Статическое тестирование fixed set

Евгений Сдобнов

29 ноября 2020 г.

1 Fixed set

1.1 Формулировка задачи

Реализуйте следующий класс для хранения множества целых чисел:

```
class FixedSet {
public:
    FixedSet();
    void Initialize(const vector<int>& numbers);
    bool Contains(int number) const;
};
```

При вызове Initialize FixedSet получает набор целых чисел, который впоследствии и будет хранить. Набор чисел не будет изменяться с течением времени (до следующего вызова Initialize). Операция Contains возвращает true, если число number содержится в наборе.

Мат. ожидание времени работы Initialize должно составлять $O(n)$, где n — количество чисел в numbers. Затраты памяти должны быть порядка $O(n)$ в худшем случае.

Операция Contains должна выполняться за $O(1)$ в худшем случае.

Размер numbers не превосходит 10^5

1.2 Основная идея

Заводим хеш таблицу первого уровня. На этом уровне от хеша практически ничего не требуется(всё-таки требуется, но об этом позже).

Можем получить множество коллизий. В каждой цепочке коллизий строим ещё одну хеш таблицу второго уровня. Такие таблицы можно построить уже без коллизий и в среднем быстро. В итоге поиск в FixedSet будет осуществляться за два отображения. Т.е. за константу.

1.3 Отбираем уникальные числа

Нам необходимо избавиться от дублей в данных. Т.к. в противном случае от коллизий будет непросто избавиться.

Например на вход подадим все одинаковые числа, тогда хешируя $((\alpha x + \beta) \mod P) \mod N$ получаем, что все они склеиваются.

Дополнительная память $O(N)$.

Сложность $O(N \log N)$.(Сортировка + избавляемся от дублей за линию)

1.4 Хеш таблицы второго уровня

Основная идея о совершенном хешировнии:

ξ - случайная величина равная числу коллизий в линейной хеш функции.

$P\{\xi = 0\} \geq 1 - \frac{N(N-1)}{2M}$, где N-число уникальных элементов, для которых строим хеш функцию, а M - число бакетов хеш функции.

Тогда, взяв M порядка N^2, получим, что вероятность отсутствия коллизии больше $\frac{1}{2}$

Это можно получить по неравенству Маркова $P\{\xi \geq 1\} \leq \frac{E\xi}{1}$. Т.к. $E\xi = \frac{N(N-1)}{2M}$

Хотим оценить сверху сколько раз в среднем придётся выбрать параметры для хеш функции, пока не получим хеш без коллизий. Для этого введём η - количество выборов до отсутствия коллизий. Параметры выбираем равномерно и независимо от прошлого шага, тогда:

$$E\xi = \sum_{k=1}^{\infty} kP\{\eta = k\} = \sum_{k=1}^{\infty} kP\{\xi_1 \neq 0, \ldots, \xi_{k-1} \neq 0, \xi_k = 0\} =$$

$$= \sum_{k=1}^{\infty} kP\{\xi_1 \neq 0\} \ldots P\{\xi_{k-1} \neq 0\}P\{\xi_k = 0\} \leq \sum_{k=1}^{\infty} k\frac{1}{2^{k-1}}P\{\xi_k = 0\} \leq$$

$$= 2\sum_{k=1}^{\infty} k\frac{1}{2^k} = 4$$

Таким образом оценили сверху, что в среднем за 4 выбора хеш функции не будем иметь коллизий.

Но в чём же проблема? Почему мы просто не можем сделать один такой хеш и всё?

Проблема заключается в квадратичной по N памяти, для достижения необходимой вероятности. В нашем случае это 10^{10}, что не вписывается в ограничения.

Поэтому такой трюк выгодно проворачивать на маленьких N. Но для этого в начале нам необходимо разделить наши данные.

1.5 Хеш таблица первого уровня

Как уже было сказано выше, нам почти без разницы какая будет хеш функция первого уровня. Но...

Попробуем сделать так, чтобы в каждом массиве коллизий мы смогли использовать трюк с хеш таблицей второго уровня. Что для этого надо?

Обозначим за L_i - количество коллизий для i-ого бакета на первом уровне.

Тогда мы хотим, чтобы сумма всех квадратов L_i-ых была линейна с какой-то константой, тогда будут соблюдены ограничения по памяти.

Для совершенного хеша можно показать следующее:

$$P\{\sum_{i=0}^{N} L_i^2 > 4N\} < \frac{1}{2}$$

Тогда полностью аналогично прошлому разу можем посчитать оценку сверху на среднее число выборов хешей до достижения необходимого условия(сумма квадратов длин меньше 4N) и получить константную оценку сверху.

В итоге по сложности мы будем делать N раз в среднем O(1) выборов хешей, тогда оценка в среднем будет O(N) по времени.

Оценка же по памяти у нас будет O(N). Т.к. мы выбирали хеш специально так, чтобы сумма квадратов длин была меньше 4N. А каждая хеш функция второго уровня будет потреблять L_i^2 памяти. В итоге памяти не больше 4N. Получили линейное ограничение по памяти.

1.6 Итог

Получаем, что суммарная сложность по памяти линейная

Средняя сложность построения хеша $O(n \log n)$ (мы ещё в начале убирали дубликаты с помощью сортировки).

Сложность определения принадлежности будет O(1) в худшем. Т.к. В начале мы идём по хеш функции первого уровня(в силу того что она из линейного семейства, то сложность константная). И на втором уровне мы также идём по хешу. А коллизий у нас на втором уровне не имеется по построению.

1.7 Статическое тестирование

Поправил код по стайлгайду.

```
clang-format -i test_fixed_set.cpp
```

```
clang-tidy test_fixed_set.cpp -- -std=c++17
```

Запустил компиляцию со следующими ключами

```
g++-8 test_fixed_set.cpp -fsanitize=address,undefined
-fno-sanitize-recover=all -std=c++17 -O2 -Wall -Werror
-Wextra -Wsign-compare -pedantic -Weffc++ -Wunreachable-code
-o debug_solution
```

В итоге в конце насыпало, что надо инициализировать все поля класса в конструкторе.

Всяких штук типа приведения в стиле C, сравнение беззнаковых и знаковых типов и прочих замечено не было.

Поставил cppcheck(в директории можно увидеть).

Запустил команду

```
./cppcheck -q -j4 ../test_fixed_set.cpp
```

Проблем не обнаружил.

Запустил с более жёсткими ключами

```
./cppcheck -q --enable=all --inconclusive ../test_fixed_set.cpp
```

В итоге насыпало очень много

```
../fixed set.cpp:23:5: warning:inconclusive:
Member variable 'LinearHash::alpha_'
is not initialized in the constructor. [uninitMemberVar]
    LinearHash() {
    ^
../fixed set.cpp:23:5: warning:inconclusive:
Member variable 'LinearHash::beta_'
is not initialized in the constructor. [uninitMemberVar]
    LinearHash() {
    ^
../fixed set.cpp:23:5: warning:inconclusive:
Member variable 'LinearHash::primary_number_'
is not initialized in the constructor. [uninitMemberVar]
    LinearHash() {
```

```
    ^
../fixed set.cpp:86:14: warning: Member variable
'ExternalHashTable::number_of_buckets_'
is not initialized in the constructor. [uninitMemberVar]
    explicit ExternalHashTable(int primary_number = 2'000'000'003)
             ^
../fixed set.cpp:154:14: warning: Member variable
'InternalHashTable::number_of_buckets_' is not initialized
in the constructor. [uninitMemberVar]
    explicit InternalHashTable(int primary_number = 2'000'000'003)
             ^
../fixed set.cpp:80:10: style: Unused private function:
'ExternalHashTable::Clear' [unusedPrivateFunction]
    void Clear() {
         ^
../test_fixed_set.cpp:143:15: style: Local variable 'i'
shadows outer variable [shadowVariable]
    for (auto i : vec) {
              ^
../test_fixed_set.cpp:136:12: note: Shadowed declaration
  for (int i = 0; i < 100; ++i) {
           ^
../test_fixed_set.cpp:143:15: note: Shadow variable
    for (auto i : vec) {
              ^
../fixed set.cpp:38:13: style: Consider using std::accumulate
algorithm instead of a raw loop. [useStlAlgorithm]
        sum += i * i;
            ^
../fixed set.cpp:97:43: style: Consider using std::any_of
algorithm instead of a raw loop. [useStlAlgorithm]
          if (value_in_bucket == value) {
                                      ^
../fixed set.cpp:131:19: style: Consider using std::fill or
std::generate algorithm instead of a raw loop. [useStlAlgorithm]
              i = std::nullopt;
                ^
../test_fixed_set.cpp:25:21: style: Consider using
std::transform algorithm instead of a raw loop. [useStlAlgorithm]
    request_answers.push_back(set.Contains(request));
                    ^
../test_fixed_set.cpp:122:0: information: Skipping configuration 'MAGIC'
since the value of 'MAGIC' is unknown.
Use -D if you want to check it.
You can use -U to skip it explicitly. [ConfigurationNotChecked]
  int second = first + MAGIC;
^
nofile:0:0: information: Cppcheck cannot find all
the include files (use --check-config for details) [missingIncludeSystem]
```