Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Застосування фрактальних кривих у задачі індексації двовимірних об'єктів
(тема)

Виконав:
студент 2 курсу, групи СШІм-21-2
Панасюк О.В.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник проф. Аврунін О.Г.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ В.О. Філатов
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____» __________ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Панасюку Олександру Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Застосування фрактальних кривих у задачі індексації двовимірних об'єктів

затверджена наказом університету від 31 березня 20 23 р. № 306Ст

2. Термін подання студентом роботи до екзаменаційної комісії 15 травня 20 23 р.

3. Вихідні дані до роботи Науково-технічні публікації, дані Інтернет-джерел та відомих наукових проектів щодо розробки існуючих методів моделювання мереж та оцінювання центральних вузлів

4. Перелік питань, що потрібно опрацювати в роботі

1) Фрактали і фрактальна розмірність

2) Структури даних для індексації лінійних та просторових об'єктів

3) Застосування фрактальної кривої Гілберта в R-деревах

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)________________

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на дипломну роботу	03.04.2023	виконано
2	Аналіз предметної області	10.04.2023	виконано
3	Постановка завдання та узгодження з керівником	11.04.2023	виконано
4	Дослідження моделей складних мереж	17.04.2023	виконано
5	Дослідження показників центральності мереж	20.04.2023	виконано
6	Програмна реалізація обчислення центральності вузлів	22.04.2023	виконано
	мереж та експериментальні дослідження		
7	Написання пояснювальної записки	28.04.2023	виконано
8	Попередній захист	15.05.2023	
9	Захист перед ЕК	17.05.2023	

Дата видачі завдання _3_ _квітня_ 20_23_ р.

Студент ________________________________
(підпис)

Керівник роботи ____________________________ _____проф. Аврунін О.Г._______
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 79 с., 21 рис., 2 табл., 2 дод., 12 джерел.

B-ДЕРЕВА, R-ДЕРЕВА, КРИВА ГІЛЬБЕРТА, ПРОСТОРОВІ ДАНІ, СТРУКТУРИ ДАНИХ, ФРАКТАЛИ.

Об’єктом досліджень є структури просторових даних.

Предметом досліджень кваліфікаційної роботи є застосування фрактальних кривих для обходу простору.

Метою роботи є дослідження ефективності побудови да доступу до даних у структурі просторових даних та вплив на показники ефективності способу обходу простору.

ABSTRACT

Explanatory note: 79 p., 21 fig., 2 tabl., 2 ann., 12 sources.

B-TREES, DATA STRUCTURES, FRACTALS, HILBERT CURVE, SPATIAL DATA, R-TREES,.

The object of research is spatial data structures.

The subject of research in the qualification work is the use of fractal curves to traverse space.

The purpose of the work is to study the effectiveness of building and accessing data in the structure of spatial data and the impact on the performance indicators of the method of bypassing 2D space.

ЗМІСТ

Вступ 7
1 Фрактали і фрактальна розмірність 8
1.1 Поняття фрактала 8
1.2 Фрактальна розмірність 11
1.3 L-системи 18
1.4 Фрактальна розмірність систем, які зростають 22
2 Структури даних для індексації лінійних та просторових об’єктів 27
2.1 Особливості роботи з дисковою пам'яттю 27
2.2 Структура B-дерев 29
2.3 Словникові операції над B-деревами 31
2.4 Модифікації B-дерев 39
2.5 R-дерева 40
3 Застосування фрактальної кривої Гілберта в R-деревах 48
3.1 Вплив порядку сканування 2D простору на ефективність R-дерев 48
3.2 Структура R-дерева Гілберта 52
3.3 Порівняння статичних R-дерев з R-деревами Гілберта 56
Висновки 60
Перелік джерел посилання 61
Додаток А Текст програми 62
Додаток Б Відомість кваліфікаційної роботи магістра 79

ВСТУП

Фрактальність чи самоподібність об'єкта, процесу, явища означає, що частини цього об'єкта / процесу / явища подібні до цілого (у певному сенсі). У цьому випадку вивчивши властивості доступної частини об'єкта, можна узагальнити їх на весь цей об'єкт. Багато об'єктів / процеси / явища в природі та техніці мають фрактальні властивості, наприклад, крони дерев, узбережжя, хмари, кровоносна система і т.д. З іншого боку, фрактальні алгоритми широко використовуються для стиснення зображень, або комп'ютерної графіки (малювання гір, хмар та інших ландшафтів).

Основною числовою характеристикою фракталів є фрактальна розмірність. Крім того, відомо, що фрактали характеризуються ступеневим розподілом (розподілом Парето) своїх структурних частин. Показник цього розподілу також характеризує фрактальні властивості об'єкта.

Одним з багатьох практичних застосувань фракталів є використання кривих, що заповнюють площину, для визначення порядку обходу площини. Така потреба виникає у зв’язку з індексацією двовимірних об’єктів на просторовій мапі. Відомо, що порядок обходу області суттєво впливає на ефективність індексування. Одними з методів підвищення цієї ефективності є обхід площини за фрактальною кривою (кривою Пеано, кривою Гілберта, Z-кривою).

У роботі розглядаються і вирішуються такі задачі:

– уявлення поняття фракталу, фрактальної розмірності, аналіз методів оцінювання фрактальної розмірності традиційних просторових (геометричних) фракталів;

– дослідження та аналіз методів індексування та «упаковки» просторових даних, зокрема, R-дерев;

– дослідження впливу методу обходу площини, зокрема за кривою Гільберта, на ефективність індексування;

– аналіз отриманих результатів.

1 ФРАКТАЛИ І ФРАКТАЛЬНА РОЗМІРНІСТЬ

1.1 Поняття фрактала

Поява фракталів (які на той час ще не отримали цього імені) в математичній літературі наприкінці XIX століття було зустрінуто з сумною ворожістю, як це бувало і в історії розвитку багатьох інших математичних ідей. Відомий тогочасний математик, Шарль Ерміт, навіть охрестив їх монстрами. Принаймні, загальна думка визнала їх патологією, що цікавить лише дослідників, які зловживають математичними чудацтвами, а не для справжніх вчених.

В результаті зусиль Бенуа Мандельброта таке ставлення змінилося, і фрактальна геометрія стала шанованою прикладною наукою. Мандельброт ввів у вжиток термін фрактал [1], ґрунтуючись на теорії фрактальної (дрібної) розмірності Хаусдорфа, запропонованої в 1919 році. За багато років до появи його першої книги з фрактальної геометрії, Мандельброт приступив до дослідження появи монстрів та інших патологій у природі. Він знайшов нішу для малих погану репутацію множин Кантора, кривих Пеано, функцій Вейерштрасса та його численних різновидів, які вважалися нонсенсом. Він та його учні відкрили багато нових фракталів, наприклад, фрактальний броунівський рух для моделювання лісового та гірського ландшафтів, флуктуації рівня річок та биття серця. З появою його книг застосування фрактальної геометрії почали з'являтися як гриби після дощу. Це торкнулося як багатьох прикладних наук, і чистої математики. На сьогоднішній день фрактали активно використовуються не тільки для аналізу даних, але й для стиснення зображень, комп'ютерної графіки (малювання гірських ландшафтів, хмар, берегових ліній тощо) у кіноіндустрії та комп'ютерних іграх.

Французький математик Анрі Пуанкаре ініціював дослідження в галузі нелінійної динаміки близько 1890, що призвело до появи сучасної

теорії хаосу. Інтерес до предмета помітно збільшився, коли Едвард Лоренц, який займався нелінійним моделюванням погоди, 1963 року виявив неможливість довгострокових прогнозів погоди. Лоренц зауважив, що навіть мізерні помилки при вимірі параметрів поточного стану погодних умов можуть призвести до абсолютно неправильних передбачень про стан погоди у майбутньому. Ця істотна залежність від початкових умов є основою математичної теорії хаосу.

Траєкторії частинок броунівського руху, яким займалися Роберт Броун ще в 1828 і Альберт Ейнштейн в 1905, являють собою приклад фрактальних кривих, хоча їх математичний опис був дано тільки в 1923 Норбертом Вінер.

У 1890 році Пеано сконструював свою знамениту криву – безперервне відображення, що переводить відрізок у квадрат і, отже, підвищує його розмірність з одиниці до двійки. Кордон сніжинки Коха (1904 рік), чия розмірність $d \approx 1.2618$ – це ще одна добре відома крива, що підвищує розмірність. Фрактал, аж ніяк не схожий на криву, який Мандельброт назвав пилом – це класична множина Кантора (1875 або раніше). Ця множина настільки розріджена, що вона не містить інтервалів, але, проте, має стільки ж точок, скільки інтервал. Мандельброт використовував такий «пил» для моделювання стаціонарного шуму в телефонії. Фрактальна пилюка того чи іншого роду з'являється в численних ситуаціях. Фактично, вона є універсальним фракталом у тому сенсі, що будь-який фрактал – атрактор системи ітерованих функцій є або фрактальним пилом, або його проекцією на простір з більш низькою розмірністю.

Різні деревоподібні фрактали застосовувалися не тільки для моделювання дерев-рослин, але і бронхіального дерева (повітряні гілки в легенях), роботи нирок, кровоносної системи до ін. Цікаво відзначити припущення Леонардо да Вінчі про те, що всі гілки дерева на даній висоті складені разом , рівні за товщиною стовбура (нижче за їх рівень). Звідси випливає фрактальна модель для крони дерева у вигляді поверхні-фрактала.

Багато чудових властивостей фракталів і хаосу відкриваються щодо ітерованих відображень. При цьому починають із деякої функції $y = f(x)$ та розглядають поведінку послідовності $f(x), f\big(f(x)\big), f(f(f(x))), \ldots$ У комплексній площині роботи такого роду сходять, мабуть, до робот Артура Келі, який досліджував метод Ньютона знаходження кореня в додатку до комплексних, а не тільки речових, функцій (1879). Чудового прогресу у вивченні комплексних ітерованих відображень домоглися Гастон Жюліа і П'єр Фату (1919). Звичайно, все було зроблено без допомоги комп'ютерної графіки. У наші дні, багато хто вже бачив барвисті постери із зображенням множин Жюліа і множини Мандельброта, тісно з ними пов'язаного. Освоєння математичної теорії хаосу природно розпочати саме з ітерованих відображень. Вивчення фракталів та хаосу відкриває чудові можливості, як у дослідженні нескінченного числа додатків, так і в галузі чистої математики. Але в той же час, як це часто трапляється, відкриття спираються на піонерські роботи великих математиків минулого. Сер Ісаак Ньютон розумів це, кажучи: «Якщо я й бачив далі за інших, то тільки тому, що стояв на плечах гігантів».

Мандельброт [3] запропонував таке визначення фракталу:

Фракталом називається множина, метрична розмірність (наприклад, розмірність Хаусдорфа-Безиковича) якої відрізняється від її топологічної розмірності.

Це визначення своєю чергою вимагає визначень термінів множина, розмірність Хаусдорфа–Безиковича (d) і топологічна розмірність (d_T яка завжди дорівнює цілому).

Відповідно до іншого визначення фрактал – це об'єкт, який має дробову (нецілочисленну) розмірність.

В даний час ключовою, можна сказати визначальною властивістю фракталів прийнято вважати масштабну інваріантність, або самоподібність.

Фракталом називається структура, що складається з частин, які в якомусь сенсі подібні до цілого.

Так, якщо дано графік фрактального процесу, але позначення на осях не вказані, неможливо визначити масштаб. Історія вивчення фракталів Бенуа Мандельбротом почалася саме з того, що він одного разу переплутав добовий графік біржових котирувань цін на бавовну з погодинним лагом та річний графік цих же цін із помісячним лагом.

Наприклад, контури гірського ландшафту при погляді здалеку складаються з величезних піків. Наблизившись до них, можна розглянути піки (зубці) поменше і так далі аж до найменшого масштабу, який ще можна розрізнити. Насправді, маючи лише зовнішній вигляд обрисів гряди і не використовуючи ніякої додаткової інформації, масштаб зображення оцінити неможливо.

Геометричні фрактали є безліччю точок, вкладеними в «природний» евклідовий простір.

1.2 Фрактальна розмірність

Суворе визначення топологічної розмірності множини досить громіздко [4], але нестрого, на інтуїтивному рівні, топологічна розмірність дорівнює кількості координат, які необхідні для позиціонування елементів множини (точок) усередині нього. Наприклад, для позиціонування точки лінії достатньо однієї координати – відстані від початку відліку, тобто довжини фрагменту лінії. Відповідно, топологічна розмірність лінії дорівнює 1. Можна відзначити, що орієнтація лінії у просторі неважлива: лінія може бути як на площині, так і в просторі, або, наприклад, на сфері. Плоска фігура, чи поверхня, мають $d_T = 2$, і справді, положення будь-якої її точки однозначно визначається двома координатами. Аналогічно, об'ємні фігури мають $d_T = 3$. З іншого боку, точка немає ні довжини, ні ширини, ні висоти тому її топологічна розмірність дорівнює нулю.

Поняття метричної розмірності множини (як і випливає з назви) пов'язане з вимірюванням цієї множини.

Найпростішим способом такого вимірювання є покриття простору, що містить цю множину, безперервною послідовністю шаблонів і підрахунок мінімальної кількості необхідних шаблонів. На цьому принципі засновано визначення розмірності Мінковського і її підрахунку, так званий box-counting метод.

Виберемо як шаблон множину з топологічною розмірністю, не меншою, ніж вимірювана множина. Наприклад, для виміру довжини кривої можна вибрати відрізки, квадратики, куби (рисунок 1.1) і т.д. Форма шаблону не має значення. З точки зору «чистої математики» найбільш правильним шаблоном є d-мірні кулі (відрізки, кола, кулі і т.д.), а з точки зору продуктивності комп'ютерних додатків зручніше використовувати d-мірні куби (відрізки, квадрати, куби).

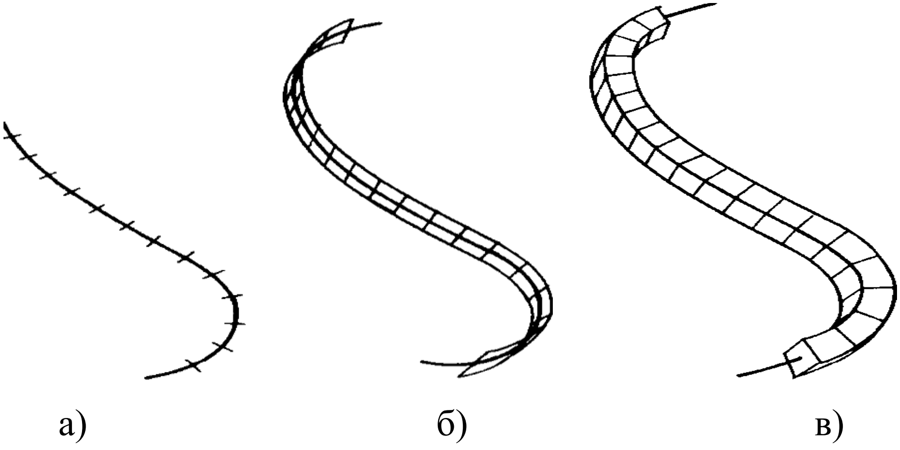


Рисунок 1.1 – Покриття кривої: а) відрізками; б) квадратами; в) кубами

Очевидно, що для звичайної кривої лінійний розмір шаблону (r) та потрібна їх кількість $N(r)$ пов'язані співвідношенням:

$$N(r) \approx L/r. \quad (1.1)$$

Воно буде виконуватися тим точніше, чим менше шаблон.

Аналогічним чином можна виміряти площу, наприклад, кола

покриваючи його квадратами розміру, який зменшується (рисунок 1.2).

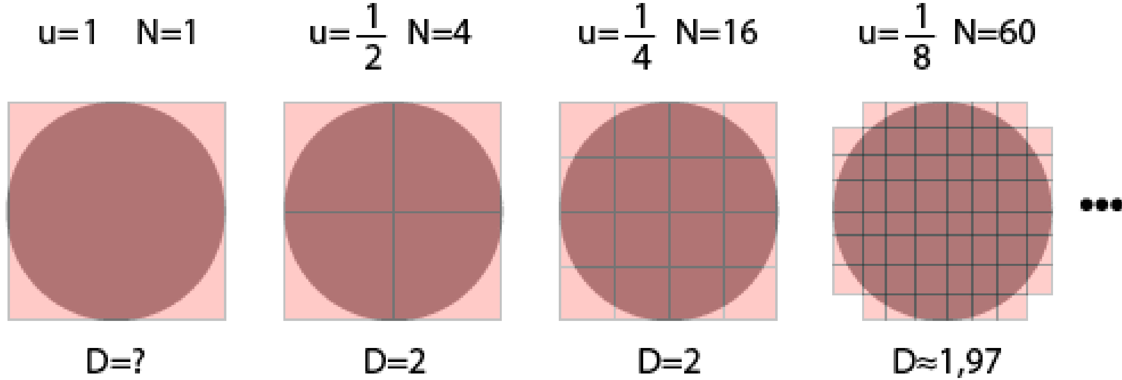


Рисунок 1.2 – Оцінювання метричної розмірності кола

У межі $r \to 0$ отримаємо:

$$N(r) = S / r^2, \qquad S = \lim_{r\to 0}\left(r^2 \cdot N(r)\right). \tag{1.2}$$

Зіставляючи вирази для $N(r)$ (1.1)-(1.2), неважко встановити, що при $r \to 0$ виконується співвідношення:

$$N(r) = const / r^d , \tag{1.3}$$

звідки прямо випливає визначення фрактальної розмірності Мінківського:

$$d = \lim_{r\to 0}\frac{\log(N(r))}{\log(1/r)} = -\lim_{r\to 0}\frac{\log(N(r))}{\log(r)} . \tag{1.4}$$

На практиці оцінка розмірності фігури методом box-counting залежатиме від того, чи вдало чи не дуже шаблони накривають цю фігуру. Так, коло не надто зручно покривається квадратними плитками – багато відходів, відповідно оцінка фрактальної розмірності матиме похибку (рисунок 1.3), причому систематичну: оцінка розмірності

наближається до справжнього значення або строго зверху, або строго знизу – залежно від фігури.

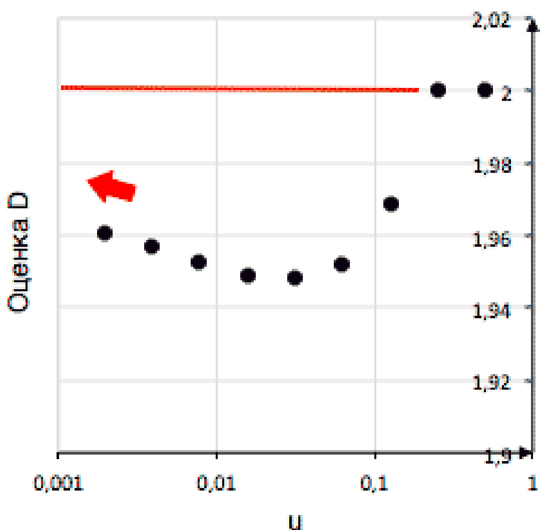


Рисунок 1.3 – Залежність оцінки розмірності кола від масштабу шаблонів

Фрактальна розмірність Хаусдорфа визначається дещо інакше: будемо покривати множину шаблонами з лінійними розмірами w_i, які не перевищують r (тобто $w_i \leq r$). Така «технологія» називається $r-$ покриттям. Альфа-мірою Хаусдорфа називається величина, що дорівнює:

$$F_{\alpha} = \lim_{r \to 0} \inf \sum_{i} w_i^{\alpha} . \tag{1.5}$$

При цьому, як і при box-counting, мається на увазі, що завжди використовується найбільш «економне» покриття серед можливих. В даному випадку використовується нижня межа зазначеної суми, взята за всіма можливими $\mathrm{r}-$покриттями множини (що і позначається символом «inf» перед знаком суми). Надалі опустимо цей оператор для простоти запису.

Площа звичайної плоскої фігури дорівнює $\alpha-$мірі Хаусдорфа (1.5) при $\alpha = 2$. Справді: якщо «заощадити» на діаметрі покриття не вийде (а для

звичайної геометричної фігури це так), тобто якщо $w_i = r$, то з (1.5) та (1.2) випливає:

$$F_2 = \lim_{r\to 0}\sum_i w_i^2 = \lim_{r\to 0}\sum_{i=1}^{N(r)} r^2 = \lim_{r\to 0}(N(r)\cdot r^2) = \lim_{r\to 0}\frac{const}{r^2}\cdot r^2 = S. \quad (1.6)$$

А що буде, якщо підрахувати в такий спосіб довжину, чи об’єм плоскої фігури (тобто $\alpha -$ міру Хаусдорфа для $\alpha \neq 2$)?

$$\begin{aligned} F_1 = L = \lim_{r\to 0}(N(r)\cdot r) &= \lim_{r\to 0}\left(\frac{const}{r^2}\cdot r\right) = \infty, \\ F_3 = V = \lim_{r\to 0}(N(r)\cdot r^3) &= \lim_{r\to 0}\left(\frac{const}{r^2}\cdot r^3\right) = 0. \end{aligned} \quad (1.7)$$

Довжина плоскої фігури виявилася нескінченною, а об'єм – нульовим. Так і має бути згідно з інтуїтивними уявленнями. Справді: єдиною змістовною мірою множини точок, що утворюють плоску фігуру (або поверхню у просторі), є площа, і, відповідно, розмірність таких фігур дорівнює двом.

Узагальнюючи (1.5) та (1.7), матимемо:

$$F_\alpha = \lim_{r\to 0}\sum_i w_i^\alpha = \begin{cases} \infty, & \alpha < d \\ 0, & \alpha > d \end{cases}. \quad (1.8)$$

Отриманий вираз (1.8) є визначенням розмірності Хаусдорфа (Хаусдорфа-Безиковича). Тобто фрактальна розмірність d задає межу значень α, в якій $\alpha -$міра Хаусдорфа переходить з нескінченності в нуль. Значення F_α при $\alpha = d$ звичайно є скінченим, проте інколи може дорівнювати нулю або нескінченності. Це несуттєво, важливо лише за якого саме значення α величина F_α зазнає стрибка.

Чисельні значення фрактальних розмірностей Мінковського (1.4) і Хаусдорфа (1.8) для більшості множин збігаються. Важлива відмінність

полягає в тому, що розмірність Хаусдорфа будь-якої лічильної множини (у тому числі і лічильно-нескінченної) дорівнює нулю, а розмірність Мінковського – не обов'язково [3], [4].

Класичним прикладом фракталу є крива, придумана в 1904г. Хельге фон Кохом (рисунок 1.4).

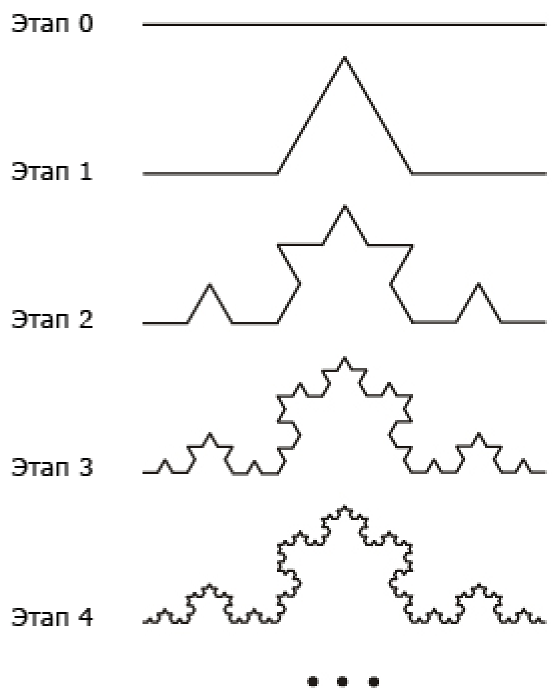


Рисунок 1.4 – Крива Коха

Фрактал розвивається внаслідок багаторазової дії деякого генеруючого перетворення. Для кривої Коха таким перетворенням є заміна відрізка на ламану, що складається із чотирьох однакових сегментів (рисунок 1.5). Починаючи з прямолінійного відрізка і поетапно застосовуючи перетворення до дедалі дрібніших елементів, у межі ми отримаємо ідеальний фрактал, криву Коха (рисунок 1.4).

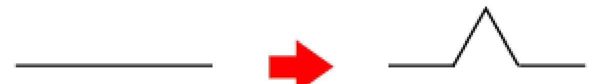


Рисунок 1.5 – Генеруюче перетворення кривої Коха

Проміжні стадії розвитку фракталу (тобто фігури, одержувані на етапах $k = 0,1,2,... < \infty$) називаються предфракталами.

Фрактали мають незвичайні властивості. Так, з погляду звичайної, «шкільної» геометрії довжина кривої Коха нескінчена! Це очевидний парадокс, оскільки ця крива обмежена та може бути покрита прямокутником скінченої площіни. Так, дійсно: на кожному етапі довжина фігури збільшується на третину. Спочатку $L_0 = 1$, після першого етапу – $L_1 = 4/3$, після другого – $L_2 = 16/9$. Довжина k –го предфрактала дорівнює $(4/3)^k$, тобто при $k \to \infty$ довжина кривої звернеться в нескінченність.

Фрактальну розмірність кривої Коха легко розрахувати аналітично. Покриватимемо її шаблонами розміром $r = 3^{-k}$ (що відповідає переходу від етапу до етапу). Очевидно, що кількість таких шаблонів становитиме $N(r) = 4^r$. Тоді згідно (1.4) розмірність Мінковського становитиме:

$$d = \lim_{r \to 0} \frac{\log(N(r))}{\log(1/r)} = \lim_{k \to \infty} \frac{\log(4^k)}{\log(3^k)} = \lim_{k \to \infty} \frac{\log(4)}{\log(3)} = \log_3 4 \approx 1.26. \quad (1.9)$$

З іншого боку, міра Хаусдорфа (1.8) для цього фрактала складає:

$$F_\alpha = \lim_{k \to \infty} \sum_i r_k^\alpha = \lim_{k \to \infty} \left(4^k \cdot 3^{-k\alpha} \right) = \lim_{k \to \infty} 3^{k(\log_3 4 - \alpha)}. \quad (1.10)$$

Очевидно, що точкою її переходу від ∞ в 0 є значення $d = \log_3 4$, таким чином обидва визначення дають те саме значення фрактальної розмірності аналізованої фігури.

Іншими прикладами геометричних фракталів є трикутник та килим Серпінського (рисунок 1.6). Їхня фрактальна розмірність становить $d_\Delta = \log_2 3 \approx 1.58$ та $d_K = \log_3 8 \approx 1.89$ відповідно.

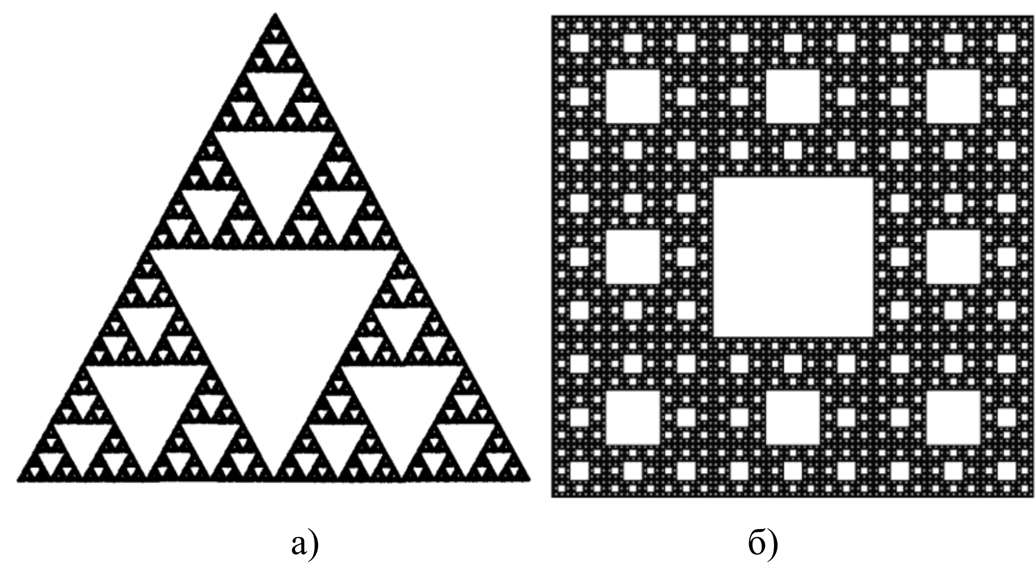


Рисунок 1.6 – Трикутник (а) та килим (б) Серпинського

Розглянуті вище геометричні фрактали є детермінованими. Крім точних фракталів, існують і випадкові. Так, берегові лінії, контури гірських хребтів, форми хмар є прикладами випадкових геометричних фракталів. Одним із перших «практичних застосувань» фрактальної розмірності був розрахунок довжини берегової лінії Великобританії, виконаний Мандельбротом [3], що послужило вагомим аргументом на користь актуальності та практичної значущості фрактального аналізу.

1.3 L-системи

Поняття L-систем з'явилося в 1968 завдяки біологу Арістиду Лінденмайеру. Очевидно, що самоподібність є характерною властивістю геометрії рослин. Гілка подібна до цілого дерева, молоді пагони подібні до гілок і т.д. Спочатку L-системи були розроблені для формального опису розвитку простих багатоклітинних організмів та для ілюстрації зв'язку між сусідніми клітинами рослини. Пізніше система була розширена для опису вищих рослин та інших складних структур, що гілкуються. Потім було

встановлено зв'язок між L-системами та формальними мовами.

По суті L-системи є набір рекурсивно застосовуваних правил, записаних формальною мовою і мають просту і наочну графічну інтерпретацію [17]. Застосування цих правил дозволяє будувати самоподібні геометричні фрактали. L-системи широко застосовуються в комп'ютерній графіці для побудови фрактальних дерев та рослин.

З формальної точки зору L-системи визначаються як кортеж:

$$L = (V, \omega, P), \tag{1.11}$$

де V (алфавит) – це множина змінних (символів, які можуть бути замінені), та констант (символи, які не можуть бути замінені),

ω (аксиома або ініціатор) – це рядок символів з **V**, який визначає початкове становище системи,

P – це множина породжуючих правил, виду A→B, де A і B є словами з алфавита V (прототип та наступник).

Усі елементи кортежу (алфавіт, аксіома, прототип, наступник, правила) є кінцевими множинами.

Правила граматики L-системи застосовуються ітеративно, починаючи з аксіоми (початкового стану). На кожній ітерації рядок стану проглядається ліворуч, і при виявленні прототипу будь-якого з правил він замінюється на відповідний наступник.

Для графічної реалізації L-систем як підсистема виведення використовується так звана тертл-графіка (turtle – черепаха). Символи рядка стану L-системи інтерпретуються як команди управління графічним курсором (черепашкою), яка рухається екраном, прокреслюючи свій слід, або переміщаючись без малювання, може змінити напрямок руху тощо.

Типовими командами черепашачої графіки є:

«F» (forward) – рух на задану відстань з промальовуванням сліду;

«f» – рух вперед на задану відстань без промальовування сліду;

«+» – зміна напрямку руху (поворот голови черепашки) на фіксований кут проти годинникової стрілки;

«–» – поворот на заданий кут за годинниковою стрілкою;

«[» – запам'ятати в стеку параметри (позицію та напрямок голови) черепашки;

«]» – відновити зі стека параметри черепашки.

Наприклад, нехай алфавіт L-системи складається із символів V = (F, +, –) (де «F» – змінна, а «+» и «–» – константи), аксіома дорівнює ω = «F», а породжувальне правило P має вигляд $F \rightarrow F + F - - F + F$. Якщо інтерпретувати одержувані рядки як команди управління черепашкою, використовуючи кути повороту $\pm 60°$ і зменшувати на кожній ітерації крок черепахи втричі, то отримаємо вже знайому криву Коха (таблиця 1.1).

Таблиця 1.1 – L-система, що породжує криву Коха

k	Рядок стану L-системи	Зображення
0	F	
1	F+F--F+F	
2	F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F	
3	F+F--F+F+F+F--F+F--F+F--F +F+F+F--F+F+F+F--F+F+F+F --F+F--F+F--F+F+F+F--F+F-- F+F--F+F+F+F--F+F--F+F--F +F+F+F--F+F+F+F--F+F+F+F --F+F--F+F--F+F+F+F--F+F	

L-система є контекстно-вільною, якщо кожне правило виведення посилається лише на індивідуальні символи, а не на їхніх сусідів. Якщо існує точно одне правило для кожного символу алфавіту, то L-система називається детермінованою (детермінована контекстно-незалежна L-система називається D0L системою). Якщо є кілька правил і кожне вибирається з певною ймовірністю на кожній ітерації, то це стохастична L-система.

Прикладом використання стека в L-системах служить система з правилом, що породжує $F \to F[-F]F[+FF]F$, аксіомою $\boldsymbol{\omega} = "F"$ та кутом поворота $\pm\pi/7$. Графічне відображення цієї системи показано на рисунку 1.7. Саме для створення подібних моделей L-системи й були створені.

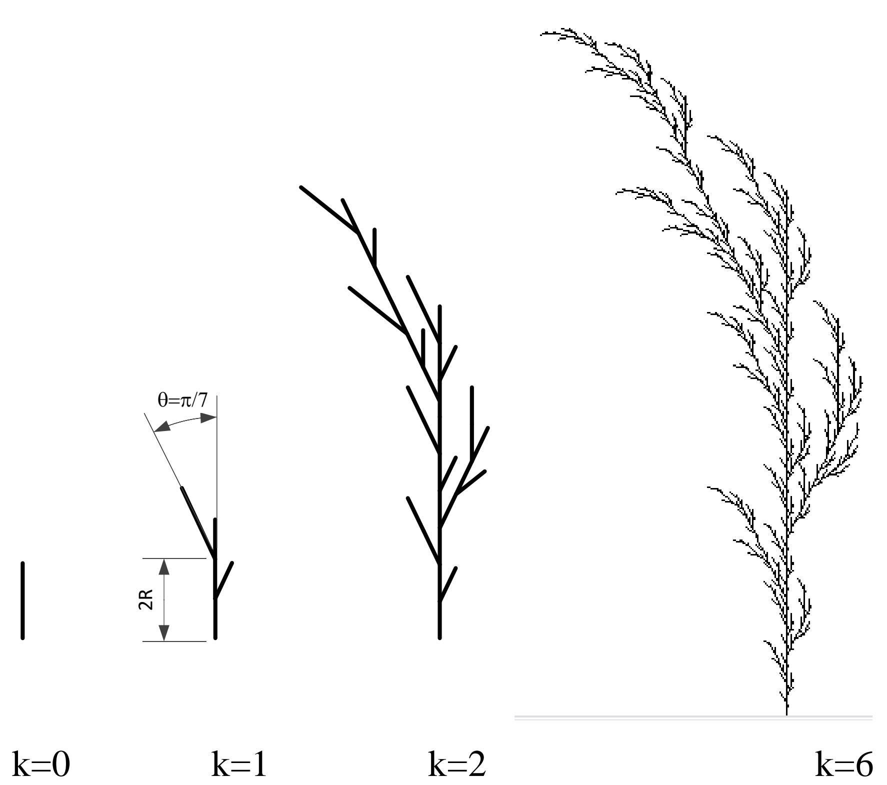


Рисунок 1.7 – Візуалізація L-системи «бур'ян»

1.4 Фрактальна розмірність систем, які зростають

L-системи, розглянуті у попередньому розділі, загальновизнано відносять до фракталів. Їхня самоподібність очевидно випливає як з алгоритму побудови (тобто зі структури рядків стану), так і проявляється візуально. Водночас, автори класичних підручників та монографій [4], [17], [18] обходять мовчанням питання про фрактальну розмірність «бур'янів» (рисунок 1.7), та інших «кущів» та «дерев», для моделювання яких ці L-системи і були створені.

Це відбувається зовсім не випадково. Вся теорія фрактальної розмірності, в тому числі й обидва класичні її визначення – Хаусдорфа (1.8) та Мінковського (1.4) – орієнтовані на множини, які є замкнутими (компактними) та нескінченно подільними. У той же час, такі фрактали, як L-системи, мають прямо протилежні властивості: вони є нескінченно зростаючими і не є нескінченно подільними; існує деякий мінімальний масштаб (відповідний вихідній аксіомі системи), дрібніше за який фрактал вже не дробиться. Точніше кажучи, сама по собі аксіома є не фракталом, а звичайною геометричною фігурою, наприклад, лінією, як у прикладах таблиці1.1 та рисунку 1.7.

Можна сміливо сказати, що традиційні фрактали «нескінченні внутрішньо», тобто. у бік мікросвіту, тоді як L-системи є прикладом фракталів, «нескінченних назовні» (у бік макросвіту). Очевидно, що для розрахунку розмірності L-систем та інших подібних фракталів необхідно внести корективи [6], [11] до визначення фрактальної розмірності (1.4), (1.8).

Насамперед варто звернути увагу на те, що у визначенні розмірності Мінковського (1.4) присутня деяка ніби «недомовність»: фрактальна розмірність, будучи безмасштабною мірою множини, визначається за допомогою величини r, яка має свій масштаб. Чому, наприклад, дорівнюватиме розмірність кривої Коха (рисунок 1.4, таблиця 1.1), якщо

довжина вихідного відрізка становить 1/3, а лінійний розмір шаблону, як і раніше, буде дорівнювати $r = 3^{-k}$? Очевидно, що тоді кількість необхідних шаблонів становитиме $N = 4^{k-1}$ (при $N_0 = 1$), та згідно з (1.4) легко отримати, що $d = \lim_{r \to 0} \frac{\log(N(r))}{\log(1/r)} = \lim_{k \to \infty} \frac{(k-1) \cdot \log(4)}{k \cdot \log(3)} = \log_3 4$. Ситуація повністю аналогічна «квадратурі кола», тобто виміру площі кола квадратиками (рисунок 1.2-1.3): значення фрактальної розмірності збережеться, але у межі, а не локально, тобто фрактал втратить право називатися «ідеальним».

Очевидно, що у визначенні фрактальної розмірності за Мінковським (1.4) неявно мається на увазі початкова умова: початковий лінійний розмір шаблону R є мінімально можливим серед таких, які забезпечують $N_0 = 1$, тобто покривають фрактал повністю. Тоді визначення розмірності набуде вигляду:

$$d = \lim_{r \to 0} \frac{\log(N(r))}{\log(R/r)}. \qquad (1.12)$$

Визначення (1.12) збігається з класичним (1.4), якщо вважати, що $R = 1$, тобто розміри шаблонів вимірюються в частках від R.

Проведене уточнення визначення розмірності Мінковського зовсім не є «косметичним», а дозволяє узагальнити це визначення на фрактали, що нескінченно ростуть, зокрема на L-системи. Зокрема, зафіксуємо розмір шаблону ($r = 1$), збільшуватимемо розмір вікна (R), тобто видимої частини нескінченного фракталу, і підраховуватимемо кількість необхідних шаблонів $N(R)$. При цьому вважаємо, що генеруюче перетворення (породжуюче правило L-систем) діє завжди на одному й тому ж самому масштабному рівні (рисунок 1.8). Фрактальна розмірність Мінковського для таких фракталів набуде вигляду:

$$d = \lim_{R \to \infty} \frac{\log(N(R))}{\log(R)}, \qquad N(1) = 1. \tag{1.13}$$

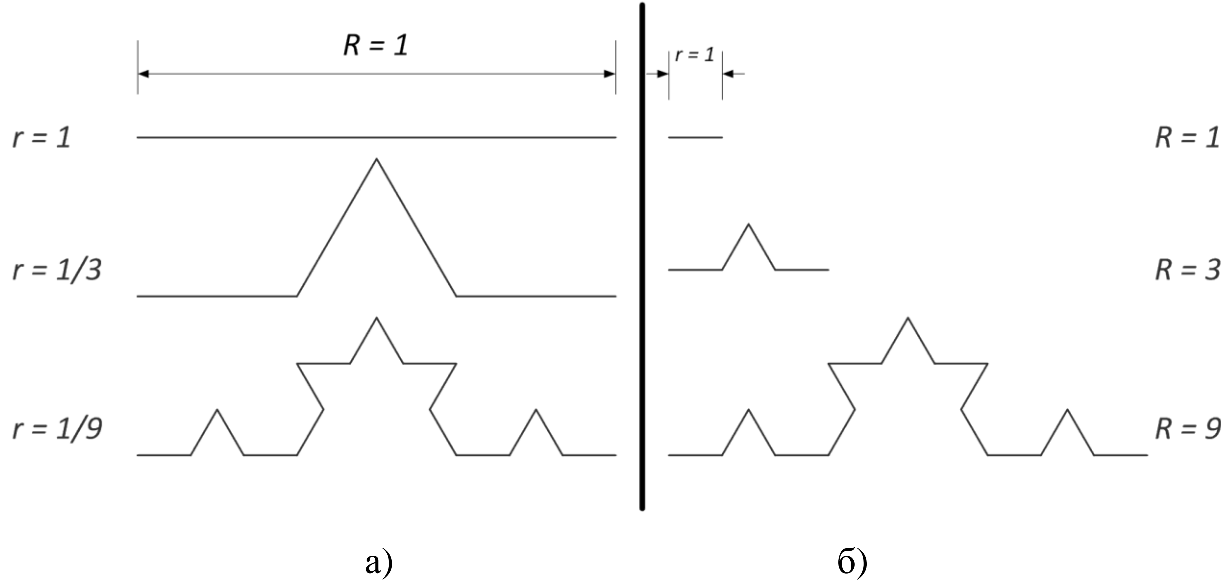


Рисунок 1.8 – Криві Коха, нескінченна всередину (а) та назовні (б)

Очевидно, що розмірність кривої фон Коха, що розглядається в цьому прикладі, в обох випадках становитиме: $d = \log_3 4 \approx 1.26$.

Як уже говорилося, такі фрактали можна розглядати як класичні компактні «вивернуті навиворіт»: кінцевому розміру звичайних фракталів (нескінчених всередину) відповідає кінцевий розмір аксіом L-систем, а нескінченому зменшенню розміру використовуваних шаблонів – нескінчене зростання розміру вікон. Позначимо $s = R / r$ («s» від слова size) – відношення лінійних розмірів вікна та шаблону. Використання цієї величини у формулах фрактальної розмірності виглядає переважним вже хоча б тому, що вона є безрозмірною. З урахуванням введеного позначення визначення фрактальної розмірності (1.12)-(1.13) можна узагальнити:

$$d = \lim_{s \to \infty} \frac{\log(N(s))}{\log(s)}, \qquad s = \frac{R}{r}. \tag{1.14}$$

Визначення (1.14) дозволяє розраховувати фрактальні розмірності L-систем. Так, якщо в моделі бур'яну (рисунок 1.7) прийняти висоту початкової пагоні за одиницю ($r=1$), то висота вікна (тобто висота «рослини») на етапі k становитиме $R=(2+2\cos\theta)^k$, а кількість шаблонів (рівна кількості символів «F» у рядку стану) буде дорівнювати $N=6^k$. Беручи до уваги, що $\theta=\pi/7$, з (1.14) отримаємо:

$$d=\lim_{s\to\infty}\frac{\log(N(s))}{\log(s)}=\lim_{k\to\infty}\frac{\log(6^k)}{\log((2+2\cos\theta)^k)}=\frac{\log(6)}{\log(2+2\cos\theta)}\approx 1{,}34\,. \quad (1.15)$$

Очевидно також, що визначення одиниці виміру шаблону впливає лише на значення α-міри Хаусдорфа F_α (1.8), але ніяк не на точку переходу цієї міри з нескінченності в нуль.

Ще одним важливим доповненням до теорії фрактальної розмірності може бути поняття локальної фрактальної розмірності [5], [10], [11].

Незважаючи на простоту, спільність та математичну коректність визначення розмірності Мінковського (1.4), практичні методи оцінювання фрактальної розмірності множин не засновані безпосередньо на цьому визначенні (як не засновані вони і на (1.13), (1.14) у разі нескінченно зростаючих множин). Зазвичай фрактальну розмірність множин оцінюють як коефіцієнт регресії за залежністю $N(s)$, побудованої у подвійному логарифмічному масштабі (тобто між величинами $\log(N)$ та $\log(s)$), тоді як згідно з визначенням слід було б обчислювати коефіцієнт нахилу асимптоти цієї залежності при $s\to\infty$.

Безвідносно до причин такої підміни (якими можуть бути і обчислювальна складність оцінювання нахилу асимптоти, і фізичні обмеження, що накладаються на експеримент, і кінцева область застосування самої фрактальної моделі) такий підхід фактично означає застосування (1.14) не в оригінальній, а в диференційній формі:

$$d(s) = \frac{\mathrm{d}\left(\log(N)\right)}{\mathrm{d}\left(\log(s)\right)}. \qquad (1.16)$$

Згідно з правилом Лопіталя, якщо локальна розмірність (1.16) має межу, то вона збігається з глобальною розмірністю (1.14). Саме цим і обґрунтовано коректність застосування диференціальної форми.

Важливою властивістю ідеальних фракталів (кривої Коха, Серпінського трикутника, L-систем та інших) є збіг локальної розмірності з глобальною. Точніше, цих фракталів існує послідовність значень s, що прагне нескінченності, така, що з будь-якому значенні з цієї послідовності локальна розмірність суворо дорівнює глобальної.

Продиференціювавши вираз (1.16), отримаємо, що локальна фрактальна розмірність d співпадає з коефіцієнтом еластичності ε_s^N функції $N(s)$ відносно s:

$$d(s) = \frac{dN}{N} \bigg/ \frac{ds}{s} = \frac{dN}{ds} \cdot \frac{s}{N} = \varepsilon_s^N(s). \qquad (1.17)$$

Саме поняття еластичності та коефіцієнт еластичності (1.17) широко використовуються в математиці (і, зокрема, у математичних моделях хімічних та економічних процесів) як міра відносних швидкостей зростання функції та її аргументу.

Таким чином, зіставивши визначення (1.14), (1.16) та (1.17), можна сказати, що фрактальна розмірність множини (за Мінковським) дорівнює межі коефіцієнта еластичності цієї множини, якщо така існує.

2 СТРУКТУРИ ДАНИХ ДЛЯ ІНДЕКСАЦІЇ ЛІНІЙНИХ ТА ПРОСТОРОВИХ ОБ'ЄКТІВ

2.1 Особливості роботи з дисковою пам'яттю

B-дерева – це збалансовані дерева пошуку призначені для ефективної роботи з дисковою пам'яттю. Вони були запропоновані Байєром та МакКрейтом (Bayer, McCreight) у 1970 році. Чи вказує буква «B» (читається «Бі») на автора, або ж на область застосування (бази даних) – невідомо.

Як відомо, словникові операції у двійковому дереві пошуку мають складність O(h), де h – висота дерева. Якщо дерево збалансоване (RBT, АВЛ, WAVL), то $h \leq 2\log n$. З цього випливає, що при $n = 10^8$ (що не дуже багато) висота дерева складе $27 \leq h \leq 54$. Саме стільки вузлів потрібно прочитати під час пошуку відсутнього значення. У кожному вузлі виконуються два елементарні порівняння (або одне тризначне: >, <, =) і одне присвоювання адресу, таким чином, загальна кількість елементарних операцій при пошуку складе $T_{Search} \leq 3h$, тобто в даному прикладі – порядка сотні.

На сучасних персональних комп'ютерах кожна з елементарних операцій виконується за 50-100 наносекунд, таким чином весь процес пошуку міг би зайняти час від 4 до 17 мкс. Очевидно, що на практиці така швидкодія недосяжна. І насамперед тому, що структури даних розміщуються не в оперативній пам'яті, а на жорсткому диску.

Час доступу до жорсткого диска (у довільному режимі) складає 8-12 мс. Він визначається переважно швидкістю обертання диска (характерна швидкість – 7200 об/мин = 120 звернень до диска на секунду). Відповідно, 54 операції читання займуть 0,45 сек. Іншими словами, в режимі довільного доступу доступ до диска в 100 000 ÷ 500 000 разів повільніше, ніж до фізичної оперативної пам'яті. Таку різницю у швидкостях можна

проілюструвати наступним прикладом: нехай електронне повідомлення надходить із Харкова до Києва за 1 секунду. Тоді швидкість, в 100000 разів менша, означає, що для його доставки потрібно 28 годин. Очевидно, що в таких умовах кількість операцій з обробки вузла (чи три, як при пошуку, чи 10-20, як при обертаннях вузлів дерева) не впливає на підсумковий час доступу.

З іншого боку, відомо, що дані диска зчитуються/записуються посторінково. Розмір сторінки становить від 512 до 64Кбайт. Задамо розмір вузла строго рівним розміру сторінки (вирівняємо розмір структури вузла межі сторінки). Тоді в одному вузлі можна буде розмістити не один ключ із двома вказівниками, як у двійковому дереві, а $m-1$ ключей с m вказівниками. Цей коефіцієнт розгалуження (називається порядком В-дерева) становить від 50 до 2048. Наприклад, при розмірі сторінки 16К та розмірі ключа з даними (або вказівником на них) 80 байт, значення m складе 196 ($195 \cdot 80 + 196 \cdot 4 = 16384$).

Мінімальний ступінь вузла в В-деревах дорівнює

$$t = \lceil m/2 \rceil, \tag{2.1}$$

таким чином, у кожному вузлі зберігається від $t-1$ до $m-1$ ключей.

Легко довести, що висота В-дерева з $n \geq 1$ ключами та мінімальним ступенем $t \geq 2$ знаходиться в межах

$$\log_m(n+1) - 1 \leq h \leq \log_t \tfrac{n+1}{2}. \tag{2.2}$$

У прикладі, що розглядається ($n = 10^8$, $t = 98$), з (2.2) отримаємо, що $2.49 \leq h \leq 3.87$. Таким чином, незважаючи на те, що пошук вимагатиме близько 500 елементарних операцій (пошук ключа всередині вузла має складність $\theta(m)$), кількість читань з диска становитиме лише h, тобто 3 або 4 (вважаючи, що кореневий вузол завжди є завантаженим в оперативну

пам'ять). В результаті час виконання пошуку в гіршому випадку становитиме $25 \div 33$мс, тобто майже в 14 разів швидше ніж для двійкового дерева. Для операцій вставки та видалення різниця в ефективності між B-і BST-деревами буде ще більшою.

З проведеного аналізу можна дійти висновку, що під час роботи з дискової пам'яттю основним показником часової складності алгоритму є кількість звернень до диску, тобто кількість оброблюваних вузлів. Тому алгоритми обробки мають бути однопрохідними, тоді покажчики на батьківські вузли відсутні за непотрібністю. Таким чином, B-дерева є природним узагальненням двійкових дерев пошуку.

Індекси більшості великих БД зберігаються саме в B, B^{+}, B^{*}-деревах.

2.2 Структура B-дерев

B-дерева мають наступні властивості:

– кожен вузол має не більше ніж m дочірніх;

– усі некореневі вузли мають не менше ніж $t = \lceil m/2 \rceil$ дочірніх;

– якщо корінь не є листом, то він має не менше 2 дочірніх;

– все листя розташоване на одній глибині (рівній висоті дерева, h).

Для реалізації цих властивостей вузли (x) повинні мати такі поля:

– фактична кількість ключів у вузлі x.n;

– масив значень ключей x.key[m-1] (дійсними є x.n элементів), впорядкований за зростанням;

– масив вказівників x.ch[m] на дочірні вузли (викорисовуються 1+x.n елементів);

– масив вказівників на дані x.dat[m-1], відповідні ключам.

Дочірній вузол x.ch[0] містить ключі, менші, ніж x.key[0], вузли x.ch[i] містять ключі, більші x.key[i-1], але менші, ніж x.key[i], ключі вузлів піддерева x.ch[n-1] більше x.key[n-2]. Таким чином, ключі

батьківського вузла є роздільниками піддіапазонів ключів, які зберігаються в дочірніх піддеревах.

Листові вузли не мають дочірніх, тому їх поля x.ch рівні NULL.

Фіктивні вузли (NIL-вузли) в даній версії B-дерев не використовуються.

Для моделювання дискових операцій зручно використовувати функції-лічильники Disk_Read(x) та Disk_Write(x). Вони імітують звернення до диска, пов'язане з читанням чи збереженням змін у вузлі x. У навчальних програмах ці функції просто підраховують кількість своїх викликів.

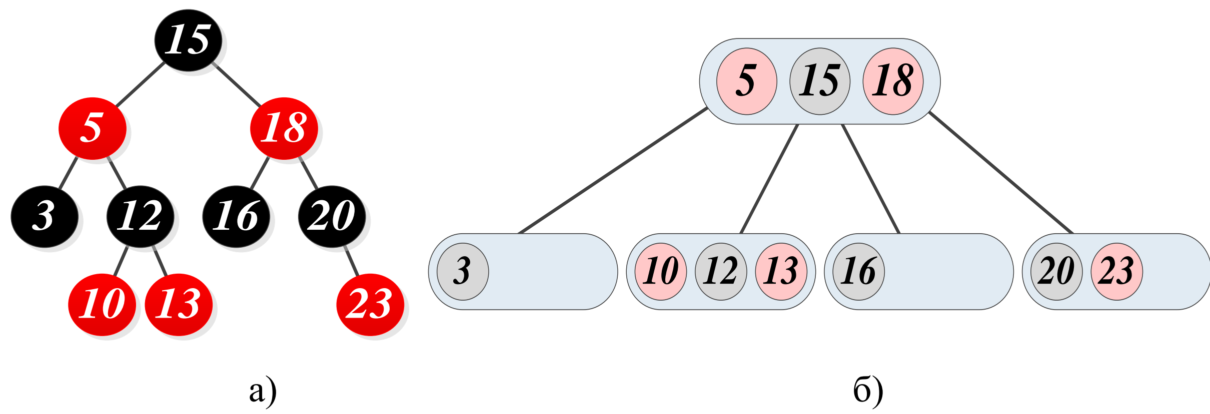
Вважається, що корінь дерева завжди відображений в оперативну пам'ять, тому викликати Disk_Read(root) ніколи не потрібно. У той же час, якщо кореневий вузол змінювався, виклик Disk_Write(root) необхідний.

Існують відмінності в термінології, позначеннях та реалізації B-дерев. Перша пов'язане з трактуванням поняття «лист». Як і для двійкових дерев пошуку, деякі автори (зокрема, Д.Кнут) називають листями (leaves) або зовнішніми вузлами (external nodes) nil-вузли, обмовляючи, що вони не містять ані ключів, ані покажчиків на дочірні елементи. Тоді звичайні вузли називають внутрішніми (internal). Інші автори (як Bayer & McCreight) називають листями вузли, розташовані на один рівень вище, тобто такі, у яких всі дочірні є NULL. Я надалі буду дотримуватись останнього варіанту трактування поняття «лист».

Друга відмінність специфічна саме для B-дерев і відноситься до поняття «порядок» дерева. Кормен ([CLRS]) характеризує B-дерево його мінімальним ступенем (minimum degree), тобто мінімально допустимою кількістю дочірніх елементів (t) для некореневого вузла. Аналогічно творці B-дерев, Байєр і МакКрейт називали порядком дерева мінімальну кількість ключів, тобто $t - 1$. Але, як справедливо зазначив Д.Кнут, при такому підході максимальна ступінь вузла (m) визначена неточно: значенням $m = 6$

і m = 5 відповідає одне й те саме значення $t = \lceil m/2 \rceil = 3$, тому доводиться додатково обговорювати, чи є m парним, чи ні.

Одним із найпростіших окремих випадків B-дерева є дерево порядку $m = 4$. Його вузли можуть мати 2, 3 або 4 дочірні вузли і, відповідно, містити 1, 2 або 3 ключі. Таке дерево називається 2-3-4-деревом. Червоно-чорні дерева (RBT) можна розглядати як частковий випадок 2-3-4 дерева (рисунок 2.1).



а) б)

Рисунок 2.1 – Червоно-чорне дерево та B-дерево порядку m=4

2.3 Словникові операції над B-деревами

Процедура пошуку за ключом (B_Tree_Search) є природним узагальненням процедури Tree_Search, що застосовується у двійкових деревах:

Лістинг 2.1 – Код функції Tree_Search

```
B_Tree_Search(root, k)
   x = root

   loop_forever
      i=0
      while (k > x.key[i] and i<x.n):    i = i + 1
      if (i < x.n and k==x.key[i]):      return x.dat
      if (x.ch[i]==NULL):                return NULL
      x = x.ch[i]
      Disk_Read(x)
   end_loop
```

Ця функція проходить вузли в низхідному порядку, кількість звернень до диску $(d-1)$ на одиницю менше глибини шуканого вузла, тобто становить $O(h)$. Для прискорення пошуку ключа у вузлі можна застосувати бінарний пошук.

Вставка ключа в B-дерево схожа, але дещо складніша за вставку в двійкове дерево пошуку загального вигляду. Після визначення позиції у листовому вузлі, у яку слід вставити новий ключ, може бути, що цей вузол вже заповнений і вставити новий ключ у нього не можна. Вузол дерева називається заповненим (full), якщо містить максимально можливу кількість (m) дочірніх елементів (тоді x.n=m-1). У цьому випадку слід провести розбиття цього вузла на два відносно медіани масиву ключів. Після цього медіану (разом із покажчиком на новий вузол) необхідно вставити у батьківський вузол. Очевидно, що батьківський вузол також може виявитися заповненим, таким чином, процес розбиття може (у гіршому випадку) піднятися висхідною до кореня. Забігаючи наперед, зазначимо, що це неприпустимо.

Описана процедура розбиття за своєю суттю (і за призначенням) є типовою Fixup-процедурою, аналогічною застосовуваним у збалансованих двійкових деревах (RBT, АВЛ, WAVL).

При реалізації розбиття саме час згадати про важливу різницю в мірах складності процедур обробки B і BST дерев. Якщо для BST-дерев перехід від одного вузла до іншого є звичайною операцією, що нічим не відрізняється за вартістю від обробки даних всередині вузла, то для B-дерев складність процедур пропорційна кількості звернень до вузла, кожне з яких вимагає від 1 до 2 дискових операцій.

З цього випливає, що процедура вставки B_Tree_Insert має бути реалізована в один прохід (нисхідний): при вставці ключа слід розбивати («про всяк випадок») всі заповнені вузли вздовж маршруту проходу. Така реалізація гарантує, що після вставки підйом по дереву не

знадобиться; кількість звернень до диску складе $O(h)$, а кількість операцій – $O(t \cdot h)$.

Лістинг 2.2 – Код функції B_Tree_Insert

```
B_Tree_Insert(root, k)
   x = root
   if x.n == m-1
      s = Allocate_Node()
      root = s;      s.n = 0;      s.ch[0] = x
      B_Tree_Split_Child(s, 0, x)
      B_Tree_Insert_Nonfull(s, k)
   else
      B_Tree_Insert_Nonfull(x, k)
```

Процедура B_Tree_Insert перевіряє заповненість кореня, при необхідності створює новий корінь, а старий розбиває. По суті вона є оболонкою для «справжньої» процедури вставки – B_Tree_Insert_Nonfull.

Допоміжна процедура Allocate_Node() виділяє пам'ять (майбутню дискову сторінку) нового вузла і повертає покажчик цей вузол. Вона не здійснює дискових операцій.

Процедура B_Tree_Insert_Nonfull(x,k) виконує основну роботу зі вставки ключа k в піддерево з незаповненим вузлом x.

Лістинг 2.3 – Код процедури B_Tree_Insert_Nonfull(x,k)

```
B_Tree_Insert_Nonfull(x, k)
   is_actual = TRUE

   while (x.ch[0] <> NULL)   // поки вузол x - не лист
      i = позиція k в масиві ключей x.key
      if not is_actual
         Disk_Read(x.ch[i])

      if ((x.ch[i]).n==m-1) //  чи не є повним дочірній?
         B_Tree_Split_Child(x,i,x.ch[i]) //дробимо,якщо 
так
         if (k > x.key[i])   // уточнюємо,в який з 
дочірніх
```

```
            i = i + 1          // слід спускатись

         is_actual = TRUE
      else
         is_actual = FALSE

      x = x.ch[i] //Спускаємось,переходячи до
next ітерації
   Вставлення ключа k в x.key   // вузол x – лист
   Инкрементувати x.n

   Disk_Write(x)
```

Процедура B_Tree_Split_Child(x,i,y) викликається у тому випадку, коли необхідно спуститися з незаповненого вузла x до i-го дочірнього вузол y (x.ch[i]=y), який заповнений. Як було сказано раніше, ця процедура розбиває вузол у медіані ключів, створює новий вузол, і вставляє цю медіану в масив ключів батьківського вузла (тобто вузла x). Функції вставки побудовані так, що на момент виклику B_Tree_Split_Child(x,i,y) обидва оброблювані вузли (x,y) вже завантажені у пам'ять. Схему виконання цієї процедури наведено на рисунку 2.2, а на рисунку 2.3 наведено приклади послідовного виконання вставки.

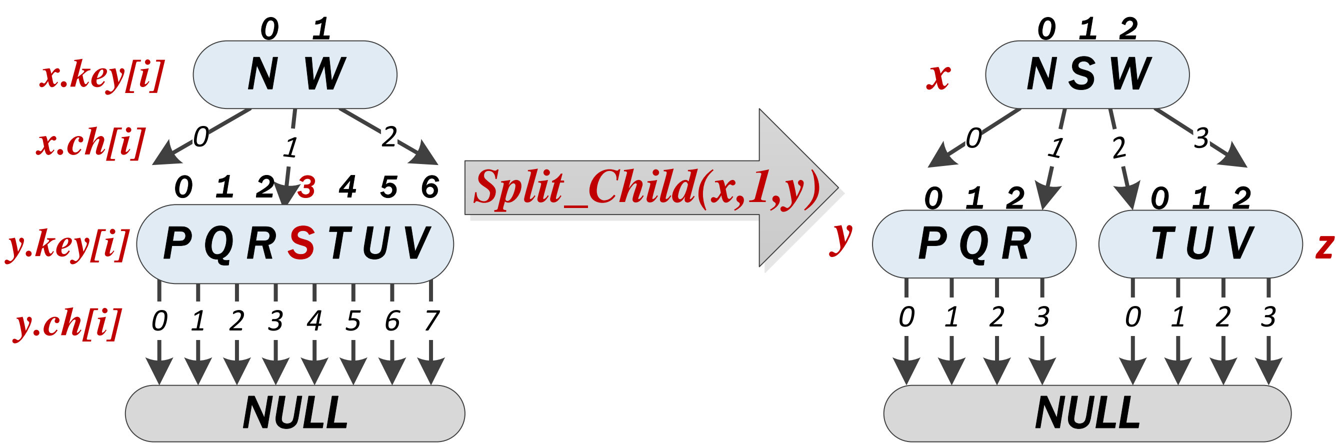


Рисунок 2.2 – Розбиття вузла в B-дереві (m=8)

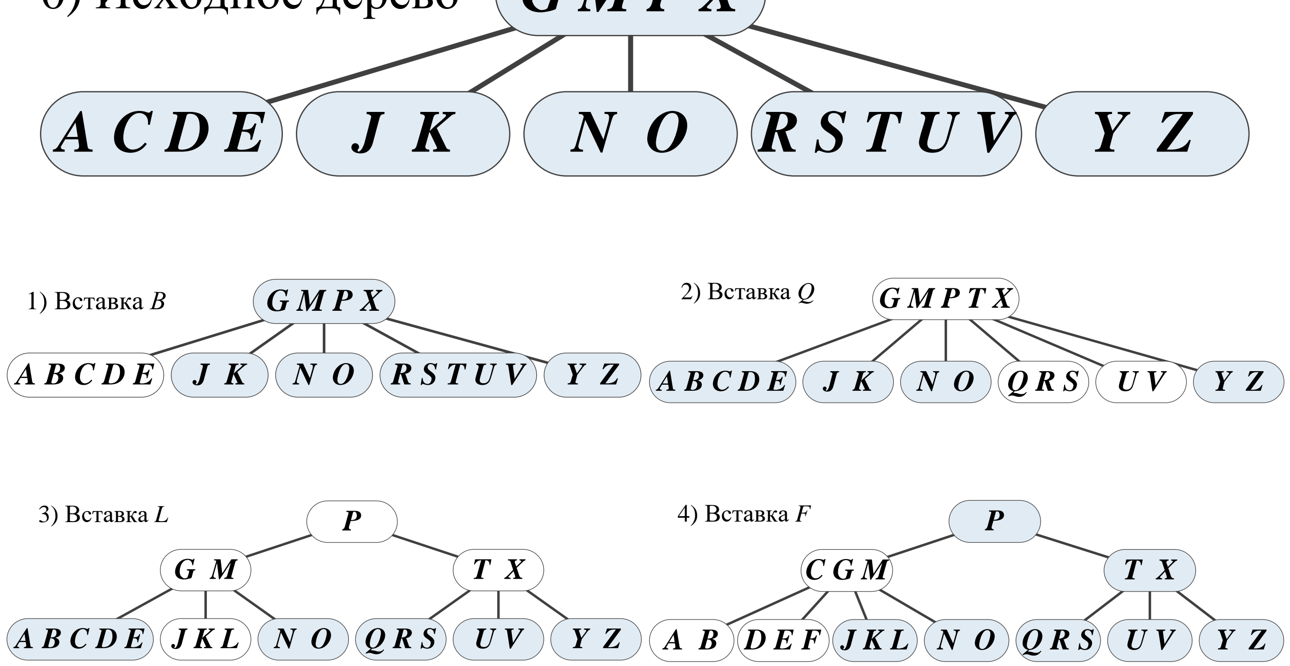


Рисунок 2.3 – Вставка ключів в B-дерево (m=6)

Лістинг 2.4 – Код процедури B_Tree_Split_Child(x, i, y)

```
B_Tree_Split_Child(x, i, y)
   z = Allocate_Node()
   t = ⌈m/2⌉ - 1         // позиція медіани
   z.n = m - t - 2       // кількість ключей в новому
вузлі z
перенести необхідні ключи та покажчики з y.key та y.ch в z
   y.n = t               // оновити кількість ключей в
вузлі y
зсунути ключі та покажчики в x та вставити ключ та адрес z
   x.n = x.n + 1         // оновити кількість ключей в
вузлі x
   Disk_Write(y)
   Disk_Write(z)
   Disk_Write(x)
```

Видалення ключа з дерева, як і для інших дерев, складніше, ніж вставка. За аналогією з двійковими деревами, вузол x, з якого видаляється ключ k=key[i], може бути листовим, чи ні. У першому випадку слід просто видалити ключ з цього листового вузла (рисунок 2.4(1)), а в другому – замінити ключ, що видаляється, ключем-predecessor або ключем-successor (тобто максимальним дочірнім вузлом x.ch[i], або мінімальним

ключем дочірнього вузла (x.ch[i+1]), після чого перейти до видалення ключа з дочірнього вузла. Наприклад, при видаленні вузла M (рисунок 2.4(2)), він замінюється максимальним ключем вузла JKL, після чого відбувається спуск у це дочірнє піддерево і вирішується задача видалення з нього ключа L. Якщо б було потрібно видалити з дерева (рисунок 2.4(1)) ключ G, то він би замінився на ключ-successor J (мінімальний в JKL). Цей випадок симетричний попередньому та на рисунку 2.4 не показаний.

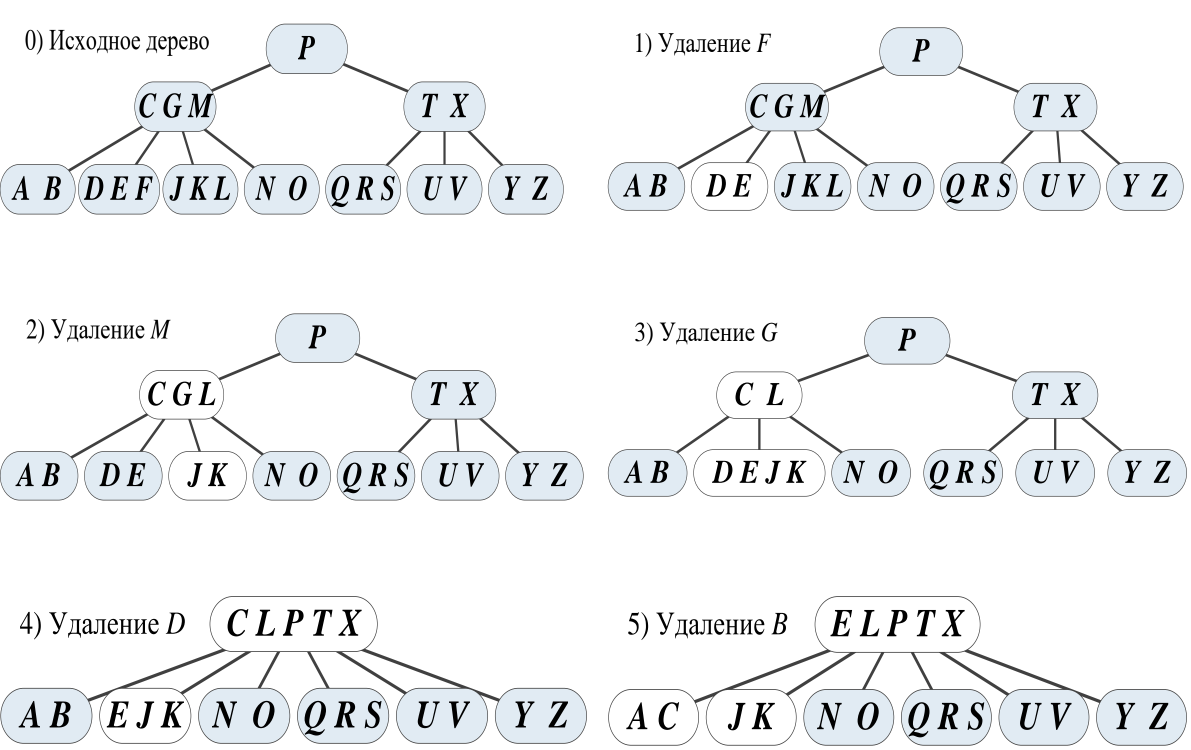


Рисунок 2.4 – Видалення ключів з B-дерева (m=6, t=3)

В обох випадках може порушитись властивість заповненості B-дерева (некореневий вузол повинен мати хоча б $t - 1$ ключів). Наприклад, у дереві рисунку 2.4(2) дочірні вузли ключа G (DE і JK) мають по два ключі, тому G не можна замінити ані predecessor'ом (E), ані successor'ом (J). Вихід у ситуації очевидний: слід об'єднати i і $i + 1$ дочірні вузли, використовуючи key[i] (у прикладі це G) як медіану (рисунок 2.4(3)). Після цього ключ G можна буде видалити з вузла CGL (не забувши зсунути його покажчики на

дочірні вузли) та перейти до задачі видалення ключа з цього об'єднаного вузла.

У всіх перерахованих випадках вузли, що містять ключ, що видаляється, мали більше мінімально необхідної кількості ключів. Якби ця умова була порушена, то довелося б рекурсивно підніматися нагору, коригуючи заповненість. Але, як зазначалося, для B-дерев такий двопрохідний алгоритм був би дуже нераціональним. Подібно до того, як при вставці перед спуском у дочірній вузол його перевіряють на заповненість і якщо треба розбивають на два «про всяк випадок», гарантуючи цим відсутність необхідності підйому, так і при видаленні слід перед спуском у дочірній вузол забезпечити, щоб він мав не менше ніж t ключів (тобто хоча б один «про запас»).

Якщо ключ (k) відсутній у поточному вузлі x, то так само, як і при пошуку, знаходять індекс (i) піддерева, в якому він повинен знаходитися. Якщо цей дочірній вузол (x.ch[i]) містить t або більше ключів, то можна безпечно спуститися і продовжити пошук/видалення. Якщо і вузол x.ch[i], і обидва його найближчих сусіда (x.ch[i-1] і x.ch[i+1]) містять по $t-1$ ключів, то вузол x.ch[i] об'єднується з одним із цих сусідів. Наприклад, при пошуку ключа D з метою його видалення з дерева (рисунок 2.4(3)), на першому ж кроці (тобто при x=root) виявиться, що вузол CL, який слід спуститися, та його правий сусід (TX) містять менше ніж t вузлів (а лівого сусіда немає). Тому (рисунок 2.4(4)) вузли CL і TX об'єднуються через медіану (P).

Як видно з цього прикладу, якщо поточний вузол є коренем, то при об'єднанні дочірніх вузлів він може стати порожнім. Тоді він, очевидно, видаляється, а його єдиний дочірній вузол (CLPTX) стає новим коренем. Та ж ситуація виникла б при видаленні самого ключа P.

Після об'єднання вузлів перевіряється заповненість необхідного дочірнього вузла (DEJK), відбувається спуск, ключ D знаходиться та видаляється з цього листа.

І, нарешті, останній випадок (рисунок 2.4(5)) відповідає ситуації, коли дочірній вузол x.ch[i] поточного вузла x має $t-1$ ключів, але один із його найближчих сусідів має їх t або більше. Тоді із цього сусіда передається ключ (разом із покажчиком) у вузол x.ch[i]. Якщо сусід лівий – то максимальний ключ, якщо правий – мінімальний. Так, при першому кроці видалення вузла з дерева (рисунок 2.4(5)) з'ясовується, що x.ch[0] (тобто вузол AB) має всього два ключа, але його сусід (EJK) має їх три. Тоді ключ E (key[0]) разом із покажчиком ch[0] вузла EJK переносяться у вузол AB. Потім відбудеться спуск у вузол ABE, ключ буде в ньому знайдений і видалений.

Нижче наведено короткий псевдокод функції B_Tree_Delete

Лістинг 2.1 – Псевдокод функції B_Tree_Delete

```
B_Tree_Delete(root, k)
  x = root
  while (x <> NULL)
    i=0
    while (k > x.key[i] and i<x.n):   i = i + 1
    if (i < x.n and k==x.key[i])
      if (x.ch[i]==NULL)      // Рис.2.4(1)
        Видалити x.key[i] та x.ch[i] з x
        Disk_Write(x)
      else                    // Рис.2.4(2), Рис.2.4(3)
        y = x.ch[i]
        Disk_Read(y)
        Перевірити заповненість вузла y. Якщо припустима, то
          замінити x.key[i] максимальним ключем y
          Disk_Write(x)
          x = y
          k = km   // km - це знайдений максимальний ключ y
        інакше
          z = x.ch[i+1]
          Disk_Read(z)
```

```
          Перевірити заповненість вузла z. Якщо припустима,то
            замінити x.key[i] мінімальним ключем z (=km)
            Disk_Write(x)
            x = z
            k = km
          інакше                // Рис.2.4(3)
           Об’єднати y та z через медіану x.key[i] в вузол y
           Видалити x.key[i],x.ch[i+1] з вузла x, видалити z
           Disk_Write(x)
           x = y
    else  // Рис.2.4(4) и рис.2.4(5)
       реализовать самостоятельно
  end_while
end
```

2.4 Модифікації В-дерев

Основними модифікаціями В-дерев є B^{+}, B^{*} и B^{*+}-дерева. Іноді їх також називають В-деревами.

Мінімальний ступінь вузла B^{*}-дерева підтримується рівним 2/3m (а не 1/2m, як у звичайному В-дереві). Перед розбиттям вузла здійснюється спроба «переливу» зайвих вузлів у сусідній (правий чи лівий), щоб їх стало порівну. Тільки якщо перелив неможливий, то відбувається розбиття: x.ch[i] та x.ch[i+1] розбиваються на три вузла. Сенс такої модифікації полягає у зменшенні кількості вузлів за рахунок більш компактного пакування. Платою за цю економію пам’яті є більша складність розбиття (та об’єднання) вузлів під час вставки/видалення.

У B^{+}-дереві дані (або покажчики на них) зберігаються лише у листах. У нелистових вузлах знаходяться лише копії ключів та вказівники на дочірні вузли (рисунок 2.5). Такий підхід дозволяє заощадити пам'ять (у листах немає покажчиків на дочірні вузли, в нелистових вузлах немає покажчиків на дані) і за рахунок цього збільшити порядок дерева (мінімальний ступінь)

вузла. Незважаючи на те, що це веде до дублювання ключів, висота дерева зменшується, а значить прискорюються словникові операції над ним. Вузол такої структури повинен мати спеціальне логічне поле x.leaf, яке показує, чи є вузол листом. Крім того, листові вузли зазвичай містять покажчики на наступний/попередній лист для прискорення послідовного доступу.

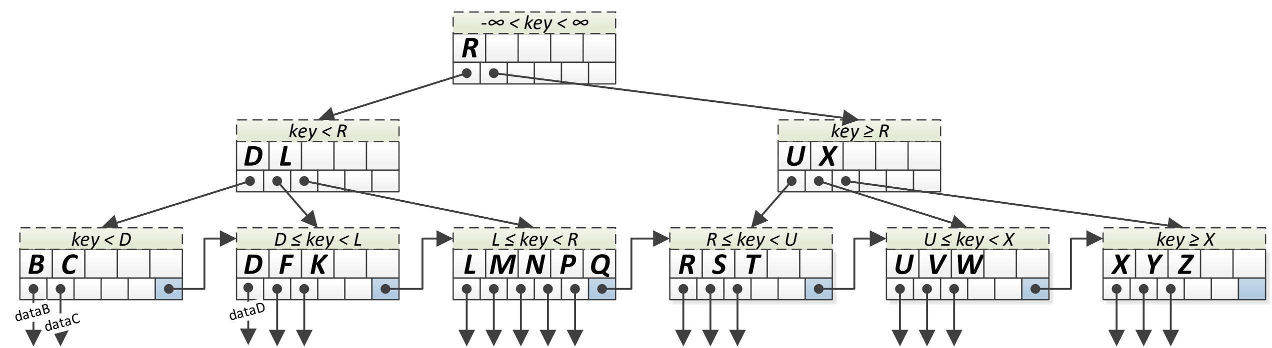


Рисунок 2.5 – Приклад B^{+}-дерева з $t = 3$

B^{*+} - дерево поєднує властивості B^{*} та B^{+} -дерев.

2.5 R-дерева

R-дерево – це деревоподібна структура даних, яка використовується для організації доступу до просторових даних, тобто для індексації багатовимірної інформації, такої, наприклад, як географічні координати, прямокутники або многокутники. R-дерево було запропоноване в 1984 році Антоніном Гуттманном [1] і знайшло значне застосування як у теоретичному, так і у прикладному аспектах [2]. Типовим запитом з використанням R-дерев міг би бути такий: «Знайти всі музеї у радіусі 2 кілометрів від мого поточного місця розташування» або «знайти всі таксі в межах кілометру від мого поточного місця розташування». R-дерево також прискорює пошук найближчого сусіда [3] для різних метрик відстані, включаючи відстань по сфері.

Слід зазначити, що координатами (або вимірами) можуть бути не тільки традиційні декартові x-y, чи широта-довгота, а й дані (частки складного ключу) довільної природи. Наприклад, вік-ціна (автомобіля, людини тощо).

У випадку двовимірного простору, R-дерево розбиває його на множину ієрархічно вкладених прямокутників (рисунок 2.6). Як ми бачимо, ці прямокутники можуть перетинатись.

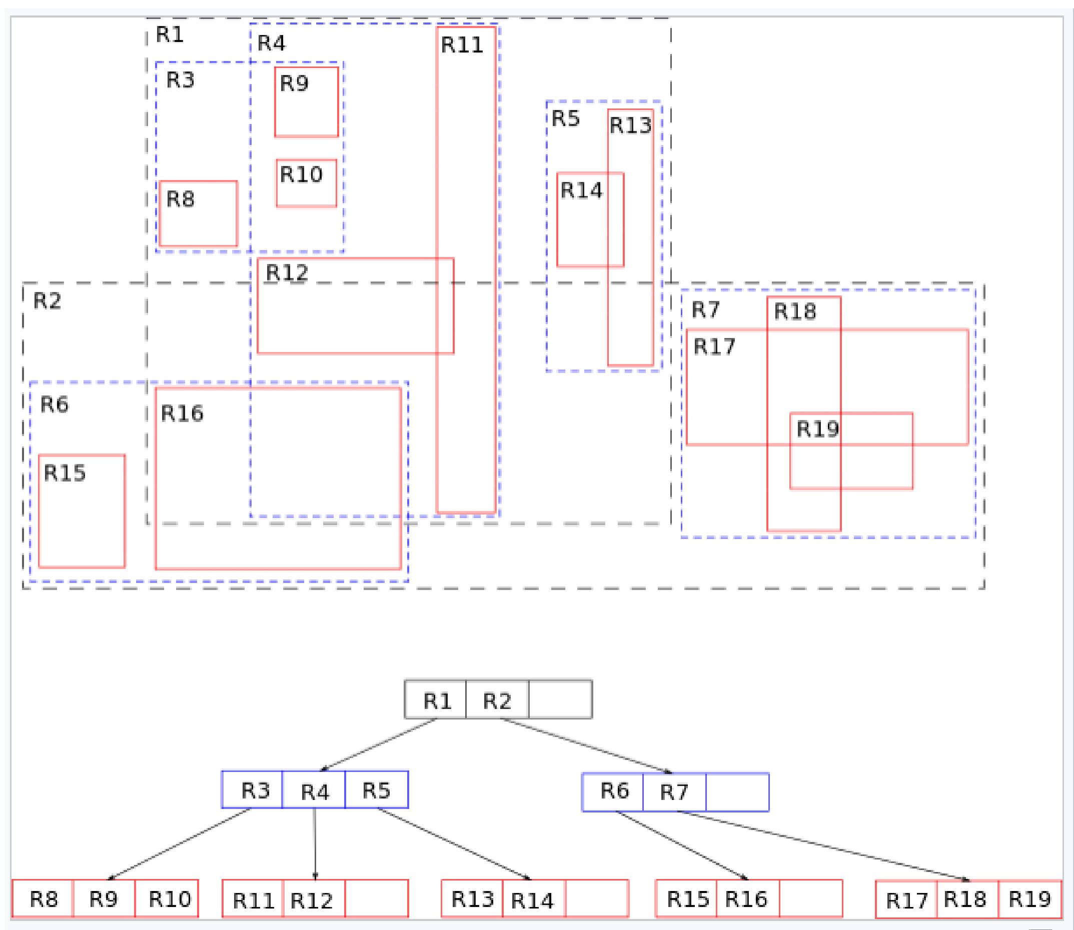


Рисунок 2.6 – Приклад R-дерева

Алгоритми вставки і видалення використовують ці обмежуючі прямокутники для забезпечення того, щоб «близько розташовані» об'єкти були поміщені в одну листову вершину. Зокрема, новий об'єкт потрапить у ту листову вершину, для якої потрібно найменше розширення її

прямокутника. Так само, як і у випадку B^+-дерев, кожен елемент листової вершини зберігає два поля: покажчик на дані, що описують об'єкт, (або самі ці дані) та ключ.

Якщо об’єкти, які зберігаються, є просторовими (на рисунку 2.6 об’єктами є прямокутники, позначені червоним, тобто R8-R19), то таким ключем є координати обмежуючого прямокутника ($x_{min}, x_{max}, y_{min}, y_{max}$).

Якщо ж об’єкти, які зберігаються, є точковими (а цей частковий випадок є дуже поширеним у практичних застосуваннях), то ключем є координати цих точок (тобто, x, y).

Так само, як і у випадку B^+-дерев, вузли R-дерев мають змінну кількість елементів (від `minNumOfEntries` до `maxNumOfEntries`). У кореневій вершині може бути від 1 до maxNumOfEntries нащадків.

Для коректної роботи алгоритмів обробки R-дерев необхідно, щоб $\text{minNumOfEntries} \leq \text{maxNumOfEntries} / 2$. Стандартним вибором є $\text{minNumOfEntries} = t - 1$, $\text{maxNumOfEntries} = 2t - 1$. Значення t визначається співвідношенням файлової сторінки або сторінки оперативної пам’яті з розміром ключів та покажчиків і, зазвичай, становить від 50 до кількох тисяч.

Відмінності між B^+ та R-деревами зумовлені тим фактом, що у останніх ключі (координати обмежуючих прямокутників) не є лінійно впорядкованою множиною. С цього випливають щонайменше два наслідка: поперше, один й той самий шуканий прямокутник (range query), або шукана точка, може належати декільком вузлам R-дерева. Наприклад, на рисунку 2.6, більшість точок R12 або R16 належать одночасно як R1, так і R2. Другим наслідком є те, що ефективність R-дерева дуже суттєво залежить від порядку, за яким воно заповнюється.

Основна ідея побудови R-дерев полягає у групуванні сусідніх точок (або листових об'єктів) так, щоб обмежуючий цю групу прямокутник був найменшим в деякому сенсі. Звідси й походить літера «R» (rectangle) у назві цього дерева. Якщо прямокутник запиту (range query) не перетинає

меж цього контура вузла, то цей запит, також не може перетинати будь-який з об'єктів прямокутника вузла.

На рівні листа, кожен прямокутник описує один об'єкт; на більш високих рівнях агрегації все більше об'єктів. Це також можна розглядати як збільшення грубої апроксимації набору даних.

За аналогією з B та B^+-деревами, R-дерево також є збалансованим деревом пошуку (тобто таким, що всі листя знаходяться на однаковій висоті), організовує дані в сторінках, і призначене для зберігання на диску (в базах даних). Як відомо, B та B^+-дерева гарантують щонайменш 50% заповнення сторінки, а B*-дерева – навіть 66%.

Як і для всіх інших пошукових дерев, алгоритми пошукових операцій для R-дерев (наприклад, перетин, локалізація, пошук найближчого сусіда) досить прості. Ключова ідея полягає в тому, щоб проаналізувавши обмежуючий прямокутник ключа, приймати рішення про те, чи варто шукати всередині відповідного піддерева.

Проте, на відміну від звичайних, тобто одновимірних, дерев (BST, RB, B, AVL тощо), всі запити на пошук у R-дереві, навіть пошук точкового ключа, слід розглядати як групові. Іншими словами, шукані об’єкти можуть одночасно належати багатьом вузлам, бо обмежуючі прямокутники вузлів перетинаються. В цих умовах використовується допоміжна структура даних: черга вузлів (як при послойному обході звичайних BST-дерев).

Функція SearchR(RTree, RangeQuery) працює наступним чином:

а) Кореневий вузол RTree (тобто посилання на нього) заноситься у чергу.

б) Доки черга не пуста,

1) з неї витягується елемент (вузол);

2) в циклі аналізуються всі ключі цього вузла:

– якщо ключ перетинається з шуканим прямокутником RangeQuery

- якщо поточний вузол є листом, то об'єкт, якому відповідає ключ, задовільняє умовам пошуку; посилання на нього заноситься у список відповіді;
- інакше (поточний вузол є внутрішнім) – дочірній вузол цього ключа заноситься у чергу.

Класичний алгоритм вставки, запропонований розробником R-дерев Антоніном Гуттманном, має наступний вигляд:

Функція `insert`:

– викликає `chooseLeaf`, щоб вибрати лист, куди ми хочемо вставити новий елемент (точку, або прямокутник). Якщо вставка здійснена, то вузол, у який вставлено, міг бути поділений. У цьому випадку `chooseLeaf` повертає дві розколоті вершини `splittedNodes` для подальшої вставки в корінь.

– викликається функція `adjustBounds`, яка розширює обмежуючий прямокутник поточного вузла (кореня піддерева) на елемент, що вставляється.

– перевіряє, якщо `chooseLeaf` повернула ненульові `splittedNodes`, то дерево росте на рівень вгору: з цього моменту коренем буде новий вузол, дочірніми елементами якого будуть вузли `splittedNodes`.

Функція `chooseLeaf`:

а) якщо на вході лист (база рекурсії), то:

1) викликає функцію `doInsert`, яка здійснює безпосередню вставку елемента в дерево і повертає два листи, якщо відбулося розділення;

2) змінює обмежуючий прямокутник ключа з урахуванням вставленої точки;

3) повертає `splittedNodes`, які нам повернув `doInsert`;

б) інакше (тобто на вході не листова вершина):

1) з усіх нащадків вибирається той, чиї межі вимагають мінімального збільшення для вставки даної точки;

2) рекурсивно викликається `chooseLeaf` для обраного нащадка;

3) поправляються обмежуючі прямокутники;

4) якщо `splittedNodes` від рекурсивного виклику нульові, то покидаємо функцію, інакше:

5) якщо `numOfEntries < maxNumOfEntries`, то додаємо новий ключ до поточного вузлу та обнуляємо `splittedNodes`;

6) інакше (коли немає місць для вставки), ми конкатенуємо масив ключів з новою вершиною і передаємо його функції `linearSplitNodes` для поділу вузла; повертаємо з `chooseLeaf` ті `splittedNodes`, які нам повернула ця функція поділу

Функція `linearSplit`:

Реалізує один з можливих варіантів поділу вузлів. Цей алгоритм є найбільш простим, має лінійну складність. Він не є оптимальним, однак практика показує, що його ефективності зазвичай достатньо:

а) по кожній координаті для всього набору поділюваних вершин обчислюється різниця між максимальною нижньою межею прямокутника з цієї координаті та мінімальної верхньої, потім ця величина нормалізується на різницю між максимальною і мінімальною координатою точок вихідного набору для побудови всього дерева.

б) знаходиться максимум цього нормалізованого розкиду по всіх координатах;

в) встановлюємо як перших дітей для повертаних вершин `node1` і `node2` ті вершини з вхідного списку, на яких досягався максимум, видаляємо їх з вхідного списку, коригуємо `bounds` для `node1` і `node2`

г) далі, виконується вставка для решти вершин:

1) якщо в списку залишилося настільки мало вершин, що якщо їх все додати в одну з вихідних вершин, то в ній виявиться

minNumOfEntries вершин, то в неї додається залишок, повернення з функції

2) якщо в якійсь з вершин вже набраний максимум нащадків, то залишок додається в протилежну, повернення

3) для чергової вершини зі списку порівнюється, на скільки треба збільшити обмежує прямокутник при вставці в кожну з двох майбутніх вершин, де менше – туди її і вставляється.

Функція фізичної вставки `doInsert`:

– якщо в вершині є вільні місця, то точка вставляється туди;

– якщо ж місць немає, то дочірні вершини вузла конкатенуються з вершиною, що додається, і викликається функція `linearSplit` (або інша аналогічна функція поділу), яка повертає два розділених вузла. Саме їх ми й повертаємо.

Одним з шляхів покращення розбиття простору є використання алгоритмів кластеризації. Отриманий варіант дерев має назву cR-дерево («c» означає clustered). В цьому варіанті для розділення вузлів використовуються алгоритми кластеризації, такі як k-means. В більшості випадків такий підхід забезпечує кращий результат, ніж класичний лінійний алгоритм поділу Гуттмана, оскільки k-means не тільки мінімізує сумарну площу обмежуючих прямокутників, але й відстань між ними та площу перекриття. Проблемою при застосуванні k-means (або інших) є те, що алгоритми кластеризації, зазвичай, не враховують таку умову, що ємність кластеру має бути обмежена зверху і знизу константами `maxNumOfEntries` та `minNumOfEntries`.

У найгіршому випадку ефективність R-дерев є катастрофічно низькою: складність як вставки, так і пошуку сягає $O(n)$. Проте, у більшості випадків, тобто «у середньому» складність цих операцій є логарифмічною ($O(\log n)$), що є цілком придатним.

Підбиваючи підсумок аналізу R-дерев, слід зазначити, що вони:

– ефективно зберігають локалізовані в просторі групи об'єктів;

– збалансовані, що мінімізує кількість вузлів, які переглядаються;

– індексування є динамічним: вставка/видалення елементу не вимагає істотної перебудови всього дерева.

Головними недоліками R-дерев є їхня чутливість до порядку додавання даних та до відповідності структури прямокутників дерева прямокутникам запитів, а також проблема перекриття вузлів.

3 ЗАСТОСУВАННЯ ФРАКТАЛЬНОЇ КРИВОЇ ГІЛБЕРТА В R-ДЕРЕВАХ

3.1 Вплив порядку сканування 2D простору на ефективність R-дерев

Як було зазначено під час дослідження структури R-дерев, продуктивність словникових операцій над ними дуже сильно залежить від конфігурації обмежуючих прямокутників, а вона в свою чергу залежить від порядку додавання об'єктів (точок). Типовою є ситуація, коли вони є відсортованими по координатам. В роботі Руссопулоса і Ляйфкера [1] наведено приклад «упакованого по x R-дерева» (рисунок 3.1). 200 точок, рівномірно розподілених на квадраті 100x100, ієрархічно впорядковані: спочатку по x, а при рівних x – по y. Результат групування цих точок у прямокутники показаний на рисунку 3.1. Вищі рівні дерева будуть створюватись аналогічно.

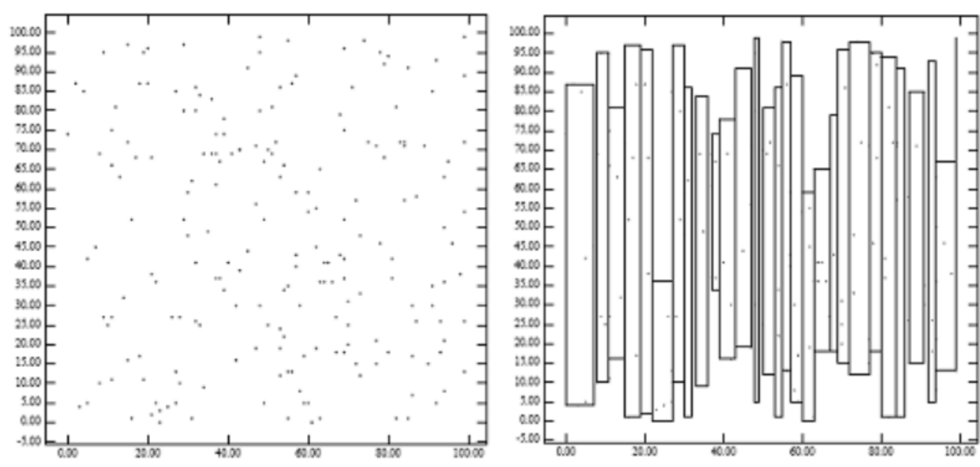


Рисунок 3.1 – Приклад R-дерева, упакованого по x

Таке групування слід розглядати як вкрай неефективне. Це зумовлене тим, що типовий запит є «квадратним», тобто розміри вікна запиту за x та y-координатами є схожими між собою. У таких умовах вікно запиту перетинає дуже багато прямокутників (вузлів дерева), що й є причиною

низької ефективності.

Тут ми стикаємось з таким самим ефектом, що й у випадку швидкого сортування `quickSort`, коли впорядкованість вхідних даних призводить до деградації швидкісних характеристик алгоритму.

Найпростішим виходом з такої ситуації є рандомізація послідовності вхідних даних, проте вона можлива лише для статичних структур даних (у даному випадку R-дерев), та й то не завжди, бо впорядкованість даних може бути зумовлена алгоритмом роботи апаратури читання (наприклад, сканера), а рандомізація стикається з обмеженістю пам’яті.

Для того, щоб мінімізувати кількість перетинів, прямокутники повинні бути якомога близькими між собою за розмірами та якомога квадратнішими.

Більш прогресивним методом боротьби з лінійним виродженням R-дерев є обхід площини у порядку, який визначається фрактальною кривою. Ця крива має бути такою, що заповнює простір. Це може бути крива Пеано, крива Гільберта, або крива Мортона (Z-крива).

Крива Гільберта на решітці 2x2 (початкова крива), що має позначення H1, показана на рисунку 3.2. Щоб отримати криву порядку i, кожна вершина основної кривої замінюється на криву порядку $i - 1$, повернутою та/або відбитою. На рисунку 3.2 показані також криві Гільберта порядків 2 та 3.

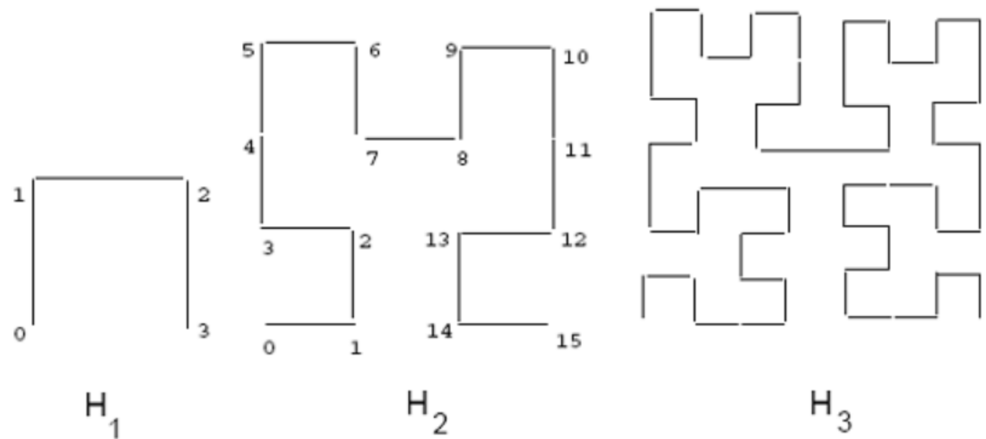


Рисунок 3.2 – Криві Гільберта порядків 1, 2 і 3

При прагненні порядку кривої до нескінченності, крива (предфрактал) перетворюється на фрактал з фрактальною розмірністю два. Можна зазначити, що криву Гільберта можна узагальнити на більш високі розмірності.

Створення кривої Гільберта можна описати у рамках L-системи. Алфавітом є множина $\{A, B, F, +, -\}$, аксіомою (фігурою H_0, яка є точкою) – «A», правила мають вигляд $A \rightarrow -BF + AFA + FB-$ та $B \rightarrow +AF - BFB - FA +$. Символ F позначає «рух вперед», $\pm$ – це повороти на 90° відповідно праворуч та ліворуч, символи A і B є такими, що не відображаються графічно.

Крива Гільберта задає лінійний порядок на множині точок площини, при якому кожній точці співставлюється її номер (відстань до початку координат вздовж кривої). Наприклад (рисунок 3.2), для решітки 4x4 точка (0,0) має номер 0, а точка (1,1) – номер 2.

Легко бачити, що об’єкти (точки), які близькі між собою за гільбертовою відстанню, є досить близькими й в звичайному, геометричному, сенсі. Завдяки цій властивості після групування обмежуючі прямокутники вузлів R-дерева з великою ймовірністю будуть близькими до квадратів прямокутниками, тобто матимуть малі площі та периметри. Невеликі значення площ та периметрів призводять до хорошої продуктивності запитів за рахунок зменшення надлишкових перетинів прямокутників запитів з прямокутниками вузлів.

Слід зазначити, що зворотнє твердження щодо близькості не завжди вірно: якщо деякі точки (x_1, y_1) та (x_2, y_2) геометрично близькі між собою, то різниця між їх гільбертовими відстанями d_1 та d_2 може бути значною.

Таким чином, першим кроком застосування Гільбертової кривої у задачі обробки R-дерев є реалізація перетворень між координатами точки та її Гільбертовою відстанню (в обох напрямках, як з (x, y) в d, так і з d в (x, y)).

Передбачається, що площина має розмір $n \times n$, де n є ступенем двійки.

Слід зауважити, що орієнтація координатних вісей є геометричною (правосторонньою), а не матричною, тобто початок координат (0,0) відповідає лівому нижньому куту квадрата, а точка (n - 1, n - 1) – правому верхньому куту. Відстань d вимірюється від початку координат ($d = 0$) та приймає максимальне значення $d = n^2 - 1$ в правому нижньому куті (лістинг 3.1).

Лістинг 3.1 – Перетворення між координатами та Гільбертовою відстанню

```
//Перетворення (x,y) у d
int xy2d (int n, int x, int y) {
    int rx, ry, s, d=0;
    for (s=n/2; s>0; s/=2) {
        rx = (x & s) > 0;
        ry = (y & s) > 0;
        d += s * s * ((3 * rx) ^ ry);
        rot(s, &x, &y, rx, ry);
    }
    return d;
}
//Перетворення d у (x,y)
void d2xy(int n, int d, int *x, int *y) {
    int rx, ry, s, t=d;
    *x = *y = 0;
    for (s=1; s<n; s*=2) {
        rx = 1 & (t/2);
        ry = 1 & (t ^ rx);
        rot(s, x, y, rx, ry);
        *x += s * rx;
        *y += s * ry;
        t /= 4;
    }
}
//обертання/віддзеркалення квадранту
void rot(int n, int *x, int *y, int rx, int ry) {
    if (ry == 0) {
        if (rx == 1) {
            *x = n-1 - *x;
            *y = n-1 - *y;
        }
        //Обмін x і y місцями
        int t  = *x;
        *x = *y;
        *y = t;
    }
}
```

Обидві функції (`xy2d` та `d2xy`) розглядають весь квадрат як 4 області розміром 2×2. Далі кожна з цих областей також розглядається як чотири більш дрібніші області і так далі. На рівні s кожна область має розмір $s \times s$. Ці рівні проходяться у циклі `for`. При цьому функція `xy2d` оброблює квадрат зверху до низу, тобто починає з старших бітів x та y, та, відповідно, формує d починаючи з його старших бітів. Інакше кажучи, починає з верхнього рівня розбиття (`s=n/2`). Функція `d2xy` працює у проилежному напрямку: починає з молодших бітів d, та будує x і y починаючи з їх молодших бітів, тобто стартує з `s=1`.

На кожній ітерації додається біт до d, або до x та y, який визначається областю (з чотирьох можливих), в якій знаходиться оброблювана точка. Області задаються парою бітів (rx, ry), тобто rx та ry приймають значення 0 або 1. Для кожної з чотирьох областей пара вихідних бітів однозначно визначається по парі вхідних. Обидві функції використовують допоміжну функцію `rot`, яка виконує обертання/віддзеркалення системи координат, що необхідно для переходу до наступної ітерації (наступного значення змінної s).

3.2 Структура R-дерева Гілберта

За визначенням, Гільбертова відстань прямокутника дорівнює гільбертовій відстані його центра.

Відмінністю структури R-дерева Гільберта від звичайного R-дерева є наявність у вузлах окрім поля `MBR` (координати мінімального обмежуючого прямокутника $x_{low}, x_{high}, y_{low}, y_{high}$), ще й поля `LHV` (Largest Hilbert Value) – найбільша Гільбертова відстань серед дочірніх елементів цього вузла.

При цьому, оскільки вузли мають в якості свого LHV гільбертову відстань одного зі своїх нащадків, то немає додаткової ціни для обчислення гільбертових відстаней MBR вузлів рівнів вище, ніж перший.

Рисунок 3.3 ілюструє організацію прямокутників у R-дерево

Гільберта. Первинні дані (на рисунку не показані) є точковими та згруповані у вузли першого рівня (дев'ять малих прямокутників на рисунку 3.3). Вузли першого рівня згруповані у вузли другого рівня (прямокутники, позначені червоним, зеленим та синім кольорами). На рисунку 3.3 у квадратних дужках наведені LHV вузлів другого рівня. Крім того, для зеленого вузла наведені LHV та MBR його дочірніх вузлів.

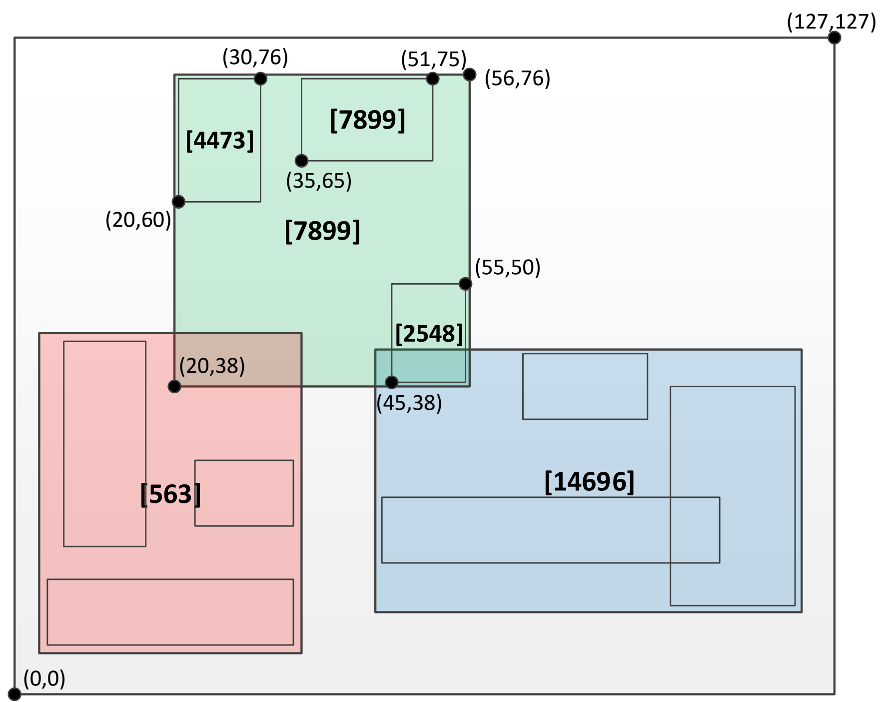


Рисунок 3.3 – Прямокутники, організовані в R-дерево Гільберта

Згідно з рисунком 3.3, перший (червоний) вузол другого рівня має $\mathrm{LHV} = 563$, другий (зелений) – $\mathrm{LHV} = 7899$, а третій (синій) – $\mathrm{LHV} = 14696$. З цього випливає, що будь-який червоний дочірній вузол має $\mathrm{LHV} \leq 563$, зелений – $563 < \mathrm{LHV} \leq 7899$, синій – $\mathrm{LHV} > 7899$. Якщо виникне потреба додати у дерево дані, тобто нову точку, то її буде додано до того з вузлів, який є найближчим за значенням LHV-відстані. Звичайно, після цього значення LHV та MBR вузла мають бути оновлені.

На рисунку 3.4 показано структуру зберігання R-дерева з рисунка 3.3 у пам'яті комп'ютера.

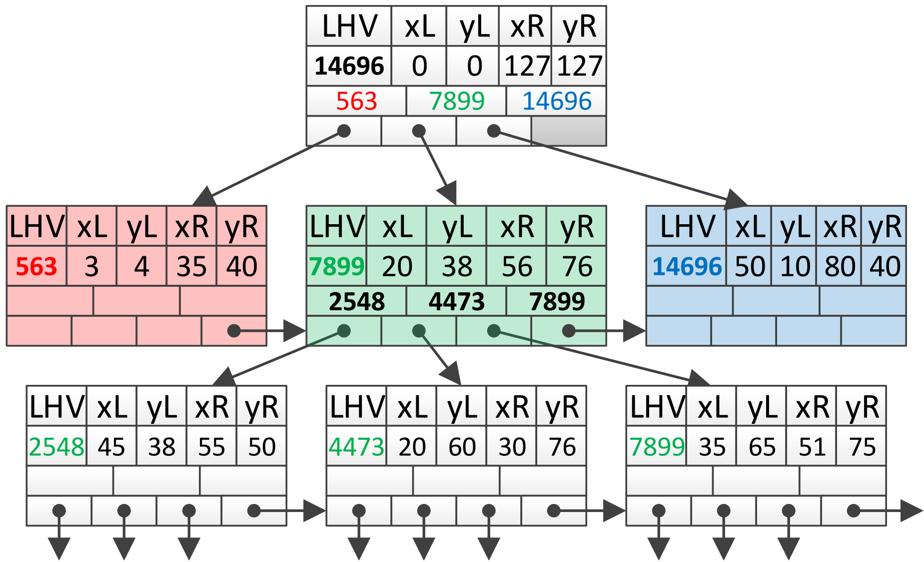


Рисунок 3.4 – Структура зберігання R-дерева у пам'яті

Пошук даних у R-дереві Гільберта нічим не відрізняється від алгоритму пошуку у звичайному R-дереві, тобто гільбертова відстань (поле LHV) не використовується. Пошук починається з кореня (тобто корень кладеться в чергу). Далі у циклі зчитується елемент з голови черги, ті з його дочірніх, що перетинаються з вікном запиту, додаються у чергу. Цикл закінчується коли черга вичерпана, або коли її головою є листовий елемент дерева. У останньому випадку черга буде містити ті й тільки ті листи (точки), які містяться в вікні запиту.

Для вставки нового прямокутника (з габаритами `mbr`) в R-дерево Гільберта обчислюється гільбертова відстань `lhv` центру цього прямокутника. Це значення потім використовується як ключ. У разі, якщо до дерева додається точка, то вона розглядається як частковий випадок прямокутника (нульового розміру, тобто $xL = xR, yR = yR$). R-дерево проглядається починаючи з кореня. Значення `MBR` та `LHV` поточного вузла оновлюються з врахуванням `mbr` та `lhv` елемента, який вставляється. Серед ключів поточного вузла обирається вузол з мінімальним значенням `LHV`, таким що перевищує `lhv`. Здійснюється перехід до відповідного дочірнього

вузла, до якого й буде додано новий прямокутник.

У прикладі (рисунок 3.3 – 3.4) будь-який новий елемент з $563 < lhv \leq 7899$ буде додаватись до другого (тобто зеленого) вузла.

При додаванні елементу до вузла R-дерева можливе переповнення вузла. Переповнення призводить до розбиття вузлів. Найчастіше використовується політика розбиття 1-в-2. Інколи, подібно до B*-дерев, використовується розбиття 2-в-3 (тобто переповнений вузол об'єднують з сусіднім та розбивають їхню сукупність на три вузли). Загалом можна говорити про політику розбиття s-в-(s+1), де s є порядком політики розбиття. Втім, у R-деревах майже завжди використовується політика розбиття 1-в-2.

Обробка переповнення може проводитись знизу вверх (за виникненням факту переповнення), або зверху вниз (у попереджувальному режимі). Перший спосіб більш простий у програмній реалізації, веде до більш економного використання пам'яті, але суттєво повільніший. Його доцільно використовувати у разі, якщо дані, що зберігаються, є статичними. Відповідно, переваги та недоліки другого способу віддзеркалюють властивості першого. Цей спосіб використовується для створення динамічних R-дерев.

Видалення елементу з R-дерева призводить до «усихання» вузлів та до їх злиття. Процедура видалення елементів у гільбертовому R-дереві відрізняється від процедури видалення у звичайному R-дереві тільки необхідністю корегувати LHV-ключі. Злиття вузлів також може проводитись як знизу вверх (за виникненням потреби злиття), так й зверху вниз (попереджувальне злиття). Для статичних R-дерев слід використовувати перший спосіб (хоч взагалі-то, для статичних дерев операція видалення не є характерною). Якщо ж дерево є динамічним, то перевагу має попереджувальне злиття.

3.3 Порівняння статичних R-дерев з R-деревами Гілберта

У роботі порівнювались статичні R-дерева Гільберта з звичайними (та також статичними) R-деревами. Порівняння відбувалось за швидкодією пошуку та додавання даних.

Мовою реалізації було обрано C#. Цей вибір обгрунтовується тим, що в роботі досліджувалась ефективність структур даних саме на низькорівневому рівні їхньої організації.

На фізичному рівні вузол є структурою даних, яка складається з заголовку та двох масивів: $(m-1)$-елементного масиву ключів та m-елементного масиву покажчиків на відповідні дочірні вузли. Кожен елемент масиву ключів є структурою з полями `LHV, xL, yL, xR, yR` дочірнього вузла (рисунок 3.5). Заголовок містить поля `LHV, xL, yL, xR, yR` поточного вузла, покажчик на «братерський» вузол (тобто сусідній справа вузол того ж рівня, що й поточний), фактичну кількість ключів (`num`), що зберігаються у поточному вузлі, ознаку (бульову змінну) `isLeave` того, чи є поточний ключ листом, та (для цілей відлагодження) – змінну-лічильник кількості звертань до вузла (`numRead`). Якщо так, то покажчики вказують не на дочірні вузли R-дерева, а безпосередньо на корисні дані, асоційовані з прямокутником (точкою).

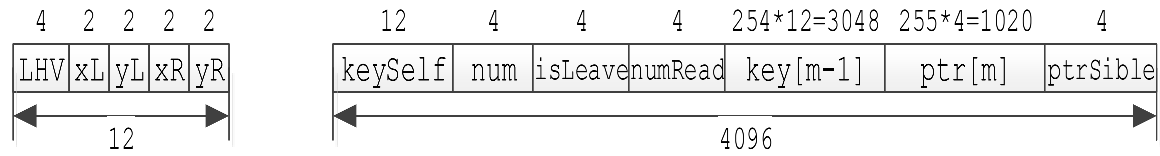


Рисунок 3.5 – Структура вузла R-дерева Гільберта

Значення m обрано рівним 255, таким чином вузол має розмір рівно 4096=4Кбайт (тобто одна сторінка пам'яти).

Як було зазначено у попередньому розділі, фактична кількість ключів

може варіюватись від $m/2$ до m, тобто $127 \leq num \leq 254$.

Розмір карти (прямокутної досліджуваної області) складав 60 000×60 000 пікселів. Кількість точок (листів дерева) становила 1 440 000.

Досліджувалась кількість вузлів, що переглядаються, в залежності від кількості елементів у пошуковому запиті. Результати дослідження представлені в таблиці 3.1, а відповідна графічна візуалізація – на рисунку 3.6.

Цей показник є головною мірою ефективності досліджуваних структур даних, оскільки саме звертання до вузлів (читання секторів з жорсткого диску комп'ютера) є найбільш затратною операцією.

Як можна бачити з наведених результатів, застосування фрактального (гільбертового) обходу площини дає змогу суттєво зменшити кількість вузлів, які проглядаються.

Таблиця 3.1 – Порівняння звичайного та гільбертового R-дерев за кількістю вузлів, які переглядаються під час запиту

Розмір вікна запиту	Середня кількість точок в запиті	Кількість вузлів, що переглядаються	
		R-tree	Hilbert-R
50×50	1	1.830	1.216
158×158	10	2.644	1.811
500×500	100	7.051	3.244
$1\,581 \times 1\,581$	1 000	22.138	11.423
$5\,000 \times 5\,000$	10 000	145.962	62.180
$15\,811 \times 15\,811$	100 000	1112.125	413.298
$50\,000 \times 50\,000$	1 000 000	10468.647	3394.250

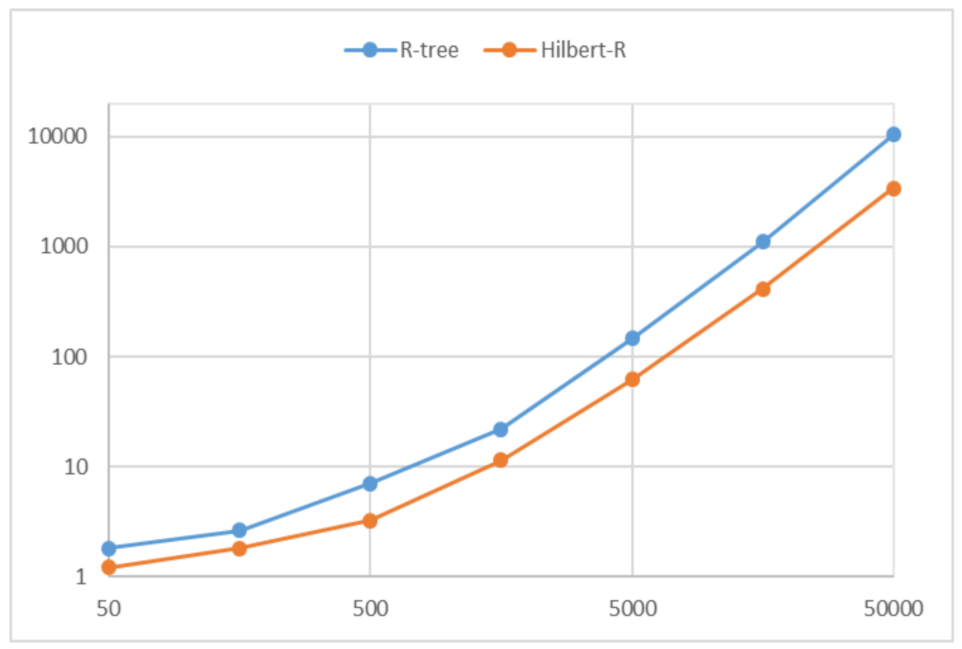


Рисунок 3.6 – Залежність кількості вузлів, які переглядаються, від розміру вікна запиту

Важливим параметром, який характеризує ефективність структур даних, є часова складність їх модифікації, тобто додавання та видалення листів. Відповідні дослідження також були проведені (таблиця.3.2): порівнювалась кількість часу, необхідна для додавання у R-дерево нових елементів (точок).

Таблиця 3.2 – Питомий час на додавання елементу у R- та HR-дерева

Кількість елементів, що додаються	Питомий час на додавання одного елементу, мкс.	
	R-дерево	Hilbert-R
1 000	16.5	22.2
10 000	18.2	24.1
100 000	23.3	28.4
1 000 000	26.7	32.3

Можна зазначити, що додавання даних до гільбертового R-дерева відбувається трохи повільніше, ніж для звичайного, що під час роботи алгоритму зумовлюється необхідністю обрахування гільбертової відстані для точок, які додаються.

Таким чином для підвищення ефективності та точності структури даних потрібно враховувати додаткові фактори, які можуть повплливати на складові в модифікації алгоримів та специфікацій фрактальних кривих у R- та HR-деревах.

ВИСНОВКИ

В роботі досліджувались методи індексування та «упаковки» 2D даних. Було зазначено, що найбільш придатною структурою таких даних є R-дерева. Вони за структурою та алгоритмами словникових операцій (пошук, додавання, видалення) дуже схожі на B+ дерева, які застосовуються для організації одновимірних даних. Проведено аналіз структури B, B+ та R-дерев, алгоритмів словникових операцій над ними.

Визначено, що ключі 2D даних (габарити обмежуючого прямокутника) не є лінійно впорядкованою множиною. Це призводить до того, що прямокутні області, які відповідають вузлам, зазвичай, перетинаються між собою. Більше того, конфігурація цих областей суттєво залежить від порядку додавання первинних елементів (точок) у дерево та значно впливає на ефективність пошуку даних.

У найбільш поширеному випадку дані є ієрархічно впорядкованими. Це призводить до того, що прямокутники вузлів є вузькими витягнутими смугами. Враховуючи, що вікна запиту, зазвичай, збалансовані (близьки до квадратних), пошук потребує звертання до багатьох вузлів.

Для подолання цієї проблеми запропоновано впорядковувати точки площини за допомогою кривої, що покриває простір. Такі криві є фрактальними. Однією з найбільш відомих серед них є крива Гільберта. У гільбертових R-деревах ключі у вузлах впорядковані за гільбертовим номером прямокутника. Ця гільбертова відстань засосовується під час модифікації дерева (додавання чи видалення елементів).

В роботі здійснено програмну реалізацію R-дерев та типових операцій над ними. Проведено порівняльний аналіз впливу методу обходу площини, зокрема за кривою Гільберта, на ефективність індексування. Визначено, що засосування гільбертового обходу дозволяє суттєво прискорити пошук даних за рахунок зменшення кількості вузлів, які проглядаються.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Мандельброт, Б. Фрактальна геометрія природи: пер. с англ. – М.: Институт компьютерных исследований, 2002.–656с
2. Шредер, М. Фрактали, хаос, статечні закони. Мініатюри з нескінченного раю: пер. с англ. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001. –528с.
3. Кроновер, Р. Фрактали та хаос у динамічних системах. Основи теорії: пер. с англ. – М.: Постмаркет, 2000. –352с.
4. Шустер, Г. Детермінований хаос. Вступ: пер. с нім. – М.:Мир, 1990, –254с.
5. Федер, Е. Фрактали: пер. с англ. –М.: Мир, 1991, –254с.
6. Кормен, Т. Алгоритми: побудова та аналіз. / Т. Кормен, Ч.Лейзерсон, Р.Ривест - М.: МЦНМО, 2002. –960 с.
7. Goodrich M.T, Tamassia R. Algorithm Design and Applications. John Wiley & Sons, New York, 2015
8. Ахо А., Хопкрофт Дж., Ульман Д. Структури даних та алгоритми. - М.: Издательский дом «Вильямс», 2003. – 384 с.
9. Goodrich M.T, Tamassia R., Mount D. Data Structures and Algorithms in C++. John Wiley & Sons, New York, 2011.
10. Goodrich M.T, Tamassia R., Goldwasser M. Data Structures and Algorithms in Java. John Wiley & Sons, New York, 2014.
11. Roussopoulos, N., Leifker D. (1985) Direct spatial search on pictorial databases using Packed R-trees. In Proc. of ACM SIGMOD, pages 17–31, Austin, TX, May 1985.
12. Schubert, E., Zimek, A., Kriegel, H. P. (2013). «Geodetic Distance Queries on R-Trees for Indexing Geographic Data». Advances in Spatial and Temporal Databases. Lecture Notes in Computer Science. Vol. 8098. p. 146

ДОДАТОК А

Тексти програм

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
namespace ConsoleApp3
{ public class Node<TK>
{ private int degree;
public Node(int degree, int pageSize, int entrySize)
//sizeof(int)
{ this.degree = degree;
Int overallSize = entrySize * degree;
var pageCounts = (overallSize / entrySize) + (overallSize %
entrySize == 0 ? 0 : 1);
this.Children = List<Node<TK>>(pageCounts * degree);
new this.Entries = new List<Entry<TK>>(pageCounts * degree);
}
public List<Node<TK>> Children { get; set; }
public List<Entry<TK>> Entries { get; set; }
public bool IsLeaf
{ get
{
return
this.Children.Count
== 0;
}
} public bool HasReachedMaxEntries
{ get { return
this.Entries.Count == (2 * this.degree) - 1;
}
} public bool HasReachedMinEntries
{ get { return
this.Entries.Count == this.degree - 1;
}
}
} public class Entry<TK> : IEquatable<Entry<TK>>
{
public TK Key { get; set; }
public bool Equals(Entry<TK> other)
{
return this.Key.Equals(other.Key);
}
} public class BTree<TK> where TK : IComparable<TK>
{ private int pageSize = 16_384; private
int entrySize = sizeof(int);
public BTree(int degree)
{ if (degree < 2)
{ throw new ArgumentException("BTree degree must be
at least 2", "degree");
}
this.Root = new Node<TK>(degree, pageSize, entrySize);
```

```
this.Degree = degree; this.Height = 1;
} public Node<TK> Root { get; private set; }
public int Degree { get; private set; }
public int Height { get; private set; } public
List<TK> NumbersInTree = new
List<TK>();
public Entry<TK> Search(TK key)
{ return this.SearchInternal(this.Root, key); }
public void Insert(TK newKey)
{ if (!this.Root.HasReachedMaxEntries)
{ this.InsertNonFull(this.Root, newKey);
return;
}
Node<TK> oldRoot = this.Root; this.Root = new
Node<TK>(this.Degree, pageSize, entrySize);
this.Root.Children.Add(oldRoot);
this.SplitChild(this.Root, 0, oldRoot);
this.InsertNonFull(this.Root, newKey);
this.Height++;
} public void Delete(TK keyToDelete)
{
this.DeleteInternal(this.Root, keyToDelete);
if (this.Root.Entries.Count == 0 &&
!this.Root.IsLeaf)
{
this.Root =
this.Root.Children.Single();
this.Height--;
}
} private void DeleteInternal(Node<TK> node, TK
keyToDelete) { int i =
node.Entries.TakeWhile(entry =>
keyToDelete.CompareTo(entry.Key) >
0).Count();
if (i < node.Entries.Count &&
node.Entries[i].Key.CompareTo(keyToDelete) == 0)
{ this.DeleteKeyFromNode(node,
keyToDelete, i); return;
}
if (!node.IsLeaf)
{ this.DeleteKeyFromSubtree(node,
keyToDelete, i);
}
} private void DeleteKeyFromSubtree(Node<TK> parentNode, TK
keyToDelete, int subtreeIndexInNode)
{
Node<TK> childNode =
parentNode.Children[subtreeIndexInNode];
if (childNode.HasReachedMinEntries)
{ int leftIndex = subtreeIndexInNode -
1;
Node<TK> leftSibling =
subtreeIndexInNode > 0 ?
```

```
parentNode.Children[leftIndex] : null;
int rightIndex = subtreeIndexInNode + 1;
Node<TK> rightSibling = subtreeIndexInNode <
parentNode.Children.Count - 1
?
parentNode.Children[rightIndex]
:
null;
if (leftSibling != null &&
leftSibling.Entries.Count > this.Degree - 1)
{
childNode.Entries.Insert(0,
parentNode.Entries[subtreeIndexInNode]);
parentNode.Entries[subtreeIndexInNode] =
leftSibling.Entries.Last();
leftSibling.Entries.RemoveAt(leftSibling.Entries.Coun t - 1);
if (!leftSibling.IsLeaf)
{ childNode.Children.Insert(0,
leftSibling.Children.Last());
leftSibling.Children.RemoveAt(leftSibling.Children.Co unt -
1);
} } else if (rightSibling != null &&
rightSibling.Entries.Count > this.Degree - 1)
{
childNode.Entries.Add(parentNode.Entries[subtreeIndex
InNode]);
parentNode.Entries[subtreeIndexInNode] =
rightSibling.Entries.First();
rightSibling.Entries.RemoveAt(0);
if (!rightSibling.IsLeaf) {
childNode.Children.Add(rightSibling.Children.First())
;
rightSibling.Children.RemoveAt(0);
} } else
{ if (leftSibling != null)
{ childNode.Entries.Insert(0,
parentNode.Entries[subtreeIndexInNode]);
var oldEntries childNode.Entries; =
childNode.Entries leftSibling.Entries;
childNode.Entries.AddRange(oldEntries);
if (!leftSibling.IsLeaf)
{
=
var oldChildren childNode.Children; =
childNode.Children =
leftSibling.Children;
childNode.Children.AddRange(oldChildren);
} parentNode.Children.RemoveAt(leftIndex);
parentNode.Entries.RemoveAt(subtreeIndexInNode);
} else
{
Debug.Assert(rightSibling != null, "Node should have at least
one
```

```
sibling");
childNode.Entries.Add(parentNode.Entries[subtreeIndex
InNode]);
childNode.Entries.AddRange(rightSibling.Entries);
if (!rightSibling.IsLeaf)
{
childNode.Children.AddRange(rightSibling.Children);
}
parentNode.Children.RemoveAt(rightIndex);
parentNode.Entries.RemoveAt(subtreeIndexInNode);
}
}
}
this.DeleteInternal(childNode, keyToDelete);
}
private void DeleteKeyFromNode(Node<TK> node,
TK keyToDelete, int keyIndexInNode)
{ if (node.IsLeaf)
{
node.Entries.RemoveAt(keyIndexInNode);
return;
}
Node<TK> predecessorChild =
node.Children[keyIndexInNode]; if
(predecessorChild.Entries.Count >= this.Degree) {
Entry<TK> predecessor
this.DeletePredecessor(predecessorChild); =
node.Entries[keyIndexInNode] predecessor; }
else {
=
Node<TK> successorChild =
node.Children[keyIndexInNode + 1];
if (successorChild.Entries.Count >= this.Degree) {
Entry<TK> successor this.DeleteSuccessor(predecessorChild); =
node.Entries[keyIndexInNode] =
successor; } else {
predecessorChild.Entries.Add(node.Entries[keyIndexInN ode]);
predecessorChild.Entries.AddRange(successorChild.Entr ies);
predecessorChild.Children.AddRange(successorChild.Chi ldren);
node.Entries.RemoveAt(keyIndexInNode);
node.Children.RemoveAt(keyIndexInNode + 1);
this.DeleteInternal(predecessorChild, keyToDelete);
}
}
} private Entry<TK> DeletePredecessor(Node<TK> node)
{ if (node.IsLeaf)
{
var result =
node.Entries[node.Entries.Count - 1];
node.Entries.RemoveAt(node.Entries.Count - 1);
return result;
}
return this.DeletePredecessor(node.Children.Last());
```

```
} private Entry<TK> DeleteSuccessor(Node<TK> node)
{ if (node.IsLeaf)
{ var result = node.Entries[0];
node.Entries.RemoveAt(0); return result;
}
return this.DeletePredecessor(node.Children.First());
} private Entry<TK> SearchInternal(Node<TK> node, TK key)
{ int i = node.Entries.TakeWhile(entry =>
key.CompareTo(entry.Key) > 0).Count();
if (i < node.Entries.Count &&
node.Entries[i].Key.CompareTo(key) == 0)
{ return node.Entries[i];
}
return node.IsLeaf ? null :
this.SearchInternal(node.Children[i], key);
} private void SplitChild(Node<TK> parentNode, int
nodeToBeSplitIndex, Node<TK> nodeToBeSplit)
{ var newNode = new Node<TK>(this.Degree, pageSize,
entrySize);
parentNode.Entries.Insert(nodeToBeSplitIndex,
nodeToBeSplit.Entries[this.Degree - 1]);
parentNode.Children.Insert(nodeToBeSplitIndex + 1, newNode);
newNode.Entries.AddRange(nodeToBeSplit.Entries.GetRan
ge(this.Degree, this.Degree - 1));
nodeToBeSplit.Entries.RemoveRange(this.Degree - 1,
this.Degree);
if (!nodeToBeSplit.IsLeaf)
{
newNode.Children.AddRange(nodeToBeSplit.Children.GetR
ange(this.Degree, this.Degree));
nodeToBeSplit.Children.RemoveRange(this.Degree,
this.Degree); }
} private void InsertNonFull(Node<TK> node, TK newKey)
{
int positionToInsert =
node.Entries.TakeWhile(entry =>
newKey.CompareTo(entry.Key) >= 0).Count();
if (node.IsLeaf)
{ node.Entries.Insert(positionToInsert, new
Entry<TK>() { Key = newKey}); return;
}
Node<TK> child =
node.Children[positionToInsert]; if
(child.HasReachedMaxEntries)
{ this.SplitChild(node,
positionToInsert, child); if
(newKey.CompareTo(node.Entries[positionToInsert].Key)
> 0)
{ positionToInsert++;
}
}
this.InsertNonFull(node.Children[positionToInsert], newKey);
}
```

```
} class
Program
{
static void Main(string[] args)
{ var binaryTree = new BTree<int>(195);
}
}
}
```

Наступними кроками було створення дерева з 1 000 000 випадково згенерованих чисел і визначення висоти побудованого дерева. А також
доданий функціонал виконання 200000 пошуків / вставок / видалень в
довільному порядку і відзначено кількість переглянутих вузлів, кількість
запланованих прочитань / записів на диск. Отримані результати були
стандартизовані за кількістю операцій відповідного типу.
Перелік оновленого коду виглядає наступним чином:

```
using System; using
System.Collections.Generic; using
System.Diagnostics; using System.Linq;
namespace ConsoleApp3
{ public class Node<TK>
 { private int degree;
 public Node(int degree, int pageSize, int
entrySize) //sizeof(int)
 { this.degree = degree; int
overallSize = entrySize * degree;
 var pageCounts = (overallSize /
entrySize) + (overallSize % entrySize == 0 ? 0 : 1);
 this.Children
List<Node<TK>>(pageCounts * degree);
= new
 this.Entries = new
List<Entry<TK>>(pageCounts * degree);
 } public List<Node<TK>> Children { get; set; }
public List<Entry<TK>> Entries { get; set; } public bool
IsLeaf
 { get { return this.Children.Count == 0; }
 } public bool
HasReachedMaxEntries
 { get { return this.Entries.Count == (2 *
this.degree) - 1; }
 } public bool
HasReachedMinEntries
 {
 get { return this.Entries.Count ==
this.degree - 1; }
 }
 }
 public class Entry<TK> : IEquatable<Entry<TK>>
 { public TK Key { get; set; }
```

```
 public bool Equals(Entry<TK> other)
 { return this.Key.Equals(other.Key);
 }
 }
 public class BTree<TK> where TK :
IComparable<TK> { private int pageSize = 16_384;
private int entrySize = sizeof(int);
 public static int InsertCount;
public static int SearchCount; public
static int DeleteCount;
 public static int InsertMemoryCount;
public static int DeleteMemoryCount; public
static int SearchMemoryCount;
 public static int InsertCountAverage;
public static int DeleteCountAverage; public
static int SearchCountAverage;
 public static
InsertMemoryCountAverage;
int
 public static
DeleteMemoryCountAverage;
int
 public static
SearchMemoryCountAverage;
 public BTree(int degree)
 { if (degree <
2)
 {
int
 throw new
ArgumentException("BTree degree must be at least 2",
"degree");
 }
 this.Root = new Node<TK>(degree,
pageSize, entrySize); this.Degree =
degree; this.Height = 1;
 }
 public Node<TK> Root { get; private set; }
 public int Degree { get; private set; }
 public int Height { get; private set; }
 public List<TK> NumbersInTree = new
List<TK>();
 public Entry<TK> Search(TK key, int
operationNumber = 3) { return
this.SearchInternal(this.Root,
key, operationNumber);
 } public void Insert(TK newKey, int
operationNumber = 1) { if
(!this.Root.HasReachedMaxEntries)
 {
this.InsertNonFull(this.Root,
newKey, operationNumber);
return;
```

```
 }
 Node<TK> oldRoot = this.Root; this.Root
= new Node<TK>(this.Degree,
pageSize, entrySize); this.Root.Children.Add(oldRoot);
this.SplitChild(this.Root, 0,
oldRoot); this.InsertNonFull(this.Root, newKey,
operationNumber);
 this.Height++;
 }
 public void Delete(TK keyToDelete, int
operationNumber = 2) {
this.DeleteInternal(this.Root,
keyToDelete, operationNumber);
 if (this.Root.Entries.Count == 0 &&
!this.Root.IsLeaf)
 {
 this.Root = this.Root.Children.Single();
this.Height--;
 }
 }
 private void DeleteInternal(Node<TK> node, TK
keyToDelete, int operationNumber)
 {
 Incrementer(operationNumber);
IncrementerMemory(operationNumber); var i =
node.Entries.TakeWhile(entry
=> keyToDelete.CompareTo(entry.Key) > 0).Count(); for
(var index = 0; index <=i; index++)
 {

IncrementerMemory(operationNumber);
 }
 if (i < node.Entries.Count &&
node.Entries[i].Key.CompareTo(keyToDelete) == 0)
 {
this.DeleteKeyFromNode(node,
keyToDelete, i, operationNumber);
return;
 }
 if (!node.IsLeaf)
 {
this.DeleteKeyFromSubtree(node,
keyToDelete, i, operationNumber);
 }
 }
 private void
DeleteKeyFromSubtree(Node<TK> parentNode, TK keyToDelete, int
subtreeIndexInNode, int operationNumber)
 {
 Node<TK> childNode =
parentNode.Children[subtreeIndexInNode];
 if (childNode.HasReachedMinEntries)
 { int leftIndex =
```

```
subtreeIndexInNode
- 1;
 Node<TK> leftSibling =
subtreeIndexInNode > 0 ?
parentNode.Children[leftIndex] : null;
 int rightIndex =
subtreeIndexInNode + 1;
 Node<TK> rightSibling =
subtreeIndexInNode < parentNode.Children.Count - 1
? parentNode.Children[rightIndex]
 : null;
 if (leftSibling != null &&
leftSibling.Entries.Count > this.Degree - 1)
 {
childNode.Entries.Insert(0,
parentNode.Entries[subtreeIndexInNode]);

parentNode.Entries[subtreeIndexInNode] =
leftSibling.Entries.Last();

leftSibling.Entries.RemoveAt(leftSibling.Entries.Coun t - 1);
if (!leftSibling.IsLeaf)
 {
 childNode.Children.Insert(0,
leftSibling.Children.Last());

leftSibling.Children.RemoveAt(leftSibling.Children.Co unt -
1);
 }
 }
 else if (rightSibling != null &&
rightSibling.Entries.Count > this.Degree - 1)
 {

childNode.Entries.Add(parentNode.Entries[subtreeIndex
InNode]);

parentNode.Entries[subtreeIndexInNode] =
rightSibling.Entries.First();
 rightSibling.Entries.RemoveAt(0);
 if (!rightSibling.IsLeaf)
 {

childNode.Children.Add(rightSibling.Children.First()) ;
 rightSibling.Children.RemoveAt(0);
 } }
else { if
(leftSibling != null)
 {
 childNode.Entries.Insert(0,
parentNode.Entries[subtreeIndexInNode]); var
oldEntries = childNode.Entries;
 childNode.Entries =
```

```
leftSibling.Entries;

childNode.Entries.AddRange(oldEntries); if
(!leftSibling.IsLeaf)
 { var
oldChildren =
childNode.Children; childNode.Children =
leftSibling.Children;

childNode.Children.AddRange(oldChildren);
 }

parentNode.Children.RemoveAt(leftIndex);

parentNode.Entries.RemoveAt(subtreeIndexInNode);
 }
else
 {
 Debug.Assert(rightSibling != null,
"Node should have at least one sibling");

childNode.Entries.Add(parentNode.Entries[subtreeIndex
InNode]);

childNode.Entries.AddRange(rightSibling.Entries);
if (!rightSibling.IsLeaf)
 {
childNode.Children.AddRange(rightSibling.Children);
 }

parentNode.Children.RemoveAt(rightIndex);

parentNode.Entries.RemoveAt(subtreeIndexInNode);
 }
 }
 }
 this.DeleteInternal(childNode,
keyToDelete, operationNumber);
 } private void DeleteKeyFromNode(Node<TK>
node, TK keyToDelete, int keyIndexInNode, int operationNumber)
{ if (node.IsLeaf)
 {

node.Entries.RemoveAt(keyIndexInNode);
return;
 }
 Node<TK> predecessorChild =
node.Children[keyIndexInNode];
if (predecessorChild.Entries.Count >=
this.Degree)
 {
 Entry<TK> predecessor =
this.DeletePredecessor(predecessorChild, operationNumber);
```

```
node.Entries[keyIndexInNode] =
predecessor;
} else
 {
 Node<TK> successorChild =
node.Children[keyIndexInNode + 1]; if
(successorChild.Entries.Count >= this.Degree)
 {
 Entry<TK> successor =
this.DeleteSuccessor(predecessorChild, operationNumber);
node.Entries[keyIndexInNode] = successor; }
else {

predecessorChild.Entries.Add(node.Entries[keyIndexInN ode]);

predecessorChild.Entries.AddRange(successorChild.Entr ies);
predecessorChild.Children.AddRange(successorChild.Chi ldren);
 node.Entries.RemoveAt(keyIndexInNode);
node.Children.RemoveAt(keyIndexInNode + 1);
 this.DeleteInternal(predecessorChild,
keyToDelete, operationNumber);
 }
 }
 }
 private Entry<TK>
DeletePredecessor(Node<TK> node, int operationNumber) {
 IncrementerMemory(operationNumber); if
(node.IsLeaf)
 {
 var result =
node.Entries[node.Entries.Count - 1];
 node.Entries.RemoveAt(node.Entries.Count - 1);
return result;
 }
return
this.DeletePredecessor(node.Children.Last(), operationNumber);
 }
 private Entry<TK>
DeleteSuccessor(Node<TK> node, int operationNumber) {
 IncrementerMemory(operationNumber); if
(node.IsLeaf)
 { var result =
node.Entries[0];
node.Entries.RemoveAt(0); return result;
 }
 return
this.DeletePredecessor(node.Children.First(),
operationNumber);
 } private Entry<TK> SearchInternal(Node<TK>
node, TK key, int operationNumber)
 {
 IncrementerMemory(operationNumber);
Incrementer(operationNumber); var i =
```

```
node.Entries.TakeWhile(entry
=> key.CompareTo(entry.Key) > 0).Count(); for (var index
= 0; index <=i; index++)
 {

IncrementerMemory(operationNumber);
 }
 if (i < node.Entries.Count &&
node.Entries[i].Key.CompareTo(key) == 0)
 { return
node.Entries[i];
 }
 return node.IsLeaf ? null :
this.SearchInternal(node.Children[i], key, operationNumber);
 }
 private void SplitChild(Node<TK>
parentNode, int
nodeToBeSplit)
 {
nodeToBeSplitIndex, Node<TK>
 var newNode = new
Node<TK>(this.Degree, pageSize, entrySize);
 parentNode.Entries.Insert(nodeToBeSplitIndex,
nodeToBeSplit.Entries[this.Degree - 1]);
 parentNode.Children.Insert(nodeToBeSplitIndex + 1,
newNode);
newNode.Entries.AddRange(nodeToBeSplit.Entries.GetRan
ge(this.Degree, this.Degree - 1));
 nodeToBeSplit.Entries.RemoveRange(this.Degree - 1,
this.Degree); if (!nodeToBeSplit.IsLeaf)
 {

newNode.Children.AddRange(nodeToBeSplit.Children.GetR
ange(this.Degree, this.Degree));
 nodeToBeSplit.Children.RemoveRange(this.Degree,
this.Degree);
 }
 }
 private void InsertNonFull(Node<TK> node, TK newKey,
int operationNumber)
 {
 Incrementer(operationNumber);
 IncrementerMemory(operationNumber);
 var positionToInsert =
node.Entries.TakeWhile(entry =>
newKey.CompareTo(entry.Key) >= 0).Count();
 for (var index = 0; index
<=positionToInsert; index++)
 {

IncrementerMemory(operationNumber);
 }
 if (node.IsLeaf)
```

```
{
node.Entries.Insert(positionToInsert, new
Entry<TK>()
{Key = newKey});
return;
}
Node<TK> child =
node.Children[positionToInsert]; if
(child.HasReachedMaxEntries)
{
this.SplitChild(node,
positionToInsert, child);
if
(newKey.CompareTo(node.Entries[positionToInsert].Key) > 0)
{
positionToInsert++;
}
}
this.InsertNonFull(node.Children[positionToInsert],
newKey, operationNumber);
}
private static void Incrementer(int
operationNumber) { switch
(operationNumber)
{
case 1:
InsertCount += 1;
break; case 2:
DeleteCount += 1;
break; case 3:
SearchCount += 1;
break;
}
} private static void
IncrementerMemory(int
operationNumber) { switch
(operationNumber)
{
case 1:
InsertMemoryCount += 1;
break; case 2:
DeleteMemoryCount += 1;
break; case 3:
SearchMemoryCount += 1;
break;
}
}
} class
Program
{ static void Main(string[] args)
{
var bTree = new BTree<int>(195);
Console.WriteLine("NEW TREE"); var rand = new
```

```
Random(); for (var i = 1; i < 1000000; i++)
 { var k =
rand.Next(2000000); if
(!bTree.NumbersInTree.Contains(k))
 {
bTree.NumbersInTree.Add(k);
 }
 bTree.Insert(k);
 }
 Console.WriteLine($"(Height - {bTree.Height})");
 for (var i = 0; i < 200000; i++)
 {
 RunRandomOperation(bTree, rand.Next(1, 4));
 }

 Console.WriteLine($"(Height -
{bTree.Height})");
 Console.WriteLine();
 CalculateResult();
 Console.WriteLine($"Touch Memory
Count during inserting
{BTree<int>.InsertMemoryCountAverage} in average");
 Console.WriteLine($"Touch Memory
Count during deleting
{BTree<int>.DeleteMemoryCountAverage} in average");
 Console.WriteLine($"Touch Memory
Count during searching
{BTree<int>.SearchMemoryCountAverage} in average");
 Console.WriteLine($"Touch Node Count during
inserting {BTree<int>.InsertCountAverage} in average");
 Console.WriteLine($"Touch Node Count during
deleting {BTree<int>.DeleteCountAverage} in average");
 Console.WriteLine($"Touch Node Count during
searching {BTree<int>.SearchCountAverage} in average");
 }
 private static List<Tuple<int, int>>
_operationHistory = new(); private static List<Tuple<int,
int>> _memoryOperationHistory = new();
 private static void RunRandomOperation(
BTree<int> bTree, int operationNumber)
 { var rand = new Random(); var
indexOfTargetNumber = rand.Next(0,
bTree.NumbersInTree.Count);
 var targetNumber =
bTree.NumbersInTree.ElementAt(indexOfTargetNumber);
while (operationNumber == 1 &&
bTree.NumbersInTree.Contains(targetNumber))
 {
 targetNumber = rand.Next(2000000);
 }
 switch (operationNumber)
 {
case 1:
```

```
 {
 Console.WriteLine($"Insert
{targetNumber}");
 _operationHistory.Add(new
Tuple<int, int>(1, BTree<int>.InsertCount));
BTree<int>.InsertCount = 0;
BTree<int>.InsertMemoryCount
= 0; bTree.Insert(targetNumber);
 _operationHistory.Add(new Tuple<int,
int>(1, BTree<int>.InsertCount));

_memoryOperationHistory.Add(new Tuple<int, int>(1,
BTree<int>.InsertMemoryCount));
break; }
case 2:
 {
 Console.WriteLine($"Delete
{targetNumber}");
 BTree<int>.DeleteCount = 0;
 BTree<int>.DeleteMemoryCount
= 0; bTree.Delete(targetNumber);

_memoryOperationHistory.Add(new Tuple<int, int>(2,
BTree<int>.DeleteMemoryCount));
 _operationHistory.Add(new
Tuple<int, int>(2, BTree<int>.DeleteCount));
break; } case 3:
 {
 Console.WriteLine($"Search
{targetNumber}");
 BTree<int>.SearchCount = 0;
BTree<int>.SearchMemoryCount
= 0; bTree.Search(targetNumber);
 _operationHistory.Add(new Tuple<int,
int>(3, BTree<int>.SearchCount));

_memoryOperationHistory.Add(new Tuple<int, int>(3,
BTree<int>.SearchMemoryCount));
break;
 }
 }
 } private static void CalculateResult()
 { var insertCount = _operationHistory
 .Where(x => x.Item1 == 1)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.InsertCountAverage = insertCount.Count
!= 0 ? insertCount.Sum() / insertCount.Count : 0;
 var deleteCount = _operationHistory
 .Where(x => x.Item1 == 2)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.DeleteCountAverage = deleteCount.Count
```

```
!= 0 ? deleteCount.Sum() / deleteCount.Count : 0;
 var searchCount = _operationHistory
 .Where(x => x.Item1 == 3)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.SearchCountAverage = searchCount.Count
!= 0 ? searchCount.Sum() / searchCount.Count : 0;
 var insertRotateCount =
_memoryOperationHistory
 .Where(x => x.Item1 == 1)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.InsertMemoryCountAverage =
insertRotateCount.Count != 0 ?
insertRotateCount.Sum() / insertRotateCount.Count : 0;
 var deleteRotateCount =
_memoryOperationHistory
 .Where(x => x.Item1 == 2)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.DeleteMemoryCountAverage =
deleteRotateCount.Count != 0 ?
deleteRotateCount.Sum() / deleteRotateCount.Count : 0;
 var searchRotateCount =
_memoryOperationHistory
 .Where(x => x.Item1 == 3)
 .Select(x => x.Item2)
 .ToList();
 BTree<int>.SearchMemoryCountAverage =
searchRotateCount.Count != 0 ?
searchRotateCount.Sum() / searchRotateCount.Count : 0;
 }
 }
}

int xy2d (int n, int x, int y) {
 int rx, ry, s, d=0;
 for (s=n/2; s>0; s/=2) {
 rx = (x & s) > 0;
 ry = (y & s) > 0;
 d += s * s * ((3 * rx) ^ ry);
 rot(s, &x, &y, rx, ry);
}
return d;
}

void d2xy(int n, int d, int *x, int *y) {
 int rx, ry, s, t=d;
 *x = *y = 0;
 for (s=1; s<n; s*=2) {
 rx = 1 & (t/2);
 ry = 1 & (t ^ rx);
 rot(s, x, y, rx, ry);
```

```
 *x += s * rx;
 *y += s * ry;
 t /= 4;
}
}

void rot(int n, int *x, int *y, int rx, int ry) {
if (ry == 0) {
if (rx == 1) {
*x = n-1 - *x;
*y = n-1 - *y;
}
int t  = *x;
*x = *y;
*y = t;
}
}
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	_79_ с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Панасюк О.В.			Застосування фрактальних кривих у задачі індексації двовимірних об'єктів	Шифр групи	Код напр./спец.
Перевірив		Аврунін О.Г.				СШІм-21-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					