 „Кібербезпека" денної та заочної форми навчання / В.М. Мельник, С.В. Лавренчук. –Луцьк: Луцький НТУ, 2016. – 176 с.
9. Codeforces [Електронний ресурс]. – Режим доступу: http://codeforces.com
10. CSacademy [Електронний ресурс]. – Режим доступу: https://csacademy.com
11. MAXimal [Електронний ресурс]. – Режим доступу: http://e-maxx.ru/algo/segment_tree

12. E-Olymp [Електронний ресурс]. – Режим доступу: https://www.e-olymp.com/uk/problems/2008

Додаток 1

Код реалізації дерева відрізків для операції суми

```
#include <bits/stdc++.h>
using namespace std;
long long v[400001];
int a[100001];

void build (int i, int l, int r)
{
   if (l==r)
   {
      v[i]=a[l];
   } else
   {
      int d=(l+r)/2;
      build (i*2,l,d);
      build(i*2+1,d+1,r);
      v[i]=v[i*2]+v[i*2+1];
   }
}

void change(int i, int l, int r, int x, int y)
{
   if (l==r)
   {
      v[i]=y;
        } else
   {
      int d=(l+r)/2;
      if (x<=d) change(i*2,l,d,x,y); else
      change(i*2+1,d+1,r,x,y);
      v[i]=v[i*2]+v[i*2+1];
   }
}

long long getsum(int i, int tl, int tr, int l, int r)
{
   if (tl>r || tr<l) return(0); else
   if (tl>=l && tr<=r) return(v[i]); else
   {
      int d=(tl+tr)/2;
      return(getsum(i*2,tl,d,l,r) + getsum(i*2+1,d+1,tr,l,r));
   }
```

```
}

int main()
{
    int n,m;
    cin>>n>>m;
    for (int i=1;i<=n;i++)
        cin>>a[i];
    build(1,1,n);
    for (int i=1;i<=m;i++)
    {
        int type;
        cin>>type;
        if (type==1)
        {
            int x,y;
            cin>>x>>y;
            change(1,1,n,x,y);
        } else
        if (type==2)
        {
            int l,r;
            cin>>l>>r;
            cout<<getsum(1,1,n,l,r)<<'\n';
        }
    }
}
```

Додаток 2

Функції дерева відрізків для знаходження максимуму на відрізку

```
void build (int i, int l, int r)
{
   if (l==r)
   {
      v[i]=a[l];
   } else
   {
      int d=(l+r)/2;
      build (i*2,l,d);
      build(i*2+1,d+1,r);
      v[i]=max(v[i*2],v[i*2+1]);
   }
}

void change(int i, int l, int r, int x, int y)
{
   if (l==r)
   {
      v[i]=y;
        } else
   {
      int d=(l+r)/2;
      if (x<=d) change(i*2,l,d,x,y); else
      change(i*2+1,d+1,r,x,y);
      v[i]=max(v[i*2],v[i*2+1]);
   }
}

int getmax(int i, int tl, int tr, int l, int r)
{
   if (tl>r || tr<l) return(0); else
   if (tl>=l && tr<=r) return(v[i]); else
   {
      int d=(tl+tr)/2;
      return(max(getsmax(i*2,tl,d,l,r) , getmax(i*2+1,d+1,tr,l,r)));
   }
}
```

Додаток 3

Код реалізації програми із зміною на відрізках

```
#include <bits/stdc++.h>
using namespace std;
long long v[400001];
int a[100001];
int push[400001];
void make_push(int i, int tl, int tr)
{
   int d=(tl+tr)/2;
   v[i*2]=push[i]*(d-tl+1);
   v[i*2+1]=push[i]*(tr-d);
   push[i*2]=push[i];
   push[i*2+1]=push[i];
   push[i]=-1;
}
void build (int i, int l, int r)
{
   push[i]=-1;
   if (l==r)
   {
      v[i]=a[l];
   } else
   {
      int d=(l+r)/2;
      build (i*2,l,d);
      build(i*2+1,d+1,r);
      v[i]=v[i*2]+v[i*2+1];
   }
}

void change(int i, int l, int r, int x, int y)
{
   if (l==r)
   {
      v[i]=y;
        } else
   {
      if (push[i]!=-1)
         make_push(i,l,r);
      int d=(l+r)/2;
      if (x<=d) change(i*2,l,d,x,y); else
      change(i*2+1,d+1,r,x,y);
```

```
        v[i]=v[i*2]+v[i*2+1];
    }
}

void change_segment(int i, int tl, int tr, int l, int r, int x)
{
    if (tl>r || tr<l) return; else
    if (tl>=l && tr<=r)
    {
        push[i]=x;
        v[i]=x*(tr-tl+1);
}else
    {
                if(push[i]!=-1)
        make_push(i,tl,tr);
        int d=(tl+tr)/2;
        change_segment(i*2,tl,d,l,r,x);
        change_segment(i*2+1,d+1,tr,l,r,x);
        v[i]=v[i*2]+v[i*2+1];
    }
}

long long getsum(int i, int tl, int tr, int l, int r)
{
    if (tl>r || tr<l) return(0); else
    if (tl>=l && tr<=r) return(v[i]); else
    {
        if (push[i]!=-1)
        make_push(i,tl,tr);
        int d=(tl+tr)/2;
        return(getsum(i*2,tl,d,l,r) + getsum(i*2+1,d+1,tr,l,r));
    }
}

int main()
{
    int n,m;
    cin>>n>>m;
    for (int i=1;i<=n;i++)
        cin>>a[i];
    build(1,1,n);
    for (int i=1;i<=m;i++)
    {
        int type;
```

```
        cin>>type;
        if (type==1)
        {
            int x,y;
            cin>>x>>y;
            change(1,1,n,x,y);
        } else
        if (type==2)
        {
            int l,r;
            cin>>l>>r;
            cout<<getsum(1,1,n,l,r)<<'\n';
        } else
        if (type==3)
        {
            int l,r,x;
            cin>>l>>r>>x;
            change_segment(1,1,n,l,r,x);
        }
    }
}
```

Додаток 4

Код реалізації задачі пошуку найбільшої підпослідовності, що зростає

```
#include <bits/stdc++.h>
using namespace std;
long long v[400001];
int dp[100001],a[100001];

void change(int i, int l, int r, int x, int y)
{
   if (l==r)
   {
      v[i]=y;
        } else
   {
      int d=(l+r)/2;
      if (x<=d) change(i*2,l,d,x,y); else
      change(i*2+1,d+1,r,x,y);
      v[i]=max(v[i*2],v[i*2+1]);
   }
}

int getmax(int i, int tl, int tr, int l, int r)
{
   if (tl>r || tr<l) return(0); else
   if (tl>=l && tr<=r) return(v[i]); else
   {
      int d=(tl+tr)/2;
      return(max(getmax(i*2,tl,d,l,r) , getmax(i*2+1,d+1,tr,l,r)));
   }
}

int main()
{
   int n;
   cin>>n;
   for (int i=1;i<=n;i++)
      cin>>a[i];
   for (int i=1;i<=n;i++)
   {
      dp[i]=getmax(1,0,100000,0,a[i]-1)+1;
      if (getmax(1,0,100000,a[i],a[i])<dp[i]) change(1,0,n,a[i],dp[i]);
   }
}
```