Варіант: V1, S3

Потрібно було реалізувати 3 структури даних для максимальної ефективності виконання наступних операцій:
1) Знайти студентів за заданим ім'ям і прізвищем (m_name, m_surname)
2) Знайти такі групи, де є студенти з однаковими (m_name, m_surname)
3) Змінити групу студенту за його електронною поштою (m_email)

Реалізації

- **Перша реалізація (StudentBase1)**:
  - **Структури:**
    - `std::list<Student>` - список (**linked list**), що зберігає всіх студентів зчитаних з файлу.
    - `std::unordered_map<std::pair<std::string, std::string>, std::vector<Student*>>` - геш-таблиця, де ключем є пара (ім'я, прізвище), а значенням список вказівників на студентів. структура використовується для константного пошуку всіх студентів за заданим ім'ям та прізвищем.
    - `std::unordered_map<std::pair<std::pair<std::string, std::string>, std::string>, size_t>` - геш-таблиця, де ключем є пара: перший елемент - пара (ім'я, прізвище), другий - група. Значенням є кількість студентів з таким поєднанням атрибутів.
    - `std::unordered_map<std::string, Student*>` - геш-таблиця, де ключем є електронна пошта, а значенням - вказівник на відповідного студента.
  - **Алгоритми:**
    - Ініціалізація: читання всіх студентів з файлу та заповнення структур. Складність - $O(n)$.
    - `getStudentsByName`: геш-пошук по `(name,surname)` повертає вектор вказівників. Складність - $O(1)$.
    - `getGroupsSameStudentName`: ітерація по геш-таблиці, яка зберігає кількість студентів з однаковим ім'ям, прізвищем, групою. Повертає множину всіх знайдених груп з **count>1**. Складність - $O(m)$, де m - кількість унікальних трійок (ім'я, прізвище, група).
    - `changeGroupByEmail`: геш-пошук за `email`, оновлення лічильників для старої/нової групи. Складність - $O(1)$.
  - Реалізація використовує оптимальні структури та алгоритми (для мінімізації часу виконання), так як всі операції виконуються з мінімально можливою складністю. Для `getStudentsByName`, `changeGroupByEmail` складність константна, тому оптимальнішої складності не існує. Для `getGroupsSameStudentName` складність лінійна. Оптимальнішого алгоритму не існує, так як не перевіривши кожну унікальну

трійку (ім’я, прізвище, група) хоч раз неможливо знайти всі групи в яких є хоча б 2 студенти з однаковим прізвищем та ім’ям. Наприклад, у випадку, коли було переглянуто всіх крім одного студента, може виявитись, що саме цей студент створює дублікат у певній групі і, в такому випадку, ми не можемо гарантовано правильно повернути всі групи з дублікатами. Отже, 2 операція також виконується оптимально.

- **Друга реалізація (StudentBase2)**:
  - Структури:
    - `std::list<Student>` - список (**linked list**), що зберігає всіх студентів, прочитаних з файлу.
    - `std::map<std::pair<std::string, std::string>, std::vector<Student*>>` - впорядковане дерево пошуку (**RB-tree**), де ключем є пара (ім’я, прізвище), а значенням - список вказівників на студентів. Ця структура використовується для пошуку всіх студентів за `(name, surname)` з логарифмічною складністю.
    - `std::map<std::pair<std::pair<std::string, std::string>, std::string>, size_t>` - впорядковане дерево пошуку, де ключем є пара: перший елемент - пара (ім’я, прізвище), другий - група. Значенням є кількість студентів з таким поєднанням атрибутів.
    - `std::map<std::string, Student*>` - впорядковане дерево пошуку, де ключем є електронна пошта, а значенням - вказівник на відповідного студента.
  - Алгоритми:
    - Ініціалізація: читання всіх студентів з файлу та заповнення структур. Складність - $O(n \log n)$, визначенна складністю побудови дерева.
    - `getStudentsByName`: пошук в `map` за ключем `(name, surname)` повертає вектор вказівників. Складність - $O(\log n)$.
    - `getGroupsSameStudentName`: ітерація по дереву, що зберігає лічильники `(name, surname, group)`, і вибір тих, де `count > 1`. Складність - $O(m)$, де m - кількість унікальних трійок (ім’я, прізвище, група).
    - `changeGroupByEmail`: логарифмічний пошук студента за email, оновлення лічильників для попередньої та нової групи. Складність - $O(\log n)$.
- **Третя реалізація (StudentBase3)**:
  - Структури:
    - `std::vector<Student>` - вектор (**dynamic array**), що зберігає всіх студентів, прочитаних з файлу.
  - Алгоритм:
    - Ініціалізація: читання всіх студентів з файлу та заповнення вектора. Складність - $O(n)$.
    - `getStudentsByName`: лінійний скан вектора, порівняння полів (ім’я, прізвище). Складність - $O(n)$.
    - `getGroupsSameStudentName`: лінійний скан з тимчасовим `unordered_set` трійок для виявлення повторів, геш-таблиця аналогічна до використаної в першій реалізації. Складність

- $O(n)$.
    - `changeGroupByEmail`: лінійний пошук по email і оновлення поля “група” в студентів. Складність - $O(n)$.
- Реалізація є оптимальною по пам’яті, так як інформація про кожного студента збережена лише раз, відсутні будь-які дублікати, однак операції виконуються неоптимально по часу.

У перших двох реалізаціях самі об’єкти студентів зберігались в одному списку, а всі інші структури містили не копії, а вказівники на студентів з списку. Це було зроблено для оптимізації використання пам’яті. Для збереження студентів було використано саме `std::list`, який реалізований через **linked list** через стабільність. Адреси об’єктів в ньому не змінюються, на відміну від динамічного масиву, в якому після кожного заповнення, при розширені, всі об’єкти переносяться в нові комірки пам’яті і відповідно попередня вказівники на нах стають недійсними. В **linked list** такого не відбувається, так як при додаванні елемента в кінець, просто виділяється пам’ять для нового елемента та змінюється вказівник **next** останнього елемента.

Сортування

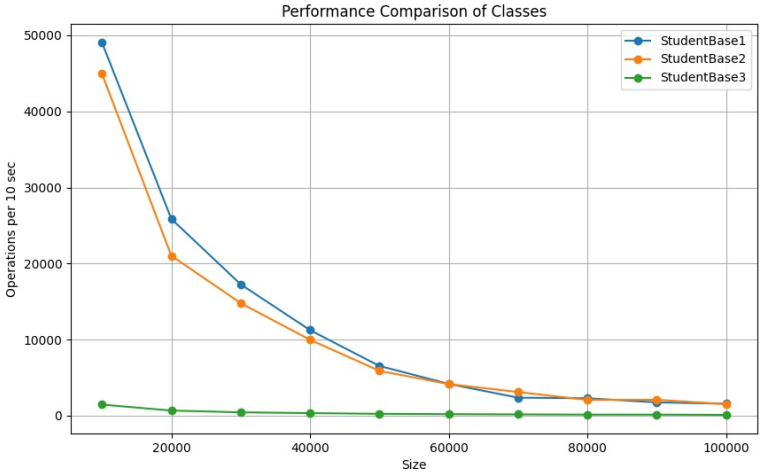
Використовується два підходи:

- `sortBuildIn` — порівняння з використанням `std::sort` / `std::ranges::sort`, використовується компаратор для коректного порівняння українських літер. Складність $O(n \log n)$.

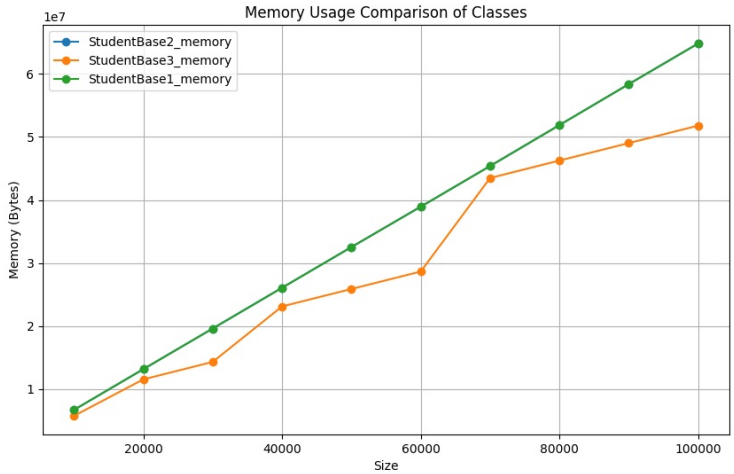
- `sortRadix` — MSD radix по українських буквах (34 бакети). Алгоритм працює рекурсивно: на кожному рівні береться символ з певної позиції в ключі студента (ім’я, прізвище) і розподіляє студентів по бакетах відповідно до цього символу. Після цього елементи з кожного бакета копіюються назад у початковий вектор, і для всіх непорожніх бакетів, крім нульового (який відповідає відсутності символу), виконується рекурсивний виклик сортування на наступній позиції символу. Таким чином, алгоритм сортує спочатку за першим символом, потім за другим і так далі, забезпечуючи лексикографічний порядок. Складність - $O(n \cdot L)$, L — середня довжина ключа (сума довжин імені та прізвища).

Продуктивність

Тестування проводилось на файлах з $10000, 20000, \dots, 100000$ студентів, які були попередньо створені шляхом вибору відповідної кількості випадкових студентів з наданого файлу. При тестуванні час читання файлу та ініціалізації не враховувався.

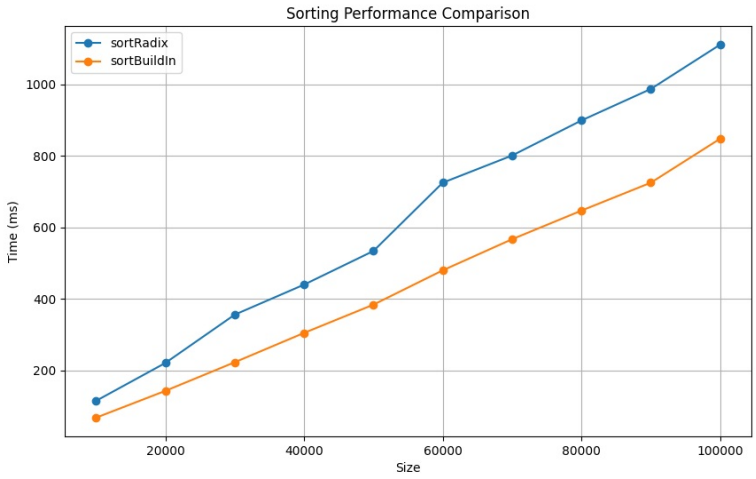


Як і очікувалось реалізація через геш таблиці найшвидша, з використанням дерев трохи повільніша, з вектором значно повільніша. Різниця в часі між реалізаціями з використанням `unordered_set` та `map` відносно не велика через те, що складність виконання 2 операції однакова для обох структур, а в тестуванні операції виконувались в наступній пропорції: 1 : 5 : 1, тобто саме 2 операція виконувалась найбільшу кількість разів. Через таку пропорцію, та те, що порівняно з двома іншими операціями друга виконується найдовше, час її виконання і був основним чинником у визначенні кількості операцій.



Витрати по пам’яті для геш таблиць та дерев однакові, для вектора витрати значно менші, так як нічого крім масиву з студентами не зберігається. Порівняння проводилось з використання функції `sizeof` та визначення кількості пам’яті необхідної для збереження даних в структури, однак додаткові метадані, які необхідні для збереження самих структур та їх властивостей не були враховані. Саме через такий метод тестування кількість пам’яті в реалізаціях з

використанням геш таблиць та дерев однакова. Фактичний підрахунок використаної **RAM** не був проведений через складність коректної реалізації.



Обидва методи сортування за графіком асимптотично подібні, однак час для Radix sort трохи більший. Для наданого файлу на 100000 входжень radix sort ще не отримує достатньої переваги так як $log_2(n) \approx 16,6$, що співставне з сумарною довжиною імені та прізвища. Крім того, вбудована реалізація високооптимізована, що забезпечує кращий час виконання. Для більших файлів переваги Radix sort можуть краще проявитись.

Структура проєкту:

- 3 реалізації баз студентів зберігаються в `./src` (**student_base_1**, **student_base_2**, **student_base_3**).
- Обидві реалізації сортування та функція для запису вектора з студентами в файл зберігаються за шляхом `./src/student_base_sort.cpp`
- Більш детальні результати бенчмарків у форматі **.csv** зберігаються в `./benchmarks/benchmark-results`
- Графіки порівнянь зберігаються в `./benchmarks/benchmark-plots`
- Файли, на яких було проведено тестування зберігаються в `./students-csv-files`