**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інформаційних систем**

**Алгоритми та складність**

**Лабораторна робота №5**

**«Персистентна множина на основі Червоно-чорного дерева»**

**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 :**

Персистентні структури даних (англ. Persistent data structure) - це структури даних, які при внесенні в них деяких змін зберігають всі свої попередні стани та доступ до цих станів.

Існує декілька рівнів персистентності :

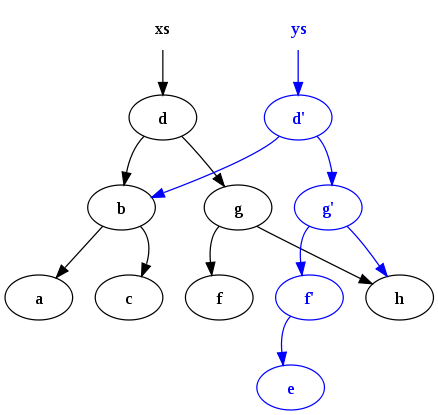
1. Часткова.
2. Повна (саме її я використав для написання задачі).
3. Конфлюентна.
4. Функціональна.

В частково-персистентних структурах даних для кожної версії можна робити запити, але змінити можна тільки останню версію структури даних.  
В повністю персистентних структурах даних можна змінити не тільки останню, але і будь-яку версію структур даних, також для будь-якої версії можна робити запити.  
Конфлюентні структури даних дозволяють об'єднати дві структури даних в одну (дерева пошуку, які можна злити).  
Функціональні структури даних повністю персистентні за визначенням, так як в них забороняються знищувальні присвоєння, тобто будь-якій змінній значення може бути присвоєно лише раз і змінювати значення змінних не можна. Якщо структура даних функціональна, то вона і конфюэнтна, якщо конфлюэнтна, то й повністю персистентна, якщо повністю персистентна, то і частково персистентна. Однак існують структури даних, не функціональні, але конфлюентні.

***Спосіб перетворення структур даних в персистентну множину, використаний в задачці :***

*Метод копіювання шляху*

Нехай є збалансоване дерево пошуку. Всі операції в ньому робляться за O(log(n)), n – кількість вершин дерева, висота дерева h = log(n).  
Нехай необхідно зробити якесь оновлення в цьому збалансованому дереві, наприклад, додати черговий елемент, але при цьому потрібно не втратити старе дерево. Візьмемо вузол, в який потрібно додати нового сина. Замість того щоб просто додавати нового сина, скопіюємо цей вузол, до копії додамо нового сина, також скопіюємо всі вузли аж до кореня, з яких можна досягти перший скопійований вузол разом з усіма покажчиками. Всі вершини, з яких змінений вузол не можна досягти, ми не чіпаємо. Кількість нових вузлів завжди буде порядку логарифма. В результаті маємо доступ до обох версій дерева.

[](https://neerc.ifmo.ru/wiki/images/d/d8/%D0%9A%D0%BE%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%83%D1%82%D0%B8.png)

Так як розглядається збалансоване дерево пошуку, то піднімаючи вершину вгору при балансуванню, потрібно робити копії всіх вершин, які беруть участь у балансуванні дерева, у яких змінилися посилання на дітей. Таких завжди не більше трьох, тому асимптотика O(log(n)) не постраждає. Коли балансування закінчиться, потрібно дійти вгору до кореня, роблячи копії вершин на шляху.  
Цей метод добре працює на стеку, двійкових (декартових, червоно-чорних) деревах. Але в разі перетворення черги в персистентну операція додавання буде дуже дорогою, так як елемент додається в хвіст черги, який можна досягти з усіх інших елементів. Також не вигідно застосовувати цей метод і в разі, коли в структурі даних є посилання на батька.

**Аналіз :** Вартість основних операцій становить в середньому O(log(n)), де n – кількість вершин.

**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).

};

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

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 RB\_tree {

Node \* son; - використовується як нульова вершина в червоно-чорному дереві

vector<Node\*> version\_control; - вектор версій дерева

unsigned int version; - кількість версій

~RB\_tree(); - деструктор

RB\_tree(); - конструктор, побудова першої версії дерева.

Далі ідуть функції роботи з червоно-чорним деревом.

void rotate\_left(Node \*x, unsigned int num\_of\_version); - поворот вліво

void rotate\_right(Node \*x, unsigned int num\_of\_version); - поворот вправо

void insert\_balance(Node \*x, unsigned int num\_of\_version); - балансування після вставки

Node \*insert\_node(product\_element\* data, unsigned int num\_of\_version); - вставка вершини

void delete\_balance(Node \*s, unsigned int num\_of\_version); - балансування після видалення

void delete\_node(Node \*m, unsigned int num\_of\_version); - видалення вершини

Node \*is\_node(string data, unsigned int num\_of\_version); - перевірка на наявність вершини

bool cout\_tree(Node \*root, unsigned int num\_of\_version); - виведення дерева

Node \*copy\_node(Node \*for\_copy, unsigned int version); - копіювання вершини (для нової версії)

bool new\_version(product\_element \*data, unsigned int num\_of\_version); - створення нової версії (або також додавання вершин до старих версій)

bool delete\_version(Node \*current, unsigned int version); - видалення версії

void check\_version(); - перевірка версій

void RBT\_menu(); - меню для роботи з програмою

**Опис алгоритму (загальний):**

1) Персистентна множина побудована на базі сбалансованого червоно-чорного дерева, для якого характерні наступні властивості :

а) Кожна вершина є або червоною, або чорною.

б) Корінь дерева є чорною вершиною.

в) Кожен лист дерева (son) є чорною вершиною.

г) Якщо вершина червона, то обидва його сини чорні.

д) Для кожної вершини всі прості шляхи від нього до листків, що є потомками даної вершини, має одну й ту ж кількість чорних вершин.

2)Node \***insert\_node**(**int** data, **unsigned** **int** num\_of\_version); - функція вставки елементів в дерево. Вставка стандартна, тобто рухаємось від кореня, порівнюючи ключі, до листків, додаємо новий елемент у дерево, після цього перебалансуємо дерево за допомогою функції : **void** **insert\_balance**(Node \*x, **unsigned** **int** num\_of\_version);

3)При балансуванні розглядаються 5 випадків :

а) Чи є вказана вершина корнем дерева (відповідної версії дерева).

б) Чи батько вершини є чорним (в кінці функції це вказується).

в) Чи батько та дядько – червоні.

г) Батько – червоний, а дядько – чорний, крім того вершина є правим сином. (данний випадок зводиться до 5 випадку (д), тому як змінюються між собою позиції батька та сина)

д) Батько – червоний, дядько – чорний, вершина – лівий син. (Це буде поворот на право).

4) Після того, як перша (оригінал дерева) побудовано, можна приступити до версіонного контролю. Створення нової версії відбувається при додаванні нової вершини :

а) Під час пошуку місця встановлення вершини копіюються всі вершини разом із зв’язками, починаючи з корня, за вказаним шляхом а також вершини що можут приймати участь в перебалансуванні копії дерева. Після копіювання нова версія (а саме корінь версії дерева) добавляється в масив.

б) Кожна наступна версії будується на основі попередньої.

в) Можна видалити будь яку версію, але новіші версії дерева не зміняться. Тобто, наприклад : 0 версія – оригінал дерева, 1 версія – копія з новою вершиною А, 2 версія – копія з новою вершиною В, 2 версія може мати в собі вершину А, тоді в цьому випадку при видалені 1 версії, в 2 версії вершина А збережеться.

5) Видалення вершин відбувається таким чином :

а) Функція **void** **delete\_node**(Node \*m, **unsigned** **int** num\_of\_version);

знаходить на заміну вершину, що є нащадком. (Функція копіює вершини, що лежать на шляху до вершини m, а також вершини, які лежать на шляху до вершини, що є заміною, для подальшого її видалення та перебалансування дерева).

В перебалансувані дерева після видалення вершини розглядаються в загальному 6 випадків:

а) Вершина, що є заміною (нехай дана вершина “N” - заміна на новій позиції) – новий корінь. (Нічого не треба робити, властивості дерева зберігаються).

б) Чи брат вершини N червоний (тоді змінюємо колір батька на червоний, виконуємо поворот дерева вліво).

в) Якщо батько і брат вершини N, а також діти брата – чорні (тоді колір брата змінюється на червоний).

г) Брат та його діти – чорні, проте батько – червоний (батько стає чорним, брат стає червоним).

д) Брат – чорний, лівий син брата – червоний, правий син брата – чорний. N – лівий син (лівий син брата = SL, стає батьком брата S, S стає правим сином SL, тобто виконано поворот направо).

е) Брат – чорний, правий син брата – червоний, N – лівий син (виконуємо поворот дерева наліво, перемальовуємо вершини для виконання властивостей дерева).

Функція Delete\_balance перевіряє всі ці умови та відновлює властивості дерева.

**Опис алгоритму (предметна область):**

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

3

candies;white bar;16.99

candies;chocoshaker;20.55

candies;cake;44.55

Починаємо обробку бази даних, після завершення зберігаємо зміни до файлу і переходимо до побудови персистентної множини.

Для початку будується перша версія дерева, що буде складатися з цих 3 вершин. Далі можна, користуючись додаванням, видаленням вершин редагувати першу версію, або ж створювати нові.

Меню :

Main version of tree is ready.

chocoshaker. cake. white bar.

Welcome to version control.

1 - Create new version.

2 - Check version.

3 - Delete version.

0 - exit.

Приклад роботи :

Main version of tree is ready.

chocoshaker. cake. white bar.

Welcome to version control.

1 - Create new version.

2 - Check version.

3 - Delete version.

0 - exit.

2 – перевірити версії

Here is a versions of this tree :

0 – 1 версія : 0

Write a number of version.

0

1.Cout version.

2.Add new node.

3.Delete node.

4.Check node.

1 – виведення дерева

chocoshaker. cake. white bar. – виведення вершин дерева 0 версії

Welcome to version control.

1 - Create new version.

2 - Check version.

3 - Delete version.

0 - exit.

**Основні модулі програми :**

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

**Alg\_ex\_5.cpp –** основна компіляція програми.

**Висновок : Плюси:** Збереження початкової версії дерева; часткове копіювання вузлів, а саме шляху до нової вершини, а не повністю всього дерева. Редагування (додавання, видалення вершин) будь якої версії не впливає на інші версії. Як результат створення такої множини дозволяє більш зручно та інформаційно працювати зі структурою даних, спрощує редагування дерева. Значно краще працювати з персистентною версією структури, тому що зберігаються всі зміни елементів в структурі, при чому не копіюється вся структура.

**Використані джерела :** https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D0%B5%D1%80%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B5\_%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D1%8B\_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85