**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інформаційних систем**

**Алгоритми та складність**

**Лабораторна робота №5**

**«B+ дерево»**

**ЗВІТ**

**Підготував студент**

**2 курсу**

**Групи К29**

**Опанюк Микита**

**Варіант 7**

**2018 рік**

**Опис предметної області :**

*Варіант 7*   
Предметная область : Продуктовый магазин   
Объекты : Категория продукта, Продукт   
Примечание : Продукты  в  магазине сгруппированы  по  категориям. Для  каждой  категории определено  множество продуктов.

**Теоретична частина :**

**B+ дерево** — тип дерева, яке подає відсортовані дані в вигляді, що дозволяє швидке додавання, отримання і видалення записів, кожен з яких ототожнений *ключем*. Це динамічний, багаторівневий індекс, з верхньою та нижньою межами на кількість ключів в кожному сегменті індекса (блоці або вершині). В B+ дереві, на відміну від дерева, всі записи зберігаються на рівні листових вузлів дерева; у внутрішніх вузлах зберігаються лише ключі.

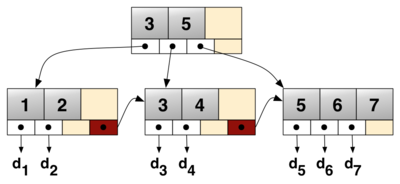
Головне значення B+ дерева — в зберіганні даних для швидкого отримання в блоково-орієнтованих сховищах — особливо, файлових системах. Це здебільшого через те, що на відміну від двійкового дерева пошуку, B+ дерева мають дуже значне розгалуження (зазвичай близько 100 й більше), що зменшує кількість операцій введення-виведення потрібних для знаходження елемента в дереві.

**Загальний огляд.**

Порядок або покажчик галуження *b* для B+ дерева вимірює місткість вершин (тобто кількість дочірніх вершин) для внутрішніх вузлів дерева. Поточна кількість дочірніх вершин, тут позначається як *m*, для внутрішніх вузлів обмежується ⌈ b / 2 ⌉ ≤ m ≤ b . Корінь становить виняток: йому дозволено мати всьог дві дочірні вершини. Наприклад, якщо порядок B+ дерева 7, кожна внутрішня вершина (за винятком кореня) може мати від 4 до 7 дочірніх. Листя не мають дітей, але мають містити від ⌊ b / 2 ⌋ до b − 1. У випадку якщо B+ дерево майже попрожнє, воно містить лише одну листову вершину, тоді корінь буде листом. Такій вершині дозволено містити від 1 до *b* ключів.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип вершини** | **Тип дитини** | **Мін. дочірніх** | **Макс. дочірніх** | **Припустимо b = 7** | **Припустимо b = 100** |
| Корінь (коли це лише одна вершина в дереві) | Ключі | 1 | b | 1 - 7 | 1 - 100 |
| Корінь | Внутрішні або листя | 2 | b | 2 - 7 | 2 - 100 |
| Внутрішя вершина | Внутрішні або листя | ⌈ b / 2 ⌉ | b | 4 - 7 | 50 - 100 |
| Вершина лист | Ключі | ⌊ b / 2 ⌋ | b - 1 | 3 - 6 | 50 - 99 |

*Простий приклад B+ дерева, яке пов'язує ключі 1-7 до значень даних d1-d7 :*

[](https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Bplustree.png)

**Аналіз :**

Простір, необхідний для зберігання дерева **O(n).** Вставка запису потребує **O(logb n)** операційПошук запису потребує **O(logb n)** операцій. Видалення (вже знайденого) запису потребує **O(logb n)** операцій.

**Реалізація алгоритму (мова) :** Алгоритм реалізований з використанням мови програмування C++ з використанням інтегрованого середовища розробки Microsoft Visual Studio.

**Інтерфейс програми :** консоль.

**Використання структур даних :**

**1)** В задачці використовується class market{}; - саме в цьому класі йде основна реалізація предметної області. А саме :

Клас працює з 2 типами структур :

struct type {

string group; - назва різновиду.

vector<product\_element \*> array\_of\_products; - вектор вказівників на продукти вказаного типу.

}; - використовується для збереження різновидів продуктів.

struct product\_element {

string group; - назва різновиду продуктів, до якого відноситься вказаний.

string name; - відповідно ім’я.

double price; - ціна (для інформативності).

int key; - для зручності роботи з B+ деревом.

};

vector<type \*> type\_of\_products; - масив типів продуктів в магазині, де кожен елемент масиву має вказівник на масив самих продуктів.

**vector<type \*> array\_of\_products; - масив продуктів в магазині, це основні дані, з якими працює B+ дерево.**

int check\_is\_product(int pos, string product); - перевіряє наявність вказаного продукту в магазині, де pos – це індекс елемента масива типів продуктів.

int check\_is\_type(string product); - перевіряє наявність відповідно типу продуктів в магазині.

void new\_data(); - додавання нового продукту з перевіркою на наявність типу цього продукту (якщо тип відсутній, то створюється новий), відповідно присутня перевірка того чи існує вже вказаний продукт.

void delete\_data(product\_element \*name); - видаляє з каталогу вказаний продукт (зменшує масив продуктів у відповідного типу)

void save\_changing(); - зберігає всі зміни каталогу, перезаписуючи файл, на якому зберігаються дані що до продуктів.

void read\_data\_from\_file(); - зчитує данні що до продуктів з файлу.

**2)** class B\_plus\_tree {

public:

Bp\_tree\_node \*root; - корінь дерева.

Bp\_tree\_node \*new\_p; - використовується для додавання вершини.

int num\_to\_split; - число, відносно якого відбувається поділ.

product\_element \*null\_node; - використовуеться для побудови

B\_plus\_tree(); - конструктор, в якому також є меню для редагування.

~B\_plus\_tree(); - деструктор.

Bp\_tree\_node \*new\_node(); - виділення пам’яті під нову вершину та її ініціалізація.

Bp\_tree\_node \*find\_data(string name, Bp\_tree\_node \*cur\_element, int &pos\_of\_data, int &pos\_of\_cur); - пошук вершини за назвою продукту.

void shift\_to\_left(Bp\_tree\_node \*cur\_element, int pos); - зсув вліво при видалені

void shift\_to\_right(Bp\_tree\_node \*cur\_element, int pos); - зсув вправо при видалені, розбитті блоку на 2 частини.

void insert(Bp\_tree\_node \*cur\_element, product\_element \*data); вставка елемента

void merge\_blocks(Bp\_tree\_node \*left\_element, Bp\_tree\_node \*right\_element, int pos\_to\_merge); - злиття блоків

bool delete\_data(string data); - видалення елементу за іменем

void split\_leaf(Bp\_tree\_node \*cur\_element); - розбиття елементу, що є листком дерева

void split\_non\_leaf(Bp\_tree\_node \*cur\_element); - розбиття елементу, що не є листком

void redistribution\_left(Bp\_tree\_node \* left\_element, Bp\_tree\_node \*right\_element, int pos\_of\_left); - переросподіл елементів вліво при розбитті блоку.

void redistribution\_right(Bp\_tree\_node \* left\_element, Bp\_tree\_node \*right\_element, int pos\_of\_left); - переросподіл елементів вправо при розбитті блоку

void cout\_tree(Bp\_tree\_node \*cur\_element); - виведення дерева.

};

**Опис алгоритму (загальний):**

Для B+ дерева задаються деякі властивості, на основі яких воно й будується :

- максимальна кількість вершин в кожному блоці (в моєму випадку = 4);

- кількість нащадків блоку = кількості вершин в блоці + 1 вказівник на «сусідній блок» (цей вказівник потрібен для більш швидкого проходження дерева під пошуку елемента);

- Якщо при додаванні нового елементу в дерево в певному блоці отримуємо кількість елементів = 4, то виконуємо процедуру розбиття дерева на 2 блоки, розглядаючи випадки:

1) void B\_plus\_tree::split\_leaf(Bp\_tree\_node \*cur\_element) - в тому випадку, якщо блок – це листок. Тоді простіше, бо відсутні нащадки блока.

2) void B\_plus\_tree::split\_non\_leaf(Bp\_tree\_node \*cur\_element) – якщо не лист, тоді крім того що треба розбити блок, треба ще перерозподілити нащадків між 2 новими блоками.

Крім того саме під час розбиття блоків задається нащадок-вказівник на «сусіда».

**Проте найважчим процесом в цій задачці є видалення елементу.** Тут аналогічно треба розглядати випадки на наявність синів, яка кількість вершин залишиться в блоці після видалення. Відповідно використовуються додаткові функції :

Зсув при видалені елементу (передається блок та позиція, з якої починати зсув.

shift\_to\_left(right\_element, 1); - зсув вліво для уникнення повторення елементу

shift\_to\_right(right\_element, 1); - зсув вправо

Якщо після видалення елементу в блоці залишається незначна кількість елементів, і якщо в сумі з елементами сусіднього блоку не порушується властивість про максимальну кількість елементів в блоці, то відбувається перерозподіл елементів між блоками та злиття цих двох блоків.

redistribution\_left(cur\_element, right\_element, 0); - перерозподіл елементів лівого блоку в правий.

redistribution\_right(left\_element, cur\_element, (pos\_of\_cur - 1)); - перерозподіл елементів правого блоку в лівий.

merge\_blocks(left\_element, cur\_element, pos\_of\_cur); - злиття блоків (злиття блоків виконую в загалом лише в один бік, відповідно перевіривши чи не порушуються умови що до кількості елементів).

**Опис алгоритму (предметна область):**

Ініціалізуємо market.h. Зчитуємо з файлу market.h :

7

candies;white bar;16.99

candies;chocoshaker;20.55

candies;cake;44.55

meat;chicken;45.99

meat;beef;60.5

fruit;orange;27.95

fruit;apple;15.55 Починаємо обробку бази даних, після завершення зберігаємо зміни до файлу і переходимо до побудови B+ дерева.

Меню роботи з самим деревом :

B+ tree is ready to use! Далі йде виведення дерева :

orange white bar

beef cake apple orange chicken white bar chocoshaker

Chose what to do :

1. Add new product.

2. Delete product from tree.

3. Cout tree.

4. Check product in tree.

5. Exit! Приклад роботи :

1 – додаємо новий елемент

Write a group of product:

vegetable

Write a name of product:

potato

Write a price of product:

9.99

New product was added.

Chose what to do :

1. Add new product.

2. Delete product from tree.

3. Cout tree.

4. Check product in tree.

5. Exit!

3 – виводимо дерево

orange white bar

beef cake apple orange **potato** chicken white bar chocoshaker – елемент був доданий до 2 з 3 блоків

Chose what to do :

1. Add new product.

2. Delete product from tree.

3. Cout tree.

4. Check product in tree.

5. Exit!

5 - вихід

**Основні модулі програми :**

**market.h –** обробка файлу з інформацією.

**Alg\_ex\_6.cpp –** основна компіляція програми.

**Висновок : Плюси:** Дозволяє спростити збереження великої кількості даних завдяки дуже значному розгалуженню в дереві порявняно зі звичайним бінарним сбалансованим деревом пошуку.

**Використані джерела :**

https://ru.wikipedia.org/wiki/B%2B-дерево