КПІ ім. Ігоря Сікорського

Інститут прикладного системного аналізу

Кафедра Системного проектування

Алгоритми та структури даних

**Лабораторна робота №2.2**

**«Дослідження структури даних хеш-таблиця»**

Виконав:

Студент групи ДА-02

Рудік Андрій

Варіант №16

Мета роботи:

Ознайомитись і дослідити структуру даних хеш-таблиця. Набути навичок

реалізації хеш-таблиці за методом ланцюжків мовою програмування C++,

познайомитись з використанням STL контейнерів на прикладі unordered\_map та

порівняти власну реалізацію з готовим бібліотечним рішенням

Варіант №16.

Завдання

Написати програму для реалізації структури даних хеш-таблиця, яка дозволить проводити дуже швидкий пошук даних у наборі структур свого варіанту.

1. Реалізувати функцію для генерації випадкового великого числа, що буде унікальним ключем-ідентифікатором даних у хеш-таблиці.
2. Реалізувати хеш-таблицю за методом ланцюжків:
   1. Створити структуру Data для зберігання різнотипних даних відповідно до варіанту, вона буде використовуватись у якості значень для хеш-таблиці. Реалізувати генерацію випадкових даних для полів структури у пустому конструкторі Data().
   2. Створити структуру HashNode для зберігання ключа та значення
   3. Створити структуру LinkedList з підтримкою додавання елементів
   4. Створити структуру HashTable, яка буде містити масив зв’язних списків bucketsArray статичного розміру M = 10000 та реалізувати в ній основні функції:

* hash(key) знаходження хеш-функції від ключа
* insert(key, value) додавання значення під відповідним ключем (заміна у випадку існування ключа)
* find(key) знаходження значення під відповідним ключем, функція повертає вказівник на знайдений елемент або NULL
* erase(key) видалення значення під відповідним ключем
* size() знаходження кількості елементів в хеш-таблиці

Всі функції повинні працювати за O(1).

1. Оптимізувати хеш-таблицю, додавши динамічну зміну розміру масиву bucketsArray в залежності від loadFactor (максимально можлива завантаженість таблиці):
   1. Початковий розмір масиву bucketsArray встановити невеликим, наприклад m = 8, при перевищенні значення loadFactor динамічно збільшувати розмір (наприклад в 2 рази)
   2. Початкове значення loadFactor тимчасово обрати рівним 8.5
2. Провести тестування, використавши вказану нижче функцію testHashTable(). Обрати оптимальне значення loadFactor. Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL unordered\_map.

Необов’язкові завдання для отримання додаткових балів:

* Реалізувати ще одну хеш-таблицю, де в якості ключів використати тип string
* Реалізувати ще одну хеш-таблицю за методом відкритої адресації

Варіант 16

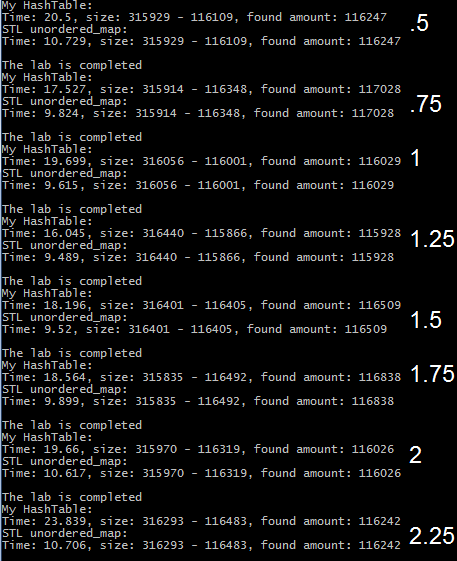
Структура Контрольна робота має наступні поля: назва предмету, кількість завдань, ймовірність списування. Створити архів робіт в якому можна швидко знайти контрольну роботу.

Лістинг

[GitHub](https://github.com/franmajka/algorithms-and-data-structures/tree/master/laba_2-2/laba_2-2) (Меня вообще не впечатляет идея копировать сюда 15 файлов так что…)

Результати

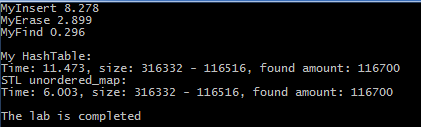
Визначення loadFactor:



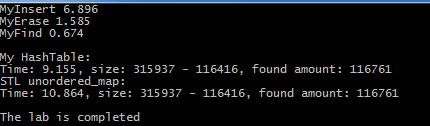
Шляхом проведення 1 повноцінного тесту було визначено, що loadFactor 1.25 дає найкращу швидкодію, але у роботі використовувався loadFactor рівний 0.75, для уніфікації хеш-таблиць реалізованих методом ланцюжків та відкритої адресації відповідно.

Швидкодія хеш-таблиць:

Метод ланцюжків:



Метод відкритої адресації:



Загалом моя реалізація дещо повільніше за unordered\_map, можливо, якщо функція хешування буде записувати в кеш ключі, то можна отримати кращий результат, але я не впевнений у цьому.

Висновок

Я ознайомився і дослідив структуру даних хеш-таблиця. Набув навичок

реалізації хеш-таблиці за методом ланцюжків мовою програмування C++,

познайомився з використанням STL контейнерів на прикладі unordered\_map та

порівняв власну реалізацію з готовим бібліотечним рішенням.