Хеш-структури C++

Аввакумов Володимир

12 листопада 2024 р.

Зміст

1 Вступ

Я обрав свою власну тему, як зрозуміло з назви — це реалізація хеш-структур, а саме словника та сета мовою C++. У подальшому тексті

опишу труднощі при створенні, вибрані мною варіанти реалізації та надам інформацію щодо необхідних бібліотек для збірки проєкту.

2 Коротко про роботу

Документація є в самому коді, тут я поясню, що в мене вийшло. Проєкт містить у собі 4 класи: HashSet, LinkedList, HashDict, LinkedList_dict — сет, зв'язний список як елемент сета, словник, зв'язний список як елемент словника відповідно. Тілом сета є масив, який динамічно збільшується, щойно заповнюється більше ніж на 75 відсотків. У комірках масиву знаходяться зв'язні списки. У сет можна додавати елементи, видаляти їх та перевіряти, чи є якийсь елемент у сеті. Додавання елемента працює наступним чином: ми беремо хеш-значення вхідної змінної, ділимо його з остачею на розмір масиву та додаємо змінну в масив за індексом, що дорівнює остачі від ділення хеша на розмір масиву. Видалення та перевірка на наявність працюють аналогічно. Щодо словника, то він виконує ті ж самі функції, але зберігає пари ключ-значення, де ключ — це те саме, що й змінна в сеті. У зв'язку з цим у словника є декілька своїх власних методів, таких як зміна значення за ключем та отримання значення за ключем. Це реалізовано перевантаженням оператора [ ], під капотом відбувається вже знайома нам логіка з хешуванням ключа та діленням з остачею на розмір.

3 Прийняті рішення для реалізації

Існує безліч варіантів для написання хеш-структур, в основному вони відрізняються методом вирішення колізій. Особисто я обрав варіант із використанням зв'язних списків (LinkedList та LinkedList_dict). Скажу наперед, це дуже впливає на розмір коду та його читабельність, адже методи розділені між двома класами. Інших недоліків помічено не було. Далі я вирішив, що спочатку напишу сет, а словник від нього успадкую. Проте, на жаль, написавши сет, я зрозумів, що успадкувати його майже неможливо (адже обидва класи шаблонні, причому у першого один шаблонний параметр, а у другого їх два), тому довелося дублювати значну частину коду.

4 Проблеми, що виникли

Найпершою проблемою була неможливість спадкування шаблонного класу з одним параметром в шаблонний клас з двома параметрами, але виявилося, що це на краще. Я частково переробив функції хешування, щоб вони приймали розмір масиву як параметр, і це виявилось доволі корисним при копіюванні основного буфера в буфер більшого розміру. Наступною ж проблемою була некоректна компіляція проєкту: лінкер постійно "губився". Це був жах; я витратив на спроби компіляції близько 4 годин, перепробував 3 основних варіанти і зрозумів, що найкращий підхід — це описати та реалізувати функції в хедері для заголовкового класу. Остання проблема трапилася, коли я написав юніт-тести для обох класів, і у мене не працювала функція `is_in` для класу `std::string`. Виявилося, що потрібно поставити перевірки на невід'ємність хеша, а також (найважливіше) врахувати, що C++ некоректно переводить клас `long long` у `int`. Після витрачених 2-х годин на дебаг я виправив цей недолік.

5 Необхідні бібліотеки для запуску

gtests (googletests)
stl (iostream, stdexcept, type_traits тощо)
Здебільшого сторонні бібліотеки використовуються лише для перевантаження оператора `«` для `std::cout` та помилок, а також для тестів, звісно.

6 Тести

6.1 Тест для HashSet

6.1.1 Конфігураційний файл (CMake)

Коротко
Файли: `dict/tests/set_test_2.cpp`, `dict/set/HashSet.h`, `dict/set/LinkedList.h`
Бібліотеки: gtests
CXX STANDARD 20
Детально

```
cmake_minimum_required(VERSION 3.28)
```

```
set(CMAKE_CXX_STANDARD 20)

find_package(GTest REQUIRED)

include_directories(${GTEST_INCLUDE_DIRS})

                    add_executable(project_directory
                    dict/tests/set_test_2.cpp
                    dict/set/HashSet.h
                    dict/set/LinkedList.h
)
target_link_libraries(project_directory GTest::gtest GTest::gtest_main pthread)
```

6.1.2 Суть тесту

Тест складається з 6 підтестів:

AddAndIsInTest — Додавання великої кількості цілих значень та перевірка їх наявності.

PopMethodLargeData — Додавання великої кількості цілих значень, видалення частини та перевірка на їх відсутність.

FloatKeys — Те ж саме, що й у `AddAndIsInTest`, але для `float`.

StringKeys — Те ж саме, що й у `AddAndIsInTest`, але для `std::string`.

ResizeOnHighOccupancy — Тест на те, що масив збільшується (він тут зайвий).

PopNonExistentElement — Перевірка на виникнення помилки при спробі видалити неіснуючий елемент.

Тести перевіряють основний функціонал хеш-сета; всі пройдені успішно.

6.2 Тести для HashDict

6.2.1 Конфігураційний файл (CMake)

Коротко:
Файли: `dict/tests/selected_test.cpp`, `dict/HashDict.h`, `dict/LinkedList_dict.h`
Бібліотеки: gtests
Варіанти для selected_ test: dict_ test_ 1, dict_ test_ 2, dict_ test_ 3

CXX STANDARD 20
Детально:

```
cmake_minimum_required(VERSION 3.28)

set(CMAKE_CXX_STANDARD 20)

find_package(GTest REQUIRED)

include_directories(${GTEST_INCLUDE_DIRS})

        add_executable(project_directory
        dict/tests/selected_test.cpp
        dict/HashDict.h
        dict/LinkedList_dict.h
)
target_link_libraries(project_directory GTest::gtest GTest::gtest_main pthread)
```

6.2.2 dict_test_1

Тут доволі багато маленьких підтестів, не бачу сенсу їх пояснювати. Він потрібен більше для відладки та бета-тестування.

6.2.3 dict_test_2

Тест складається з 4 підтестів:

LargeFloatKeys — Додавання великих кількостей `float` ключів.

LargeStringKeys — Додавання великих кількостей `std::string` ключів.

IntPointerKeys — Додавання пар з цілими числами та покажчиками на цілі числа.

FloatPointerKeys — Додавання пар з дійсними числами та покажчиками на дійсні числа.

Тут всі підтести перевіряють додавання пар ключ-значення та діставання значень за ключем за допомогою оператора [ ], відрізняються лише типом ключа: ціле число, дійсне число, покажчик на ціле число, покажчик на дійсне число відповідно. Це зрозуміло з їхньої назви.

6.2.4 dict_test_3

Тест складається з 3 підтестів:

PopMethodLargeData — Додає велику кількість пар (`int`, `std::string`), видаляє половину, далі перевіряє, чи дійсно видалені елементи видалились, а не видалені залишилися.

BracketOperatorLargeData — Те ж саме, що й у `dict_test_2`, але також тестується доступ до неіснуючих елементів.

MixedTypePopAndAccess — Схожий на `PopMethodLargeData`, але видаляє кожен третій елемент.

6.2.5 Загалом про тести

Всі тести, звісно, проходяться ідеально. Також зазначу, що використовував цей словник в домашній роботі з `int` ключами та `array` значеннями, проблем не виникло.

7 Самокритика

Опишу основні недоліки.

7.1 Dict

1. При використанні вказівників у якості ключів чи значень їх треба вручну видаляти, що викликає дискомфорт при роботі з класом `HashDict`.
2. Неможливість одразу записати всі елементи, тобто так: `HashDict d{1:2, 3:4, 5:6}`.
3. Недосконалість хеш-функцій. Вони доволі повільні та часто дають колізії.

7.2 Set

Теж саме, що й у `Dict`, але ще є кілька недоліків, у зв'язку з тим, що я робив його першим і він був такою собі пробою пера:

1. Хеш-функція повертає число, яке вже потім у потрібній нам функції ділиться на розмір масиву. Це збільшує кількість коду.
2. Ітератор у `LinkedList` доволі зайвий.