Matryoshka Trees

A Modification-Friendly, Cache-Conscious
B^+ Tree with Parametrisable Hierarchical SIMD Blocking

Design Document

February 20, 2026

This document describes the design of the *matryoshka tree*, a B^+ tree variant that embeds FAST-style hierarchical blocking [5] within each leaf node while retaining the standard B^+ tree's support for efficient insertions, deletions, and range scans. The blocking hierarchy is *parametrisable at runtime*: the number of blocking levels, their depths, and the leaf allocation size (from 4 kB pages to 2 MiB superpages) are configured per-tree via an `mt_hierarchy_t` descriptor. Deletion uses Jannink's eager algorithm with redistribute and merge, propagating rebalancing up to the root. Leaf nodes are co-located within superpage-aligned arenas for TLB locality. The name reflects the nested ("Russian doll") architecture: SIMD blocks nest inside cache-line blocks, which nest inside page-sized or superpage-sized B^+ nodes, which compose the full tree.

Contents

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 matryoshka tree reconciles these by restricting FAST-style blocking to *within* each B$^+$ node (specifically, within leaf nodes), and using standard pointer-based navigation *between* nodes. The result is:

- **Fast intra-node search**: Each leaf contains up to B keys (511 for 4 kB pages, 262143 for 2 MiB superpages) in a FAST-blocked layout, searched via SIMD in $\lceil d_N/d_K \rceil$ SSE operations.
- **Parametrisable hierarchy**: The number of blocking levels, their depths, and the leaf allocation size are configured per-tree at runtime, enabling architecture-specific tuning (e.g. SPARC's 8 kB base pages with power-of-8 superpages).
- **Eager deletion**: Jannink's redistribute/merge algorithm maintains minimum leaf occupancy invariants, with cascading rebalancing through internal nodes up to the root.
- **Arena allocation**: Leaf nodes are co-located within superpage-aligned arenas (via `mmap(MAP_HUGETLB)` on Linux), reducing TLB misses during tree traversal.
- **Efficient modifications**: Insertions and deletions rebuild only the affected leaf ($\mathcal{O}(B)$ work) and propagate splits or merges up $\mathcal{O}(\log_B n)$ internal levels, for overall $\mathcal{O}(B \cdot \log_B n)$ cost.
- **Range scans**: A doubly-linked leaf list supports efficient sequential iteration across leaves.

The name "matryoshka" reflects the nested architecture:

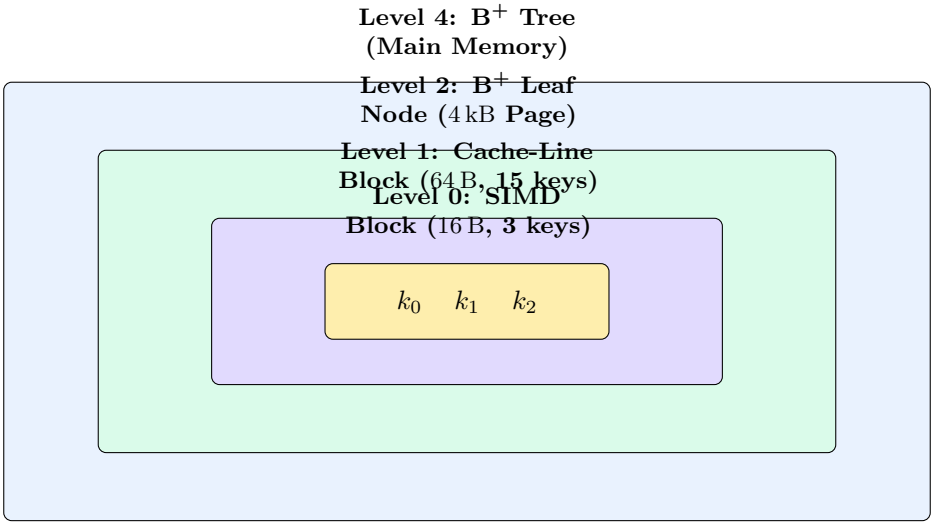


Figure 1: Nested blocking hierarchy. The matryoshka architecture nests SIMD register–sized blocks (**Level 0**, gold) inside cache-line–sized blocks (**Level 1**, violet), which reside inside page-sized B^+ leaf nodes (**Level 2**, green), composing the full B^+ tree (**Level 4**, blue). Within each nesting level, navigation uses offset arithmetic (no pointers); between B^+ nodes, navigation uses explicit child pointers.

2 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 FAST/B-tree.

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 FAST-style blocking is most effective.

Positioning of the matryoshka tree. The matryoshka tree occupies a specific niche: it provides the FAST tree's intra-node search efficiency while supporting B$^+$-tree-style modifications. Table 1 summarises the trade-offs.

Table 1: Comparison of index structures for sorted integer keys. n = number of keys, B = node capacity, d_K = SIMD block depth.

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	$\mathcal{O}(\log_B n \cdot d_N/d_K)$	$\mathcal{O}(B \cdot \log_B n)$	linked list	yes

3 Architecture Overview

A matryoshka tree is a B$^+$ tree with two node types and a runtime-configurable blocking hierarchy:

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 store keys in a FAST-style hierarchically blocked layout. Leaf size is determined by the hierarchy configuration: 4 kB pages (511 keys, `int16_t` rank) or 2 MiB superpages (262143 keys, `int32_t` rank). Leaves are doubly linked for range scans.

Hierarchy configuration (`mt_hierarchy_t`) specifies the blocking levels (finest to coarsest), each with a tree depth and hardware unit size. Factory functions provide defaults for x86-64 (SIMD + cache-line, 4 kB pages), x86-64 with superpages (SIMD + cache-line + page, 2 MiB), and custom architectures. The hierarchy is stored per-tree and governs leaf build, search, and allocation.

Figure 2 shows the overall tree structure.

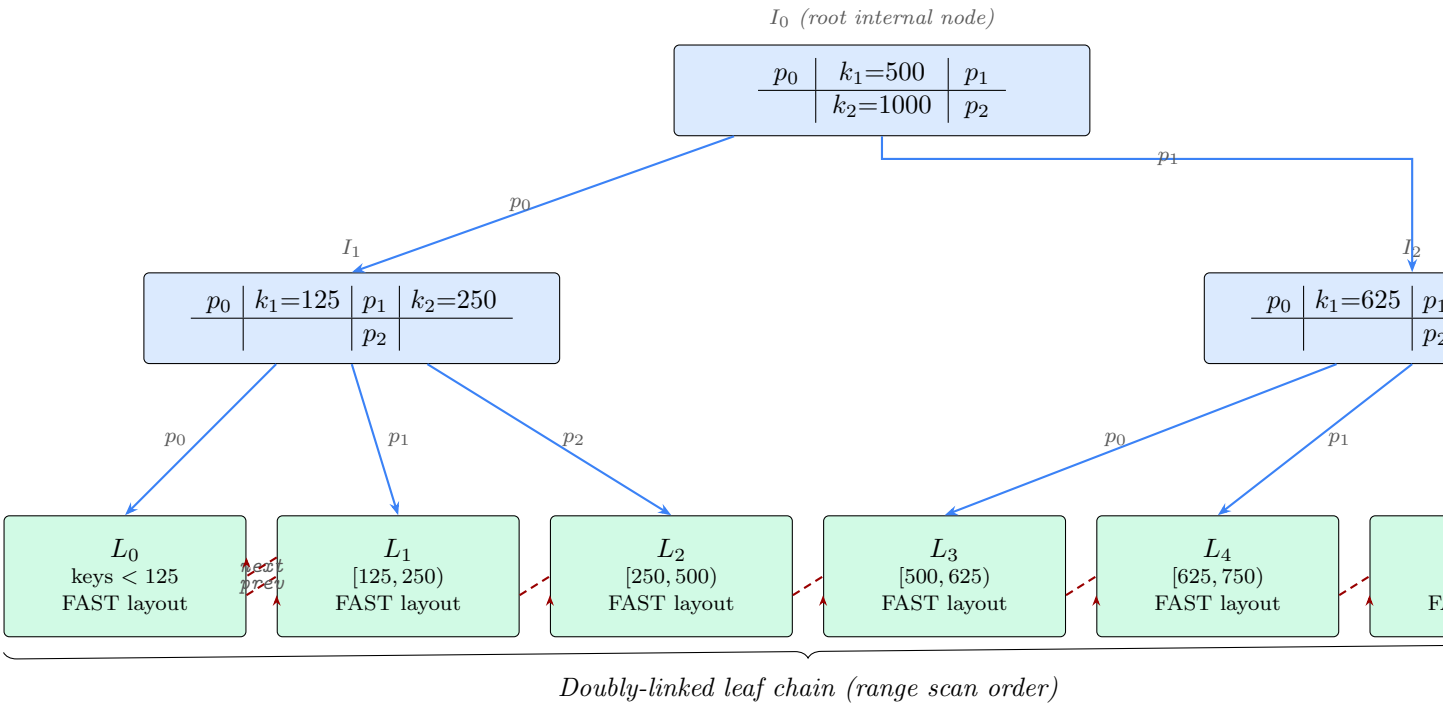


Figure 2: Overall matryoshka tree structure. Internal nodes I_0–I_2 (blue) contain sorted separator keys (k_1, k_2) and child pointers (p_0, p_1, p_2). Leaf nodes L_0–L_5 (green) contain FAST-blocked key layouts. Red dashed arrows show the doubly-linked leaf list (`next`/`prev` pointers) used for range scans. Blue solid arrows show child pointers followed during top-down search. The tree has height 2: two internal levels, one leaf level.

4 Node Structures

Internal nodes are allocated as page-aligned 4 kB buffers via `posix_memalign`. Leaf nodes are allocated from a *superpage arena allocator* (`mt_allocator_t`), which sub-divides superpage-aligned regions (allocated via `mmap(MAP_HUGETLB)` on Linux, with `posix_memalign` fallback) into page-sized slots tracked by a bitmap. Co-locating leaves within the same superpage reduces TLB misses during sequential scans and tree traversals.

4.1 Internal Node (`mt_inode_t`)

Internal nodes store sorted keys and child pointers in a conventional B$^+$ layout:

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

where $k_1 < k_2 < \cdots < k_m$ are separator keys, and child pointer p_i leads to the subtree containing keys in the range $[k_i, k_{i+1})$ (with p_0 covering $(-\infty, k_1)$ and p_m covering $[k_m, +\infty)$).

The maximum number of keys per internal node is determined by the page budget:

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

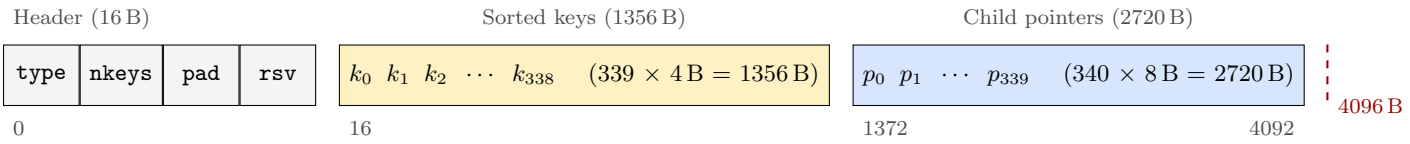


Figure 3: Internal node (`mt_inode_t`) memory layout. The header (gray, 16 B) contains the node type discriminant (`type`=`MT_NODE_INTERNAL`), key count (`nkeys`), and padding. The sorted key array (gold) holds up to 339 `int32_t` keys. The child pointer array (blue) holds up to 340 `mt_node_t*` pointers. Total: $16 + 1356 + 2720 = 4092$ bytes, fitting within one 4 kB page.

Internal node search. For nodes with ≤ 32 keys, `mt_inode_search` uses SIMD-accelerated linear scan: it loads 4 keys per iteration into an SSE register via `_mm_loadu_si128`, compares all 4 against the broadcast query key using `_mm_cmpgt_epi32`, and extracts the first hit via `_mm_movemask_ps` and `__builtin_ctz`. For larger nodes (up to 339 keys), it falls back to scalar binary search. The result is the child index i such that $k_{i-1} \leq key < k_i$, which directly indexes into the `children[]` array.

4.2 Leaf Node (`mt_lnode_t`)

Leaf nodes use a FAST-style blocked layout rather than a simple sorted array. The leaf stores two parallel arrays:

- `layout[512]` (`int32_t`): Keys in FAST-blocked (hierarchically BFS-ordered) positions.
- `sorted_rank[512]` (`int16_t`): For each position i, `sorted_rank[`i`]` gives the index that `layout[`i`]` would have in the sorted key sequence.

The `sorted_rank` array serves as the bridge between the cache-efficient blocked layout (optimised for search) and the logical sorted order (needed for predecessor resolution and key extraction).

The page budget for a leaf is:

$$\underbrace{24}_{\text{header}} + \underbrace{512 \times 4}_{\texttt{layout}} + \underbrace{512 \times 2}_{\texttt{sorted\_rank}} = 3096 \leq 4096 \tag{3}$$

The maximum tree depth within a leaf is $d = 9$ (since $2^9 - 1 = 511$ nodes), giving a maximum key capacity of **511 keys per leaf**.

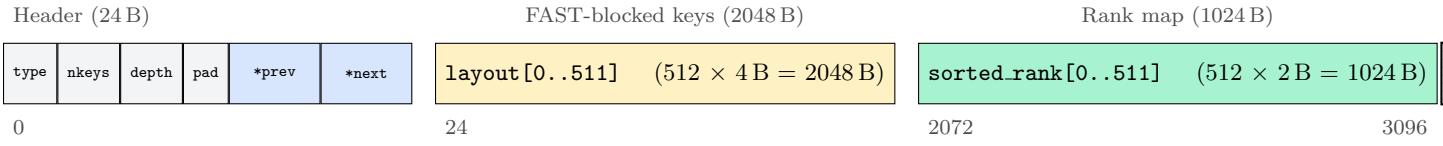


Figure 4: Leaf node (`mt_lnode_t`) memory layout. The header (gray, 24 B) contains the type discriminant, key count, tree depth, padding, and doubly-linked list pointers `*prev`/`*next` (blue). The FAST-blocked key array (gold, 2048 B) stores keys in hierarchically blocked BFS order. The rank map (mint, 1024 B) maps each layout position to its index in sorted order. Total payload: 3096 B; remaining 1000 B within the 4 kB page is unused.

5 FAST Blocking within Leaf Nodes

This section describes how keys within a single leaf node are arranged in the FAST hierarchically blocked layout, following the scheme of Kim et al. [5].

5.1 Background: Complete Binary Search Trees in BFS Order

Given n sorted keys $k_0 < k_1 < \cdots < k_{n-1}$, we first embed them into a complete binary search tree. The tree has depth $d_N = \lceil \log_2(n+1) \rceil$ and $2^{d_N} - 1$ nodes (padding unused positions with $+\infty$).

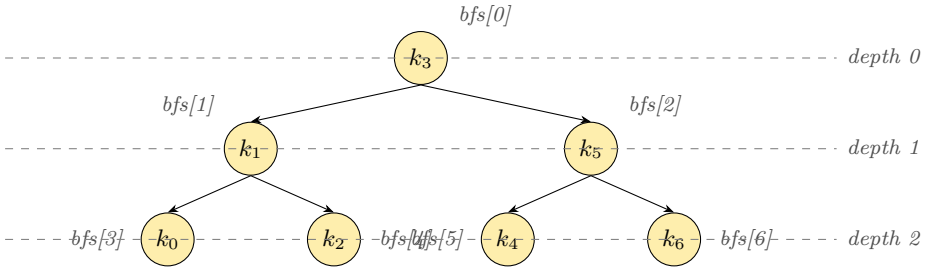


Figure 5: Complete binary search tree with 7 keys in BFS order. Sorted keys $k_0 < k_1 < \cdots < k_6$ are placed so that an in-order traversal recovers the sorted sequence. Each node is labelled with its BFS index (bfs[0] through bfs[6]). The root at bfs[0] holds the median key k_3; left children hold smaller keys, right children hold larger keys. The `bfs_to_sorted` mapping gives: bfs[0]→3, bfs[1]→1, bfs[2]→5, bfs[3]→0, bfs[4]→2, bfs[5]→4, bfs[6]→6.

A standard BFS array stores the tree nodes level by level: the root at index 0, its children at indices 1 and 2, their children at indices 3–6, etc. For a node at BFS index i, its left child is at $2i + 1$ and its right child at $2i + 2$.

5.2 Hierarchical Blocking

Storing keys in plain BFS order does not respect the memory hierarchy: for deep trees, a root-to-leaf traversal touches nodes scattered across many cache lines. FAST solves this by *hierarchically blocking* the BFS tree into nested subtrees, each sized to fit a hardware unit.

The blocking is defined by a sequence of depths:

Level	Depth	Keys per block	Hardware target
0	$d_K = 2$	$N_K = 2^{d_K} - 1 = 3$ keys (12 B)	SSE register (16 B)
1	$d_L = 4$	$N_L = 2^{d_L} - 1 = 15$ keys (60 B)	Cache line (64 B)
2	d_P	$N_P = 2^{d_P} - 1$ keys	Page / Superpage

In the current matryoshka implementation, only SIMD-level blocking ($d_K = 2$) is used within leaf nodes. The entire leaf tree is decomposed into SIMD blocks of 3 keys each. Cache-line blocking ($d_L = 4$) is deferred to a future optimisation—it requires matching changes to the search-side traversal code.

5.3 SIMD Block Layout

Each SIMD block is a complete binary tree of depth $d_K = 2$, containing $N_K = 3$ keys. The keys within a block are stored in BFS order:

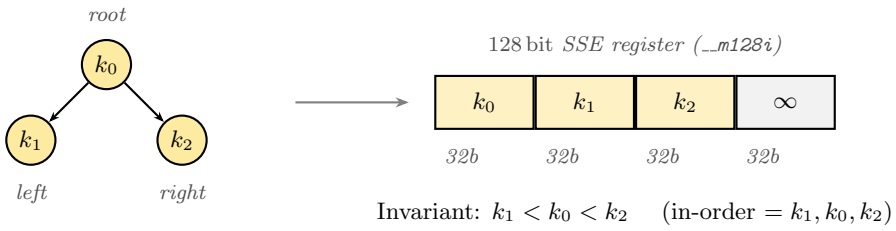
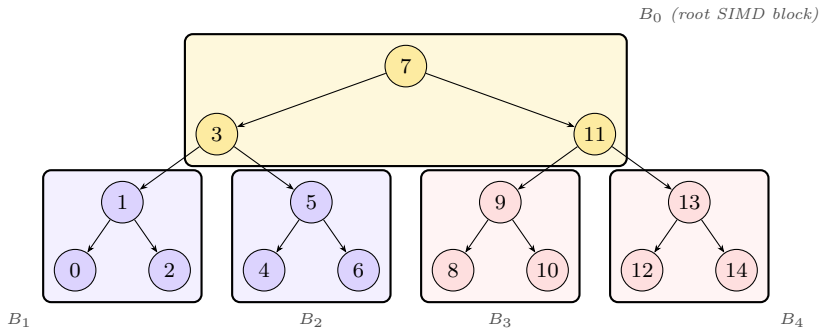


Figure 6: A single SIMD block (depth $d_K = 2$, 3 keys). Left: the block as a complete binary tree with root k_0, left child k_1, and right child k_2. Right: the same 3 keys packed into a 128 bit SSE register in BFS order (k_0, k_1, k_2), with the fourth 32 bit lane filled with $+\infty$ as padding. The in-order (sorted) traversal is $k_1 < k_0 < k_2$.

5.4 Recursive Blocked Layout

The FAST-blocked layout is constructed by recursively decomposing the complete binary search tree into SIMD blocks. The algorithm (`lay_out_subtree`) works as follows:

1. Extract the top $d_K = 2$ levels of the current subtree as a SIMD block (3 keys in BFS order). Write these keys contiguously to the output array.
2. This SIMD block has $2^{d_K} = 4$ child subtrees (at the next level below). Recursively lay out each child subtree in the same fashion, writing the blocks for child 0, then child 1, then child 2, then child 3, contiguously.
3. Base case: when the remaining depth is $\leq d_K$, write the final small block directly.



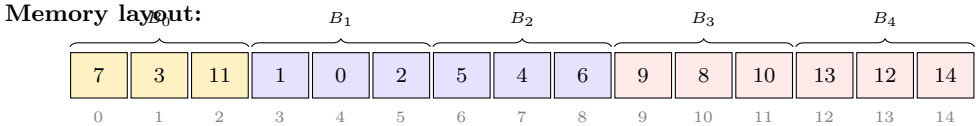


Figure 7: Recursive SIMD-block decomposition of a depth-4 tree (15 keys). Top: the complete binary search tree with keys 0–14 (in-order values). The root SIMD block B_0 (gold, depth 2) contains BFS nodes {7, 3, 11}. Its four child subtrees form blocks B_1 (violet, {1, 0, 2}), B_2 (violet, {5, 4, 6}), B_3 (rose, {9, 8, 10}), and B_4 (rose, {13, 12, 14}). Bottom: the `layout[]` array stores blocks contiguously in pre-order: B_0, B_1, B_2, B_3, B_4. Array indices are shown below each cell.

5.5 The `sorted_rank` Mapping

Because the blocked layout reorders keys from their sorted sequence, we need a way to recover the sorted position of any key encountered during search. The `sorted_rank[]` array provides this:

$$\texttt{sorted\_rank}[i] = j \quad \Longleftrightarrow \quad \texttt{layout}[i] \text{ is the } j\text{-th smallest key in the leaf}$$

This mapping is computed during leaf construction (`mt_leaf_build`) by performing an in-order traversal of the implicit BFS tree and recording the visit count.

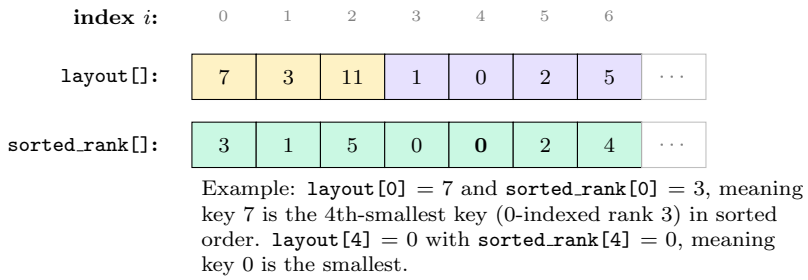


Figure 8: The `sorted_rank` mapping for the first 7 positions of the blocked layout from Figure 7. Each entry `sorted_rank[`i`]` gives the position that `layout[`i`]` occupies in sorted order.

6 SIMD Search Operations

6.1 SIMD Comparison and Child Selection

Within a SIMD block of 3 keys (k_0, k_1, k_2 in BFS order), a predecessor query for key q proceeds as follows:

1. Load the 3 block keys plus a padding ∞ into a 128 bit SSE register: $\mathbf{v}_{\text{tree}} = (k_0, k_1, k_2, \infty)$.
2. Broadcast the query key: $\mathbf{v}_q = (q, q, q, q)$.
3. Compute the element-wise comparison $\mathbf{v}_{\text{cmp}} = \texttt{\_mm\_cmpgt\_epi32}(\mathbf{v}_q, \mathbf{v}_{\text{tree}})$, yielding a 128 bit mask where lane j is all-ones iff $q > k_j$.
4. Extract the sign bits as a 4-bit integer: $mask = \texttt{\_mm\_movemask\_ps}(\mathbf{v}_{\text{cmp}})$.
5. Index into the lookup table `MT_SIMD_LOOKUP[mask & 0x7]` to get the child index $c \in \{0, 1, 2, 3\}$.

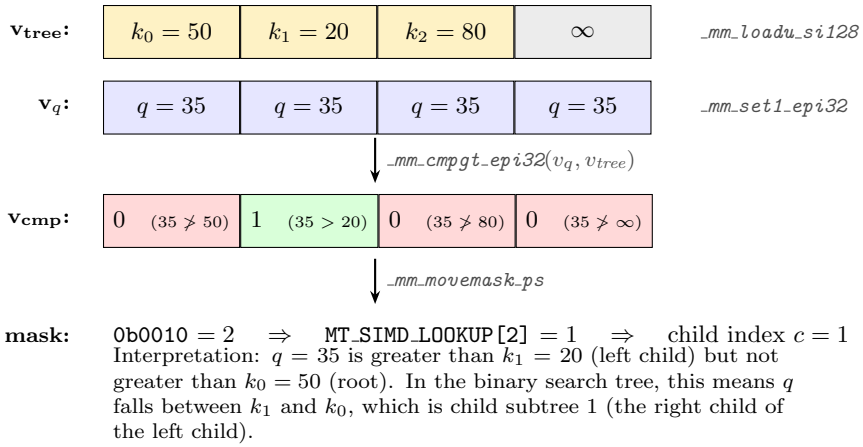


Figure 9: SIMD comparison within one SIMD block. Query $q = 35$ is compared against block keys $(50, 20, 80)$. The `cmpgt` result shows $q > 20$ only (lane 1). The movemask extracts `0b0010`$= 2$, and the lookup table maps this to child index $c = 1$. This selects the second of four child subtrees.

Lookup table derivation. The 3 keys in BFS order form a complete binary tree: root k_0, left child $k_1 < k_0$, right child $k_2 > k_0$. The 3-bit comparison mask encodes which keys the query exceeds:

Mask (bits 2:1:0)	Child	Meaning
`000`	0	$q \leq k_1 < k_0 < k_2$: leftmost subtree
`010`	1	$k_1 < q \leq k_0 < k_2$: second subtree
`011`	2	$k_1 < k_0 < q \leq k_2$: third subtree
`111`	3	$k_1 < k_0 < k_2 < q$: rightmost subtree

Other mask values (e.g., `001`, `100`) are impossible for valid search trees because $k_1 < k_0 < k_2$ always holds.

6.2 Full Leaf Search Path

A search within a leaf traverses a sequence of SIMD blocks from the root block B_0 down to a leaf block. At each block, the SIMD comparison yields a child index c, and the next block's offset is computed as:

$$\mathit{offset}_{\text{next}} = \mathit{offset}_{\text{current}} + N_K + c \cdot (\text{child subtree size}) \tag{4}$$

where the child subtree size is $2^{d_{\text{remaining}}} - 1$ keys.

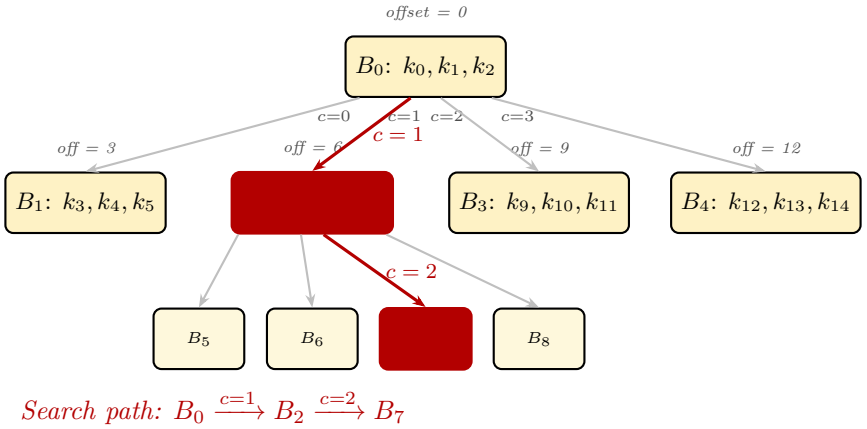


Figure 10: Search path through SIMD blocks within a leaf. Starting at root block B_0 (offset 0), the SIMD comparison yields child index $c = 1$, directing the search to block B_2 (offset 6). A second comparison at B_2 yields $c = 2$, directing to block B_7. The red highlighted path shows the blocks visited. At each step, the next offset is computed as $\mathit{offset} + 3 + c \times$ (child subtree size). Gray arrows show unvisited branches.

6.3 Predecessor Resolution

After the SIMD traversal reaches the bottom of the blocked tree, we must resolve the actual predecessor key. The search terminates at a SIMD block with a child index c that identifies the "gap" in sorted order where the query falls. Using the `sorted_rank[]` mapping, we determine the rank of the boundary key adjacent to this gap, then extract the predecessor from the sorted key sequence.

Specifically, after the final SIMD block at offset o with child index c:

Child c	Predecessor rank
0	`sorted_rank`$[o+1] - 1$ (left of leftmost key)
1	`sorted_rank`$[o+1]$ (between left child and root)
2	`sorted_rank`$[o]$ (between root and right child)
3	`sorted_rank`$[o+2]$ (right of rightmost key)

This rank is then used to index into the sorted key array (reconstructed on-stack via `mt_leaf_extract_sorted`) for the final predecessor value.

7 Tree Operations

7.1 Bulk Load

Bulk loading constructs the tree bottom-up from a pre-sorted array of n keys in $\mathcal{O}(n)$ time.

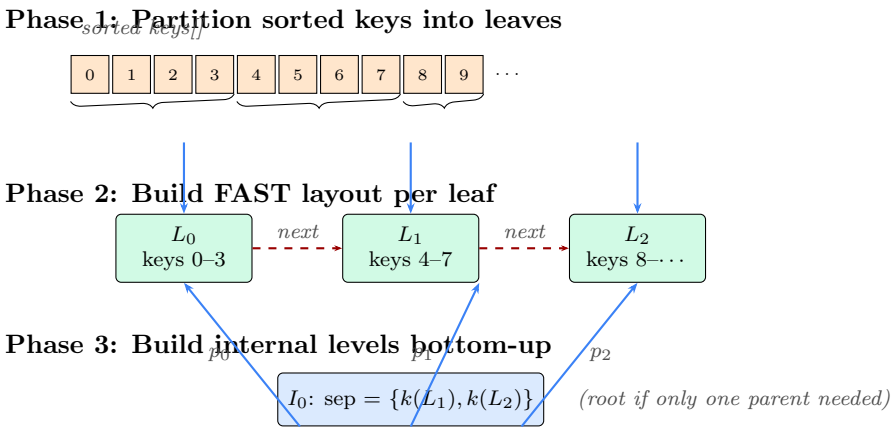


Figure 11: Bulk load construction. Phase 1: The sorted key array is partitioned into groups of up to 511 keys each. **Phase 2**: Each group is passed to `mt_leaf_build`, which constructs the FAST-blocked layout and `sorted_rank` array. Leaves are linked via `prev`/`next` pointers. **Phase 3**: Internal nodes are built bottom-up. Each parent receives separator keys (the minimum key of each non-first child) and child pointers. If more than one parent is needed, the process recurses upward until a single root remains.

The algorithm distributes keys evenly across $\lceil n/511 \rceil$ leaves, builds each leaf's FAST layout via `mt_leaf_build`, links leaves into a doubly-linked list, then iteratively constructs internal levels. At each internal level, child nodes are grouped into parents of up to 340 children (339 separator keys), and the process repeats until a single root remains.

Separator keys are the minimum key of each non-leftmost child, which is tracked during construction via the `build_entry_t` structure (avoiding the cost of walking into each child to find its minimum).

7.2 Point Query and Predecessor Search

A predecessor search for query q finds the largest key $k \leq q$ in the tree:

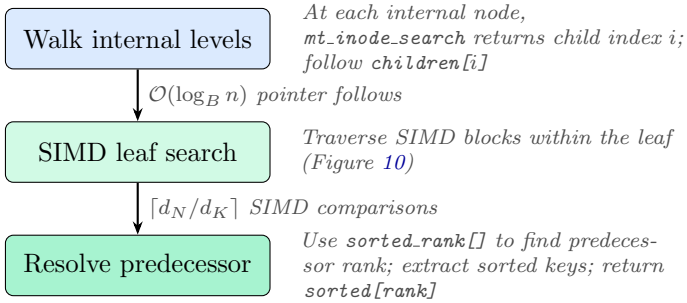
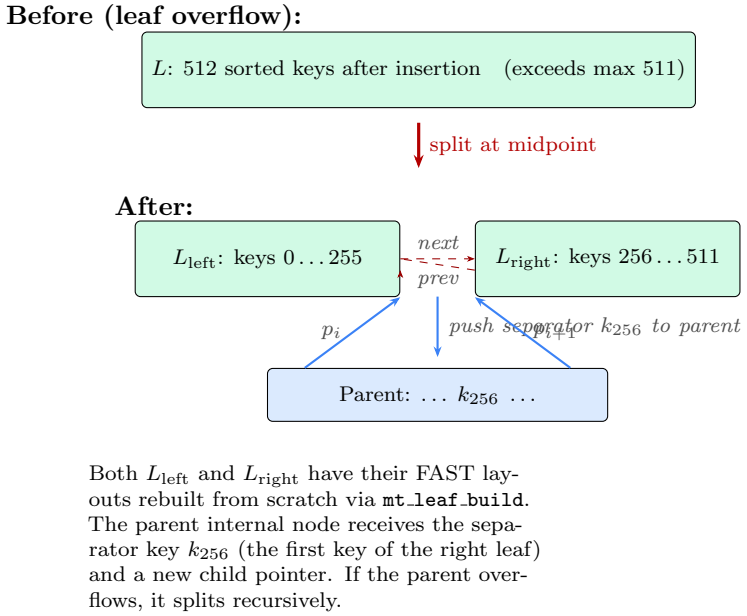


Figure 12: Predecessor search procedure. Three phases: (1) top-down traversal through internal levels using SIMD-accelerated search at each node; (2) SIMD block traversal within the target leaf; (3) predecessor resolution using the rank mapping.

7.3 Insert

Insertion follows the standard B^+ tree procedure, with the key difference that each leaf modification requires a full rebuild of the FAST layout:

1. **Find leaf**: Walk the tree from root to the target leaf, recording the path of internal nodes and child indices in a `mt_path_t` stack.
2. **Extract sorted keys**: Call `mt_leaf_extract_sorted` to recover the logical sorted order from the FAST layout.
3. **Insert into sorted array**: Binary-search for the insertion point and shift elements (rejecting duplicates).
4. **If no overflow** ($n \leq 511$): Rebuild the leaf's FAST layout via `mt_leaf_build`. Done.
5. **If overflow** ($n = 512$): **Split** the leaf.



Both L_{left} and L_{right} have their FAST layouts rebuilt from scratch via `mt_leaf_build`. The parent internal node receives the separator key k_{256} (the first key of the right leaf) and a new child pointer. If the parent overflows, it splits recursively.

Figure 13: Leaf split during insertion. When a leaf exceeds 511 keys, it is split at the midpoint into two leaves. Both leaves' FAST layouts are rebuilt. The separator key (minimum key of the right leaf) is pushed to the parent internal node. The leaf linked list is updated to splice in the new right leaf.

If the parent internal node also overflows (exceeds 339 keys), it splits analogously: the keys and children are divided at the midpoint, with the middle key promoted to the grandparent. If splits propagate all the way to the root, a new root is created with one key and two children, increasing the tree height by 1.

Modification cost. Each insert extracts the leaf's sorted keys ($\mathcal{O}(B)$), inserts into the sorted array ($\mathcal{O}(B)$), and rebuilds the FAST layout ($\mathcal{O}(B)$), where $B = 511$. Splits propagate through at most $\mathcal{O}(\log_B n)$ internal levels, each requiring $\mathcal{O}(B)$ work. Thus the total insert cost is $\mathcal{O}(B \cdot \log_B n)$.

7.4 Delete (Eager, Jannink's Algorithm)

Deletion uses **eager rebalancing** following Jannink [12], maintaining the invariant that every non-root leaf has at least $\lfloor B/2 \rfloor$ keys. The algorithm proceeds as follows:

1. **Find and remove**: Walk from root to leaf (recording the path), extract sorted keys, remove the target key, rebuild the leaf's FAST layout.
2. **Check underflow**: If the leaf is the root or has $\geq \lfloor B/2 \rfloor$ keys, stop.

3. **Redistribute**: Try the left sibling first, then the right. If a sibling has more than $\lfloor B/2 \rfloor$ keys, redistribute keys to balance both leaves (each gets $\lfloor (\text{left} + \text{right})/2 \rfloor$ keys). Update the separator in the parent. Both leaves' FAST layouts are rebuilt.
4. **Merge**: If neither sibling can spare keys, merge with a sibling (prefer left). Combine both leaves' sorted keys into one, rebuild the FAST layout, unlink the empty leaf from the doubly-linked list, remove the separator from the parent, and free the empty leaf.
5. **Propagate upward**: If the parent internal node underflows ($< \lfloor 339/2 \rfloor$ keys), apply the same redistribute-or-merge logic at the internal level. Internal redistribute rotates a separator through the parent: pull a separator key down from the grandparent, push a boundary key up from the sibling. Internal merge combines two nodes with the intervening separator. Repeat up to the root.
6. **Root collapse**: If the root has 0 keys and 1 child, replace the root with its child, decreasing tree height.

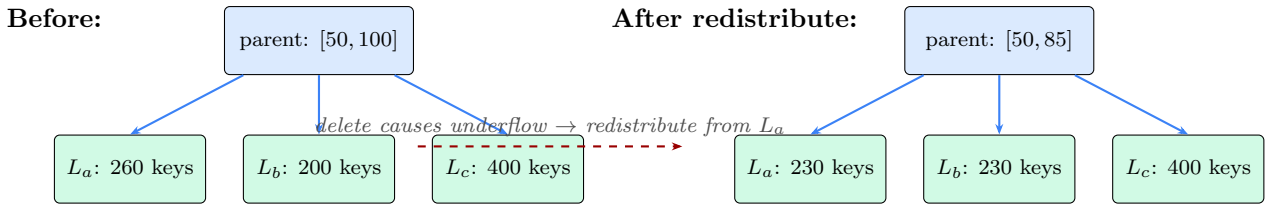


Figure 14: Eager deletion: leaf redistribute. After deleting a key from L_b (200 keys, below the minimum of 255), keys are redistributed from the left sibling L_a (which has 260 spare keys). Both leaves end up with 230 keys each, and the parent separator is updated from 100 to 85 (the new first key of L_b). Both leaves' FAST layouts are rebuilt.

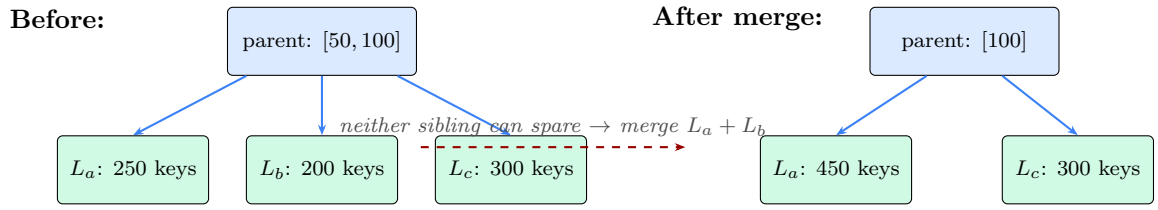


Figure 15: Eager deletion: leaf merge. Neither sibling (L_a with 250 and L_c with 300) can spare keys (both are at or near minimum). L_b is merged into L_a (combined 450 keys), the separator key 50 and L_b's child pointer are removed from the parent, and L_b is freed. If this causes the parent to underflow, the same redistribute-or-merge logic is applied at the internal node level, cascading up to the root.

7.5 Range Scan (Iteration)

Range scans exploit the doubly-linked leaf list. An iterator is initialised by walking from the root to the leaf containing the start key, extracting that leaf's sorted keys into an internal buffer, and positioning at the first key $\geq$ start. Subsequent `iter_next` calls advance through the buffer; when the buffer is exhausted, the iterator follows the `next` pointer to the adjacent leaf, extracts its sorted keys, and continues.

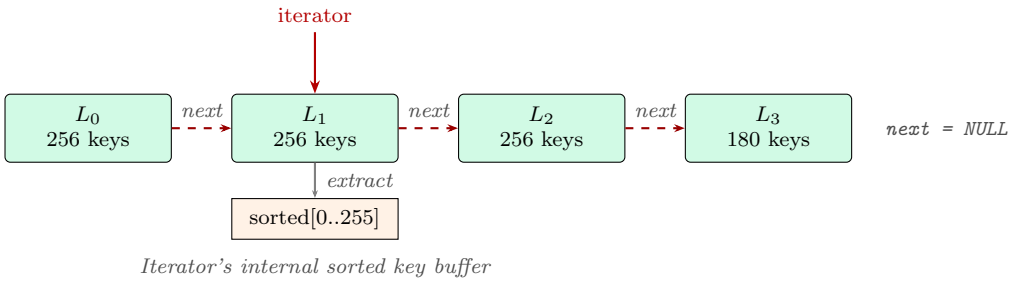


Figure 16: Iterator traversal via leaf linked list. The iterator is positioned at leaf L_1. It has extracted L_1's sorted keys into an internal buffer and yields them one by one. When the buffer is exhausted, the iterator follows L_1.`next` to L_2, extracts its keys, and continues. The red dashed arrows show the `next` pointers forming the linked list.

8 Complexity Analysis

Table 2 summarises the asymptotic costs of matryoshka tree operations, where n is the total number of keys and B is the leaf node capacity (511).

Table 2: Operation costs. $B = 511$ (leaf capacity), $F = 340$ (internal fanout), $h = \lceil \log_F(n/B) \rceil$ (tree height).

Operation	Asymptotic Cost	Notes
Bulk load	$\mathcal{O}(n)$	Bottom-up, one pass
Point search	$\mathcal{O}(h + d_N/d_K)$	h pointer follows + SIMD leaf search
Predecessor	$\mathcal{O}(h + d_N/d_K)$	Same as point search + rank lookup
Insert	$\mathcal{O}(B \cdot h)$ worst case	Leaf rebuild + possible splits
Delete	$\mathcal{O}(B \cdot h)$ worst case	Eager (Jannink): redistribute/merge, cascading
Range scan	$\mathcal{O}(h + m)$	h for initial seek, m keys scanned
Iteration (next)	$\mathcal{O}(1)$ amortised	$\mathcal{O}(B)$ per leaf transition

Concrete numbers. For $n = 10{,}000{,}000$ keys:

- Leaves: $\lceil 10^7/511 \rceil = 19{,}570$ leaves.
- Internal fanout: $F = 340$.
- Height: $h = \lceil \log_{340}(19{,}570) \rceil \approx 2$.
- Leaf tree depth: $d_N = 9$ levels, requiring $\lceil 9/2 \rceil = 5$ SIMD comparisons.
- Total comparisons per search: 2 internal node lookups + 5 SIMD block comparisons = 7 comparison steps.

9 Cache Behaviour Analysis

The matryoshka tree's performance depends critically on how its data structures interact with the CPU cache hierarchy. We analyse cache behaviour at each level.

9.1 Within a Leaf (SIMD Blocks)

Each SIMD block occupies 3 keys $\times$ 4 B = 12 B, fitting entirely within a single 16 B SSE register. The SIMD `loadu` instruction loads the block in one cycle (if the data is in L1). Because the blocked

layout stores parent and child SIMD blocks contiguously, a root-to-leaf traversal within a leaf typically touches at most 2–3 cache lines (for the `layout[]` array) plus 1–2 cache lines (for the `sorted_rank[]` array) during predecessor resolution.

9.2 Between Nodes (B$^+$ Pointers)

Each internal-to-child pointer follow incurs a potential TLB miss if the child 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 the tree height h, which is typically 1–2 for trees up to 10^8 keys.

9.3 Comparison with FAST

FAST trees eliminate all pointer follows by storing the entire index in a contiguous array. For trees that fit in L2/L3 cache, FAST achieves higher throughput because no TLB miss ever occurs. The matryoshka tree trades a small number of pointer follows (1–2 per search) for the ability to modify the tree without rebuilding. For workloads with even moderate insertion rates, the $\mathcal{O}(n)$ rebuild cost of FAST quickly dominates, making the matryoshka tree's $\mathcal{O}(B \cdot \log_B n)$ insert cost advantageous.

10 Implementation Notes

10.1 Memory Allocation

Internal nodes are allocated via `posix_memalign` with 4096 B alignment. Leaf nodes are allocated from a *superpage arena allocator* (`mt_allocator_t`), which manages one or more superpage-aligned regions:

```
/* Arena allocation (Linux) */
mmap(NULL, arena_size, PROT_READ | PROT_WRITE,
     MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
/* Fallback: posix_memalign(&p, arena_size, arena_size) */
```

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

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

The `mt_node_t` union allows type-punning between internal and leaf nodes via a shared `type` field at offset 0.

10.2 Leaf Build Algorithm

The `mt_leaf_build` function constructs the FAST-blocked layout from a sorted key array, parameterised by the tree's `mt_hierarchy_t`:

1. **Compute tree depth**: $d_N = \lceil \log_2(n+1) \rceil$, giving $2^{d_N} - 1$ tree nodes (some may be $+\infty$ padding).
2. **Build in-order map**: An iterative in-order traversal of the implicit BFS tree computes `bfs_to_sorted[i]` = the sorted rank of BFS node i.
3. **Construct BFS tree**: For each BFS node i, set `bfs_tree[`i`] = sorted_keys[bfs_to_sorted[`i`]]`, or $+\infty$ if the rank exceeds n.

4. **Recursive layout**: `lay_out_subtree` decomposes the BFS tree according to the blocking depths from `hier->levels[0..num_levels-1]`, writing each block's keys and `sorted_rank` values contiguously to the output arrays. The recursion starts at the coarsest level and decomposes downward to the finest (SIMD) level, matching the search traversal exactly.

The blocking depths are read from the hierarchy at runtime, not hardcoded. For example, with the default hierarchy (`mt_hierarchy_init_default`), depths are $[d_K{=}2, d_L{=}4]$; with the superpage hierarchy, depths are $[2, 4, 10]$. Zero-fill of unused slots uses `hier->tree_cap` as the bound.

10.3 Limitations and Future Work

- **Per-search key reconstruction**: The current search implementation extracts sorted keys on each query ($\mathcal{O}(B)$ per search) for predecessor resolution. This can be eliminated by directly computing the predecessor during the SIMD traversal, avoiding the reconstruction entirely.
- **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.
- **Cross-node superpage blocking**: The current arena allocator co-locates leaves for TLB locality, but the blocking hierarchy does not extend across B^+ node boundaries. True cross-node superpage-level blocking (grouping multiple leaves into a single contiguous FAST layout) would further reduce search latency for very large datasets.
- **Architecture-specific tuning**: Factory functions currently target x86-64. Adding presets for ARM (NEON), SPARC (8 kB pages), and POWER would broaden applicability.

11 Compilation Constants

Table 3: Compile-time constants defined in `matryoshka_internal.h`.

Constant	Value	Description
`MT_PAGE_SIZE`	4096	Page size in bytes; internal node allocation unit
`MT_DK`	2	SIMD block depth (finest level)
`MT_NK`	3	Keys per SIMD block ($2^{d_K} - 1$)
`MT_DL`	4	Cache-line block depth
`MT_NL`	15	Keys per cache-line block ($2^{d_L} - 1$)
`MT_MAX_IKEYS`	339	Max keys per internal node
`MT_MIN_IKEYS`	169	Min keys per internal node ($\lfloor 339/2 \rfloor$)
`MT_LNODE_TREE_CAP`	512	Array capacity in default leaf ($\geq 2^9 - 1$)
`MT_MAX_LKEYS`	511	Max keys per default leaf ($2^9 - 1$)
`MT_MAX_LEVELS`	8	Maximum blocking hierarchy levels
`MT_MAX_HEIGHT`	32	Maximum tree height (stack depth)
`MT_KEY_MAX`	$2^{31} - 1$	Sentinel value (`INT32_MAX`)

Runtime hierarchy fields. Leaf capacity, minimum leaf keys, tree depth, and rank width are no longer compile-time constants. They are computed per-tree from the `mt_hierarchy_t` configuration:

Table 4: Runtime hierarchy fields in `mt_hierarchy_t`.

Field	Default	Description
`num_levels`	2	Number of blocking levels
`levels[]`	$[\{2, 16\}, \{4, 64\}]$	Per-level depth and hardware size
`leaf_alloc`	4096	Leaf allocation size in bytes
`leaf_depth`	9	Maximum tree depth within a leaf
`leaf_cap`	511	Maximum keys per leaf ($2^{\text{leaf\_depth}} - 1$)
`tree_cap`	512	Array capacity ($2^{\text{leaf\_depth}}$)
`min_lkeys`	255	Minimum keys per non-root leaf ($\lfloor \text{leaf\_cap}/2 \rfloor$)
`rank_wide`	false	True if `leaf_cap` > 32767 (use `int32_t` rank)

12 Summary

The matryoshka tree provides a practical middle ground between static SIMD-optimised indexes (FAST) and traditional pointer-based B$^+$ trees. By confining FAST-style hierarchical blocking to within each leaf node and connecting nodes via standard B$^+$ tree pointers, it achieves:

1. **SIMD-parallel intra-node search** with $\lceil d_N/d_K \rceil$ comparisons per leaf, using the full multi-level blocking hierarchy (SIMD → cache-line → page);
2. **Parametrisable hierarchy**: the number of blocking levels, their depths, and the leaf allocation size (from 4 kB pages to 2 MiB superpages) are configured per-tree via `mt_hierarchy_t`, enabling architecture-specific tuning;
3. **Eager deletion** via Jannink's algorithm, maintaining minimum occupancy invariants through redistribute and merge operations that cascade through internal nodes to the root;
4. **Superpage arena allocation**: leaf nodes are co-located within superpage-aligned arenas (via `mmap(MAP_HUGETLB)` on Linux), reducing TLB misses during tree traversal;
5. **Efficient modifications**: insertions and deletions rebuild only the affected leaf ($\mathcal{O}(B)$ work) and propagate splits or merges up $\mathcal{O}(\log_B n)$ internal levels;
6. **Efficient range scans** via a doubly-linked leaf chain; and
7. **Variable-width rank**: leaves with > 32767 keys automatically use `int32_t` for `sorted_rank`, supporting superpage-sized leaves with up to 262143 keys.

The nested architecture—SIMD blocks inside cache-line blocks inside page-sized or superpage-sized B$^+$ nodes—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.