Про `flat_map`

Who needs them? They're just like **`std::map`**. We all have them.

Pavel Novikov

@cpp_ape

You will learn

- a refresher for the standard associative containers and when to use them
- what is **flat map** and when to use it
  - Boost `flat_map` and `std::flat_map`
    + performance comparison with standard associative containers
- what else can we do with **flat map**?

- a refresher for the standard associative containers and when to use them
- what is **flat map** and when to use it
  - Boost `flat_map` and `std::flat_map`
    + performance comparison with standard associative containers
- what else can we do with **flat map**?

You will learn

C++ Russia
2023

Что-то у меня тормозит: заглядываем внутрь C++ контейнеров

Илья
Шишков

Яндекс

```
std::vector<std::pair<int, std::string>> getItems();
```

```
std::unordered_map<int, std::string> getItems();
```

```
std::map<int, std::string> getItems();
```

Standard associative containers

```cpp
std::vector<std::pair<int, std::string>> getItems();

std::unordered_map<int, std::string> getItems();

std::map<int, std::string> getItems();
```

Standard associative containers

```
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```
std::unordered_map<int, std::string> getItems();
```

```
std::map<int, std::string> getItems();
```

Standard associative containers

```
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```
std::unordered_map<int, std::string> getItems();


std::map<int, std::string> getItems();
```

Standard associative containers

```
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```
std::unordered_map<int, std::string> getItems();
```

- unique keys
- unordered

```
std::map<int, std::string> getItems();
```

Standard associative containers

```
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```
std::unordered_map<int, std::string> getItems();
```

- unique keys
- unordered

```
std::map<int, std::string> getItems();
```

Standard associative containers

```cpp
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```cpp
std::unordered_map<int, std::string> getItems();
```

- unique keys
- unordered

```cpp
std::map<int, std::string> getItems();
```

- unique keys
- ordered

Standard associative containers

```
std::vector<std::pair<int, std::string>> getItems();
```

- non-unique "keys"
- unordered (w.r.t. "keys")

```
std::unordered_map<int, std::string> getItems();
```

- unique keys
- unordered

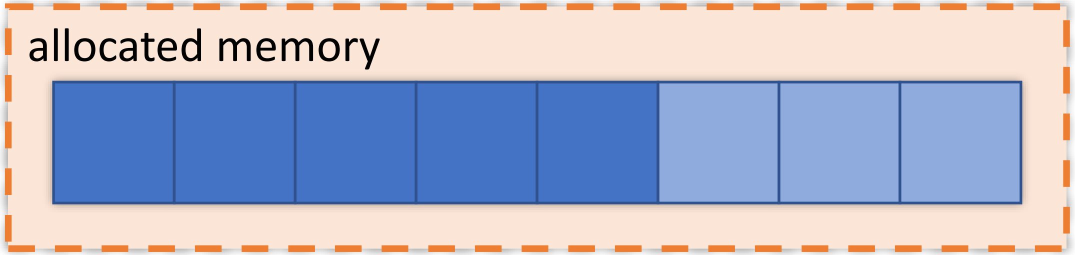
```
std::map<int, std::string> getItems();
```

- unique keys
- ordered

```
flat_map<int, std::string> getItems();
```

Standard associative containers

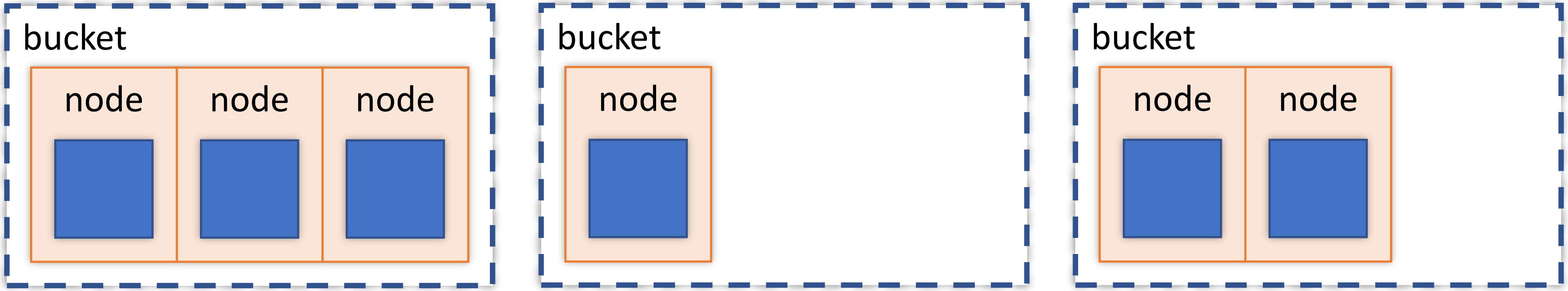
std::vector<std::pair<int, std::string>>



Complexity:

search	linear ***O(N)***
element insertion/removal	linear ***O(N)***
element insertion at the end	amortized constant ***O(1)***

```
std::unordered_map<Key, Value, std::hash<Key>,
                               std::equal_to<Key>>
```

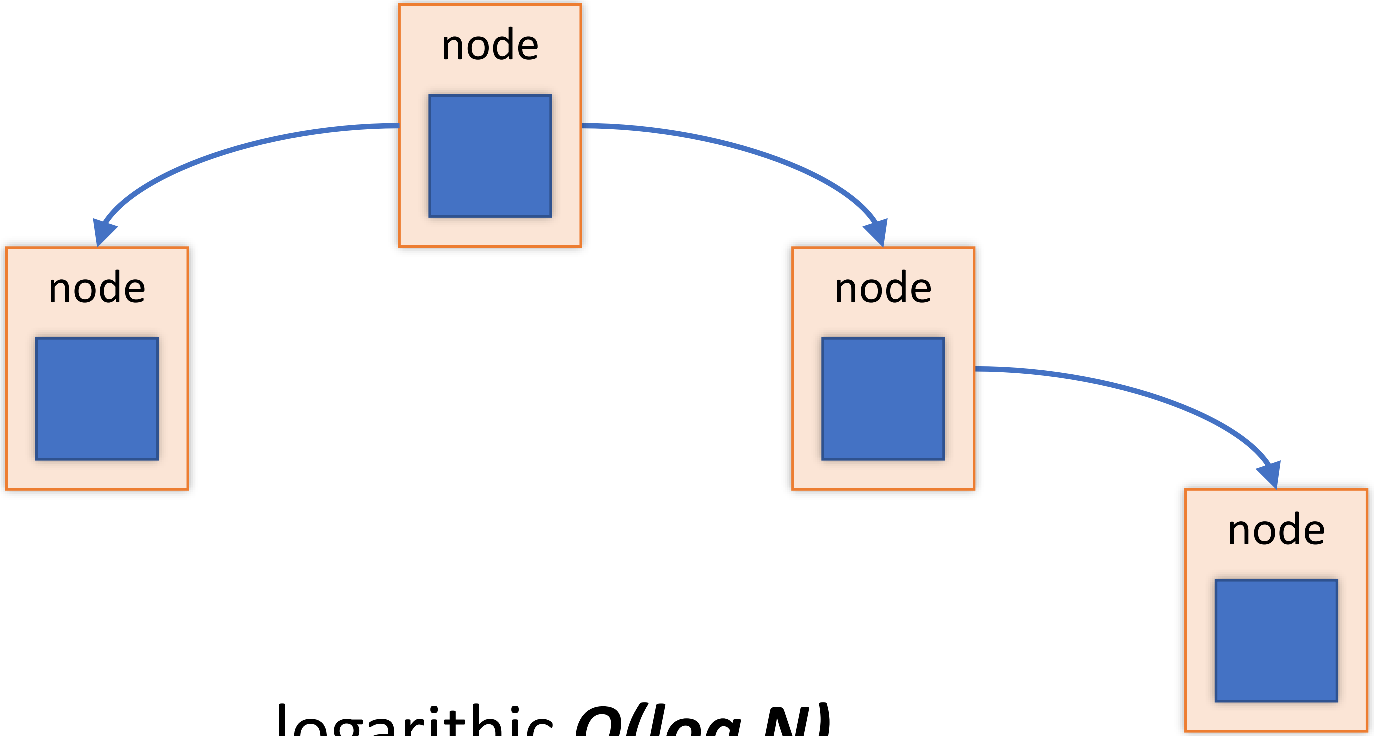


Complexity:

search	constant ***O(1)*** on average
element insertion/removal	constant ***O(1)*** on average

Standard associative containers

```
std::map<int, std::string, std::less<int>>
```



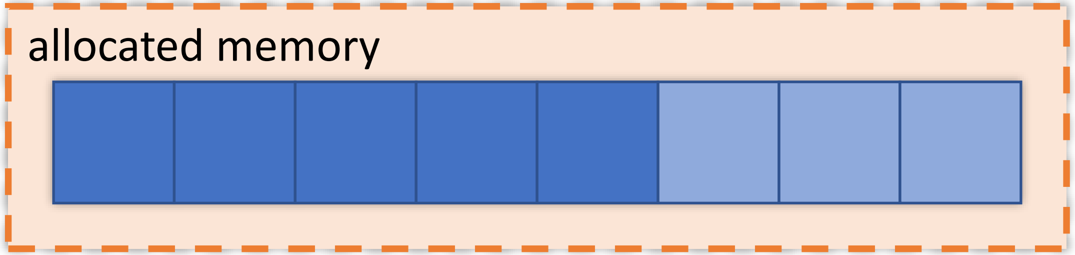
Complexity:

search	logarithmic ***O(log N)***
element insertion/removal	logarithmic ***O(log N)***

Standard associative containers

```
flat_map<int, std::string, std::less<int>>
```

usually adapts vector-like container



Complexity:

search	logarithic ***O(log N)***
element insertion/removal	linear ***O(N)***

Flat map associative container

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container =
    std::vector<std::pair<Key, Value>>
>
class FlatMap;
```

Let’s make a flat map

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container =
    std::vector<std::pair<Key, Value>>
>
class FlatMap;
```

Let’s make a flat map

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container =
    std::vector<std::pair<Key, Value>>
>
class FlatMap;
```

Let’s make a flat map

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container =
    std::vector<std::pair<Key, Value>>
>
class FlatMap;
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  Container container;

public:
  using key_type = Key;
  using mapped_type = Value;
  using value_type = std::pair<const key_type, mapped_type>;
  using key_compare = Compare;
  //...
```

for empty base optimization

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  Container container;

public:
  using key_type = Key;
  using mapped_type = Value;
  using value_type = std::pair<const key_type, mapped_type>;
  using key_compare = Compare;
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  Container container;

public:
  using key_type = Key;
  using mapped_type = Value;
  using value_type = std::pair<const key_type, mapped_type>;
  using key_compare = Compare;
  //...
```

```cpp
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  Container container;

public:
  using key_type = Key;
  using mapped_type = Value;
  using value_type = std::pair<const key_type, mapped_type>;
  using key_compare = Compare;
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using reference =
    std::pair<const key_type&, mapped_type&>;
  using const_reference =
    std::pair<const key_type&, const mapped_type&>;
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using reference =
    std::pair<const key_type&, mapped_type&>;
  using const_reference =
    std::pair<const key_type&, const mapped_type&>;
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using size_type = size_t;
  using difference_type = ptrdiff_t;
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using iterator =
    Iterator<const Key, Value,
             typename Container::iterator>;
  using const_iterator =
    Iterator<const Key, const Value,
             typename Container::const_iterator>;
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using iterator =
    Iterator<const Key, Value,
             typename Container::iterator>;
  using const_iterator =
    Iterator<const Key, const Value,
             typename Container::const_iterator>;
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using reverse_iterator =
    std::reverse_iterator<iterator>;
  using const_reverse_iterator =
    std::reverse_iterator<const_iterator>;
  using container_type = Container;
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  using reverse_iterator =
    std::reverse_iterator<iterator>;
  using const_reverse_iterator =
    std::reverse_iterator<const_iterator>;
  using container_type = Container;
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap() = default;

  size_t size() const { return container.size(); }
  //...
```

Let’s make a flat map

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  iterator begin() {
    return iterator{ container.begin() };
  }
  const_iterator begin() const {
    return const_iterator{ container.begin() };
  }
  const_iterator cbegin() const { return begin(); }
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  iterator end() {
    return iterator{ container.end() };
  }
  const_iterator end() const {
    return const_iterator{ container.end() };
  }
  const_iterator cend() const { return end(); }
  //...
```

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  auto needle = findNeedle(key, predicate);
                                           // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

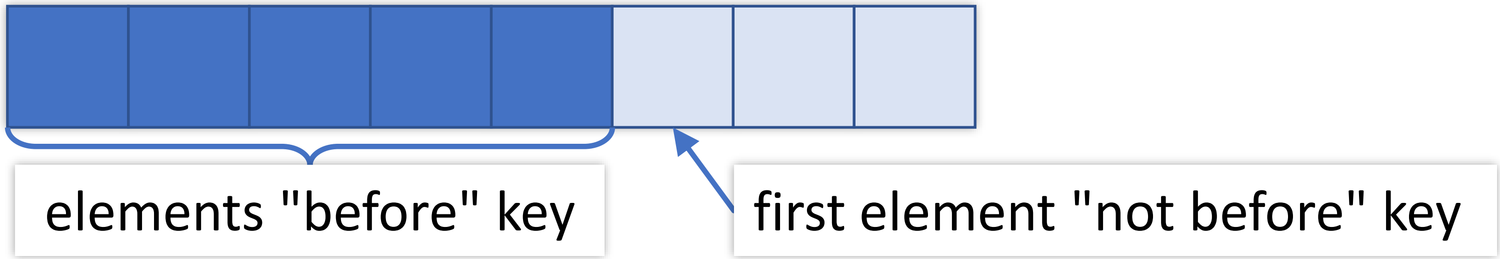
Let’s make a flat map

```
auto findNeedle(const Key &key, const Predicate &predicate) {
  return std::lower_bound(container.begin(),
                          container.end(),
                          key,
                          predicate);
}
```

binary search

`std::lower_bound()` finds *"first element not before"** key

**comment in MSVC standard library source code*



elements "before" key

first element "not before" key

Let’s make a flat map

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
struct Predicate {
  explicit Predicate(const Compare &comp) : comp{ comp } {}

  auto operator()(const Key &a, const Key &b) const {
    return comp(a, b);
  }
  auto operator()(const Key &a, const typename Container::value_type &b) const {
    return comp(a, b.first);
  }
  auto operator()(const typename Container::value_type &a, const Key &b) const {
    return comp(a.first, b);
  }
  auto operator()(const typename Container::value_type &a,
                  const typename Container::value_type &b) const {
    return comp(a.first, b.first);
  }

  const Compare &comp;
};
```

```
Predicate makePredicate() {
  return Predicate{ *static_cast<Compare*>(this) };
}
```

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                               // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
const_iterator find(const Key &key) const {
  return const_iterator{ const_cast<FlatMap*>(this)->find(key).i };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {

  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();

  return iterator{ needle };
}
```

Let’s make a flat map

```
std::pair<iterator, bool> insert(const value_type &value) {
  const auto &key = value.first;
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return { iterator{ container.insert(needle, value) }, true };

  return { iterator{ needle }, false };
}
```

Let’s make a flat map

```
std::pair<iterator, bool> insert(const value_type &value) {
  const auto &key = value.first;
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return { iterator{ container.insert(needle, value) }, true };

  return { iterator{ needle }, false };
}
```

Let’s make a flat map

```
std::pair<iterator, bool> insert(const value_type &value) {
  const auto &key = value.first;
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return { iterator{ container.insert(needle, value) }, true };

  return { iterator{ needle }, false };
}
```

Let’s make a flat map

```
std::pair<iterator, bool> insert(const value_type &value) {
  const auto &key = value.first;
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return { iterator{ container.insert(needle, value) }, true };

  return { iterator{ needle }, false };
}
```

Let’s make a flat map

```
std::pair<iterator, bool> insert(value_type &&value) {
  const auto &key = value.first;
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return { iterator{ container.insert(needle, std::move(value)) },
             true };

  return { iterator{ needle }, false };
}
```

Let’s make a flat map

```
iterator find(const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return end();



  return iterator{ needle };
}
```

Let’s make a flat map

```
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return container.emplace(needle,
                             std::piecewise_construct,
                             std::forward_as_tuple(key),
                             std::tuple<>())->second;
  return needle->second;
}
```

Let’s make a flat map

```
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return container.emplace(needle,
                             std::piecewise_construct,
                             std::forward_as_tuple(key),
                             std::tuple<>())->second;
  return needle->second;
}
```

Let’s make a flat map

```
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                              // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return container.emplace(needle,
                             std::piecewise_construct,
                             std::forward_as_tuple(key),
                             std::tuple<>())->second;
  return needle->second;
}
```

this behavior is required by the C++ standard

```
template<class... Args1, class... Args2>
pair(std::piecewise_construct_t,
     std::tuple<Args1...> first_args,
     std::tuple<Args2...> second_args);
```

Let’s make a flat map

```cpp
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return container.emplace(needle,
                             std::piecewise_construct,
                             std::forward_as_tuple(key),
                             std::tuple<>())->second;
  return needle->second;
}
```

Let’s make a flat map

```
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```cpp
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```cpp
auto map = FlatMap<int, int>{};

map[1] = 23;

map.insert(std::pair{ 2, 42 });

const auto v = std::pair{ 3, 9000 };
map.insert(v);

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

Let’s make a flat map

```
for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

```
auto &&__range = map;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
  const auto &[key, value] = *__begin;
  // loop-statement
}
```

Let’s make a flat map

```
for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

```
auto &&__range = map;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
  const auto &[key, value] = *__begin;
  // loop-statement
}
```

Let’s make a flat map

```cpp
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```cpp
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  temp
};
```

```
using iterator       = Iterator<const Key, Value,
                                typename Container::iterator>;
using const_iterator = Iterator<const Key, const Value,
                                typename Container::const_iterator>;
```

```
template<typename K, typename V, typename I>
struct Iterator {
  //...
  std::pair<K&, V&> operator*() const;
  //...
};

auto map = FlatMap<int, int>{};
//...
*map.begin();
```

returns
`std::pair<const int&, int&>`
not a reference!

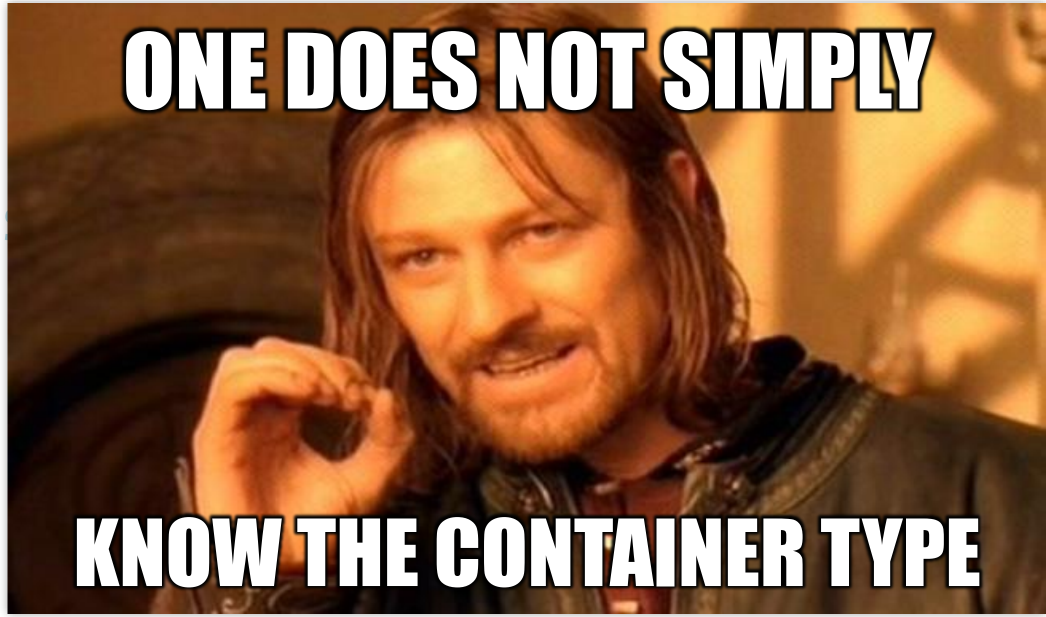
The iterator nuance

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container =
    std::vector<std::pair<Key, Value>>
>
class FlatMap;

//...
using value_type = std::pair<const key_type, mapped_type>;
//...
```

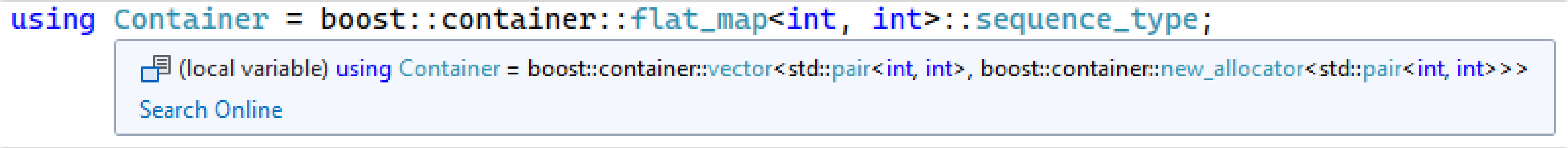
The iterator nuance

```
auto map = boost::container::flat_map<int, int>{};
auto map = boost::container::flat_map<int, int>{};

//...
typedef typename sequence_type::iterator iterator;
typedef typename sequence_type::const_iterator const_iterator;
//...
```

```
auto map = boost::container::flat_map<int, int>{};

//...
typedef typename sequence_type::iterator iterator;
typedef typename sequence_type::const_iterator const_iterator;
//...

template <class Key
         ,class T
         ,class Compare  = std::less
         ,class Allocator = void >
class flat_map;
```



```
auto map = boost::container::flat_map<int, int>{};
```



```
using Container = boost::container::flat_map<int, int>::sequence_type;
```

(local variable) using Container = boost::container::vector<std::pair<int, int>, boost::container::new_allocator<std::pair<int, int>>>

The iterator nuance

```
auto map = boost::container::flat_map<int, int>{};
```

```
boost::container::vector<std::pair<int, int>>
```

The iterator nuance

```
auto map = boost::container::flat_map<int, int>{};


boost::container::vector<std::pair<int, int>>

                  int     int
//...
typedef std::pair<Key, T> value_type;
//...
```

The iterator nuance

The iterator nuance

```
auto map = boost::container::flat_map<int, int>{};
```

```
boost::container::vector<std::pair<int, int>>
```

int int

```
//...
typedef std::pair<Key, T> value_type;
//...
```

```
*map.begin();
```

returns
std::pair<int, int>& meh...

```
auto map = boost::container::flat_map<const int, int>{};
```

```
boost::container::vector<std::pair<const int, int>>
```

`const int` `int`

```
//...
typedef std::pair<Key, T> value_type;
//...
```

```
*map.begin();
```

returns
`std::pair<const int, int>&`

The iterator nuance

The iterator nuance

```
auto map = FlatMap<int, int>{};
```

```
std::vector<std::pair<int, int>>
```

```
//...
using value_type = std::pair<const key_type, mapped_type>;
//...
```

```
*map.begin();
```

returns

`std::pair<const int&, int&>`

```
for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

```
auto &&__range = map;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
  const auto &[key, value] = *__begin;
  // loop-statement
}
```

returns
`std::pair<const int&, int&>`

The iterator nuance

```
for (auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

```
auto &&__range = map;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
  auto &[key, value] = *__begin; // does not work
  // loop-statement
}
```

returns
std::pair<const int&, int&>

The iterator nuance

```
std::vector v1 = { 1, 2, 3 };
std::vector v2 = { 23, 42, 9000 };
// does not work
for (auto &[a, b] : std::views::zip(v1, v2))
  std::cout << a << "\t" << b << '\n';
```

```
*std::views::zip(v1, v2).begin();
```

returns

`std::tuple<int&, int&>`

The iterator nuance

Recommendation:

```
for (auto &&v : range)
  // loop body
```

The iterator nuance

Recommendation:

```
for (auto &&v : range)
  // loop body
```

```
// works
for (auto &&[a, b] : std::views::zip(v1, v2))
  std::cout << a << "\t" << b << '\n';
```

The iterator nuance

```
for (auto &&[key, value] : map)
  std::cout << key << "\t" << value << '\n';
```

```
auto &&__range = map;
auto __begin = __range.begin();
auto __end = __range.end();
for (; __begin != __end; ++__begin) {
  auto &&[key, value] = *__begin; // works
  // loop-statement
}
```

returns
`std::pair<const int&, int&>`

The iterator nuance

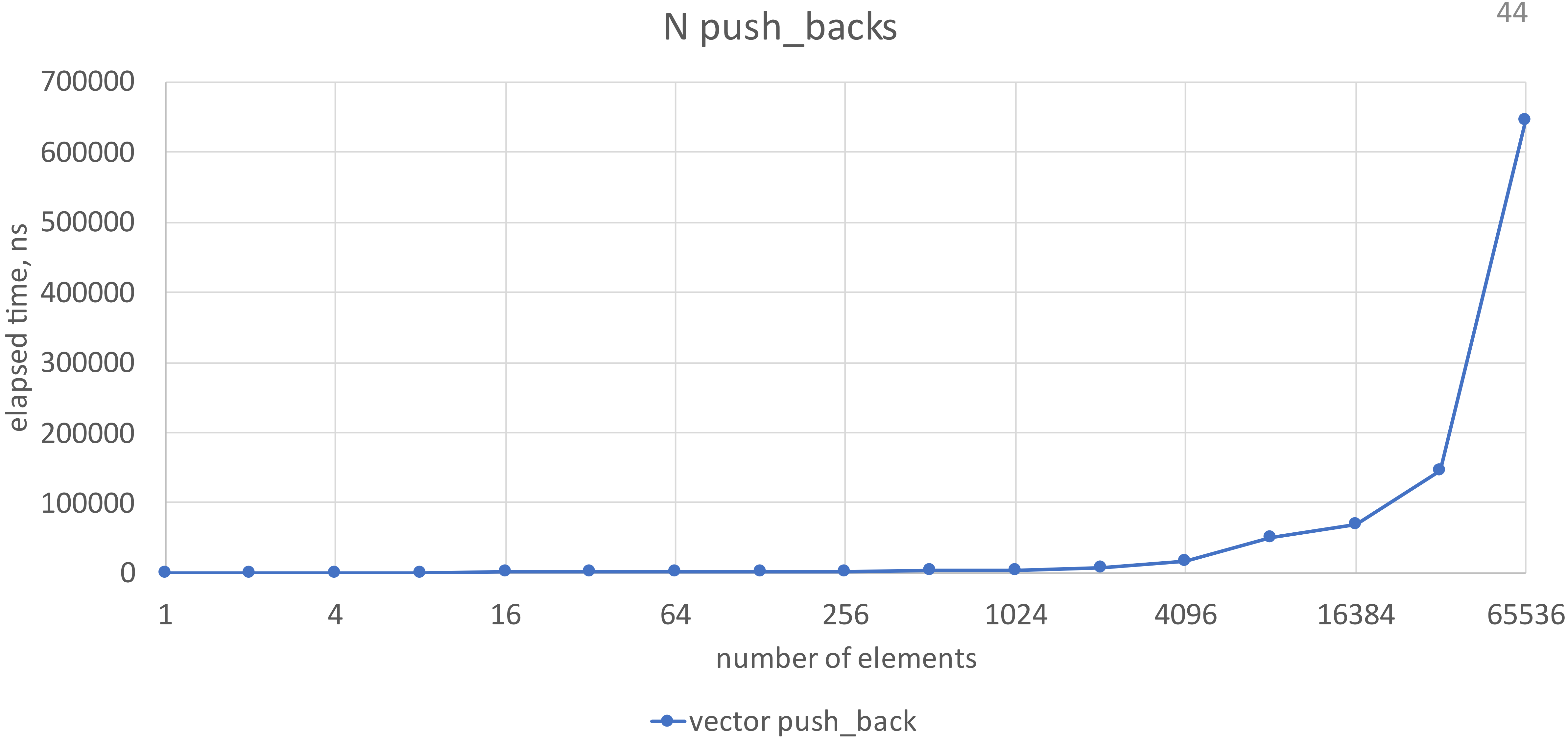
```
auto map = FlatMap<int, int>{};
//...

for (const auto &[key, value] : map)
  std::cout << key << "\t" << value << '\n';

for (     auto &&[key, value] : map) {
  // can modify 'value', but not 'key'
  ++value;
}
```

The iterator nuance

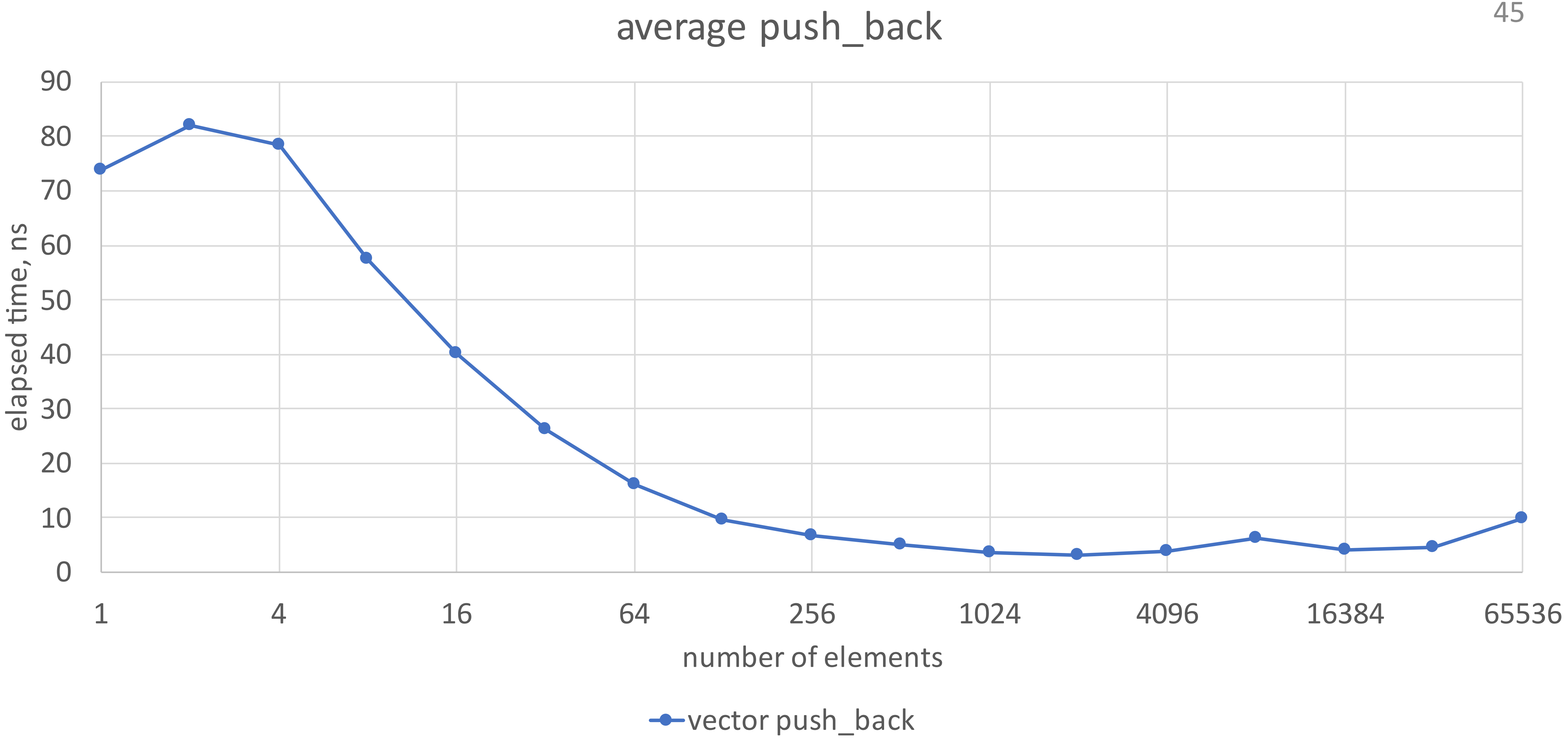
```
namespace std {
  template<
    class Key,
    class T,
    class Compare = less<Key>,
    class KeyContainer = vector<Key>,
    class MappedContainer = vector<T>>
  class flat_map;
}
```

P0429 A Standard `flat_map` by Zach Laine

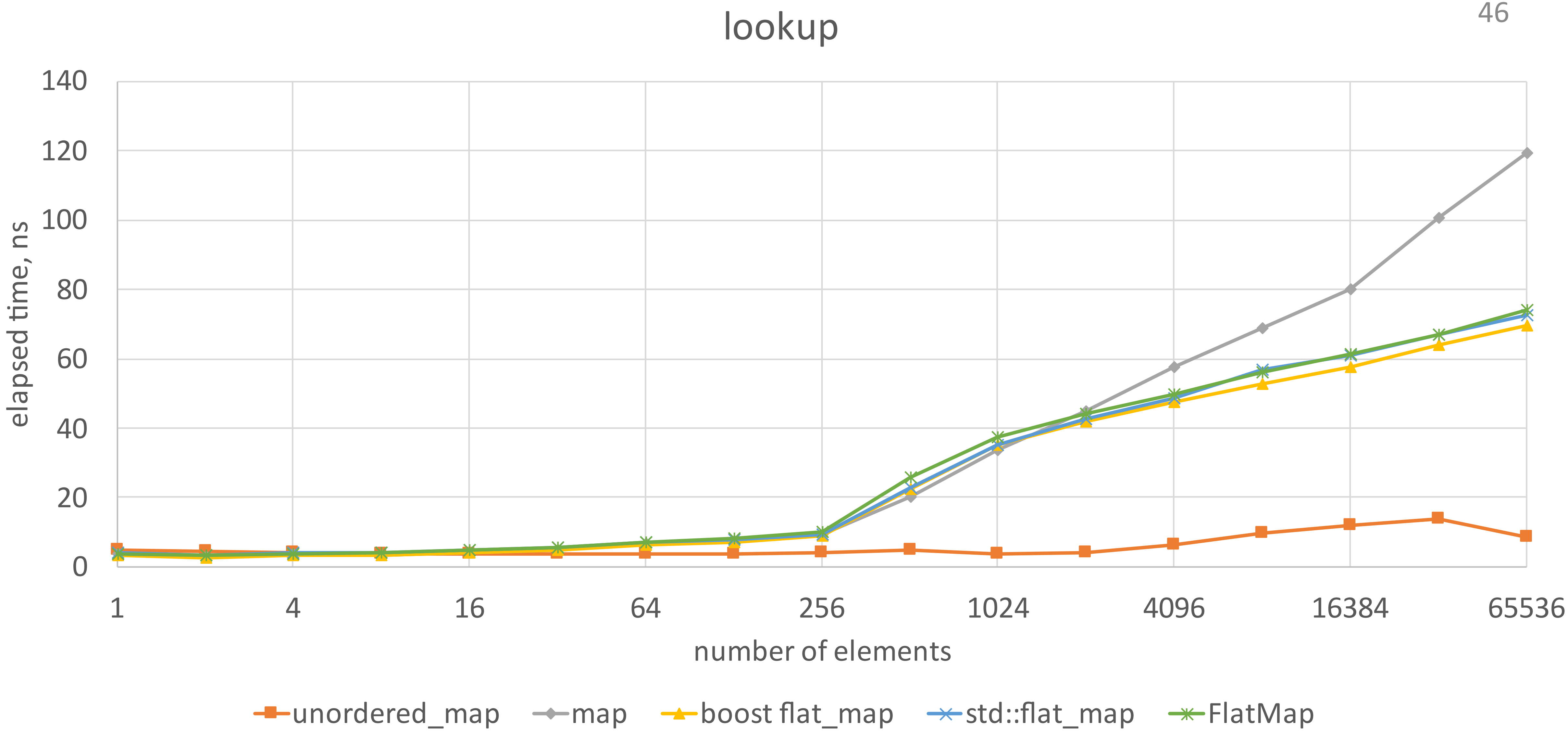
The iterator nuance



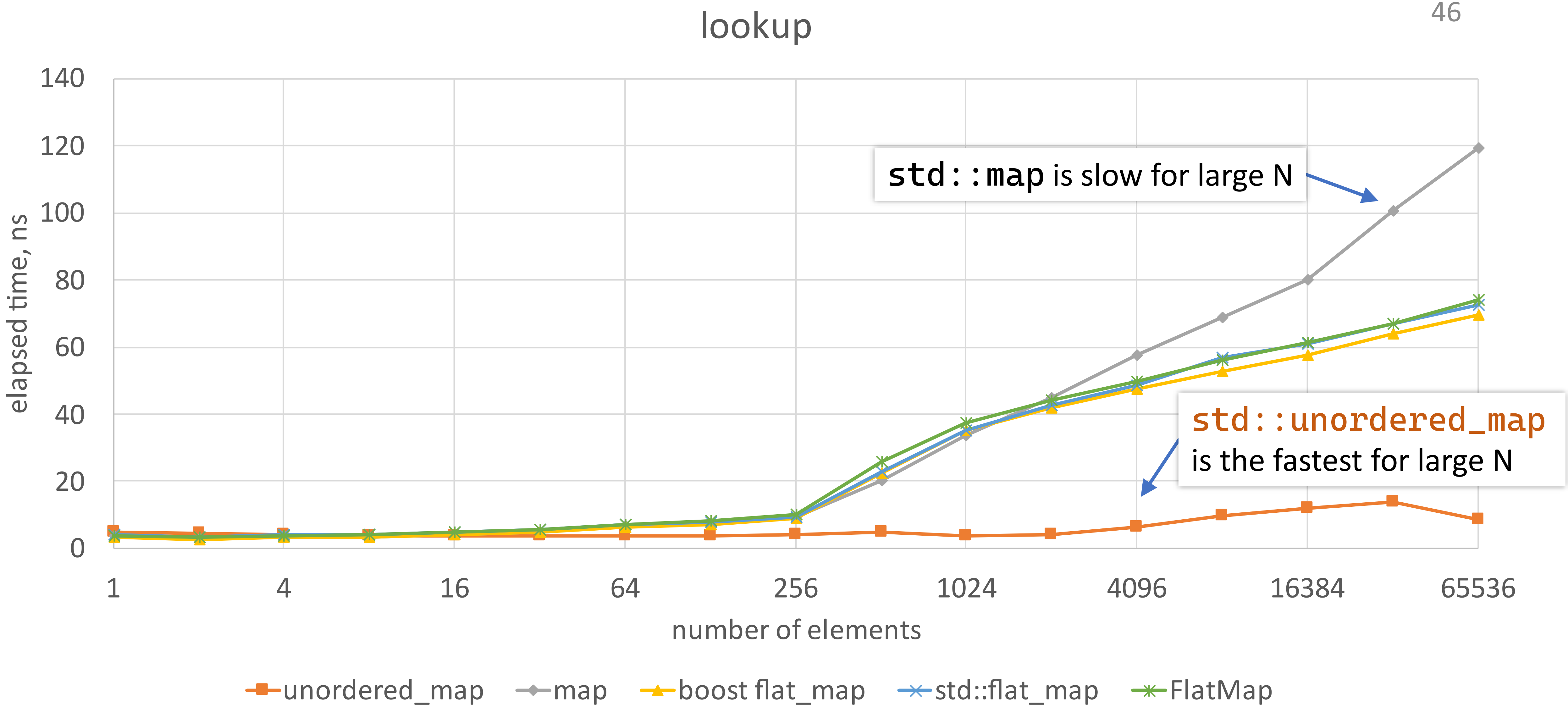
Benchmarking



Benchmarking



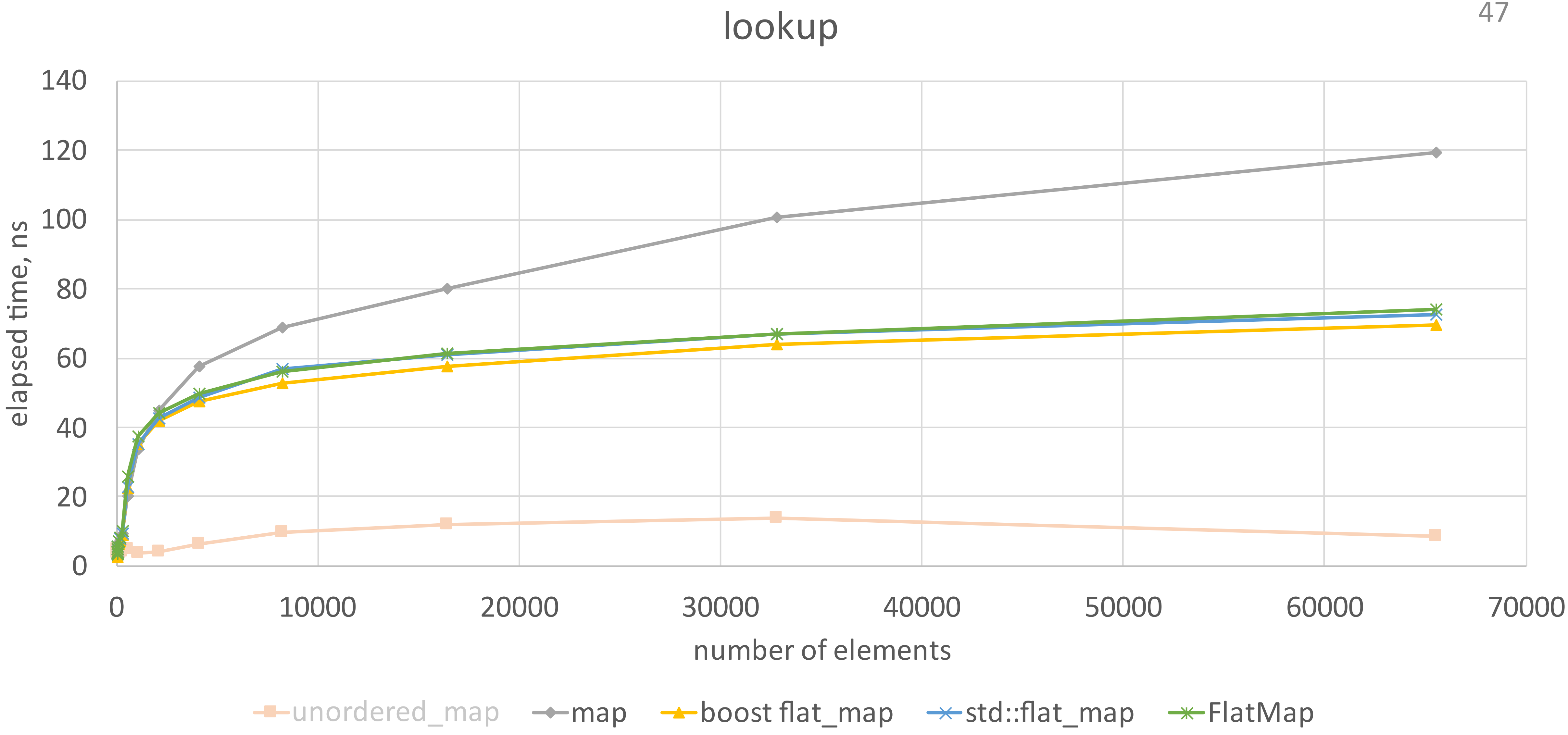
Benchmarking



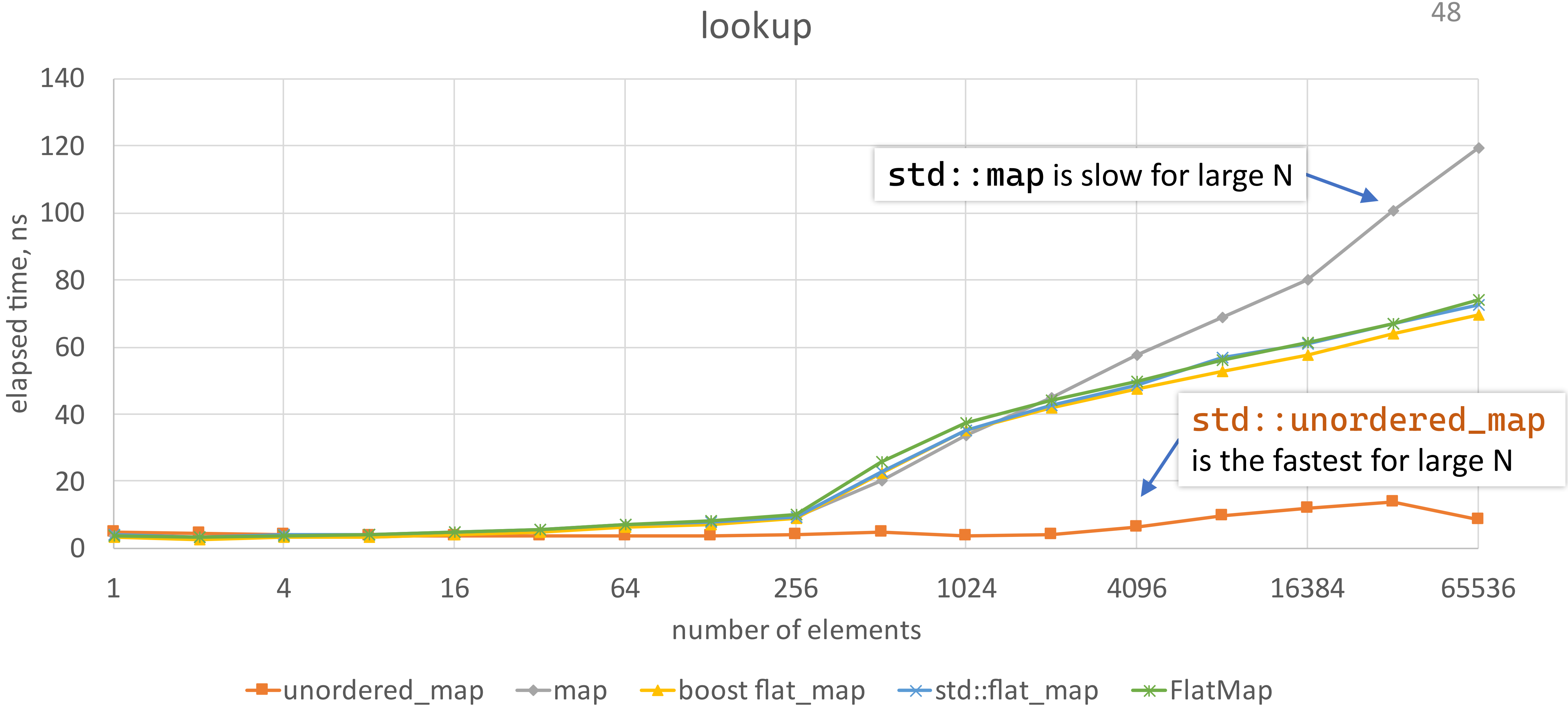
`std::map` is slow for large N

`std::unordered_map` is the fastest for large N

Benchmarking



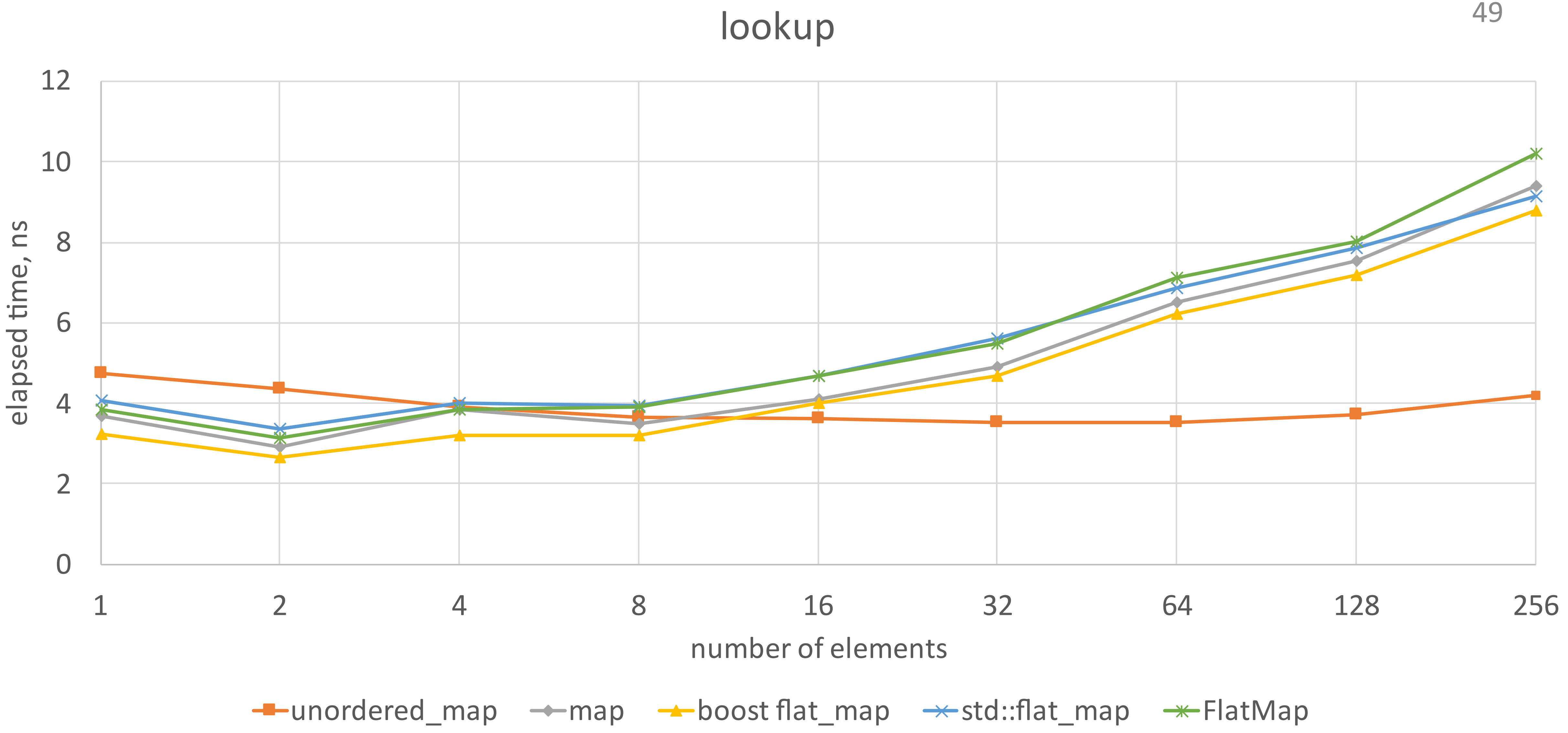
Benchmarking



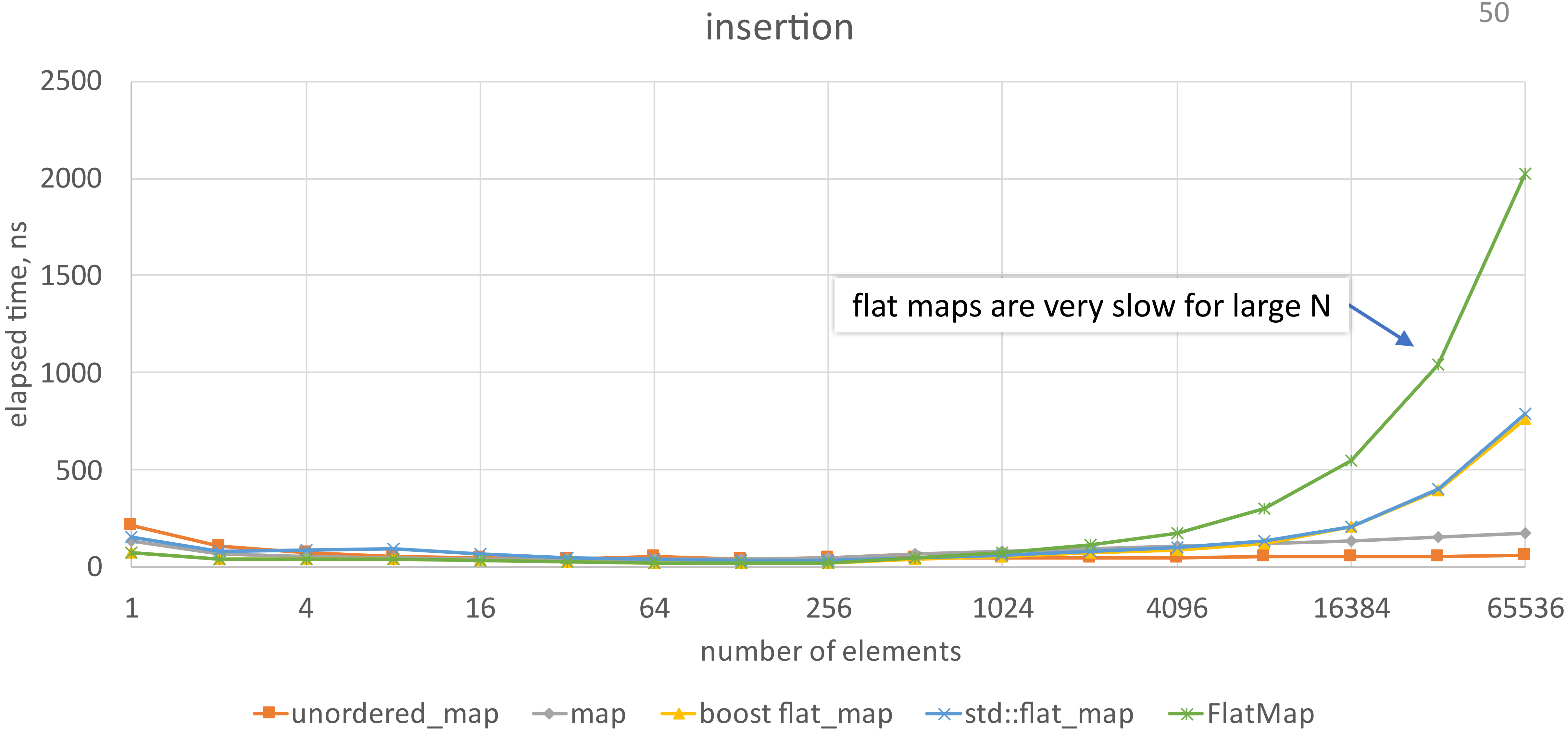
`std::map` is slow for large N

`std::unordered_map` is the fastest for large N

Benchmarking

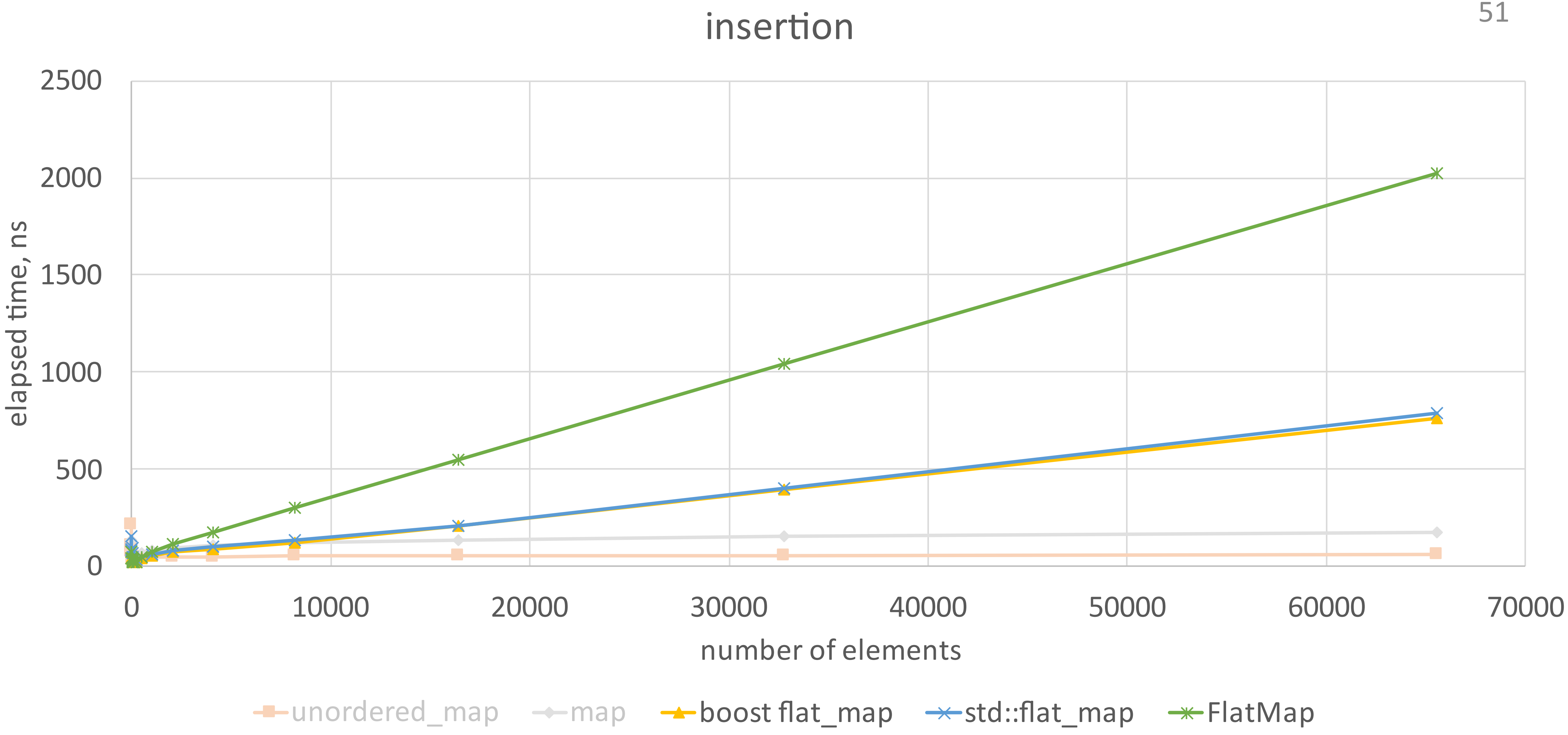


Benchmarking

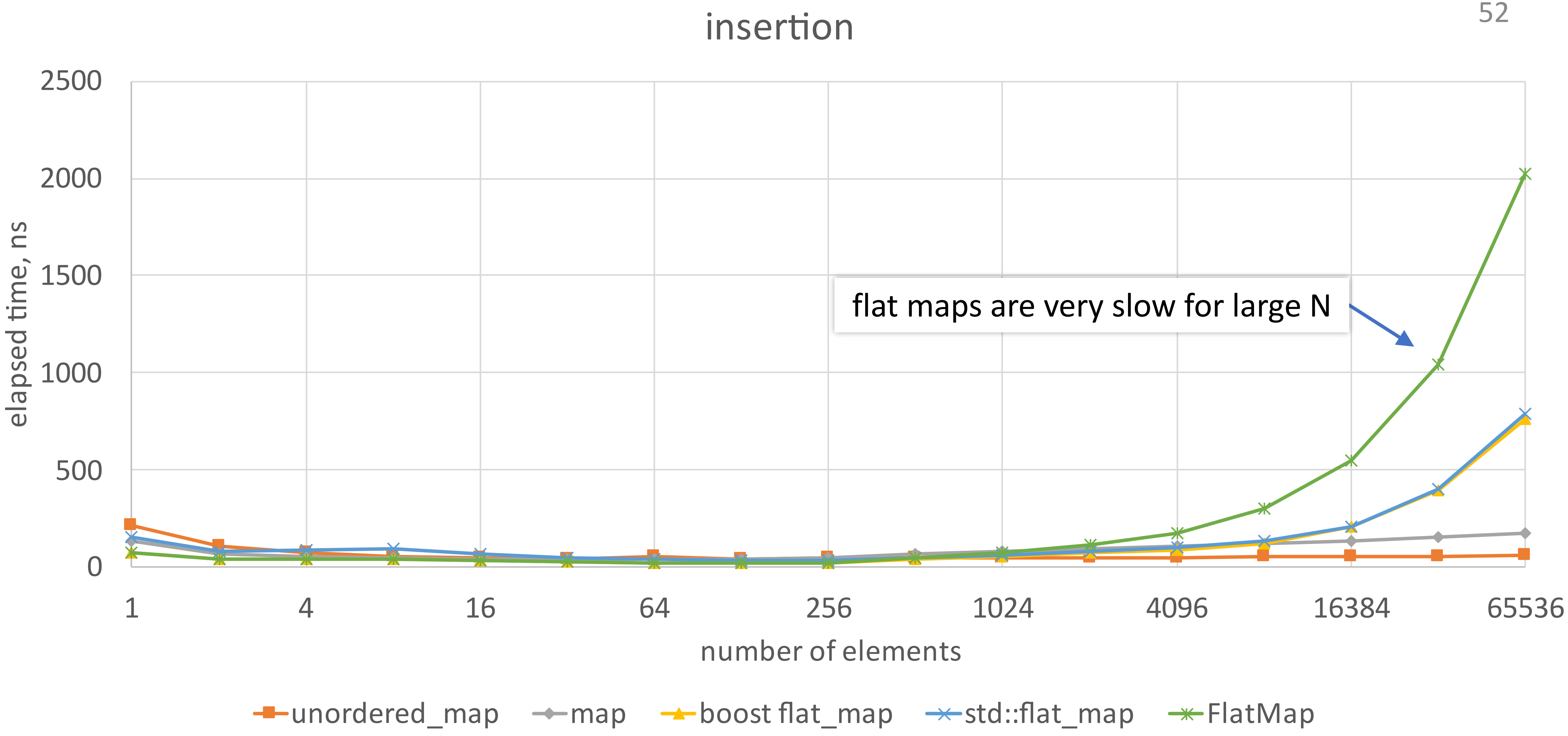


flat maps are very slow for large N

Benchmarking

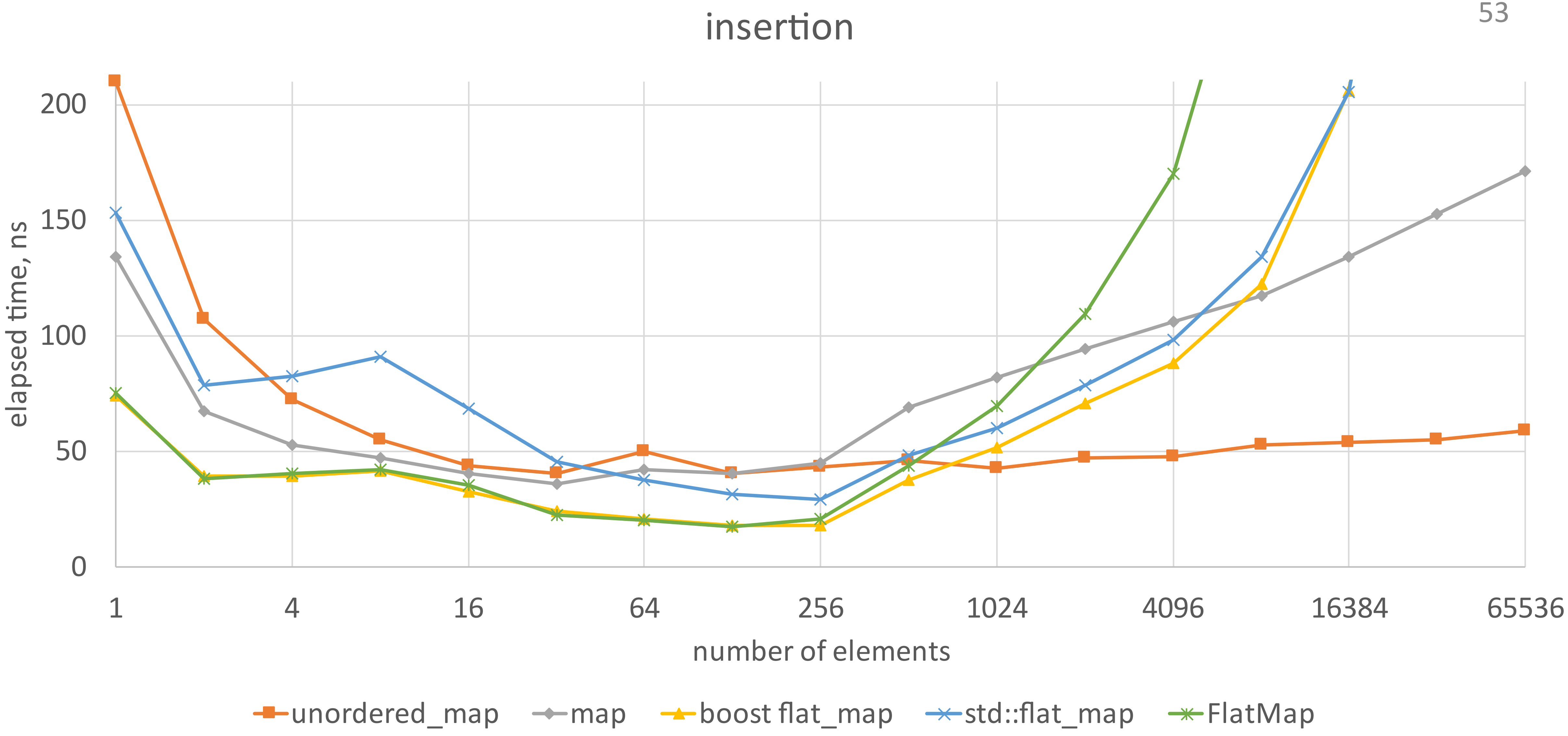


Benchmarking



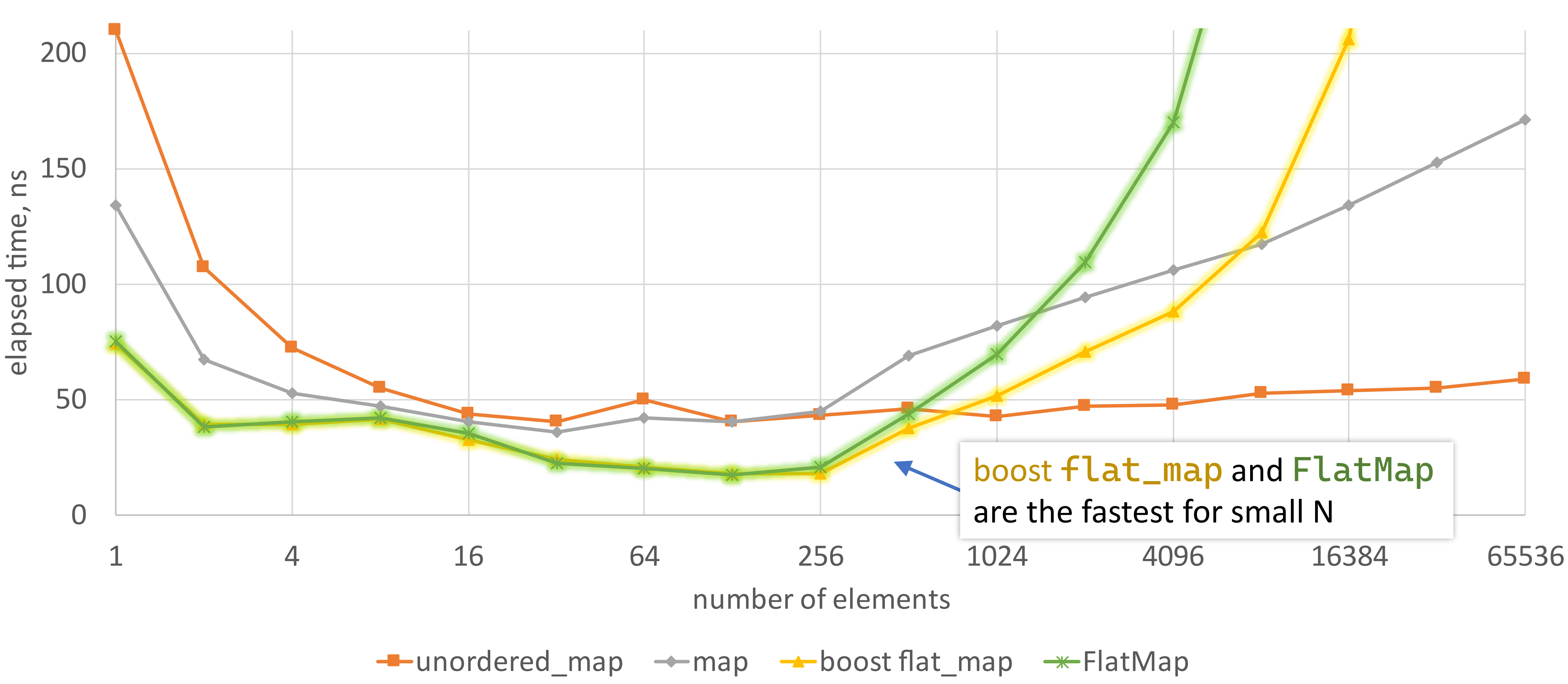
flat maps are very slow for large N

Benchmarking



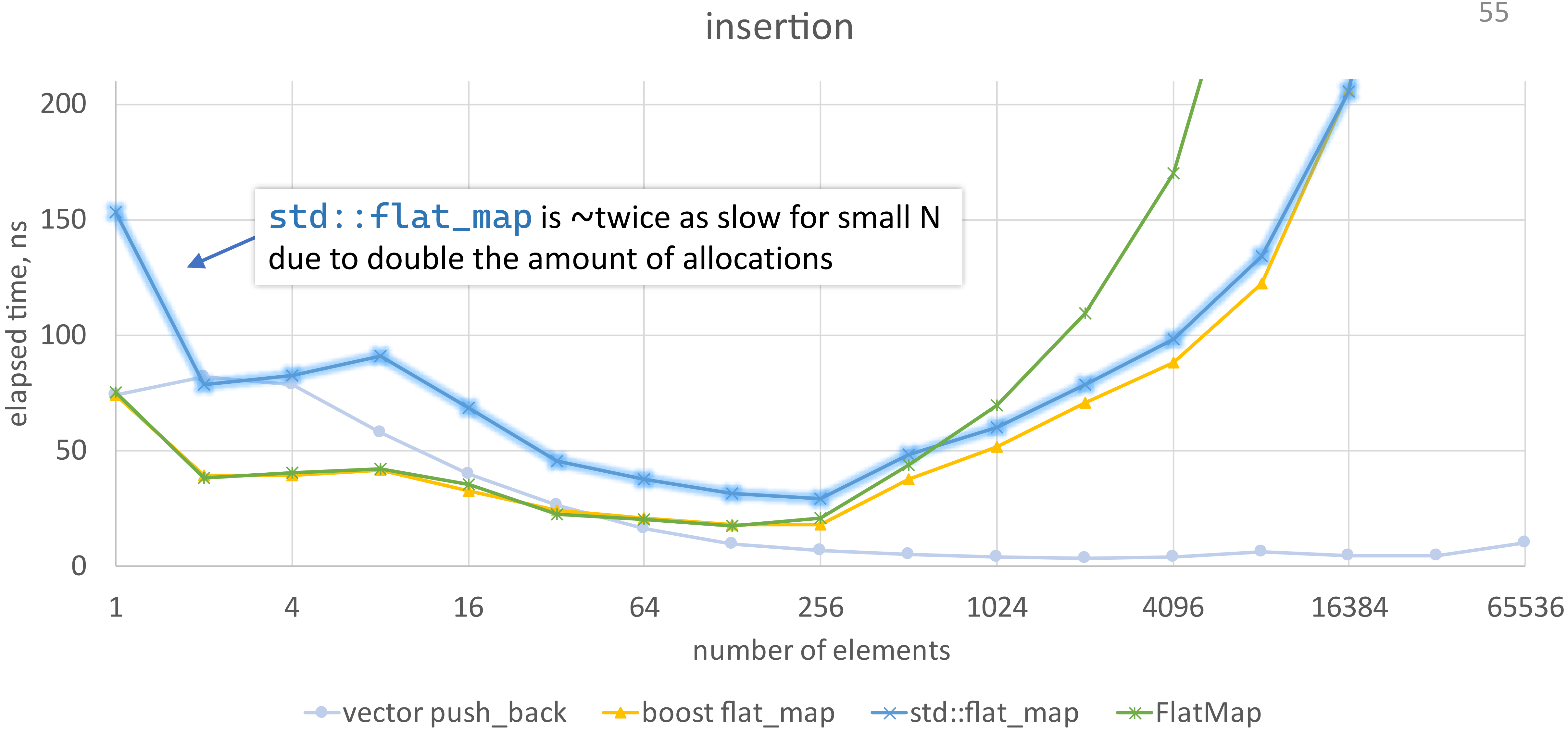
Benchmarking

insertion



boost `flat_map` and `FlatMap` are the fastest for small N

Benchmarking



`std::flat_map` is ~twice as slow for small N due to double the amount of allocations

Benchmarking

```
namespace std {
  template<
    class Key,
    class T,
    class Compare = less<Key>,
    class KeyContainer = vector<Key>,
    class MappedContainer = vector<T>>
  class flat_map;
}
```

Benchmarking

```
namespace std {
  template <class Key, class T, class Compare = less<Key>,
            class KeyContainer = vector<Key>,
            class MappedContainer = vector<T>>
    class flat_map {
    //...
    using key_container_type = KeyContainer;
    using mapped_container_type = MappedContainer;
    //...
    struct containers {
      key_container_type keys;
      mapped_container_type values;
    };
    //...
    private:
      containers c; // exposition only
  };
}
```

Flat maps should be used:

- for small constant maximum number of elements (known at compile time)
  e.g. for maximum number of elements = 30?

Flat maps may be used:

- to do *mostly lookups* if `std::unordered_map` is unavailable to use
  e.g. initialization at once, and then only doing lookups
- to do *mostly iterations in order*
  e.g. rare lookups/insertions/erasures, frequent iterations

When to use flat map?

When to use flat map?

Flat maps should be used:

- for small constant maximum number of elements (known at compile time)
  e.g. for maximum number of elements = 30?

Flat maps may be used:

- to do *mostly lookups* if `std::unordered_map` is unavailable to use
  e.g. initialization at once, and then only doing lookups
- to do *mostly iterations in order*
  e.g. rare lookups/insertions/erasures, frequent iterations

Flat maps should be used:

- for small constant maximum number of elements (known at compile time)
  e.g. for maximum number of elements = 30?

Flat maps may be used:

- to do *mostly lookups* if `std::unordered_map` is unavailable to use
  e.g. initialization at once, and then only doing lookups
- to do *mostly iterations in order*
  e.g. rare lookups/insertions/erasures, frequent iterations

When to use flat map?

Universal advice:

- measure
- measure
- measure

When to use flat map?

```
auto map = boost::container::flat_map<int, int>{};
//...
auto items = map.extract_sequence();
```

My favourite feature of flat maps

```
auto map = boost::container::flat_map<int, int>{};
//...
~~boost::container::vector<std::pair<int, int>> items =~~
  ~~map.extract_sequence();~~

boost::container::vector<
  std::pair<int, int>,
  boost::container::new_allocator<std::pair<int, int>>>
  items = map.extract_sequence();
```

My favourite feature of flat maps

```cpp
using BoostFlatMap = boost::container::flat_map<int, int,
  std::less<int>,
  std::vector<std::pair<int, int>>>;

auto map = BoostFlatMap{};
//...
std::vector<std::pair<int, int>> items =
  map.extract_sequence();
```

My favourite feature of flat maps

```
using BoostFlatMap = boost::container::flat_map<int, int,
  std::less<int>,
  std::vector<std::pair<int, int>>>;

auto map = BoostFlatMap{};
//...
std::vector<std::pair<int, int>> items =
  map.extract_sequence();
```

My favourite feature of flat maps

```
using BoostFlatMap = boost::container::flat_map<int, int,
  std::less<int>,
  std::vector<std::pair<int, int>>>;

std::vector<std::pair<int, int>> items;
//...
auto map = BoostFlatMap{};
map.adopt_sequence(std::move(items));
```

My favourite feature of flat maps

```cpp
using BoostFlatMap = boost::container::flat_map<int, int,
  std::less<int>,
  std::vector<std::pair<int, int>>>;

auto map = BoostFlatMap{};
//...
auto items = map.extract_sequence();
//...
map.adopt_sequence(std::move(items));
```

My favourite feature of flat maps

```
using BoostFlatMap = boost::container::flat_map<int, int,
  std::less<int>,
  std::vector<std::pair<int, int>>>;

auto map = BoostFlatMap{};
//...
auto items = map.extract_sequence();
//...
map.adopt_sequence(boost::container::ordered_unique_range,
                   std::move(items));
```

My favourite feature of flat maps

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  container_type extract() {
    return std::move(container);
  }
  //...
```

My favourite feature of flat maps

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap(container_type &&c) : container{ std::move(c) } {
    std::sort(container.begin(), container.end(),
              makePredicate());
    auto end = std::unique(container.begin(), container.end(),
                           makeEqualToPredicate());
    container.erase(end, container.end());
  }
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap(container_type &&c) : container{ std::move(c) } {
    std::sort(container.begin(), container.end(),
              makePredicate());
    auto end = std::unique(container.begin(), container.end(),
                           makeEqualToPredicate());
    container.erase(end, container.end());
  }
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap(container_type &&c) : container{ std::move(c) } {
    std::sort(container.begin(), container.end(),
              makePredicate());
    auto end = std::unique(container.begin(), container.end(),
                           makeEqualToPredicate());
    container.erase(end, container.end());
  }
  //...
```

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap(container_type &&c) : container{ std::move(c) } {
    std::sort(container.begin(), container.end(),
              makePredicate());
    auto end = std::unique(container.begin(), container.end(),
                           makeEqualToPredicate());
    container.erase(end, container.end());
  }
  //...
```

```
auto makeEqualToPredicate() {
  struct EqualTo {
    auto operator()(const typename Container::value_type &a,
                    const typename Container::value_type &b) const {
       const auto &k1 = a.first;
       const auto &k2 = b.first;
       return !(comp(k1, k2) || comp(k2, k1));
    }

    const Compare &comp;
  };
  return EqualTo{ *static_cast<Compare *>(this) };
}
```

My favourite feature of flat maps

```
template<typename Key,
         typename Value,
         typename Compare,
         typename Container>
class FlatMap : private Compare {
  //...
  FlatMap(SortedUnique_t, container_type &&c) :
    container{ std::move(c) }
  {}
  //...
```

```
struct SortedUnique_t {} inline constexpr SortedUnique;
```

My favourite feature of flat maps

```
auto map = FlatMap<int, int>{};
//...
std::vector<std::pair<int, int>> items = map.extract();
//...
map = { std::move(items) };
```

My favourite feature of flat maps

```cpp
auto map = FlatMap<int, int>{};
//...
auto items = map.extract();
//...
map = { std::move(items) };
```

My favourite feature of flat maps

```
auto map = FlatMap<int, int>{};
//...
auto items = map.extract();
//...
map = { SortedUnique, std::move(items) };
```

My favourite feature of flat maps

My favourite feature of flat maps

```
auto map = std::flat_map<int, int>{};
//...
std::flat_map<int, int>::containers items =
  std::move(map).extract();
```

```
struct containers {
  key_container_type keys;
  mapped_container_type values;
};
```

```
auto map = std::flat_map<int, int>{};
//...
auto containers = std::move(map).extract();
```

must be moved

My favourite feature of flat maps

```cpp
auto map = std::flat_map<int, int>{};
//...
auto [keys, values] = std::move(map).extract();
//...
map = { std::move(keys), std::move(values) };
```

My favourite feature of flat maps

```cpp
auto map = std::flat_map<int, int>{};
//...
auto [keys, values] = std::move(map).extract();
//...
map = { std::sorted_unique,
        std::move(keys), std::move(values) };
```

My favourite feature of flat maps

```
auto map = std::flat_map<int, int>{};
//...
auto [keys, values] = std::move(map).extract();
//...
// 'keys' should be sorted and contain unique elements
map.replace(std::move(keys), std::move(values));
```

My favourite feature of flat maps

```
using BoostFlatMap = boost::container::flat_map<
  int, int,
  std::less<int>,
  std::vector<std::tuple<int, int>>>;
```

```
v.first
v.second
```

not provided by `std::tuple`

What else could we do?

```
FlatMap<int, int,
        std::less<int>,
        std::vector<std::tuple<int, int>>
> map;
```

```
v.first          std::get<0>(v)
           →
v.second         std::get<1>(v)
```

What else could we do?

```
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return std::get<1>(
      *container.emplace(needle,
                         std::piecewise_construct,
                         std::forward_as_tuple(key),
                         std::tuple<>())
    );
  return std::get<1>(*needle);
}
```

no piecewise constructor in `std::tuple`

```
mapped_type &operator[](const Key &key) {
  const auto predicate = makePredicate();
  const auto needle = findNeedle(key, predicate);
                                          // key < *needle
  if (needle == container.end() or predicate(key, *needle))
    return std::get<1>(
      *container.emplace(needle,
                         key,
                         Value{})
    );

  return std::get<1>(*needle);
}
```

unwanted temporary object and move/copy

```
template<
  typename Key,
  typename Value,
  typename Compare = std::less<Key>,
  typename Container = std::vector<std::pair<Key, Value>>,
  auto KeyGetter = [](auto &v)->decltype(auto) {
    return std::get<0>(v); },
  auto ValueGetter = [](auto &v)->decltype(auto) {
    return std::get<1>(v); }
>
class FlatMap;
```

What else could we do?

```
struct MyStruct {
  int key;
  int value;
};

FlatMap<int, int,
        std::less<int>,
        std::vector<MyStruct>,
        [](auto &v)->decltype(auto) { return (v.key); },
        [](auto &v)->decltype(auto) { return (v.value); }
> map;
```

What else could we do?

- extend flat map to be able to use tuples
  - and arbitrary types
- add policy choice between single container and separate containers for keys and values
  - or rather make a separate type, e.g. like `zip_flat_map`?

Possible future work

Thanks for listening!

Про flat_map

Who needs them? They're just like `std::map`. We all have them.

Pavel Novikov

@cpp_ape

Thanks to Zach Laine.

Slides: bit.ly/3o2iyot

References

- P0429: A Standard `flat_map` https://wg21.link/P0429
- `std::flat_map` proof of concept implementation by Zach Laine https://github.com/tzlaine/flat_map

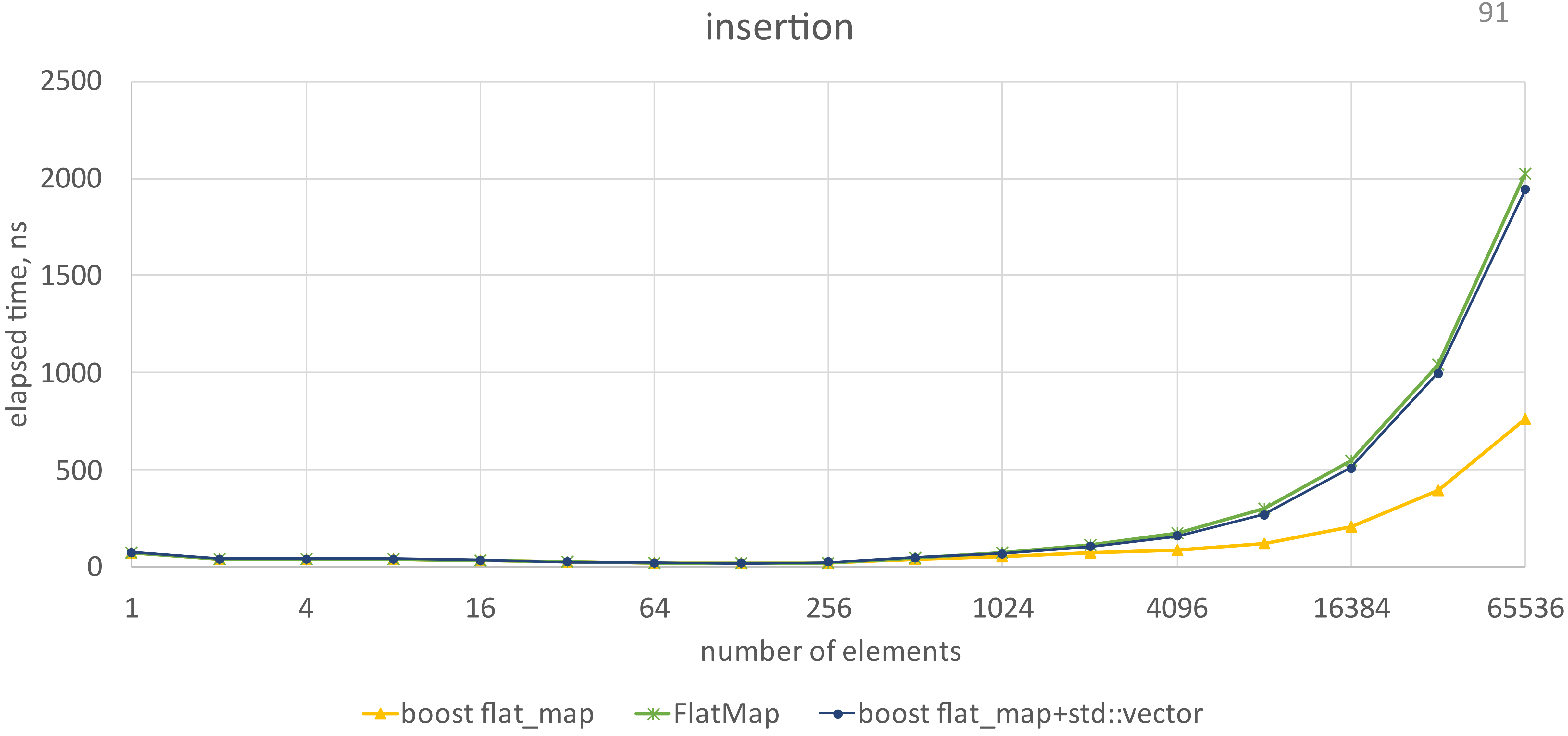
Bonus slides

```
template<typename K, typename V, typename I>
struct Iterator {
  explicit Iterator(I i) : i{ std::move(i) } {}

  Iterator &operator++() { ++i; return *this; }
  std::pair<K&, V&> operator*() const { return { i->first, i->second }; }
  auto operator->() const {
    //...
  }

  friend auto operator<=>(const Iterator&, const Iterator&) = default;

private:
  I i;

  template<typename, typename, typename, typename> friend class FlatMap;
};
```

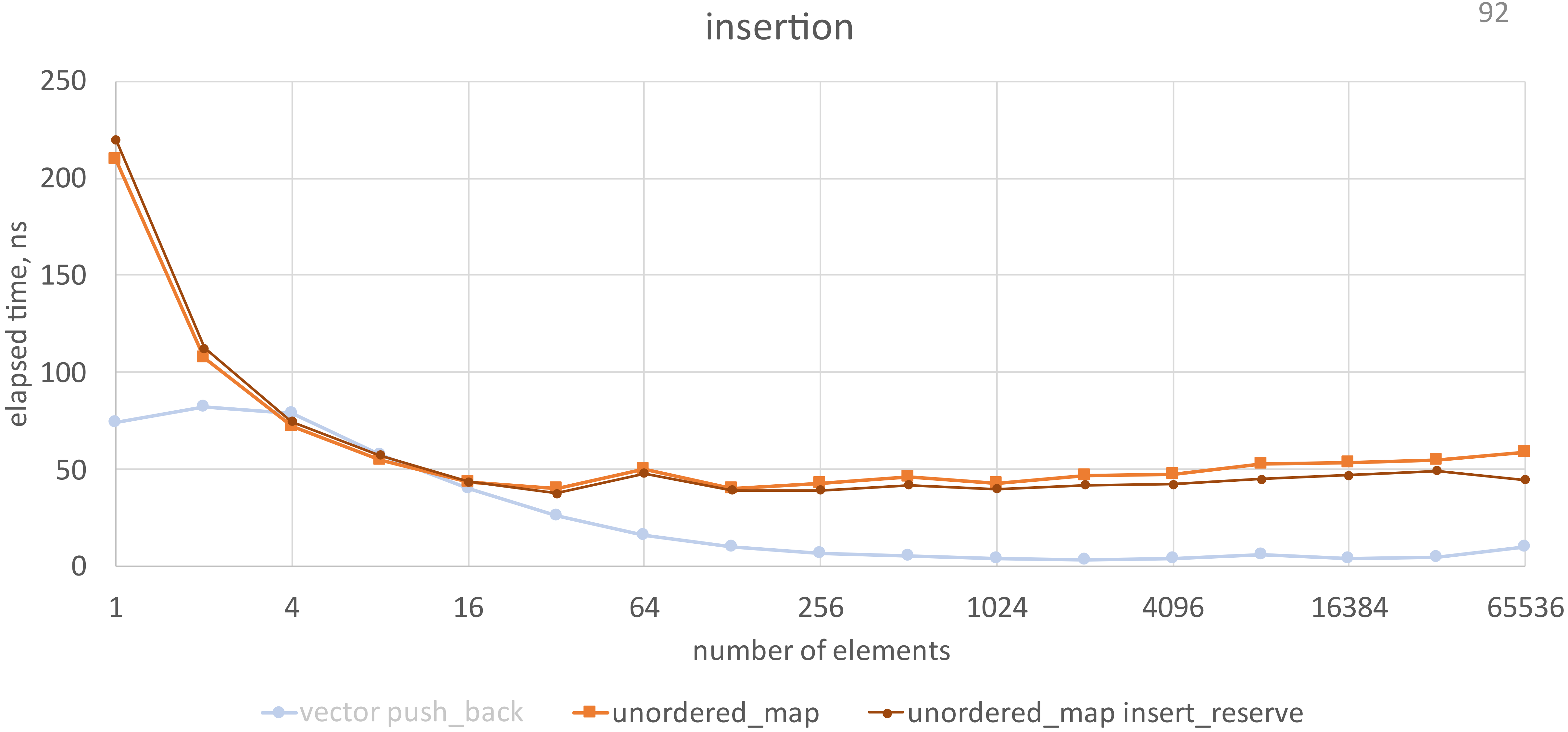
```
template<typename K, typename V, typename I>
struct Iterator {
  //...
  auto operator->() const {
    struct Helper {
      std::pair<K&, V&> ref;

      auto operator->() const { return &ref; }
    };

    return Helper{ **this };
  }
  //...
};
```

```
map.begin()->second
```

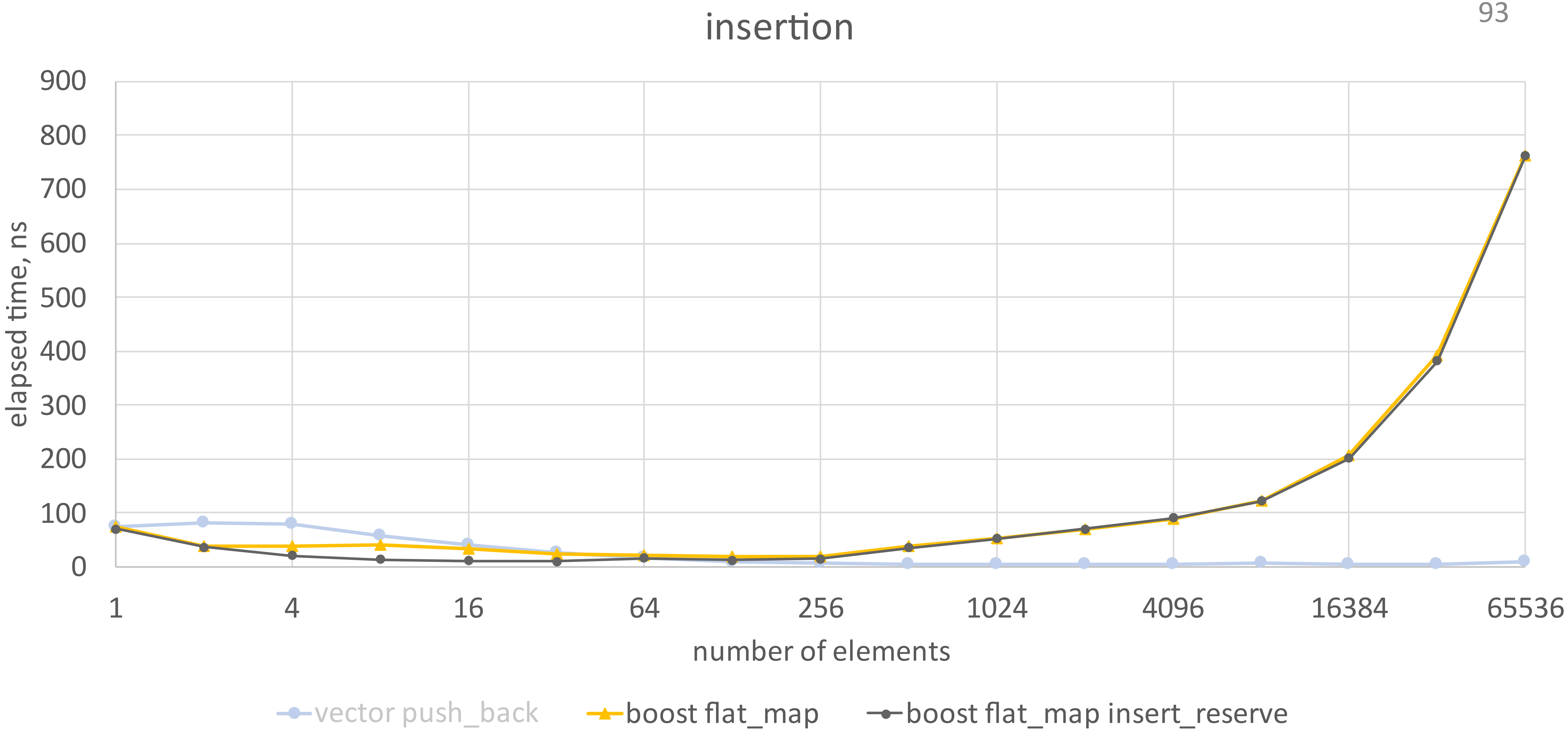
iterator operator->



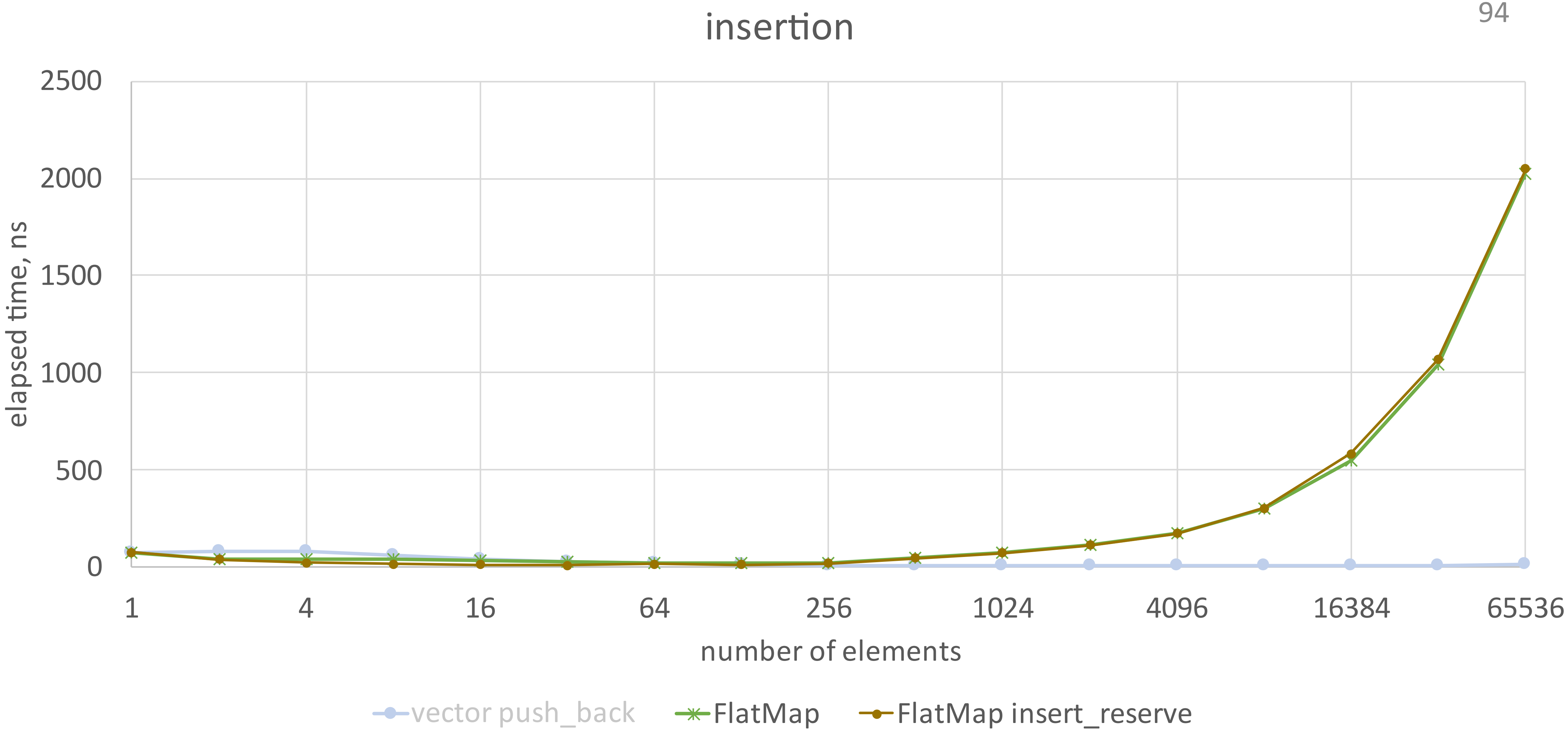
boost `flat_map` + `std::vector`



effect of reserve



effect of reserve



effect of reserve