glass: ordered set data structure for client-side order books

Viktor Krapivensky
AKB System

April 20, 2025

Abstract

The "ordered set" abstract data type with operations insert, erase, find, min, max, next and prev is ubiquitous in computer science. It is usually implemented with red-black trees, B-trees, or B^+-trees. We present our implementation of ordered set based on a trie. It only supports integer keys (as opposed to keys of any strict weakly ordered type) and is optimized for market data, namely for what we call *sequential locality*. The following is the list of what we believe to be novelties:

- *Cached path* to exploit *sequential locality*, and fast truncation thereof on erase operation;
- A hash table (or, rather, a *cache table*) with hard $O(1)$ time guarantees on any operation to speed up key lookup (up to a pre-leaf node);
- Hardware-accelerated "find next/previous set bit" operations with **BMI2** instruction set extension on x86-64;
- Order book-specific features: the *preemption principle* and the *tree restructure* operation that prevent the tree from consuming too much memory.

We achieve the following speedups over C++'s standard `std::map` container: **6x**—**20x** on modifying operations, **30x** on lookup operations, **9x**—**15x** on real market data, and a more modest **2x**—**3x** speedup on iteration. In this paper, we discuss our implementation.

1 Notation

By $\mathbb{N}$ we mean $\{0, 1, 2, \ldots\}$.

By $a \bmod b$, where $a \in \mathbb{Z}, b \in \mathbb{N}, b \neq 0$, we mean the integer r such that $0 \leq r < b$ and $r \equiv a \pmod{b}$.

By $\mathbb{Z}_m$ we mean the ring of integers modulo m. We identify elements of $\mathbb{Z}_m$ with elements of $\mathbb{N}$ as follows:

- we identify $x \in \mathbb{Z}_m$ with the minimal $n \in \mathbb{N}$ such that $\underbrace{x = \hat{1} + \hat{1} + \cdots + \hat{1}}_{n \text{ times}}$;
- we identify $n \in \mathbb{N}$ with the unique $x \in \mathbb{Z}_m$ such that $\underbrace{x = \hat{1} + \hat{1} + \cdots + \hat{1}}_{n \text{ times}}$.

Above, $\hat{1}$ means the multiplicative identity of $\mathbb{Z}_m$.

2 Introduction

The "ordered set" is an abstract data type which maintains a set ξ of elements, which is initially empty, and has at least the following operations implemented:

- $\underline{\text{insert}}(\langle k, v \rangle)$: assigns $\xi \longleftarrow \xi \cup \{\langle k, v \rangle\}$ if $\nexists \langle k', v' \rangle \in \xi : k' = k$;
- $\underline{\text{erase}}(\langle k, v \rangle)$: assigns $\xi \longleftarrow \xi \setminus \{\langle k, v \rangle\}$;
- $\underline{\text{find}}(k)$: if $\exists \langle k', v \rangle \in \xi : k' = k$, returns v; otherwise, returns the special blank symbol "#";
- $\underline{\text{min}}()$ and $\underline{\text{max}}()$: return $\widetilde{\min} \{k \mid \langle k, v \rangle \in \xi\}$ and $\widetilde{\max} \{k \mid \langle k, v \rangle \in \xi\}$, correspondingly;
- $\underline{\text{next}}(k)$: returns $\widetilde{\min} \{k' \mid \langle k', v \rangle \in \xi, k' > k\}$;
- $\underline{\text{prev}}(k)$: returns $\widetilde{\max} \{k' \mid \langle k', v \rangle \in \xi, k' < k\}$.

In the list above, $\widetilde{\min}$ and $\widetilde{\max}$ specify regular min and max set operations except that they return the special blank symbol "#" if the argument is empty.

Typically all operations are $O(\log n)$. Additionally, $\underline{\text{min}}$ and $\underline{\text{min}}$ may be cached and thus execute in $O(1)$.

This abstract data type is ubiquitous in computer science. Examples of where it is used include databases, file systems, and schedulers and epoll file descriptors in the Linux kernel [11].

The C++ programming language has standard containers `std::set` and `std::map`, which are typically implemented via red-black trees [9]. The Java

programming language provides `TreeSet` and `TreeMap` collections (under `java.utils` package), which are also implemented via red-black trees [12] [13]. The Rust programming language provides `BTreeSet` and `BTreeMap` collections (under `std::collections` module) [4] [3], which are based on B-trees.

There are alternatives to red-black trees and variations of B-trees for implementation of this abstract data type, e.g. AVL-trees and trees exploiting the structure of the keys (for example, tries and radix trees for integers and strings, van Emde Boas and fusion trees for integers).

3 Ordered set applied to client-side order book management

For client-side order book management, we maintain an ordered set with keys being prices and values being non-zero amounts. An pair of price and amount is called a *price level*. We need to be able to handle the following queries:

- $\underline{\text{adjust}}(\pi, \Delta), \Delta \neq 0$: add Δ to the previous amount at price π (if there is no previous amount at this price, set the new amount to Δ). If the new amount is zero, delete this price level. Δ can be negative, but it is guaranteed that no price level has negative amount;
- $\underline{\text{min}}()$ and $\underline{\text{max}}()$: get minimum or maximum price of a level (or the special blank symbol "#" if the order book is empty);
- $\underline{\text{next}}(\pi)$ and $\underline{\text{prev}}(\pi)$: get next or previous (after or before π) price of a level (or the special blank symbol "#" if there is none).

$\underline{\text{min}}(), \underline{\text{max}}(), \underline{\text{next}}(\pi)$ and $\underline{\text{prev}}(\pi)$ can trivially be expressed in terms of same-name ordered set operations. The $\underline{\text{adjust}}(\pi, \Delta)$ operation can be expressed in terms of ordered set abstract data type as follows:

Procedure adjust(π, Δ)

Input: Price π, signed amount Δ such that $\Delta \neq 0$.

begin

  $A \longleftarrow \underline{\text{find}}(\pi)$

  if $A = \#$ **then**

    $a \longleftarrow 0$

  else

    $a \longleftarrow A$

    $\underline{\text{erase}}(\langle \pi, a \rangle)$

  end

  $a' \longleftarrow a + \Delta$

  if $a' \neq 0$ **then**

    $\underline{\text{insert}}(\langle \pi, a' \rangle)$

  end

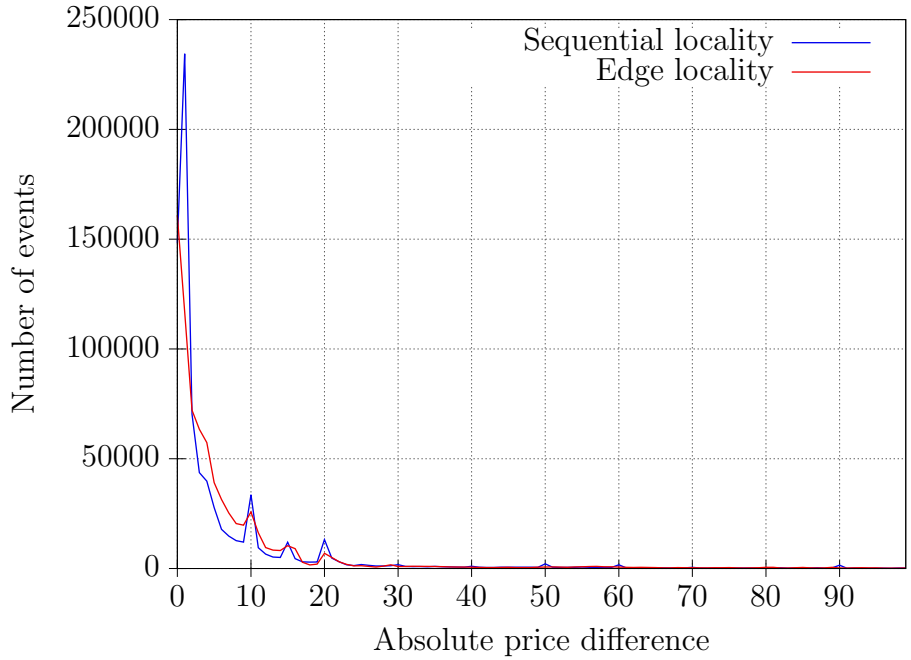
end

3.1 Sequential locality and edge locality in market data

We define *sequential locality* as the closeness of the price of an event to the price of previous event; In order words, the smaller $|\pi_i - \pi_{i-1}|$, the greater is the sequential locality is. We also define *edge locality* as the closeness of the price of an event to the best price; In order words, the smaller $|\pi_i - \pi_{\text{best}}|$, the greater the edge locality is.

We recorded market data on MOEX (Moscow Exchange) for instrument **CRM5** (futures contract for CNY/RUB) during the main trading session of May 20, 2025. We then processed the recorded market data to visualize both sequential locality and edge locality:

Figure 1: Sequential locality and edge locality in market data



We see that market data exhibits both strong sequential locality and strong edge locality. Note also the peaks at what humans percept as "nice round numbers", e.g. 10, 15, 20, 50.

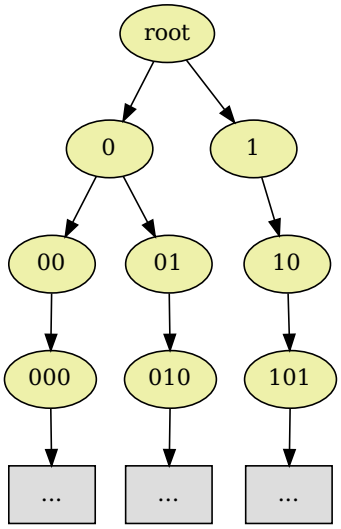
What this means is that we can cache the path to the previously accessed key to exploit sequential locality. We could cache the path to the "best" (minimal or maximal, depending on the side of the order book) key to exploit edge locality, but this would be slower because, after deletion of the best key there is no quick way to locate the next best one. Caching both would also result in suboptimal performance because we would need to maintain two cached paths instead of one.

4 Baseline implementation

4.1 The trie

Our implementation is based on an uncompressed trie. We fix K, the number of bits in a key. We also fix N, the maximum number of children of a node. N must be a power of two. We define $C = \log_2 N$. If C does not divide K, we pretend the higher bits of the key are zero. The tree, unless empty, has the unique root node.

Figure 2: A trie



Above, $K = 4$, $C = 1$, $N = 2$. We call the grey "..." nodes "post-leafs", and nodes one level higher (with three-digit labels) "pre-leafs".

Any node contains a N-bit mask indicating which children are present; array of N index-pointers to children nodes, and an index-pointer to the parent node (with a special value for "no parent" reserved for the root node). An iterator contains the index of a pre-leaf node and the key, the C least significant bits of which specify the index of a child of the post-leaf node.

We define $\mathcal{L} = \lceil K/C \rceil$, the distance from the root of the tree to a post-leaf.

4.2 Index-pointers, slot allocator and multiple-node allocation

Each index-pointer is an index into the array of nodes that a *glass* maintains. Throughout this section, this array is referred to as $\mathcal{N}$.

The main reason for the introduction of index-pointers is cache locality: if the trie can never grow to 2^{32} (sans two special values for "end/not found" and "bad") nodes, it is beneficial to only store 32-bit indices (even considering that this introduces another level of indirection). The same applies to 16-bit indices.

The data structure keeps track of free nodes via the *slot allocator* principle:

- free nodes are organized as a singly-linked list: a *glass* maintains the "first free node" index-pointer, and one of the fields in a free node is used as the "next free node" index-pointer;
- allocating a node consists of consulting the "first free node" index-pointer; say its value is p. If p is the special "invalid" value, grow the nodes array and load p again. Read the "next free node" field of $\mathcal{N}[p]$, call that value q. Assign q to the "first free node" index-pointer, return p as the index of the allocated node. Note that, unless the array is grown, this is an O(1) operation. Note also that array growth becomes exponentially less likely as the size of a *glass* grows; if full pre-allocation is used, the array never grows at all;
- de-allocating a node with index p consists of writing the value of the current "first free node" index-pointer into the "next free node" field of $\mathcal{N}[p]$, then replacing the current "first free node" index-pointer by p. Note this is an O(1) operation.

See the listings below for allocate-node and deallocate-node functions ("#" denotes the special "invalid" value of an index-pointer).

Procedure allocate-node

```
Data: 𝒩, the array of nodes. 𝔣, the "first-free-node" index-pointer.
Output: The index-pointer to the allocated node.
begin
    if 𝔣 = # then
        grow-array()
    end
    p ⟵ 𝔣
    𝔣 ⟵ 𝒩[p].next_free_node
    return p
end
```

Procedure deallocate-node(p)

```
Data: 𝒩, the array of nodes. 𝔣, the "first-free-node" index-pointer.
Input: p, the index-pointer to the node to de-allocate.
begin
    𝒩[p].next_free_node ⟵ 𝔣
    𝔣 ⟵ p
end
```

The main drawback of such a scheme is that we never "give back" memory that once has been used but currently is not, to the system. But this is widely considered a good trade-off: many user-space memory allocators, such as the one found in glibc [5], do not give back memory either. Memory management in some dynamic programming languages, such as Lua [14] (PUC Lua, i.e. the official implementation at `https://lua.org/`), uses the default libc's allocator as a base for implementing more complex allocation schemes.

We currently do not suffer from this drawback because we use full pre-allocation.

We also have a procedure that allocates many nodes at once, without saving and restoring the "first free node" index-pointer each time. It falls back to the standard approach of allocating one-by-one if there are too few nodes available.

4.3 Searching for next or previous set bit in mask

In the implementations of next and prev operations, we need to find the index of the next/previous bit set in a mask, starting from the specified bit index i (not including i) and returning a special "invalid" value if there is no such bit.

This can be done via zeroing lower/higher bits (up to, and including, the bit indexed i) and invoking the find-first-set/find-last-set instruction on the result. So the problem can be reduced to the question of how to efficiently zero lower of higher bits up to the specified index, inclusively.

For zeroing $(i+1)$ lowest bits, the "obvious" solution of shifting right and then left by $(i+1)$ is incorrect: in C/C++, shifting by bit width is undefined behavior, and non-SIMD shift instructions of x86-64 ISA treat the shift count modulo W for bit width W, so, for example, shifting a 64-bit value by 64 is equivalent to shifting by 0 (that is, no-op) instead of zeroing out the value. We can mask (perform bitwise AND) the value with $((-1) \ll i) \ll 1$, or, equivalently, with $(-2) \ll i$, which has one operation less. Above, negative numbers are modulo 2^W, where W is the bit width: (-1) denotes all-ones value, (-2) denotes $(W-1)$ leading ones and then one zero bit.

For zeroing higher bits starting with, and including, bit index i, we can use the **bzhi** instruction from **BMI2** [7] [8] [2] instruction set extension on x86-64. If **bzhi** cannot be used, we can mask (perform bitwise AND) the value with $(1 \ll i) - 1$.

4.4 Calculating an upper bound on capacity needed for a size and maximum size for a given capacity

We now want to obtain an upper bound on the maximum number of nodes required for a size $\mathfrak{S}$ of a tree.

Define

$$r = \begin{cases} K \bmod C, & K \bmod C \neq 0; \\ C, & K \bmod C = 0, \end{cases}$$

the number of bits actually used to discriminate between different children of the root node.

There are two crucial facts: no level can have more than $\mathfrak{S}$ nodes, and no level can "give birth" to more than 2^C nodes (2^r for the root node).

Then, the upper bound on the number of the maximum number of nodes at level i can be calculated as

$$\mathcal{M}_i = \begin{cases} \min\{\mathfrak{S},\, 1\}, & i = 0; \\ \min\{\mathfrak{S},\, 2^r\}, & i = 1; \\ \min\{\mathfrak{S},\, 2^C \cdot \mathcal{M}_{i-1}\}, & i \geq 2. \end{cases}$$

The upper bound on the maximum number of nodes can then be calculated as

$$\sum_{i=0}^{\mathcal{L}-1} \mathcal{M}_i.$$

To perform the inverse computation, that is, to calculate the number of elements that will definitely fit into a tree with a given capacity, we can use binary search using the formulae above.

4.5 Preemption principle and tree restructuring

We can now calculate the maximum tree size if all index-pointers are 16-bit or 32-bit. We assign $C = 5$, $K = 50$ (on MOEX, prices have 14 decimal digits; we allocate an extra higher bit for sign). The size of node is 48 bytes for 16-bit index-pointers, 80 bytes for 32-bit index-pointers.

16-bit index-pointers	
Tree size	Memory consumption
$9 \cdot 10^2$	339.05 Kb
$9 \cdot 10^3$	2.93 Mb
$9 \cdot 10^4$	N/A
$9 \cdot 10^5$	N/A

32-bit index-pointers	
Tree size	Memory consumption
$9 \cdot 10^2$	565.08 Kb
$9 \cdot 10^3$	4.89 Mb
$9 \cdot 10^4$	43.78 Mb
$9 \cdot 10^5$	414.57 Mb

"N/A" means that the resulting capacity is more than index-pointers of such size are able to address.

It is not unusual for order books to contain more than 10^6 entries. We estimate the number of instruments we want to receive market data from to be around 100. Say we have 8 Gb of memory to spend on the order books. Note that we are going to need two trees per instrument: one for asks and one for bids. It is easy to see that the maximum number of entries in a tree that we can afford lies in $[9 \cdot 10^3; 9 \cdot 10^4]$.

It may seem that we cannot use a trie for managing order books because it needs to contain much more elements than we can afford. But note that:

- in practice, we only need to iterate over no more than 25 best prices;
- the situation when the best price goes through more than $9 \cdot 10^3$ non-empty levels (either up or down) in a single trading session is extremely unlikely.

So we propose the following solution to handle to handle this: on insert, if the resulting size of the *glass* would be greater that the maximum size, preempt the new level into a hash table. For min/max and next/prev, if the result cannot be found in the *glass* (assuming the restrictions above, we can prove that the size of the *glass* in this case is strictly less than the maximum size), run the costly procedure of flushing the entries from the hash table back to the tree.

More specifically:

- define number $\mathcal{S}$ as the maximum size of the *glass* (around $9 \cdot 10^3$ in our case);
- define, for a min-*glass* (where the best price is the minimal one),

$$\widehat{\infty} = +\infty,$$
$$\pi_1 \lhd \pi_2 = \pi_1 < \pi_2;$$

  for a max-*glass* (where the best price is the maximal one),

$$\widehat{\infty} = -\infty,$$
$$\pi_1 \lhd \pi_2 = \pi_1 > \pi_2.$$

  In other words, $\widehat{\infty}$ means price that is worse than any "real" price that a *glass* may contain, and $\pi_1 \lhd \pi_2$ means that price π_1 is better than π_2.

- maintain a hash table that maps "preempted" prices to amounts, initially empty;
- maintain a number called "preemption threshold price", denoted as π_{thres}, initially $\widehat{\infty}$;
- during insertion with price π:
  - ⋆ if $\pi \lhd \pi_{\text{thres}}$, insert as usual, except that if the insertion would "overflow" the *glass* (the size would be greater than $\mathcal{S}$), do not insert it into the *glass*, but instead insert into the hash table and assign $\pi_{\text{thres}} \longleftarrow \pi$.
  - ⋆ otherwise, insert into the hash table;

  We call the operation of inserting into the hash table instead of the *glass* itself a "preemption".
- during find and erase with price π: if $\pi \lhd \pi_{\text{thres}}$, perform the operation on the *glass*; otherwise, perform it on the hash table.
- as for the min/max and next/prev operations: in this setting, we only support min and next for min-*glass*, and max and prev for max-*glass*. Even then, next/prev are only supported within best $\mathcal{B}$ prices, $\mathcal{B} < \mathcal{S}$. We define the notion of *exceptional situation* as the situation when:
  - ⋆ we need to handle operation min/max/next/prev, and the result of this operation on the *glass* itself (without elements preempted to the hash table) would be #, and;
  - ⋆ $\pi_{\text{thres}} \neq \widehat{\infty}$ (or, equivalently, the hash table is not empty).

  In *exceptional situation*, we need to perform a *tree restructure*, which consists of the following:
  - ⋆ calculate the number of entries to un-preempt from the hash table as $n_{\text{unpreempt}} = \mathcal{S} - \sigma$, where σ is the current size of the *glass*. Note that $n_{\text{unpreempt}}$ cannot be zero:
    - ◇ for min/max operations, *exceptional situation* means $\sigma = 0$;
    - ◇ for next/prev on price π, *exceptional situation* means that σ is the 0-based rank, counting from the best prices to the worst ones, of π in the *glass*. As we only support values of π within best $\mathcal{B}$ prices, $\sigma \leq \mathcal{B} < \mathcal{S}$;

- ⋆ select $n_{\text{unpreempt}}$ elements (or less if the size of the hash table is less) with best prices from the hash table. This can be done either via sorting in $O(n \log n)$, or via "partial sorting" in $O(n + k \log k)$, where n is the size of the hash table, $k = n_{\text{unpreempt}}$;
- ⋆ insert those elements to the *glass* and remove them from the hash table;
- ⋆ assign to π_{thres} the best price remaining in the hash table, or, if the hash table is now empty, $\widehat{\infty}$.

We need, then, to perform the operation that caused *exceptional situation* again: now, an *exceptional situation* cannot arise.

That's a lot of text, but it boils downs to pretty compact code:

Procedure ob-init

```
Data: π_thres, the preemption threshold price.
Γ, the glass data structure.
χ, the hash table.
begin
    π_thres ⟵ ∞̂
    glass-init(Γ)
    hash-table-init(χ)
end
```

Procedure ob-insert(π, a)

```
Data: π_thres, the preemption threshold price.
Γ, the glass data structure.
χ, the hash table.
begin
    if π ◁ π_thres then
        if glass-size(Γ) < glass-max-size(Γ) then
            glass-insert(Γ, π, a)
        else
            hash-table-insert(χ, π, a)
            π_thres ⟵ π
        end
    else
        hash-table-insert(χ, π, a)
    end
end
```

Procedure ob-erase(π)

Data: π_{thres}, the preemption threshold price.
Γ, the *glass* data structure.
χ, the hash table.
begin
  if $\pi \lhd \pi_{\text{thres}}$ **then**
    glass-erase(Γ, π)
  else
    hash-table-erase(χ, π)
  end
end

Procedure ob-find(π)

Data: π_{thres}, the preemption threshold price.
Γ, the *glass* data structure.
χ, the hash table.
begin
  if $\pi \lhd \pi_{\text{thres}}$ **then**
    glass-find(Γ, π)
  else
    hash-table-find(χ, π)
  end
end

Procedure ob-best

Data: π_{thres}, the preemption threshold price.
Γ, the *glass* data structure.
χ, the hash table.
begin
  if glass-size$(\Gamma) = 0$ **and** $\pi_{\text{thres}} \neq \widehat{\infty}$ **then**
    ob-restructure()
  end
  if *this is a min-orderbook* **then**
    return glass-min(Γ)
  else
    return glass-max(Γ)
  end
end

Procedure ob-next-best-after(π)

Data: π_{thres}, the preemption threshold price.
Γ, the *glass* data structure.
χ, the hash table.
begin
  if *this is a min-orderbook* **then**
    $f \longleftarrow$ glass-next
  else
    $f \longleftarrow$ glass-prev
  end
  $r \longleftarrow f(\Gamma)$
  if $r = \#$ **and** $\pi_{\text{thres}} \neq \widehat{\infty}$ **then**
    ob-restructure()
    $r \longleftarrow f(\Gamma)$
  end
  return r
end

Procedure ob-restructure

Data: π_{thres}, the preemption threshold price.
Γ, the *glass* data structure.
χ, the hash table.
begin
  $\sigma \longleftarrow \underline{\text{glass-size}}(\Gamma)$
  $S \longleftarrow \underline{\text{glass-max-size}}(\Gamma)$
  if $\sigma = S$ **then**
    error *"ob-next-best-after()" price too far from best*
  end
  $n_{\text{avail}} \longleftarrow S - \sigma$
  $B \longleftarrow \underline{\text{hash-table-best-n}}(\chi, \min\{n_{\text{avail}}, \underline{\text{hash-table-size}}(\chi)\})$
  foreach $\langle \pi, a \rangle \in B$ **do**
    $\underline{\text{glass-insert}}(\chi, \pi, a)$
    $\underline{\text{hash-table-erase}}(\chi, \pi)$
  end
  if $\underline{\text{hash-table-size}}(\chi) = 0$ **then**
    $\pi_{\text{thres}} \longleftarrow \widehat{\infty}$
  else
    $\langle \pi_0, a_0 \rangle \longleftarrow \underline{\text{hash-table-best-1}}(\chi)$
    $\pi_{\text{thres}} \longleftarrow \pi_0$
  end
end

4.6 Exact division of bit offset by C

In internal iterators, we represent the bit offset of a key as a signed number κ, with depth-δ iterator having

$$\kappa = C \cdot (\mathcal{L} - 1 - \delta).$$

Thus, an iterator referring to the root node has $\kappa = C \cdot (\mathcal{L} - 1)$, an iterator referring to a pre-leaf node has $\kappa = 0$, an iterator referring to a post-leaf node has $\kappa = -C$. This encoding helps us to traverse the tree: we can

- adjust the depth up or down by incrementing or decrementing κ by C;
- load the current chunk as $(\underline{\text{key}} \gg \kappa)\,\&\,M$, where $M = (1 \ll C) - 1$ is a compile-time constant;
- insert a chunk η into the current position via $\underline{\text{key}'} = \underline{\text{key}}\,|\,(\eta \ll \kappa)$;

and perform other similar actions.

Unfortunately, during insertion, we need to map κ back to the depth δ in order to calculate the number of nodes to allocate. If we make κ represent offsets in C-sized chunks, not in bits, then, in operations involving bit shifts, we would have to shift by $C \cdot \kappa$, not simply by κ, which is much slower. If we represent offsets via pairs $\langle \kappa, \kappa/C \rangle$, then we would have to perform arithmetic on two numbers instead of one when adjusting the depth up or down.

The formula for mapping κ back to depth δ is

$$\delta = \mathcal{L} - 1 - (\kappa/C).$$

Note that the division is always exact.

We can use approach from [6] to reduce this division to a shift and a multiplication modulo 2^W, where W is the bit width of the integer type in which κ is represented. Specifically, we decompose C into a product

$$C = 2^\ell \cdot \omega,$$

where $\ell, \omega \in \mathbb{N}$, ω is odd. Note that such a decomposition exists and is unique for integer $C > 0$: set ℓ to the exponent of the maximal (integer) power of two that divides C, set ω to $C/2^\ell$.

If a division by odd ω is known to be exact, we can:

- in compile-time: calculate ω^{-1}, the inverse element of ω in $\mathbb{Z}_{2^W}$;
- in run-time: multiply the dividend by ω^{-1} modulo 2^W.

Finally, we can calculate the depth as

$$\delta = \mathcal{L} - 1 - \left(\left((\kappa \gg \ell) \cdot \omega^{-1} \right) \bmod 2^W \right).$$

If C is odd, then $\ell = 0$, so we do not need the shift. If C is a power of two, then $\omega = \omega^{-1} = 1$, so we do not need the multiplication.

5 Optional features

5.1 Cached path

5.1.1 Basics

In this section, bit sequences in **bold** mean keys, while underlined bit sequences refer to nodes corresponding to those sequences. As a special case, $\underline{\varepsilon}$ refers to the root node.

As we have previously shown, market data exhibits strong sequential locality. To exploit this, we can cache the path (up to a pre-leaf node) in the trie to the last inserted element.

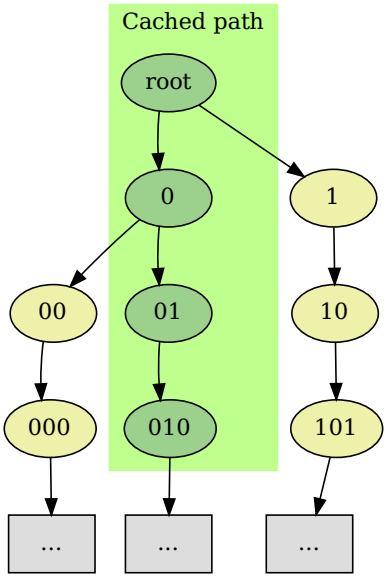
The cached path consists of:

- the last key;
- a *path*, which is an array ρ of index-pointers of length $\mathcal{L}$;
- the number $d \in \mathbb{N}, d \leq \mathcal{L}$ representing the actual size of ρ.

The latter is needed in order to be able to truncate the cached path on erase operation instead of invalidating all of the cached path.

Here is the situation where the cached path is full ($\mathcal{L} = d = 4$):

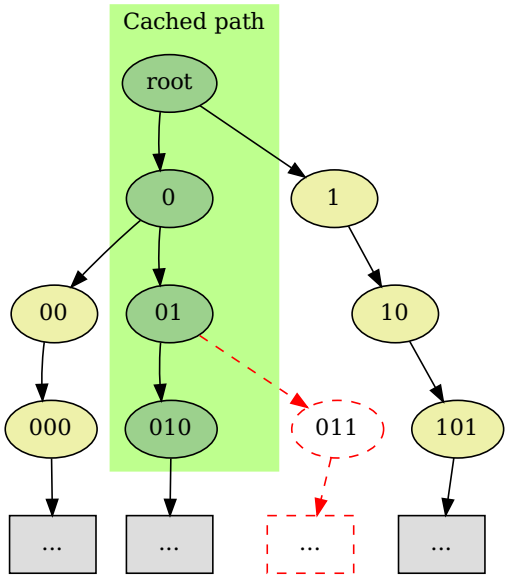
Figure 3: A trie with full cached path



5.1.2 Insertion and lookup

Above, the last operation was insertion of **010**, so last key is **010**, ρ is $\langle \underline{\varepsilon}, \underline{0}, \underline{01}, \underline{010} \rangle$, $d = 3$. Suppose now we want to insert **011**:

Figure 4: A trie with a to-be-inserted node



The main idea is that we can quickly calculate the number of nodes in the common prefix of the last key and the new key. Let k_1, k_2 be two keys, W bits each. We can calculate the length λ of the common prefix, in chunks of C bits, of k_1 and k_2 as following:

$$\lambda = \left\lfloor \frac{\beta - (W - K) + \underline{\mathrm{clz}}_W(k_1 \oplus k_2)}{C} \right\rfloor, \tag{1}$$

where bias $\beta = (-K) \bmod C$, $\oplus$ denotes bitwise XOR operation, and $\underline{\mathrm{clz}}_W(x)$ is the count-leading-zero-bits operation for bit width W, which returns W if x is zero: it counts the number of consecutive zero bits, starting with the most significant one, in x.

In the example above, we want to calculate the common prefix of last key **010** and new key **011**. Let us say $W = 8$; the shape of the tree implies $K = 3, C = 1$.

- We calculate $\underline{\mathrm{clz}}_8(\mathbf{00000010} \oplus \mathbf{00000011}) = 7$;
- $\beta = 0$, $(W - K) = 5$, so the numerator is $0 - 5 + 7 = 2$;
- $\lambda = \lceil 2/1 \rceil = 2$.

So we jump right into $\rho[\lambda]$, which is $\underline{01}$; it is, indeed, the lowest common ancestor of $\underline{010}$ and $\underline{011}$ (if the latter would be in the tree).

The same sequence of steps can be used to *locate* the key **011** in the tree if the last key is **010**.

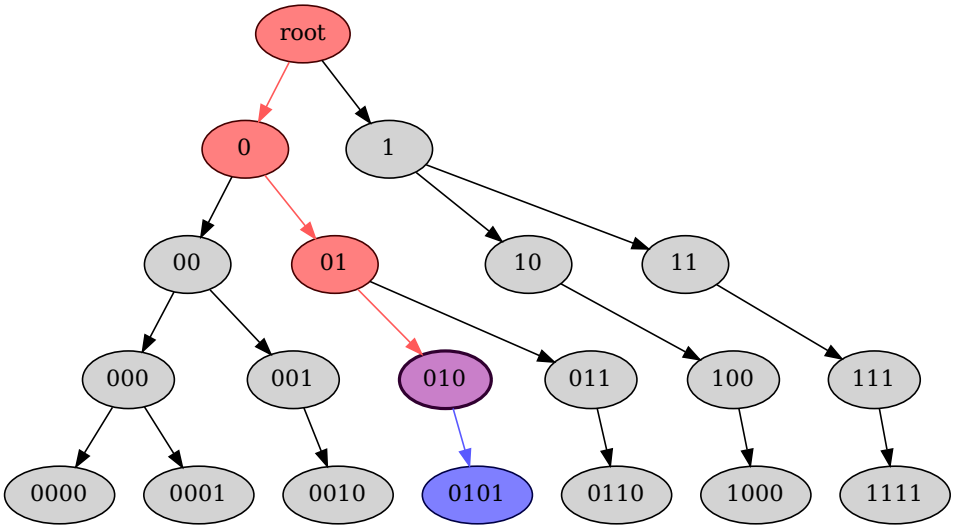
The count-leading-zeros instruction is readily available on all modern hardware, and is reasonably fast. Apart from it, the computation compiles down to a XOR, an addition or a subtraction, and division by a constant. Modern compilers optimize integer division by a constant, using results from [6], down to a sequence of cheaper operations. For values of C such that $C \leq 6$ and C is not a power of two ($C \in \{3, 5, 6\}$), the sequence only involves single multiplication and single right shift.

We use 6 as a realistic upper bound on C because, on $C > 6$, the *glass* would occupy an unrealistic amount of memory; also, the mask would need to contain more than 64 bits, which would slow down common operations on 64-bit hardware.

5.1.3 Erasure

Suppose we have a tree (not a trie!), where all leafs are at the same depth, and two paths: the red one goes from the root down to some node (not necessarily a leaf); the blue one goes from a node (not necessarily the root) down to a leaf. In the picture below, the red path is $\langle \underline{\varepsilon}, \underline{0}, \underline{01}, \underline{010} \rangle$; the blue path is $\langle \underline{010}, \underline{0101} \rangle$; their intersection, the node $\underline{01}$, is colored purple:

Figure 5: A tree with red and blue paths



Suppose we know that:

- the distance from a root to a leaf is $\mathbf{L}$ (on the picture above, $\mathbf{L} = 4$);
- the length of red path, in edges, is $\mathbf{R}$ (on the picture above, $\mathbf{R} = 3$);
- the length of the blue path, in edges, is $\mathbf{B}$ (on the picture above, $\mathbf{B} = 1$);
- the distance from the root to the lowest common ancestor of the last nodes of red and blue paths, is $\mathbf{Z}$ (on the picture above, the lowest common ancestor is 010, so $\mathbf{Z} = 3$).

The number $\mathbf{I}$ of nodes in the intersection of the red and blue paths can be calculated as follows:

$$\mathbf{I} = \max\{0,\, \mathbf{Z} + \mathbf{B} + 1 - \mathbf{L}\}.$$

On the picture above, $\mathbf{I} = 1$. Also note that it turns out we do not even need $\mathbf{R}$ for the calculation of $\mathbf{I}$.

On erasure, our "red path" is the cached path, and the "blue path" consists of the nodes and edges that have been removed. We substitute $\mathbf{L} = \mathcal{L}$, and calculate $\mathbf{Z}$ as minimum of:

- d, and
- the length of the common prefix, in chunks of C bits, of the last key and the erased key (see formula 1).

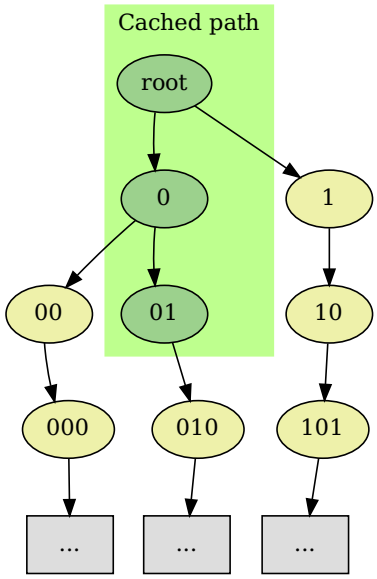
We then truncate the cached path by $\mathbf{I}$: that is, we assign

$$d \longleftarrow \max\{0,\, d - \mathbf{I}\}.$$

The max operator is needed because, if the whole tree has been removed, including the root, $\mathbf{I} = d + 1 > d$.

Let us now briefly go back to our trie examples in the previous subsubsection. Here is what the cached path looks like after the deletion of the key **011** (after it has been inserted):

Figure 6: A trie after erasure: cached path is truncated



The cached path is now truncated ($d < \mathcal{L}$).

5.2 Hash table

In order to further speed up key lookup, the *glass* also supports a hash table (or, rather, a *cache table*). It maps keys without the last chunk (the least

significant C bits) into index-pointers to the pre-leaf the key belongs to. It uses separate chaining. However, it is different from a standard hash table in the following ways:

- A chain is a doubly-linked list, instead of singly-linked, in order to support hard $O(1)$ removal by pointer. Insertion is always done into the beginning of a chain;
- A lookup only inspects first J elements of a chain, so that lookup is hard $O(1)$. If a match is found among the first J elements, it returns "exists" and the pre-leaf that the key is mapped to. If the chain length is less or equal to J, it returns "doesn't exist". If the chain length is greater then J, it returns "don't know". J is a compile-time constant which is currently set to 5;
- On a resize (the hash table only supports growing, not shrinking), the relative order of the elements within new chains that previously were in the same chain is preserved. This is done via moving of some elements in the old chains into the beginning of the new chains and then reversing the order of elements in the new chains. This is done because we believe that recently inserted elements are more likely to be accessed.

The next-in-hash-table/previous-in-hash-table index-pointers are embedded into the nodes of the tree, although only used in pre-leaf nodes. To be able to compare keys during lookup, we also have "hash table key" field in every node, although it is also only used in pre-leaf nodes. The first-in-hash-table index-pointers (we don't need pointers to the end of the chains) are stored in a separate array. Its size is kept at the largest power of two which is not greater than the tree's capacity.

5.3 Probability of hash table's "don't know" answer

We can use results from [10] to calculate the probability of hash table's "don't know" answer for a key that is present.

Assume simple uniform hashing. The probability that a given bucket in a hash table with n elements and b buckets has size $k, 0 \leq k \leq n$, is

$$\mathbf{p}(k) = \binom{n}{k} \left(\frac{1}{b}\right)^k \left(\frac{b-1}{b}\right)^{n-k}.$$

We can calculate the probability of finding our key in the first J buckets,

among $k, 1 \leq k \leq n$ buckets, as follows:

$$\mathbf{q}(k) = \begin{cases} 1, & k \leq J; \\ J/k, & k > J. \end{cases}$$

We can calculate the probability of hash table's "don't know" answer, if the key is present, as follows:

$$p_+ = 1 - \frac{\sum_{k=1}^{n} \mathbf{p}(k)\mathbf{q}(k)}{1 - \mathbf{p}(0)}.$$

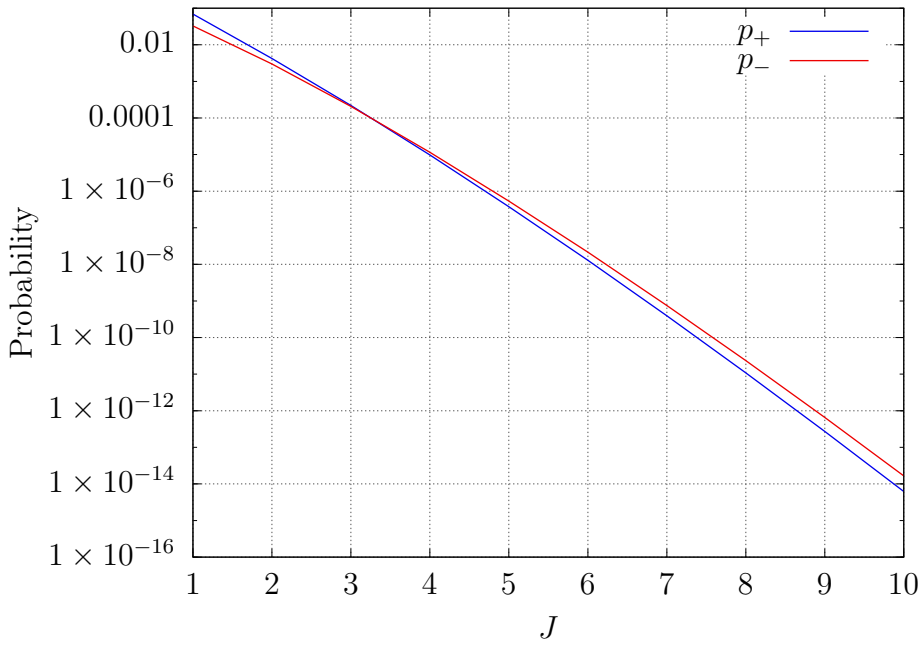
The division is because we only interested in cases where the bucket is not empty (otherwise the key can not be present in this bucket).

The probability of hash table's "don't know" answer if the key is **not** present is just a probability that a given bucket contains more than J elements:

$$p_- = \sum_{k=J+1}^{n} \mathbf{p}(k).$$

We calculated p_+ and p_- as functions of J, with $n = 9210$ (the exact upper bound on the number of elements in a trie with 16-bit index pointer), $b = 2^{15}$ (the number of buckets in a trie with 16-bit pointers with maximal possible capacity):

Figure 7: Probabilities of hash table's "don't know" answer



For $J = 5$, $p_+ \approx 3.76 \cdot 10^{-7}$, $p_- \approx 2.14 \cdot 10^{-8}$.

5.4 Cached iterators to the first and last elements

glass supports caching iterators to the first and last elements. There are two modes of caching: *eager* and *lazy*.

The iterators can be either valid, point to *end* (meaning the trie is empty), or be in *bad* state (the latter is only possible in the lazy mode).

On insertion, if the inserted element is less/greater than the previous first/last element, the corresponding cached iterator (or both, if the tree was empty) is updated.

Eager and lazy modes differ in the behaviour on erasure (without loss of generality, assume the trie is not empty after erasure; otherwise, we simply assign *end* to both cached iterators instead):

- in eager mode, if the first or last element has been removed, the corresponding cached iterator is updated immediately (the relatively costly procedure of finding the first or last element is performed). On first/last query, the corresponding cached iterator is returned;
- in lazy mode, if the first or last element has been removed, the corresponding cached iterator is put into *bad* state. On first/last query, if the corresponding cached iterator is not in *bad* state, it is returned; otherwise, the procedure of finding the first or last element is performed, the corresponding cached iterator is updated and returned.

We can always afford a separate value for iterator's *bad* state because we always set maximum capacity to at most $2^W - 2$, where W is the width, in bits, of an index-pointer.

5.5 Trash encoding

Trash encoding is a way to lower memory usage at a small runtime cost. *Glass* requires an allocator that produces zeroed out memory at allocation (in which case the whole new chunk must be zeroed out) and reallocation that grows an allocated chunk (in which case the new memory must be zeroed out). This is required so that we don't need to zero out a new node's mask: it is either freshly-allocated memory, or was zeroed out on erasure. The default allocator uses `calloc`/`realloc`+`memset`/`free`.

The idea is that we can use the `mmap`/`mremap`/`munmap` system calls for these operations. They have granularity of a page (4096 bytes on modern hardware) and, on Linux, they are able to overcommit memory (produce pages that look zeroed out from the userspace' point of view, but are

only associated with physical pages on the first write). This only works if `vm.overcommit_memory` parameter is set to 0 ("heuristic overcommit handling"; it is the default) or 1 ("always overcommit") [1].

The trash encoding is just a special way to encode an unused node's "next-free-node" field:

- we introduce a new special value that means "the next node in the nodes array"; it is encoded as 0;
- the special value that means "no next node" is encoded as 1;
- a reference to the node with index i is encoded as $(i+2)$. This addition can never overflow because we always set maximum capacity to at most $2^W - 2$, where W is the width, in bits, of an index-pointer; and the index of a node is always less than the capacity.

When "next-free-node" fields are encoded in this way, we do not need to additionally initialize all the nodes in the beginning and the new nodes after a reallocation, so the never-used portion of nodes in the end of the nodes array (up to a page boundary) does not consume memory.

The "next-free-node" field with value v of an unused node with index j is decoded as follows:

- if $v = 0$, the result is $(j + 1)$;
- it $v = 1$, the result is "there's no next node";
- otherwise, the result is $(v - 2)$.

5.6 Compressed iterators

If the hash table is used (so that all the pre-leaf nodes contain "hash table key" field) we can only store the C least significant bits of the key in an iterator instead of the full key. We can then recover the full key via lookup in the node array and a bitwise OR operation, although this is slower than using an "uncompressed" iterator.

Assume $C \leq 8$. A compressed iterator would then consist of $\langle i, \mathfrak{K} \rangle$ pair, where i is an index-pointer, $\mathfrak{K}$ is an 8-bit value with the C least significant bits of the key. For our setup with 16-bit index pointers, $K = 50$, $C = 5$, this reduces the size of an iterator from 16 bytes to 4 bytes.

Glass supports such "compressed" iterators (but only if the hash table is used and $C \leq 8$). It provides functions to convert compressed iterator to/from uncompressed ones, and to get a pointer to the element behind a compressed iterator to perform read/write access through it.

6 Implementation details

Our implementation targets C99 with GNU extensions, although it also compiles as C++11 with GNU extensions in order to be able to compare it against C++' `std::map`.

We use a custom pre-processor that helps in writing "X macro"-styled generic code, and also managing macros (undefining them in the end), including settings that are passed as preprocessor defines. The code including the glass source must define `GLASS_PREFIX`; all functions will be prefixed with it. For example, if `GLASS_PREFIX` is `my_glass`, the creation function will be called `my_glass_create`. In order to do this, we define `GLASS_NAME(SUFFIX)` macro that concatenates together (with `##`) `GLASS_PREFIX`, "_" and `SUFFIX`. The name of a function then can be written as `GLASS_NAME(create)`. The pre-processor allows us to write `@create` instead of `GLASS_NAME(create)`. It also serves to reduce error-prone boilerplate related to keeping track of macros that should be undefined in the end, including the settings definitions.

The preprocessor is called "ato", because that's the name of the "`@`" character in Japanese.

7 Benchmarks

7.1 Set and setting

All measurements were performed on Xiaomi RedmiBook 15 TM2039-44450 laptop.

We have taken the following measures to ensure the benchmarks are as fair as possible:

- in order to minimize possible interferences, the benchmark was run in Linux kernel's system console; any other applications, including the X server, were not launched;
- `/sys/devices/system/cpu/cpufreq/policy*/scaling_governor` policies were set to "performance";
- before running the benchmark, the driver process has been reniced with "`renice -n -20`";
- the driver process was pinned to a single core with "`taskset -c 1`";

- timing measurements were taken with "`mfence`, `lfence`, `rdtsc`, `lfence`" sequence of assembly instructions, but there is no significant difference in results if `clock_gettime(CLOCK_MONOTONIC, ...)` is used instead.

In order to be fair, we have also implemented a custom allocator for `std::map`, which uses the same "slot allocator" principle that the *glass*' allocator uses. It is faster than glibc's allocator used by default because:

- slot allocator doesn't do locking;
- slot allocator's allocation (assuming no need to allocate a new arena, which is very rare) and deallocation run in hard $O(1)$ time;
- glibc's allocator uses red-black trees to manage the list of free chunks, which is slow because of pointer chasing.

It is consistently faster in benchmarks. At first we used a separate slot allocator for each `std::map` copy, but the approach with a single common slot allocator turned out not only to be faster, but also to produce the results that make more sense.

7.2 Synthetic vs real market data

For benchmarking, we can use real market data. Unfortunately, as it is, it does not provide us a way to measure the performance of specific operations (insert, erase, find of existing/non-existing element, iteration over 25 best prices).

In order to measure these operations separately, we generate synthetic data: to generate a sequence of unique prices, we use a random number generator to generate differences between successive prices that are distributed just like the real market data, except that zero difference is prohibited. We generated synthetic price sequences using the method described above for insert, erase, and find (of existing/non-existing element) operations.

7.3 Amplification

Amplification is an action of replacing an operation in a sequence with multiple identical copies of it.

It only makes sense to amplify read-only operations (find and iteration), because a second insert with the same key degenerates to a lookup (which would say the key already exists in the *glass*), and likewise a second erase with the same key degenerates to a lookup (which would say the key is not present

in the *glass*). These modifying operations also modify the cached path, so any time measurements of their amplified copies would not be representative.

Because we can measure the performance of find operation with synthetic data, we only amplify the operation of iteration over the best 25 prices (in the captions of the graphs below referred to as "iter"). The amplification is done for the sequence of operations representing the real market data. We chose to set the amplification coefficient to 100x.

When a read-only operation is amplified, it also makes sense to remove all other read-only operations from the input in order to be closer to only measuring this operation. In the context of the previous paragraph, this means that we remove find operations from the input when measuring amplified iteration.

7.4 Multiple copies

In order to have a more realistic benchmark, we create multiple copies of the data structure being benchmarked (either *glass* or `std::map`) that are operated upon: instead of applying the operation to a single copy, we apply it to all the copies.

Since in reality we are going to operate upon order books of up to 100 instruments, benchmarks with multiple copies are more faithful, in particular regarding the behaviors related to CPU caches.

The graphs below are parameterized by the number of copies (from 1 to 32, inclusively).

7.5 Graphs

Each test was run with the following number of iterations (n is the number of copies):

- $\lfloor \frac{2500}{n} \rfloor$ for tests that use synthetic data;
- $\lfloor \frac{7500}{n} \rfloor$ for tests that use the real market data.

Figure 8: Synthetic data: **insert**

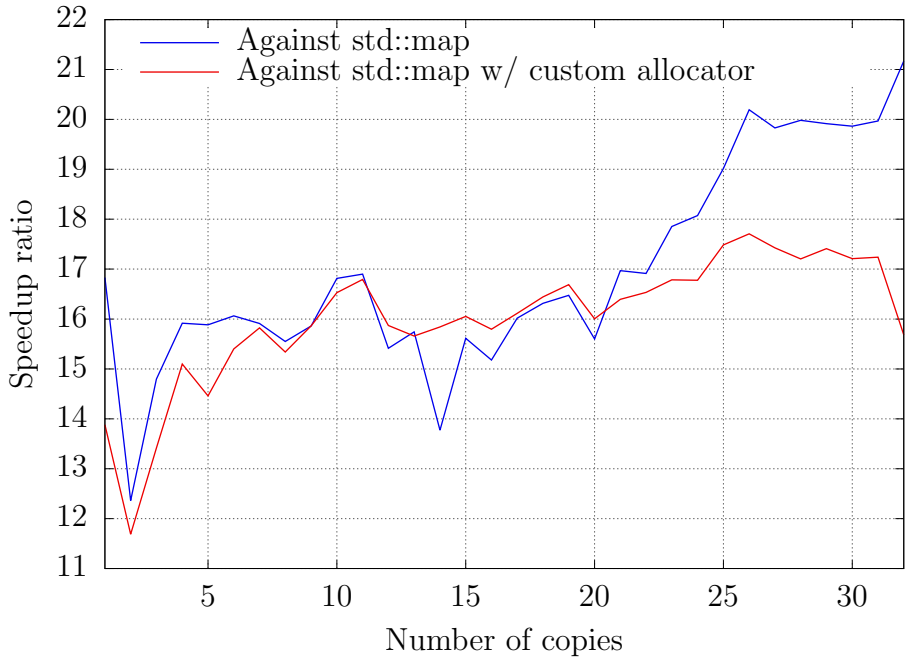


Figure 9: Synthetic data: **erase**

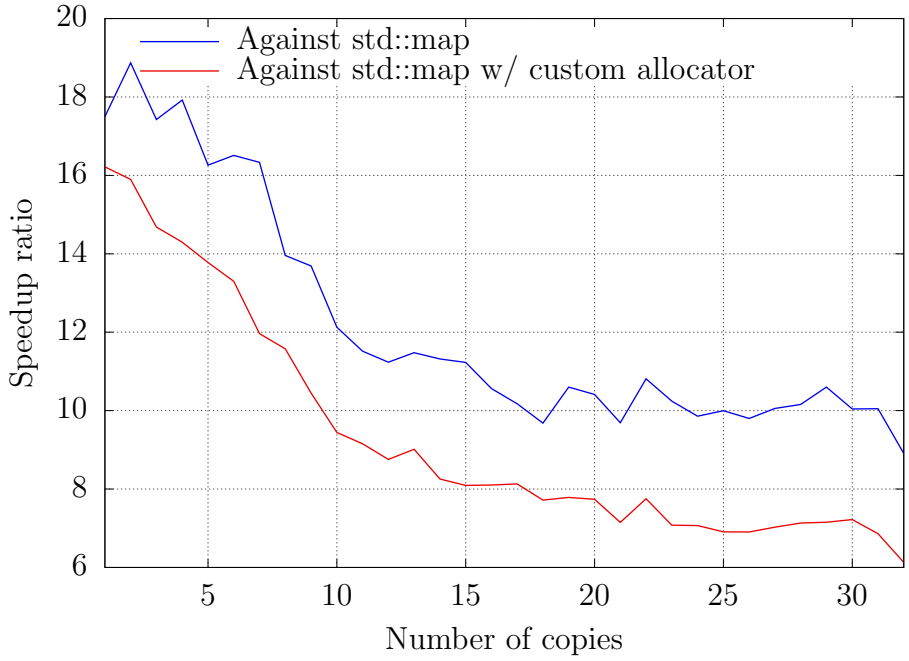


Figure 10: Synthetic data: **find existing**

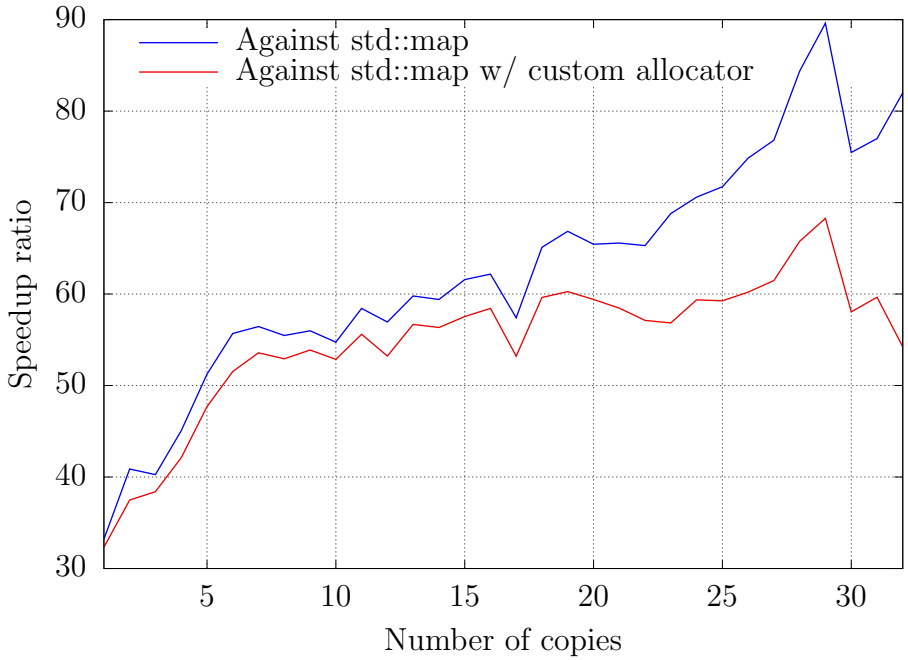


Figure 11: Synthetic data: **find non-existing**

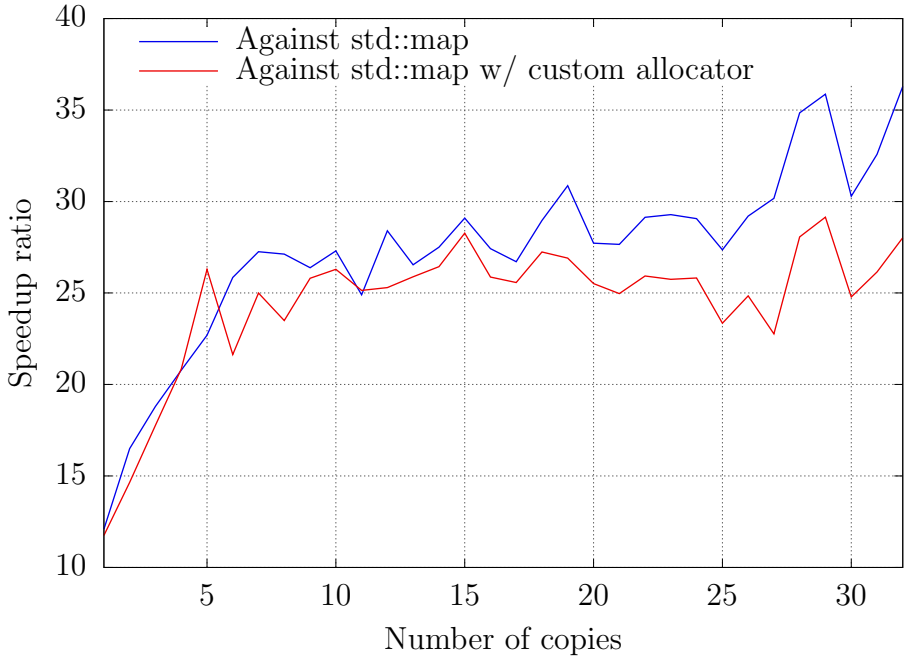


Figure 12: Real market data: **no amplification**

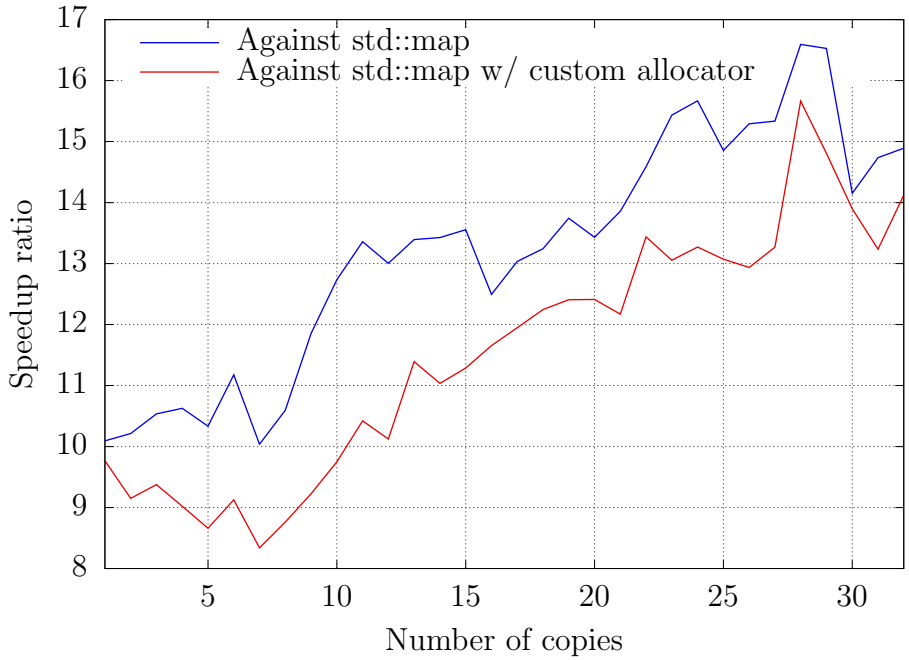
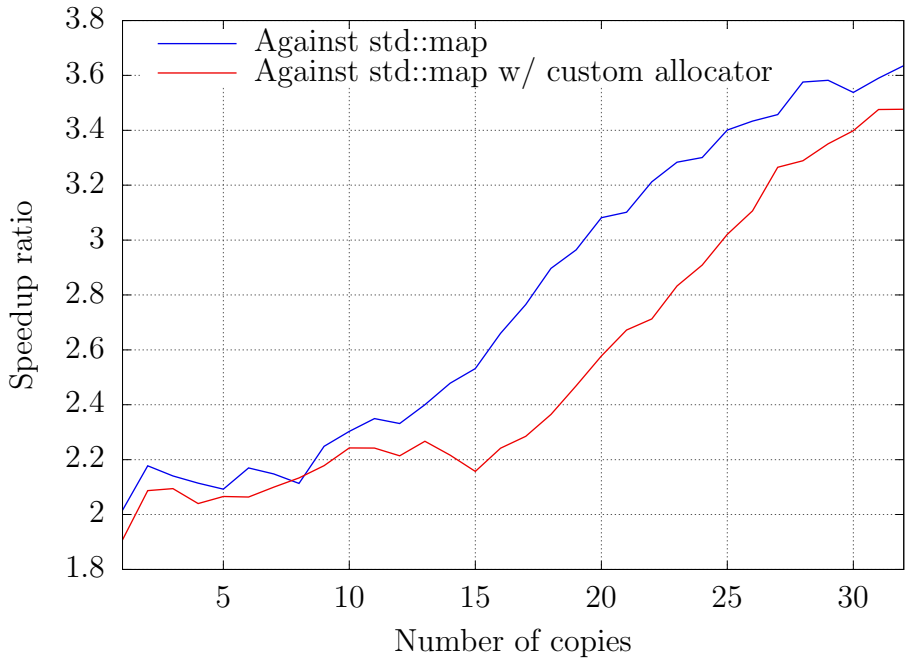


Figure 13: Real market data: **iter amplified 100x**



8 Availability

The code of our implementation and LaTeX source of this paper are available at `https://github.com/shdown/glass-paper`. The code is licensed under the MIT license. The source of this paper is licensed under the Creative Commons BY 4.0 license.

References

[1] Linux kernel developers. `https://www.kernel.org/doc/Documentation/vm/overcommit-accounting`. [Online; accessed 27-May-2025].

[2] AMD. AMD64 technology: AMD64 architecture programmer's manual. volume 3: General-purpose and system instructions. `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf`. [Online; accessed 27-May-2025].

[3] Rust documentation. BTreeMap in std::collections. `https://doc.rust-lang.org/std/collections/struct.BTreeMap.html`. [Online; accessed 27-May-2025].

[4] Rust documentation. BTreeSet in std::collections. `https://doc.rust-lang.org/std/collections/struct.BTreeSet.html`. [Online; accessed 27-May-2025].

[5] Free Software Foundation. Freeing after malloc (the GNU C library). `https://www.gnu.org/software/libc/manual/html_node/Freeing-after-Malloc.html`. [Online; accessed 27-May-2025].

[6] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *SIGPLAN Not.*, 29(6):61–72, June 1994.

[7] Intel. Documentation for `_bzhi_u32` and `_bzhi_u64` intrinsics. `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/bzhi-u32-64.html`. [Online; accessed 27-May-2025].

[8] Intel. Intel® 64 and ia-32 architectures software developer's manual, volume 2 (2A, 2B, 2C, & 2D): Instruction set reference. `https://cdrdv2-public.intel.com/789581/325383-sdm-vol-2abcd.pdf`. [Online; accessed 27-May-2025].

[9] ISO. ISO/IEC JTC1 SC22 WG21 N 4860: Programming languages — C++. https://isocpp.org/files/papers/N4860.pdf, 2020. [Online; accessed 27-May-2025].

[10] R. Christopher Lacher. Hash table analysis (course material). https://www.cs.fsu.edu/~lacher/courses/notes/hashanalysis.pdf. [Online; accessed 27-May-2025].

[11] Rob Landley. Red-black trees (rbtree) in Linux. Linux kernel documentation, file "rbtree.txt". https://www.kernel.org/doc/Documentation/rbtree.txt. [Online; accessed 21-April-2025].

[12] Oracle. TreeMap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html. [Online; accessed 27-May-2025].

[13] Oracle. TreeSet (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html. [Online; accessed 27-May-2025].

[14] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. Lua 5.3 reference manual — lua_Alloc. https://www.lua.org/manual/5.3/manual.html#lua_Alloc. [Online; accessed 27-May-2025].