Ассоциативные контейнеры — конспект темы

Введение в ассоциативные контейнеры

Ассоциативные контейнеры, такие как `map`, эффективно ищут нужный элемент по ключу.

Часто ключ — это само значение. Так происходит в `set`, похожем на `map`. `set` — ассоциативный контейнер, потому что он умеет быстро находить нужный элемент.

`vector` — линейный контейнер, но имеет признаки ассоциативного.

Так `map` используется для поиска слов, наиболее часто встречаемых в тексте:

```
// файл wordstat.cpp

#include <iostream>
#include <map>
#include <string>
#include <tuple>

using namespace std;

int main() {
    string word;
    map<string, int> counts_map;

    while (cin >> word) {
        ++counts_map[move(word)];
    }

    cout << "Слово - Количество упоминаний в тексте"s << endl;
    // выводим первые 10 слов
    for (auto [iter, i] = tuple(counts_map.begin(), 0); i < 10 && iter != counts_map.end(); ++i, ++iter) {
        cout << iter->first << " - "s << iter->second << endl;
    }
}
```

Ускоряем, отказываясь от сортировки

Ускорить программу можно, изменив всего одну строчку. Сначала важно понять, какой этап программы занимает большее время. Для этого используйте макрос `LOG_DURATION`:

```
// файл wordstat.cpp

#include "log_duration.h"

#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <tuple>
#include <vector>

using namespace std;
```

```
vector<pair<string, int>> GetSortedWordCounts(vector<string> words) {
    map<string, int> counts_map;

    {
        LOG_DURATION("Filling"s);

        for (auto& word : words) {
            ++counts_map[move(word)];
        }
    }

    {
        LOG_DURATION("Moving & sorting"s);

        vector<pair<string, int>> counts_vector(move_iterator(counts_map.begin()), move_iterator(counts_map.end()));
        sort(counts_vector.begin(), counts_vector.end(), [](const auto& l, const auto& r) {
            return l.second > r.second;
        });

        return counts_vector;
    }
}

int main() {
    vector<string> words;
    string word;

    while (cin >> word) {
        words.push_back(word);
    }

    auto counts_vector = GetSortedWordCounts(move(words));

    cout << "Слово - Количество упоминаний в тексте"s << endl;
    // выводим первые 10 слов
    for (auto [iter, i] = tuple(counts_vector.begin(), 0); i < 10 && iter != counts_vector.end(); ++i, ++iter) {
        cout << iter->first << " - "s << iter->second << endl;
    }
}
```

Этап filling медленнее этапа moving and sorting. Оптимизировать нужно его. Для этого перепишите одну лексему: замените `map` на `unordered_map`. Включите дополнительный заголовочный файл с названием `<unordered_map>`.

Хеш-функции

Хеш-функция ниже возвращает цифровую часть номера, полностью игнорируя буквы и регион, и присваивает номер объекту произвольного типа:

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <vector>

using namespace std;

class VehiclePlate {
public:
    VehiclePlate(char l0, char l1, int digits, char l2, int region)
        : letters_{l0, l1, l2}
        , digits_(digits)
        , region_(region) {
```

```
    }

    string ToString() const {
        ostringstream out;
        out << letters_[0] << letters_[1];
        // чтобы дополнить цифровую часть номера слева нулями
        // до трёх цифр, используем подобные манипуляторы:
        // setfill задаёт символ для заполнения,
        // right задаёт выравнивание по правому краю,
        // setw задаёт минимальное желаемое количество знаков
        out << setfill('0') << right << setw(3) << digits_;
        out << letters_[2] << setw(2) << region_;

        return out.str();
    }

    int Hash() const {
        return digits_;
    }

private:
    array<char, 3> letters_;
    int digits_;
    int region_;
};

ostream& operator<<(ostream& out, VehiclePlate plate) {
    out << plate.ToString();
    return out;
}
```

Значение хеш-функции объекта называется его **хешем**.

Устройство unordered_map и unordered_set

Согласно cppreference.com, сложность методов `unordered_map::operator[]` и `unordered_map::insert` в среднем лучше, чем у `map`.

Если хеш-функция имеет константную сложность, сложность методов — $O(1)$.

В худшем случае возникает $O(N)$, где N — количество элементов в контейнере.

`unordered_map` и `unordered_set` используют хеш-функцию для хранения объектов и корзинки для их размещения. В `unordered_map` и `unordered_set` реализованы алгоритмы для определения количества корзинок и разрешения коллизий.

`unordered_set` не использует отдельную корзинку для каждого возможного значения хеш-функции. Поэтому хеши объектов могут быть большими, а выделенных корзинок будет немного. Контейнер сам организует размещение объектов по корзинкам.

Чтобы сообщить `unordered_set` и `unordered_map`, как вычислять хеш произвольного объекта, создайте специальный класс **хешер** и укажите его как шаблонный параметр контейнера. Объект хешера должен быть вызываемым — например, переопределять оператор «круглые скобки». Вызов этого объекта должен возвращать число типа `size_t`.

Реализация хешера:

```
...

class VehiclePlateHasher {
```

```
public:
    size_t operator()(const VehiclePlate& plate) const {
        return static_cast<size_t>(plate.Hash());
    }
};

int main() {
    // явно указываем хешер шаблонным параметром
    unordered_set<VehiclePlate, VehiclePlateHasher> plate_base;

    plate_base.insert({'B', 'H', 840, 'E', 99});
    plate_base.insert({'O', 'K', 942, 'K', 78});
    plate_base.insert({'O', 'K', 942, 'K', 78});
    plate_base.insert({'O', 'K', 942, 'K', 78});
    plate_base.insert({'O', 'K', 942, 'K', 78});
    plate_base.insert({'H', 'E', 968, 'C', 79});
    plate_base.insert({'T', 'A', 326, 'X', 83});
    plate_base.insert({'H', 'H', 831, 'P', 116});
    plate_base.insert({'A', 'P', 831, 'Y', 99});
    plate_base.insert({'P', 'M', 884, 'K', 23});
    plate_base.insert({'O', 'C', 34, 'P', 24});
    plate_base.insert({'M', 'Y', 831, 'M', 43});
    plate_base.insert({'B', 'P', 831, 'M', 79});
    plate_base.insert({'K', 'T', 478, 'P', 49});
    plate_base.insert({'X', 'P', 850, 'A', 50});

    for (auto& plate : plate_base) {
        cout << plate << endl;
    }
}
```

В стандартной библиотеке определены хешеры для стандартных объектов: строк, чисел, указателей, `optional` и некоторых других. Эти хешеры реализованы в шаблонном классе `hash`.

Порядок `unordered_set` не определяется его содержимым.

Контейнером `unordered_map` можно заменять `map`, если уместен произвольный порядок элементов. Как и в `unordered_set`, в этом контейнере требуется, чтобы ключи удовлетворяли двум условиям:

- имели хешер — класс, вычисляющий хеш-функцию и указываемый шаблонным параметром;
- имели определённый `operator==`.

В `unordered_map` эти условия должны быть соблюдены для ключей, а значения могут быть произвольными.

Если не хотите определять `operator==`, поменяйте компаратор, который задаётся шаблонным параметром после хешера. Компаратор использует `operator()` для выполнения действия. Но, в отличие от хешера, он не вычисляет характеристику одного объекта, а определяет понятие равенства.

Искусство хеш-функций

Неупорядоченный контейнер сильно зависит от хеш-функции. Если хешер плохой, контейнер будет работать неэффективно.

Почему хеш-функция бывает плохой:

- она может учитывать не все данные объекта, провоцируя коллизии,
- она может медленно вычисляться.

Назначение хеш-функции — перемешивать данные, делать их неузнаваемыми. Если при этом происходит что-нибудь необратимое и непонятное — прекрасно. Значит, коллизии будут возникать реже.

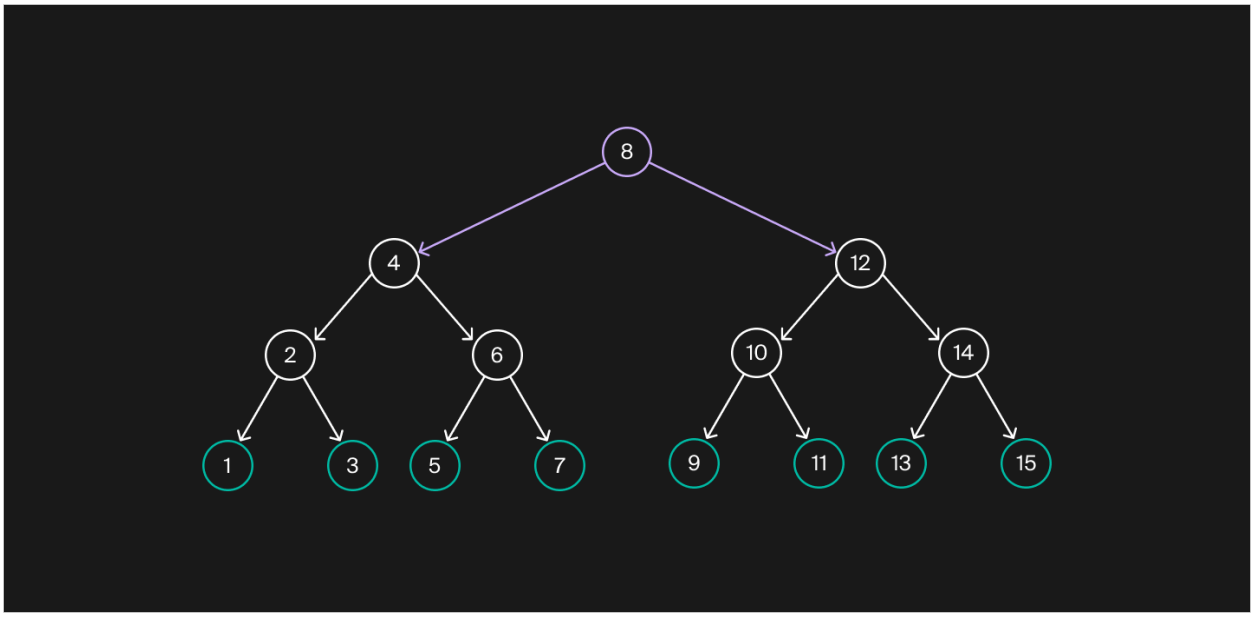
Деревья и поиск

В основе `map` лежит **двоичное сбалансированное дерево поиска** — структура, обеспечивающая быстрый поиск нужного ключа среди множества элементов.

Дерево похоже на список, у него тоже есть узлы. Она хранят значение и ссылаются на последующие. Но последующих элементов больше одного. Дерево принимает разветвлённую форму, напоминающую крону. Первый элемент называется **корень**.

У каждого узла может быть не более двух последующих. Если их нет, узел называется **листом**. Если последующих узлов два, все потомки одного из них можно назвать **левым поддеревом**, а все потомки другого — **правым поддеревом**. Если последующий узел один, он может начинать левое или правое поддерево.

В **идеально сбалансированном** дереве все пути имеют равную длину. В **несбалансированном** левое поддерево может быть меньше правого. Это усложняет поиск.



Идеально сбалансированное двоичное дерево поиска. Сиреневым выделен корень, зелёным — листья. Примерно половина всех узлов — листья

В контейнерах `map` и `set` используются деревья поиска. В `set` узел хранит значения, а в `map` пары ключ-значение.

Контейнер должен определять, какой элемент больше, а какой меньше. Это делает компаратор. Компаратор и хеш-функция задаются шаблонным параметром.

Компаратор по умолчанию `less` требует, чтобы ключи контейнера можно было сравнивать операцией `<`. Достаточно определить эту операцию — и можете использовать тип в `set` или в качестве ключа в `map`.

Галопом по map'ам и set'ам

Контейнеры `map` и `set` имеют метод `upper_bound`. Также есть одноимённый алгоритм. Но алгоритм работает намного дольше.

Концепция **LegacyRandomAccessIterators** (совместимый итератор произвольного доступа) объединяет итераторы произвольного доступа, то есть такие, которым можно прибавить число, и операция прибавления будет очень быстрой — константной.

Итераторы в `unordered_map` и `unordered_set` не поддерживают произвольный доступ и перемещение назад.

Элементы в `unordered_map` и `unordered_set` хранятся в корзинках. В одной корзинке часть скапливается несколько элементов. Хранение значений в корзинках реализуется внутри библиотеки обычно в виде односвязного списка.

В этом случае итератор может переходить вперёд: либо к следующему элементу по связи, либо в следующую непустую корзинку. Перемещение проще, чем в `set`, но ход назад запрещён. Значит, и

Итераторы в `unordered_map` и `unordered_set` **однонаправленные**: умеют шагать только вперёд.

А итераторы в `map` и `set` — **двунаправленные**.

В `unordered_map` и `unordered_set` нельзя использовать итератор, если он был сохранён до добавления нового элемента в контейнер. Но можно использовать ссылки или указатели на элементы даже после добавления.

В этом отношении обычные `map` и `set` удобнее: итераторы гарантированно остаются корректными после любых добавлений.