Основы разработки на C++. Эффективное использование ассоциативных контейнеров

Оглавление

Эффективное использование ассоциативных контейнеров

1.1 Введение в ассоциативные контейнеры

Ассоциативные контейнеры — это контейнеры, которые хранят значения по различным ключам. Например, контейнер `map` — это их типичный представитель. Пример:

```
map<string, int> counter;
```

В примере ключом является строка. Зная эту строку-ключ, вы можете **добавлять**:

```
counter["apple"] = 1;
```

модифицировать:

```
++counter["apple"];
```

искать:

```
cout << counter["apple"] << endl;
```

и **удалять данные** в контейнере:

```
counter.erase("apple");
```

Другим примером, еще более простым, можно считать контейнер `vector`, где ключи — это просто индексы, целые числа.

```
vector<double> values;
```

Для того чтобы познакомиться с другими, более эффективными контейнерами, давайте решим наглядную задачу. Для задачи нам нужен хороший, длинный текст. Мы выбрали текст про Шерлока Холмса. Давайте проанализируем его и узнаем, какие слова встречались в тексте чаще всего. Обратимся к нашему коду.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <unordered_map>
```

```
#include "head.h"
#include "profile.h"

using namespace std;
```

Мы завели контейнер map от типов `string, int`, чтобы считать количество строк, и текст записан в файле input.txt

```
int main() {
    map<string, int> freqs;
    ifstream fs("input.txt");
```

Давайте прочитаем его и положим содержимое файла в контейнер `freqs` с частотами. Напишем цикл, мы должны это считывать в какую-то переменную строчки. Пока у нас файл не пустой, мы в контейнер `freqs` добавляем эту строчку, если ее нет. А если есть, то увеличиваем значения.

```
    string text;
    while (fs >> text) {
        ++freqs[text];
    }
```

Давайте распечатаем содержимое этого контейнера в цикле и убедимся, что там лежит то, что мы ожидаем. Для этого воспользуемся функцией `head`, которая объявлена в заголовочном файле `head.h`, вы ее можете помнить из наших прошлых курсов. Мы ее использовали затем, чтобы распечатать только первые десять элементов из нашего контейнера.

```
    for (const auto& [k,v] : Head(freqs, 10)) {
        cout << k << '\t' << v << endl;
    }
    cout << endl;
```

Давайте соберем нашу программу и запустим. Что мы видим? Что у нас действительно есть список слов. И у нас есть частоты этих слов, то есть мы знаем, сколько раз каждое слово встречалось в тексте. И это здорово, однако мы хотим решить не ту задачу.

Здесь у нас слова отсортированы по алфавиту, потому что у нас в контейнере `map` ключом является строка, а нам нужно отсортировать их по частоте, чтобы выбрать самые часто используемые. Как нам это сделать?

Давайте просто переложим эти слова из `map` в вектор `pair`. Почему `pair`? Потому что у нас в контейнере `map` хранятся пары ключ/значение. Мы эти самые пары ключ/значение, собственно такого ровно типа, переложим в вектор `words` и скопируем в него все эти пары из контейнера `freqs` от начала до конца.

```
    vector<pair<string, int>> words(freqs.begin(), freqs.end());
```

Затем мы отсортируем этот вектор в нужном нам порядке. Напишем компаратор, который сравнивает частоты. И таким образом после сортировки этого вектора в его начале будет лежать то, что нам необходимо — самые часто используемые слова.

```
    sort(words.begin(), words.end(), [](const auto& l, const auto& r) {
        return l.second > r.second;
    }
```

Что же, давайте выберем эти слова. Возьмем наш цикл, который печатает, и распечатаем, правда, не контейнер `freqs`, а распечатаем вектор `words`, который у нас к этому моменту уже отсортирован.

```
    for (const auto& [k,v] : Head(words, 10)) {
        cout << k << '\t' << v << endl;
    }
    cout << endl;
}
```

Компилируем и запускаем программу, смотрим, что получилось. В первом выводе то, что у нас лежит в контейнере `map`, это первые десять слов по алфавиту. Они отсортированы по ключу. Во втором — это слова, которые отсортированы по частотам. Вот у нас самое часто используемое слово — это артикль the. И дальше идут предлоги, другие артикли, местоимения, собственно, ожидаемо. Так и должно быть в английском языке. Мы можем сделать предположение, что мы сейчас правильно решили нужную нам задачу. Далее мы узнаем, как решить эту задачу эффективнее.

1.2 Размен отсортированности на производительность

Мы собираемся решить эту задачу более эффективно. Что значит более эффективно? Значит мы хотим, чтобы решение работало быстрее, чем у нас сейчас есть, и давайте для начала замерим, сколько же по времени работает наше имеющееся решение. Для этого возьмем код, написанный выше, и для измерения используем макрос `LOG_DURATION`, который вы знаете из прошлых курсов. Этот макрос определен в заголовочном файле `profile.h`.

Здесь мы хотим посмотреть, какие этапы были в нашем решении, и сколько по времени работал каждый этап. Первый этап можем выделить, это когда мы заполняли контейнер `map` словами из файла.

```
    {
        LOG_DURATION("Fill");
        ifstream fs("input.txt");
        string text;
        while (fs >> text) {
            ++freqs[text];
        }
    }
```

И что у нас идет дальше? Дальше мы делали две операции: заполняли вектор копиями из `map` и затем сортировали вектор. Давайте замерим, сколько у нас занимала оставшаяся часть тоже.

```
    LOG_DURATION("Copy and Sort");
    vector<pair<string, int>> words(freqs.begin(), freqs.end());
```

```
sort(words.begin(), words.end(), [](const auto& l, const auto& r) {
    return l.second > r.second;
}
```

Компилируем программу, запускаем ее. Что мы видим? Вот у нас программа работает как раньше. Слова, слова отсортированные в другом порядке, заполнение у нас заняло около секунды, а вот пересортировка почти нисколько, то есть у нас получается почти все время программа заполняла `map` словами из файла.

То есть, если мы хотим ускорить выполнение программы, нам нужно ускорять в первую очередь заполнение контейнера словами из файла, а все остальное оно и так быстро работает.

Давайте посмотрим где же мы тут можем попробовать сэкономить. Мы видим, что слова в контейнере `map`, упорядочены по алфавиту, по ключу, ключ у нас это строка, именно поэтому они упорядочены по алфавиту. Мы не хотим, собственно, иметь эти слова отсортированные по алфавиту, потому что нам этот порядок вообще не интересен. Мы их потом все равно пересортируем в нужном нам порядке по частоте, поэтому было бы здорово, если бы мы смогли от этой сортировки избавиться в пользу какого-нибудь другого контейнера, который делает то же самое, но не тратит время на сортировку, и оказывается, такой контейнер действительно есть, и он называется ожидаемым именем. Он называется `unordered_map`, и чтобы его использовать в этой программе, достаточно просто изменить тип `map` на `unordered_map`.

```
unordered_map<string, int> freqs;
```

Давайте скомпилируем, проверим что все собралось. Изменив всего лишь тип контейнера, мы получили работоспособную программу, запускаем ее и смотрим, что же получилось. Во-первых, к счастью ответ получился точно такой, как нам нужен. Это точно такой же ответ как был в прошлый раз, это самые частотные слова, упорядоченные по чистоте в порядке убывания, вот мы видим тот же самый список предлогов, артиклей и так далее, а вот те слова, которые были получены из файла. Мы видим, что это наверное те же самые слова, только распечатанные сейчас совсем в другом порядке. Они сейчас не упорядочены по алфавиту, и по частоте не упорядочены, они вообще никак не упорядочены, но нам это и не важно, нам тут никакой порядок не нужен, потому что потом мы их скопируем и отсортируем как нам нужно. Но что здесь хорошо? То, что у нас заполнение контейнера стало работать почти в три раза быстрее.

Что у нас в итоге получилось сейчас? Мы сравнили два способа решения задачи:

- один способ — это используя контейнер `map`, когда мы читаем слова из файла, складываем их в контейнер map, потом копируем вектор и сортируем в нужном нам порядке, и такое решение работает по продолжительности около секунды.
- Второе решение, аналогичное, только в качестве контейнера мы используем **не map**, а `unordered_map`. Все остальное то же самое, результат — точно такое же. Но оно работает почти в три раза быстрее и это хорошо, это нам подходит намного больше.

1.3 Внутреннее устройство ассоциативных контейнеров

Мы узнали, что контейнер `unordered_map` в некоторых случаях работает быстрее, чем обычный `map`. Разберем почему он быстрее. Для этого мы решим задачу, в которой нам необходимо реализовать свой простой ассоциативный контейнер, а именно электронную копилку. Мы хотим уметь считать сколько купюр разного номинала у нас накопилось. Давайте напишем код и посмотрим, что у нас получается.

```
#include <iostream>
#include <vector>
#include <array>

using namespace std;

int main() {
```

Значит, купюра у нас задается номиналом и пускай у нас все купюры приходят из входного потока.

```
    int nominal;
    while (cin >> nominal) {

    }
```

Вот мы их все считали. Дальше, нам их надо куда-то сложить. Давайте для этого заведем вектор, и индексом в этом векторе будут номиналы купюр. Этот вектор мы вынуждены завести длиной по 5001, потому что у нас максимальная купюра это 5000 и еще плюс один, потому что индексация начинается с нуля.

```
    vector<int> cash(5001);
```

Когда мы вводим число с клавиатуры или с входного потока, мы складываем его в нашу копилку, то есть увеличиваем счетчик. Таким образом, мы сейчас считали все номиналы купюр из входного потока и положили их вектор. В итоге:

```
    vector<int> cash(5001);
    int nominal;
    while (cin >> nominal) {
        cash[nominal]++;
    }
```

Что делаем дальше? Дальше нам нужно напечатать, что же у нас получилось. Значит, пробежимся по всему вектору, и для тех купюр, которые присутствуют в копилке: присутствует — значит, что у них счетчик не равен нулю, мы напечатаем их количество.

```
    for (int i = 0; i < cash.size(); ++i) {
        if (cash[i] != 0) {
            cout << i << " - " << cash[i] << endl;
        }
    }
```

Запускаем, собрали. Где мы возьмем купюры? Они у нас заранее приготовлены и лежат во входном файле input.txt.

Вот мы видим, что у нас есть набор каких-то купюр. Запустим нашу программу на этом входном файле. Наша программа работает и она действительно подсчитывает для купюр их количество — то, что надо.

Казалось бы, то, что надо, но давайте посмотрим какие недостатки есть у нашей программы:

- Из плюсов отметим то, что **программа очень простая**, она просто использует номинал купюр в качестве индекса и делает инкремент в векторе. Все суперпросто, ошибиться невозможно.
- Какие недостатки? Мы для такой простой задачи выделили **вектор длиной 5001 — это чрезмерно много**, особенно, если мы вспомним, что различных купюр у нас намного меньше (всего лишь 8 в файле input.txt).

Представьте, что у нас максимальная купюра была бы не 5000, а просто какого-нибудь сумасшедше большого номинала, у нас бы даже памяти не хватило выделить такой большой вектор, как бы мы тогда решали задачу? Вернемся к коду, исправим 5001 на 8 и думаем: как же нам быть? Мы можем сложить разные купюры в вектор длины 8, если напишем функцию, которая вычисляет индекс для купюры: для каждого номинала она проверяет, какая купюра пришла на вход, и возвращает соответствующий индекс. Таких сравнений у нас должно быть, соответственно, 8, в них будут фигурировать все наши купюры, и соответственно, они будут возвращать индексы.

```
int GetIndex(int n) {
    if (n == 10) return 0;
    if (n == 50) return 1;
    if (n == 100) return 2;
    if (n == 200) return 3;
    if (n == 500) return 4;
    if (n == 1000) return 5;
    if (n == 2000) return 6;
    if (n == 5000) return 7;
}
```

Там где мы обращаемся к элементам вектора мы используем функцию получения индекса.

```
    while (cin >> nominal) {
        cash[GetIndex(nominal)]++;
    }
```

Казалось бы все, но единственное, что будет некрасиво, что мы будем распечатывать индекс от нуля до семи вместо обозначения купюры. Давайте заведем вспомогательный вектор, тоже длинны 8, в который сложим человеческие названия наших купюр. Это будут, конечно, те же самые номиналы.

```
    static array<int, 8> names = {
        10, 50, 100, 200, 500, 1000, 2000, 5000
    };
```

```
    for (int i = 0; i < cash.size(); ++i) {
        if (cash[i] != 0) {
            cout << names[i] << " - " << cash[i] << endl;
        }
    }
```

Запускаем, собрали. Отлично, теперь мы видим, что программа работает, что она возвращает тот же самый результат, который мы добились с самого начала. И мы сэкономили очень большое количество памяти написав функцию получения индекса.

Это отлично, более того, из кода становится понятно, что мы можем задавать купюры и другим способом, например, используя их названия, а не числовой номинал. Для нас задача практически не меняется, мы просто перепишем функцию получения индекса, где будем сравнивать строки вместо чисел, и таким образом, мы сможем использовать строковые названия в качестве ключей.

Более того, вместо купюр мы можем решать аналогичную задачу для произвольного набора строк. Все что нам понадобится сделать это написать функцию перевода этих строк в индекс. Например, если мы сможем написать такую функцию для набора цветов, то мы сможем использовать эти цвета в качестве ключей.

```
int GetIndex(int n) {
    if (n == "orange") return 0;
    if (n == "red") return 1;
    if (n == "yellow") return 2;
    ...
}
```

Для такой функции получения индекса есть специальное название, она называется **хеш-функцией**. Мы увидели, из примеров, как хеш-функция помогает нам использовать строки в качестве ключей, отображая их в индексы. Что же дальше? Дальше мы отметим, что хеш-функция может, в общем случае, отображать в индексы не только строки, но и произвольные объекты.

Рассмотрим в качестве объекта российский автомобильный номер, он состоит из нескольких цифр, букв и номера региона. Предложим простую хеш-функцию, которая определяет индекс автомобильного номера, как число из его середины. С помощью нее мы можем раскладывать номера по тысяче разных корзин. Обратите внимание, что всевозможных автомобильных номеров существует очень много, их никак не получится разложить всего лишь по тысяче корзин без повторений. Если два номера отличаются между собой только буквами или регионом, то они попадают в одну корзину, такая ситуация называется **коллизией**.

1.4 Внутреннее устройство unordered_map и unordered_set

Изучим, как внутри устроены контейнеры `unordered_map` и `unordered_set`. Они основаны на так называемых **хеш-таблицах**. Давайте рассмотрим как хеш-таблица устроена. Она состоит из нескольких корзин, по которым раскладываются объекты, и как мы уже знаем, для получения индекса корзины применяется хеш-функция. Это все выглядит очень просто, пока мы не вспомним о существовании коллизии.

Попытаемся добавить в хеш-таблицу автомобильный номер, для которого хеш-функция выдает

индекс корзины 227. Пусть эта корзина уже занята другим номером с другими буквами и из другого региона. Что же делать? Не нужно бояться. Существует несколько способов разрешения коллизий, и один из наиболее распространенных — **метод цепочек**. Этот метод заключается в том, что вместо самих объектов в корзинах хранятся списки объектов. При попытке вставить объект в корзину, сначала проверяется, нет ли его уже в списке, и если нет, то объект добавляется. Таким образом в случае возникновения коллизий в корзине оказывается более одного объекта.

Таким образом цепочки могут удлинятся и удлинятся при вставках, но обычно на практике такие цепочки не становятся слишком длинными. Те хеш-таблицы, которые широко распространены, устроены таким образом, чтобы цепочки оставались короткими.

Сейчас мы рассмотрели, что происходит при вставке объекта в хеш-таблицу. Другие операции — поиск и удаление — работают похожим образом. **Все операции состоят из двух основных этапов**:

1. Сначала с помощью хеш-функции для объекта вычисляется индекс корзины;
2. Если корзина не пуста, происходит последовательный поиск объекта по списку (объект сравнивается со всеми имеющимися).

Теперь давайте поговорим о сложности операций в хеш-таблицах, таких как вставка, поиск и удаление. Это **сложность складывается из двух основных слагаемых**:

1. Вычисление хеш-функции — $O(1)$
2. Поиск объекта в корзине. В среднем — $O(1)$

Итак, мы узнали, что в основе `unordered`-контейнеров лежат хеш-таблицы и рассмотрели, как они работают: `unordered_set` хранит в хеш-таблице ключи, а `unordered_map` хранит в них пару: ключ-значение.

1.5 Внутреннее устройство map и set

Теперь давайте рассмотрим контейнеры `map` и `set` и разберемся, почему они работают медленнее. В документации написано, что внутри `map` представляет из себя **сбалансированное двоичное дерево поиска (красно-черное дерево)**. Каждое из этих слов важно; разберемся, что они все значат.

- Двоичное дерево — у каждого узла дерева может быть не более двух потомков;
- Поиска — объекты в дереве упорядочены. Все значения узлов поддерева, меньшие данного узла, попадает в его левое поддерево, а все значение большие данного узла — в правое.
- Сбалансированное — для каждого узла высоты его левого и правого поддеревьев примерно равны. В некоторых реализациях, например, они могут отличаться не более чем на один.

Давайте посмотрим, как же работает поиск в бинарных деревьях поиска. Работает он очень просто: если мы вспомним о том, что узлы упорядочены. Если мы начинаем искать какое-то значение, мы сравниваем его всегда сначала с корнем. Если значение меньше, чем значение в корне, то мы уходим в левое поддерево. Если там мы натыкается на число большее, то мы уходим в правое поддерево и так далее, мы опускаемся все ниже и ниже по дереву, пока не найдем то число, которое нас интересует.

Так же устроен поиск числа, которого в бинарном дереве на самом деле нет. Мы спускаемся все ниже и ниже, и когда доходим до крайних листьев и дальше идти нам некуда, а число мы все еще не нашли, ну значит этого значения в дереве нет, поиск завершился неудачей.

Похожим образом работает вставка. Мы спускаемся ниже и ниже, пока не находим место, где число могло бы быть. Оно могло бы там быть, но его пока нет, поэтому мы со спокойной совестью его туда вставим. Все довольны, у нас получается очень красивое дерево.

Но вы можете задуматься, а что будет, если при серии вставок дерево перекосит, и оно перестанет быть сбалансированным. К счастью на практике такие деревья имеют механизмы балансировки. Суть его в том, что имеющиеся узлы переупорядочиваются таким образом, чтобы заполнить поддеревья равномерно.

Поговорим о **сложности операции в двоичных деревьях поиска**, таких как вставка, поиск и удаление. Во всех этих операциях в худшем случае мы должны пройти путь от корня дерева до какого-то его листа. Вспомнив о том, что деревья сбалансированы, мы можем оценить высоту дерева, как логарифм от количества элементов в нем. Получается, что путь от корня до листа длиной в логарифм.

- Все операции работают за $O(h)$, где h — это высота дерева
- Если дерево сбалансированное, то $h \sim \log_2 N$
- find $\sim \log_2 N$; insert $\sim \log_2 N$; delete $\sim \log_2 N$;

Итак, мы узнали, что в основе контейнеров `map` и `set`, лежат сбалансированные двоичные деревья поиска и рассмотрели как они работают. Собственно, `set` хранит в таком дереве ключи, а `map` хранит там пару: ключ-значение.

Вспомнив о том, что в `unordered`-контейнерах сложности этих же операций в среднем константные, мы понимаем, почему они быстрее.

1.6 Итераторы в map. Почему лучше использовать собственные методы для поиска

Рассмотрим, как работают итераторы в `map`. Для начала установим итератор на `begin`.

```
auto it = m.begin();
```

Мы знаем, что когда мы итерируемся по целому контейнеру `map`, мы получаем отсортированные по возрастанию элементы. Поэтому логично предположить, что `begin` указывает на минимальный элемент, то есть на самый левый узел.

При **инкременте итератора**

```
++it;
```

происходит понятная вещь: мы переходим на следующий по возрастанию элемент, производя обход дерева, и так далее. Каждый вызов оператора инкремента будет нас продвигать все к большим и большим значениям. И мы будем постепенно обходить дерево дальше и дальше.

Аналогично происходит и **декремент итератора**, только в обратном порядке, то есть мы будем двигаться в сторону уменьшения элементов.

Вы можете задаться вопросом: а не являются ли тогда вызовы операторов инкремента и декремента тяжелыми? Ведь некоторые очередные сдвиги итераторов выглядят очень сложными, потому что требуется перескочить очень много уровней.

Давайте рассмотрим этот момент подробнее. Представим, что нам надо обойти все дерево, то есть проитерироваться от `begin` до `end`. Всего в дереве у нас N элементов, поэтому будет $(N-1)$ ребро, так как каждый элемент кроме корня имеет ровно одно входящее в него ребро. И каждое ребро мы проходим дважды по направлению туда и обратно. То есть всего для обхода всех N элементов мы сделаем порядка N проходов ребер. Получается, что в среднем на переход от элемента к элементу мы совершаем какую-то константную работу. Поэтому в среднем вызовы операторов инкремента и декремента стоят недорого, несмотря на то, что некоторые из них тяжелые.

Вы можете спросить, а зачем же мы все это разбирали? Неужели только затем, чтобы сказать, что инкремент работает за константное время в среднем? Но не только. Оказывается, что это знание может пригодиться в весьма неожиданном месте, а именно, при выборе одного из двух похожих вызовов поиска: через глобальный алгоритм

```
lower_bound(begin(m), end(m), ...);
```

либо через встроенный метод

```
m.lower_bound(...);
```

Давайте напишем код и посмотрим, к какой разнице это приводит. У нас есть код, вы его можете помнить по задаче о Шерлоке Холмсе. У нас есть какое-то произведение о нем, оно лежит в файле, мы считываем из него все строки и складываем в `set`.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <unordered_map>
#include "head.h"
#include "profile.h"

using namespace std;

int main() {
    set<string> s_words;

    ifstream fs("input.txt");
```

```
    string text;
    while (fs >> text) {
        s_words.insert(text);
    }
    for (const auto& s : Head(s_words, 5)) {
        cout << s << << endl;
    }
    cout << endl;
}
```

Что мы хотим сделать? Мы хотим вызвать методы поиска в `lower_bound` двумя способами. Сначала вызвать метод `lower_bound` у `set`, а затем глобальный алгоритм `lower_bound`.

Вызываем их с одними и теми же входными данными, то есть пробегаемся по всем буквам от A до Z и ищем слова на эти буквы. Тесты одинаковые, но посмотрим, с какой скоростью они работают.

```
    {
        LOG_DURATION("set::lower_bound method");
        for (char c = 'a'; c < 'z'; ++c) {
            string s(1,c);
            s_words.lower_bound(s);
        }
    }

    {
        LOG_DURATION("global lower_bound in set");
        for (char c = 'a'; c < 'z'; ++c) {
            string s(1,c);
            lower_bound(begin(s_words), end(s_words), s);
        }
    }
```

Что мы видим? Что хоть они и работают одинаково, но время требуют очень разное. **Метод** `lower_bound` выполняется почти мгновенно, а **глобальный** алгоритм `lower_bound` работает сильно дольше, вполне ощутимое время. Мы можем сделать вывод: а может, это у нас глобальный `lower_bound` такой плохой, может быть, он всегда так работал? Но нет, вы же помните из предыдущих курсов, что на отсортированном векторе он работает очень даже хорошо. И давайте это проверим сейчас. Возьмем вектор строк, скопируем в него строки из нашего set, он у нас уже сразу будет отсортированный, потому что в set слова отсортированы. Вызовем на этом векторе точно такой же тест и посмотрим, за какое время выполнится он.

```
    vector<string> v_words(begin(s_words), end(s_words));

    for (const auto& s : Head(v_words, 5)) {
        cout << s << << endl;
    }
    cout << endl;
```

```
{
    LOG_DURATION("global lower_bound in vector");
    for (char c = 'a'; c < 'z'; ++c) {
        string s(1,c);
        lower_bound(begin(v_words), end(v_words), s);
    }
}
```

Запускаем. Что мы видим? Слова вектора лежат те же самые, и работает он очень быстро. Он работает так же быстро, как вызов метода set. То есть намного быстрее, чем этот же алгоритм на контейнере set. Давайте разберемся, почему у нас такая большая разница во времени.

`lower_bound` **в отсортированном векторе использует двоичный поиск**, производя арифметические операции над итераторами. Это возможно, потому что итераторы в векторах являются итераторами произвольного доступа (двоичный поиск и итераторы рассматривались в «Желтом поясе»). Заметим, что **глобальный алгоритм** `lower_bound` **принимает на вход только пару итераторов и ничего не знает об устройстве контейнера**. Поэтому становится важно, насколько мощны эти итераторы, могут ли они позволить, например, обращаться к произвольному элементу между ними. В случае map вряд ли, это видно из структуры дерева. Представьте, что у нас есть два итератора — на `begin` и на `end`, и мы хотим найти элемент ровно между ними посередине. Это довольно трудно сделать, потому что непонятно, где он находится. Посередине как бы находится корень, но это совсем не та середина, это не значит, что элемент строго между `begin` и `end`.

Давайте посмотрим, что у нас итераторы в `set` действительно не такие крутые. Заведем итератор в нашем контейнере, возьмем, скажем, `begin`, пока у нас все хорошо. Но если мы прибавим к нему любое произвольное число, мы просто увидим, что **этот код не компилируется**.

```
auto it = s_words.begin() + 5;
```

Нам компилятор подсказывает, что оператор + не может быть применен к такому итератору.

Мы поняли, что итераторы в `map` не позволяют производить над собой арифметические операции. Они умеют делать только инкремент и декремент. Такие итераторы называются **двунаправленными**. Поэтому для map **глобальный алгоритм** `lower_bound` применяет линейный поиск по порядку. В подтверждение наших слов мы можем открыть документацию для глобального алгоритма `lower_bound` и убедиться, что **он гарантирует логарифмическую сложность только для итераторов произвольного доступа. В противном случае сложность будет линейной**. Заметим, что **встроенный метод** `lower_bound` в отличие от глобального алгоритма знает о внутреннем устройстве контейнера `map`, и это позволяет ему работать за логарифмическое время.

1.7 Итераторы в unordered_map. Инвалидация итераторов в ассоциативных контейнерах

Сейчас мы поговорим о том, как работают итераторы в `unordered_map`, то есть в хеш-таблицах. Посмотрим, например, таблицы, и для начала установим итератор на `begin`. Он указывает на первый элемент списка в первой непустой корзине.

```
auto it = h.begin();
```

Инкремент итератора происходит очень просто, это просто итерирование по односвязному списку.

```
++it;
```

Если при очередном инкременте мы дошли до конца цепочки, то потом мы переходим далее, в следующую непустую корзину, и в том и в другом случае инкремент будет стоить очень дешево. Нам надо лишь перейти по показателям на следующий элемент либо перейти в следующую корзину. Согласитесь, это намного проще, чем вычисления на деревьях.

И казалось бы с декрементом все должно быть аналогично, но нет, оказывается мы вообще **не можем вызывать оператор декремента для таких итераторов**.

То есть итераторы в хеш-таблицах еще менее мощны чем в двоичных деревьях поиска. Для них есть специальное название, **последовательные**, либо, по-простому — **forward-итератор**.

Кажется что мы уже закончили разбираться с итераторами в хеш-таблицах, они же заметно проще, чем итераторы в деревьях, но не совсем, есть важный нюанс: помните, мы затрагивали важное свойство хеш-таблиц, что их цепочки должны быть короткими. Для поддержания этого свойства иногда может случаться **рехеширование**, перекладывание существующих элементов по другим корзинкам, чтобы не держать их переполненными.

Давайте задумаемся, что же будет с порядком элементов хеш-таблицы после рехеширования. Вообще, судя по названию `unordered_map`, элементы и так не были упорядоченны, а мы еще и разложим их по корзинам по-новому.

А что же будет с итераторами после рехеширования? Можно ли ими продолжать пользоваться. И для ответа на этот вопрос обратимся к документации. Откроем cppreference и посмотрим, что нам говорит стандарт. Действительно, при вставке элементов в хеш-таблицу может случиться рехеширование, и в этом случае все итераторы **инвалидируются**, то есть перестают указывать туда, куда должны, и их использование может привести либо к падению программы, либо к неверному результату, либо к неопределенному поведению.

Заметим, что итераторы в контейнере `map`, то есть в двоичном дереве поиска, более устойчивы и живучи в этом смысле. Даже когда при вставке нового элемента в дерево происходит перебалансировка и какой-то элемент, на который указывал итератор, смещается — проблемы нет, он просто меняет связи с соседями, но с ним можно продолжать работать далее.

Давайте еще раз опишем инвалидацию итераторов в `unordered`-контейнерах:

- использовать имеющиеся итераторы после вставки в хеш-таблицу — плохая идея, потому что они могут инвалидироваться;
- удалять можно не опасаясь за итераторы других элементов (при этом, конечно, сам удаленный итератор инвалидируется).

1.8 Использование пользовательских типов в ассоциативных контейнерах

Давайте разберем, как помещать свои собственные типы данных в ассоциативные контейнеры. Для этого напишем какой-нибудь свой собственный тип данных, который будет описывать автомобильный номер:

```
#include <iomanip>
#include <iostream>
#include <tuple>

using namespace std;

struct Plate {
    char C1;
    int Number;
    char C2;
    char C3;
    int Region;
};
```

Напишем для этой структуры оператор помещения в поток, для того, чтобы мы могли распечатывать автомобильные номера

```
ostream& operator << (ostream& out, const Plate& p) {
    out << p.C1;
    out << setw(3) << setfill('0') << p.Number;
    out << p.C2;
    out << p.C3;
    out << setw(2) << setfill('0') << p.Region;
    return out;
}
```

Что мы хотим сделать дальше? Мы хотим завести ассоциативный контейнер `set` и складывать в него эти автомобильные номера

```
#include "generator.h"
#include "profile.h"
#include <set>
#include <unordered_set>

using namespace std;

int main() {
    PlateGenerator pg;
    set<Plate> s_plates;
    const int N = 10;
```

```
    return 0;
}
```

Для того, чтобы генерировать случайные номера, мы написали генератор, в котором есть функция `GetRandomPlate`, которая возвращает случайный автомобильный номер.

```
#include "plates.h"
#include <array>
#include <random>

class PlateGenerator {
public:
    Plate GetRandomPlate() {
        Plate p;
        p.C1 = Letters[LetterDist(RandEng)];
        p.Number = NumberDist(RandEng);
        p.C2 = Letters[LetterDist(RandEng)];
        p.C3 = Letters[LetterDist(RandEng)];
        p.Region = RegionDist(RandEng);
        return p;
    }

private:
    const static int N = 12;
    const array<char, N> Letters = {
        'A', 'B', 'C', 'E', 'H', 'K', 'M', 'O', 'P', 'T', 'X', 'Y'
    };

    default_random_engine RandEng;
    uniform_int_distribution<int> LetterDist{0, N - 1};
    uniform_int_distribution<int> NumberDist{1, 999};
    uniform_int_distribution<int> RegionDist{1, 99};
};
```

Давайте проверим, что это работает

```
    cout << pg.GetRandomPlate();
```

Собираем программу, запускаем — успешно.

Теперь возьмем 10 случайных автомобильных номеров и поместим их в контейнер `set`. Но **код не компилируется**!

```
    for (int i = 0; i < N; ++i) {
        s_plates.insert(pg.GetRandomPlate());
    }
```

Компилятор говорит, что не хватает оператора сравнения $<$ (меньше). Зачем же нам нужен этот оператор, если мы просто хотим поместить номера в контейнер? Для ответа на этот вопрос вспомним, что `set` устроен как красно-черное дерево, а значит элементы в нем упорядочены,

поэтому нужно уметь сравнивать элементы друг с другом.

Давайте напишем оператор $<$ (меньше) для расположения элементов в лексикографическом порядке

```
bool operator < (const Plate& l, const Plate& r) {
    return tie(l.C1, l.Number, l.C2, l.C3, l.Region) < tie(r.C1, r.Number, r.C2, r.C3,
        r.Region);
}
```

Теперь программа собралась успешно. Давайте распечатаем полученные 10 автомобильных номеров

```
    for (const auto& p : s_plates) {
        cout << p << endl;
    }
    cout << endl;
```

Теперь мы видим 10 номеров, отсортированных в лексикографическом порядке.

Решим эту же задачу с `unordered_set`:

```
#include "generator.h"
#include "profile.h"
#include <set>
#include <unordered_set>

using namespace std;

int main() {
    PlateGenerator pg;
    set<Plate> s_plates;
    unordered_set<Plate> h_plates;
    const int N = 10;

    for (int i = 0; i < N; ++i) {
        s_plates.insert(pg.GetRandomPlate());
    }
    for (const auto& p : s_plates) {
        cout << p << endl;
    }
    cout << endl;

    for (int i = 0; i < N; ++i) {
        h_plates.insert(pg.GetRandomPlate());
    }
    for (const auto& p : h_plates) {
        cout << p << endl;
    }
    cout << endl;
```

```
    return 0;
}
```

У нас **снова сообщение об ошибке**! Ошибка связана с тем, что нам не хватает чего-то связанного с хешированием. Давайте напишем хеш-функцию для нашего `unordered_set`, которая будет возвращать целое число для автомобильного номера.

```
struct PlateHasher {
    size_t operator() (const Plate& p) const {
        return p.Number;
     }
};

int main() {
    ...
    unordered_set<Plate, PlateHasher> h_plates;
    ...
```

У нас **снова сообщение об ошибке**! Компилятор говорит, что нам не хватает оператора сравнения. А зачем нам нужен оператор сравниения, если мы просто хотим поместить элементы в хеш-таблицу? Конечно же он нужен! При поиске элементов в хеш-таблице каждый элемент сравнивается с теми, которые уже находятся в хеш-таблице. Давайте напишем оператор сравнения:

```
bool operator == (const Plate& l, const Plate& r) {
    return (l.C1 == r.C1) && (l.Number == r.Number) && (l.C2 == r.C2) && (l.C3 == r.C3) &&
        (l.Region == r.Region);
}
```

Теперь программа успешно компилируется. В итоге мы видим, что нам вывелись 10 номеров из двух контейнеров.

Таким образом,

- если вы хотите поместить произвольный тип в `map` или `set`, то необходимо написать для него оператор сравнения `<` (меньше);
- если вы хотите поместить произвольный тип в `unordered`-контейнер, то необходимо для него определить хеш-функцию и оператор сравнения `==`.

1.9 Зависимость производительности от хеш-функции

Давайте узнаем, как правильно следует писать свою собственную хеш-функцию для хеш-таблицы. Замерим производительность: насколько эффективно наши конструкции работают.

```
...
int main() {
    PlateGenerator pg;
```

```
    set<Plate> s_plates;
    unordered_set<Plate> h_plates;
    const int N = 50000;

    {
        LOG_DURATION("set");
        for (int i = 0; i < N; ++i) {
            s_plates.insert(pg.GetRandomPlate());
        }
        for (int i = 0; i < N; ++i) {
            s_plates.find(pg.GetRandomPlate());
        }
    }
    {
        LOG_DURATION("unordered_set");
        for (int i = 0; i < N; ++i) {
            h_plates.insert(pg.GetRandomPlate());
        }
        for (int i = 0; i < N; ++i) {
            h_plates.find(pg.GetRandomPlate());
        }
    }

    return 0;
}
```

Видим, что у нас тест продолжает работать, но мы наблюдаем какой-то странный эффект. Мы видим, что наш `unordered_set` работает медленнее почти в два раза, чем обычный `set`. А это странно, мы ведь с вами знаем из предыдущих занятий, что хеш-таблица работает очень быстро. Значит, у нас что-то пошло не так. Давайте попытаемся это объяснить.

Как мы знаем, у нас для разрешения коллизий в хеш-таблицах используется метод цепочек. И мы с вами говорили, что важно, чтобы такие цепочки были короткими. Давайте посмотрим, есть ли надежда в нашем коде, что цепочки будут короткими. Такой надежды, получается, что у нас и нет. Потому что у нас эксперимент состоит из 50000 различных автомобильных номеров, а в качестве номеров корзин, то есть в качестве значений хеш-функции, мы используем только числа от 1 до 1000. То есть у нас есть всего лишь 1000 различных корзин для 50000 автомобильных номеров. Этого явно недостаточно. И у нас будет очень много коллизий. Поэтому производительность `unordered_set` и снижается так сильно. Надо что-то предпринять. Каким образом?

Нам нужно использовать более, чем эти 1000 корзин. Мы ведь можем написать более хитрую хеш-функцию. И мы можем написать ее следующим образом. Можем использовать не только центральное число из автомобильного номера, но еще и код региона. С помощью нехитрой арифметической магии мы можем взять номер, умножить его на 100 и прибавить регион. Таким образом, у нас получится пятизначное число, и мы это пятизначное число уже сможем использовать в качестве значения хеш-функции. Давайте это и сделаем. Перепишем наш хеш и посмотрим, изменится ли от этого производительность

```
struct PlateHasher {
```

```
    size_t operator() (const Plate& p) const{
        size_t result = p.Number;
        result *= 100;
        result += p.Region;
        return result;
    }
};
```

Компилируем. Запускаем. Получилось неплохо, у нас `unordered_set` начал выигрывать у `set`. И значит, мы на верном пути. Мы делаем все правильно и правильно понимаем, как работает хеш-таблица.

Но давайте усугубим наш эксперимент. Возьмем побольше, не 50 000 тестовых номеров, а миллион. Что-нибудь изменится или нет? Собираем. Запускаем. `set` у нас отработал за две секунды, `unordered_set` — больше, чем за две секунды. То есть у нас, когда тест становится серьезнее, `unordered_set` все-таки проигрывает по производительности. А мы знаем из теории, что не должен. И опять-таки понятно почему.

Потому что у нас различных корзин 100000 получается (у нас пятизначные числа используются), а тест состоит из миллиона номеров! И у нас опять много коллизий. Окончательно решить эту проблему можно, только использовав всю информацию, которая имеется на госномере. То есть кроме чисел с госномера использовать еще и буквы. Вы спросите: «А как же нам буквы превратить в значение хеш-функции? Ведь буквы — это буквы». Ничего страшного. Мы можем буквы конвертировать в их порядковый номер в алфавите: отняв букву A, например, можно получить порядковый номер. И затем эти порядковые номера использовать в такой арифметической магии. И таким образом получить для этого автомобильного номера большое число. Ну что ж, давайте сделаем это.

```
struct PlateHasher {
    size_t operator() (const Plate& p) const{
        size_t result = p.Number;
        result *= 100;
        result += p.Region;

        int s1 = p.C1 - 'A';
        int s2 = p.C2 - 'A';
        int s3 = p.C3 - 'A';
        int s = (s1*100 + s2)*100 + s3;

        result *= 1000000;
        result += s;

        return result;
    }
};
```

Компилируем и запускаем. Супер. Мы видим, что наш `unordered_set` стал почти в два раза быстрее. Мы сделали все, что от нас требовалось. Мы использовали всю информацию с госномера,

и поэтому у нас получилась отличная хеш-функция.

Таким образом, качество хеш-функции очень сильно влияет на производительность хеш-таблицы, то есть на производительность контейнера `unordered_set`. И хорошая хеш-функция, которую вы пишете для достижения хорошей производительности, должна охватывать широкий диапазон корзин, а также по возможности равномерно распределять по ним объекты.

1.10 Рекомендации по выбору хеш-функции

Давайте напишем еще какую-нибудь собственную структуру. Поместим в нее поля и попробуем написать хешер для нее.

Что там будет? Ну давайте для начала поместим туда число типа `double`, и у нас сразу встает вопрос: как же мы будем писать для хеш-функцию для `double`? Ведь **значения хеш-функции — это целые числа**, а `double` — это дробное число и не очень понятно, как его однозначно перевести. Но еще больше было бы непонятно, если бы у нас было вообще не число, а какая-нибудь строка. Как нам переводить строку в хеш-функцию в число? Неужели снова писать такую же сложную хеш-функцию, как для автомобильных номеров? И это было бы удивительно, потому что тогда мы бы не смогли положить в `unordered_set` по умолчанию ни double, ни string. Но давайте убедимся, что мы на самом деле можем это делать спокойно, без написания собственных хеш-функций. Вот мы создаем `unordered_set` из переменных типа `double` и код компилируется прекрасно, и если мы заменим на `string`, он тоже прекрасно компилируется.

```
...
struct MyType {
    double d;
    string str;
};

int main() {
    unordered_set<double> ht1;
    unordered_set<string> ht2;
    return 0;
}
```

От нас не требуют написать свою собственную хеш-функцию для этих типов. Это значит, что для этих типов стандартные хеш-функции уже написаны, и мы можем их использовать из стандартной библиотеки. Они реализованы в шаблонных структурах `hash`, параметризованных каким-нибудь типом.

Давайте напишем хешер для нашей структуры, в которую поместим в поле тот самый автомобильный номер:

```
struct MyType {
    double d;
    string str;
    Plate plate;
};
```

```
struct MyHasher {
    size_t operator() (const MyType& p) const{
        size_t r1 = dhash(p.d);
        size_t r2 = shash(p.str);
        size_t r3 = phash(p.plate)
        // ax^2 + bx + c
        size_t x = 37;
        return (r1*x*x + r2*x + r3);
    }

    hash<double> dhash;
    hash<double> shash;
    PlateHasher phash;
};

int main() {
    unordered_set<MyType, MyHasher> ht2;
    return 0;
}
```

Все скомпилировалось, мы смогли все правильно сделать.

Таким образом, в качестве распространенных подходов написания хеш-функций для собственных типов можно отметить следующие:

- если ваш класс содержит поля стандартных типов, то для хеширования этих полей можно использовать стандартные хешеры из библиотеки;
- для некоторых типов могут быть уже написаны хешеры, например, вами; для комбинации нескольких хешей, часто используют их как коэффициенты при вычислении некоторого многочлена;
- после применения всех рассмотренных рекомендаций ваша хеш-функция должна давать равномерное распределение по корзинам.

1.11 map::extract и map::merge

Заведем `set` строк и положим в него для начала три каких-нибудь строчки. Напечатаем то, что в них находится, чтобы убедиться в том, что мы сделали все правильно.

```
...
int main() {
    set<string> ss;
    ss.insert("Aaa");
    ss.insert("Bbb");
    ss.insert("Ccc");
```

```
    for (const auto& el : ss) {
        cout << el << ' ';
    }
    cout << endl;
}
```

Что мы хотим сделать дальше? Мы хотим модифицировать дерево, которое хранит эти три строчки. Мы хотим, чтобы все строчки, которые хранятся в нем, начинались с маленькой буквы, а не с большой.

Давайте начнем, скажем, с первого элемента. Установим итератор на `begin`, он указывает на какую-то строчку, и попробуем эту строчку изменить.

```
    auto it = ss.begin();
    string& temp = *it;
    temp[0] = tolower(temp[0]);
```

Но компилятор сообщает нам об ошибке. Он говорит о том, что мы потеряли где-то константность. Что бы это могло значить? По-простому это говорит о том, что мы не можем изменить ключ объекта, который находится в дереве. Вот у нас итератор указывает на `begin`, и мы хотим изменить это значение, то есть мы хотим изменить значение объекта, который лежит в дереве. Но мы знаем, что мы не можем это просто взять и сделать, потому что **элементы у нас в дереве упорядочены**. И если мы захотим поменять ключ, то нам придется поместить этот объект в другое место в дереве. А работая с итератором, мы не можем этого сделать, потому что **итератор указывает на один элемент и про структуру всего дерева ничего не знает**.

Какие есть способы такую задачу решить? Мы можем взять эту строчку, скопировать из дерева, то есть взять копию, а не ссылку, и изменить. Дальше из дерева старое значение удалить и новое измененное значение вставить.

```
    auto it = ss.begin();
    string temp = *it;
    temp[0] = tolower(temp[0]);
    ss.erase(it);
    ss.insert(temp);
```

Так у нас задача решится, решение компилируется. Распечатаем снова дерево, чтобы убедиться, что произошло ожидаемое. Да, мы видим, что у нас строчка, которая раньше начиналась на A, сейчас начинается на a, то есть мы добились, чего хотели, и, казалось бы, задача решена, она очень простая. И в чем же тут подвох?

Давайте разберемся, какие же этапы решения у нас были. У нас сначала было дерево. Потом мы скопировали строку. Потом мы удалили старый объект из дерева, поработали с копией в памяти, скопировали это значение обратно в дерево, и эта копия у нас, получается, была лишней почти все время. Мы зря сделали два копирования: из дерева и в дерево, только для того, чтобы снаружи дерева модифицировать строчку. И, казалось бы, так придется делать всегда, ведь у нас ключи в дереве неизменяемые, это единственный способ. Но нужно понимать, что даже **такой единственный способ вам не подойдет, если у вас объекты очень тяжелые**, вы не хотите тратить на них копирование. **Либо эти объекты вообще нельзя скопировать, потому что они move only**, потому что их можно только перемещать, а копировать нельзя.

И давайте рассмотрим именно такой пример для иллюстрации. Возьмем строчки, которые нельзя скопировать, вы их должны помнить с прошлых задач нашего курса. Здесь у нас имеются конструкторы, унаследованные от строки. Подчеркнуто, что конструктор копирования запрещен, и подчеркнуто, что конструктор перемещения разрешен.

```
struct NCString : public string {
    using string::string;
    NCString(const NCString&) = delete;
    NCString(NCString&&) = default;
};
```

И давайте заведем сейчас дерево из таких некопируемых строк.

```
int main() {
    set<NCString> ss;
    ss.insert("Aaa");
    ss.insert("Bbb");
    ss.insert("Ccc");

    for (const auto& el : ss) {
        cout << el << ' ';
    }
    cout << endl;

    auto it = ss.begin();
    NCString temp = *it;
    temp[0] = tolower(temp[0]);
    ss.erase(it);
    ss.insert(temp);
}
```

Компилятор говорит, что мы не можем вставить в дерево строчку такого типа. Мы ее не можем даже скопировать наружу, потому что она не копируемая, а только перемещаемая. И мы бы хотели ее переместить именно поэтому, но как же перемещать из дерева?

Оказывается, такой способ есть. У дерева есть специальный метод, он называется `extract` и работает так, как нам нужно. Мы можем взять и извлечь из дерева даже не строчку, а весь узел целиком, то есть мы можем оторвать узел от дерева. Для этого воспользуемся словом auto, потому что узел будет иметь специальный тип, отличный от типа строки. У нас есть теперь какой-то node, который мы извлекли из дерева. И этот node внутри себя содержит нужную нам строчку. Мы сейчас можем взять эту строчку и вытащить ее для того, чтобы модифицировать. Заметьте, что мы вытаскиваем ее по ссылке, чтобы не делать лишнее копирование.

```
    auto it = ss.begin();
    auto node = ss.extract(it);
    string& temp = node.value();
```

И мы сейчас вытащили эту строчку по ссылке, произвели ее модификацию. Мы теперь можем это делать, потому что узел у нас находится не в дереве, а отдельно сам по себе. И затем мы хотим

поместить обратно этот узел. Удалять нам уже ничего не нужно, мы же весь узел вытащили уже, а чтобы вставить, нам нужно вставить не строчку, а этот узел целиком.

```
temp[0] = tolower(temp[0]);
ss.insert(move(node));
```

Программа успешно компилируется, она успешно работает. Она работает отлично, она делает то, что и раньше. Она модифицирует значение ключа без избыточного копирования.

В итоге, у нас было дерево из трех узлов, затем мы один узел просто оторвали наружу без копирования — это просто перемещение из дерева. В этом отдельно взятом узле мы изменили значение строки, потом мы вставили этот узел обратно в дерево. Заметьте, что здесь **не было никаких копирований**. Мы просто вытащили узел из дерева и затем вставили его в другое место, чтобы он связался с родителями и сыновьями по-другому.

Давайте рассмотрим еще один интересный пример — как это можно использовать еще удобнее. Допустим, что у нас есть два дерева. Одно дерево строк и другое дерево строк, два сета. Возьмем второй сет, заведем в нем новые объекты, и, допустим, мы хотим слить два этих дерева, взять и переместить все узлы из второго дерева в первое.

```
set<NCString> ss2;
ss2.insert("Xxx");
ss2.insert("Yyy");
ss2.insert("Zzz");
```

И мы можем это сделать! Мы можем вытаскивать каждый узел из дерева 2 и вставлять его в дерево 1, например, в цикле. Либо вместо этого, чтобы это делать не в цикле, использовать функцию `merge` и просто перенести все дерево 2 в дерево 1.

```
ss.merge(ss2);
```

Напечатаем то, что получилось после вызова merge. Шесть элементов. Супер. И, чтобы убедиться, что никаких копирований не было, давайте распечатаем содержимое дерева 2.

```
for (const auto& el : ss) {
    cout << el << ' ';
}
cout << endl;

for (const auto& el : ss2) {
    cout << el << ' ';
}
cout << endl;
```

Конечно, никаких копирований здесь быть не могло, у нас же конструктор копирования запрещен, и мы это видим: у нас распечатана пустая строка там, где мы печатали элементы дерева 2. Когда мы сливали, никаких копирований не произошло, мы просто в дереве связали элементы по-другому, указателями, и сэкономили кучу накладных расходов.

Таким образом, мы убедились, что нельзя напрямую изменять ключ объекта в контейнере `map`. Для того чтобы изменить этот ключ, мы вынуждены скопировать ключ во временный объект,

изменить его там, удалить из старого места, а потом скопировать временную копию обратно. Если мы хотим избежать промежуточных копирований, мы можем использовать функции `extract` и `insert`; а при необходимости перенести все элементы из одного дерева в другое, мы можем использовать метод `merge`.

1.12 Коротко о главном

Пожалуй, самое важное, что следует помнить, это то, что контейнеры `map` и `set` устроены внутри как двоичные деревья поиска, а их `unordered` версии — как хеш-таблицы. Это приводит к тому, что в `map` и `set` элементы хранятся в **отсортированном порядке**, а `unordered` никакой **порядок не гарантируется**.

Когда мы с вами говорили о производительности, то обнаружили, что `unordered`-контейнеры работают заметно быстрее. Операции вставки, поиска и удаления работают в них за **константное в среднем** время, в то время, как в обычных `map` и `set` — за **логарифм**.

Есть еще некоторые нюансы связанные с итераторами: по скорости работы они, можно считать, не отличаются, но в обычных `map` и `set` итераторы переживают операции вставки с последующими перебалансировками деревьев, а в хеш-таблицах может произойти рехеширование и тогда итераторы станут инвалидными. Учитывайте это при решении своих задач.