Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Центр Післядипломної освіти
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Індексування об’єктів, що рухаються, на основі B-дерев
(тема)

Виконав:
студент 2 курсу, групи СШІздм-21-1
Сідельник Д.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. Удовенко С.Г.
(посада, прізвище, ініціали)

Допускається до захисту

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

2023 р.

Харківський національний університет радіоелектроніки

Центр Післядипломної освіти
(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 Сідельнику Дмитру Валерійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Індексування об'єктів, що рухаються, на основі B-дерев

затверджена наказом університету від 31 березня 20 23 р. № 73Стз
2. Термін подання студентом роботи до екзаменаційної комісії 23 травня 20 23 р.
3. Вихідні дані до роботи Науково-технічні публікації, дані Інтернет-джерел та відомих наукових проєктів

4. Перелік питань, що потрібно опрацювати в роботі
1) Деревовидні структури даних для індексації точкових об'єктів
2) Модифікації B-дерев
3) Індексація рухомих об'єктів за допомогою TPR-дерев

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 кваліфікаційну роботу	03.04.2023	виконано
2	Аналіз предметної області	04.04.2023-09.04.2023	виконано
3	Дослідження структур даних для індексації точкових	10.04.2023-16.04.2023	виконано
	об'єктів		
4	Дослідження структур даних для індексації просторових	17.04.2023-23.04.2023	виконано
	об'єктів		
5	Дослідження структур даних для індексації рухомих	24.04.2023-30.04.2023	виконано
	об'єктів		
6	Програмна реалізація R- та TPR-дерев, експериментальні	01.05.2023-08.05.2023	виконано
	дослідження		
7	Попередній захист	16.05.2023	виконано
8	Захист перед ЕК	23.05.2023	виконано

Дата видачі завдання 3 квітня 2023 р.

Студент _____ О. Єдельман
(підпис)

Керівник роботи _____ проф. Удовенко С.Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 79 с., 20 рис., 1 табл., 2 дод., 12 джерел.

ПРОСТОРОВІ ДАНІ, РУХОМІ ОБ’ЄКТИ, СТРУКТУРИ ДАНИХ, B-ДЕРЕВА, R-ДЕРЕВА, TPR-ДЕРЕВА.

Об’єктом досліджень є структури просторових даних.

Предметом досліджень кваліфікаційної роботи є індексування рухомих об’єктів.

Метою роботи є дослідження R- та TPR-дерев для індексування рухомих об’єктів та порівняння цих структур за ефективністю пошукових запитів.

ABSTRACT

Explanatory note: 79 p., 20 fig., 1 tabl., 2 ann., 12 sources.

B-TREES, DATA STRUCTURES, MOVING OBJECTS, R-TREES, SPATIAL DATA, TPR-TREES.

The object of research is spatial data structures.

The subject of research in the qualification work is indexing of moving objects.

The purpose of the work is studying R- and TPR-trees for indexing moving objects and comparison of these structures according to the effectiveness of search queries.

ЗМІСТ

ВСТУП

В роботі досліджуються методи та алгоритми індексування точкових 2D об’єктів, які рухаються. Базовими структурами даних, призначеними для зберігання даних щодо великої кількості об’єктів, є B-дерева. Дослідження цих структур даних, їх особливостей та відмінностей між собою, а також методів реалізації типових словникових операцій над даними, є першим завданням, що виконується у цій роботі.

Відомо, що для зберігання просторових даних найбільш придатною структурою даних є R-дерева. Вони є розширенням B+ дерев на випадок інтервальних просторових даних, отже дуже схожі на них за структурою та алгоритмами словникових операцій (пошук, додавання, видалення). Проте, інтервальність даних та їхня багатовимірність призводять до того, що елементи даних у деревах втрачають дуже важливу властивість – лінійну впорядкованість. Наслідком є те, що ефективність деревовидних структур, призначених для зберігання просторових даних, суттєво залежить від порядку додавання елементів даних до дерева.

Що стосується проблеми індексування рухомих об’єктів, то здається природним використовувати ті ж самі структури, тобто R-дерева, але додати до ключів просторові компоненти швидкості об’єктів. Втім, через рух точок межі вузлів змінюються, внаслідок чого будь-яке, навіть оптимальне, розбиття точок по вузлах втрачає свою оптимальність з впливом часу.

Альтернативою звичайним R-деревам є TPR-дерева, у яких групування точок у вузли відбувається на підставі не лише поточних позицій точок, а й майбутніх, згідно з поточними швидкостями. Це призводить до того, що отримане групування зостається приємним за якістю не лише на момент його створення, а й протягом заданого горизонта планування у часі.

Аналіз TPR-дерев, їхня програмна реалізація, моделювання та співставлення з R-деревами формують сутність цієї кваліфікаційної роботи.

1 ДЕРЕВОВИДНІ СТРУКТУРИ ДАНИХ ДЛЯ ІНДЕКСАЦІЇ ТОЧКОВИХ ОБ'ЄКТІВ

1.1 Двійкове дерево пошуку

Деревовидні структури є найбільш широко вживаними структурами, призначеними для зберігання наборів даних, які змінюються (динамічних наборів даних). Це зумовлено як відносною простотою організації таких структур, так і надзвичайно високою ефективністю виконання типових словникових операцій над даними: пошук за ключем, додавання, видалення.

Найпростішою, проте базовою, деревовидною структурою даних є так зване двійкове дерево пошуку (ДДП, або BST, binary search tree), запропоноване у 1960р. Ендрю Бутом. ДДП є двійковим деревом, для якого виконуються три додаткові умови (властивості дерева пошуку):

– обидва піддерева (ліве та праве) є двійковими деревами пошуку;

– у всіх вузлів лівого піддерева довільного вузла *X* значення ключів даних менше чи рівні, ніж значення ключа даних самого вузла *X*;

– у всіх вузлів правого піддерева довільного вузла *X* значення ключів даних більше, ніж значення ключа даних самого вузла *X*.

Ключі можуть мати довільну природу (цілі числа, дійсні числа, символьні рядки тощо), єдиною вимогою до них є порівнюваність. Тобто, для довільних ключей X та Y має бути визначена операція порівняння. При цьому

$$(X < Y) or (X == Y) or (X > Y) = True \tag{1.1}$$

У більшості випадків передбачається, що ключі є унікальними.

Для цілей реалізації двійкове дерево пошуку можна визначити так:

– вузли є записами, які складаються з полів (`key`, `value`, `left`, `right`), де `key` – ключ, `value` – деякі дані, прив'язані до вузла (або покажчики на них), `left` та `right` – покажчики на дочірні

вузли. Іноді до вузлів додається ще поле `parent` – покажчик на батьківський елемент.

– для будь-якого вузла *X* виконуються властивості дерева пошуку: ключ батьківського вузла строго більше ключів лівого дочірнього та нестрого менше ключів правого дочірнього:

$$\texttt{X.left.key} \le \texttt{X.key} < \texttt{X.right.key} \qquad (1.2)$$

Базовий інтерфейс ДДП складається з трьох словникових операцій:

– `FIND(K)` – пошук вузла, ключ якого дорівнює K.

– `INSERT(K, V)` – додавання до дерева пари (key, value) = (K, V).

– `REMOVE(K)` – видалення вузла, в якому зберігаються дані з ключем K.

Власне, двійкове дерево пошуку – це структура даних, здатна зберігати таблицю пар (key, value) і підтримує три операції: FIND, INSERT, REMOVE.

Зазвичай, інтерфейс двійкового дерева включає ще додаткові операції, які реалізують обхід вузлів дерева. Наприклад, INFIX_TRAVERSE – обхід ДДП за неспаданням ключів вузлів.

Зазначені словникові операції є досить простими в реалізації. Наприклад, пошук елемента (FIND) виконується наступним чином: вхідними даними є покажчик на корень дерева (T) та значення шуканого ключа (K). Необхідно перевірити, чи існує у дереві ключ K, і якщо так, то повернути посилання на цей вузол. Псевдокод наведений на рисунку1.1.

Додавання елемента (INSERT) означає вставку пари K, V (ключ, дані) у дерево T таким чином, щоб не порушити ключову властивість ДДП (1.2).

Алгоритм додавання дуже схожий на алгоритм пошуку (бо перш ніж вставляти дані треба спочатку знайти для них місце). Головною відмінністю є те, що зберігати (та потім використовувати) потрібно покажчик не на вузол з ключем K (бо він завжди NULL), а на його батьківський вузол (рис.1.2).

```
X=T (починаємо з кореневого вузла).
До тих пір, поки дерево T не порожнє, працюємо:
   Якщо K<X.key, то продовжуємо пошук у лівому піддереві
   (X=X.left).
      Інакше якщо K>X.key, то продовжуємо пошук у правому
      піддереві (X=X.right).
         Інакше якщо K==X.key, то повернути посилання на
         цей вузол (return X) та завершити роботу (ключ
         знайдено)
– Вузол не знайдено; повертаємо NULL та виходимо з функції.
```

Рисунок 1.1 – Псевдокод функції FIND для ДДП

```
X = T, Y = NULL;  // починаємо з кореневого вузла
while X {         //поки дерево не порожнє, опускаємось
униз
     if K<=X.key {
          X = X.left; Y = X;} //переходимо до лівого
          піддерева
     else {
          X = X.right; Y = X;} //переходимо до правого
Створюємо новий вузол X;
X.key = K; X.value = V; X.left = X.right = NULL;
if (K <= Y.key) Y.left = X
   else Y.right = X;
return
```

Рисунок 1.2 – Псевдокод функції INSERT для ДДП

Видалення вузла (REMOVE).

Дано: дерево із коренем T та ключем вузла, що видаляється, K.

Якщо дерево T порожнє, зупинитися;

Інакше порівняти K із ключем X кореневого вузла n.

Якщо K>X, рекурсивно видалити K з правого піддерева T;

Якщо K<X, рекурсивно видалити K з лівого піддерева T;

Якщо K = X, необхідно розглянути три випадки.

Якщо обох дітей немає, то видаляємо поточний вузол і обнуляємо

посилання нього батьківського вузла;

Якщо одного з дітей немає, значення полів дитини m ставимо замість відповідних значень кореневого вузла, затираючи його старі значення, і звільняємо пам'ять, займану вузлом m;

Якщо обидві дитини присутні, то

Якщо лівий вузол m правого піддерева відсутня (n->right->left)

Копіюємо з правого вузла в поля, що видаляється K, V і посилання на правий вузол правого нащадка.

Інакше.

Візьмемо найлівіший вузол m, правого піддерева n->right;

Скопіюємо дані (крім посилань на дочірні елементи) з m до n;

Рекурсивно видалимо вузол m.

INFIX_TRAVERSE

Якщо дерево порожнє, зупинитися.

Інакше

Рекурсивно обійти ліве піддерево T.

Застосувати функцію f до кореневого вузла.

Рекурсивно обійти праве піддерево T.

У найпростішому випадку функція f може виводити значення пари (K, V). У разі використання операції INFIX_TRAVERSE будуть виведені всі пари в порядку зростання ключів.

Час роботи базових словникових функцій (FIND, INSERT, REMOVE) дорівнює $O(h)$, де h – глибина дерева. Якщо дерево гарно збалансоване, то кожен вузол (окрім вузлів найнижчого рівня) має рівно два дочірніх. Тоді $h = \log_2 n$ (де n – кількість вузлів у дереві). Але можливо й таке, що $h = n - 1$ – це той випадок, коли всі вузли витягнуті у ланцюг.

Тому ДДП у своєму «чистому» вигляді застосовуються рідко. Частіше використовуються так звані збалансовані дерева двійкового пошуку: АВЛ, WAVL, червоно-чорні, косі (splay) та інші. Втім, всі вони підтримують основну властивість ДДП (1.2), а отже й функції (методи) FIND та

INFIX_TRAVERSE без змін. Методи INSERT та REMOVE більш складні, бо побудовані так, щоб постійно підтримувати баланс.

1.2 Структура B-дерева

Одним з можливих узагальнень ДДП є k-ічні дерева пошуку, тобто дерева з коефіцієнтом розгалуження $k > 2$.

У таких деревах в кожному вузлі зберігається не одна, а $k-1$ пар (`key,data`) та, відповідно, k покажчиків на дочірні вузли (рис. 1.3).

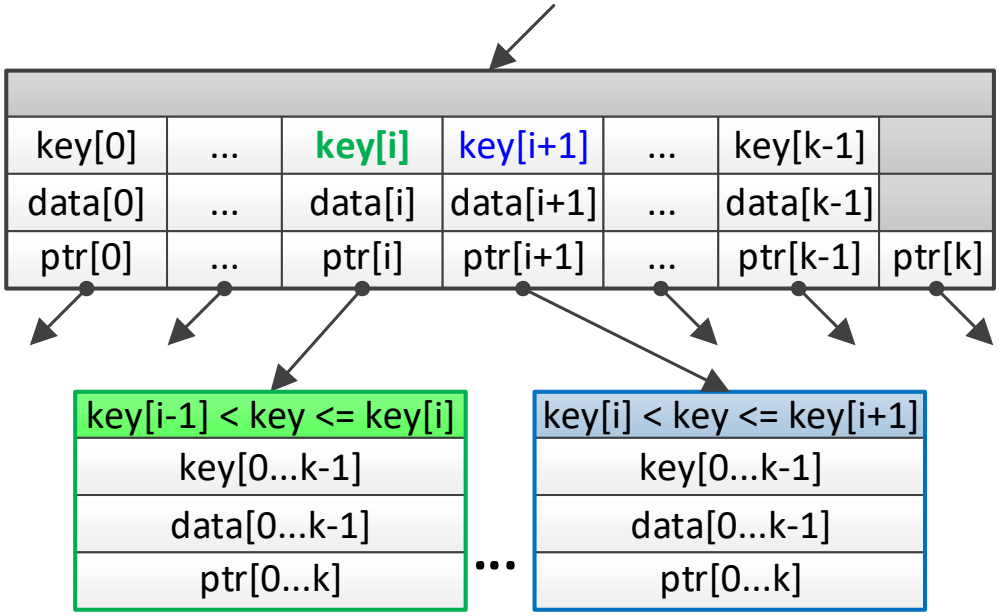


Рисунок 1.3 – Структура k-ічного дерева пошуку

Легко бачити, що такий підхід дозволяє суттєво (асимптотично – вдвічі) зекономити пам’ять на зберігання покажчиків: на кожний елемент даних (`key,data`) припадає $k/(k-1) \approx 1$ покажчик, а у звичайному ДДП – два.

Проте, у чистому вигляді k-ічні дерева пошуку майже ніколи не застосовуються. Це зумовлено тим, що підтримка такого «щільно упакованого» стану під час додавання чи видалення елементів, є дуже витратною.

На практиці широкого поширення набув більш м’який варіант k-ічних дерев пошуку: так звані B-дерева. В них фактичний коефіцієнт розгалуження вузла не є фіксованим (k), а може коливатись від деякого мінімально припустимого значення k_{min} до максимального k_{max}. Завдяки цьому додавання чи видалення елементів відносно рідко призводить до розбиття чи злиття вузлів, до переносу ключів з одного вузла до іншого. Це дуже прискорює виконання цих словникових операцій.

Головною метою створення B-дерев було прискорення словникових операцій за тих умов, що структура даних (тобто дерево) не є повністю завантаженою у фізичну пам’ять (RAM), а зберігається на жорсткому диску.

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 $h \le 2\log n$. З цього випливає, що при $n = 10^8$ (що не дуже багато) висота дерева складе $27 \le h \le 54$. 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 $T_{Search} \le 3h$, 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 100 000 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* B-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 *B*-деревах дорівнює

$$t = \lceil m / 2 \rceil, \tag{1.3}$$

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

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

$$\log_m (n+1) - 1 \leq h \leq \log_t \frac{n+1}{2}. \tag{1.4}$$

У прикладі, що розглядається ($n=10^8$, $t=98$), з (1.4) отримаємо, що $2.49 \le h \le 3.87$. 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.

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 (використовуються 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.

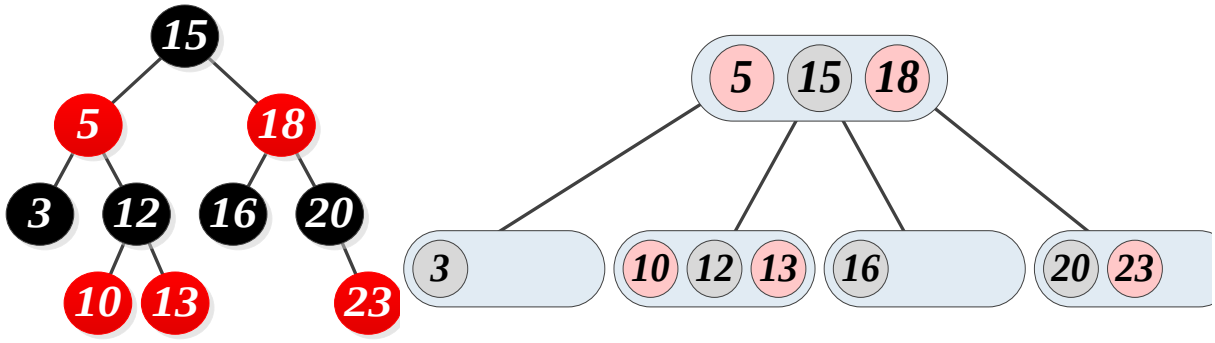
Вважається, що корінь дерева завжди відображений в оперативну пам'ять, тому викликати Disk_Read(root) ніколи не потрібно. У той же час, якщо кореневий вузол змінювався, виклик Disk_Write(root) необхідний.

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 (рис.1.4).



а) б)

Рисунок 1.4 – Червоно-чорне дерево та B-дерево порядку *m=4*

1.3 Словникові операції над B-деревами

Процедура пошуку за ключом (B_Tree_Search) є природним узагальненням процедури Tree_Search, що застосовується у двійкових деревах (лістинг 1.1).

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.

Лістинг 1.1 – Процедура пошуку за ключом (B_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
```

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 (лістинг 1.2). 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 1.2 – Процедура вставки 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 1.3).

Лістинг 1.3 – Процедура 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?
```

Продовження лістингу 1.3

```
      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 (лістинг 1.4). 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.

Лістинг 1.4 – Процедура 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)
```

Схему виконання цієї процедури наведено на рисунку 1.5, а на рисунку 1.6 наведено приклади послідовного виконання вставки.

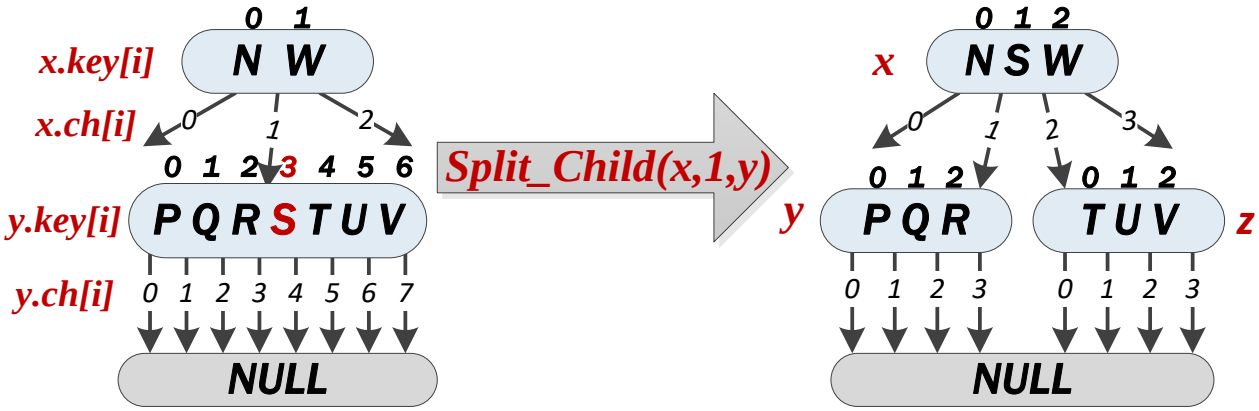


Рисунок 1.5 – Розбиття вузла в *B*-дереві (*m=8*)

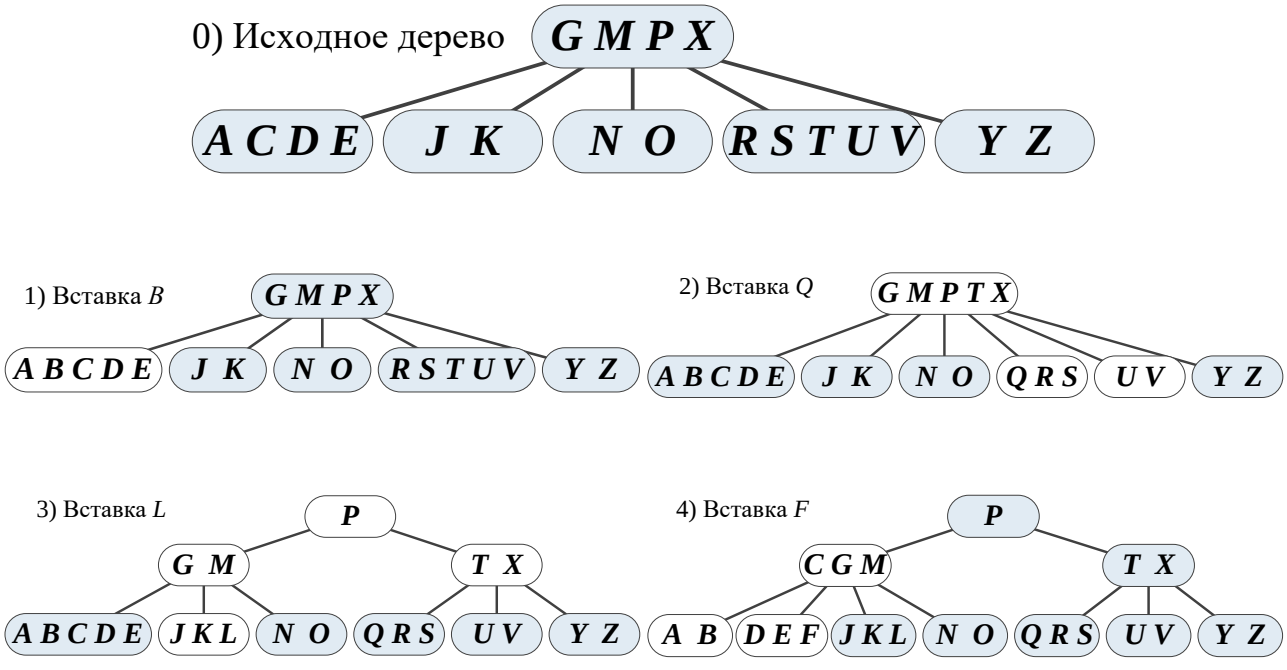


Рисунок 1.6 – Вставка ключів в *B*-дерево (*m=6*)

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 (рис. 1.7(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 (рис.1.7(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 рис.1.7(1) ключ G, то він би замінився на ключ-successor J (мінімальний в JKL). Цей випадок симетричний попередньому та на рисунку 1.7 не показаний.

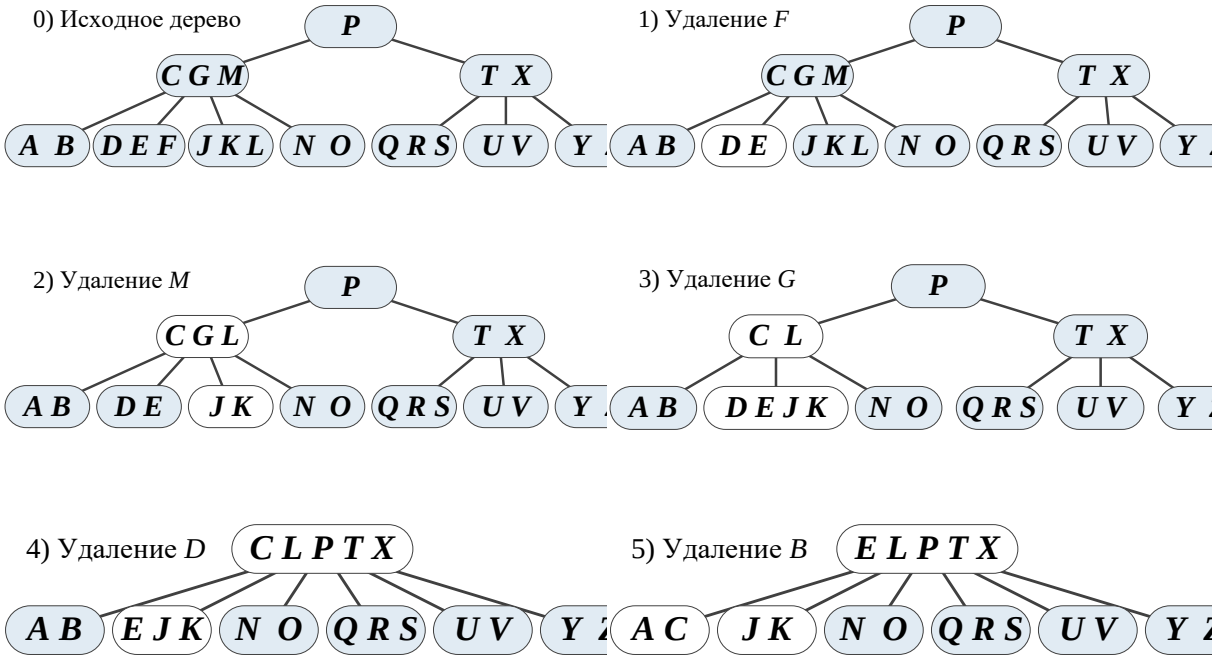


Рисунок 1.7 – Видалення ключів з *B*-дерева (*m=6, t=3*)

В обох випадках може порушитись властивість заповненості B-дерева (некореневий вузол повинен мати хоча б $t-1$ ключів). Наприклад, у дереві рис.1.7(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 (рис.1.7(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 рис.1.7(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 (рис. 1.7(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 (рис.1.7(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 рис.1.7(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.

У лістингу 1.5 наведено псевдокод функції B_Tree_Delete.

Лістинг 1.5 – 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.1.7(1)
        Видалити x.key[i] та x.ch[i] з x
        Disk_Write(x)
      else                    // Рис.1.7(2), Рис.1.7(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.1.7(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
```

Продовження лістингу 1.3

```
            Disk_Write(x)
            x = y
    else  // Рис.1.7(4) та рис.1.7(5)
       реалізувати самотужки
  end_while
end
```

Можна бачити, що видалення даних з B-дерев є значно більш громіздке, ніж додавання чи пошук. Проте, це стосується не тільки B-дерев, а будь-яких впорядкованих дерев, від найпростіших дерев двійкового пошуку до збалансованих дерев пошуку на B-дерев.

При цьому варто зазначити, що, поперше, асимптотична складність видалення не більше, ніж додавання (обидві $O(h)$, де h – висота дерева), по-друге, видалення ключів хоч і відноситься до базового набору методів B-дерев, але дуже часто воно потребується не настільки часто, як інші словникові операції – додавання та пошук.

2 МОДИФІКАЦІЇ B-ДЕРЕВ

2.1 B+-, B*- та B*+-дерева

B-дерева є базовою моделлю структур, що призначені для індексування великомасштабних наборів даних. Проте, існує багато модифікацій таких дерев, які враховують специфічні потреби та відображають специфічні властивості відповідних застосувань. Прикладами таких модифікацій є B^+, B^* и B^{*+}-дерева. Інколи ці дерева також називають B-деревами.

B*-дерево відрізняється від звичайного B-дерева тим, що мінімальний ступінь вузла у B*-дереві дорівнює $2/3m$ (а не $1/2m$). Це призводить до того, що при необхідності розбиття вузла (під час додавання ключа) здійснюється спроба "переливу" зайвих ключів у сусідній (праворуч) вузол. Цей "перелив" здійснюється так, щоб кількість ключів у цих двох вузлах стала рівною. Якщо такий "перелив" неможливий (а це може бути тільки якщо обидва вузли є заповненими (`full`), тобто містять по $m - 1$ ключу), то відбувається розбиття. Якщо у B-деревах розбивається лише поточний вузол (навпіл), то у B*-деревах відбувається розбиття “2 на 3”, тобто з заповнених поточного вузла та його правого сусіда формуються три вузла (заповнені рівною мірою, на 2/3).

Аналогічним чином, якщо під час видалення ключа вузол має мінімально припустиму заповненість $2/3m$ (тобто далі худнути не може), то відбувається об’єднання цього вузла з сусідним. Під час об’єднання ключі розподілюються між ними навпіл. Якщо таке об’єднання неможливе, то виконується об’єднання “3 в 2”.

Таким чином, B*-дерева мають меншу кількість вузлів, ніж B-дерева, за рахунок більш компактного пакування. Платою за цю економію пам’яті є більша складність розбиття та об’єднання вузлів під час вставки/видалення.

Забігаючи наперед, відзначимо, що B*+-дерева поєднують в собі властивості B*- та B+-дерев (тобто мають структуру B+-дерева та нижній поріг заповнюваності $2/3m$).

Найбільш поширеною модифікацією B-дерев є B+-дерева. Головною рисою B+-дерев є те, що дані (або покажчики на них) зберігаються лише у листах. Зазвичай листи містять покажчики на наступний лист: це спрощує та прискорює послідовний доступ. Інші, нелистові, вузли застосовуються лише для організації доступу до листів, тому вони не містять даних, а містять лише ключі та покажчики на дочірні вузли (рис. 2.1).

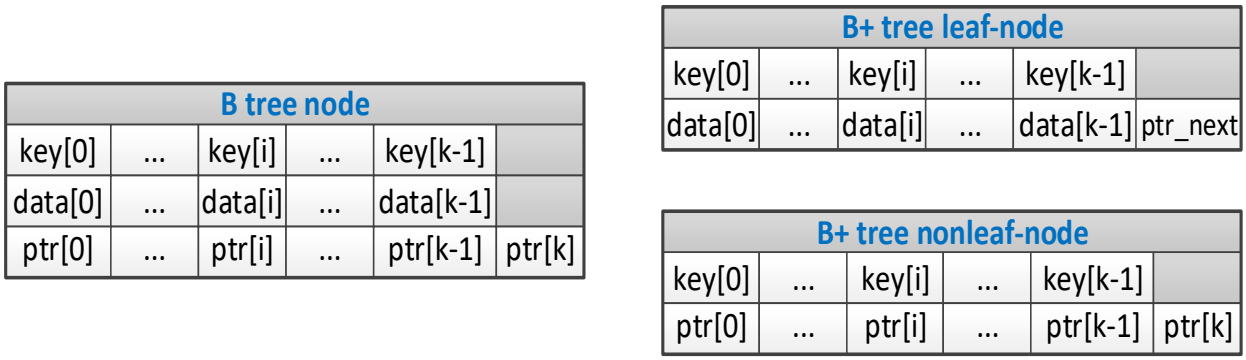


Рисунок 2.1 – Різниця між структурою вузлів B та B+ дерев

Зрозуміло, що ключі нелистових вузлів є копіями “справжніх” ключів – ключів даних листів (рис. 2.2). Для розрізнення типу вузла він повинен мати спеціальне логічне поле `x.leaf`, яке показує, чи є вузол листом.

На перший погляд може здаватись, що цей підхід (тобто дублювання ключів) не є раціональним. Проте, відсутність поля `data` у нелистових вузлах та поля `ptr` у листових вузлах дозволяє збільшити порядки цих вузлів відносно B-дерев. При цьому порядки (тобто кількість покажчиків = кількість ключів + 1) листових та нелистових вузлів можуть відрізнятись між собою.

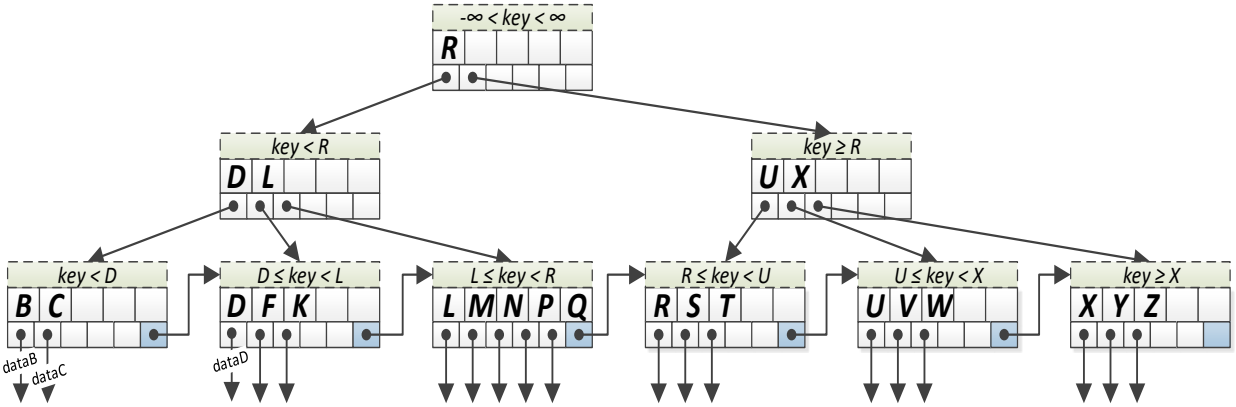


Рисунок 2.2 – Приклад B^+-дерева порядка $m = 6$

Останній фактор (можливість використання різних порядків для листів та не-листів) суттєво проявляється у разі, якщо дані, асоційовані з ключами, мають значний розмір.

Розберемо два приклади. Перший приклад – дані є малими за розміром. Нехай ключі (`key`) мають розмір 24 байта, дані (`data`) – 4 байта, та поле `ptr` – теж 4 байта (це звичайний розмір покажчиків). Якщо розмір сторінки (вузла) дорівнює 16 Кбайт, то максимальний коефіцієнт розгалуження B-дерева дорівнює $m = 16K/(24 + 4 + 4) = 512$. Тоді для зберігання $N = 1$ млрд. $\approx 2^{30}$ елементів необхідно від 2 до 4 млн. сторінок (від N/m до $2N/m$). Висота такого B-дерева складе від $\log_m N$ до $\log_{m/2} N$ (з округленням до цілого вгору), тобто 4. Таким чином, базові словникові операції (пошук, додавання, видалення) вимагатимуть завантаження в пам’ять не більше, ніж трьох сторінок, бо корньова сторінка вважається завжди попередньо завантаженою.

У B+-дереві розміри емність листових та нелистових сторінок співпаде (бо розмір `data` дорівнює розміру `ptr`): $m_L = m_{NL} = \lfloor 16K/28 \rfloor = 585$ елементів. Тоді кількість листових вузлів дорівнюватиме від N/m_L до $2N/m_L$, тобто від 1.7 до 3.4 млн. Відповідна кількість нелистових вузлів (суто індексних сторінок) становитиме від 3 до 12 тисяч. Висота B+-дерева співпаде з висотою B-дерева, тобто 4.

А тепер розглянемо приклад, у якому розмір поля `data` є відносно великим: 100 байт. Тоді максимальний коефіцієнт розгалуження B-дерева дорівнюватиме $m = 16K/(24 + 100 + 4) = 128$. Зберігання $N = 1$ млрд. $\approx 2^{30}$ елементів потребуватиме від 7.8 до 15.6 млн. сторінок. Висота такого B-дерева складе від $\log_m N$ до $\log_{m/2} N$ (з округленням до цілого вгору), тобто 5.

У B+-дереві на одній листовій сторінці можна буде розмістити $m_L = \lfloor 16K/124 \rfloor = 132$ пари ключ-дані. Відповідно кількість листових вузлів дорівнюватиме від 7.6 до 15.2 млн. А от для індексних сторінок (нелистових вузлів) максимальний коефіцієнт розгалуження буде таким же, як і для прикладу з чотирьохбайтним полем даних, тобто $m = \lfloor 16K/28 \rfloor = 585$. Кількість нелистових вузлів становитиме від 13 до 52 тисяч. Але при цьому висота B+-дерева буде менше, ніж у B-дерева: 4 проти 5. В результаті пошук (а також додавання та видалення) відбуватиметься в $(5 - 1)/(4 - 1)$ разів, тобто на третину швидше, ніж у відповідному B-дереві.

Таким чином, можна зробити висновок: B+-дерева не тількі більш економні щодо використання пам’яті, ніж B-дерева, але й можуть мати меншу висоту, що призводить до суттєвого пришвидшення словникових операцій. Цей ефект визначається розміром поля даних: чим воно більше, тим значніше пришвидшення.

2.2 R-дерева

Ще однією варіацією B (B+) дерев є R-дерево. Ця структура даних використовується для організації доступу до просторових даних, тобто для індексації даних за складеним (двомірним чи тримірним) ключем. R-дерево було запропоноване в 1984 році Антоніном Гуттманном [6], [7], [8]. Типовим запитом з використанням R-дерев міг би бути такий: «знайти всі таксі в межах 100 метрів від мене».

Хоча R-дерева були запропоновані Гуттманном для індексування саме геолокаційних даних (тобто координатами, або вимірами, слугували саме традиційні декартові координати x, y, чи широта-довгота), проте сфера застосування цієї структури даних набагато ширша: по суті ключі можуть мати довільну природу. Наприклад, вік-ціна автомобіля.

R-дерево розбиває двовимірний простір (прямокутню область площини) на множину ієрархічно вкладених обмежуючих прямокутників (рис. 2.3).

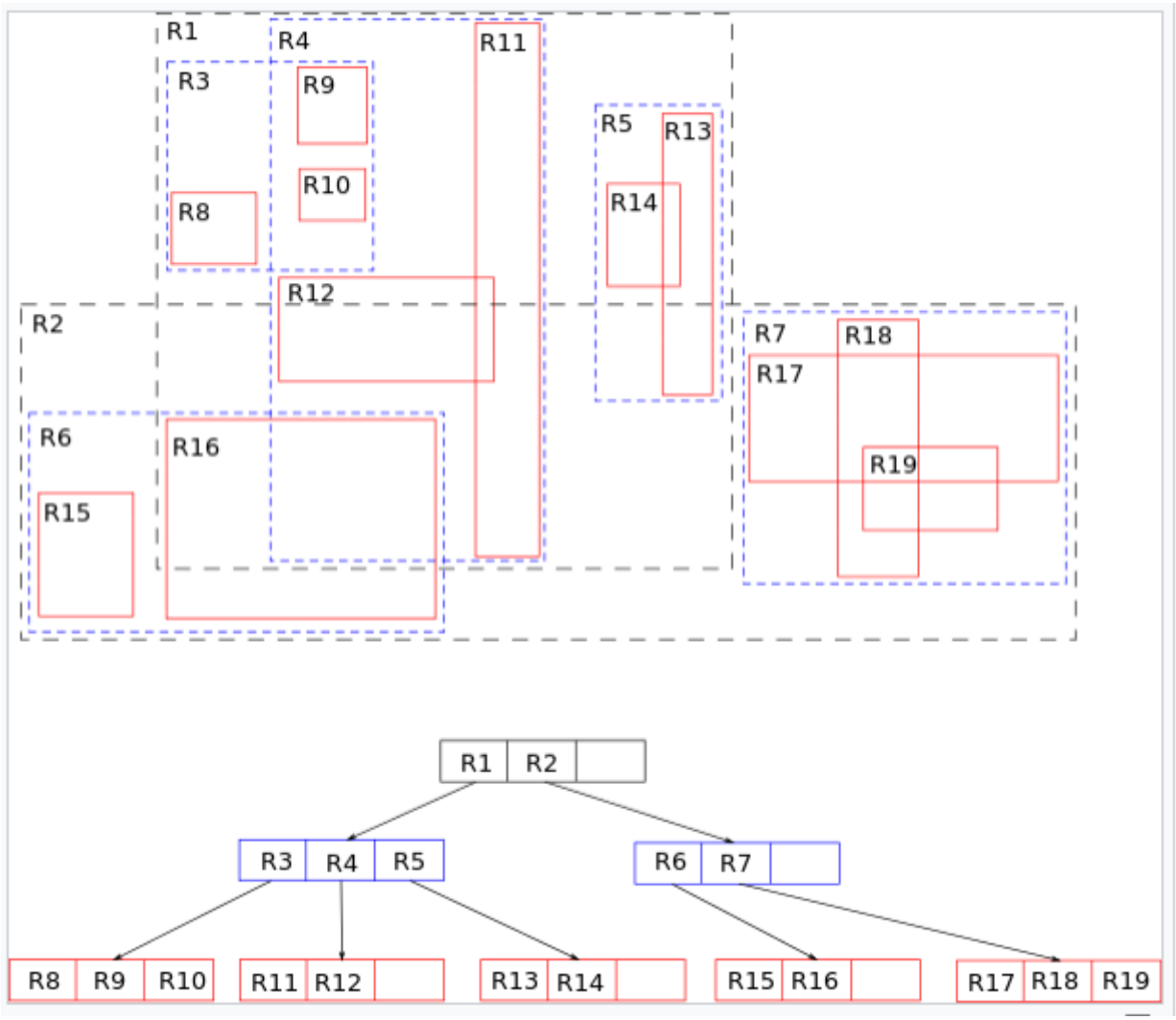


Рисунок 2.3 – Приклад R-дерева

Так само, як і у випадку B^+-дерев, кожен елемент листової вершини зберігає два поля: дані, що описують об'єкт, та ключ. Елементи нелистових вузлів мають поля ключ та покажчик на відповідний дочірній вузол.

Первинні об’єкти (дані), які зберігаються, характеризуються ключем, яким є координати обмежуючого прямокутника ($x_{min}, x_{max}, y_{min}, y_{max}$). Ці первинні об’єкти можуть бути як просторовими (тобто мати ширину та довжину), так і точковими (для них $x_{min} = x_{max}, y_{min} = y_{max}$), або лінійними.

Так само, як і у випадку B^+-дерев, вузли R-дерев мають змінну кількість дочірніх елементів (від `minNumOfEntries` до `maxNumOfEntries`).

У кореневій вершині може бути від 2 до $maxNumOfEntries$ дочірніх елементів.

Для коректної роботи алгоритмів вставки та видалення необхідно, щоб $minNumOfEntries \leq maxNumOfEntries / 2$. Стандартним вибором є $minNumOfEntries = t - 1$, $maxNumOfEntries = 2t - 1$. Зазвичай, це значення коефіцієнта розгалуження (m) є різним для листових та нелистових вершин. Воно (тобто t , або $m = 2t$) визначається співвідношенням розміру файлової сторінки з розміром ключів, покажчиків та даних. Найчастіше воно становить від 50 до кількох тисяч.

У порівнянні з звичайними B^+ -деревами R-дерева мають дві важливі особливості: поперше, ключі R-дерев (координати обмежуючих прямокутників) не є лінійно впорядкованою множиною. По-друге, ключі R-дерев є інтервальними (а не точковими, скалярними). З цього випливають щонайменше два наслідка: поперше, один й той самий шуканий прямокутник, або шукана точка, може належати декільком вузлам R-дерева. Візуально це проявляється в тому, що обмежуючі прямокутники вузлів можуть перетинатись. Наприклад, на рис. 2.3, більшість точок R12 або R16 належать одночасно як R1, так і R2. При цьому дочірній прямокутник (чи точка) насправді належать лише одному батьківському вузлу (R12 належить

R4, який в свою чергу належить R1, так саме $R16 < -R6 < -R2$). Другим наслідком інтервальності ключів та відсутності лінійного порядку є те, що ефективність R-дерева дуже суттєво залежить від порядку, за яким воно заповнюється.

Основна ідея побудови R-дерев полягає у групуванні сусідніх дочірніх об’єктів (первинних листів – точок, вторинних вузлів – прямокутників) так, щоб обмежуючий цю групу, тобто вузол, прямокутник був найменшим в деякому сенсі. Звідси й походить літера «R» (rectangle) у назві цього дерева.

На рівні листа, прямокутник описує один об'єкт, можливо точку; на більш високих рівнях агрегації – множину об'єктів.

Оскільки R-дерева є різновидом *B*, B^+ та *B**-дерев, то вони також є збалансованими деревами пошуку. Це означає, що всі листя знаходяться на однаковій висоті. Дані організовані в сторінках, передбачається, що структура даних зберігається на диску, тобто 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. На підставі співставлення ключа (тобто координат обмежуючого прямокутника) з прямокутником запиту (range query) приймається рішення про те, чи варто шукати всередині відповідного піддерева.

Якщо прямокутник запиту (range query) знаходиться поза межами обмежуючого прямокутника вузла, то жоден, з дочірніх об’єктів цього вузла не задовільняє запиту. Інакше шукані дані можуть належати цьому вузлу, а можуть й не належати.

Таким чином, на відміну від звичайних, тобто одновимірних, дерев (BST, RB, B, B+, AVL, WAVL та інших), всі пошукові запити у R-дереві, навіть пошук поодинокої точки, слід розглядати як групові. Інакше кажучи, неможливо однозначно локалізувати дочірній вузол, до якого

належить шуканий об’єкт, бо обмежуючі прямокутники вузлів перетинаються. Тому необхідно під час пошуку формувати (та використовувати) допоміжну структуру даних: чергу підозрілих вузлів. Механізм формування та застосування такої черги є аналогічним тому, який використовується для реалізації послойного обходу звичайних дерев двійкового пошуку.

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.

Розглянемо приклад пошуку даних у дереві з рисунку 2.3, розташованих на місці літери R у прямокутнику R12. Під час аналізу кореневого вузла дерева ще невідомо, у якому з прямокутників (R1 чи R2) вони знаходяться. Тому обидва ці вузли заносяться в чергу. Зчитуємо (з видаленням з черги) та аналізуємо головний її елемент, тобто R1. Шукані дані не можуть знаходитись у прямокутниках R3 та R5 (бо вони не перетинаються з запитом), а от у R4 – можуть. Тому заносимо R4 у чергу. Переходимо до обробки наступного елемента черги – R2. Його дочірні прямокутники (R6 та R7) не задовільняють запиту. Наступна голова черги – це R4. Дочірній елемент R11 не підходить, а R12 – підходить. Заносим його

у хвіст черги. Зчитуємо його ж з голови та знаходимо у ньому шуканий елемент (який, як було сказано, є однією з точок літери R).

Як видно з аналізу алгоритму вставки, пошук буде тим ефективнішим, чим меншою є кількість (та площина) перетинання прямокутників-вузлів між собою. Другим фактором підвищення ефективності є вимога щодо того, щоб «близько розташовані» об'єкти під час групування потрапляли (за можливістю) в один вузол. Ця вимога обгрунтовується тим, що типові запити до R-дерев є не точковими, а груповими. Тобто часто шукається не поодинокий об’єкт x, y, а всі об’єкти, що потрапляють у заданий прямокутник пошукового запиту $(x_{min}, x_{max}, y_{min}, y_{max})$. Типовий запит такого роду «знайти всі таксі в межах 100 метрів від мене» вже зазначався вище.

Для забезпечення ефективності пошуку під час додавання нових об’єктів використовується правило найменшого розширення: новий об'єкт потрапить у той вузол, для якого потрібно найменше розширення його обмежуючого прямокутника.

Класичний алгоритм вставки, запропонований розробником R-дерев Антоніном Гуттманном [6], описаний нижче.

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:

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

– змінює обмежуючий прямокутник ключа з урахуванням вставленої точки;

– повертає splittedNodes, які нам повернув doInsert;

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

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

– рекурсивно викликається chooseLeaf для обраного нащадка;

– поправляються обмежуючі прямокутники;

– якщо splittedNodes від рекурсивного виклику нульові, то покидаємо функцію, інакше:

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

– 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 варіант простий, має лінійну складність. Він не оптимальний, але широко застосовується:

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:

- якщо в списку залишилося настільки мало вершин, що якщо їх все додати в одну з вихідних вершин, то в ній виявиться minNumOfEntries вершин, то так і робиться; повернення з функції
- якщо в якійсь з вершин вже набраний максимум нащадків, то залишок додається в протилежну; повернення
- 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 вставляють.

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 вони й повертаються.

Одним з шляхів покращення розбиття простору є використання алгоритмів кластеризації. Цей варіант дерев має назву cR-дерево («c» означає clustered). В таких деревах для розділення вузлів використовуються алгоритми кластеризації, наприклад k-means.

У найгіршому випадку ефективність R-дерев є вкрай низькою: складність як вставки, так і пошуку є лінійною, тобто $O(n)$. 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.

2.3 Особливості індексування рухомих об’єктів

Швидкий і постійний прогрес у системах позиціонування, бездротових комунікаційних технологіях та електроніці загалом робить широко застосованим відстеження та запис мінливих положень об’єктів, здатних до безперервного руху. Безперервний рух ставить перед технологією баз даних нові виклики. У звичайних базах даних передбачається, що дані залишаються постійними, якщо вони явно не змінені. Фіксація безперервного руху з цим припущенням призведе до виконання дуже частих оновлень або запису застарілих, неточних даних, жодне з яких не є привабливою альтернативою.

Треба прийняти іншу тактику. Безперервний рух слід фіксувати безпосередньо, щоб просте просування часу не вимагало явних оновлень. Іншими словами, замість збереження просто позицій слід зберігати функції руху, які виражають положення об’єктів. Тоді оновлення будуть необхідні лише тоді, коли змінюються параметри цих функцій.

Найпростішою моделлю руху об’єктів є лінійних рух, тоді параметрами функції руху є вектор позиції та швидкості об’єкта на момент надсилання цих параметрів до бази даних.

Головною проблемою, яка має бути вирішена, щоб підтримувати додатки, що включають безперервний рух, є індексація поточних і очікуваних майбутніх положень рухомих об'єктів. Є й інша проблема:

індексація історій або траєкторій положень рухомих об’єктів, проте вона виходить за межі поточного дослідження.

Одним з підходів до вирішення задачі індексації об’єктів, які рухаються, є використання R-дерев, параметризованих параметрами часу (TPR-дерево, time-parameterized R-tree), яке ефективно індексує поточні та очікувані майбутні положення рухомих точкових об’єктів (або скорочено «рухомих точок»). Цей підхід можна розглядати як розширення (чи узагальнення) R*-дерева [9], [10], [11], [12].

Можна виділити декілька можливих підходів до індексації майбутніх лінійних траєкторій рухомих точок. По-перше, підходи можуть відрізнятися залежно від простору, який вони індексують. Якщо припустити, що об’єкти рухаються в d-вимірному просторі ($d = 1; 2; 3$), їхні майбутні траєкторії можуть бути проіндексовані як лінії в $(d + 1)$-вимірному просторі [11]. Як альтернатива, можна відобразити траєкторії на точки у просторі вищої розмірності, які потім індексуються [10]. В такому випадку запити також повинні бути перетворені, щоб відповідати перетворенню даних. Ще однією альтернативою є індексування даних у їх рідному, d-вимірному просторі, що можливо шляхом параметризації структури індексу за допомогою векторів швидкості, що дозволяє «переглядати» індекс у будь-який майбутній час. TPR-дерево базується на цій, останній альтернативі, оскільки відсутність трансформацій (відображень на точки у просторі вищої розмірності) дає досить інтуїтивно зрозумілу техніку індексування.

Іншим класифікуючим фактором є те, чи індекс розбиває дані (як це роблять R-дерева) чи він розбиває простір, в якому вони знаходяться (як це роблять Quadtrees). Під час індексування даних у їх рідному просторі більш підходящим є індекс на основі розділення саме даних.

По-третє, індекси можуть відрізнятися за ступенем реплікації даних, яку вони передбачають. Реплікація може покращити продуктивність запитів, але також може негативно вплинути на продуктивність оновлення. TPR-tree не використовує реплікацію.

По-четверте, ми можемо розрізняти підходи залежно від того, потребують вони періодичної перебудови індексу чи ні. Деякі підходи (наприклад, [11]) використовують окремі індекси, які функціональні лише протягом певного періоду часу. У цих підходах новий індекс повинен бути наданий до того, як його попередник перестане працювати. Інші підходи можуть використовувати індекс, який в принципі залишається функціональним невизначений час [10], але який може бути оптимізований для певного часового горизонту та, можливо, погіршується з плином часу. TPR-дерево належить до останньої категорії.

У TPR-дереві обмежувальні прямокутники в дереві є функціями часу, як і рухомі точки, що індексуються. Інтуїтивно зрозуміло, що обмежувальні прямокутники можуть безперервно стежити за точками даних або іншими прямокутниками, які переміщуються. Як і R-дерева, новий індекс здатний індексувати точки в одно-, дво- та тривимірному просторі. Крім того, принципи нового індексу можна поширити на неточкові об’єкти.

Більшість робіт з проблеми, яка розглядається, зосереджені на точках, що рухаються в одновимірному просторі.

Тайєб та ін. [11] використовують PMR-Quadtrees для індексації майбутніх лінійних траєкторій одновимірних рухомих точкових об’єктів як відрізків ліній у (x, t)-просторі. Сегменти охоплюють інтервал часу, який починається в поточний час і продовжує H одиниць часу в майбутнє. Дерево закінчується після U одиниць часу, і нове дерево має бути доступним для запиту. Цей підхід забезпечує значну реплікацію даних в індексі — сегмент лінії зазвичай зберігається в кількох вузлах.

Колліос та ін. [10] використовують подвійне перетворення даних, де лінія $x = x(t_{ref}) + v * (t - t_{ref})$ перетворюється на точку $(x(t_{ref}), v)$, що дозволяє використовувати регулярні просторові індекси. Стверджується, що індекси, засновані на Kd-деревах, добре підходять для цієї проблеми, оскільки вони найкраще відповідають формам (перетворених) запитів до даних. Автори припускають (але не досліджують детально), що цей підхід

можна розширити до двох та більше вимірів. Вони також встановили теоретичні нижні межі для цієї проблеми індексування, припускаючи статичний набір даних і $H = \infty$. Дозволяючи індексу використовувати лінійний простір, пошукові запити мають часову складність

$$O\left(n^{(2d-1)/2d} + k\right) \tag{2.1}$$

де d – кількість вимірів простору, де переміщуються об’єкти;

n – кількість блоків даних;

k – розмір відповіді на запит у блоках.

Щоб досягти цього обмеження, можна використовувати версію дерев розділів зовнішньої пам’яті. Стверджується, що, незважаючи на хороші асимптотичні межі продуктивності, дерева розділення не є практичними через великі постійні чинники.

Баш та ін. пропонують так звані кінетичні структури даних основної пам'яті для мобільних об'єктів. Ідея полягає в тому, щоб запланувати майбутні події, які оновлюють структуру даних, щоб підтримувати необхідні інваріанти. Агарвал та ін. застосувати ці ідеї до зовнішніх дерев діапазонів. Їхній підхід, можливо, може бути застосований до R-дерев або R-дерев із часовими параметрами, де події виправлятимуть MBR, хоча незрозуміло, як боротися з майбутніми запитами, які надходять у нехронологічному порядку. Агарвал та ін. розглядають нехронологічні запити за допомогою методів часткової стійкості, а також показують, як поєднувати дерева кінетичного діапазону з деревами розділів, щоб досягти компромісу між кількістю кінетичних подій і продуктивністю запиту.

Проблема індексації рухомих точок пов'язана з проблемою індексації відносних часових даних. GRtree – це індекс на основі R-дерева для відносних бітемпоральних даних. Комбінація дійсних і транзакційних інтервалів часу з кінцевими часами, пов’язаними з безперервним прогресом поточного часу, призводить до регіонів, які ростуть, хоча й обмежено. Ідея

цього індексу полягає в тому, щоб пристосуватись до зростаючих регіонів даних шляхом введення обмежувальних регіонів, які також ростуть. Зокрема, обмежувальні області параметризовані за часом, і їхні межі обчислюються кожного разу, коли надсилається запит.

RST-дерево є просторово-часовим індексом, який індексує історії положень об’єктів. Передбачається, що позиції залишаються постійними між явними оновленнями індексів, а їх історії фіксуються шляхом зв’язування з ними дійсних інтервалів часу та часу транзакцій, які можуть бути відносними. Безперервність, таким чином, випливає з часових аспектів, а не просторових, і таке індексування схоже на індексування в GR-дереві.

3 ІНДЕКСАЦІЯ РУХОМИХ ОБ'ЄКТІВ ЗА ДОПОМОГОЮ TPR-ДЕРЕВ

3.1 Використання R-дерев для індексації рухомих об'єктів

Оскільки рухомі точкові об'єкти, які розглядаються, знаходяться у багатовимірному (зазвичай, $d = 2$, хоча можливо й у 3D) просторі, то для їхнього індексування доцільно використовувати розглянуті вище R-дерева. При цьому рухомість цих об'єктів може суттєво знизити ефективність їхнього індексування.

Розглянемо приклад (рис.3.1). Верхня ліва діаграма рисунку 3.1(а) показує позиції та вектори швидкостей 7 точкових об'єктів у момент часу $t = 0$.

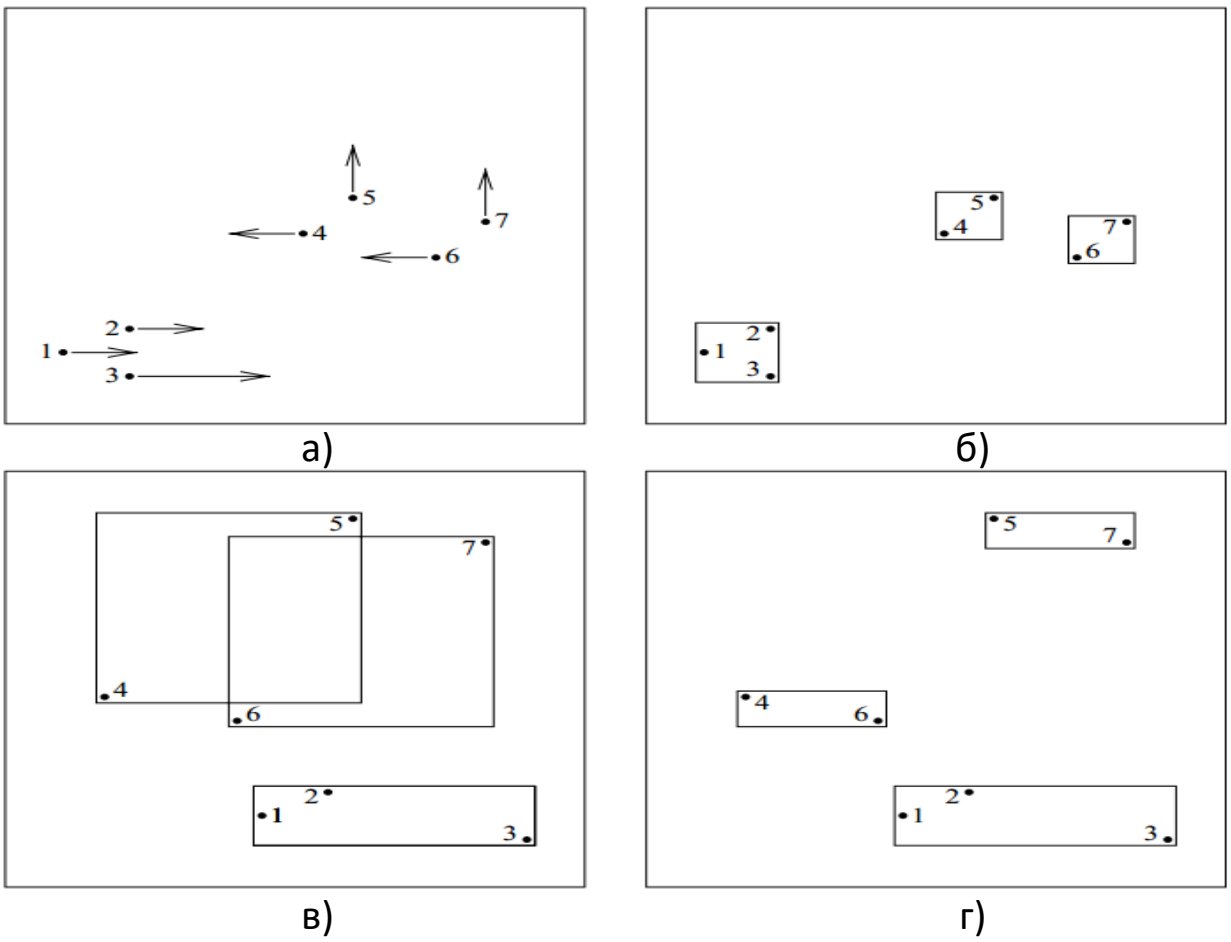


Рисунок 3.1 – Приклад групування рухомих об'єктів у R-дерево

Нехай ми створюємо R-дерево в момент часу $t = 0$. На рисунку 3.1(б) показано можливе групування об’єктів у вузли. Легко бачити, що перекрить немає, периметр кожного вузла (тобто його MBR – мінімального обмежуючого прямокутника) є мінімальним, тобто таке групування видається вдалим. Однак, хоча це дійсно так у поточний час, рух об’єктів може негативно вплинути на ефективність структури даних.

На рисунку 3.1.(в) показано розташування об’єктів та відповідні MBR вузлів у наступний момент часу ($t = 3$). Об’єкти, що належать до одного вузла (наприклад, об’єкти 4 і 5), спочатку були поруч, але різні напрямки їх руху призводять до того, що їхні позиції швидко розходяться і, отже, MBR вузла зростає. В результаті ми бачимо суттєве перекриття вузлів (4-5) та (6-7).

Зрозуміло, що з точки зору ефективності запитів, було б краще перерозподілити об’єкти іншим чином (4-6 та 5-7), як показано на рисунку 3.1(г). При цьому слід розуміти, що це неможливо було зробити заздалегідь (в момент часу 0), оскільки на той момент таке групування було б гіршим, ніж застосоване (рис.3.1(б)).

Можна зробити висновок, що групування об’єктів у вузли має бути сінхронізовано з надходженням запиту.

Позначимо положення об’єкта в d-вимірному просторі в момент часу t як $x(t) = (x_1(t), x_2(t), \dots, x_d(t))$. Найпростіший спосіб змоделювати рух такого об’єкта – це лінійна функція часу, яка визначається двома параметрами. Перший – це опорне положення $x(t_{ref})$, тобто положення об’єкта в момент часу t_{ref} (початковий, чи опорний момент часу). Другим параметром є вектор швидкості об’єкта $v = (v_1, v_2, \dots, v_d)$. Таким чином,

$$x(t) = x(t_{ref}) + v \cdot (t - t_{ref}). \qquad (3.1)$$

Моделювання положень рухомих об’єктів як функції часу не тільки дає змогу робити попередні прогнози майбутнього, але й вирішує проблему

частих оновлень, які в іншому випадку були б потрібні для наближення безперервного руху в традиційних умовах. Наприклад, об’єкти можуть повідомляти про своє положення та вектори швидкості, якщо їхнє фактичне положення відхиляється від того, що вони раніше повідомляли, на певний поріг. Вибір частоти оновлення залежить від типу руху, бажаної точності та технічних обмежень.

Опорне положення та швидкість можна використовувати не лише для запису майбутніх траєкторій рухомих точок, але й для представлення координат обмежувальних прямокутників в індексі як функцій часу.

Структура даних, яка реалізує вищеописану концепцію, має назву TPR-дерево (time-parameterized R-tree, тобто R-дерево, параметризоване позначками часу) [9]. Воно ефективно індексує поточні та очікувані майбутні положення рухомих точкових об’єктів.

Розглянемо види запитів, які має підтримувати TPR-дерево.

Запити, які підтримує індекс, отримують усі точки з позиціями в межах указаних регіонів. Ми розрізняємо три види залежно від регіонів, які вони визначають. d-вимірний прямокутник R (гіперпрямокутник) задається його проекціями $[q_1^L; q_1^H], [q_2^L; q_2^H], \dots, [q_d^L; q_d^H]$ на координатні осі (де $q_j^L \leq q_j^H$ – нижня та верхня межі по координаті j відповідно). Розглянемо три d-вимірних прямокутника R, R^L, R^H в моменти часу t, t^L, t^H ($t^L \leq t^H$):

– запит часового зрізу: $Q_1(R, t)$ визначає гіперпрямокутник R, у момент часу t.

– віконний (за часом) зріз: $Q_2(R, t^L, t^H)$ визначає гіперпрямокутник R, який покриває часовий інтервал $[t^L, t^H]$. Іншими словами, цей запит стосується точок з траєкторіями в (x, t)-просторі, які перетинають $(d+1)$-вимірний гіперпрямокутник $([q_1^L; q_1^H], [q_2^L; q_2^H], \dots, [q_d^L; q_d^H], [t^L; t^H])$.

– пересувний зріз: $Q_3(R^L, R^H, t^L, t^H)$ визначає $(d+1)$-вимірну трапецію, отриману шляхом з’єднання d-вимірного прямокутника R^L у момент часу t^L з d-вимірним прямокутником R^H у момент часу t^H.

Легко бачити, що запит другого типу (Q_2) узагальнює запит Q_1 і сам є частковим випадком запиту третього типу. Щоб проілюструвати типи запитів, розглянемо одновимірний набір даних на рисунку 3.2. Запити Q_0 та Q_1 є запитами часового зрізу, Q_2 є віконним зрізом, а Q_3 є пересувним запитом.

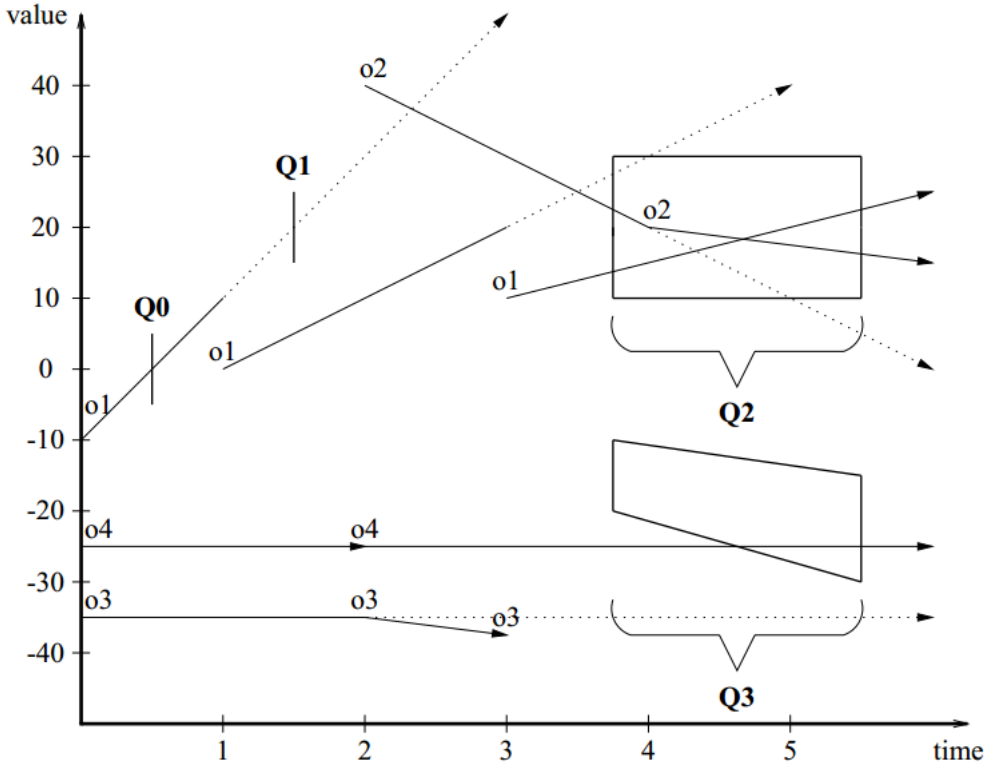


Рисунок 3.2 – Три вида запитів щодо рухомих об’єктів

Нехай $t(Q)$ позначає час надходження запиту Q. Зрозуміло, що базова позиція (x) та вектор швидкості (v), об’єкта, який бачить запит Q, залежать від $t(Q)$, оскільки об’єкти оновлюють свої параметри з часом. Розглянемо об'єкт $o1$: його рух описується однією траєкторією для запитів з $t(Q) < 1$, іншою – для запитів з $1 \leq t(Q) < 3$ та третьою траєкторією для запитів з $t(Q) \geq 3$. Наприклад, відповіддю на запит Q_0 є $o1$, а відповіддю на Q_1 є $\emptyset$, бо жоден з об’єктів не відповідає вимогам цього запиту. А от якщо б позиція та швидкість $o1$ не оновилися в момент $t = 1$, то цей об’єкт задовільнив би

запит Q_1.

Цей приклад показує, що запити щодо далекого майбутнього не мають великої цінності, оскільки позиції, передбачені під час запиту, стають усе менш точними. Тому передбачається, що програми реального світу надсилають запити, зосереджені в якомусь обмеженому часовому вікні, починаючи з поточного часу.

Позначимо t_l – час створення/завантаження індексу. Актуальність індексів характеризується трьома параметрами:

– тривалість запиту (W) показує як далеко запити можуть «заглядати» в майбутнє. Таким чином, $t(Q) \leq t < t(Q) + W$ для запитів типу 1 та $t(Q) \leq t^L \leq t^H < t(Q) + W$ для запитів типів 2 і 3;

– період оновлення індексу (U): інтервал часу, протягом якого індекс використовуватиметься для запитів. Таким чином, $t_l \leq t(Q) \leq t_l + U$;

– часовий горизонт (H): довжина інтервалу часу, якому можуть належати значення моментів часу t, t^L, t^H, указані в запитах. Часовий горизонт для індексу – це час використання індексу плюс вікно запиту: $H = U + W$.

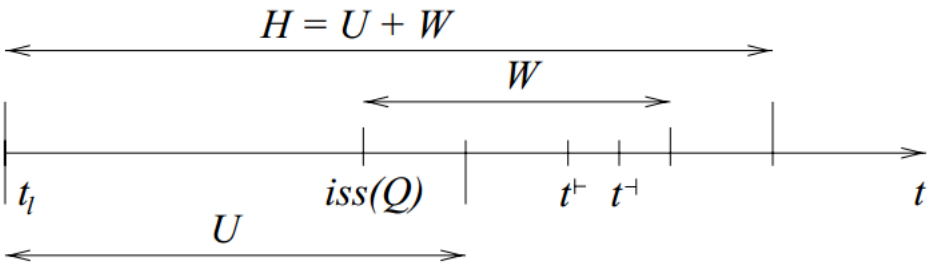


Рисунок 3.3 – Тривалість запиту (W), час використання індексу(U) та часовий горизонт (H)

Таким чином, щойно створений індекс повинен підтримувати запити, які досягають H одиниць часу в майбутньому. Хоча параметр U є більш актуальним для статичних наборів даних і масового завантаження, цей

параметр також корисний у динамічному налаштуванні, де дозволено оновлення індексів. Зокрема, незважаючи на те, що TPR-дерево функціонує весь час після його створення, використання різних значень для параметра U під час вставок впливає на властивості цього дерева.

3.2 Структура індексу TPR-дерева

Як було зазначено вище, TPR-дерево має структуру R-дерева. Записи в листових вузлах містять пари ключ-значення, де ключом є положення рухомої точки, а значенням – покажчики на дані щодо цієї точки. Записи у внутрішніх вузлах є складаються з покажчиків на піддерева та ключів, якими є прямокутники, які обмежують положення всіх рухомих точок або інших обмежувальних прямокутників, що входять у відповідне піддерево.

Положення рухомої точки представлено її початковою позицією та вектором швидкості, тобто парою (x, v), виміреною в момент створення індексу часу t_l.

Щоб зв’язати групу d-вимірних рухомих точок у групу (вузол дерева), використовуються d-вимірні обмежувальні прямокутники, які також є такими, що змінюються у часі, тобто їхні координати (x^L, x^H, y^L, y^H) є функціями часу. Параметризований за часом обмежувальний прямокутник обмежує всі точки або прямокутники цього вузла в будь-який час не раніше t_l.

Отже, структура вузлів TPR-дерева (листових та нелистових) має вигляд, показаний на рисунку 3.4 (літера “L” позначає low, а літера “H” – high).

Існує компроміс між тим, наскільки “міцно” обмежуючий прямокутник обмежує рухомі точки (або прямокутники) та часом, витраченим на його створення, тобто періодом оновлення індексу (U).

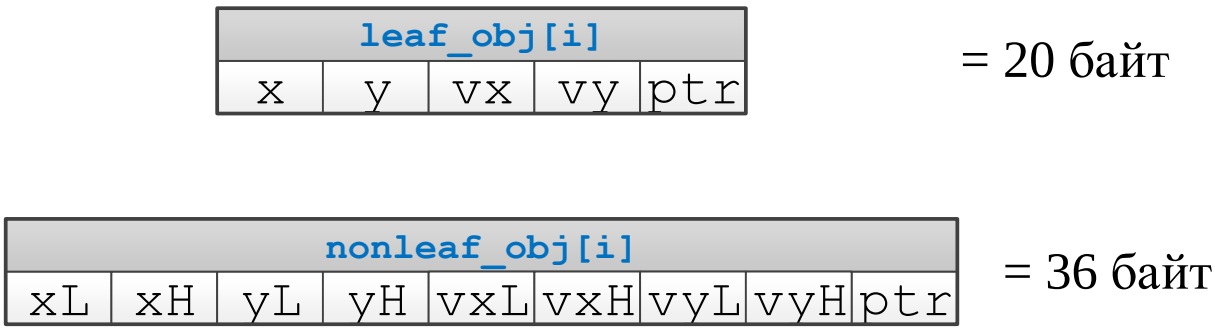


Рисунок 3.4 – Структура вузлів TPR-дерева

Замість використання справжніх, завжди мінімальних обмежувальних прямокутників, TPR-дерево використовує так звані консервативні обмежувальні прямокутники, які є мінімальними на момент створення, але, можливо (і швидше за все!) не пізніше. Нижня межа консервативного інтервалу встановлюється так, щоб рухатися з мінімальною швидкістю замкнутих точок, тоді як верхня межа встановлюється так, щоб рухатися з максимальною швидкістю замкнутих точок (швидкості від'ємні або додатні, в залежності від напрямку). Це гарантує, що консервативні обмежувальні інтервали справді є обмежувальними для всіх моментів часу, які розглядаються.

Рисунок 3.5 ілюструє консервативні обмежувальні інтервали. Вузол складається з двох точок A і B, що рухаються назустріч одна одній. Кожна з цих точок відіграє роль межі мінімального обмежувального інтервалу в певний момент часу. Ліва межа консервативного інтервалу починається в положенні об'єкта A в момент часу 0 та рухається вліво зі швидкістю `B.vx` об'єкта B, а права межа починається в об'єкті B в момент часу 0 та рухається вправо зі швидкістю `A.vx` об'єкта A. Відповідно, верня межа рухається вверх зі швидкістю `B.vy`, а нижня – вниз зі швидкістю `A.vy`.

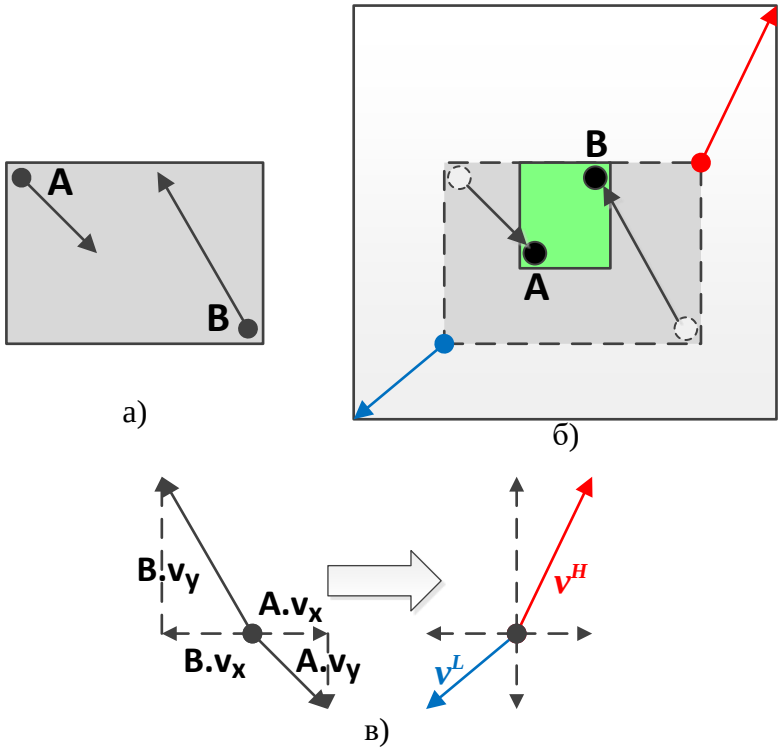


Рисунок 3.5 – Консервативні інтервали: а) вузол в початковий момент часу; б) вузол в момент часу $t = 1$; в) формування швидкостей зсуву меж вікна

Легко бачити, що консервативні обмежувальні інтервали ніколи не звужуються. У найкращому випадку, коли всі замкнуті точки мають однаковий вектор швидкості, консервативний обмежувальний інтервал має постійний розмір, проте він може рухатися.

Дотримуючись представлення рухомих точок, обмежуючий інтервал вузла є функцією від положення та швидкостей точкових об’єктів (x^L, x^H, v^L, v^H):

$$[x^L(t); x^H(t)] = [x^L(t_l) + v^L(t_l) \cdot (t - t_l);\ x^H(t_l) + v^H(t_l) \cdot (t - t_l)] \qquad (3.2)$$

де

$$\begin{aligned} x^L &= \min_i \{obj[i].x\}, \quad x^H = \max_i \{obj[i].x\}, \\ v^L &= \min_i \{obj[i].v\}, \quad v^H = \max_i \{obj[i].v\}. \end{aligned} \qquad (3.3)$$

Прямокутники (3.2)-(3.3) задають обмеження для моментів часу $t \geq t_l$. Оскільки ці прямокутники ніколи не зменшуються, але можуть занадто сильно збільшуватися, бажано мати можливість час від часу їх коригувати.

Оскільки запит охоплює лише моменти часу, які перевищують або дорівнюють поточний час, то доцільно коригувати обмежувальні прямокутники кожного разу, коли будь-яка з рухомих точок або прямокутників, які вони зв'язують, оновлюється. Наведені нижче формули визначають коригування обмежувальних прямокутників, які можна внести під час оновлення:

$$\begin{aligned} x^L &= \min_i \left\{ obj[i].x(t_{upd}) \right\} - v^L \cdot (t_{upd} - t_l), \\ x^H &= \max_i \left\{ obj[i].x(t_{upd}) \right\} - v^H \cdot (t_{upd} - t_l), \end{aligned} \tag{3.4}$$

де t_{upd} – це час оновлення.

Формули (3.4) застосовуються для як точкових об'єктів (листя дерева), так і прямокутних вузлів.

Слід враховувати, що складові формул (3.4) можуть дуже сильно відрізнятися одна від одної, тому необхідно приділяти особливу увагу управлінню помилками округлення, які можуть виникнути в арифметиці з плаваючою точкою кінцевої точності.

3.3 Пошукові запити, вставка та видалення точок у TPR-дереві

Відповідь на запит часового зрізу відбувається як для звичайного R-дерева, єдина відмінність полягає в тому, що для перевірки перетину вікна запиту з обмежуючими прямокутниками вузла координати останніх обчислюються на момент часу t_q, указаний у запиті. Таким чином, обмежувальний інтервал (рис.3.6), заданий параметрами (x^L, x^H, v^L, v^H), повністю задовільняє запиту $\left([q^L, q^H], t_q\right)$ тоді і тільки тоді, коли

$$\left(x^L + v^L \cdot (t_q - t_l) \le q^H\right) \text{and} \left(x^H + v^H \cdot (t_q - t_l) \ge q^L\right). \qquad (3.5)$$

Щоб відповісти на віконні запити, необхідно перевірити, чи перетинається трапеція запиту з трапецією, утвореною межами обмежуючого прямокутника на часовому інтервалі запиту. Графічна іллюстрація запитів (для випадку однієї просторової координати) наведена на рисунку 3.6.

Для більшої кількості вимірювань можна використовувати загальні тести перетину двох багатогранників, проте існує простіший і ефективніший алгоритм, описаний в [9], [12].

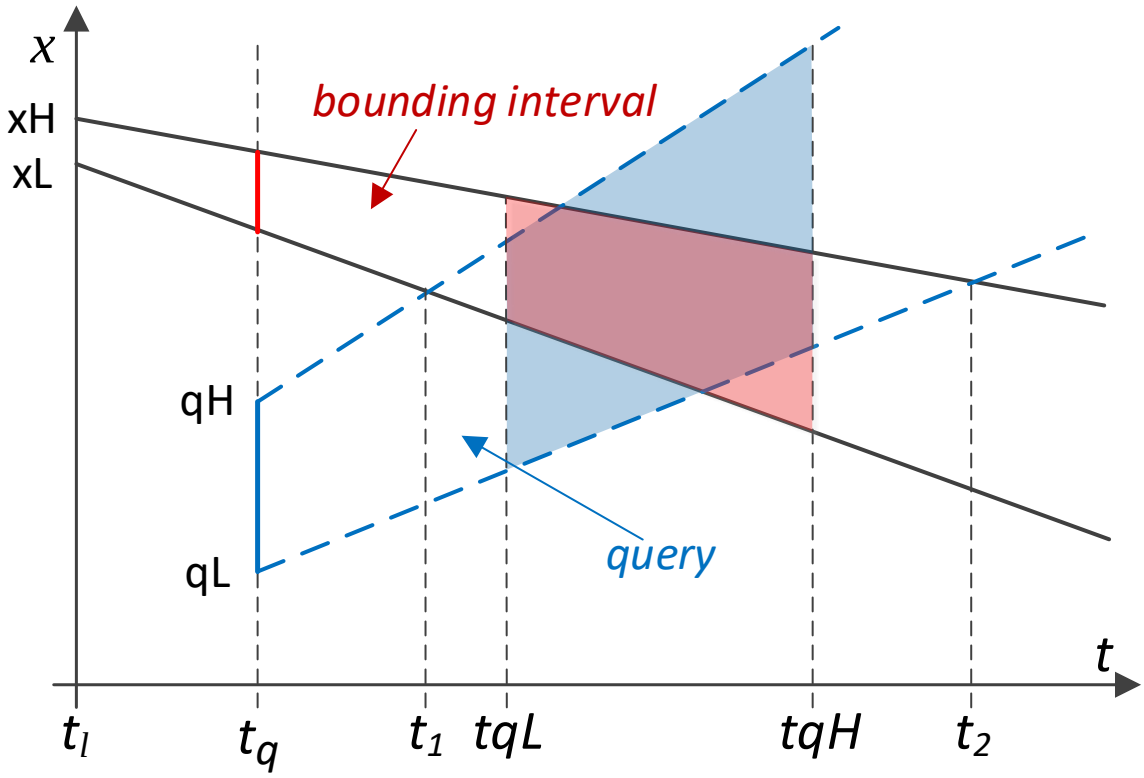


Рисунок 3.6 – Графічна іллюстрація запитів
для одновимірного випадку

Вставка рухомих об’єктів у дерево, так само як і подальше групування цих об’єктів у вузли, мають бути такими, щоб забезпечити максимальну ефективність запитів на протязі горизонту планування H. Для цього необхідно визначити принципи або евристики, які застосовувані як до

динамічних вставок, так і до групового завантаження, а також до будь-якої кількості вимірів.

Якщо H близьке до нуля, то дерево може просто використовувати існуючі алгоритми вставки та масового завантаження R-дерева. Рух точкових об'єктів і зростання обмежувальних прямокутників стають неважливими; мають значення лише їх початкові позиції та розміри. Навпаки, коли H велике, групування рухомих точок відповідно до їхніх векторів швидкостей має суттєве значення. Бажано, щоб обмежувальні прямокутники завжди були якомога меншими на протязі інтервалу $[t_l;\ t_l + H]$ де t_l (time_loading) – момент завантаження індексу. Важливим аспектом у досягненні цього є збереження низьких темпів зростання обмежувальних прямокутників і, отже, значень їхніх «протяжностей швидкості».

Це призводить до наступного підходу: алгоритм вставки TPR-дерева має бути майже таким самим, як і для R-дерева з однією відмінністю: У R-деревах критерієм обрання вузла, до якого буде додана нова точка, є мінімізація статичної площі прямокутника вузла (чи площі або периметру перетину його з іншими). Натомість у TPR-дереві таким критерієм є мінімізація об'єму перетинів 3D-трапеції (зрізаної призми), часовим зрізом якої є прямокутник вузла, на протязі часу від t_l до $t_l + H$ з призмами, що відповідають іншим (сусіднім) вузлам. При цьому алгоритм який обчислює цей об'єм, є розширенням алгоритму для перевірки, чи такі прямокутники перекриваються. У кожну точку часу, коли прямокутники перетинаються, область перетину є прямокутником, і в кожному вимірі верхня (нижня) межа цього прямокутника визначається верхньою (нижньою) межею одного з двох прямокутників, що перетинаються.

Алгоритм таким чином ділить часовий інтервал, повернутий алгоритмом перевірки перекриття, на послідовні часові інтервали так, що протягом кожного з них перетин визначається параметризованим часом прямокутником. Потім ці площі сумуються.

Нарешті, на додаток до сортування вздовж просторових розмірів, алгоритм розбиття розширено, щоб врахувати також сортування вздовж вимірювань швидкості, тобто сортування, отримане шляхом сортування за координатами векторів швидкостей. Обґрунтування полягає в тому, що розподіл рухомих точок на основі розмірів швидкості може призвести до обмежувальних прямокутників із меншими «протяжностями швидкості», які, отже, ростуть повільніше.

Видалення в TPR-дереві виконуються так само, як і в R-дереві. Якщо вузол недостатньо заповнений, він видаляється, а його записи вставляються повторно.

3.4 Програмна реалізація та моделювання TPR-дерева

Як було встановлено під час дослідження структури R-дерев, ефективність операцій над ними (зокрема словникових, тобто пошуку, вставлення, видалення), залежить від конфігурації обмежувальних прямокутників. Ефективність дерева буде тим вище, чим менше ці прямокутники перетинаються між собою та чим менше вони перетинаються з вікнами запиту. Типове вікно запиту є збалансованим по вимірам, тобто квадратним, чи наближеним до квадрату. В той же час, у разі послідовного додавання точок обмежувальні прямокутники вузлів можуть мати «полосову структуру» (рис. 3.7). Це зумовлено тим, що точки скануються, чи формуються, не випадковим чином, а послідовно, у двох вкладених циклах (`for x … for y …`). В результаті точки вставляються у дерево відсортованими по координатам, що призводить до лінійного виродження R-дерева. На рис.3.7 продемонстровано зовнішній вигляд обмежуючих прямокутників вузлів для R-дерева, яке складається з 200 точок, рівномірно розподілених на квадраті 100x100, за умов, що послідовність цих точок відсортована: спочатку по x, а при рівних x – по y.

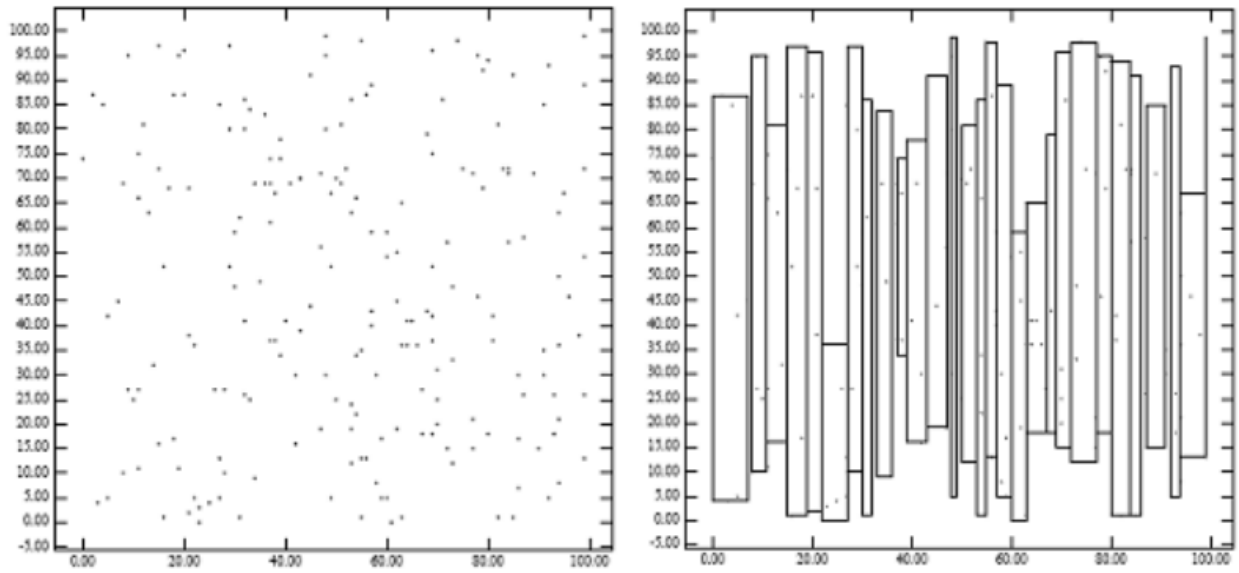


Рисунок 3.7 – Лінійне виродження R-дерев

Групування, схоже на рисунку 3.7, є дуже неефективним. Це зумовлене тим фактом, що типовий запит є квадратним, тобто розміри вікна запиту за обома координатами (x та y) є схожими між собою. Якщо так, то у разі лінійного виродження дерева вікно запиту перетинає багато вузлів дерева, що й є причиною низької ефективності.

Можна сказати, що діє той самий ефект, який знижує ефективність алгоритму швидкого сортування `quickSort`, коли впорядкованість вхідних даних призводить до деградації алгоритму (за швидкістю), а випадковий характер вхідних даних – навпаки є найкращім варіантом.

Логічним виходом з такої ситуації є рандомізація послідовності вхідних даних. Для цього під час створення TPR-дерева використовується процедура масового (групового) завантаження точкових даних, яка передбачає в тому числі й рандомізацію послідовності вхідних даних.

У роботі моделювались R-дерева та TPR-дерева. Досліджувалась ефективність виконання віконних запитів на пошук даних за швидкодією цього процесу.

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 листовий вузол є структурою даних, яка складається з заголовку та m-елементного масиву даних. Елементами масиву є структури з полями `x, y, vx, vy, ptr`, тобто 2D-координати, 2D-швидкості точок та покажчики на дані, асоційовані з цими точками (рис.3.8). Заголовок містить булеву змінну – ознаку того, що вузол є листовим, та може за необхідністю містити додаткову інформацію.

Нелистовий вузол також є структурою даних, яка складається з заголовку та m-елементного масиву даних. Елементами масиву є структури з полями `xL,xH,yL,yH,vxL,vxH,vyL,vyH,ptr`, що описують координати обмежуючого прямокутника вузла (його лівий нижній та правий верхній кути), консервативні швидкості пересування цих кутів та покажчики на відповідні дочірні прямокутники-вузли. Заголовок містить булеву змінну – ознаку того, що вузол є нелистовим, та може містити додаткову інформацію.

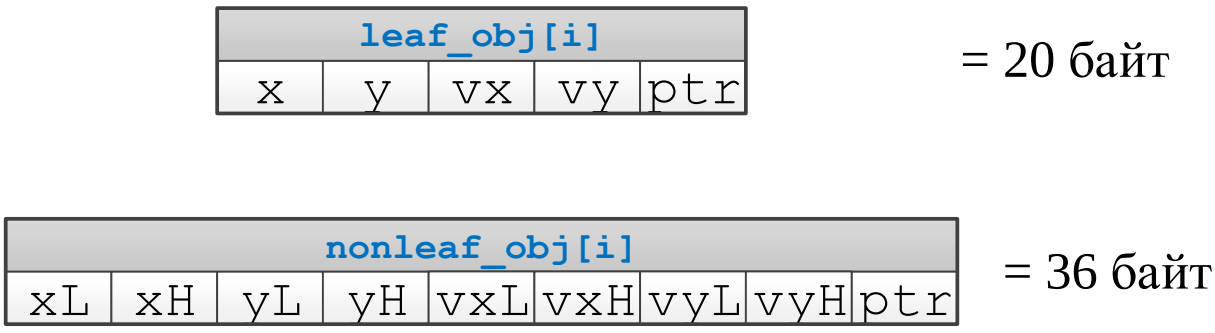


Рисунок 3.8 – Структура вузлів TPR-дерева

Розмір сторінки (та відповідно розмір вузла дерева) встановлено на 4 Кб. Це призводить до $m_L = \lfloor 4096/20 \rfloor = 204$ записів на листовий вузол та $m_{NL} = \lfloor 4096/36 \rfloor = 113$ записів на нелистовий вузол. Як було зазначено у попередньому розділі, фактична кількість записів у вузлах може варіюватись від $m/2$ до m, тобто $102 \leq num_L \leq 204$, $57 \leq num_{NL} \leq 113$.

Використовується сторінковий буфер розміром 200 тис. байт, тобто 50 сторінок, де закріплюється корінь дерева та використовується політика заміни сторінок, які використовувалися найменше. Вузли, змінені під час операції індексування, позначаються як «брудні» в буфері та записуються на диск наприкінці операції або коли їх іншим чином потрібно видалити з буфера.

Моделювались точкові дані, які розміщені на прямокутній (квадратній) області розміром $100\,000 \times 100\,000$ метрів. Кількість точок (листів дерева) складає $N = 200\,000$. Початкові координати точок обираються випадково рівноймовірно. Покоординатні компоненти швидкості обирались також рівноймовірно з ряду $\{-1.5; -1; 0; 1; 1.5\}$ км/хв. Якщо деякий об’єкт досягає межі області, то його відповідна компонента швидкості змінює знак на протилежний (як у більярді).

Період оновлення дерева становить $U = 20$ хв. Тобто в ці моменти точковим об’єктам призначаються інші швидкості (з цього ж ряду), а TPR-дерево повністю переформатовується.

Запити мають тип часового зрізу та надходять у довільні моменти часу з інтенсивністю μ запитів за період оновлення ($\mu \in \{2, 3, 4, 5, 6\}$). Прямокутники запиту є квадратними розміром 5×5 км., їхня позиція на мапі області є випадковою. В середньому кожному запиту має задовільняти 500 точкових об’єктів.

Альтернативою TPR-дереву було звичайне R-дерево, в якому точкові об’єкти групуються у вузли тільки на підставі поточних координат, тобто без врахування швидкостей.

Досліджувалась кількість вузлів, що переглядаються, в залежності від інтенсивності запитів. Результати дослідження представлені в таблиці 3.1, а відповідна графічна візуалізація – на рисунку 3.9.

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.

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

Інтенсивність запитів (кількість запитів за період оновлення), μ	Кількість вузлів, що переглядаються	
	TPR-tree	R-tree
2	6,64	17,17
3	6,01	13,67
4	4,83	9,41
5	4,25	6,39
6	3,54	4,12

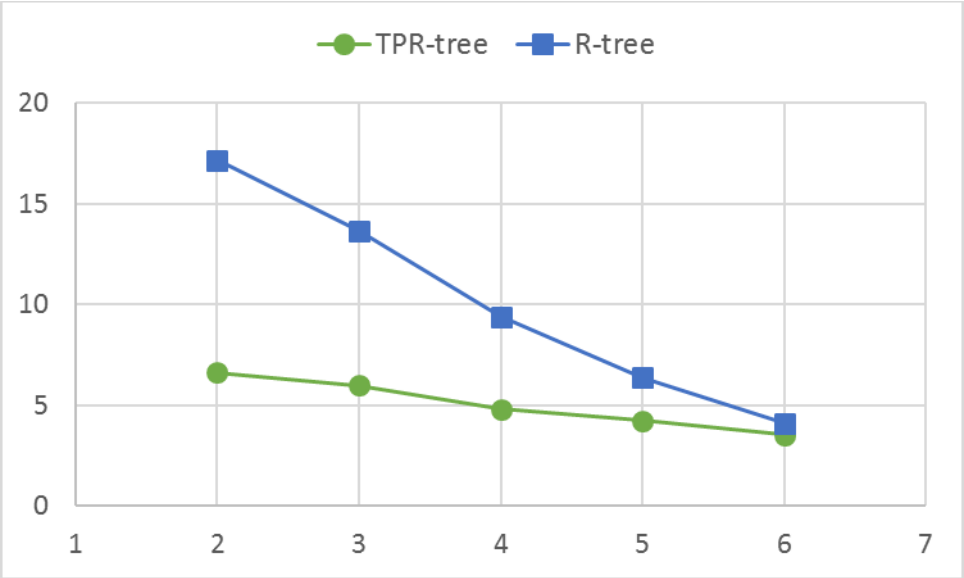


Рисунок 3.9 – Залежність кількості вузлів, які переглядаються, від інтенсивності запитів

Як можна бачити з наведених результатів, TPR-дерево є ефективнішим за звичайне R-дерево. Різниця у ефективності тим більша, чим більше інтервал між запитами (тобто чим менше інтенсивність запитів відносно періоду оновлення структури).

Іншим важливим параметром, який характеризує ефективність структур даних, є відсоток точок, які алгоритм змушений переглядати, але які відкидаються, як такі, що не задовільняють запиту. Відповідні дослідження також були проведені. Згідно з результатами моделювання цей відсоток склав 60% для TPR-дерева та 81% для R-дерева (при інтенсивності запитів $\mu = 4$.

Такі великі значення є природними, оскільки вікно запиту майже завжди перетинає кілька вузлів, а алгоритми пошуку змушені передивлятись всі точки, що знаходяться в «підозрілих» вузлах. Втім, TPR-дерева забезпечують суттєво нижчий відсоток відсіву, ніж R-, що зумовлено якраз врахуванням швидкісних даних точок на етапі формування дерев. Завдяки цьому консервативні розміри вузлів у TPR-деревах значно менші, отже менша й кількість вузлів, що перекриваються з запитом, а отже й кількість точкових об’єктів в них.

ВИСНОВКИ

В роботі досліджувались методи індексування точкових 2D об’єктів, які рухаються. Було проведено аналіз деревовидних структур даних, таких як B та B+ дерев та алгоритмів словникових операцій над ними. Визначено, що для зберігання просторових даних найбільш придатною структурою таких даних є R-дерева. Вони є розширенням B+ дерев на випадок інтервальних просторових даних, отже дуже схожі на них за структурою та алгоритмами словникових операцій (пошук, додавання, видалення).

Згідно з результатами досліджень предметної області, виявлено, що для індексування рухомих об’єктів слід виміряти та зберігати як компоненти ключів не тільки просторові координати точок та кутів обмежуючих прямокутників, а й швидкості цих точок та кутів. Проте, внаслідок руху точок консервативні межі обмежуючих прямокутників вузлів зростають досить швидко. Це призводить до збільшення перекриття вузлів, та, як наслідок, зниження ефективності R-дерев.

В роботі розглянуто та досліджено TPR-дерева. Це така варіація R-дерев, у якій групування точок у прямокутні вузли (з метою зменшення перетинань та просторових розмірів вузлів) відбувається на підставі не лише поточних позицій точок, а й майбутніх, згідно з поточними швидкостями. Це призводить до того, що отримане групування зостається приємним за якістю не лише на момент його створення (як це є для R-дерев), а й протягом заданого горизонта планування у часі (H).

В роботі здійснено програмну реалізацію R- та TPR-дерев та типових пошукових запитів над ними. Проведено порівняльний аналіз цих дерев щодо ефективності індексування. Визначено, що TPR-дерево забезпечує більшу швидкість оброблення запитів за рахунок меншої кількості вузлів, які переглядаються, та меншої кількості точок, які переглядаються. Це зумовлено саме впливом прогнозуючих властивостей TPR-дерев.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 2002. 960 с.

2. Goodrich M.T., Tamassia R. Algorithm Design and Applications. John Wiley & Sons, New York, 2015.

3. Ахо А., Хопкрофт Дж., Ульман Д. Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2003. 384 с.

4. Goodrich M.T., Tamassia R., Mount D. Data Structures and Algorithms in C++. John Wiley & Sons, New York, 2011.

5. Goodrich M.T., Tamassia R., Goldwasser M. Data Structures and Algorithms in Java. John Wiley & Sons, New York, 2014.

6. Roussopoulos N., Leifker D. Direct spatial search on pictorial databases using Packed R-trees. In Proc. of ACM SIGMOD, pages 17–31, Austin, TX, May 1985.

7. Schubert E., Zimek A., Kriegel H. P. Geodetic Distance Queries on R-Trees for Indexing Geographic Data. Advances in Spatial and Temporal Databases. *Lecture Notes in Computer Science*. 2013. Vol. 8098. P. 146.

8. Kamel I., Faloutsos C. Parallel R-Trees. In Proc. of ACM SIGMOD Conf., pages 195–204 San Diego, CA, June 1992.

9. Saltenis S., Jensen C.S., Leutenegger S.T., Lopez M. Indexing the Positions of Continuously Moving Objects. In Proc. ACM SIGMOD. 2000. P. 331–342.

10. Kollios G., Gunopulos D., Tsotras V.J. On Indexing Mobile Objects. In *Proc. of the PODS Conf.* 1999. P. 261–272.

11. Tayeb J., Ulusoy O., Wolfson O. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*. 1998. № 41(3). P. 185–200.

12. Hellerstein J.M., Naughton J.F., Pfeffer A. Generalized Search Trees for Database Systems. In *Proc. of the VLDB Conf.* 1995. P. 562–573.

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);
}
}
}

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;
 }
 }
}
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
	Пояснювальна записка	79 с.
	Інші документи	
	Презентаційні матеріали	17 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Сідельник Д.В.			Індексування об'єктів, що рухаються, на основі B-дерев	Шифр групи	Код напр./спец.
Перевірив		Удовенко С.Г.				СШІздм-21-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					