УДК 004.422.635.33

Макар’ян В.Б., ст. гр. СП-11-1д, Пазюк Ю.М., доцент, к.т.н., науковий керівник

# ОСОБЛИВОСТІ ВИКОРИСТАННЯ ЧЕРВОНО-ЧОРНИХ ДЕРЕВ ДЛЯ РЕАЛІЗАЦІЇ АБСТРАКТНИХ ТИПІВ ДАНИХ

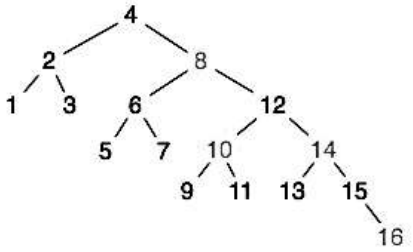
*Запорізька державна інженерна академія, кафедра ПЗАС*

У даній статті розглядаються особливості реалізації одного з різновидів двійкових пошукових дерев – червоно-чорних дерев, які відносяться до збалансованих дерев. Такі дерева використовуються для вирішення різних завдань, наприклад, в одній з реалізацій планувальника ядра ОС Linux (completely fair scheduler), є основою асоціативних контейнерів бібліотеки STL (map, set, multiset, multimap) [1] та класу TreeMap [2] мови Java.

Червоно-чорне дерево являє собою бінарне дерево пошуку з одним додатковим бітом кольору в кожному вузлі. Колір вузла може бути або чорним, або червоним. Відповідно з обмеженнями на вузли дерева, жоден шлях в червоно-чорному дереві не відрізняється від іншого по довжині більш ніж у два рази, тобто червоно-чорні дерева є наближено збалансованими. Оцінкою як середнього, так і найгіршого часу виконання основних операцій є O(log n).

Червоно-чорні дерева були запропоновані німцем Рудольфом Байєром під назвою «симетричні бінарні B-дерева». Назва «червоно-чорне дерево» структура даних отримала в статті Л. Гімпаса і Р. Седжвіка (1978) після того, як вони запропонували використовувати концепцію кольорів [3].

Для того щоб склалося правильне враження про червоно-чорні дерева була обрана саме ця ілюстрація. Дерево, зображене на ній, не збалансовано і різнокольорові вершини не розташовані рядами. Завдяки цьому не створюється помилкове відчуття, що дерево завжди ідеально збалансовано [4].




Для зручності роботи з червоно-чорним деревом замінюють всі листи одним обмежуючим вузлом – фіктивної вершиною, що представляє значення nil. У червоно-чорному дереві фіктивна вершина являє собою об’єкт з тими ж полями, що і звичайний вузол дерева. Вершина пофарбована в чорний колір, а її покажчики на ліві і праві піддерева не грають ніякої ролі, хоча для зручності ми можемо послатися на цю ж вершину. Це скоротить перевірки на нульові посилання при роботі, вставки і видаленні. Результати тестування показують, що при інтенсивній роботі зі вставкою і видаленням, таке дерево працює трохи швидше.

Використання обмежувача дозволяє нам розглядати фіктивну вершину як звичайний вузол. Хоча можна було б використовувати різні обмежувачі для кожної такої вершини (що дозволило б точно визначати їх батьківські вузли), цей підхід призвів би до невиправданої перевитрати пам'яті. Замість цього ми використовуємо єдиний обмежувач для представлення всіх nil-вершин - як листків, так і батьківського вузла кореня.

При роботі з червоно-чорними деревами необхідно часто звертатися до батьківських елементів. У такому випадку можна описати додаткове поле вершини - покажчик на батьківський елемент. Але такий підхід занадто витратний в плані пам'яті в тих випадках, коли розмір кожного з елементів даних близький до розміру покажчика. Так само ми можемо написати функцію для пошуку батька, але при великій кількості вузлів це збільшить час роботи

програми. Виходом є створення допоміжного класу - кеша для зберігання покажчиків на батьківські вузли конкретних елементів. Оновлювати кеш необхідно при обертанні вліво, вправо, вставці і видалення вузла. В першу чергу, при пошуку батька, перевіряється кеш, якщо результат незадовільний - застосовується алгоритм пошуку батьківського вузла.

**Приклад реалізації з використанням контейнера STL unordered_map:**

```
template <typename T> class Cache
{
private:
      unordered_map <T*,T*> _storage;
public:
      Cache () { }
      Cache (const Cache <T> & c) { for (auto & x: c._storage) Add (x.first,
      x.second); }
      ~Cache () { Clean (); }
      void Add (T * key, T * value) { _storage [key] = value; }
      void Clean () { _storage.clear (); }
      void Remove (T * key) { _storage.erase (key); }
      T * Get (T * key)
      {
            auto & got = _storage.find (key);
            if (got ! = _storage.end ())
                  return got-> second;
            else
                  return nullptr;
      }
      T * operator () (T * key) { return Get (key); }
};
```

**Висновки:**

• Запропоновані такі способи оптимізації червоно-чорних дерев:

1) використання фіктивної листової вершини;
2) збереження покажчиків на батьківські вершини в кожному вузлі;
3) збереження покажчиків на батьківські вершини в спеціальному кеші.

• Реалізована структура даних червоно-чорне дерево з урахуванням всіх перерахованих вище особливостей.

• Проведено порівняльні тести що підтверджують ефективність запропонованих рішень.

**Література**


1. Dr Dobbs - STL's Red-Black Trees [Електронний ресурс] — Режим доступу до статті : http://www.drdobbs.com/cpp/stls-red-black-trees/184410531
2. Опис класу TreeMap мови Java [Електронний ресурс] — Режим доступу до статті : http://java.sun.com/j2se/1.4.2/docs/api/java/util/TreeMap.html
3. Кормен Т., Лейзерсон Ч., Рівестом Р., Штайн К. Алгоритми: побудова й аналіз = Introduction to algorithms. - 2-ге вид. - М.: Видавничий дім «Вільямс», 2011. - С. 336-364.
4. Advanced Haskell Data Structures: Red-Black Trees [Електронний ресурс] — Режим доступу до статті : http://scienceblogs.com/goodmath/2009/11/30/advanced-haskell-data-structur/