# Ownership of Memory

## Guidelines for Dynamic Allocation

By David Stone

# The Basics

- Automatic storage ("the stack")

```
T t;
```

- Free store ("the heap")

```
new T();
std::make_unique<T>();
std::make_shared<T>();
```

# Hierarchy of Ownership

1) Automatic variables
2) std::unique_ptr
3) std::shared_ptr

Note: raw pointers are not on this list

# "Large" values and rvalues

- For this presentation, a "large" value refers to the value returned by sizeof
- std::vector<T> is small
  - 24 bytes on 64-bit
  - Regardless of number of elements
- std::array<char, 1000> is large
  - 1000 bytes

# "Large" values and rvalues

- Small values are cheap to move
- Large values are expensive to move

# Why dynamic?

# Run-time sized collections

- std::vector
- std::deque
- std::map

# Polymorphism

- Dynamic allocation is required for runtime polymorphism in function returns
  - auto create_object() -> std::unique_ptr<Base>;
- Usually required for member and local variables
  - std::unique_ptr<Base> m_some_object;
- Generally not required for function parameters

# When move is not an optimization of copy

- Some objects must remain at the same address (reference stability)
  - std::mutex
  - Anything that other objects reference
  - Multithreaded code
- std::unique_ptr<T> is always movable, even if T is not

# When move is an optimization of move

- std::unique_ptr<T> is always fast to move, even when T is not
  - std::array<int, 10000> is slow to move
- std::unique_ptr<T> requires constant stack space of one pointer
  - std::array<std::array<int, 1024>, 1024> will probably overflow your stack

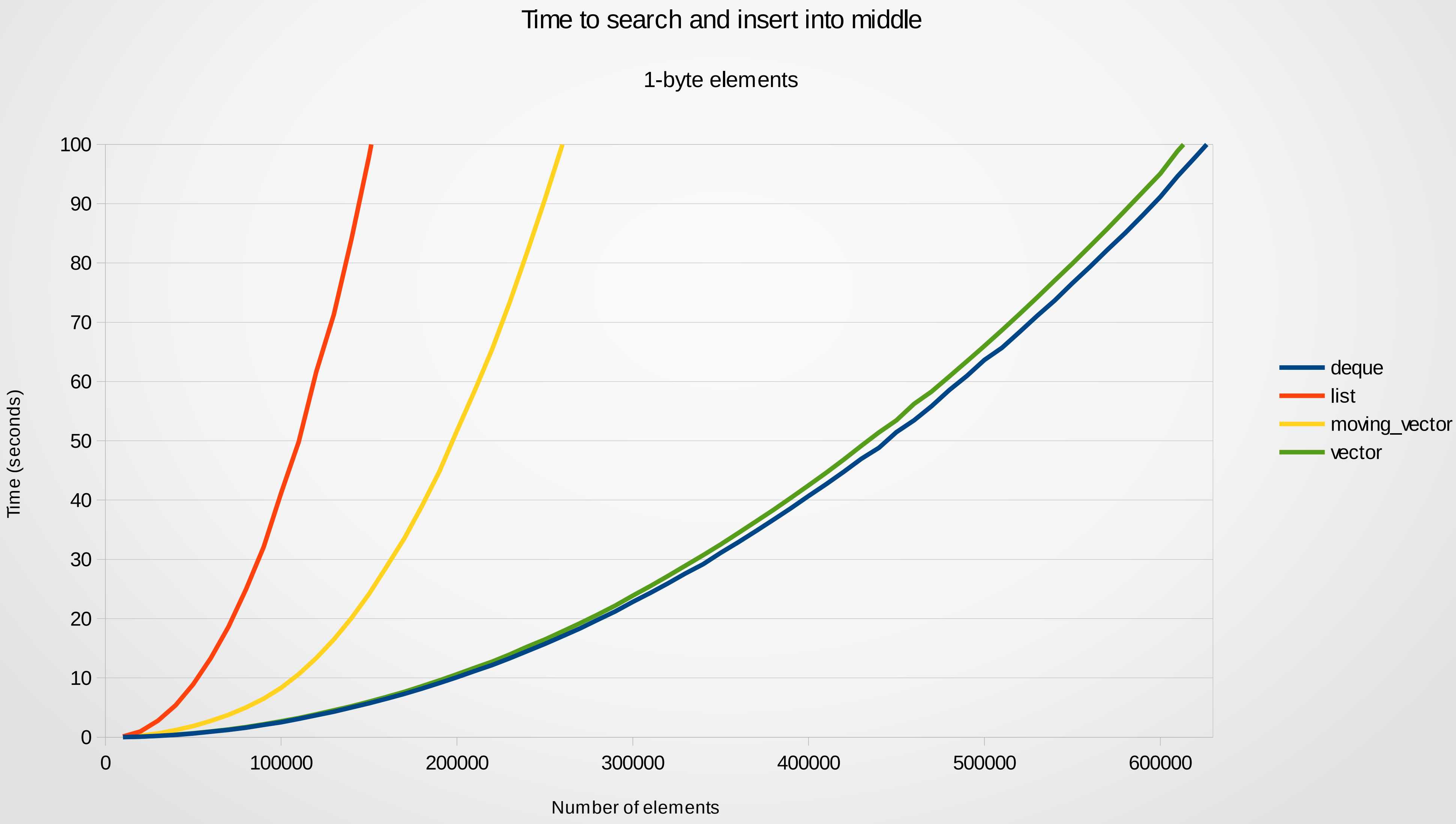
# Cache-friendliness, sequence

- std::deque<T>
- std::list<T>
- std::vector<std::unique_ptr<T>>
  - moving_vector
- std::vector<T>

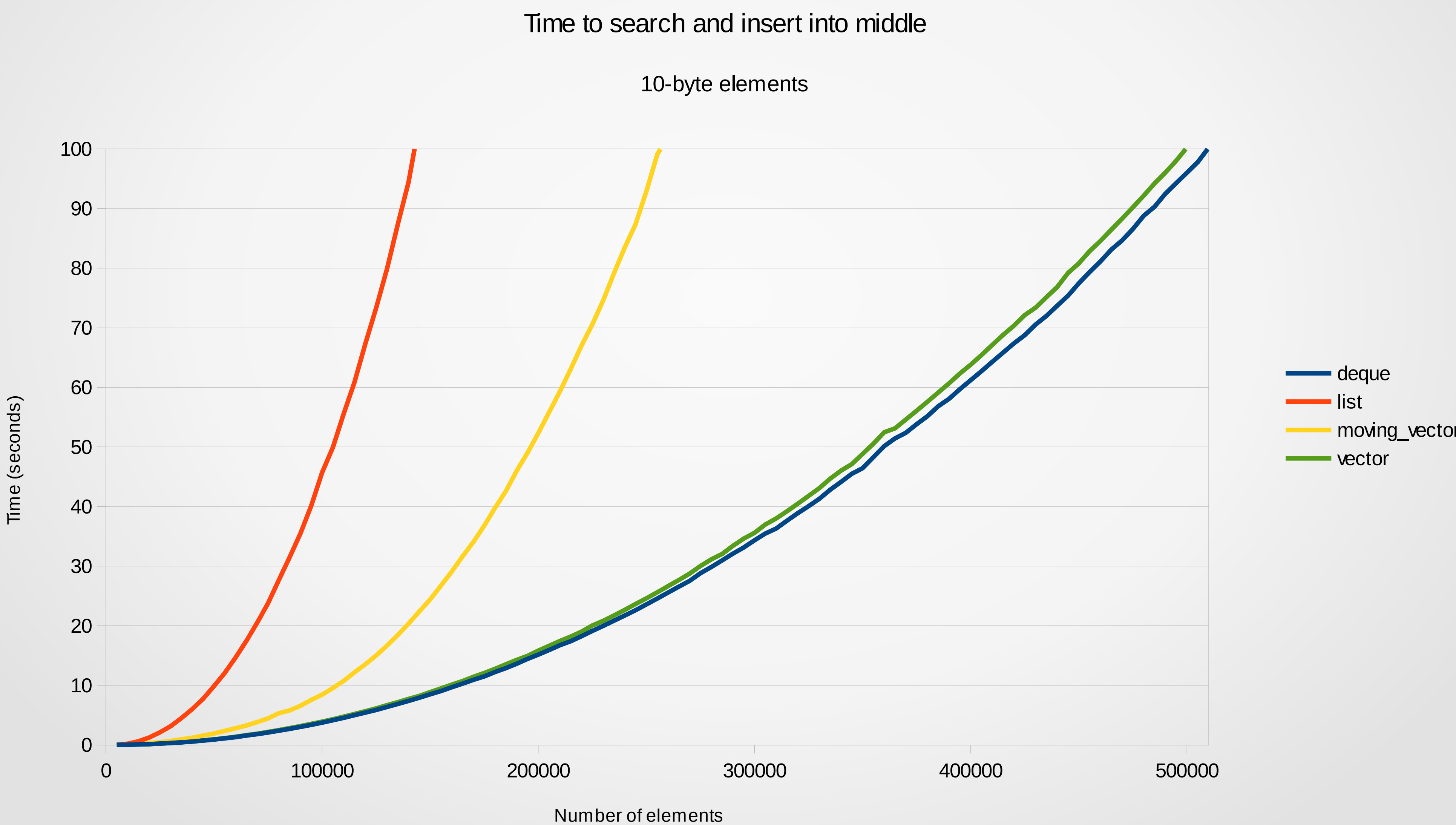
# Source of data

- https://bitbucket.org/davidstone/containers
- Compiled with g++ 4.8.2
  - Ofast
  - march=native
  - fipa-pta
  - funsafe-loop-optimizations
  - flto

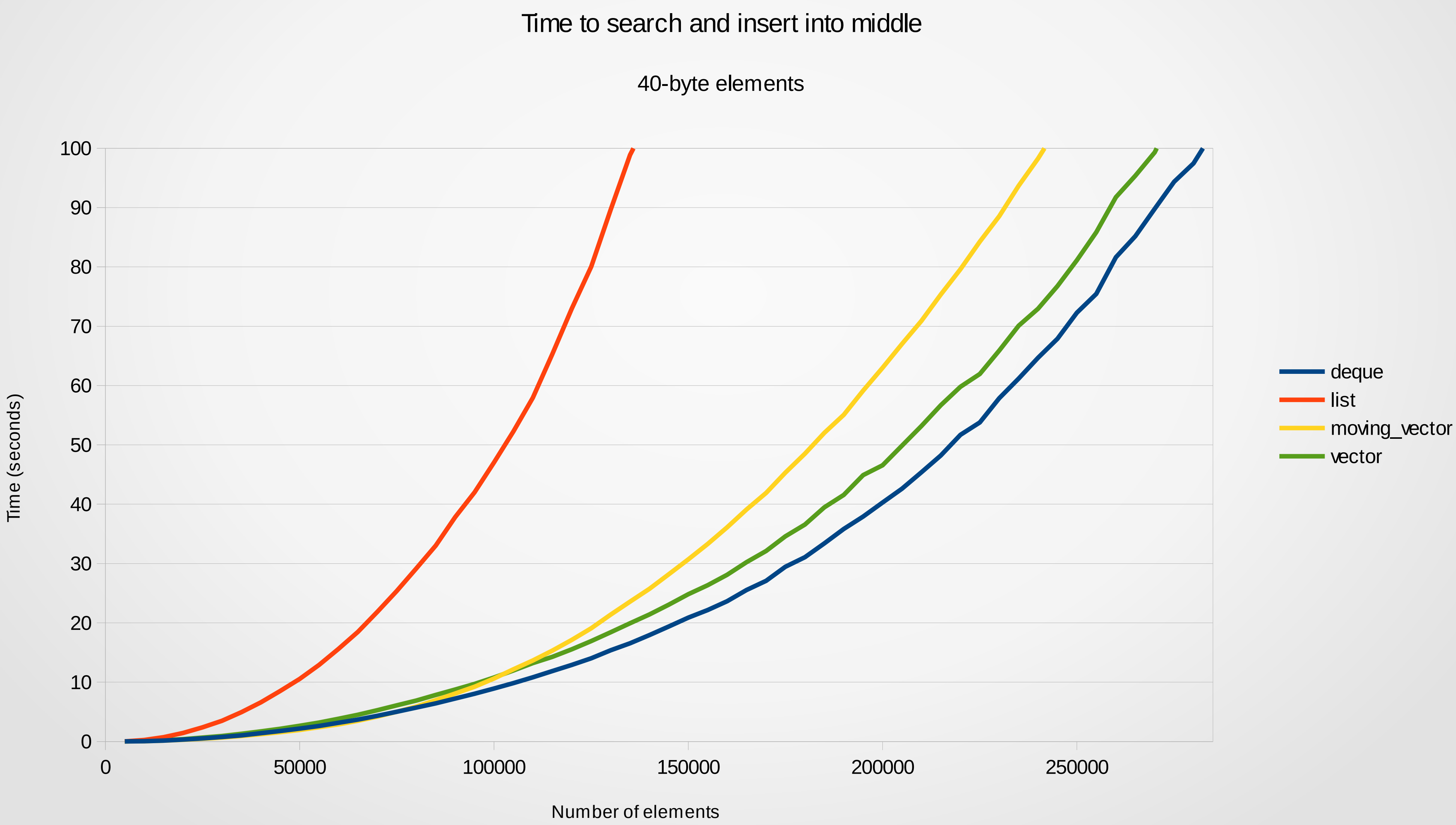
# Cache-friendliness, sequence



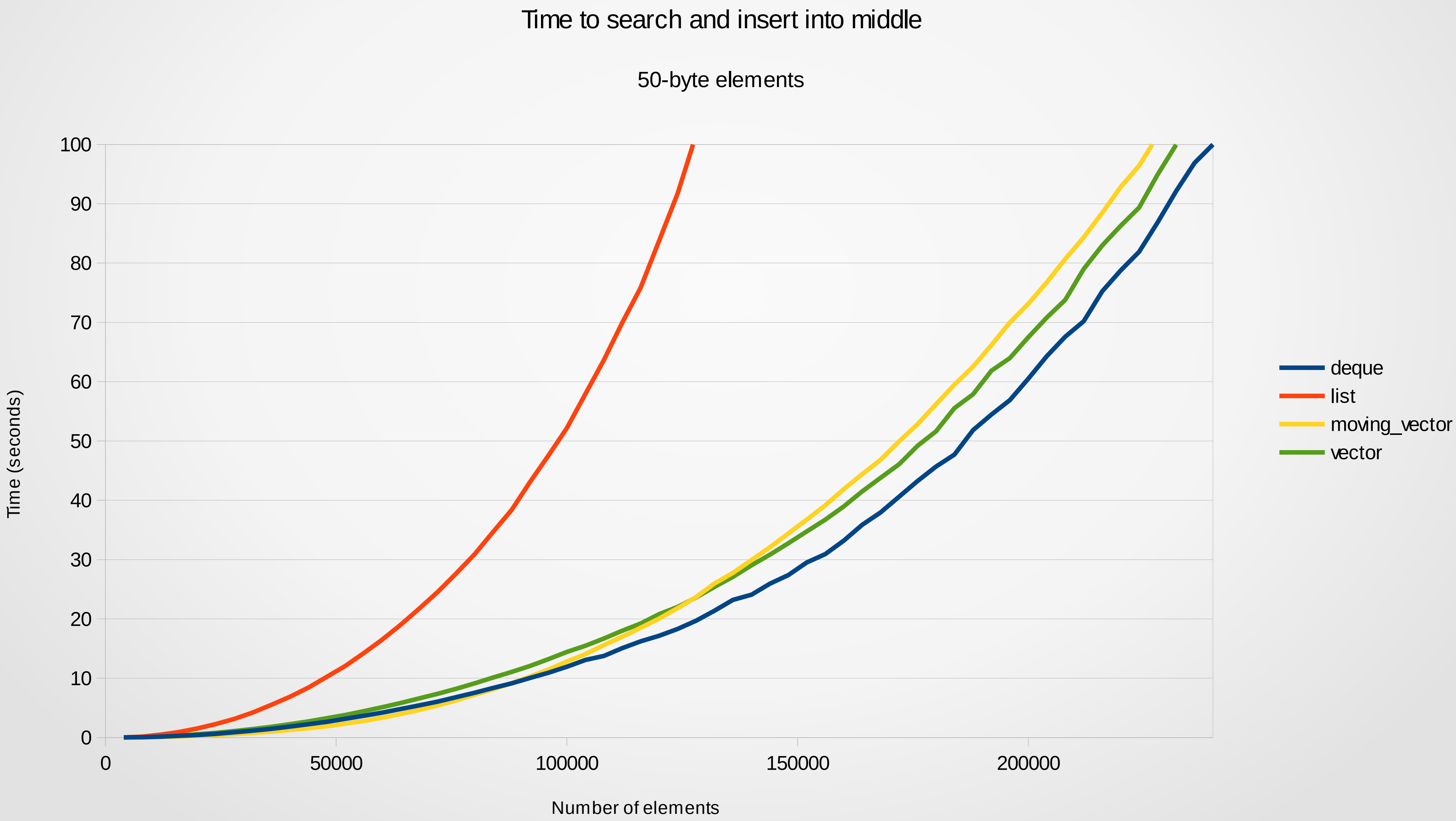
# Cache-friendliness, sequence



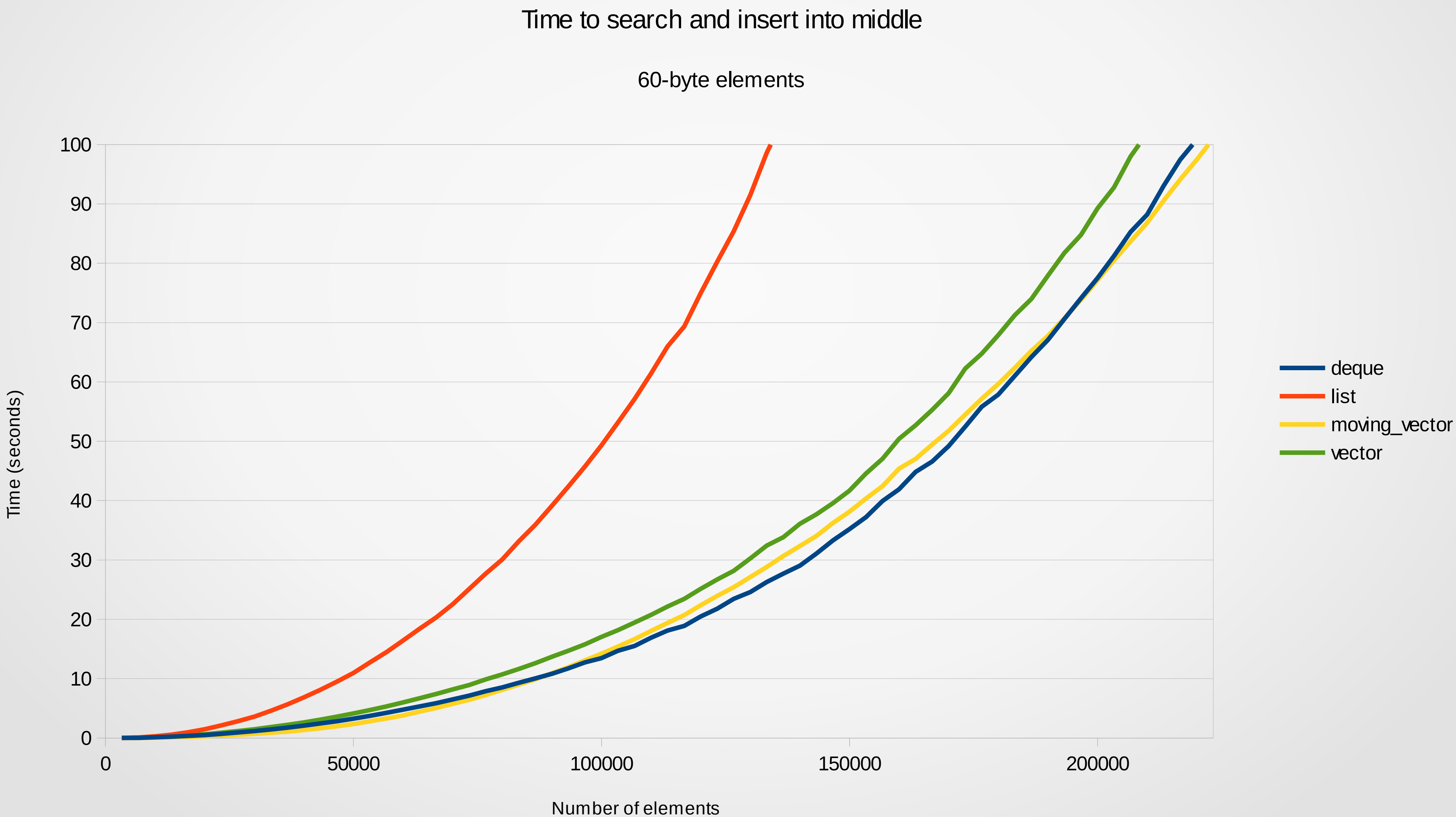
# Cache-friendliness, sequence



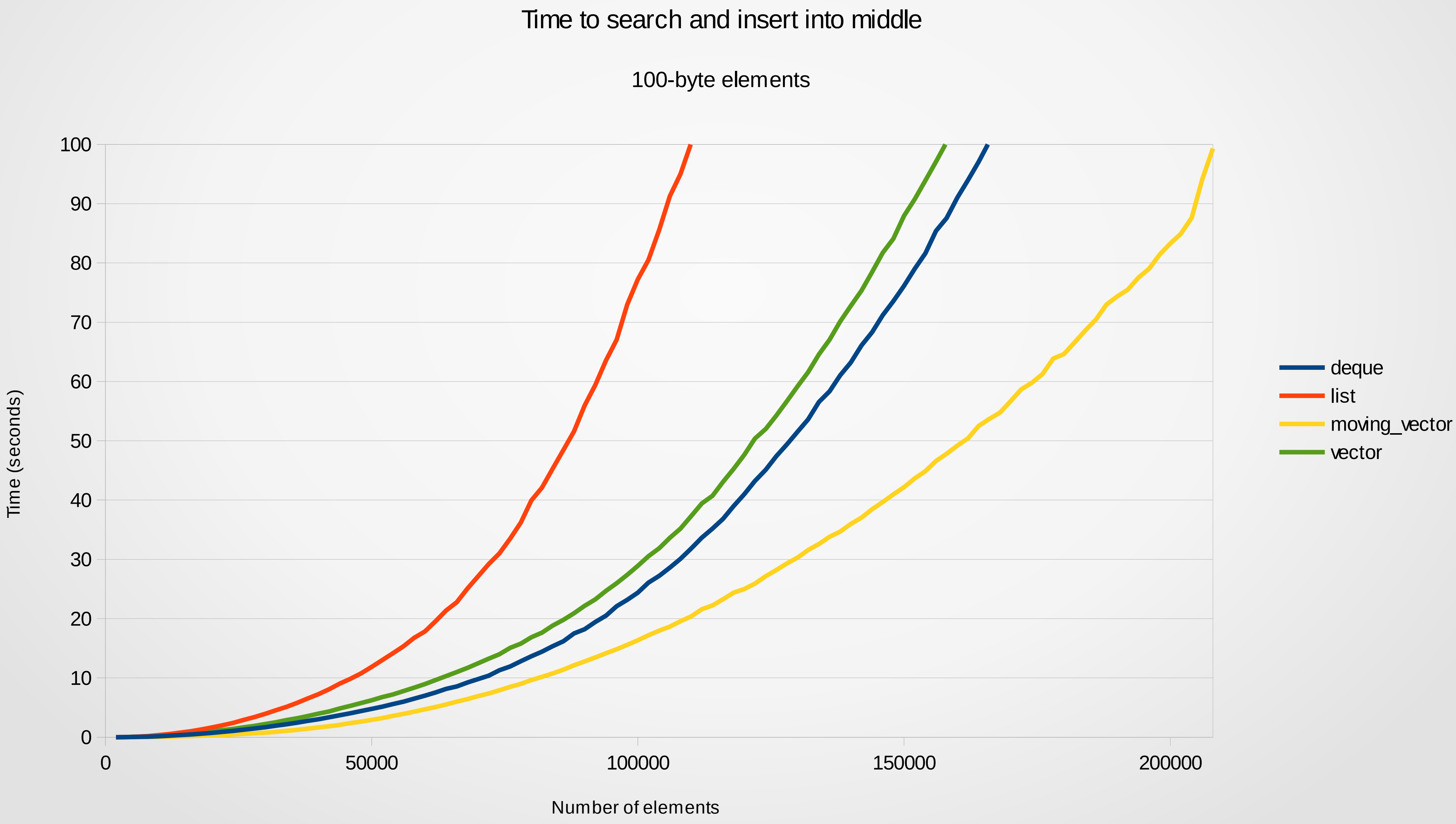
# Cache-friendliness, sequence



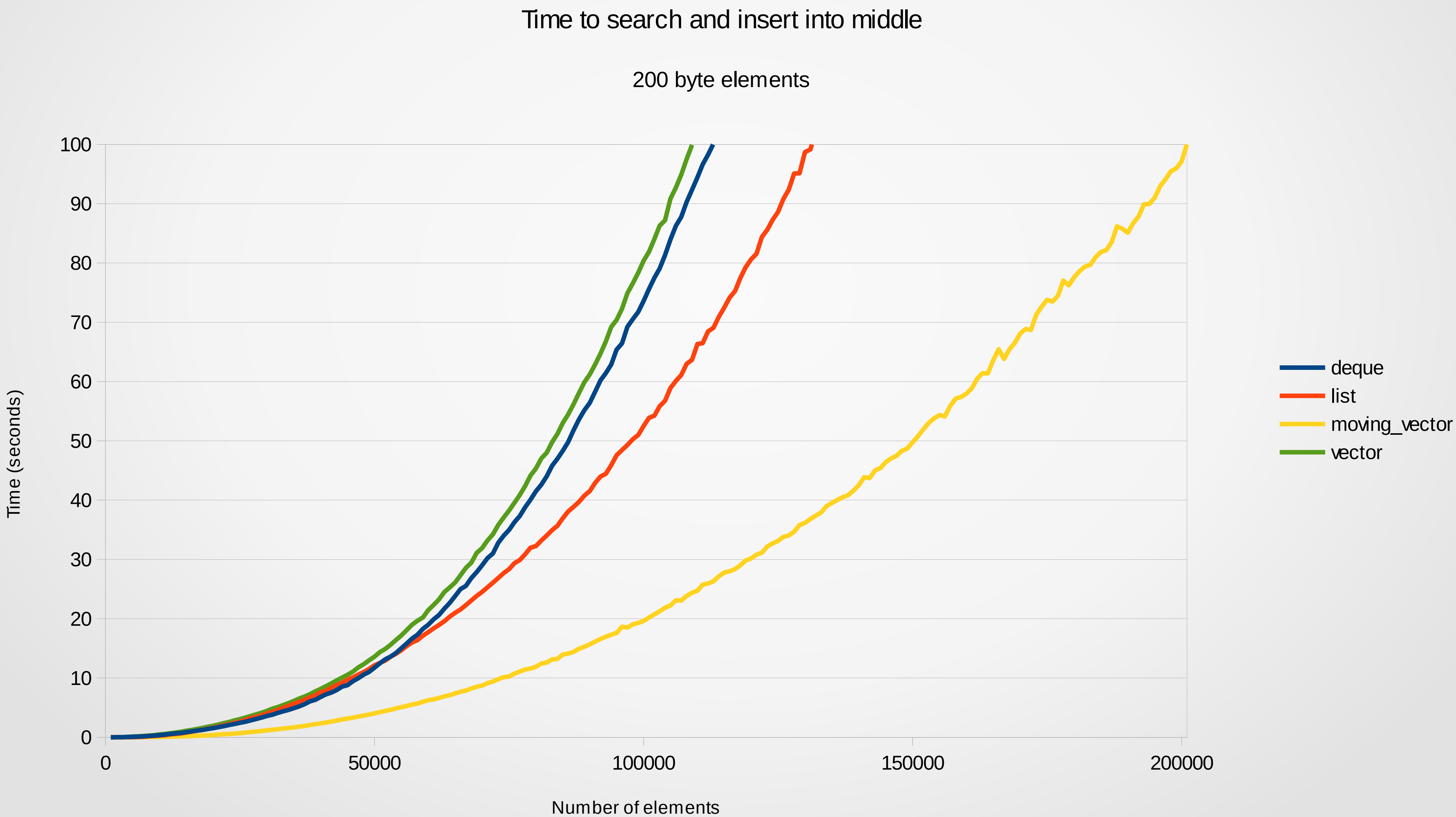
# Cache-friendliness, sequence



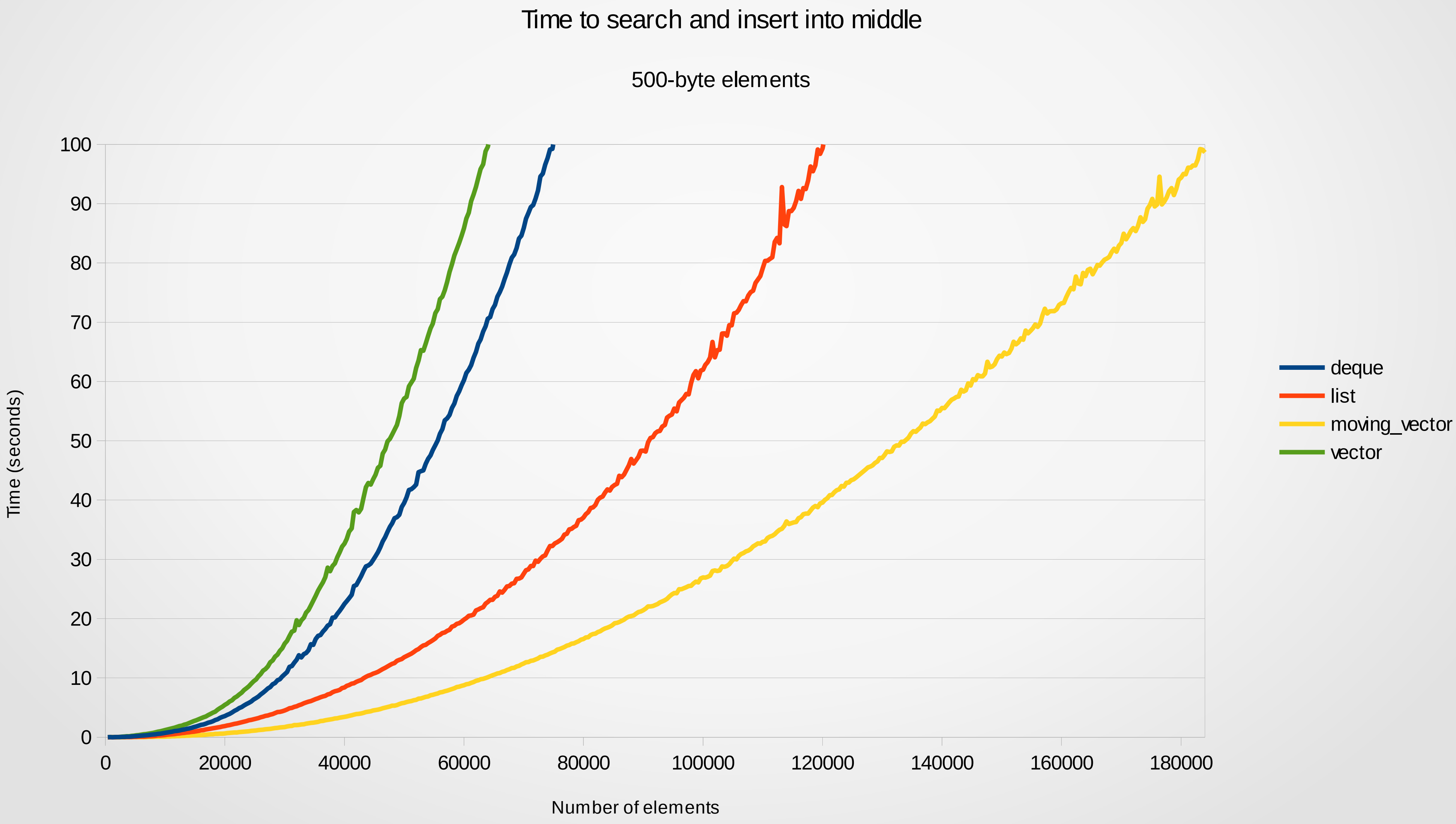
# Cache-friendliness, sequence



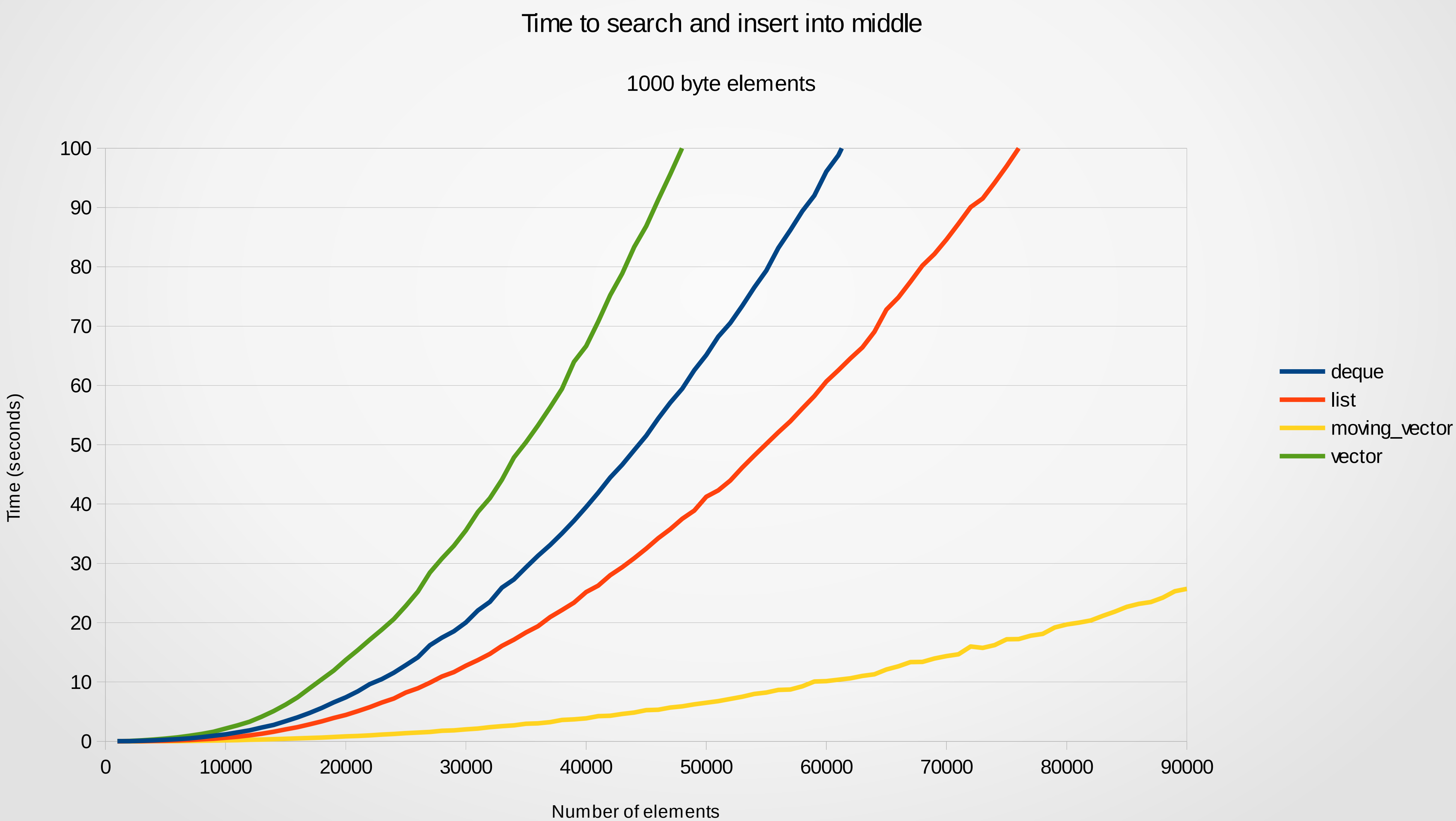
# Cache-friendliness, sequence



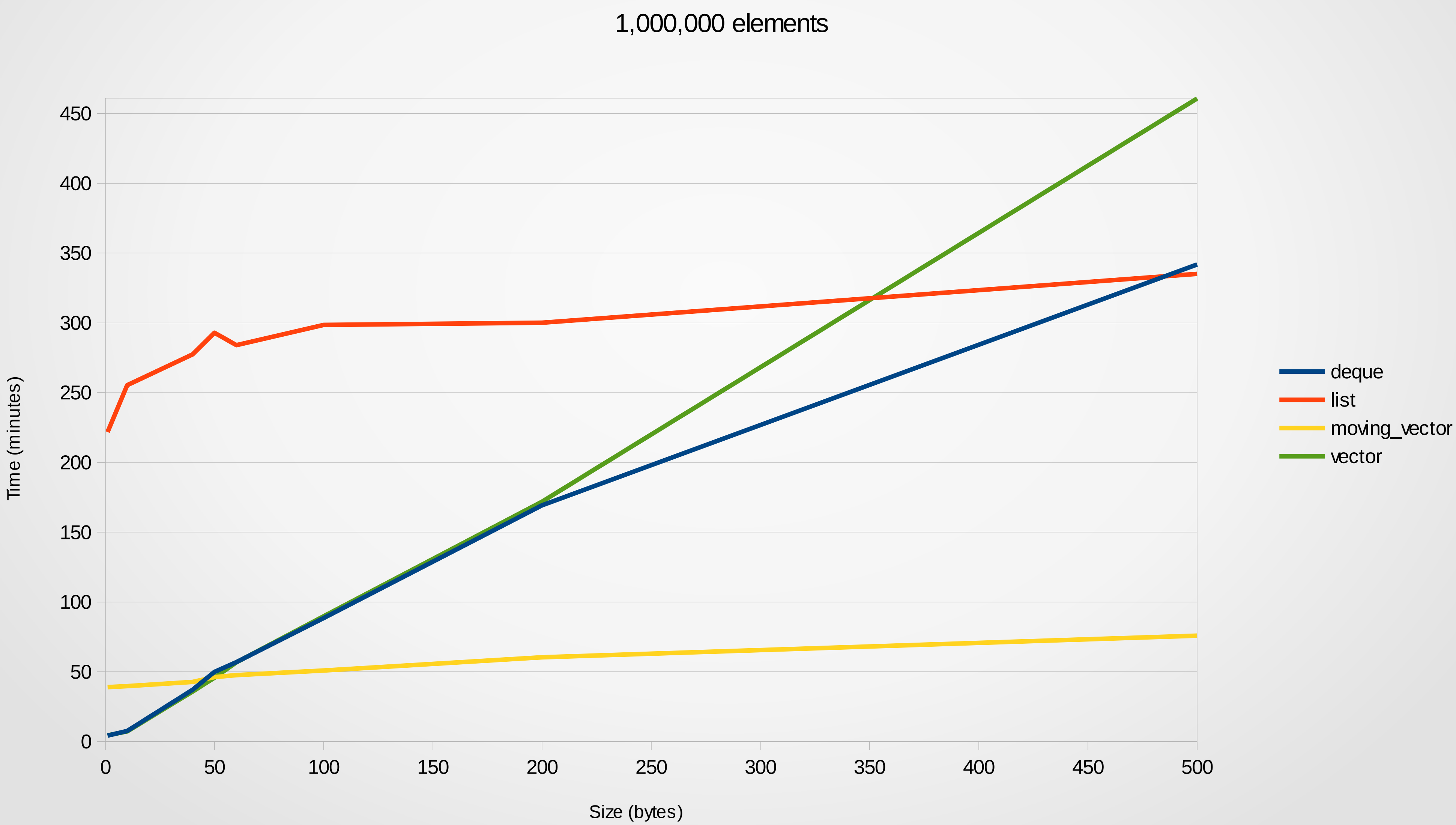
# Cache-friendliness, sequence



# Cache-friendliness, sequence



# Cache-friendliness, sequence

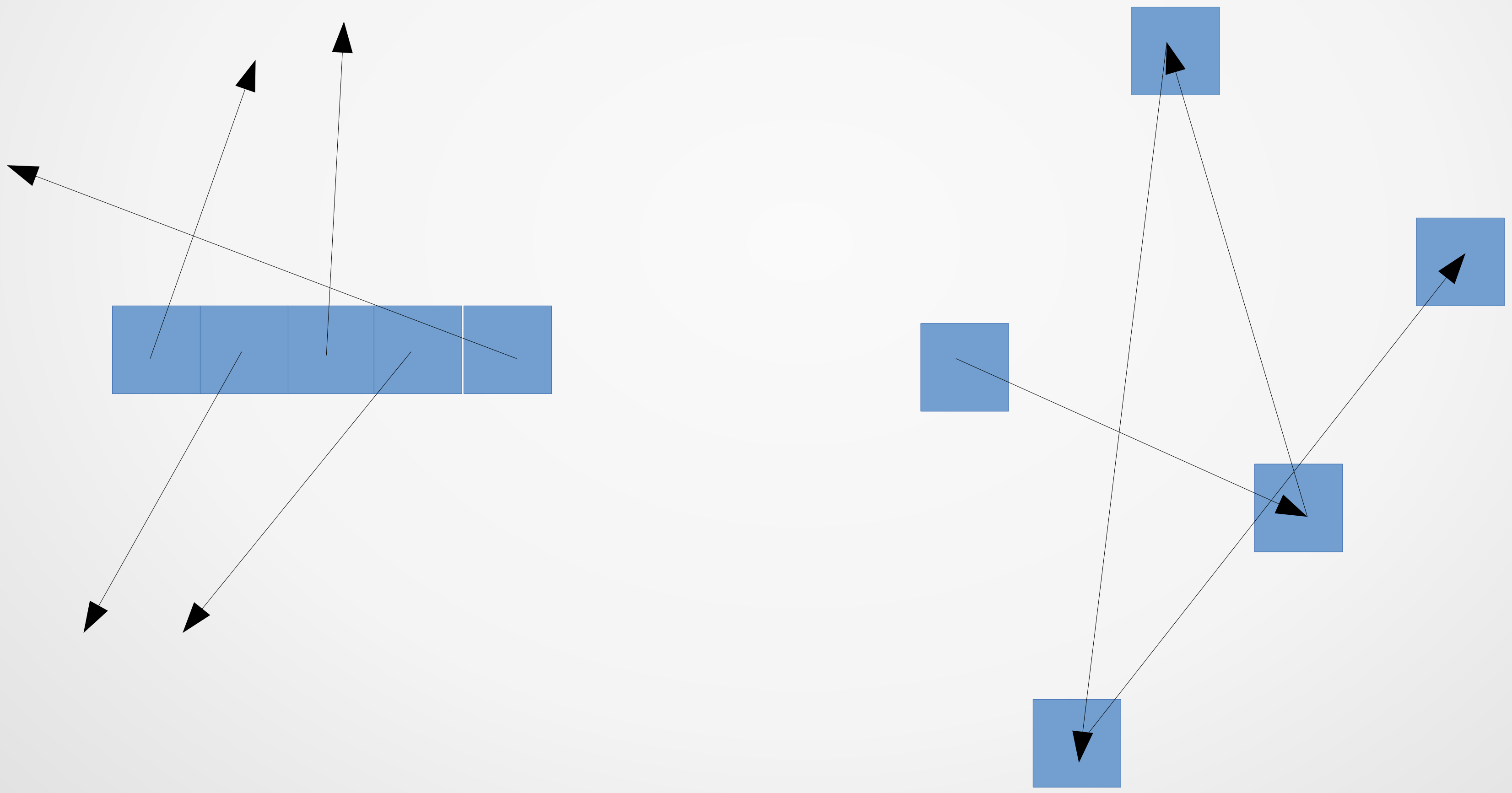


# Explanation of results

- Locality of reference
  - Explains plain vector's performance

# Explanation of results

- Pipelining



# std::list

- If you are iterating, do not use std::list
- If you are not iterating, do not use std::list

# Cache-friendliness, associative

- std::map<T>
- sorted std::vector<T>
  - unstable_flat_map
- sorted std::vector<std::unique_ptr<T>>
  - stable_flat_map

# Cache-friendliness, associative

No performance graphs.

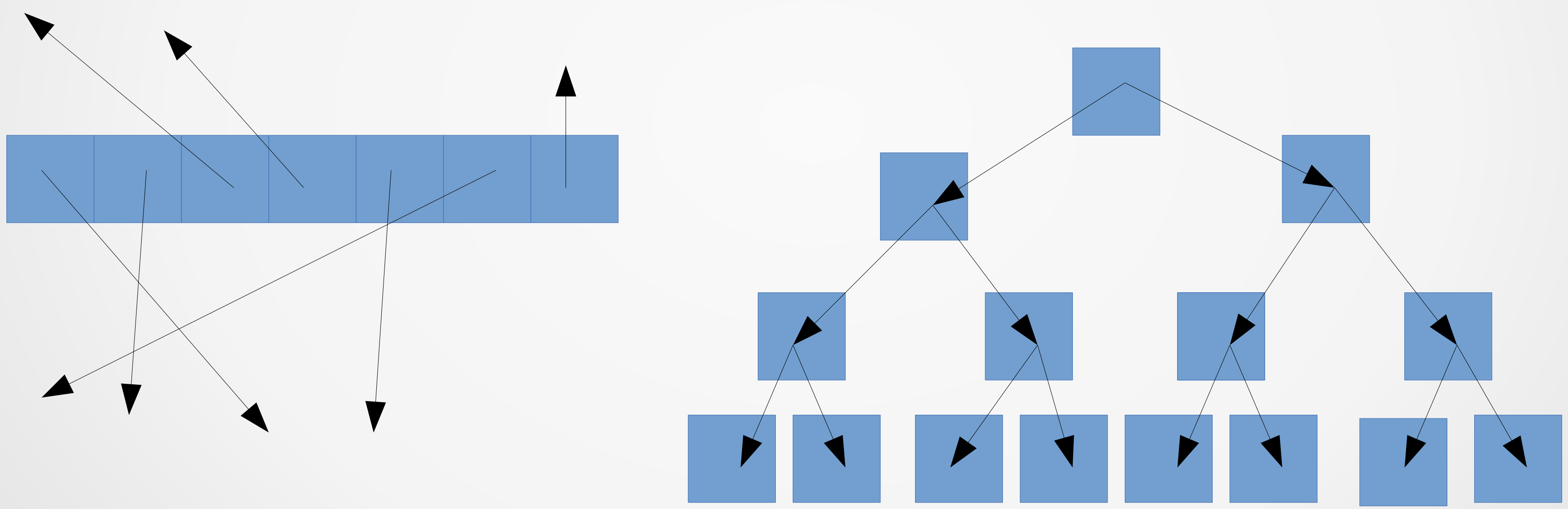
=(

# Cache-friendliness, associative

- unstable_flat_map provides the fastest lookup
- std::map provides the fastest insertion
- For small element sizes, unstable_flat_map is faster for everything else (including batch insertion)
  - Small is around 100 bytes

# Cache-friendliness, associative

- stable_flat_map was never the best choice
- Why does it perform so much worse at associative tasks?



# Optional values

- What would a function look like that needs to output an optional value?
  - What does it look like if it just needs to return "by reference"
- Which interface to use?
  - Which is easier?
  - Which is faster?

# Optional values

- Typically, use something like the proposed std::optional
  - https://github.com/akrzemi1/Optional
- std::optional<T> is always at least as large as T
- Null std::unique_ptr<T> is sizeof(T *)

# Do not be afraid of "by value"

- The compiler will elide copies
  - Return value optimization
- Most moves are cheap

# Returning a dynamically allocated value

- auto u() -> std::unique_ptr<T>;
- auto s() -> std::shared_ptr<T>;

# Constructing a shared_ptr

- std::shared_ptr<T>(new T());
- std::make_shared<T>();

- The importance of std::make_shared
- The difference between libraries and applications

# Value semantics

- std::shared_ptr is last on the hierachy
- indirection is powerful
  - With great power comes great difficulty in comprehension
- std::shared_ptr<T const> is not as bad

# Scope-Bound Resource Management

- Destructor is a fundamental part of C++
- Structured code automatically creates nested life times
  - Functions, not goto everywhere
- Do not use std::shared_ptr to avoid structuring code
- std::shared_ptr is used with multi-threading
  - When you are modeling something that must hold onto data, but the duration is dependent on run-time factors

# Raw Pointers

- Raw pointers never own memory
- Smart pointers do not deprecate raw pointers
- Raw pointers reference memory
- A pointer is an optional reference

# Compilation firewall

```
#include <lots_of_headers>
class Class {
public:
    functions();
private:
    Thing1 m_1;
    Thing2 m_2;
    ...
    ThingN m_n;
};
```

# Compilation firewall

```
class Class {
public:
    functions();
private:
    class Impl;
    std::unique_ptr<Impl>
};
```

# Compilation firewall

- Minimize compile times
- Minimize recompilation
- Stable ABI
- If used with care, can work around binary compatibility problems in third-party libraries

# Summary

- Use a smart pointer only for:
  - Dynamic polymorphism when you cannot achieve this without dynamic allocation
  - Minimizing compilation dependencies (PImpl)
  - Optional values that could be very large
  - Enabling moves on non-movable types
  - Optimizing moves on slow-to-move types
- In all other cases, use an automatic variable
- Prefer cache-friendly data structures