Хеш-функция - это математический алгоритм, который отображает данные произвольного размера в битовый массив фиксированного размера.

Результат, производимый хеш-функцией, называется «хеш-суммой» или же просто «хешем», а входные данные часто называют «сообщением».

Для идеальной хеш-функции выполняются следующие условия:

a) хеш-функция является детерминированной, то есть одно и то же сообщение приводит к одному и тому же хеш-значению
b) значение хеш-функции быстро вычисляется для любого сообщения
c) невозможно найти сообщение, которое дает заданное хеш-значение
d) невозможно найти два разных сообщения с одинаковым хеш-значением
e) небольшое изменение в сообщении изменяет хеш настолько сильно, что новое и старое значения кажутся некоррелирующими

Хеш-таблицы

Мотивация использовать хеш-таблицы

Для наглядности рассмотрим стандартные контейнеры и асимптотику их наиболее часто используемых методов.

Контейнер \ операция	**insert**	**remove**	**find**
Array	O(N)	O(N)	O(N)
List	O(1)	O(1)	O(N)
Sorted array	O(N)	O(N)	O(logN)
Бинарное дерево поиска	O(logN)	O(logN)	O(logN)
Хеш-таблица	O(1)	O(1)	O(1)

Все данные при условии хорошо выполненных контейнерах, хорошо подобранных хеш-функциях

Из этой таблицы очень хорошо понятно, почему же стоит использовать хеш-таблицы. Но тогда возникает противоположный вопрос: *почему же тогда ими не пользуются постоянно?*
Ответ очень прост: как и всегда, невозможно получить все сразу, а именно: и

скорость, и память. Хеш-таблицы тяжеловесные, и, хоть они и быстро отвечают на вопросы основных операций, пользоваться ими все время очень затратно.

Понятие хеш-таблицы

Хеш-таблица — это контейнер, который используют, если хотят быстро выполнять операции вставки/удаления/нахождения. В языке C++ хеш-таблицы скрываются под флагом unoredered_set и unordered_map. В Python вы можете использовать стандартную коллекцию set — это тоже хеш-таблица.
Реализация у нее, возможно, и не очевидная, но довольно простая, а главное — как же круто использовать хеш-таблицы, а для этого лучше научиться, как они устроены.

Для начала объяснение в нескольких словах. Мы определяем функцию хеширования, которая по каждому входящему элементу будет определять натуральное число. А уже дальше по этому натуральному числу мы будем класть элемент в (допустим) массив. Тогда имея такую функцию мы можем за O(1) обработать элемент.

Теперь стало понятно, почему же это именно **хеш**-таблица.

Проблема коллизии

Естественно, возникает вопрос, почему невозможно такое, что мы попадем дважды в одну ячейку массива, ведь представить функцию, которая ставит в сравнение каждому элементу совершенно различные натуральные числа просто невозможно. Именно так возникает проблема коллизии, или проблемы, когда хеш-функция выдает одинаковое натуральное число для разных элементов.

Существует несколько решений данной проблемы: метод цепочек и метод двойного хеширования. В данной статье я постараюсь рассказать о втором методе, как о более красивом и, возможно, более сложном.

Решения проблемы коллизии методом двойного хеширования

Мы будем (как несложно догадаться из названия) использовать две хеш-функции, возвращающие взаимопростые натуральные числа.

Одна хеш-функция (при входе g) будет возвращать натуральное число s, которое будет для нас начальным. То есть первое, что мы сделаем, попробуем поставить элемент g на позицию s в нашем массиве. Но что, если это место уже занято? Именно здесь нам пригодится вторая хеш-функция, которая будет возвращать t — шаг, с которым мы будем в дальнейшем искать место, куда бы поставить элемент g.

Мы будем рассматривать сначала элемент s, потом s + t, затем s + 2*t и т.д. Естественно, чтобы не выйти за границы массива, мы обязаны смотреть на номер элемента по модулю (остатку от деления на размер массива).

Наконец мы объяснили все самые важные моменты, можно перейти к непосредственному написанию кода, где уже можно будет рассмотреть все оставшиеся нюансы. Ну а строгое математическое доказательство корректности использования двойного хеширования можно найти тут.

Реализация хеш-таблицы

Для наглядности будем реализовывать хеш-таблицу, хранящую строки.

Начнем с определения самих хеш-функций, реализуем их методом Горнера. Важным параметром корректности хеш-функции является то, что возвращаемое значение должно быть взаимопросто с размером таблицы. Для уменьшения дублирования кода, будем использовать две структуры, ссылающиеся на реализацию самой хеш-функции.

```
int HashFunctionHorner(const std::string& s, int table_size, const int key)
{
   int hash_result = 0;
   for (int i = 0; s[i] != s.size(); ++i)
      hash_result = (key * hash_result + s[i]) % table_size;
   hash_result = (hash_result * 2 + 1) % table_size;
   return hash_result;
}
struct HashFunction1
{
   int operator()(const std::string& s, int table_size) const
   {
      return HashFunctionHorner(s, table_size, table_size - 1); // ключи должны быть
взаимопросты, используем числа <размер таблицы> плюс и минус один.
   }
};
struct HashFunction2
{
   int operator()(const std::string& s, int table_size) const
   {
      return HashFunctionHorner(s, table_size, table_size + 1);
   }
};
```

Чтобы идти дальше, нам необходимо разобраться с проблемой: что же будет, если мы удалим элемент из таблицы? Так вот, его нужно пометить флагом deleted, но просто

удалять его безвозвратно нельзя. Ведь если мы так сделаем, то при попытке найти элемент (значение хеш-функции которого совпадет с ее значением у нашего удаленного элемента) мы сразу наткнемся на пустую ячейку. А это значит, что такого элемента и не было никогда, хотя, он лежит, просто где-то дальше в массиве. Это основная сложность использования данного метода решения коллизий.

Помня о данной проблеме построим наш класс.

```
template <class T, class THash1 = HashFunction1, class THash2 = HashFunction2>
class HashTable
{
   static const int default_size = 8; // начальный размер нашей таблицы
   constexpr static const double rehash_size = 0.75; // коэффициент, при котором произойдет увеличение таблицы
   struct Node
   {
      T value;
      bool state; // если значение флага state = false, значит элемент массива был удален (deleted)
      Node(const T& value_) : value(value_), state(true) {}
   };
   Node** arr; // соответственно в массиве будут хранится структуры Node*
   int size; // сколько элементов у нас сейчас в массиве (без учета deleted)
   int buffer_size; // размер самого массива, сколько памяти выделено под хранение нашей таблицы
   int size_all_non_nullptr; // сколько элементов у нас сейчас в массиве (с учетом deleted)
};
```

На данном этапе мы уже более-менее поняли, что у нас будет храниться в таблице. Переходим к реализации служебных методов.

```
...
public:
   HashTable()
   {
      buffer_size = default_size;
      size = 0;
      size_all_non_nullptr = 0;
      arr = new Node*[buffer_size];
      for (int i = 0; i < buffer_size; ++i)
         arr[i] = nullptr; // заполняем nullptr - то есть если значение отсутствует, и никто раньше по этому адресу не обращался
   }
   ~HashTable()
   {
      for (int i = 0; i < buffer_size; ++i)
         if (arr[i])
            delete arr[i];
      delete[] arr;
   }
```

Из необходимых методов осталось еще реализовать динамическое увеличение, расширение массива — метод **Resize**.

Увеличиваем размер мы стандартно вдвое.

```
void Resize()
   {
      int past_buffer_size = buffer_size;
      buffer_size *= 2;
      size_all_non_nullptr = 0;
      size = 0;
      Node** arr2 = new Node * [buffer_size];
      for (int i = 0; i < buffer_size; ++i)
         arr2[i] = nullptr;
      std::swap(arr, arr2);
      for (int i = 0; i < past_buffer_size; ++i)
      {
         if (arr2[i] && arr2[i]->state)
            Add(arr2[i]->value); // добавляем элементы в новый массив
      }
      // удаление предыдущего массива
      for (int i = 0; i < past_buffer_size; ++i)
         if (arr2[i])
            delete arr2[i];
      delete[] arr2;
   }
```

Немаловажным является поддержание асимптотики O(1) стандартных операций. Но что же может повлиять на скорость работы? Наши удаленные элементы (deleted). Ведь, как мы помним, мы ничего не можем с ними сделать, но и окончательно обнулить их не можем. Так что они тянутся за нами огромным балластом. Для ускорения работы нашей хеш-таблицы воспользуемся рехешом (как мы помним, мы уже выделяли под это очень странные переменные).

Теперь воспользуемся ими, если процент реальных элементов массива стал меньше 50, мы производим **Rehash**, а именно делаем то же самое, что и при увеличении таблицы (resize), но не увеличиваем. Возможно, это звучит глуповато, но попробую сейчас объяснить. Мы вызовем наши хеш-функции от всех элементов, переместим их в новых массив. Но с deleted-элементами это не произойдет, мы не будем их перемещать, и они удалятся вместе со старой таблицей.

Но к чему слова, код все разъяснит:

```
void Rehash()
   {
      size_all_non_nullptr = 0;
      size = 0;
      Node** arr2 = new Node * [buffer_size];
```

```
        for (int i = 0; i < buffer_size; ++i)
            arr2[i] = nullptr;
        std::swap(arr, arr2);
        for (int i = 0; i < buffer_size; ++i)
        {
            if (arr2[i] && arr2[i]->state)
                Add(arr2[i]->value);
        }
        // удаление предыдущего массива
        for (int i = 0; i < buffer_size; ++i)
            if (arr2[i])
                delete arr2[i];
        delete[] arr2;
    }
```

Ну теперь мы уже точно на финальной, хоть и длинной, и полной колючих кустарников, прямой. Нам необходимо реализовать вставку (Add), удаление (Remove) и поиск (Find) элемента.

Начнем с самого простого — метод **Find** элемент по значению.

```
bool Find(const T& value, const THash1& hash1 = THash1(), const THash2& hash2 = THash2())
    {
        int h1 = hash1(value, buffer_size); // значение, отвечающее за начальную позицию
        int h2 = hash2(value, buffer_size); // значение, ответственное за "шаг" по таблице
        int i = 0;
        while (arr[h1] != nullptr && i < buffer_size)
        {
            if (arr[h1]->value == value && arr[h1]->state)
                return true; // такой элемент есть
            h1 = (h1 + h2) % buffer_size;
            ++i; // если у нас i >=  buffer_size, значит мы уже обошли абсолютно все ячейки, именно для этого мы считаем i, иначе мы могли бы зациклиться.
        }
        return false;
    }
```

Далее мы реализуем удаление элемента — **Remove**. Как мы это делаем? Находим элемент (как в методе Find), а затем удаляем, то есть *просто меняем значение **state** на false*, но сам Node мы не удаляем.

```
bool Remove(const T& value, const THash1& hash1 = THash1(), const THash2& hash2 = THash2())
    {
        int h1 = hash1(value, buffer_size);
        int h2 = hash2(value, buffer_size);
        int i = 0;
        while (arr[h1] != nullptr && i < buffer_size)
        {
```

```
            if (arr[h1]->value == value && arr[h1]->state)
            {
                arr[h1]->state = false;
                --size;
                return true;
            }
            h1 = (h1 + h2) % buffer_size;
            ++i;
        }
        return false;
    }
```

Ну и последним мы реализуем метод **Add**. В нем есть несколько очень важных нюансов. Именно здесь мы будем проверять на необходимость рехеша.

Помимо этого в данном методе есть еще одна часть, поддерживающая правильную асимптотику. Это запоминание первого подходящего для вставки элемента (даже если он deleted). Именно туда мы вставим элемент, если в нашей хеш-таблицы нет такого же. Если ни одного deleted-элемента на нашем пути нет, мы создаем новый Node с нашим вставляемым значением.

```
bool Add(const T& value, const THash1& hash1 = THash1(),const THash2& hash2 = THash2())
    {
        if (size + 1 > int(rehash_size * buffer_size))
            Resize();
        else if (size_all_non_nullptr > 2 * size)
            Rehash(); // происходит рехеш, так как слишком много deleted-элементов
        int h1 = hash1(value, buffer_size);
        int h2 = hash2(value, buffer_size);
        int i = 0;
        int first_deleted = -1; // запоминаем первый подходящий (удаленный) элемент
        while (arr[h1] != nullptr && i < buffer_size)
        {
            if (arr[h1]->value == value && arr[h1]->state)
                return false; // такой элемент уже есть, а значит его нельзя вставлять повторно
            if (!arr[h1]->state && first_deleted == -1) // находим место для нового элемента
                first_deleted = h1;
            h1 = (h1 + h2) % buffer_size;
            ++i;
        }
        if (first_deleted == -1) // если не нашлось подходящего места, создаем новый Node
        {
            arr[h1] = new Node(value);
            ++size_all_non_nullptr; // так как мы заполнили один пробел, не забываем записать, что
это место теперь занято
        }
        else
        {
            arr[first_deleted]->value = value;
            arr[first_deleted]->state = true;
```

```
    }
    ++size; // и в любом случае мы увеличили количество элементов
    return true;
}
```