**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інформаційних систем**

**Алгоритми та складність**

**Лабораторна робота №7**

**«Фібоначчієва купа»**

**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 (Аналі+з операцій) :**

|  |  |
| --- | --- |
| **Операція** | **Амортизаційний час** |
| Make-Heap | Θ ( 1 ) |
| Insert | Θ ( 1 ) |
| Minimum | Θ ( 1 ) |
| Extract-Min | O ( lg ⁡ n ) |
| Union | Θ ( 1 ) |
| Decrease-Key | Θ ( 1 ) |
| Delete | O ( lg n ) |

**Купа Фібоначчі** — абстрактна структура даних, яка є ефективною реалізацією черги з пріоритетом.

З теоретичної точки зору купи Фібоначчі особливо варто використовувати, коли кількість Extract-Min і Delete операцій мала порівняно з кількістю інших операцій. Наприклад, деякі алгоритми на графах можуть викликати Decrease-Key на кожному ребрі. Для насиченого графу амортизаційний час Θ (1) кожного виклику Decrease-Key складається у велику перевагу в порівнянні з Θ ( lg n ) у найгіршому випадку в бінарній купі. Це можна побачити на прикладі алгоритмів про найкоротші шляхи з одного входу і мінімального кістякового дерева. З практичної точки зору сталий множник прихований у складності алгоритму і складність у програмуванні купи Фібоначчі роблять її менш бажаною, ніж звичайну бінарну або *d*-арну купу для більшості застосувань.

***Представлення Фібоначчієвої кучі***

Купа Фібоначчі являє собою набір дерев, кожне з яких є мінкупою змінної арності з такими властивостями:

1. Вузли дерев відповідають елементам, що зберігаються в черзі.
2. Корені куповпорядкованих дерев поєднані у двобічно зв'язаний список.
3. Зберігається вказівник на корінь дерева, який відповідає елементу з найменшим ключем (варто зауважити, що те, що кожне дерево є мінкупою, гарантує, що цей елемент буде коренем одного з дерев).
4. Для кожного вузла зберігається його ранг (степінь), тобто кількість його дітей, а також чи він позначений (мету позначення ми визначимо пізніше).
5. *Вимога розміру*: якщо вузол *u* має степінь *k*, то піддерево з коренем *u* має щонайменше Fk + 2 вузлів, де Fi - це *i-*те число Фібоначчі, тобто F0 = 0 , F1 = 1 і Fi = Fi − 1 + Fi − 2 для i ≥ 2.

Кожен вузол *x* містить вказівник *x.p* на свого предка і вказівник *x.child* на один із його нащадків. Нащадки зв'язані в циклічний двобічно зв'язаний список. Кожен нащадок *y* має вказівники *y.left* та *y.right.* Якщо вузол є єдиним нащадком, тоді *y = y.left = y.right.* Нащадки елемента можуть перебувати у будь-якому порядку. Використання двобічно зв'язаних списків уможливлює вставляння і видалення елементів за час *O(1),* а також зв'язування двох списків за *O(1).* Також кожен вузол має атрибут *x.degree,* що дорівнює кількості нащадків, і атрибут *x.mark,* що показує, чи втратив вузол нащадка з моменту, як він став нащадком свого поточного предка.

Доступ до купи відбувається через вказівник *H.min,* який вказує на корінь дерева з найменшим ключем. Якщо дерево порожнє, то *H.min* дорівнює NULL.

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

int check\_is\_product(string product); - перевіряє наявність вказаного продукту в магазині.

void new\_data(); - додавання нового продукту з перевіркою того чи існує вже вказаний продукт.

void delete\_data(product\_element \*name); - видаляє з каталогу вказаний продукт.

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.

Для роботи з кучею клас вершин кучі:

class tree\_node { - клас вершин, з яких складається купа

public:

string group; - інформація про групу продуктів

string name; - назва продукту

double price; - ціна

int key; - ключ (для роботи з купою)

int degree; - степінь вершини (для дерев в купі, означає рівень, на якому знаходиться вершина)

bool mark; - перевіряємо наявність синів у вершини

tree\_node \*parent; - батько

vector<tree\_node \*> children; - вектор синів

tree\_node \*left; - вказівник на лівий ключ в купі від данного

tree\_node \*right; - вказівник на правий

tree\_node(product\_element \*data){} - констуктор

};

struct Heap { - стуктура для роботи купою

vector<tree\_node \*> nodes;

int n;

tree\_node \*min;

};

class Fibonacci\_heap { - основний клас для роботи з купою

public:

vector<Heap \*> heaps; - масив куп

Fibonacci\_heap(); **- конструктор, будує першу купу, стягуючи інформацію файла продуктів, перехід до меню роботи з програмою**

~Fibonacci\_heap() {}; **- деструктор, видаляє всі купи**

Heap \*construct(); **- будує нову купу**

int find\_position(Heap \*f\_heap, tree\_node \*x); **- допоміжна функція, передає індекс місця положення серед вершин 0 рівня**

void cut(Heap \*f\_heap, tree\_node \*x, tree\_node \*y); **- допоміжна функція, що використовується при зменшенні значення ключа. Якщо властивість піраміди в дереві порушена, відбувається операція вирізання**

void cascading\_cut(Heap \*f\_heap, tree\_node \*y); **- допоміжна функція, що використовується при зменшенні значення ключа. Каскадне вирізання робить вершину новим коренем, якщо вона була помічена, і рекурсивно піднімається вище, або помічає її і зупиняється. Процес також зупиняється при досягненні кореня. Тобто при каскадному видаленні вершина вирізатиметься, якщо вона перед цим втратила другого сина.**

void heap\_link(Heap \*f\_heap, tree\_node \*x, tree\_node \*y); - Прив’язати y до x: видалити y зі списку коренів та зробити сином x. Поле degree[x] при цьому збільшується, а мітка mark[y], якщо вона була, знімається.

void insert\_data(Heap \*f\_heap, product\_element \*data); - вставка нового продукту в купу

void insert\_node(Heap \*f\_heap, tree\_node \*x); - вставка вершини (використовується при видаленні, перебалансуванні)

void unite\_heaps(int first, int second); - об’єднання 2 куп

void heap\_delete(Heap \*f\_heap); - видалення купи

void delete\_node(Heap \*f\_heap, tree\_node \*x); - видалення вершини, використовується функція пониження значення ключа до мінімального в дереві, видалення мінімального

void consolidate(Heap \*f\_heap); **- допоміжна функція, що після видалення *вершини* консолідує список коренів *H*. Консолідація відбувається виконуючи наступні кроки допоки у списку коренів присутні корені з однаковим степенем.**

1. **Знайти два корені *x, y* з однаковим степенем. Припустимо, без втрати загальності, що *x.key =< y.key.***
2. **Приєднуємо *y* до *x*. Тут ми збільшуємо *x.degree* і очищаємо позначку на *y*.**

tree\_node \*extract\_min(Heap \*f\_heap); - **видалення мінімального елементу**

**Починаємо з видалення *s\** (на нього вказує *H.min*) і додавання усіх його дітей як дерев у колекцію. Тепер, ми продивляємось усю колекцію, знаходимо найменший елемент і оновлюємо *H.min* відповідно.**

**На цьому можна було б і завершити, оскільки ми отримали правильну Фібоначчієву купу. Однак, не складно побачити, що досі описані операції над купою, роблять список коренів довшим і довшим. Отже, проходження через увесь список коренів ставатиме обчислювально дорожчим. Отже, якщо ми вже маємо зробити якусь роботу у будь-якому випадку, то ми виконаємо очищення зараз, щоб уникнути більшої роботи у наступному. Тому, доти доки наявні два дерева з однаковим рангом, скажімо *k*, ми зливаємо ці дерева, щоб отримати дерево рангу *k+1*. Злиття полягає у порівнянні коренів і додавання дерева з більшим коренем як дочірнього до другого кореня. Зауважимо, що оскільки злиття може створити друге дерево рангу *k+1* у колекції, один корінь може взяти участь у кількох злиттях.**

void decrease\_key(Heap \*f\_heap, tree\_node \*x, int data); - **пониження значення ключа вершини в кучі. Детальне пояснення :**

**Коли ми зменшуємо ключ елемента *s*, якщо умови мінкупи все ще задовільняються, то нам більше не потрібно робити нічого. Інакше, ми просто відтинаємо піддерево з коренем в *s* і вставляємо його як нове піддерево у колекцію. Ми порівнюємо новий ключ *s* із попереднім мінімальним елементом і змінюємо вказівник *H.min* відповідно.**

**Таким чином, ми отримуємо щось, що виглядає подібно до бажаної Фібоначчієвої купи. Однак, проблема із простим відтинанням кожного такого *s* полягає в тому, що ми можемо врешті решт порушити вимогу розміру. Отже, щоб полегшити ситуацію, ми впроваджуємо додаткове правило, що коли ми відтинаємо *s* ми перевіряємо, чи не помічений його батьківський елемент. Якщо так, то ми також відтинаємо батьківській елемент і знімаємо позначку з нього. Інакше, ми просто позначаємо батьківський елемент. Перевірка батьківського елемента виконується рекурсивно, тобто, якщо батько батька позначений, ми відтинаємо і його також. Очевидно,що якщо ми відтинаємо корінь, то ми не робимо нічого, тому немає сенсу позначати корінь. Тому, ця потенційно каскадна процедура, завжди завершується.**

};**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 Фібоначчієвої купи.

Fibonacci\_heap:: Fibonacci \_heap() {}; ініціалізуємо зчитані з файлу продукти в кучу. Після чого починається робота з купою :

Меню роботи з самим деревом :

**If file is not empty, reading data from file!**

**1)Add new data.**

**2)Delete data.**

**3)Use this data.**

**3**

**products from file were added to Fibonacci heap.**

**Heaps : Main heap (products from file) is 0 – список куп**

**1.Create another heap.**

**2.Unite 2 selected heaps.**

**3.Delete selected heap.**

**4.Change selected heap.**

**5.Exit.**

**4 – редагувати купу**

**Which heap you want to select ?**

**Index :0 – редагувати купу під індексом 0**

**white bar || chocoshaker || chicken || orange || apple || bzzz ||**

**1.Get min.**

**2.Insert new product.**

**3.Delete product from heap.**

**4.Exctract min.**

**5.Exit.**

**4 – видалити мінімальне (в даному випадку це bzzz)**

**white bar || chocoshaker || chicken || orange || apple ||**

**1.Get min.**

**2.Insert new product.**

**3.Delete product from heap.**

**4.Exctract min.**

**5.Exit.**

**1 – вивести мінімальне**

**530 - ключ**

**Apple - назва**

**Fruit - тип**

**15.55 - ціна**

**white bar || chocoshaker || chicken || orange || apple ||**

**1.Get min.**

**2.Insert new product.**

**3.Delete product from heap.**

**4.Exctract min.**

**5.Exit.**

**5 - вихід**

**Основні модулі програми :**

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

**Alg\_ex\_8.cpp –** main частина.

Fibonacci\_heap.h – сама куча та методи для роботи з нею

**Висновок : Плюси: Вставка нового елементу за О(1), швидке звернення до найменшої вершини в кучі (не треба обходити всі дерева). Об’єднання куп та пониження значення ключа – О(1), що значно краще порівняно з біноміальною купою. Мінуси: Пошук рандомного елементу може зайняти більше часу, ніж в збалансованому дереві.**

**Використані джерела :**

https://uk.wikipedia.org/wiki/Фібоначчієва\_купа

Лекції Оксани Степанівни (Фібоначчієва купа)