Ассоциативные контейнеры

Ассоциативные контейнеры

Ассоциативный контейнер - контейнер, обеспечивающий быстрый поиск элементов.

В C++ ассоциативные контейнеры представлены шаблонными классами:

- `std::set` / `std::multiset`
- `std::map` / `std::multimap`
- `std::unordered_set` / `std::unordered_multiset`
- `std::unordered_map` / `std::unordered_multimap`

`std::set`/`std::multiset`

`std::set` / `std::multiset`

`std::set<KeyT, Compare = std::less<KeyT>>` - множество уникальных ключей.
`std::multiset<KeyT, Compare = std::less<KeyT>>` - множество ключей.

- `Compare` - функтор, т.е. тип, у которого определена операция `()`, принимающая два объекта `KeyT` и возвращающая `true`, если первый операнд меньше (строго!) второго.
- Элементы упорядочены по ключу согласно сравнению `Compare`.
- Обеспечивают логарифмический поиск, вставку и удаление элементов.
- Как правило, реализованы с помощью бинарного дерева поиска.

`std::set` / `std::multiset`: конструкторы

```
std::set<int> s;  // пустое множество
```

```
std::array<int, 5> arr{1, 2, 3, 1, 3};
std::set<int> s(arr.begin(), arr.end());  // {1, 2, 3}
```

```
std::array<int, 5> arr{1, 2, 3, 1, 3};
std::multiset<int> s(arr.begin(), arr.end());  // {1, 1, 2, 3, 3}
```

```
std::set<int, std::greater<>> s{5, 2, 7, 1, 5};  // {7, 5, 2, 1}
```

`std::set`/`std::multiset`: поиск

- `count(key)` - число элементов `key` во множестве; [O(log n + #key)]
- `find(key)` - итератор на элемент `key` (`end()`, если ключа нет); [O(log n)]

```
if (s.find(key) != s.end()) /* vs */ if (s.count(5))
```

Менять значение ключа по итератору нельзя!

```
*s.find(5) = 10;  // CE
```

- `lower_bound(key)` - итератор на первый элемент >= `key`; [O(log n)]
- `upper_bound(key)` - итератор на первый элемент > `key`; [O(log n)]
- `equal_range(key)` - пара (`std::pair`) `lower_bound(key)`, `upper_bound(key)` [O(log n)]

Не путать с алгоритмами из `<algorithm>`!

`std::set`/`std::multiset`: вставка

- `insert(key)` - вставка элемента `key`, возвращает пару успех/неуспех и итератор на элемент; [O(log n)]

```
auto [success, it] = s.insert(5);
success;  // inserted/not inserted (true/false)
it;       // iterator
```

- `emplace(Args&&...)` - вставляет элемент с параметрами конструктора `Args&&...`, возвращает пару успех/неуспех и итератор на элемент; [O(log n)]
- `insert(begin, end)` - вставляет последовательность элементов; $[O(klog(n+k))]$

```
s.insert(arr.begin(), arr.end());
```

`std::set`/`std::multiset`: вставка с подсказкой

При вставке можно передать "подсказку" - итератор на ближайший элемент *больший* вставляемого.

То есть, если дано множество $\{1, 2, 4\}$, то при вставке 3 можно дополнительно передать итератор на элемент 4. При вставке 0 подсказка - итератор на $begin$. При вставке 5 подсказка - итератор на end.

В случае верной подсказки вставка будет работать амортизировано за $O(1)$. Иначе $O(logn)$.

- `insert(hint, key)` - вставка `key`, `hint` - итератор, подсказка, куда вставить. Возвращает итератор на элемент; [O(1) - O(log n)]
- `emplace_hint(hint, Args&&...)` - аналогично; [O(1) - O(log n)]

`std::set` / `std::multiset`: удаление

- `erase(iterator)` - удаляет элемент по итератору, возвращает итератор на следующий по величине элемент; [O(log n)]
- `erase(begin, end)` - удаляет последовательность элементов; [O(log n + distance(begin, end))]
- `erase(key)` - удаляет все элементы с ключом `key`, возвращает число удаленных элементов; [O(log n + #key)]

`std::set` / `std::multiset`: прочее

- `empty()`, `size()`, `swap()`; [O(1)]
- `clear()`; [O(n)]
- При обходе от `begin()` до `end()` элементы перечисляются в порядке возрастания.

```
for (const auto& x : s) {
  std::cout << x << ' ';  // выводятся в порядке возрастания
}
```

`std::set`/`std::multiset`: изменение ключа

Изменить ключ в ассоциативном контейнере нереально, но я мечту свою лелея...

```
s.erase(old_key);   // удаляем старый ключ и узел
s.insert(new_key);  // создаем новый узел и ключ
```

Начиная с C++17 можно извлечь узел из контейнера, изменить в нем в ключ и вставить обратно.

```
auto node = s.extract(old_key);  // извлекаем узел
node.value() = new_key;          // меняем ключ в узле
s.insert(std::move(node));       // вставляем обратно
```

`std::map`/`std::multimap`

`std::map`/`std::multimap`

`std::map<KeyT, ValueT, Compare = std::less<KeyT>>` - отображение из множества уникальных ключей в значения.

`std::multimap<KeyT, ValueT, Compare = std::less<KeyT>>` - множество пар "ключ-значение".

- `Compare` - функтор, т.е. тип, у которого определена операция `()`, принимающая два объекта `KeyT` и возвращающая `true`, если первый операнд меньше (строго!) второго.
- Пары "ключ-значение" упорядочены по ключу согласно сравнению `Compare`.
- Обеспечивают логарифмический поиск, вставку и удаления ключей.
- Как правило, реализованы с помощью бинарного дерева поиска.

`std::map`/`std::multimap`: методы

Методы аналогичны `std::set` и `std::multiset`, только в качестве параметров передаются пары "ключ-значение":

```
std::array<std::pair<int, std::string>, 3> arr[3]{
  {1, "one"},
  {2, "two"},
  {3, "three"}
};

std::map<int, std::string> m(arr.begin(), arr.end());
m.insert(std::make_pair(4, std::string("four")));
m.emplace(5, "five");

auto it = m.find(3);
it->first;   // key == 3
it->second;  // value == "three"
```

`std::map`/`std::multimap`: методы

- `operator[](key)` (только std::map) - возвращает ссылку на значение, связвнное с ключом `key`

Важно!

Если ключа нет, то он вставляется и создается значение по умолчанию (если `ValueT` - базовый тип, то инициализируется нулем!).

Поэтому нельзя использовать для константных мап, а также для мап, в которых `ValueT` не имеет конструктора по умолчанию.

```
std::map<int, int> m;
const std::map<int, int> cm;
//...
m[0];   // Ok
cm[0];  // CE
```

`std::map`/`std::multimap`: методы

Плохо:

```
std::map<std::string, int> counter;
std::string str = GetString();
if (counter.find(str) != counter.end()) {
  ++counter[str];
} else {
  counter.emplace(str, 1);
}
```

Норм:

```
std::map<std::string, int> counter;
std::string str = GetString();
++counter[str];
```

`std::map`/`std::multimap`: методы

- `at(key)` (кроме `std::multimap`) - возвращает ссылку на значение, связанное с ключом `key`

Важно! Если ключа нет, то бросается исключение `std::out_of_range`.

Может быть использован в константных мапах:

```
const std::map<int, int> cm{{1, 1}, {2, 2}, {3, 3}};

std::cout << cm[1];     // CE

std::cout << cm.at(1);  // Ok
```

`std::map`/`std::multimap`: изменение ключа

Аналогично `std::set`

```
auto node = m.extract(old_key);  // извлекаем узел
node.key() = new_key;            // меняем ключ в узле
m.insert(std::move(node));       // вставляем обратно
```

`std::unordered_set`/`std::unordered_multiset`

`std::unordered_map`/`std::unordered_multimap`

`std::unordered_...`

- `std::unordered_set<KeyT, Hash = std::hash<KeyT>, Equal = std::equal_to<KeyT>>`
- `std::unordered_map<KeyT, ValueT, Hash = std::hash<KeyT>, Equal = std::equal_to<KeyT>>`

Аналогичны `std::set`, `std::map`, `std::multiset`, `std::multimap`, но в среднем более эффективный поиск, вставка и удаление [O(1)], так как основаны на хеш таблицах.

Однако не хранят порядок элементов. Поэтому отсутствуют методы `lower_bound`, `upper_bound`, `equal_range`. Обход осуществляется в произвольном порядке.

`std::unordered_...`: дополнительные методы

- `float load_factor() const` - возвращает отношение числа элементов к числу корзин.
- `float max_load_factor() const` - возвращает `load_factor` при котором происходит перехеширование (увеличение числа корзин).
- `void max_load_factor(float)` - устанавливает `load_factor`, при котором должно происходить увеличение числа корзин.
- `rehash(count)` - изменить число корзин на `count`.
- `reserve(count)` - устанавливает число корзин достаточное для хранения `count` элементов.
- `bucket(key)` - номер корзины для элемента `key`.
- `bucket_size(id)` - размер корзины `id`.
- `bucket_count()` - число корзин.