**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інтелектуальних програмних систем**

**Алгоритми та складність**

**Завдання № 5**

**Варіант № 3**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2020**

**Умова задачі**

Реалізація персистентної множини на основі червоно-чорного дерева. Час роботи вставки і видалення в найгіршому випадку і  об‘єм  необхідної пам’яті  мають бути О(log  n).

Предметна область: відділ кадрів (варіант 3). Обєкти: відділення фірми, працівники. Маємо множину відділень, у кожному відділенні зберігається множина працівників.

**Теорія**

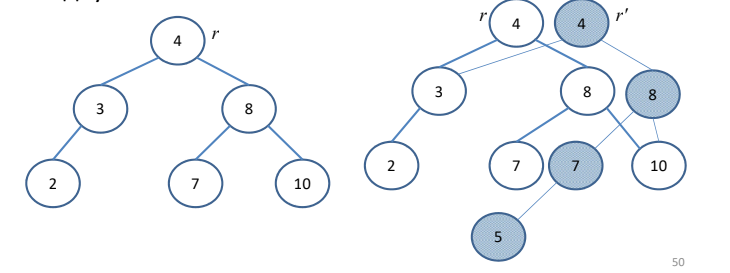
Припустимо, що нам необхідно мати відсортований список працівників фірми, в алфавітному порядку. Додавання, видалення, та пошук елементів для такого списку повинно бути швидким. У такому випадку використовуємо **збалансовані дерева пошуку**. У лабораторній роботі №2 ми розглядали червоно-чорні дерева.

Також розглянемо додаткову умову: в процесі внесення змін у цю множину необхідно зберігати її попередні версії.

Зрозуміло, що можна було б копіювати всю множину після кожної зміни, але це зовсім не ефективно. Тоді будь-яка операція, що змінює дерево, займала би O(n) часу та O(n) пам’яті для копіювання всіх n елементів.

Розглянемо реалізацію персистентної множини з операціями пошуку, видалення та вставки на основі бінарного дерева пошуку.

* Для кожної версії множини зберігається свій корінь.
* Фактично будується копія лише тієї вітки (шляху), де відбулися зміни.

 Приклад вставки елемента у персистентну множину.

У цій роботі реалізована динамічна персистентна множина на основі червоно-чорних дерев. Зберігаються усі версії множини, тому замість r, r’ будемо мати список коренів r0, r1, r2...

**Алгоритми**

Особливістю персистентних множин на основі дерев полягає у тому, що вузли не можуть зберігати вказівник на батька parent.

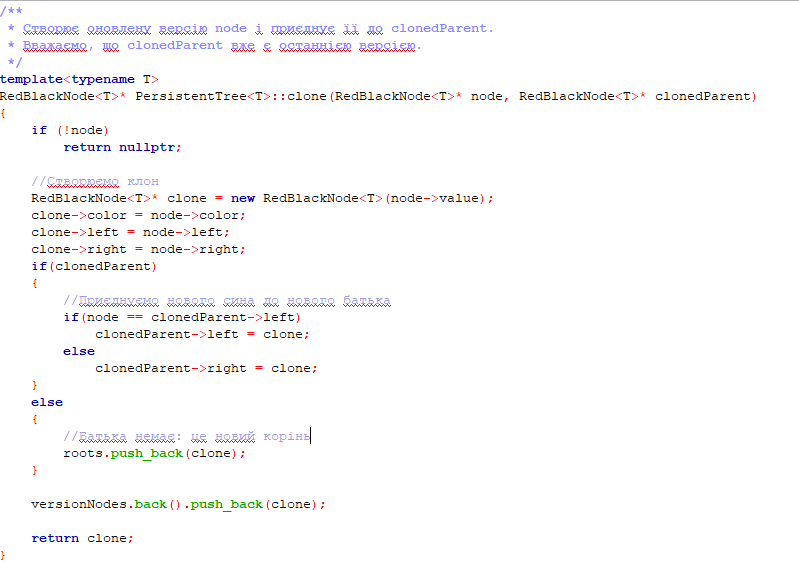
(Припустимо, що збергіається вказівник parent, тоді для нових версій дерева при зміні батька необхідно буде оновити значення parent у вузлі сина. Це би призевло до того, що при будь-якій операції рекурсивно копіюється все дерево, бо при копіюванні вузла-кореня необхідно оновити обох його дітей, а тоді і їх дітей, і так далі)

Тому доведеться модифікувати алгоритми, пов’язані з червоно-чорним деревом, таким чином:

1. Копіюємо елементи вітки (шляху), на якій відбуваються зміни.
2. Операції виконуються без використання parent.

**Власне алгоритми над червоно-чорним деревом ми не будемо аналізувати, оскільки це вже робилося у лабораторній роботі №2. Тому будемо розглядати саме модифікації до алгоритмів.**

Спершу введемо допоміжну функцію clone:



Ця операція створює нову версію вузла, за умови, що є нова версія батька. (тоді до нового батька приєднається новий вузол). Якщо батька немає, вважається, що це корінь, тоді він додається до списку roots.

Також вказівники на всі новостворені вузли версії k зберігаються у масив versionNodes[k]. Це важливо для імплементації деструктора та для операції revert (повернення): аби повернутися до версії k-2, треба видалити усі вузли з versionNodes[k], versionNodes[k-1].

За рахунок цього структура займає на O(n) більше пам’яті (бо на кожний створений вузол є один “зайвий” вказівник). Але оскільки ми прибрали усі вказівники parent, це не є суттєво важливою зміною.

**Додавання елементу**

Як відомо, операція додавання елементу в червоно-чорному дереві проводиться у два етапи: спершу додавання, як у звичайному дереві, потім операція insertFix().

На першому етапі відбувається O(log n) “спусків по дереву” — на цей раз на кожному кроці ми будемо:

* клонувати пройдений вузол
* зберігаємо його у тимчасовий список ancestors.

(обидві операції виконуються за час O(1))

Тоді в кінці першого етапу список ancestors вказуватиме на усіх предків нового вставленого вузла, від безпосереднього батька до кореня. Це нам знадобиться на наступному етапі.

Наступний етап: insertFix(). На цей раз ідемо “вверх” по дереву і виконуємо різні маніпуляції: перефарбовуємо вузли, робимо rotateLeft/rotateRight. (докладніше про це у лабораторній №2).

Операції rotateLeft, rotateRight зазвичай потребують наявності parent. Якщо pivot поміняється місцями з якимось новим node, то в батька pivot повинен змінитися вказівник дитини з pivot на node. Так як у нас немає вказівників на parent, доведеться скористатися списком ancestors.

Перед операціями rotateLeft, rotateRight обов’язково створюється нова версія для “брата” того вузла, який розглядаємо.

При кожному підйомі по дереву ми робимо anestors.pop\_back(), тепер розглядаємо вже батька. pop\_back() для списку займає час O(1).

Взагалі цей модифікований алгоритм **використовує O(log n) додаткової пам’яті** за рахунок того, що ancestors зберігає цілу гілку дерева, а для збалансованого дерева гілка матиме довжину O(log n). **Витрата часу залишається незмінною: O(log n).**

**Видалення елементу**

Аналогічно до додавання: є два етапи: remove, removeFix. Спускаючись по дереву, копіюємо елементи і записуємо їх у ancestors, потім за допомогою ancestors піднімаємось наверх і відновлюємо червоно-чорні властивості. Додаткова пам’ять: O(log n), витрати часу: O(log n).

**Пошук елементу**

Аналогічно до пошуку елементу у звичайному дереві пошуку. Єдина відмінність: починаємо пошук не з єдиного кореня root, а з будь-якого кореня у списку roots.

**Основні модулі програми**

**Мова програмування: C++11.**

У своїй програмі я розбив модулі на дві категорії: ті, що стосуються саме предметної області (HumanResources.\*) і ті, що стосуються персистентної множини (PersistentTree.hpp)

PersistentTree<T> реалізовано як template-клас, тобто він може зберігати обєкти будь-якого класу. Єдина умова: для класу Т має бути реалізований оператор “<”, за допомогою якого будуть сортуватися елементи.

Основні операції над деревом:

* Додавання елемента (void PersistentTree<T>::insert(T value))
  + Створює нову версію дерева, що тепер містить value.
* Пошук елемента (bool PersistentTree<T>::contains(const T& value) const, size\_t versionNum = 0)
  + Повертає true, якщо value міститься у версії дерева versionNum, в іншому випадку повертає false.
  + Якщо versionNum = 0, то розглядається остання версія дерева.
* Видалення елемента (bool PersistentTree<T>::remove(const T& value))
  + Викликає приватну функцію PersistentTree<T>\* findNode(const T& value, size\_t versionNum) для останньої версії дерева. Якщо вона знаходить потрібний вузол, створює нову версію дерева, видаляє з нього value і повертає true, інакше — false.
* Повернення дерева до попередньої версії (void PersistentTree<T>::revert(size\_t versionNum))
  + Видаляє останні корені дерева, таким чином останньою версією дерева стає versionNum.
  + Використовуючи списки versionNodes, видаляє усі вузли, пов’язані з останніми коренями, таким чином звільняється пам’ять.

**Інтерфейс користувача, тестові приклади:**

У програмі є інтерактивний режим, який дозволяє користувачеві

* додавати, та видаляти елементи в останню версію множини.
* виводити на екран елементів дерева у структурованому вигляді, для буль-якої версії множини. (червоні вузли позначені RED, чорні - BLK)
* шукати елемент у будь-якій версії множини
* повертатися до минулої версії (пам`ять звільняється)

Після кожної команди програма виводить всі елементи, що зберігаються в останній версії множини, в алфавітному порядку.

Available commands: add, remove, print, print-old, search, search-old, revert, help, exit

> **add**

Enter employee name: **Bob**

{ "Bob" }

> **add**

Enter employee name: **Alice**

{ "Alice", "Bob" }

> **add**

Enter employee name: **Carl**

{ "Alice", "Bob", "Carl" }

> **add**

Enter employee name: **Daniel**

{ "Alice", "Bob", "Carl", "Daniel" }

> **print**

Left:

BLK "Alice"

BLK "Bob"

Right:

BLK "Carl"

Right:

RED "Daniel"

> **add**

Enter employee name: **Edward**

{ "Alice", "Bob", "Carl", "Daniel", "Edward" }

> **print**

Left:

BLK "Alice"

BLK "Bob"

Right:

Left:

RED "Carl"

BLK "Daniel"

Right:

RED "Edward"

> **remove**

Enter employee name: **Bob**

{ "Alice", "Carl", "Daniel", "Edward" }

> **print**

Left:

BLK "Alice"

BLK "Carl"

Right:

BLK "Daniel"

Right:

RED "Edward"

> **print-old**

Enter version number (enter 0 to print latest version): **2**

Left:

RED "Alice"

BLK "Bob"

> **search-old**

Enter version (enter 0 to print latest version): **2**

Enter employee name: **Carl**

Tree contains Carl? false

> **search-old**

Enter version (enter 0 to print latest version): **3**

Enter employee name: **Carl**

Tree contains Carl? true

> **revert**

Enter version: **2**

Reverted to version 2.

{ "Alice", "Bob" }

> **add** Enter employee name: **Yurii** { "Alice", "Bob", "Yurii" }

**Висновки**

Нам вдалося імплементувати персистентну множину на основі червоно-чорного дерева. Виявилося, що така імплементація досить ефективна в плані швидкодії, а також використання пам’яті (як і під час виконання дій над множиною так і взагалі для збереження елементів). Отже, якщо є необхідність зберігати всі версії множини під час модифікацій, цей варіант є набагато більш оптимальним в порівнянні зі звичайним backup-методом, де кожного разу копіюються всі збереженні дані.

**Список використаних джерел:**

* Лекції 2 та 3 з курсу “Алгоритми та складність” Шкільняк О.С.