**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Алгоритми та складність**

**Завдання № 2**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2020**

**Умова задачі**

Реалізуйте дерево порядкової статистики на основі червоно-чорного дерева.

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

**Опис алгоритмів**

**i-та порядкова статистика множини** – елемент з i-м в порядку зростання ключем.

**Ранг елемента** – його порядковий номер в лінійно впорядкованій множині.

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**.

Взагалі, **дерево пошуку** — це структура даних, що підтримує ряд операцій з

динамічними множинами:

* пошук елемента;
* пошук мінімального та максимального значення;
* пошук попереднього і наступного елемента;
* вставка та видалення елемента.

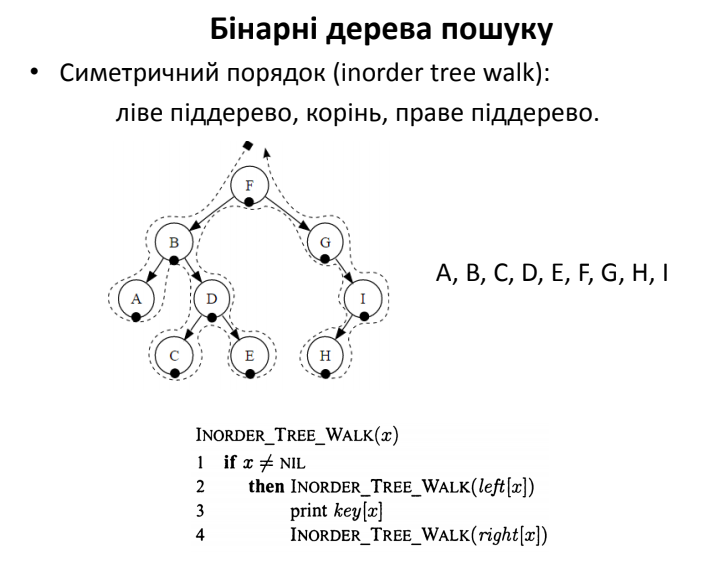
Набір операцій дозволяє використовувати дерево пошуку для реалізації як словника, так і черги з пріоритетами. Основні операції виконуються за час, пропорційний висоті дерева.

Бінарне дерево пошуку містить:

* Поле-ключ key.
* Вказівники на батька, лівого та правого синів p, left, right.
  + x – корінь: p[x]=NIL.
  + x – лист: left[x]=right[x]=NIL.

Для кожного вузла x виконується властивість бінарного дерева пошуку:

* якщо вузол y знаходиться в лівому піддереві вузла x, то key[y] ≤ key[x];
* якщо вузол y знаходиться в правому піддереві вузла x, то key[x] ≤ key[y].



Дерево **збалансоване**, якщо його висота гарантовано не перевищує log n для n елементів. Отже, для збалансованих дерев час виконання операцій над динамічними множинами навіть в найгіршому випадку Ο(log n).

Приклад збалансованого дерева — **червоно-чорне дерево**:

Бінарне дерево пошуку з додатковим бітом кольору в кожному вузлі: червоний чи чорний.

* Дерева наближено збалансовані: жоден шлях в червоно-чорному дереві не відрізняється від іншого більше ніж удвічі.
* Кожен вузол містить поля key, color, p, left, right.
* Всі вузли, що містять ключ, – внутрішні.
* Розглядатимемо значення NIL як вказівники на листя (зовнішні вузли) бінарного дерева пошуку.
* Бінарне дерево пошуку буде червоно-чорним, якщо задовольнятиме червоно-чорні властивості.

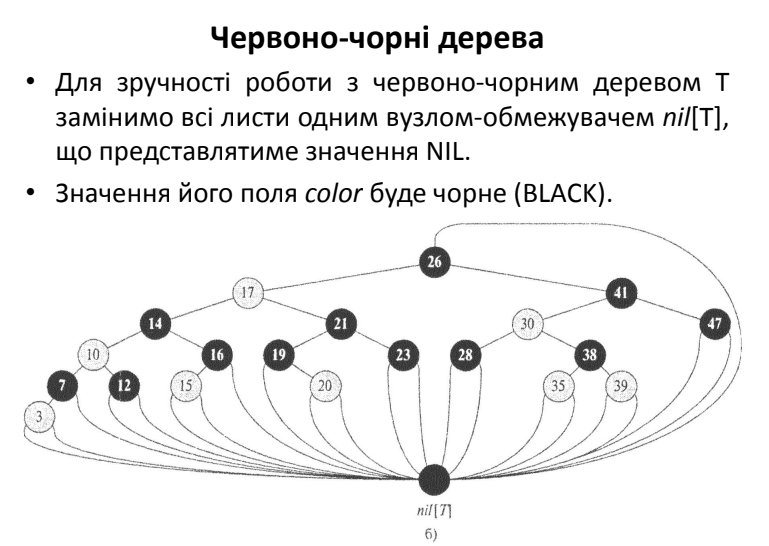
**Червоно-чорні властивості:**

1. Кожен вузол є або червоним, або чорним.
2. Корінь дерева – чорний.
3. Кожен лист дерева (NIL) – чорний.
4. Якщо вузол червоний, то обидва його сини чорні.
5. Для кожного вузла всі шляхи від нього до листів-потомків містять однакову кількість чорних вузлів.

**Чорна висота вузла x** (bh(x), black-height) – кількість чорних вузлів на шляху від вузла x (не рахуючи його самого) до листів; вона визначається однозначно.

**Чорна висота дерева** – чорна висота його кореня.



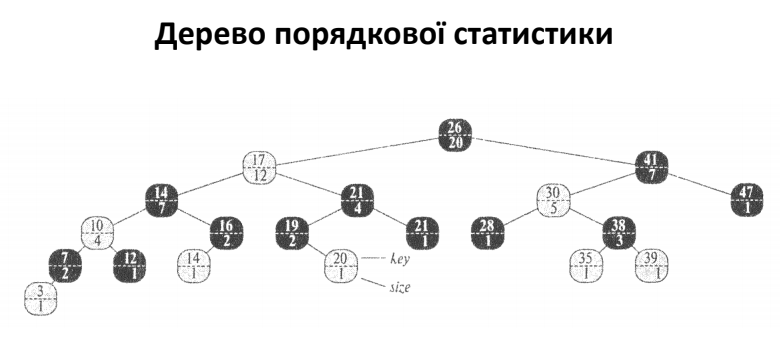


Модифіковане червоно-чорне дерево дозволяє знайти ранг і порядкову статистику за час О(lg n).

**Дерево порядкової статистики** T (order-statistic tree) – червоно-чорне дерево з додатковим інформаційним полем size[x] (розмір піддерева з коренем x). Поклавши size[nil[T]]=0, отримуємо тотожність

size[x] = size[left[x]] + size[right[x]] + 1.

Оскільки ключі потенційно не унікальні, під рангом розумітимемо позицію елемента в дереві при симетричному обході.



**Модулі програми:**

У своїй програмі я розбив модулі на дві категорії: ті, що стосуються саме предметної області (HumanResources.\*) і ті, що стосуються дерева порядкової статистики (RedBlackTree.h)

RedBlackTree<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 RedBlackTree<T>::insert(T value))
  + Виконує додавання елемента так само, як і у звичайному бін. дереві пошуку, а потім викликає приватну функцію void insertFix(RedBlackNode<T>\* newNode), яка забезпечує виконання червоно-чорних властивостей. Виконується в найгіршому випадку за O(log n).
* Пошук елемента (bool RedBlackTree<T>::contains(const T& value) const)
  + Викликає приватну функцію RedBlackNode<T>\* findNode(const T& value). Якщо вона знаходить потрібний вузол, повертає true, інакше — false. Виконується в найгіршому випадку за O(log n).
* Видалення елемента (bool RedBlackTree<T>::remove(const T& value))
  + Викликає приватну функцію RedBlackNode<T>\* findNode(const T& value). Якщо вона знаходить потрібний вузол, видаляє його і повертає true, інакше — false. Виконується в найгіршому випадку за O(log n).
* Пошук елемента за рангом ( T RedBlackTree<T>::findByRank(int rank) const)
  + Виводить елемент з рангом rank, якщо він існує. Інакше видає exception. Виконується в найгіршому випадку за O(log n).
* Визначення ранга елемента (int RedBlackTree<T>::getRank(const T& value) const)
  + Виводить найменший ранг елемента, що дорівнює value, якщо такий існує. Інакше видає exception. Виконується в найгіршому випадку за O(log n).

**Інтерфейс користувача, тестові приклади:**

У програмі є інтерактивний режим, який дозволяє користувачеві додавати, видаляти, шукати елементи і визначати їх ранг. Після кожної команди програма виводить всі елементи, що зберігаються на даний момент у дереві, в алфавітному порядку.

Available commands: add, remove, find-by-name, exit, get-rank, find-by-rank

<empty>

> **add**

Enter employee name: **Yurii**

{ 1: { Name: Yurii } }

> **add**

Enter employee name: **Dasha**

{ 1: { Name: Dasha }, 2: { Name: Yurii } }

> **add**

Enter employee name: **Sasha**

{ 1: { Name: Dasha }, 2: { Name: Sasha }, 3: { Name: Yurii } }

> **add**

Enter employee name: **Pasha**

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Sasha }, 4: { Name: Yurii } }

> **add**

Enter employee name: **Pasha2**

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha }, 5: { Name: Yurii } }

> **remove**

Enter employee name: **Yurii**

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **find-by-name**

Enter employee name: **Dasha**

Tree contains Dasha? true

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **find-by-name**

Enter employee name: **Yurii**

Tree contains Yurii? false

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **get-rank**

Enter employee name: **Pasha**

Pasha is rank #2

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **get-rank**

Enter employee name: **Yurii**

Tree does not contain value

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **find-by-rank**

Enter employee rank: **3**

Employee #3 is Pasha2

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **find-by-rank**

Enter employee rank: **69**

Tree does not contain that many elements

{ 1: { Name: Dasha }, 2: { Name: Pasha }, 3: { Name: Pasha2 }, 4: { Name: Sasha } }

> **exit**

**Список використаних джерел:**

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