Matryoshka Trees

A Cache-Conscious B^+ Tree with
Matryoshka-Nested Sub-Trees at Every Memory Hierarchy Level

Design Document

February 20, 2026

This document describes the design of the *matryoshka tree*, a B^+ tree variant in which each leaf node (a 4 kB page) contains a complete B^+ sub-tree of cache-line-sized (64 B) sub-nodes, each searched via SIMD register–width (16 B) comparisons. This "matryoshka nesting"—trees within trees, each level mapped to a memory hierarchy level—replaces the flat FAST-blocked arrays of the earlier design with a true hierarchical B^+ structure at every scale. Insert and delete operate on individual cache-line sub-nodes via split, merge, and redistribute, giving $\mathcal{O}(\text{sub\_height} \times 15)$ work per leaf modification instead of $\mathcal{O}(B)$ flat-array rebuilds. The name reflects the nested Russian-doll architecture: SIMD registers nest inside cache-line sub-nodes, which form a B^+ sub-tree inside each page-sized leaf node, which composes the full outer B^+ tree.

Contents

1 Introduction **3**

2 Conceptual Motivations **4**
- 2.1 Matryoshka Dolls . . . 4
- 2.2 Inception Analogy . . . 5

3 Related Work **5**

4 Architecture Overview **7**
- 4.1 Memory Hierarchy Mapping . . . 7
- 4.2 Overall Tree Structure . . . 7

5 Node Structures **8**
- 5.1 CL Leaf Sub-Node (`mt_cl_leaf_t`) . . . 8
- 5.2 CL Internal Sub-Node (`mt_cl_inode_t`) . . . 8
- 5.3 Page Header (`mt_page_header_t`) . . . 9
- 5.4 Leaf Page Layout . . . 9
- 5.5 Capacity Analysis . . . 10
- 5.6 Outer B^+ Tree Internal Node (`mt_inode_t`) . . . 10

6 Page-Level Sub-Tree Operations **11**
- 6.1 Intra-Page Sub-Tree Structure . . . 11
- 6.2 Search . . . 11
- 6.3 Insert . . . 12
- 6.4 Delete . . . 13
- 6.5 Page Split . . . 14
- 6.6 Bulk Load . . . 14

7 Outer B^+ Tree Operations **15**
- 7.1 Bulk Load . . . 15
- 7.2 Point Query and Predecessor Search . . . 15
- 7.3 Insert . . . 15
- 7.4 Delete . . . 16
- 7.5 Range Scan (Iteration) . . . 16

8 Complexity Analysis **17**

9 Cache Behaviour Analysis **17**
- 9.1 Within a Page (CL Sub-Tree) . . . 17
- 9.2 Between Pages (Outer B^+ Tree) . . . 18
- 9.3 Comparison with Old Design . . . 18
- 9.4 Comparison with Pure FAST . . . 18

10 Implementation Notes **18**
- 10.1 Arena Allocator . . . 18
- 10.2 Slot Allocation within a Page . . . 19
- 10.3 Compilation Constants . . . 19

10.4 Runtime Hierarchy Configuration 19
10.5 Limitations and Future Work 19

11 Summary **20**

1 Introduction

The matryoshka tree addresses a long-standing tension in main-memory index design: *search throughput versus modification cost.*

On one end of the spectrum, the FAST tree [5] achieves near-optimal search throughput by mapping an implicit (pointerless) search tree onto the memory hierarchy through a recursive blocking scheme. Each level of the blocking matches a level of the hardware: SSE registers (16 B), cache lines (64 B), pages (4 kB), and superpages (2 MiB). Because the layout is implicit (positions are computed arithmetically, not followed via pointers), it eliminates pointer-chasing stalls and maximises hardware prefetch effectiveness. The price is that any structural modification—insertion or deletion—requires a full $\mathcal{O}(n)$ rebuild.

On the other end, classical B$^+$ trees [1, 2] support $\mathcal{O}(\log_B n)$ modifications via node-local splits and merges, but suffer from pointer-chasing at every level of the tree, causing a TLB miss and often an L2/L3 cache miss per node visited.

The original matryoshka design confined FAST-style flat blocked arrays *within* each B$^+$ leaf node. While this retained B$^+$ tree navigation between nodes, every insertion or deletion into a leaf still required an $\mathcal{O}(B)$ extraction-and-rebuild of the entire FAST layout (where $B = 511$ for 4 kB pages). The flat layout could not be modified in place.

The current design eliminates this bottleneck. Instead of a flat blocked array, each leaf page now contains a genuine B$^+$ *sub-tree* of cache-line-sized sub-nodes. Insertions and deletions operate on individual 64 B sub-nodes—splitting, merging, and redistributing cache-line leaves and cache-line internal nodes—without touching the rest of the page. The result is:

- **Fast intra-node search**: Each leaf page contains a B$^+$ sub-tree of up to 63 cache-line sub-nodes, navigated via SIMD-accelerated scans at each sub-tree level. A search touches at most sub_height + 1 cache lines.
- **Efficient per-key modifications**: Insert and delete modify only the affected CL leaf (64 B, 15 keys max) and propagate splits or merges through at most sub_height CL internal levels. Cost: $\mathcal{O}(\text{sub\_height} \times 15)$ per leaf modification, versus $\mathcal{O}(511)$ for the old flat rebuild.
- **Matryoshka nesting**: The tree is self-similar at every scale—a B$^+$ tree of pages, each page containing a B$^+$ tree of cache lines, each cache line searched via SIMD registers—mirroring the CPU memory hierarchy.
- **Eager deletion**: Jannink's redistribute/merge algorithm operates at both the CL sub-tree level (within a page) and the outer B$^+$ tree level (between pages), maintaining minimum occupancy invariants at every level.
- **Arena allocation**: Leaf pages are co-located within 2 MiB arenas. `MAP_HUGETLB` is attempted; on failure, `posix_memalign` with `madvise(MADV_HUGEPAGE)` provides transparent huge pages.
- **Range scans**: A doubly-linked leaf list supports efficient sequential iteration across pages.

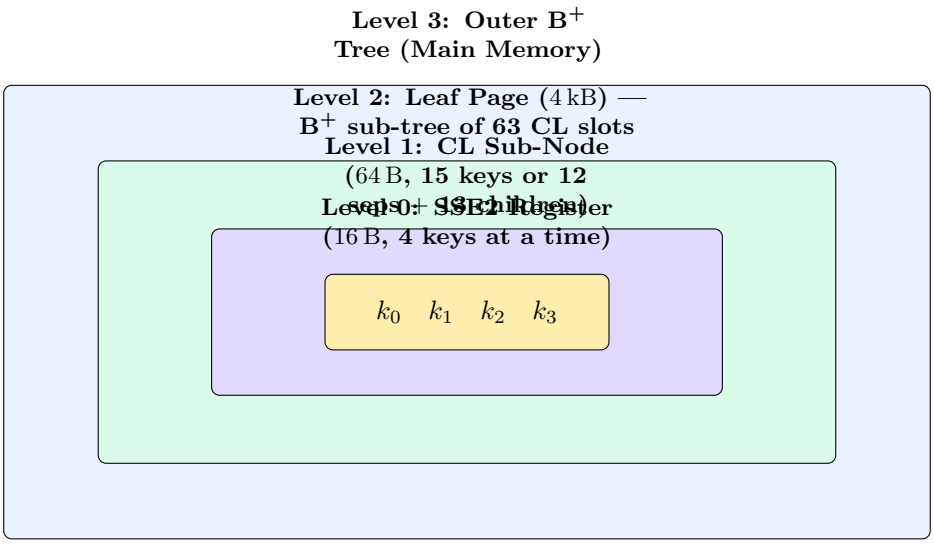

Figure 1: Matryoshka nesting hierarchy. Each level physically nests within the next. SIMD registers (**Level 0**, gold) scan within cache-line sub-nodes (**Level 1**, violet), which form a B$^+$ sub-tree inside each page-sized leaf (**Level 2**, green), composing the full outer B$^+$ tree (**Level 3**, blue). Within each page, navigation uses slot indices (1–63); between pages, navigation uses explicit child pointers.

2 Conceptual Motivations

2.1 Matryoshka Dolls

The project is named after Russian *matryoshka* (nesting) dolls—a set of hollow wooden figures of decreasing size, each placed inside the next larger one. When you open the outermost doll, you find a slightly smaller doll inside; open that, and there is a yet smaller one, and so on down to the smallest solid figure at the centre.

The tree structure mirrors this physical nesting precisely. The outermost "doll" is the B$^+$ tree itself, navigated via pointers in main memory. Open a leaf page—the next doll—and inside you find not a flat array but a complete B$^+$ sub-tree of cache-line-sized nodes. Open a cache-line sub-node—a still smaller doll—and its sorted keys are searched via SSE2 register–width comparisons, four keys at a time.

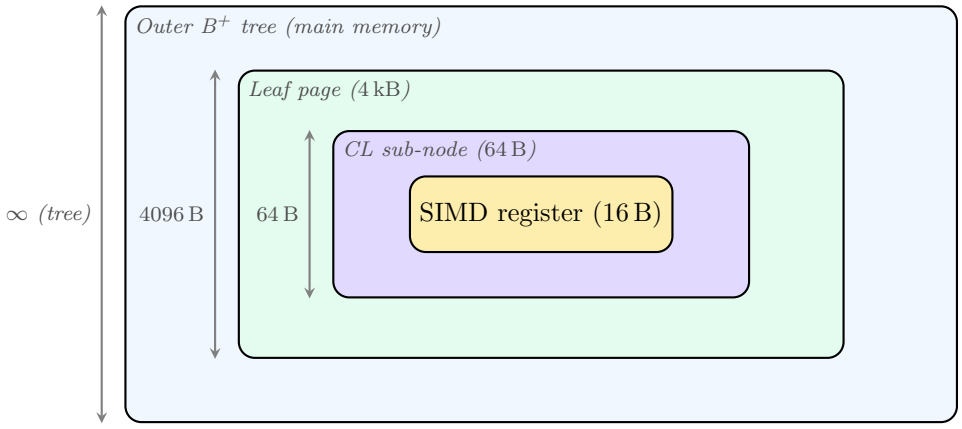

Figure 2: The matryoshka doll analogy. Each level of the data structure physically nests within the next larger one, just as each wooden doll nests within its larger shell.

Each level nests *physically* within the next: every CL sub-node resides within a page, and every SIMD comparison operates on bytes within a CL sub-node. The analogy is not merely decorative—it captures the essential design principle. Just as a matryoshka set is a single unified artefact despite comprising multiple nested objects, the matryoshka tree is a single coherent index despite operating under different structural rules at each nesting level.

2.2 Inception Analogy

The nesting also evokes Christopher Nolan's film *Inception* (2010), whose premise involved dreams nested within dreams to great depth, each level operating under its own rules and timescale.

In the matryoshka tree, each nesting level operates under its own structural rules:

- The **outer B$^+$ tree** uses pointers and page-granularity splits. Its branching factor is up to 340 children.
- The **CL sub-tree** within each page uses slot indices (1–63) and cache-line-granularity split, merge, and redistribute operations. CL leaves hold 15 keys; CL internal nodes hold 12 separators and 13 children.
- **SIMD search** operates on register-width vectors, comparing 4 keys per instruction in a sorted scan.

The increasing time compression in deeper dream levels in *Inception* parallels the decreasing access latency at deeper nesting levels in the matryoshka tree: main memory access takes $\sim$100 ns, L1 cache hits take $\sim$1 ns, and register operations complete in a fraction of a nanosecond. Each level inward is both physically smaller and faster, just as each dream level inward compressed subjective time.

However, unlike *Inception*'s concern with subjective reality and the instability of deeply nested dream states, the matryoshka tree's nesting is structurally stable. Each level reinforces the others through well-defined split/merge invariants: a CL leaf split propagates to its CL internal parent; if the page runs out of CL slots, a page-level split propagates to the outer B$^+$ tree. The nesting never "collapses"—it is the foundation of correctness, not a source of fragility.

3 Related Work

The matryoshka tree draws from several lines of research in cache-conscious indexing. We situate it within the literature below.

B-trees and B$^+$ trees. Bayer and McCreight [1] introduced the B-tree as a balanced, external-memory search tree with $\mathcal{O}(\log_B n)$ search and modification costs. The B$^+$ tree variant, formalized in surveys by Comer [2] and later by Graefe [3], stores all data in leaf nodes connected by a linked list, with internal nodes serving purely as a routing index. This separation enables efficient range scans and simplifies concurrency control [6].

Cache-sensitive search trees (CSS-trees). Rao and Ross [9] observed that the pointer overhead in B$^+$ trees wastes cache space and proposed *CSS-trees*: implicit directory levels stored in sorted arrays, with child offsets computed arithmetically. This eliminates pointers within the directory, improving cache utilization. However, CSS-trees are static (read-only after bulk construction).

Cache-conscious B$^+$ trees (CSB$^+$-trees). Rao and Ross [10] extended their cache-conscious approach to support modifications by storing child nodes of each internal node in a contiguous

array, so a single pointer suffices per node (plus an offset). This halves pointer storage compared to standard B$^+$ trees. Hankins and Patel [4] subsequently showed that the optimal node size for cache-conscious B$^+$ trees is typically the cache-line size or a small multiple thereof, depending on the access pattern.

SIMD-accelerated search. Zhou and Ross [13] demonstrated that SIMD instructions (specifically SSE2's `_mm_cmpgt_epi32`) could accelerate database operations including sorted-array search, achieving 4× parallelism per comparison. Schlegel et al. [11] generalized this to k-ary search on modern processors, showing that $k = 2^{d_K}$ simultaneous comparisons reduce tree depth by a factor of d_K relative to binary search.

FAST: Fast Architecture Sensitive Tree. Kim et al. [5] combined hierarchical blocking with SIMD comparisons to build a static index that maps directly onto the memory hierarchy. The FAST tree defines blocking depths d_K (SIMD register), d_L (cache line), and d_P (page/superpage) such that each block fits exactly in the corresponding hardware unit. Within each block, keys are stored in BFS (breadth-first search) order of a complete binary tree, enabling SIMD-parallel comparisons at each level. The FAST tree achieves throughput within a factor of two of a hardware-optimal binary search, but requires $\mathcal{O}(n)$ reconstruction for any modification.

Masstree. Mao et al. [8] built Masstree, a trie of B$^+$ trees designed for concurrent multicore key-value stores. Each B$^+$ tree level handles a fixed-width slice of the key, amortizing cache misses across key bytes. Masstree uses optimistic concurrency (version numbers per node) rather than locking, achieving high throughput under contention. Like the matryoshka tree, Masstree nests simpler structures inside larger ones, though it uses a trie/B-tree nesting rather than sub-tree/B-tree nesting.

Adaptive Radix Tree (ART). Leis et al. [7] proposed ART, which adaptively selects among four node sizes (4, 16, 48, 256 children) to balance space and search efficiency. ART uses SIMD for its smallest node type (Node16, using `_mm_cmpeq_epi8` for 16-way key matching). While ART excels for variable-length string keys, the matryoshka tree targets fixed-size integer keys where cache-line-granularity B$^+$ operations are most effective.

Positioning of the matryoshka tree. The matryoshka tree occupies a specific niche: it provides SIMD-accelerated intra-page search while supporting per-key modifications that touch only the affected cache-line sub-nodes, avoiding the $\mathcal{O}(B)$ rebuild cost of FAST-within-B$^+$ designs and the $\mathcal{O}(n)$ rebuild cost of pure FAST. Table 1 summarises the trade-offs.

Table 1: Comparison of index structures for sorted integer keys. n = number of keys, B = node capacity, h = sub-tree height within a leaf.

Structure	Search	Insert/Delete	Range Scan	SIMD
B$^+$-tree [1]	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log_B n)$	linked list	no
CSS-tree [9]	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(n)$ rebuild	scan	no
CSB$^+$-tree [10]	$\mathcal{O}(\log_B n)$	$\mathcal{O}(\log_B n)$	linked list	no
FAST [5]	$\mathcal{O}(\log_2 n / d_K)$	$\mathcal{O}(n)$ rebuild	scan	yes
ART [7]	$\mathcal{O}(k)$ depth	$\mathcal{O}(k)$ depth	DFS	partial
Masstree [8]	$\mathcal{O}(k/w \cdot \log_B n)$	$\mathcal{O}(k/w \cdot \log_B n)$	linked list	no
Matryoshka	$\mathcal{O}(\log_B n + h)$	$\mathcal{O}(h \cdot b + \log_B n)$	linked list	yes

In the table above, h denotes the sub-tree height within a leaf page (typically 0–2) and b denotes the CL sub-node capacity (15 keys for CL leaves, 12 separators for CL internals).

4 Architecture Overview

A matryoshka tree is a B^+ tree with two node types. The distinguishing feature is that leaf nodes are not flat arrays of keys but instead contain a nested B^+ sub-tree of cache-line-sized sub-nodes:

Internal nodes store sorted keys and child pointers in a conventional B^+ layout. Intra-node search uses SIMD-accelerated linear scan (for small fanout) or binary search (for large fanout). Each internal node fits in one 4 kB page, holding up to 339 keys and 340 child pointers.

Leaf nodes are 4 kB pages divided into 64 cache-line-sized (64 B) slots. Slot 0 is a page header. Slots 1–63 hold CL sub-nodes—either *CL leaves* (sorted arrays of up to 15 `int32_t` keys) or *CL internals* (separator keys and child slot indices). These sub-nodes form a B^+ sub-tree within the page. Leaves are doubly linked for range scans.

4.1 Memory Hierarchy Mapping

Each level of the matryoshka nesting corresponds to a level of the CPU memory hierarchy:

Table 2: Memory hierarchy mapping. Each nesting level is sized to match a hardware unit.

Level	Hardware Unit	Size	Role in Matryoshka Tree
0	SSE2 register	16 B	SIMD scan: 4 keys per comparison
1	Cache line	64 B	CL sub-node: 15 keys (leaf) or 12 seps + 13 children (internal)
2	Page	4 kB	Leaf page: B^+ sub-tree of 63 CL slots
3	Main memory	∞	Outer B^+ tree: pointer-based navigation

The key insight is that the structure is *self-similar* at each level: the outer tree is a B^+ tree of pages, and each page internally contains a B^+ tree of cache lines. At the finest granularity, each sorted cache-line array is searched by scanning 4 keys at a time in an SSE2 register. This recursive B^+ structure at every scale is what distinguishes the matryoshka design from approaches that use a flat array or implicit tree within each node.

4.2 Overall Tree Structure

Figure 3 shows the complete tree structure, from the outer B^+ tree down through the CL sub-tree within each leaf page.

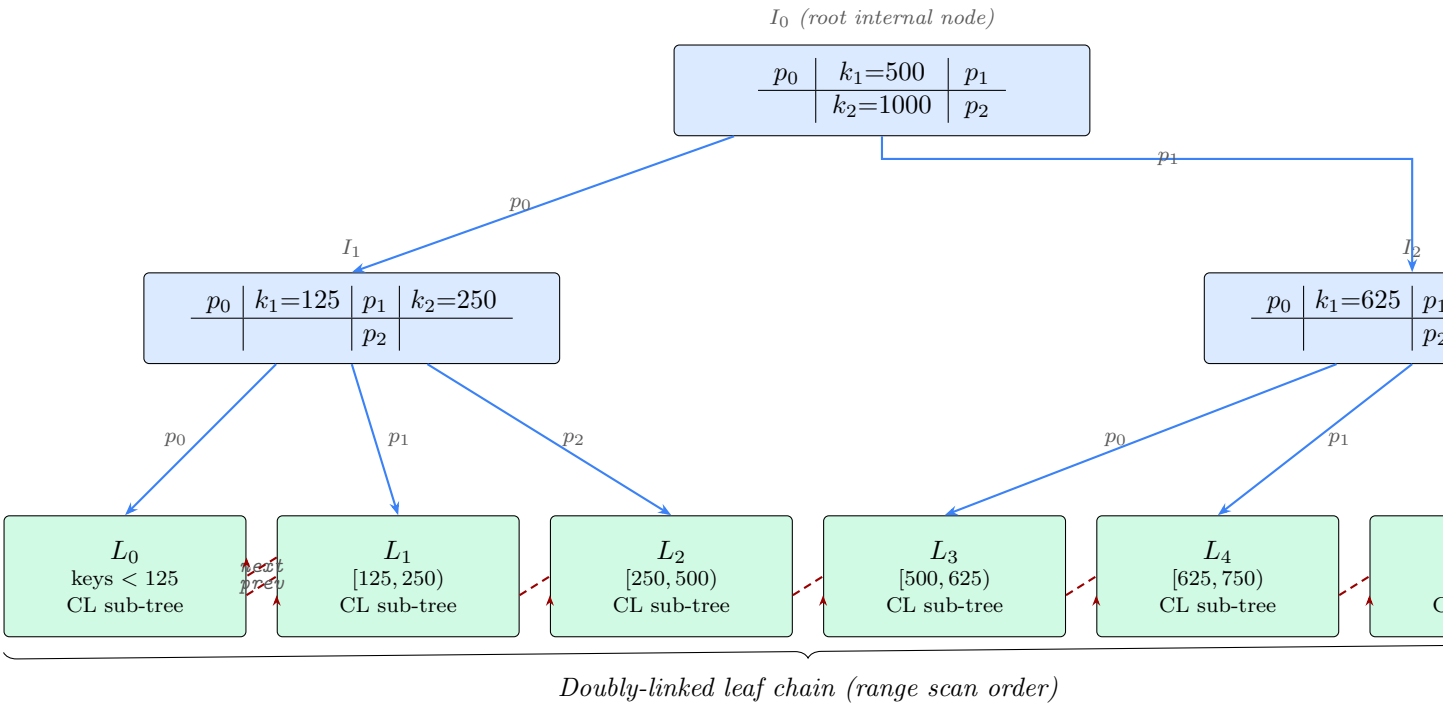

Figure 3: Overall matryoshka tree structure. Internal nodes I_0–I_2 (blue) contain sorted separator keys and child pointers. Leaf nodes L_0–L_5 (green) each contain a B$^+$ sub-tree of CL sub-nodes (not a flat array). Red dashed arrows show the doubly-linked leaf list; blue solid arrows show child pointers.

5 Node Structures

5.1 CL Leaf Sub-Node (`mt_cl_leaf_t`)

A cache-line leaf stores a sorted array of up to 15 `int32_t` keys in exactly 64 B:

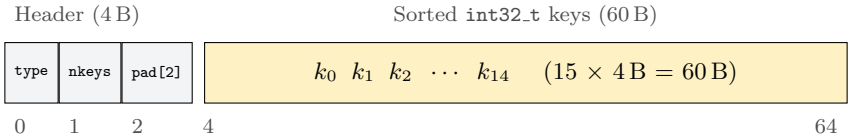

Figure 4: CL leaf sub-node layout (`mt_cl_leaf_t`, 64 B). `type` = `MT_CL_LEAF` (1 byte), `nkeys` = number of valid keys (1 byte), 2 bytes padding, then up to 15 sorted `int32_t` keys. Total: $1 + 1 + 2 + 60 = 64$ bytes, exactly one cache line.

Keys are maintained in sorted order. Insertion shifts at most 15 keys (`memmove` of at most 60 B); deletion similarly shifts keys left. Both operations are fast because the entire node fits in a single cache line.

Minimum fill: 7 keys ($\lfloor 15/2 \rfloor$). Maximum: 15 keys.

5.2 CL Internal Sub-Node (`mt_cl_inode_t`)

A cache-line internal node stores separator keys and child slot indices in 64 B:

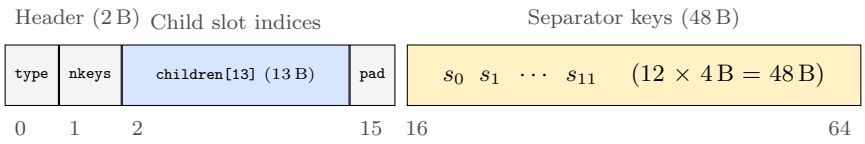

Figure 5: CL internal sub-node layout (`mt_cl_inode_t`, 64 B). `type` = `MT_CL_INTERNAL` (1 byte), `nkeys` = number of separator keys (1 byte), `children[13]` = slot indices (13 bytes, each a `uint8_t` in range 1–63), 1 byte padding, then up to 12 sorted `int32_t` separator keys. Total: $1 + 1 + 13 + 1 + 48 = 64$ bytes.

Children are `uint8_t` slot indices (1–63) pointing to other CL sub-nodes within the same page. The routing semantics follow the standard B$^+$ tree convention: child c_i leads to keys in range $[s_{i-1}, s_i)$, with c_0 covering $(-\infty, s_0)$ and c_n covering $[s_{n-1}, +\infty)$.

Maximum: 12 separator keys, 13 children. Minimum: 7 children ($\lceil 13/2 \rceil$), 6 separator keys.

5.3 Page Header (`mt_page_header_t`)

Slot 0 of every leaf page is the page header, which occupies one cache line:

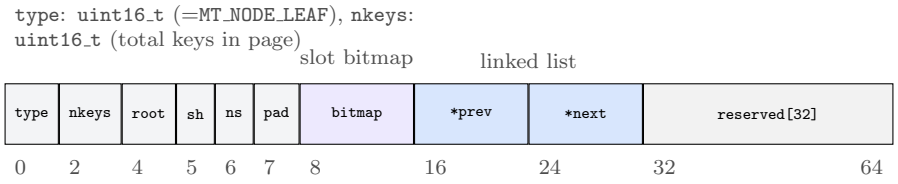

Figure 6: Page header layout (`mt_page_header_t`, 64 B). `type` (2 B): outer node type. `nkeys` (2 B): total keys in page. `root_slot` (1 B): CL slot index of sub-tree root. `sub_height` (1 B): 0 = single CL leaf. `nslots_used` (1 B): allocated CL slot count. `slot_bitmap` (8 B): bits 1–63 track allocation. `prev`/`next` (8 B each): leaf linked list. `reserved` (32 B): padding to 64 B.

5.4 Leaf Page Layout

Each leaf page is a 4 kB (4096 B) region divided into 64 cache-line slots of 64 B each:

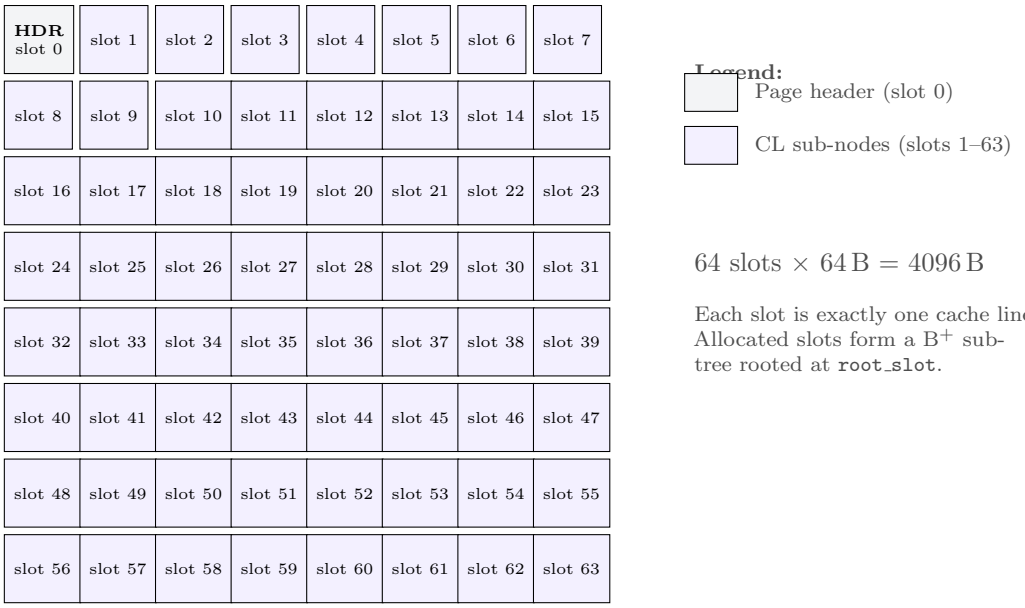

Figure 7: Leaf page layout. A 4 kB page contains 64 cache-line slots. Slot 0 is the page header. Slots 1–63 are available for CL sub-nodes. Allocation is tracked by a `uint64_t` bitmap in the header (bits 1–63). Free slots have type `MT_CL_FREE`; allocated slots are either CL leaves or CL internals forming the intra-page B^+ sub-tree.

5.5 Capacity Analysis

The maximum number of keys per page depends on the sub-tree height:

Table 3: Page capacity by sub-tree height. Each CL leaf holds up to 15 keys; each CL internal has up to 13 children.

Sub-height	CL Internals	CL Leaves	Slots Used	Max Keys
0	0	1	1	$1 \times 15 = 15$
1	1	13	14	$13 \times 15 = 195$
2	5–6	57	62–63	$57 \times 15 = 855$

At sub-height 2, the page is nearly fully utilised (63 of 63 usable slots). For a 4 kB page, the practical maximum is approximately 855 keys—somewhat more than the 511-key flat array of the previous design, though the effective limit depends on fill factor after splits.

The sub-tree grows in height as CL leaves fill and split, allocating CL internal nodes from free slots. When no free slots remain, the page signals `MT_PAGE_FULL` to the outer tree, which splits the page.

5.6 Outer B^+ Tree Internal Node (`mt_inode_t`)

Internal nodes store sorted keys and child pointers in a conventional B^+ layout, unchanged from the classical design:

$$\underbrace{p_0}_{\text{child}_0} \; k_1 \; \underbrace{p_1}_{\text{child}_1} \; k_2 \; \underbrace{p_2}_{\text{child}_2} \; \cdots \; k_m \; \underbrace{p_m}_{\text{child}_m} \tag{1}$$

The maximum number of keys per internal node:

$$\underbrace{16}_{\text{header}} + 4m + 8(m+1) \leq 4096 \quad \Longrightarrow \quad m \leq 339 \tag{2}$$

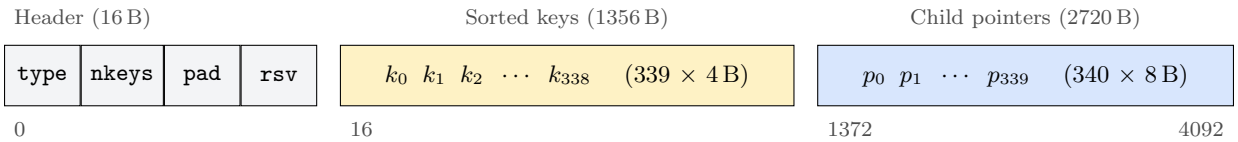

Figure 8: Internal node (`mt_inode_t`) memory layout. The header (16 B) contains the node type, key count, and padding. The sorted key array holds up to 339 `int32_t` keys. The child pointer array holds up to 340 `mt_node_t*` pointers. Total: $16 + 1356 + 2720 = 4092$ bytes.

Internal node search uses SIMD-accelerated linear scan for ≤ 32 keys (loading 4 keys per `_mm_loadu_si128`, comparing via `_mm_cmpgt_epi32`) and falls back to scalar binary search for larger nodes.

6 Page-Level Sub-Tree Operations

This section describes operations on the CL B$^+$ sub-tree within a single leaf page. All operations are confined to the 4 kB page boundary.

6.1 Intra-Page Sub-Tree Structure

The CL sub-nodes within a page form a standard B$^+$ sub-tree:

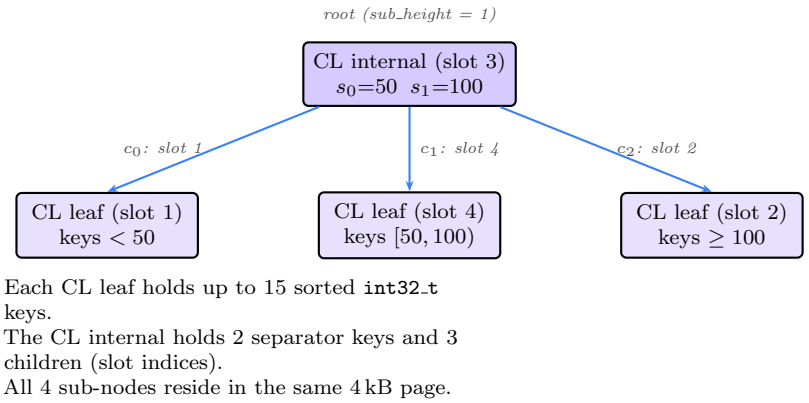

Figure 9: Intra-page CL sub-tree (sub-height 1). The root is a CL internal node at slot 3, with two separator keys (50, 100) and three child slot indices pointing to CL leaf sub-nodes. Slot indices are `uint8_t` values in the range 1–63.

6.2 Search

Searching within a page traverses the CL sub-tree from root to CL leaf:

1. Start at the sub-tree root slot (`header.root_slot`).

2. At each CL internal node, perform an SIMD-accelerated scan of the separator keys (up to 12 keys, scanned 4 at a time via `_mm_cmpgt_epi32`). This yields the child index c_i; follow `children[`c_i`]` to the next slot.
3. Repeat for `sub_height` levels until reaching a CL leaf.
4. In the CL leaf, perform an SIMD predecessor scan of the sorted keys (up to 15 keys, scanned 4 at a time). Return the largest key $\leq q$.

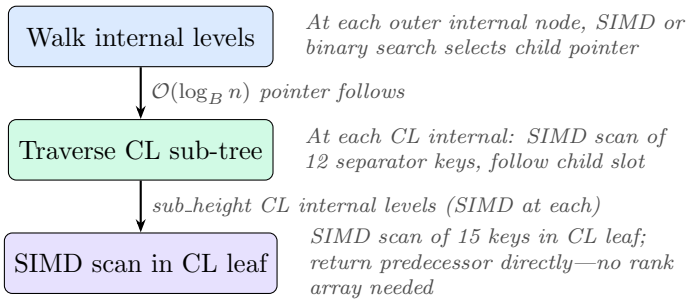

Figure 10: Search procedure. The search is $\mathcal{O}(\log_B n + \text{sub\_height})$, where each sub-tree level does $\mathcal{O}(1)$ SIMD work (scanning ≤ 15 keys).

Because CL leaves store keys in sorted order, the predecessor is found directly—there is no need for a separate `sorted_rank[]` array or sorted-key extraction, as was required by the old FAST-blocked design.

If the predecessor is not in the current CL leaf (i.e., the query is smaller than all keys in this leaf), the search walks left through the sub-tree path to find the rightmost key of the preceding CL leaf.

6.3 Insert

Insertion into a page proceeds as follows:

1. **Navigate**: Traverse the CL sub-tree to the target CL leaf.
2. **Insert into CL leaf**: If the CL leaf has room (< 15 keys), binary-search for the insertion point, shift keys right (`memmove` of at most 60 B), and insert. Done.
3. **CL leaf full**: Split the CL leaf. Allocate a new CL slot from the page bitmap. Move the upper half of keys to the new CL leaf. The separator (first key of the right half) is pushed up to the parent CL internal.
4. **CL internal full**: If the parent CL internal has no room ($= 12$ separator keys), split it too. The median separator is promoted further up.
5. **Sub-tree root split**: If the split propagates to the sub-tree root, allocate a new CL internal as the new root, increasing `sub_height` by 1.
6. **Page full**: If no CL slots are available for the split, return `MT_PAGE_FULL`. The outer tree handles this by splitting the entire page.

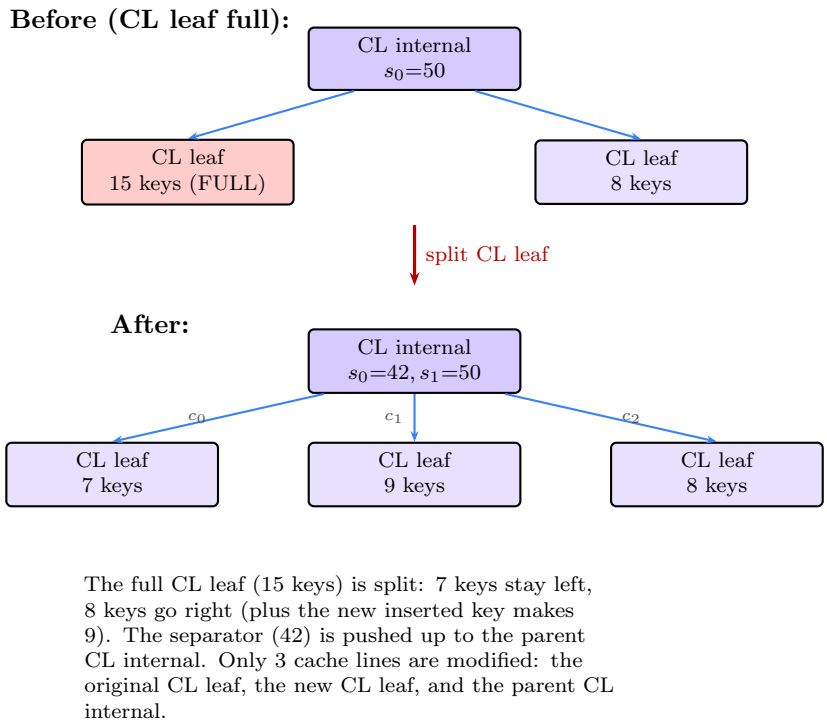

Figure 11: CL leaf split during insert. When a CL leaf overflows, it is split and the separator is promoted to the parent CL internal node. All operations are within the same 4 kB page.

Cost. Inserting into a CL leaf costs $\mathcal{O}(15)$ (shift at most 15 keys). A CL leaf split touches 2 CL leaves and 1 CL internal. In the worst case, a split propagates through `sub_height` CL internal levels, each requiring $\mathcal{O}(12)$ key shifts. Total per-page insert cost: $\mathcal{O}(\text{sub\_height} \times 15)$.

6.4 Delete

Deletion mirrors insertion, with underflow handling:

1. **Navigate and remove**: Traverse to the CL leaf, delete the key (`memmove` left).
2. **Check underflow**: If the CL leaf has ≥ 7 keys ($\lfloor 15/2 \rfloor$), done.
3. **Redistribute**: Try borrowing a key from a sibling CL leaf (via the parent CL internal). If a sibling has > 7 keys, move one key and update the parent separator.
4. **Merge**: If no sibling can spare keys, merge with a sibling. Copy all keys into one CL leaf, free the other slot (clear its bitmap bit), and remove the separator from the parent CL internal.
5. **Propagate**: If the parent CL internal underflows (< 7 children), apply redistribute or merge at the CL internal level (key rotation through the grandparent). Continue up to the sub-tree root.
6. **Root collapse**: If the sub-tree root has 0 keys and 1 child after a merge, replace the root with its child, decreasing `sub_height`.
7. **Page underflow**: If the total key count falls below the page minimum, return `MT_UNDERFLOW`. The outer tree handles page-level redistribute or merge.

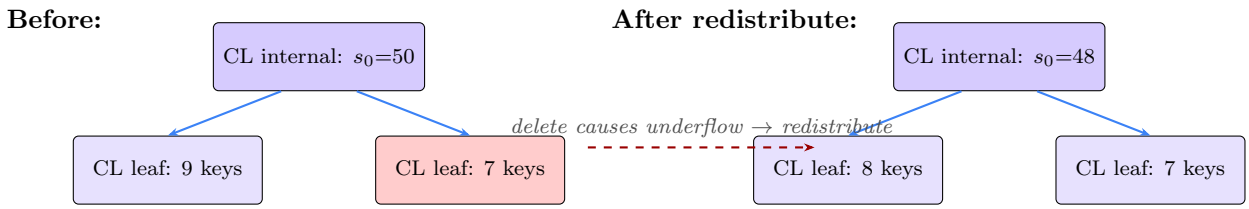

Figure 12: CL leaf redistribute during delete. After deletion, the right CL leaf has 6 keys (below minimum of 7). One key is moved from the left sibling (9 keys, above minimum), and the parent separator is updated. Only 3 cache lines are touched.

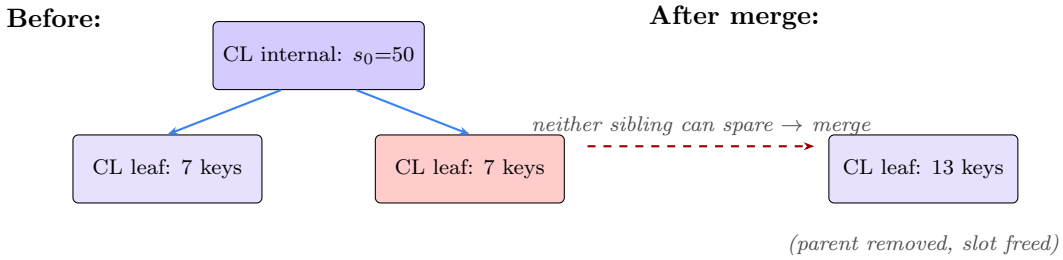

Figure 13: CL leaf merge during delete. Both CL leaves are at minimum occupancy (7 keys each). They are merged into a single CL leaf (13 keys), the other slot is freed back to the bitmap, and the separator is removed from the parent CL internal.

6.5 Page Split

When a page runs out of free CL slots, it must be split:

1. **Extract**: Perform an in-order traversal of the CL sub-tree to extract all keys in sorted order.
2. **Divide**: Split the sorted keys at the midpoint.
3. **Bulk load**: Rebuild each half into a fresh page via `mt_page_bulk_load`, which fills CL leaves sequentially and builds CL internal levels bottom-up.
4. **Return separator**: The first key of the right page becomes the separator pushed to the outer B^+ tree parent.

Page split is the only operation that touches all keys in a page ($\mathcal{O}(n)$ where n is the page key count). It occurs only when the page is completely full (all 63 CL slots allocated), which is infrequent during normal operation.

6.6 Bulk Load

Bulk loading constructs an optimal CL sub-tree from a sorted key array:

1. **Partition**: Divide keys evenly across CL leaves, each holding up to 15 keys.
2. **Fill CL leaves**: Allocate a CL slot for each leaf, copy keys directly.
3. **Build internal levels**: Group CL leaves into parents of up to 13 children each. If more than one parent is needed, repeat at the next level until a single root remains.
4. **Set header**: Record `root_slot`, `sub_height`, `nkeys`, and `nslots_used`.

This produces a balanced, densely packed sub-tree in $\mathcal{O}(n)$ time.

7 Outer B$^+$ Tree Operations

The outer B$^+$ tree uses standard pointer-based navigation, with leaf pages as its leaf nodes and conventional internal nodes (`mt_inode_t`).

7.1 Bulk Load

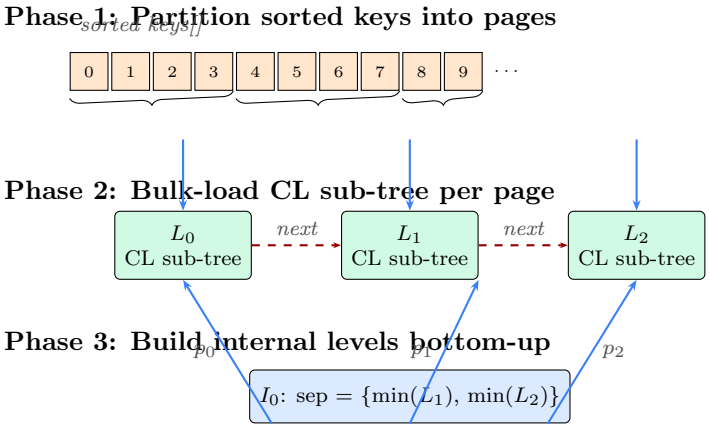

Figure 14: Bulk load construction. Phase 1: Sorted keys are partitioned across pages. **Phase 2**: Each page's CL sub-tree is bulk-loaded via `mt_page_bulk_load`. Pages are linked via `prev`/`next`. **Phase 3**: Internal nodes are built bottom-up with separator keys (the minimum key of each non-first child page).

7.2 Point Query and Predecessor Search

A predecessor search for query q combines outer tree traversal with intra-page sub-tree search:

1. Walk from the root through internal levels ($\mathcal{O}(\log_B n)$ pointer follows, with SIMD search at each internal node).
2. At the target leaf page, call `mt_page_search_key`, which traverses the CL sub-tree (`sub_height` levels of SIMD scans) to find the predecessor.
3. If the predecessor is not found in this page (the query is smaller than all keys in this page), the outer tree follows the `prev` pointer to check the preceding leaf.

7.3 Insert

1. **Find leaf page**: Walk from root to the target page, recording the path.
2. **Page insert**: Call `mt_page_insert`, which navigates the CL sub-tree and inserts into the appropriate CL leaf, splitting CL sub-nodes as needed.
3. **If** `MT_OK`: Done. Only individual CL sub-nodes were modified.
4. **If** `MT_DUPLICATE`: Return false (key exists).
5. **If** `MT_PAGE_FULL`: The page has no free CL slots. Split the entire page via `mt_page_split`: extract all keys, divide in half, bulk-load each half. Push the separator to the parent internal node. If the parent overflows, split it too (standard B$^+$ tree internal split).

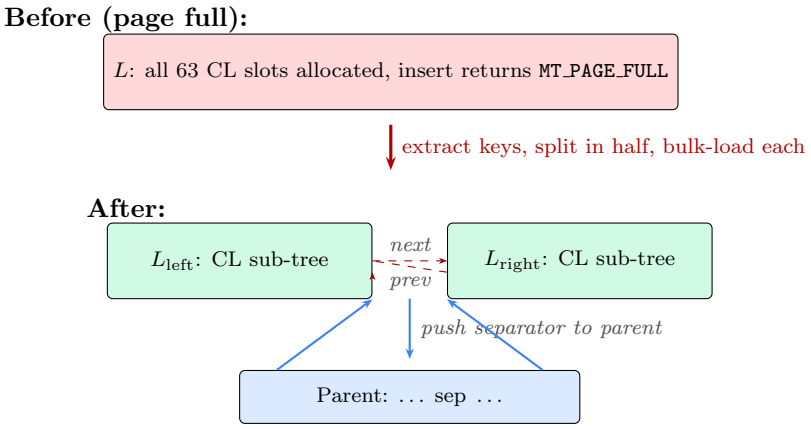

Figure 15: Page split during insertion. When a page is full and cannot accommodate a CL sub-node split, all keys are extracted, divided in half, and each half is bulk-loaded into a fresh page. The separator is pushed to the parent internal node.

7.4 Delete

1. **Find and remove**: Walk from root to page. Call `mt_page_delete`, which navigates the CL sub-tree, deletes from the CL leaf, and handles CL-level redistribute/merge.
2. **If** `MT_OK`: Done.
3. **If** `MT_NOT_FOUND`: Return false.
4. **If** `MT_UNDERFLOW`: The page has too few keys. Try redistributing keys from a sibling page (extract both pages' sorted keys, redistribute evenly, bulk-load both). If redistribution is not possible, merge with a sibling (combine keys, bulk-load one page, free the other). Remove the separator from the parent. Propagate upward if the parent underflows.
5. **Root collapse**: If the root internal node has 0 keys and 1 child, replace it with its child.

7.5 Range Scan (Iteration)

An iterator is positioned at a leaf page and extracts its sorted keys into a buffer (via in-order traversal of the CL sub-tree). Advancing yields keys from the buffer; when exhausted, the iterator follows the `next` pointer and extracts the next page's keys.

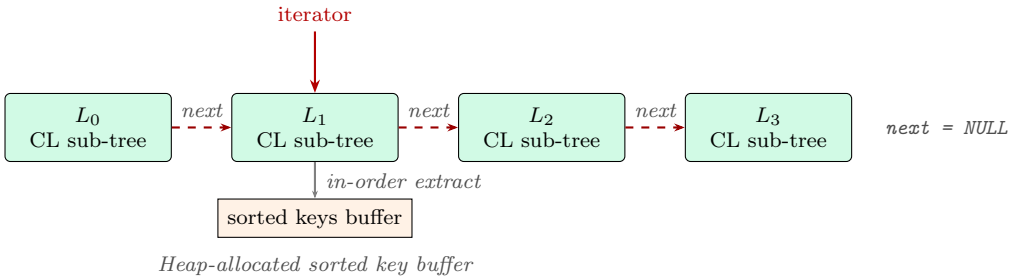

Figure 16: Iterator traversal via leaf linked list. The iterator extracts a page's sorted keys by in-order traversal of its CL sub-tree, then yields keys from the buffer. When exhausted, it follows `next` to the adjacent page.

8 Complexity Analysis

Table 4 summarises the asymptotic costs of matryoshka tree operations. Let n denote the total key count, B the maximum keys per page (approximately 855 at sub-height 2), $b = 15$ the CL leaf capacity, $f = 13$ the CL internal fanout, $F = 340$ the outer internal fanout, h_s the intra-page sub-tree height, and h_o the outer tree height.

Table 4: Operation costs. $h_o = \mathcal{O}(\log_F(n/B))$, $h_s \leq 2$ for 4 kB pages.

Operation	Asymptotic Cost	Notes
Bulk load	$\mathcal{O}(n)$	Bottom-up construction
Point search	$\mathcal{O}(h_o + h_s)$	h_o pointer follows + h_s CL internal scans + 1 CL leaf scan
Predecessor	$\mathcal{O}(h_o + h_s)$	Same as point search (no rank array needed)
Insert (no split)	$\mathcal{O}(h_o + h_s \cdot b)$	Navigate + CL operations
Insert (page split)	$\mathcal{O}(B + h_o)$	Extract + rebuild both halves + parent update
Delete (no merge)	$\mathcal{O}(h_o + h_s \cdot b)$	Navigate + CL operations
Delete (page merge)	$\mathcal{O}(B + h_o)$	Extract + rebuild + parent update
Range scan	$\mathcal{O}(h_o + m)$	h_o for initial seek, m keys scanned
Iteration (next)	$\mathcal{O}(1)$ amortised	$\mathcal{O}(B)$ per page transition (re-extract)

Concrete numbers. For $n = 10{,}000{,}000$ keys with pages at sub-height 2 ($B \approx 855$):

- Pages: $\lceil 10^7/855 \rceil \approx 11{,}700$ leaf pages.
- Outer tree height: $h_o = \lceil \log_{340}(11{,}700) \rceil \approx 2$.
- Sub-tree height: $h_s = 2$ (1 root CL internal + up to 5 CL internals + 57 CL leaves).
- Total cache lines per search: 2 outer internal nodes (1 CL each) + 3 CL sub-nodes (root + internal + leaf) = 5 cache-line accesses.

Comparison with old design. The old FAST-blocked flat array design required $\mathcal{O}(B) = \mathcal{O}(511)$ work per insertion or deletion (full extract, sort, and FAST-layout rebuild). The new CL sub-tree design requires $\mathcal{O}(h_s \times b) = \mathcal{O}(2 \times 15) = \mathcal{O}(30)$ in the common case (no page split). This is a roughly **15–20× reduction** in per-modification cost, at comparable search performance.

9 Cache Behaviour Analysis

The matryoshka tree's performance depends critically on how its data structures interact with the CPU cache hierarchy.

9.1 Within a Page (CL Sub-Tree)

Each CL sub-node is exactly 64 B—one cache line. Navigating the CL sub-tree from root to leaf touches exactly $h_s + 1$ cache lines (one per CL internal level plus the CL leaf). For sub-height 2, this is 3 cache lines, all within the same 4 kB page.

Within each CL sub-node, the SIMD scan processes 4 keys per SSE2 comparison. A CL leaf of 15 keys requires $\lceil 15/4 \rceil = 4$ SIMD comparisons in the worst case. A CL internal of 12 separators requires $\lceil 12/4 \rceil = 3$ comparisons.

Because all CL sub-nodes reside in the same page, they share the same TLB entry. Traversing the CL sub-tree incurs zero TLB misses after the initial page access.

9.2 Between Pages (Outer B$^+$ Tree)

Each pointer follow between outer tree nodes may incur a TLB miss if the target resides on a different virtual page. Since nodes are page-aligned, each node occupies exactly one TLB entry. The TLB cost per search is bounded by $h_o + 1$ (one per internal level plus the leaf page), typically 2–3 for trees up to 10^8 keys.

9.3 Comparison with Old Design

The old FAST-blocked design stored up to 511 keys in a flat array within each leaf, requiring $\lceil 9/2 \rceil = 5$ SIMD block traversals per search (9 levels of binary tree, 2 levels per SIMD block). The new CL sub-tree design requires $h_s + 1 = 3$ cache-line accesses with SIMD scans of 12–15 keys each. The number of SIMD *instructions* is comparable (5 old vs. 3–4 new per sub-tree level), but each new access touches a full cache line worth of sorted keys rather than a 3-key SIMD block, giving better data density per cache-line fetch.

9.4 Comparison with Pure FAST

Pure FAST trees eliminate all pointer follows by storing the entire index in a contiguous array. For trees fitting in L2/L3 cache, FAST achieves higher raw search throughput. The matryoshka tree trades a small number of pointer follows ($h_o = 1$–2 per search) for the ability to modify the tree efficiently—$\mathcal{O}(h_s \times b)$ per insert/delete versus $\mathcal{O}(n)$ for FAST's full rebuild.

10 Implementation Notes

10.1 Arena Allocator

Leaf pages are allocated from a 2 MiB arena allocator (`mt_allocator_t`). Each arena is a superpage-sized region:

```
/* Try huge page allocation first. */
void *p = mmap(NULL, arena_size, PROT_READ | PROT_WRITE,
               MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
if (p == MAP_FAILED) {
    /* Fallback: aligned allocation + transparent huge page hint. */
    posix_memalign(&p, arena_size, arena_size);
    madvise(p, arena_size, MADV_HUGEPAGE);
}
```

Each arena is subdivided into page-sized slots tracked by a `uint64_t` bitmap (one bit per page). Allocation scans the bitmap for a free bit via `__builtin_ctzll`; deallocation clears the bit. When all arenas are full, a new arena is allocated.

Co-locating leaf pages within the same superpage-aligned arena reduces TLB misses during sequential scans: pages allocated during bulk load or adjacent inserts end up in the same 2 MiB region, covered by a single TLB entry.

Internal nodes are allocated via `posix_memalign` with 4096 B alignment (one page each, outside the arena).

10.2 Slot Allocation within a Page

CL slot allocation within a leaf page uses the `slot_bitmap` field in the page header. Bit 0 is always set (it represents the header slot). Bits 1–63 represent CL sub-node slots:

```
/* Allocate a CL slot: find lowest clear bit in range 1-63. */
uint64_t avail = ~page->header.slot_bitmap & ~1ULL;
if (avail == 0) return 0;  /* page full */
int slot = __builtin_ctzll(avail);
page->header.slot_bitmap |= (1ULL << slot);
```

Deallocation clears the corresponding bit. This provides $\mathcal{O}(1)$ allocation and deallocation.

10.3 Compilation Constants

Table 5: Compile-time constants defined in `matryoshka_internal.h`.

Constant	Value	Description
`MT_PAGE_SIZE`	4096	Page size in bytes
`MT_CL_SIZE`	64	Cache-line size in bytes
`MT_CL_KEY_CAP`	15	Keys per CL leaf
`MT_CL_MIN_KEYS`	7	Minimum CL leaf occupancy ($\lfloor 15/2 \rfloor$)
`MT_CL_SEP_CAP`	12	Separator keys per CL internal
`MT_CL_CHILD_CAP`	13	Children per CL internal
`MT_CL_MIN_CHILDREN`	7	Minimum CL internal children ($\lceil 13/2 \rceil$)
`MT_PAGE_SLOTS`	63	Usable CL slots per page (slots 1–63)
`MT_MAX_IKEYS`	339	Max keys per outer internal node
`MT_MIN_IKEYS`	169	Min keys per outer internal node ($\lfloor 339/2 \rfloor$)
`MT_MAX_LEVELS`	8	Maximum hierarchy levels
`MT_KEY_MAX`	$2^{31} - 1$	Sentinel value (`INT32_MAX`)

10.4 Runtime Hierarchy Configuration

The `mt_hierarchy_t` structure stores per-tree configuration:

Table 6: Runtime hierarchy fields in `mt_hierarchy_t`.

Field	Default	Description
`leaf_alloc`	4096	Leaf page allocation size
`cl_key_cap`	15	Keys per CL leaf
`cl_sep_cap`	12	Separator keys per CL internal
`cl_child_cap`	13	Children per CL internal
`page_slots`	63	Usable CL slots per page
`page_max_keys`	$\sim$855	Max keys per page (sub-height dependent)
`min_page_keys`	varies	Minimum page occupancy for outer tree
`min_cl_keys`	7	Minimum CL leaf occupancy
`min_cl_children`	7	Minimum CL internal children

10.5 Limitations and Future Work

- **Concurrency**: No locking or versioning is implemented. Optimistic lock coupling [6] or epoch-based reclamation would be needed for concurrent access.

- **Variable-length keys**: The current design is specialised for 32-bit integers. Supporting variable-length keys would require indirection or key normalisation, similar to Masstree's approach.
- **Superpage-level nesting**: The current design nests only two levels (CL sub-tree within page, pages within outer tree). A future extension could add a third nesting level: a B^+ sub-tree of pages within a 2 MiB superpage, further reducing TLB misses for very large datasets.
- **Architecture-specific tuning**: Factory functions currently target x86-64 with SSE2. Adding presets for ARM NEON, AVX2 (8 keys per comparison), and AVX-512 (16 keys per comparison) would broaden applicability and increase the effective SIMD scan width.

11 Summary

The matryoshka tree is a B^+ tree in which each leaf page contains a B^+ sub-tree of cache-line-sized sub-nodes. This matryoshka nesting—trees within trees at every level of the memory hierarchy—delivers both efficient search and efficient modification:

1. **SIMD-accelerated intra-page search**: The CL sub-tree is navigated via SIMD scans of 12–15 keys at each level, touching $h_s + 1$ cache lines (typically 2–3) per leaf page.
2. **Per-key modifications without flat-array rebuild**: Insert and delete operate on individual CL sub-nodes—splitting, merging, and redistributing cache-line-sized structures—costing $\mathcal{O}(h_s \times 15)$ per modification versus the old $\mathcal{O}(511)$ flat rebuild. This is a $\sim$15–20$\times$ improvement.
3. **Standard B^+ tree outer structure**: The outer tree uses pointer-based navigation with page-level splits and merges, supporting $\mathcal{O}(\log_B n)$ structural modifications.
4. **Eager deletion at every level**: Jannink's redistribute/merge algorithm operates at both the CL sub-tree level and the outer tree level, maintaining minimum occupancy invariants throughout.
5. **Arena allocation with huge pages**: Leaf pages are co-located in 2 MiB arenas for TLB locality.
6. **Efficient range scans**: A doubly-linked leaf list enables sequential iteration; an in-order traversal of each page's CL sub-tree extracts sorted keys.

The nested architecture—SIMD registers inside cache-line sub-nodes inside page-sized sub-trees inside the outer B^+ tree—mirrors the memory hierarchy itself, giving the structure its name: like a matryoshka doll, each level contains a smaller, self-similar structure within.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[3] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[4] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B^+-trees. In *Proc. ACM SIGMETRICS*, pages 283–294, 2003.

[5] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. ACM SIGMOD*, pages 339–350, 2010.

[6] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems*, 6(4):650–670, 1981.

[7] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. IEEE ICDE*, pages 38–49, 2013.

[8] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. ACM EuroSys*, pages 183–196, 2012.

[9] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proc. VLDB*, pages 78–89, 1999.

[10] J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In *Proc. ACM SIGMOD*, pages 475–486, 2000.

[11] B. Schlegel, R. Gemulla, and W. Lehner. K-ary search on modern processors. In *Proc. DaMoN Workshop*, pages 52–60, 2009.

[12] J. Jannink. Implementing deletion in B^+-trees. *ACM SIGMOD Record*, 24(1):33–38, 1995.

[13] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. ACM SIGMOD*, pages 145–156, 2002.