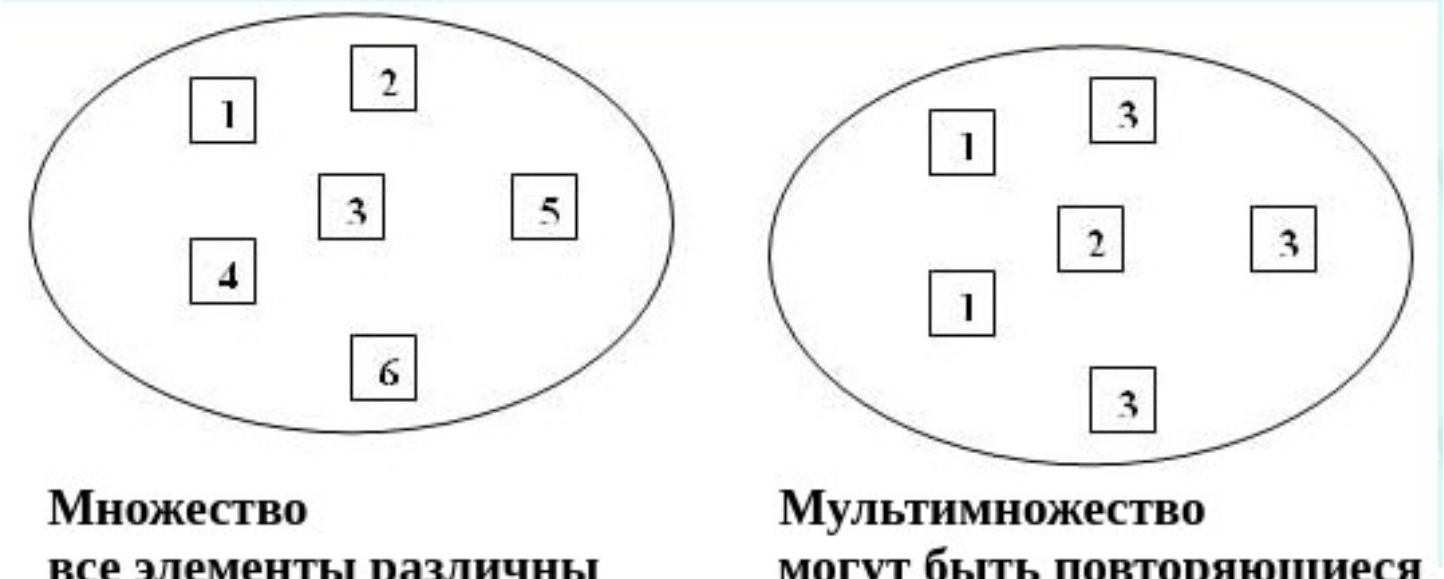
Ассоциативные контейнеры

Ассоциативные контейнеры — это упорядоченные коллекции, в которых позиция элемента зависит от его значения (или ключа, если элемент представляет собой пару ключ-значение)
set, multiset, map, multimap

Неупорядоченные контейнеры

Неупорядоченные контейнеры — позиция элемента не имеет значения, смысл имеет только принадлежность конкретного элемента такой коллекции.
unordered set, unordered multiset, unordered map, unordered multimap

Множество и мультимножество (set и miltiset)



Множество
все элементы различны

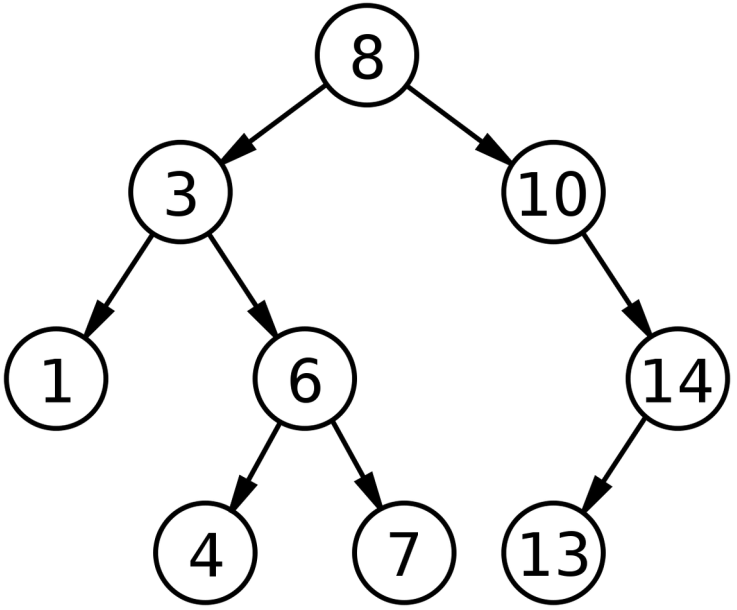
Мультимножество
могут быть повторяющиеся элементы

Ассоциативные контейнеры

Ассоциативные контейнеры автоматически сортируют свои элементы в соответствии с определенным значением. Элементы могут быть или значениями любого типа, или парой ключ-значение. Критерий сортировки может задаваться в виде функции, сравнивающей значения или ключи. По умолчанию контейнеры сравнивают элементы или ключи с помощью операции < .

Обычно ассоциативные контейнеры реализуются с помощью бинарных деревьев.

- Основное преимущество ассоциативных контейнеров в том, что поиск элемента с заданным значением выполняется за логарифмическое время.
- Недостаток ассоциативных контейнеров в том, что значения невозможно иззменять непосредственно, потому что при этом может быть нарушен порядок элементов.



Сортировка в ассоциативных контейнерах

Основное преимущество автоматической сортировки в том, что бинарное дерево допускает эффективный поиск конкретного значения. Функция поиска имеет логарифмическую сложность. Например, для поиска элементов в множестве или мультимножестве, состоящем из 1000 элементов, поиск по дереву в среднем требует в 50 раз меньше сравнений, чем при линейном поиске (который осуществляется алгоритмами поиска, обходящими все элементы).

В то же время автоматическая сортировка накладывает следующее ограничение на множества и мультимножества: значение элемента нельзя изменить непосредственно, потому что это может нарушить правильный порядок следования элементов.

Таким образом, для модификации элемента нужно удалить элемент, имеющий старое значение, и вставить элемент, имеющий новое значение.

Критерий сортировки

Критерий сортировки можно задать двумя способами:
1. **Как шаблонный параметр**:
    std::set<int, std::greater<int>> coll;

2. **Как параметр конструктора**:
    set c(beg, end, op) — создание множества с критерием сортировки op, инициализированное элементами из интервала [beg, end)

Немодифицирующие операции над множествами и мультимножествами

c.key_comp() - возвращает критерий сравнения
c.empty() - возвращает результат проверки того, что контейнер пуст
c.max_size() - возвращает максимально возможное число элементов

Функции поиска для множеств и мультимножеств

c.count(val) — возвращает количество элементов, имеющих значение val

c.find(val) — возвращает позицию первого элемента, имеющего значение val (или позицию end(), если элемент не найден)

c.lower_bound(val) — возвращает первую позицию, в которую можно вставить элемент со значением val (первый элемент, удовлетворяющий условию >=val)

c.upper_bound(val) — возвращает последний элемент, в который можно вставить элемент со значением val (первый элемент, удовлетворяющий условию > val)

c.equal_range(val) — возвращает интервал со всеми элементами, значение которых равно val (то есть первую и последнюю позицию, в которые можно вставить элемент со значением val)

Вставка и удаление элементов

c.insert(val) — вставляет копию элемента val, возвращает позицию нового элемента

c.insert(beg, end) — вставляет копии всех элементов интервала [beg, end), не возвращаяя ничего.

c.insert(initlist) — вставляет копию всех элементов списка инициализации initlist, не возвращает ничего.

c.erase(val) — удаляет все элементы, равные val, возвращает количество удаленных элементов

c.erase(pos) — удаляет элемент, занимающий позицию pos, и возвращает следующую позицию

c.erase(beg, end) — удаляет все элементы интервала [beg, end), ивозвращает следующую позицию

c.clear() - удаляет все элементы (опустошает контейнер)

set
примеры

```cpp
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    // множество без повторяющихся элементов, в котором элементы сортируются по убыванию
    set<int,greater<int>> coll1;
    // вставка элементов
    coll1.insert({4,3,5,1,6,2});
    coll1.insert(5);
    // вывод элементов на экран
    for (int elem : coll1) {
        cout << elem << ' ';
    }
    cout << endl;
    // вставляем 4, если status.first == True, т.е. успешно вставили элемент
    // то вычисляем расстояние от начала множества до позиции, где был вставлен элемент
    auto status = coll1.insert(4);
    if (status.second) {
        cout << "4 inserted as element "
             << distance(coll1.begin(),status.first) + 1 << endl;
    } // если status.first == False, значит, элемент уже есть в множестве
    else {
        cout << "4 already exists" << endl;
    }
```

set
примеры

```
    // создаем новое множество coll2 с элементами множества coll1 в обратном порядке
    set<int> coll2(coll1.cbegin(),coll1.cend());
    // выводим на экран элементы с использованием потоковых итераторов
    copy(coll2.cbegin(), coll2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;
    // удаление элементов, равных 3
    coll2.erase(coll2.begin(), coll2.find(3));
    // удаление элементов, равных 5
    int num;
    num = coll2.erase (5); // num - сколько элементов удалили
    cout << num << " element(s) removed" << endl;
    copy (coll2.cbegin(), coll2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

set
примеры

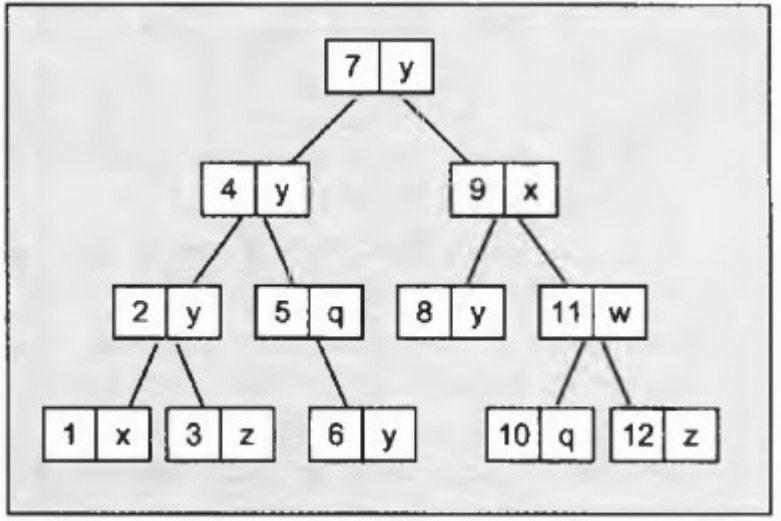
```cpp
#include <iostream>
#include <set>
using namespace std;

int main ()
{
    set<int> c;

    c.insert(1);
    c.insert(2);
    c.insert(4);
    c.insert(5);
    c.insert(6);

    cout << "lower_bound(3): " << *c.lower_bound(3) << endl; // выведет 4
    cout << "upper_bound(3): " << *c.upper_bound(3) << endl; // выведет 4
    cout << "equal_range(3): " << *c.equal_range(3).first << " "
                               << *c.equal_range(3).second << endl; // выведет 4 4
    cout << endl;
    cout << "lower_bound(5): " << *c.lower_bound(5) << endl; // выведет 5
    cout << "upper_bound(5): " << *c.upper_bound(5) << endl; // выведет 6
    cout << "equal_range(5): " << *c.equal_range(5).first << " "
                               << *c.equal_range(5).second << endl; // выведет 5 6
}
```

Отображения (map)

Отображения — это контейнеры, элементами которых являются пары ключ-значение. Эти контейнеры автоматически упорядочивают свои элементы в соответствии со значением ключа. Существуют также мультиотображения, в которых значения ключей могут повторяться.

Отображения обычно реализуются в виде сбалансированных бинарных деревьев.

Таким образом, отображения и мультиотображения имеют все возможности и операции, которые имеют множества и мультимножества.



Внутренняя структура отображений и мультиотображений

Отображения. Критерий сортировки

Критерий сортировки можно задать двумя способами:
1. **Как шаблонный параметр**:
    std::set<float, std::string, std::greater<float>> coll;

2. **Как параметр конструктора**:
    map c(beg, end, op) — создание отображения с критерием сортировки op, инициализированное элементами из интервала [beg, end)

Отображения. Примеры

```cpp
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    typedef map<string,float> StringFloatMap;
    // отображение, ключи - типа string, значения - типа float
    StringFloatMap stocks;
    // вставка элементов
    stocks["BASF"] = 369.50;
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
    stocks["Siemens"] = 842.20;
    // вывод элементов на экран
    StringFloatMap::iterator pos;
    cout << left;  // задаем выравнивание
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << setw(12) << pos->first
             << "price: " << pos->second << endl;
    }
    cout << endl;
```

Отображения. Примеры

```
    // удвоение в цикле всех значений отображения
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        pos->second *= 2;
    }
    cout << endl;
    /* переименование ключа "VW" в "Volkswagen"
       для этого добавляем элемент с ключом "Volkswagen" и значением, равным
       значению по ключу "VW", а старый элемент удаляем */
    stocks["Volkswagen"] = stocks["VW"];
    stocks.erase("VW");
}
```

Неупорядоченные контейнеры

Неупорядоченные контейнеры содержат все вставленные в произвольном порядке. Иначе говоря, неупорядоченный контейнер можно рассматривать как мешок: в него можно класть элементы, но когда вы открываете мешок для того, чтобы достать элементы, вы делаете это в случайном порядке.

В противоположность множествам и отображениям в неупорядоченных контейнерах нет критерия сортировки.

В противоположность последовательным контейнерам в неупорядоченных контейнерах нелбзя разместить элемент в определенной позиции.

Неупорядоченные контейнеры реализованы в виде **хэш-таблиц**.

Неупорядоченные контейнеры

Хэш-таблицы

Чтобы разобраться, что такое хеш-таблицы, представьте, что вас попросили создать библиотеку и заполнить ее книгами. Но вы не хотите заполнять шкафы в произвольном порядке.

Первое, что приходит в голову — разместить все книги в алфавитном порядке и записать все в некий справочник. В этом случае не придется искать нужную книгу по всей библиотеке, а только по справочнику.

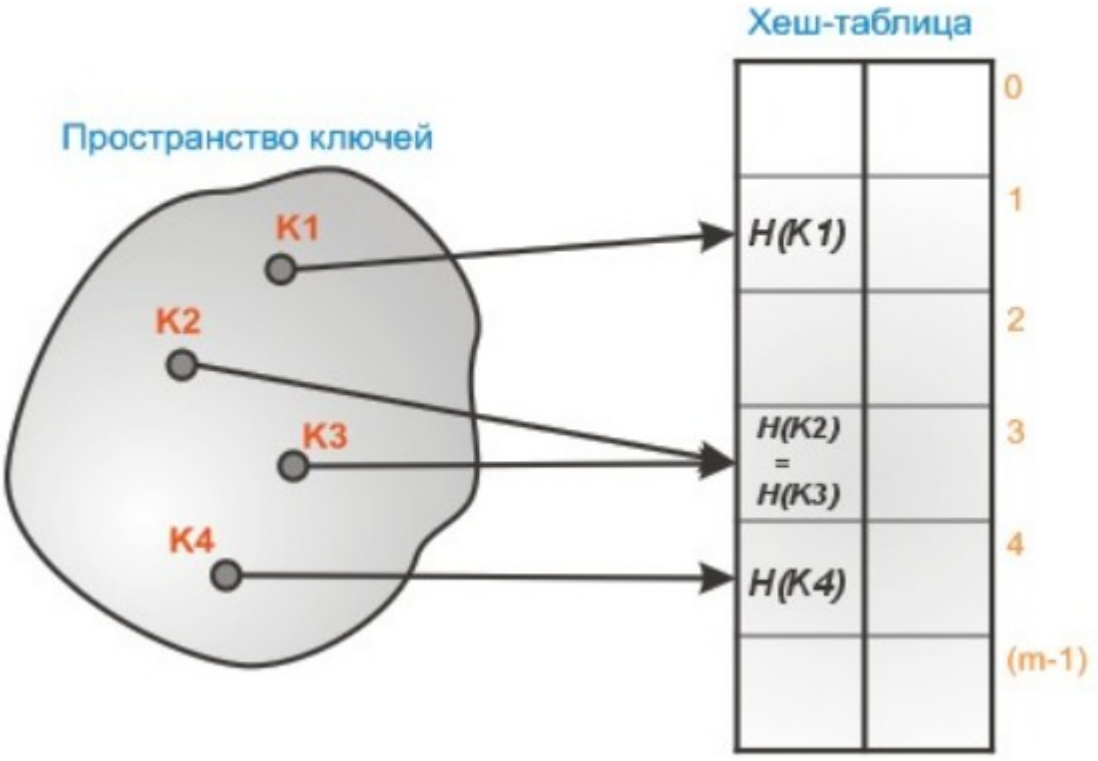
А можно сделать еще удобнее. Если изначально отталкиваться от названия книги или имени автора, то лучше использовать некий алгоритм хеширования, который обрабатывает входящее значение и выдает номер шкафа и полки для нужной книги.

Зная этот алгоритм хэширования, вы быстро найдете нужную книгу по ее названию.

Хеш-функция должна иметь следующие свойства:

- Всегда возвращать один и тот же адрес для одного и того же ключа;
- Не обязательно возвращает разные адреса для разных ключей;
- Использует все адресное пространство с одинаковой вероятностью;
- Быстро вычислять адрес.

Хэш-таблицы

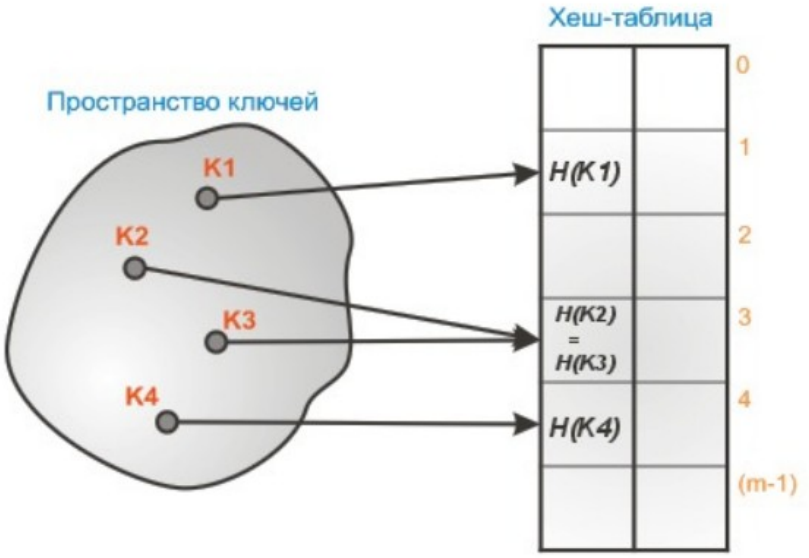

Хеш-таблица
Пространство ключей
K1
K2
K3
K4
H(K1)
H(K2)
=
H(K3)
H(K4)
0
1
2
3
4
(m-1)

Хэш-таблицы

Так же этот рисунок иллюстрирует одну из основных проблем.

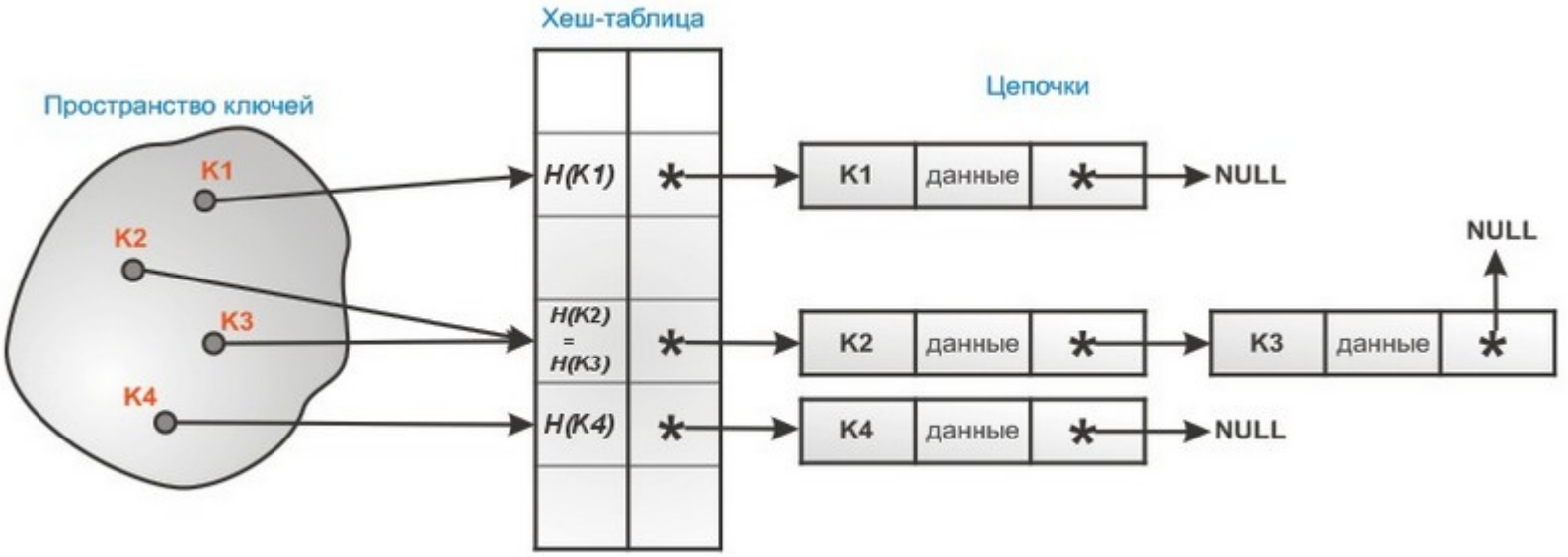
При достаточно маленьком значении m (размера хеш-таблицы) по отношению к n (количеству ключей) или при плохой хеш-функции, может случиться так, что два ключа будут хешированны **в одну и ту же ячейку** массива H.

Такая ситуация называется **коллизией.** Хорошие хеш-функции стремятся минимизировать вероятность коллизий, однако, учитывая то, что пространство всех возможных ключей может быть больше размера нашей хеш-таблицы H, всё же избежать их вряд ли удастся. На этот случай имеются несколько технологий для разрешения коллизий.



Хэширование с цепочками (используется в STL)

Мы объединяем элементы, хешированные в одну и ту же ячейку, в **связный список**. Если при добавлении в хеш-таблицу в заданную ячейку мы встречаем ссылку на элемент связного списка, то случается коллизия. Так, мы просто вставляем наш элемент как узел в список. При поиске мы проходим по цепочкам, сравнивая ключи между собой на эквивалентность, пока не доберёмся до нужного. При удалении ситуация такая же.



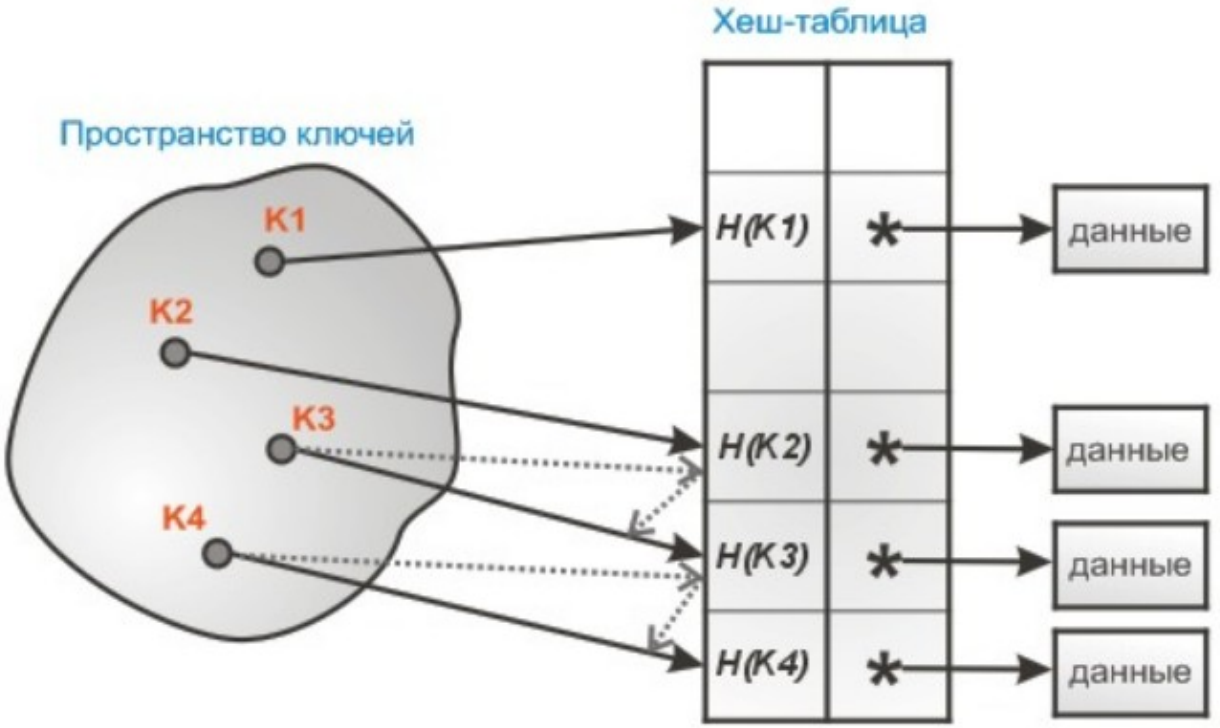
Хэширование с цепочками

Процедура вставки выполняется даже в наихудшем случае за O(1), учитывая то, что мы предполагаем отсутствие вставляемого элемента в таблице. Время поиска зависит от длины списка, и в худшем случае равно O(n). Эта ситуация, когда все элементы хешируются в единственную ячейку.

Если функция распределяем n ключей по m ячейкам таблицы равномерно, то в каждом списке будет содержаться порядка n/m ключей. Это число называется коэффициентом заполнения хеш-таблицы.

Математический анализ хеширования с цепочками показывает, что в среднем случае все операции в такой хеш-таблице в среднем выполняются за время O(1).

Хеширование с открытой адресацией



Все элементы хранятся непосредственно в хеш-таблице, без использования связанных списков.
При возникновении коллизии следующие за текущей ячейки проверяются одна за другой, пока не найдётся пустая ячейка, куда и помещается наш элемент. Так, при достижении последнего индекса таблицы, мы перескакиваем в начало, рассматривая её как «цикличный» массив.

Конструкторы неупорядоченных контейнеров

Unord c — конструктор по умолчанию, создает пустой контейнер

Unord c(bnum) — создает пустой неупорядоченный контейнер, который использует по крайней мере bnum сегментов (сегмент — это связный список, ключи, для которых значение хэш — функции получается одно и то же, добавляются в один и тот же сегмент)

Unord c(bnum, hf) — создает пустой неупорядоченный контейнер, который использует по крайней мере bnum сегментов и хэш-функцию hf

Unord c(bnum, hf, cmp) - создает пустой неупорядоченный контейнер, который использует по крайней мере bnum сегментов и хэш-функцию hf и критерий сравнения cmp для идентификации одинаковых значений

Конструкторы неупорядоченных контейнеров

Unord c(beg, end) — контейнер инициализируется элементами интервала [beg, end)

Unord c(beg, end, bnum)

Unord c(beg, end, bnum, hf)

Unord c(beg, end, bnum, hf, cmp)

Unord c(initlist)

Возможные типы Unord

`unordered_set<`*Elem*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_set<`*Elem*`,` *Hash*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_set<`*Elem*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_multiset<`*Elem*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multiset<`*Elem*`,` *Hash*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multiset<`*Elem*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*

Возможные типы Unord

`unordered_multiset<`*Elem*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_map<`*Key*`,` *T*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_map<`*Key*`,` *T*`,` *Hash*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_map<`*Key*`,` *T*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное отображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_multimap<`*Key*`,` *T*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>`
`unordered_multimap<`*Key*`,` *T*`,` *Hash*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multimap<`*Key*`,` *T*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*

Структурные операции над неупорядоченными контейнерами

`c.hash_function()`	Возвращает хеш-функцию
`c.key_eq()`	Возвращает предикат эквивалентности
`c.bucket_count()`	Возвращает текущее количество сегментов
`c.max_bucket_count()`	Возвращает максимально возможное количество сегментов
`c.load_factor()`	Возвращает текущий коэффициент заполнения
`c.max_load_factor()`	Возвращает текущий коэффициент максимального заполнения
`c.max_load_factor(`*val*`)`	Задает коэффициент максимального заполнения равным *val*
`c.rehash(`*bnum*`)`	Повторно хеширует контейнер, так, чтобы его размер сегмента был не меньше *bnum*
`c.reserve(`*num*`)`	Повторно хеширует контейнер, чтобы он мог содержать не меньше *num* элементов (по стандарту C++11)

unordered set
примеры

```cpp
#include <unordered_set>
#include <numeric> // для функции accumulate
#include <iostream>
#include <string>

template <typename T>
inline void PRINT_ELEMENTS (const T& coll,
                            const std::string& optstr="")
{
    std::cout << optstr;
    for (const auto&  elem : coll) {
        std::cout << elem << ' ';
    }
    std::cout << std::endl;
}

using namespace std;

int main()
{
    // создание и инициализация неупорядоченного множества
    unordered_set<int> coll = { 1,2,3,5,7,11,13,17,19,77 };
    PRINT_ELEMENTS(coll);
    /* вставка элементов. Может вызвать рехэширование и элементы
       могут переместиться в другом порядке */
    coll.insert({-7,17,33,-11,17,19,1,13});
    PRINT_ELEMENTS(coll);
    coll.erase(33); // удаление элемента
```

unordered set
примеры

```
    // вставка суммы всех элементов
    coll.insert(accumulate(coll.begin(),coll.end(),0));
    PRINT_ELEMENTS(coll);
    // проверка, есть ли элемент 19 в  контейнере
    if (coll.find(19) != coll.end()) {
        cout << "19 is available" << endl;
    }
    // удаление всех отрицательных значений
    unordered_set<int>::iterator pos;
    for (pos=coll.begin(); pos!= coll.end(); ) {
        if (*pos < 0) {
            pos = coll.erase(pos);
        }
        else {
            ++pos;
        }
    }
    PRINT_ELEMENTS(coll);
}
```

Сравнение различных контейнеров

Контейнер	Вставка	Доступ	Удаление	Поиск
vector	В конец: O(1) В остальные места: O(N)	O(1)	В конец: O(1) В остальные места: O(N)	Отсортироованный: O(log(N)) Не отсортированный: O(N)
deque	В начало/конец: O(1) В остальные места: O(N)	O(1)	В начало/конец: O(1) В остальные места: O(N)	Отсортироованный: O(log(N)) Не отсортированный: O(N)
list	В начало/конец: O(1) В остальные места: O(N)	начало/конец: O(1) Остальные места: O(N)	В начало/конец: O(1) В остальные места: O(N)	O(N)
set/map	O(log(N))	–	O(log(N))	O(log(N))
unordered_set/ unordered_map	O(1) или O(N)	O(1) или O(N)	O(1) или O(N)	O(1) или O(N)

Boost.MultiIndex

boost::multi_index используется, если в контейнере необходим доступ по более чем одному ключу или по комбинации ключей.

Создадим структуру и далее boost::multi_index_container для экземпляров этой структуры

```
using namespace boost::multi_index;

// структура, содержащая два элемента: название животного и количествоо его ног
struct animal
{
  std::string name;
  int legs;
};
```

Boost.MultiIndex

```
// структура, содержащая два элемента: название животного и количествоо его ног
struct animal
{
  std::string name;
  int legs;
};

// будем использовать multi_index_container
typedef multi_index_container<
  animal,           // структура, поля которой будут использоваться как ключи для поиска
  indexed_by<
    hashed_non_unique< // поле animal::name будет хешироваться
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique< // поле animal::legs также будет ключом для поиска и оно хешируется
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;
```

Boost.MultiIndex

```cpp
int main()
{
  // создаем multi_index_container
  animal_multi animals;

  // вставка элементов в контейнер
  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  // подсчет количества элементов с данным значением одного из ключей
  std::cout << animals.count("cat") << '\n';

  // создаем объект legs_index для поиска в контейнере по 2-му полю структуры, т.е. legs
  const animal_multi::nth_index<1>::type &legs_index = animals.get<1>();
  // подсчет количества элементов, где legs = 8
  std::cout << legs_index.count(8) << '\n';
}
```

Boost.MultiIndex

```cpp
int main()
{
  // создаем multi_index_container
  animal_multi animals;

  // вставка элементов в контейнер
  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  /* создаем объект legs_index для поиска в контейнере
     по 2-му полю структуры, т.е. legs */
  auto &legs_index = animals.get<1>();
  // поиск элемента, где legs = 4
  // это элемент ("cat", 4)
  auto it = legs_index.find(4);
  // вывод на экран соотв. полей найденной структуры
  cout << it->name << ' ' << it->legs << endl;
  // изменение поля name в найденной структуре
  legs_index.modify(it, [](animal &a){ a.name = "dog"; });

  cout << animals.count("dog") << '\n';
}
```

Boost.MultiIndex

```
typedef multi_index_container<
  animal,
  indexed_by<
    sequenced<>, /* последовательность, в которой элементы упорядочены
                    в порядке добавления */
    ordered_non_unique<
      member<
        animal, int, &animal::legs // поиск по полю legs структуры animal
      >
    >,
    random_access<> // можно обращаться по индексу к элементу контейнера
  >
> animal_multi;
```

Boost.MultiIndex

```cpp
int main()
{
  animal_multi animals;

  // добавление элементов
  animals.push_back({"cat", 4});
  animals.push_back({"shark", 0});
  animals.push_back({"spider", 8});

  // объект для поиска элементов в  контейнере по полю legs
  auto &legs_index = animals.get<1>();
  for(auto it = legs_index.begin(); it != legs_index.end(); it++)
    std::cout << it->name << ' ' << it->legs << std::endl;
  std::cout << std::endl;

  // объект, чтобы обращаться к элементам контейнера по индексу
  const auto &rand_index = animals.get<2>();
  std::cout << rand_index[0].name << '\n';
}
```

Boost.Bimap

bimap отличается от map тем, что можно использовать в качестве ключа любое из двух значений.

```cpp
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  // bimap, где первое значение типа string, второе - типа int
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

  // вставка элементов в bimap
  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  // подсчет количества элементов, где первое значение "cat"
  std::cout << animals.left.count("cat") << '\n';
  // подсчет количества элементов, где второе значение 8
  std::cout << animals.right.count(8) << '\n';

  // вывод элементов bimap на экран
  for (auto it = animals.begin(); it != animals.end(); ++it)
    std::cout << it->left << " has " << it->right << " legs\n";
}
```