Яндекс Такси

# Ещё чуть быстрее

## делаем свой C++ контейнер

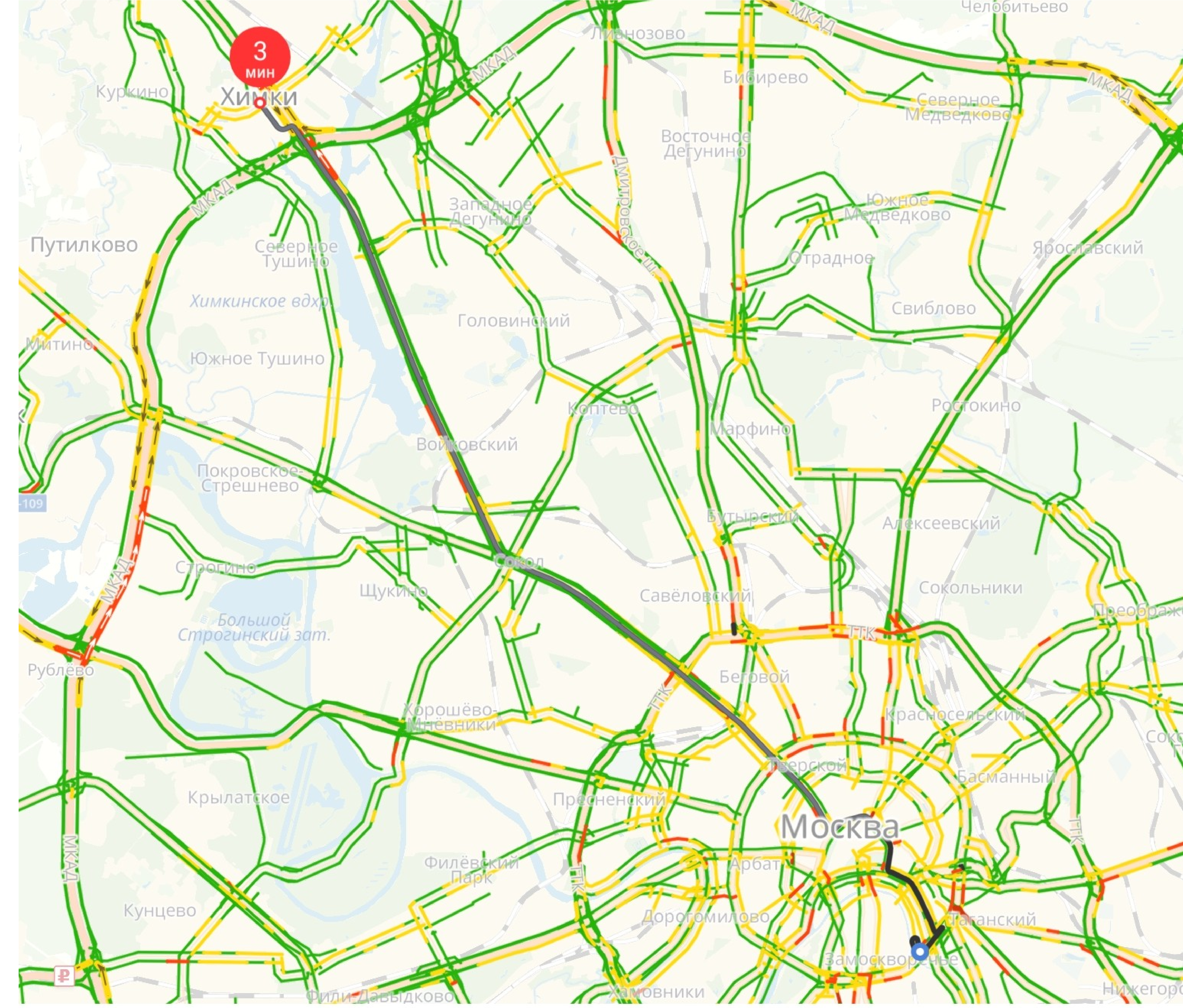
Полухин Антон

Antony Polukhin

Яндекс Такси

# Содержание

- Stack Overflow
- Немного здравого смысла
- C++17
- Boost
- More Boost!
- ???




C++ :-(

Подъезд

C++ :-)

+

ЭКОНОМ 4₽

КОМФОРТ 8₽

**КОМФОРТ+ 9₽**

БИЗНЕС 34₽

МИНИВЭН 15₽

ДЕТСКИЙ 2₽

Комментарий, пожелания

Способ оплаты
Команда Яндекс.Такси

Ещё чуть быстрее

# LRU

# LRU

# LRU

Lru(MaxSize)

# LRU

Lru(MaxSize)

- O(1):

# LRU

Lru(MaxSize)

- O(1):
  - bool Has(const T& key)
    - «Обновляет» использование записи

# LRU

Lru(MaxSize)

- O(1):
  - bool Has(const T& key)
    - «Обновляет» использование записи
  - bool Put(const T& key)
    - Заменяет самую старую запись новой
    - или «обновляет» использование имеющейся записи

# LRU #1

# LRU #1

```
  bool Has(const T& key) {
    auto it = map_.find(key);
    if (it == map_.end()) return false;

    auto list_it = it->second;
    list_.erase(list_it);
    it->second = list_.insert(list_.end(), key);
    return true;
  }

 private:
  std::list<T> list_;
  std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
  std::size_t max_size_;
```

# LRU #1

```
  bool Has(const T& key) {
    auto it = map_.find(key);
    if (it == map_.end()) return false;

    auto list_it = it->second;
    list_.erase(list_it);
    it->second = list_.insert(list_.end(), key);
    return true;
  }

private:
  std::list<T> list_;
  std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
  std::size_t max_size_;
```

# LRU #1

```
  bool Has(const T& key) {
    auto it = map_.find(key);
    if (it == map_.end()) return false;

    auto list_it = it->second;
    list_.erase(list_it);
    it->second = list_.insert(list_.end(), key);
    return true;
  }

 private:
  std::list<T> list_;
  std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
  std::size_t max_size_;
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    list_.pop_front();
    map_.erase(last);
    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    list_.pop_front();
    map_.erase(last);
    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    list_.pop_front();
    map_.erase(last);
    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    list_.pop_front();
    map_.erase(last);
    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    list_.pop_front();
    map_.erase(last);
    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>              95907 ns      95905 ns        7306

Has<v1::LruSet<unsigned>>              25036 ns      25035 ns       27432

PutOverflow<v1::LruSet<unsigned>>      70947 ns      70946 ns        9376
```

# Будем улучшать!

# Будем улучшать!

Disclaimer: для краткости в коде отсутствуют некоторые std::move и noexcept

# LRU #2

# LRU #1

```
  bool Has(const T& key) {
    auto it = map_.find(key);
    if (it == map_.end()) return false;

    auto list_it = it->second;
    list_.erase(list_it);
    it->second = list_.insert(list_.end(), key);
    return true;
  }

 private:
  std::list<T> list_;
  std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
  std::size_t max_size_;
```

# LRU #1

```
  bool Has(const T& key) {
    auto it = map_.find(key);
    if (it == map_.end()) return false;

    auto list_it = it->second;
    list_.erase(list_it);
    it->second = list_.insert(list_.end(), key);
    return true;
  }

 private:
  std::list<T> list_;
  std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
  std::size_t max_size_;
```

# LRU #2

```
    bool Has(const T& key) {
      auto it = map_.find(key);
      if (it == map_.end()) return false;

      auto list_it = it->second;

      list_.splice(list_.end(), list_, list_it);
      return true;
    }

  private:
    std::list<T> list_;
    std::unordered_map<T, typename std::list<T>::iterator, Hash, Equal> map_;
    std::size_t max_size_;
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.pop_front();

    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #1

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.erase(it->second);
    it->second = list_.insert(list_.end(), key);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.pop_front();

    map_[key] = list_.insert(list_.end(), key);
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #2

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    auto list_it = it->second;
    list_.splice(list_.end(), list_, list_it);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    map_[key] = --list_.end();
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #2

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    auto list_it = it->second;
    list_.splice(list_.end(), list_, list_it);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    map_[key] = --list_.end();
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>                95907 ns      95905 ns         7306
Put<v2::LruSet<unsigned>>                96491 ns      96489 ns         7085




Has<v1::LruSet<unsigned>>                25036 ns      25035 ns        27432
Has<v2::LruSet<unsigned>>                11694 ns      11694 ns        63427




PutOverflow<v1::LruSet<unsigned>>        70947 ns      70946 ns         9376
PutOverflow<v2::LruSet<unsigned>>        65341 ns      65338 ns        11186
```

# LRU #3

Ещё чуть быстрее

# LRU #2

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    map_[key] = --list_.end();


  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #2

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    map_.erase(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    map_[key] = --list_.end();


  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>              95907 ns      95905 ns        7306
Put<v2::LruSet<unsigned>>              96491 ns      96489 ns        7085
Put<v3::LruSet<unsigned>>              95387 ns      95384 ns        6908



Has<v1::LruSet<unsigned>>              25036 ns      25035 ns       27432
Has<v2::LruSet<unsigned>>              11694 ns      11694 ns       63427
Has<v3::LruSet<unsigned>>              11799 ns      11799 ns       56695



PutOverflow<v1::LruSet<unsigned>>      70947 ns      70946 ns        9376
PutOverflow<v2::LruSet<unsigned>>      65341 ns      65338 ns       11186
PutOverflow<v3::LruSet<unsigned>>      43203 ns      43202 ns       16447
```

# LRU #4

# Хм...

# Хм...

На 1 значение у нас динамически аллоцируются 2 ноды

# Хм...

На 1 значение у нас динамически аллоцируются 2 ноды:

- Каждая содержит ключ
- Не кеш дружелюбно
- Указатели\итератор на ноду

# Хм...

На 1 значение у нас динамически аллоцируются 2 ноды:

- Каждая содержит ключ
- Не кеш дружелюбно
- Указатели\итератор на ноду

А если делать одну ноду?

# LRU #4 intrusive list

```
#include <boost/intrusive/link_mode.hpp>
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/list_hook.hpp>

using LinkMode = boost::intrusive::link_mode<
#ifdef NDEBUG
    boost::intrusive::normal_link
#else
    boost::intrusive::safe_link
#endif
    >;

using LruHook = boost::intrusive::list_base_hook<LinkMode>;
class LruNode final : public LruHook {};
```

# LRU #4 intrusive list

```cpp
#include <boost/intrusive/link_mode.hpp>
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/list_hook.hpp>

using LinkMode = boost::intrusive::link_mode<
#ifdef NDEBUG
    boost::intrusive::normal_link
#else
    boost::intrusive::safe_link
#endif
    >;

using LruHook = boost::intrusive::list_base_hook<LinkMode>;
class LruNode final : public LruHook {};
```

# LRU #4 intrusive list

```
#include <boost/intrusive/link_mode.hpp>
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/list_hook.hpp>

using LinkMode = boost::intrusive::link_mode<
#ifdef NDEBUG
    boost::intrusive::normal_link
#else
    boost::intrusive::safe_link
#endif
    >;

using LruHook = boost::intrusive::list_base_hook<LinkMode>;
class LruNode final : public LruHook {};
```

# LRU #4 intrusive list

```
  private:
   using List = boost::intrusive::list<
       LruNode,
       boost::intrusive::constant_time_size<false>
       >;

   using Map = std::unordered_map<T, LruNode, Hash, Equal>;

   Map map_;
   List list_;
   std::size_t max_size_;

   const T& GetLeastRecentKey() {
     using Pair = typename Map::value_type;
     constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
     return *reinterpret_cast<const T*>(                                  // Фу, гадость!
         reinterpret_cast<const char*>(&list_.front()) - offset);         // Фу, гадость!
   }
```

# LRU #4 intrusive list

```
private:
 using List = boost::intrusive::list<
     LruNode,
     boost::intrusive::constant_time_size<false>
     >;

 using Map = std::unordered_map<T, LruNode, Hash, Equal>;

 Map map_;
 List list_;
 std::size_t max_size_;

 const T& GetLeastRecentKey() {
   using Pair = typename Map::value_type;
   constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
   return *reinterpret_cast<const T*>(                                   // Фу, гадость!
       reinterpret_cast<const char*>(&list_.front()) - offset);          // Фу, гадость!
 }
```

# LRU #4 intrusive list

```
  private:
   using List = boost::intrusive::list<
       LruNode,
       boost::intrusive::constant_time_size<false>
       >;

   using Map = std::unordered_map<T, LruNode, Hash, Equal>;

   Map map_;
   List list_;
   std::size_t max_size_;

   const T& GetLeastRecentKey() {
     using Pair = typename Map::value_type;
     constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
     return *reinterpret_cast<const T*>(                                   // Фу, гадость!
         reinterpret_cast<const char*>(&list_.front()) - offset);          // Фу, гадость!
   }
```

# LRU #4 intrusive list

```cpp
private:
  using List = boost::intrusive::list<
      LruNode,
      boost::intrusive::constant_time_size<false>
      >;

  using Map = std::unordered_map<T, LruNode, Hash, Equal>;

  Map map_;
  List list_;
  std::size_t max_size_;

  const T& GetLeastRecentKey() {
    using Pair = typename Map::value_type;
    constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
    return *reinterpret_cast<const T*>(                                   // Фу, гадость!
        reinterpret_cast<const char*>(&list_.front()) - offset);         // Фу, гадость!
  }
```

# LRU #3

```
bool Has(const T& key) {
  auto it = map_.find(key);
  if (it == map_.end()) return false;
  auto list_it = it->second;
  list_.splice(list_.end(), list_, list_it);
  return true;
}
```

# LRU #3

```
bool Has(const T& key) {
  auto it = map_.find(key);
  if (it == map_.end()) return false;
  auto list_it = it->second;
  list_.splice(list_.end(), list_, list_it);
  return true;
}
```

# LRU #4

```
bool Has(const T& key) {
  auto it = map_.find(key);
  if (it == map_.end()) return false;

  list_.splice(list_.end(), list_, list_.iterator_to(it->second));
  return true;
}
```

# LRU #4

```
bool Has(const T& key) {
  auto it = map_.find(key);
  if (it == map_.end()) return false;

  list_.splice(list_.end(), list_, list_.iterator_to(it->second));
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (list_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());


    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;

    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (list_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());


    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;

    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #3

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, it->second);
    return false;
  }
  if (list_.size() == max_size_) {
    T last = list_.front();
    auto node = map_.extract(last);
    list_.front() = key;
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    node.mapped() = --list_.end();
    map_.insert(std::move(node));
  } else {
    map_[key] = list_.insert(list_.end(), key);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (list_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());


    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;

    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>                95907 ns       95905 ns         7306
Put<v2::LruSet<unsigned>>                96491 ns       96489 ns         7085
Put<v3::LruSet<unsigned>>                95387 ns       95384 ns         6908
Put<v4::LruSet<unsigned>>              1079726 ns     1079707 ns          688


Has<v1::LruSet<unsigned>>                25036 ns       25035 ns        27432
Has<v2::LruSet<unsigned>>                11694 ns       11694 ns        63427
Has<v3::LruSet<unsigned>>                11799 ns       11799 ns        56695
Has<v4::LruSet<unsigned>>                 9617 ns        9616 ns        70000


PutOverflow<v1::LruSet<unsigned>>        70947 ns       70946 ns         9376
PutOverflow<v2::LruSet<unsigned>>        65341 ns       65338 ns        11186
PutOverflow<v3::LruSet<unsigned>>        43203 ns       43202 ns        16447
PutOverflow<v4::LruSet<unsigned>>      2417542 ns     2417459 ns        290
```

# LRU #4 intrusive list

```
  private:
   using List = boost::intrusive::list<
       LruNode,
       boost::intrusive::constant_time_size<false>
       >;
   using Map = std::unordered_map<T, LruNode, Hash, Equal>;
   Map map_;
   List list_;
   std::size_t max_size_;

   const T& GetLeastRecentKey() {
     using Pair = typename Map::value_type;
     constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
     return *reinterpret_cast<const T*>(                                   // Фу, гадость!
         reinterpret_cast<const char*>(&list_.front()) - offset);          // Фу, гадость!
   }
```

# LRU #4 intrusive list

```
  private:
   using List = boost::intrusive::list<
       LruNode,
       boost::intrusive::constant_time_size<false> // Beware!
       >;
   using Map = std::unordered_map<T, LruNode, Hash, Equal>;
   Map map_;
   List list_;
   std::size_t max_size_;

   const T& GetLeastRecentKey() {
     using Pair = typename Map::value_type;
     constexpr auto offset = offsetof(Pair, second) - offsetof(Pair, first);// Сомнительно!
     return *reinterpret_cast<const T*>(                                   // Фу, гадость!
         reinterpret_cast<const char*>(&list_.front()) - offset);          // Фу, гадость!
   }
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (list_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (list_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (map_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>                95907 ns     95905 ns      7306
Put<v2::LruSet<unsigned>>                96491 ns     96489 ns      7085
Put<v3::LruSet<unsigned>>                95387 ns     95384 ns      6908
Put<v4::LruSet<unsigned>>                64934 ns     64933 ns     10200


Has<v1::LruSet<unsigned>>                25036 ns     25035 ns     27432
Has<v2::LruSet<unsigned>>                11694 ns     11694 ns     63427
Has<v3::LruSet<unsigned>>                11799 ns     11799 ns     56695
Has<v4::LruSet<unsigned>>                 9617 ns      9616 ns     70000


PutOverflow<v1::LruSet<unsigned>>        70947 ns     70946 ns      9376
PutOverflow<v2::LruSet<unsigned>>        65341 ns     65338 ns     11186
PutOverflow<v3::LruSet<unsigned>>        43203 ns     43202 ns     16447
PutOverflow<v4::LruSet<unsigned>>        41245 ns     41244 ns     17054
```

# Потребление оперативы

# Потребление оперативы

На одно значение мы теперь аллоцируем меньше на:

# Потребление оперативы

На одно значение мы теперь аллоцируем меньше на:

- sizeof(Key) OR sizeof(Key*)

# Потребление оперативы

На одно значение мы теперь аллоцируем меньше на:

- sizeof(Key) OR sizeof(Key*)
- sizeof(std::list::iterator)

# LRU #5

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (map_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #4

```cpp
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (map_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (map_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey()); // iterator_to ????
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #5

```cpp
using LruListHook = boost::intrusive::list_base_hook<LinkMode>;
using LruHashSetHook = boost::intrusive::unordered_set_base_hook<LinkMode>;

template <class Key>
class LruNode final : public LruListHook, public LruHashSetHook {
 public:
  explicit LruNode(Key&& key) : key_(std::move(key)) {}

  const Key& GetKey() const noexcept { return key_; }
  void SetKey(Key key) { key_ = std::move(key); }

 private:
  Key key_;
};
```

# LRU #5

```cpp
using LruListHook = boost::intrusive::list_base_hook<LinkMode>;
using LruHashSetHook = boost::intrusive::unordered_set_base_hook<LinkMode>;

template <class Key>
class LruNode final : public LruListHook, public LruHashSetHook {
 public:
  explicit LruNode(Key&& key) : key_(std::move(key)) {}

  const Key& GetKey() const noexcept { return key_; }
  void SetKey(Key key) { key_ = std::move(key); }

 private:
  Key key_;
};
```

# LRU #5

```cpp
using LruListHook = boost::intrusive::list_base_hook<LinkMode>;
using LruHashSetHook = boost::intrusive::unordered_set_base_hook<LinkMode>;

template <class Key>
class LruNode final : public LruListHook, public LruHashSetHook {
 public:
  explicit LruNode(Key&& key) : key_(std::move(key)) {}

  const Key& GetKey() const noexcept { return key_; }
  void SetKey(Key key) { key_ = std::move(key); }

 private:
  Key key_;
};
```

# LRU #5

```cpp
using LruListHook = boost::intrusive::list_base_hook<LinkMode>;
using LruHashSetHook = boost::intrusive::unordered_set_base_hook<LinkMode>;

template <class Key>
class LruNode final : public LruListHook, public LruHashSetHook {
 public:
  explicit LruNode(Key&& key) : key_(std::move(key)) {}

  const Key& GetKey() const noexcept { return key_; }
  void SetKey(Key key) { key_ = std::move(key); }

 private:
  Key key_;
};
```

# LRU #5

```
struct LruNodeHash : Hash {
  template <class NodeOrKey> auto operator()(const NodeOrKey& x) const {
    return Hash::operator()(v5::GetKey(x));
  }
};
struct LruNodeEqual : Equal {
  template <class NodeOrKey1, class NodeOrKey2>
  auto operator()(const NodeOrKey1& x, const NodeOrKey2& y) const {
    return Equal::operator()(v5::GetKey(x), v5::GetKey(y));
  }
};

using Map = boost::intrusive::unordered_set<
    LruNode, boost::intrusive::constant_time_size<true>,
    boost::intrusive::hash<LruNodeHash>,
    boost::intrusive::equal<LruNodeEqual>>;
```

# LRU #5

```
struct LruNodeHash : Hash {
  template <class NodeOrKey> auto operator()(const NodeOrKey& x) const {
    return Hash::operator()(v5::GetKey(x));
  }
};
struct LruNodeEqual : Equal {
  template <class NodeOrKey1, class NodeOrKey2>
  auto operator()(const NodeOrKey1& x, const NodeOrKey2& y) const {
    return Equal::operator()(v5::GetKey(x), v5::GetKey(y));
  }
};

using Map = boost::intrusive::unordered_set<
    LruNode, boost::intrusive::constant_time_size<true>,
    boost::intrusive::hash<LruNodeHash>,
    boost::intrusive::equal<LruNodeEqual>>;
```

# LRU #5

```
struct LruNodeHash : Hash {
  template <class NodeOrKey> auto operator()(const NodeOrKey& x) const {
    return Hash::operator()(v5::GetKey(x));
  }
};
struct LruNodeEqual : Equal {
  template <class NodeOrKey1, class NodeOrKey2>
  auto operator()(const NodeOrKey1& x, const NodeOrKey2& y) const {
    return Equal::operator()(v5::GetKey(x), v5::GetKey(y));
  }
};

using Map = boost::intrusive::unordered_set<
    LruNode, boost::intrusive::constant_time_size<true>,
    boost::intrusive::hash<LruNodeHash>,
    boost::intrusive::equal<LruNodeEqual>>;
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                    // noexcept
  list_.insert(list_.end(), *node);      // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```cpp
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                     // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                       // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                        // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                     // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                        // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                    // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                       // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #4

```
bool Put(const T& key) {
  auto it = map_.find(key);
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(it->second));
    return false;
  }
  if (map_.size() == max_size_) {
    auto node = map_.extract(GetLeastRecentKey());
    list_.splice(list_.end(), list_, list_.begin());
    node.key() = key;
    map_.insert(std::move(node));
  } else {
    auto [it, ok] = map_.emplace(key, LruNode{});
    assert(ok);
    list_.insert(list_.end(), it->second);
  }
  return true;
}
```

# LRU #5

```
bool Put(const T& key) {
  auto it = map_.find(key, map_.hash_function(), map_.key_eq());
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(*it));
    return false;
  }
  if (map_.size() == buckets_.size()) {
    auto node = ExtractNode(list_.begin());
    node->SetKey(key);
    InsertNode(std::move(node));
  } else {
    auto node = std::make_unique<LruNode>(T(key));
    InsertNode(std::move(node));
  }
  return true;
}
```

# LRU #5

```
std::unique_ptr<LruNode> ExtractNode(typename List::iterator it) noexcept {
  std::unique_ptr<LruNode> ret(&*it);
  map_.erase(map_.iterator_to(*it));
  list_.erase(it);
  return ret;
}

void InsertNode(std::unique_ptr<LruNode> node) noexcept {
  if (!node) return;
  map_.insert(*node);                     // noexcept
  list_.insert(list_.end(), *node);  // noexcept
  [[maybe_unused]] auto ignore = node.release();
}
```

# LRU #5

```
bool Put(const T& key) {
  auto it = map_.find(key, map_.hash_function(), map_.key_eq());
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(*it));
    return false;
  }
  if (map_.size() == buckets_.size()) {
    auto node = ExtractNode(list_.begin());
    node->SetKey(key);
    InsertNode(std::move(node));
  } else {
    auto node = std::make_unique<LruNode>(T(key));
    InsertNode(std::move(node));
  }
  return true;
}
```

# LRU #5

```
bool Put(const T& key) {
  auto it = map_.find(key, map_.hash_function(), map_.key_eq());
  if (it != map_.end()) {
    list_.splice(list_.end(), list_, list_.iterator_to(*it));
    return false;
  }
  if (map_.size() == buckets_.size()) {
    auto node = ExtractNode(list_.begin());
    node->SetKey(key);
    InsertNode(std::move(node));
  } else {
    auto node = std::make_unique<LruNode>(T(key));
    InsertNode(std::move(node));
  }
  return true;
}
```

# LRU benchmark

```
Put<v1::LruSet<unsigned>>                95907 ns      95905 ns       7306
Put<v2::LruSet<unsigned>>                96491 ns      96489 ns       7085
Put<v3::LruSet<unsigned>>                95387 ns      95384 ns       6908
Put<v4::LruSet<unsigned>>                64934 ns      64933 ns      10200
Put<v5::LruSet<unsigned>>                52176 ns      52175 ns      12075

Has<v1::LruSet<unsigned>>                25036 ns      25035 ns      27432
Has<v2::LruSet<unsigned>>                11694 ns      11694 ns      63427
Has<v3::LruSet<unsigned>>                11799 ns      11799 ns      56695
Has<v4::LruSet<unsigned>>                 9617 ns       9616 ns      70000
Has<v5::LruSet<unsigned>>                 9773 ns       9773 ns      70022

PutOverflow<v1::LruSet<unsigned>>        70947 ns      70946 ns       9376
PutOverflow<v2::LruSet<unsigned>>        65341 ns      65338 ns      11186
PutOverflow<v3::LruSet<unsigned>>        43203 ns      43202 ns      16447
PutOverflow<v4::LruSet<unsigned>>        41245 ns      41244 ns      17054
PutOverflow<v5::LruSet<unsigned>>        14276 ns      14276 ns      47829
```

# LRU #6

# LRU #6 ???

# Можно и быстрее!

# Можно и быстрее!

- Использовать хеширование по степени 2ки

# Можно и быстрее!

- Использовать хеширование по степени 2ки
- Для количества элементов меньших 4kkk (unsigned int max) мы можем заменить указатели на unit32_t

# Можно и быстрее!

- Использовать хеширование по степени 2ки
- Для количества элементов меньших 4kkk (unsigned int max) мы можем заменить указатели на unit32_t
  - на x86 sizeof(void*) * 4 - sizeof(unit32_t) * 4 == 16 байт экономии на элемент

# Можно и быстрее!

- Использовать хеширование по степени 2ки
- Для количества элементов меньших 4kkk (unsigned int max) мы можем заменить указатели на unit32_t
  - на x86 sizeof(void*) * 4 - sizeof(unit32_t) * 4 == 16 байт экономии на элемент
- Для количества элементов меньших 65k (unsigned short max) мы можем заменить указатели на unit16_t
  - на x86 sizeof(void*) * 4 — sizeof(unit16_t) * 4 == 24 байт экономии на элемент

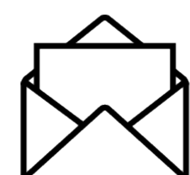
# Можно и быстрее!

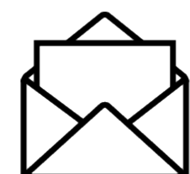
- Использовать хеширование по степени 2ки
- Для количества элементов меньших 4kkk (unsigned int max) мы можем заменить указатели на unit32_t
  - на x86 sizeof(void*) * 4 - sizeof(unit32_t) * 4 == 16 байт экономии на элемент
- Для количества элементов меньших 65k (unsigned short max) мы можем заменить указатели на unit16_t
  - на x86 sizeof(void*) * 4 — sizeof(unit16_t) * 4 == 24 байт экономии на элемент
- ???

# Спасибо

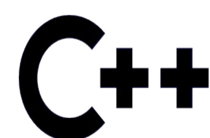
# Полухин Антон

Эксперт-разработчик C++



antoshkka@gmail.com


antoshkka@yandex-team.ru

https://github.com/apolukhin



https://stdcpp.ru/

https://t.me/CppQuizzBot

# Спасибо