The Li-Chao Tree: Algorithm Specification and Analysis

Chao Li
chnlich@live.com

March 2, 2026

Abstract

The Li-Chao tree (LICT) was first introduced in lecture materials [2] as an efficient data structure for dynamic lower envelope maintenance. In the years since, it has achieved widespread adoption within the competitive programming community, yet no formal specification has appeared in the peer-reviewed literature. This paper provides the definitive formalization of the Li-Chao tree, serving as both the official specification and an expansion of the original lecture materials [2]. We present complete algorithmic specifications, establish formal correctness proofs, analyze theoretical complexity, and provide empirical performance characterization. The LICT offers distinct advantages in implementation simplicity, numerical stability, and extensibility to advanced variants such as persistence and line segments.

1 Introduction

Dynamic Lower Envelope maintenance is a fundamental problem in computational geometry with extensive applications. We focus on the Li-Chao tree (LICT), a data structure that maintains a set of linear functions while supporting insertion and query operations. The problem is defined as follows: given a dynamic set of linear functions $y = kx + b$, support efficient insertion of new lines and querying the minimum (or equivalently, maximum) value at arbitrary x coordinates. This report focuses on minimum queries; maximum queries are obtained by negating line parameters.

Formally, we require a data structure supporting two operations:

1. **Add Line:** Insert a new line $y = kx + b$ into the structure.
2. **Query:** Given x_0, compute
$$\min_i \{k_i x_0 + b_i\}$$
over all lines currently in the structure.

The LICT provides pseudo-polynomial $O(\log C)$ time per operation for minimum (or maximum) queries, where
$$C = \frac{\text{coordinate range}}{\text{precision level}}$$
represents the ratio of the coordinate range to the precision level. For integer coordinates with range $[0, 10^9]$, $C = 10^9$; for the same range with precision 10^{-6}, $C = 10^{15}$.

The structure also supports line segments (lines defined only on finite intervals $[x_l, x_r]$) in addition to infinite lines.

2 Related Work

Several approaches exist for Dynamic Lower Envelope maintenance, each with distinct trade-offs in terms of time complexity, implementation complexity, and applicability constraints. We review these solutions to establish the context for the LICT.

Dynamic maintenance of geometric configurations has been studied extensively in computational geometry.

2.1 Overmars and van Leeuwen (1981)

Overmars and van Leeuwen [1] presented foundational work on dynamic convex hull maintenance. Their data structure supports insertion and deletion of lines while maintaining the lower envelope, enabling efficient querying of the minimum value at any point.

Their approach uses a balanced binary search tree to explicitly maintain the convex hull. Each node stores a line, and the tree is ordered by slope. Intersection points between adjacent lines are computed to determine the hull structure. Queries take $O(\log N)$ time, while insertions and deletions require $O(\log^2 N)$ time.

2.2 Monotonic Convex Hull Trick

The monotonic convex hull trick addresses the special case where line slopes are inserted in monotonically increasing or decreasing order.

When insertions arrive in order of monotonically increasing (or decreasing) slopes, a deque-based approach achieves $O(1)$ amortized time per insertion and $O(1)$ amortized time per query. This variant, widely used in dynamic programming optimization, maintains the convex hull incrementally without requiring balanced tree structures. However, the monotonicity restriction limits its applicability to problems where line slopes are known to follow a specific order.

2.3 Dynamic Convex Hull Trick

For arbitrary insertion sequences without monotonicity constraints, a balanced binary search tree maintains the hull explicitly. Each insertion and query requires $O(\log N)$ amortized time. The implementation complexity stems from the need to compute and maintain intersection points between adjacent lines in the hull.

The Dynamic CHT requires computing intersection points between adjacent lines in the hull as

$$x_{\text{intersect}} = \frac{b_2 - b_1}{k_1 - k_2},$$

necessitating careful handling of precision (near parallel lines).

3 Overview

This section presents the Li-Chao tree.

3.1 Core Insight

The Li-Chao tree is built upon a fundamental observation about line dominance over intervals. Consider two lines defined over an interval $[l, r]$:

$$f_1(x) = k_1 x + b_1 \quad \text{and} \quad f_2(x) = k_2 x + b_2.$$

Since two distinct lines intersect at most once, one line must be strictly lower than the other on more than half of the interval.

Without loss of generality, assume f_1 dominates (achieves the lower value) on the majority subinterval $[m, r]$ where

$$m = \frac{l + r}{2}.$$

We designate f_1 as the *representative* line for $[l, r]$, serving as the optimal answer for any query $x \in [l, m]$.

The key insight is that f_2 only requires consideration when the query falls in the minority subinterval $[m, r]$, which comprises at most half the original interval. This observation enables a recursive decomposition: either the query is answered by the current representative, or we recurse on a subproblem of strictly half the size.

This interval-halving property yields the logarithmic query time and forms the basis of the tree structure.

3.2 Algorithmic Approach

The LICT offers a fundamentally different approach based on implicit envelope maintenance through interval subdivision. Rather than tracking the convex hull geometry explicitly, the structure maintains the best line at each interval, recursively partitioning the query range.

We first consider the simplest and most common setting: all queried x-coordinates are integers. In this case, if the coordinate range is $[0, C - 1]$ for some positive integer C, the tree recursively bisects this range into subintervals $[l, r]$ where $l, r \in \mathbb{Z}$. Every leaf corresponds to a single integer, so the tree has depth $\lceil \log_2 C \rceil$ and every query lands at a unique leaf. This integer setting is the one most frequently encountered in competitive programming and in many algorithm design applications, and we use it throughout the examples below.

Remark (non-integer queries). The same structure handles queries at non-integer coordinates by refining the discretisation. If queries require precision ϵ (e.g., $\epsilon = 10^{-6}$), one can rescale the coordinate range by $1/\epsilon$, effectively treating each precision step as one unit. The coordinate universe size becomes $C = \text{coordinate range}/\epsilon$, and the tree depth grows to $\lceil \log_2 C \rceil$ accordingly. All algorithmic details remain identical; only the universe size changes. For example, a range of $[0, 10^9]$ with precision 10^{-6} yields $C = 10^{15}$ and tree depth ≈ 50.

Each node in the tree represents an interval $[l, r]$ and stores one line.

Insertion. A line is *routed* through the tree following a single path from root to leaf. At each node on this path, the new line is compared with the currently stored line. The line that achieves the lower value at the midpoint

$$m = \frac{l + r}{2}$$

is *stored* at that node; the suboptimal line continues downward to be routed through the appropriate child subtree.

The appropriate child is determined by where the relative ordering changes. Let $f_{\text{new}}(x) = k_{\text{new}}x + b_{\text{new}}$ denote the new line and $f_{\text{stored}}(x) = k_{\text{stored}}x + b_{\text{stored}}$ denote the line currently stored at the node. Define boolean comparison values:

$$L = [f_{\text{new}}(l) < f_{\text{stored}}(l)]$$
$$M = [f_{\text{new}}(m) < f_{\text{stored}}(m)]$$

where $[\cdot]$ denotes 1 if true and 0 if false. The child selection rule is:

$$\text{child} = \begin{cases} \text{left} & \text{if } L \neq M \\ \text{right} & \text{if } L = M \end{cases}$$

This follows from the single-intersection property: if the ordering changes between l and m, the intersection point lies in $[l, m]$; otherwise, it lies in $[m, r]$.

Algorithms 1 and 2 present the insertion and query procedures.

When the insertion reaches maximum depth ($r - l < \epsilon$, where ϵ is the required precision), the interval is too narrow to distinguish any two query points. The stored line already resolves every query in that interval, so the losing line is simply dropped: no further recursion is needed. For integer coordinates ($\epsilon = 1$), this condition simplifies to $l = r$.

Algorithm 1 LICT Line Insertion

Require: Node pointer *node*, new line *new_line*, interval $[l, r]$
if *node* is null **then**
    node $\leftarrow$ new Node(*new_line*)
    return
end if
$m \leftarrow l + (r - l)/2$
$lef \leftarrow new\_line.eval(l) < node.line.eval(l)$
$midf \leftarrow new\_line.eval(m) < node.line.eval(m)$
if *midf* **then**
    swap(*node.line*, *new_line*)
end if
if $r - l < \epsilon$ **then** ▷ Max depth reached; drop the losing line
    return
end if
if $lef \neq midf$ **then**
    Insert(*node.left*, *new_line*, l, m)
else
    Insert(*node.right*, *new_line*, m, r)
end if

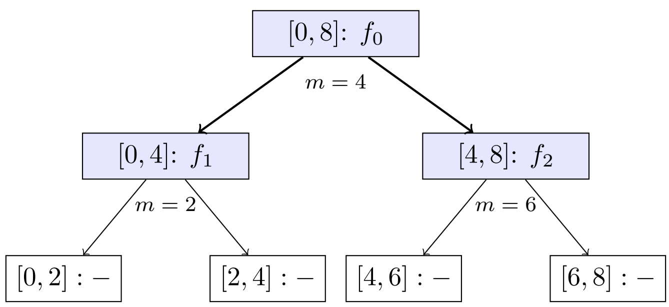

Figure 1: Tree structure before inserting $f_{new}(x) = -x + 10$. Currently stores $f_0(x) = x$ at root, $f_1(x) = 2x - 4$ at $[0, 4]$, and $f_2(x) = 0.5x + 2$ at $[4, 8]$.

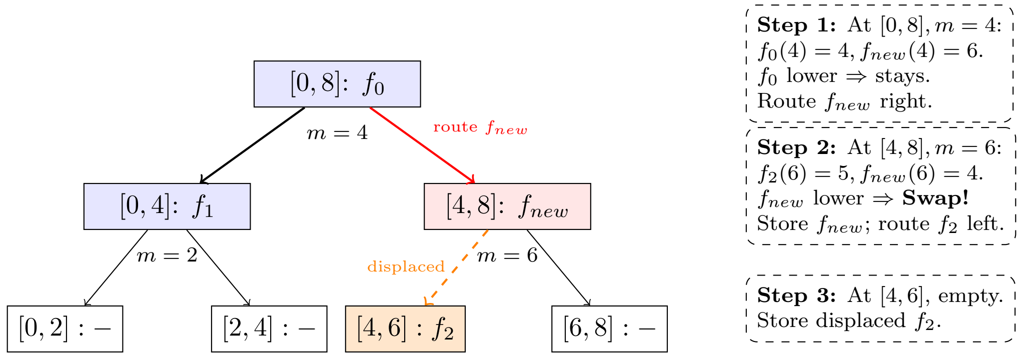

Figure 2: Tree structure after inserting $f_{new}(x)=-x+10$. At $[4,8]$, $f_{new}(6)=4<f_2(6)=5$, so f_{new} swaps in and f_2 is displaced to $[4,6]$.

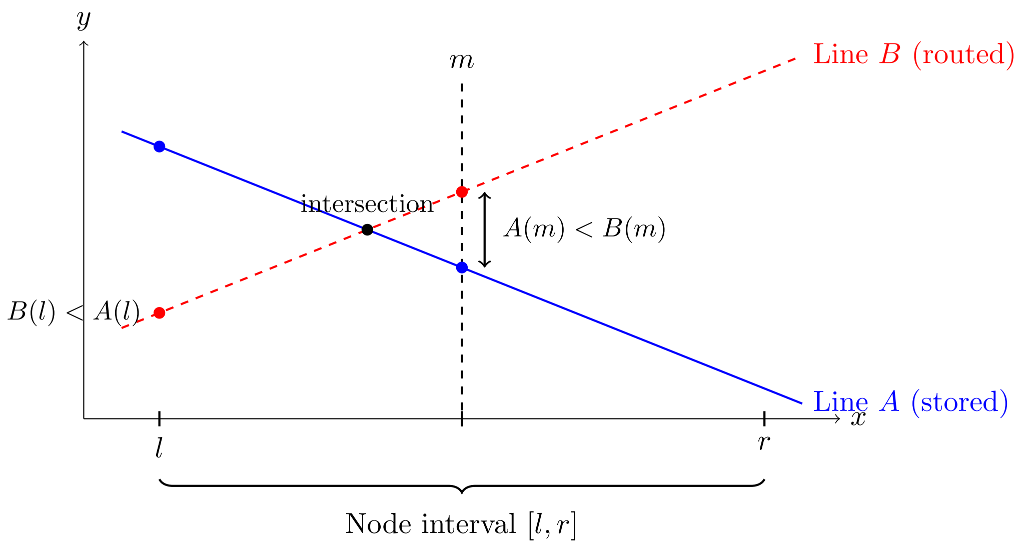

Figure 3: Interval Advantage Line Diagram. The node stores Line A because it achieves the lower value at the midpoint. Line B is routed to the left child where it may be optimal.

Note that this is *local* optimality only. Lines routed down other branches may achieve lower values at the midpoint, so the stored line is not necessarily globally optimal at that coordinate.

Query. We traverse the path to x_0, evaluating all stored lines along the path. The minimum value encountered equals the lower envelope at x_0 because any line that could be optimal at x_0 is stored on this path.

Algorithm 2 LICT Query

Require: Node pointer *node*, query coordinate x, interval $[l, r]$
Ensure: Minimum value at coordinate x

```
if node is null then
    return +∞
end if
m ← l + (r − l)/2
val ← node.line.eval(x)
if r − l < ϵ then
    return val
end if
if x ≤ m then
    return min(val, Query(node.left, x, l, m))
else
    return min(val, Query(node.right, x, m, r))
end if
```

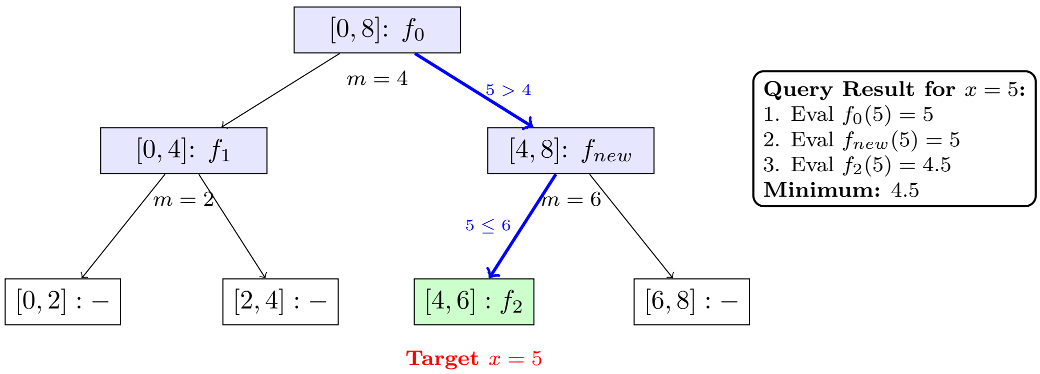

Figure 4: Query Tree View. To query at $x = 5$, we follow the path $[0, 8] \to [4, 8] \to [4, 6]$. We evaluate each stored line along the path (f_0, f_{new}, and f_2) and return the minimum.

3.3 Theoretical Analysis

We now present a formal analysis of the LICT's correctness and complexity.

Definition 1 (Coordinate Universe)**.** The LICT operates on a discrete domain where

$$C = \frac{\text{coordinate range}}{\text{precision level}}.$$

The tree depth is $h = \lceil \log_2 C \rceil$.

3.3.1 Correctness

We establish the correctness of the LICT through three results: a foundational property of linear functions, a routing invariant maintained by insertion, and a global correctness theorem for queries.

Lemma 1 (Linear Domination)**.** *Let $f(x) = k_1 x + b_1$ and $g(x) = k_2 x + b_2$ be linear functions. If $f(a) \leq g(a)$ and $f(b) \leq g(b)$ for $a \leq b$, then $f(x) \leq g(x)$ for all $x \in [a, b]$.*

Proof. Let $h(x) = f(x) - g(x)$. Since h is linear with $h(a) \leq 0$ and $h(b) \leq 0$, for any $x \in [a, b]$:

$$h(x) = h(a) \cdot \frac{b - x}{b - a} + h(b) \cdot \frac{x - a}{b - a} \leq 0. \qquad \square$$

Lemma 2 (Routing Dominance)**.** *At a node with interval $[l, r]$ and midpoint m, if line A is kept $(A(m) \leq B(m))$ and line B is pushed to one child, then A dominates B on the opposite child's interval.*

Proof. Without loss of generality, suppose B is pushed to the left child, so $B(l) < A(l)$. Since $A(m) \leq B(m)$ and $B(l) < A(l)$, the two lines intersect in $[l, m]$. For all $x \in [m, r]$, the difference $A(x) - B(x)$ has the same sign as at m, so $A(x) \leq B(x)$. By Lemma 1, A dominates B on $[m, r]$. □

Lemma 3 (Query Correctness)**.** *For any query point x_0, the value returned by the query operation equals $\min_i\{k_i x_0 + b_i\}$ over all lines in the structure.*

Proof. The query traverses the unique root-to-leaf path P containing x_0, returning the minimum of the stored lines' values at x_0. Since these are a subset of all inserted lines, query$(x_0) \geq \min_i L_i(x_0)$. It remains to show the reverse inequality.

Suppose for contradiction that query$(x_0) > \min_i L_i(x_0)$. Let L^* be a line achieving the global minimum at x_0. Then no line stored on P has value $\leq L^*(x_0)$ at x_0; in particular, L^* is not stored on any node of P.

Since L^* enters at the root (on P) and is not stored on P in the final tree, at some node $w \in P$, L^* was pushed to the child not containing x_0. By Lemma 2, the line S kept at w satisfies $S(x_0) \leq L^*(x_0)$, contradicting the assumption.

If S is later displaced from w by a new line C:

- If S is routed toward x_0: S moves deeper on P, preserving the contradiction.
- If S is routed away from x_0: by Lemma 2, $C(x_0) \leq S(x_0) \leq L^*(x_0)$, and C is at $w \in P$. If C is also displaced, repeat.

Since displacements are finite, the chain terminates with a contradiction. □

3.3.2 Complexity Analysis

Lemma 4 (Time Complexity)**.** *Both insertion and query operations require $O(\log C)$ time.*

Proof. The LICT is a binary tree of depth

$$h = \lceil \log_2 C \rceil = O(\log C).$$

For insertion: The new line follows exactly one root-to-leaf path. At each node, we perform: (1) $O(1)$ line evaluations $(kx + b)$; (2) $O(1)$ comparisons; and (3) optionally a line swap. Each operation takes $O(1)$ time. With $O(\log C)$ nodes visited, total time is $O(\log C)$.

For query: The traversal follows one root-to-leaf path with $O(1)$ work per node (line evaluation and comparison). Total time is $O(\log C)$. □

Lemma 5 (Space Complexity)**.** *The LICT stores at most $O(N)$ nodes in the worst case, where N is the number of inserted lines.*

Proof. Each full-line insertion follows a single root-to-leaf path. At each existing node, the algorithm performs $O(1)$ comparisons and recurses into exactly one child. If the path reaches a null child, it creates one new node and returns; if the path terminates at an existing leaf, no new node is created. Therefore each insertion creates at most one new node. With N insertions, the total number of nodes is at most N, giving $O(N)$ space.

Remark. For line *segment* insertion, each segment may decompose into $O(\log C)$ canonical intervals, creating up to $O(\log C)$ nodes per insertion and $O(N \log C)$ nodes overall. □

Deletion. The standard LICT does not support efficient deletion. Removing a line would require traversing all nodes where that line might be stored and recomputing optimal lines from descendants, which requires $\Omega(N)$ time in the worst case. For applications requiring deletion, an alternative approach is reconstructing the tree periodically.

The LICT's complexity depends on the coordinate range rather than the number of lines. This provides consistent performance regardless of hull size, though it cannot exploit cases where the hull remains small relative to the number of insertions.

4 Benchmarks

Reference implementations and benchmarks are available at `https://github.com/chnlich/lichao-tree`.

To validate the theoretical complexity analysis and characterize practical performance differences between the LICT and Dynamic CHT, we conducted systematic empirical evaluation across varying problem scales and input distributions.

4.1 Experimental Setup

All benchmarks were conducted with the following configuration.

Hardware Specifications:

- **CPU:** AMD Ryzen 9 3950X 16-Core Processor @ 3.50GHz (Zen 2 microarchitecture)
- **Core Configuration:** 16 cores, 32 threads (SMT enabled)
- **L1 Cache:** 512 KiB L1 data cache, 512 KiB L1 instruction cache
- **L2 Cache:** 8 MiB
- **L3 Cache:** 16 MiB (shared)
- **Memory:** 64 GB DRAM
- **Platform:** Linux x86_64 (WSL2 virtualized)

Software Configuration:

- **Compiler:** g++ 11.4.0
- **Optimization Flags:** `-O3 -std=c++17 -Wall -Wextra`
- **Operating System:** Ubuntu 22.04.5 LTS (WSL2, kernel 6.6.87)

Experimental Parameters:

- **Test sizes:** 10^5, 10^6, and 10^7 operations
- **Coordinate range:**
$$C = 10^9$$
(assuming integer precision $\epsilon = 1$), with $x, k, m \in [-10^9, 10^9]$
- **Random seed:** 42

- **Measurement protocol:** Each test was run 10 times; reported times are the average. Variance was low ($< 5\%$ coefficient of variation across runs). Benchmarks were run on an isolated system with no other user processes to minimize timing noise.
- **Distributions:** We write
$$X \sim U(a, b)$$
to denote that random variable X is drawn from a continuous uniform distribution over the interval $[a, b]$.
  - **Random:** Slopes $k \sim U(-10^9, 10^9)$, intercepts $m \sim U(-10^9, 10^9)$. Expected hull size:
$$\Theta(\log N).$$
By point-line duality [3], each line $y = kx + m$ maps to the dual point (k, m). The upper envelope size of the lines equals the upper convex hull size of the dual point set, which is uniformly distributed in the rectangle $[-10^9, 10^9]^2$. By the Rényi and Sulanke theorem [6], the expected number of convex hull vertices of N uniform random points in a convex polygon is $\Theta(\log N)$.
  - **All on Hull:** Lines $y = -(i+1)x + (i+1)^2$ for $i \in [0, N-1]$. All N lines contribute to the hull.

4.2 Results

Table 1 presents the performance comparison.

Table 1: Performance Comparison (Time in milliseconds)

N	Distribution	Algorithm	Insert (ms)	Query (ms)	Total (ms)
10^5	Random	LICT	·	·	·
10^5	Random	Dynamic CHT	·	·	·
10^5	All on Hull	LICT	·	·	·
10^5	All on Hull	Dynamic CHT	·	·	·
10^6	Random	LICT	·	·	·
10^6	Random	Dynamic CHT	·	·	·
10^6	All on Hull	LICT	·	·	·
10^6	All on Hull	Dynamic CHT	·	·	·
10^7	Random	LICT	·	·	·
10^7	Random	Dynamic CHT	·	·	·
10^7	All on Hull	LICT	·	·	·
10^7	All on Hull	Dynamic CHT	·	·	·

4.3 Parameter-Matched Comparison: The $N = C$ Regime

The theoretical complexity analysis reveals that Dynamic CHT achieves $O(\log N)$ time while LICT achieves $O(\log C)$. To directly compare these regimes, we conduct experiments where the number of lines N equals the coordinate universe size C. This configuration mirrors dynamic programming optimizations common in competitive programming, where the state recurrence takes the form:
$$dp[i] = \min_{0 \le j < i} \{dp[j] + \text{cost}(j, i)\}$$
Here, $\text{cost}(j, i)$ represents a linear function of i. Since the queries (indices i) are within a fixed range $[0, N]$, the coordinate universe size C effectively equals N.

In this static universe setting, an iterative segment tree [5] implementation (often called ZKW segment tree) can be employed to reduce recursion overhead and improve cache locality. We therefore include a third variant, **ZKW LICT**, in this comparison to evaluate the benefits of this specialized optimization against the standard LICT and Dynamic CHT.

- **Configuration:** $N = C = 10^5$, $N = C = 10^6$, and $N = C = 10^7$
- **Lines (Random):** $k, b \sim U(-C/2, C/2)$
- **Lines (All on Hull):** $y = -(i+1)x + (i+1)^2$ for $i \in [0, N/2-1]$, shuffled before insertion
- **Query points:** $x \sim U(-C/2, C/2)$ (Random), $x \sim U(0, N)$ (All on Hull)
- **Operation mix:** $N/2$ insertions followed by $N/2$ queries

Table 2: Performance in $N = C$ Regime (Time in milliseconds)

N = C	Distribution	Algorithm	Insert (ms)	Query (ms)	Total (ms)
10^5	Random	LICT	2.23	1.00	3.22
10^5	Random	ZKW LICT	2.39	0.59	2.98
10^5	Random	Dynamic CHT	2.15	0.36	2.51
10^5	All on Hull	LICT	8.05	5.73	13.78
10^5	All on Hull	ZKW LICT	5.41	1.93	7.34
10^5	All on Hull	Dynamic CHT	8.86	7.69	16.54
10^6	Random	LICT	26.77	9.89	36.65
10^6	Random	ZKW LICT	27.50	6.60	34.10
10^6	Random	Dynamic CHT	23.49	3.61	27.10
10^6	All on Hull	LICT	174.34	246.87	421.20
10^6	All on Hull	ZKW LICT	83.63	66.59	150.22
10^6	All on Hull	Dynamic CHT	259.19	346.92	606.11
10^7	Random	LICT	314.03	104.11	418.13
10^7	Random	ZKW LICT	328.50	80.27	408.77
10^7	Random	Dynamic CHT	230.87	45.14	276.01
10^7	All on Hull	LICT	4683.66	5762.05	10445.72
10^7	All on Hull	ZKW LICT	1724.87	1435.00	3159.87
10^7	All on Hull	Dynamic CHT	6738.87	7096.22	13835.08

4.4 Analysis

In the case that C is larger than N, both algorithms have similar performance, even if the theoretical complexity of LICT is worse than Dynamic CHT. This is because the $O(\log C)$ factor is small enough to be negligible compared to the $O(\log N)$ factor, while the constant factor of LICT is smaller than that of Dynamic CHT.

In the $N = C$ regime, the ZKW LICT variant demonstrates substantial gains over both alternatives on dense hull inputs, with the $N = C = 10^7$ all-on-hull benchmark completed approximately $4\times$ faster than Dynamic CHT.

The LICT's division-free operations ensure consistent numerical accuracy across all inputs. For applications involving floating-point coordinates or extreme value ranges, the LICT provides robustness guarantees that the Dynamic CHT by avoiding intersection point computation.

5 Conclusion

The LICT is the preferred choice in the following scenarios:

5.1 Advantages

Line segment support required. When the problem involves line segments (lines valid only on subranges) rather than infinite lines, the LICT provides natural $O(\log^2 C)$ insertion. The Dynamic CHT can support segments but requires significantly more complex machinery.

Persistence required. Path copying [4] in the LICT is straightforward: insertions modify only nodes along a single root-to-leaf path, so copying those nodes creates a new version sharing unmodified subtrees with the previous version. Achieving persistence in the Dynamic CHT is substantially more complex due to the need to maintain hull invariants across versions.

Near parallel lines. When working on lines with subtle slope differences, the LICT's division-free operations avoid numerical instability.

Implementation time constraints. In settings such as competitive programming where implementation speed matters, the LICT's simplicity offers a clear advantage. The reduced code size and elimination of geometric corner cases allow for faster, more confident implementation.

5.2 Limitations

The LICT has several limitations that affect its applicability:

No efficient deletion. As noted above, the standard LICT does not support deletion of individual lines. Removing a line would require traversing all nodes where that line might be stored and recomputing optimal lines from descendants, requiring $\Omega(N)$ time in the worst case.

Coordinate range dependency. The LICT's complexity depends on C, the coordinate range divided by precision. For very large coordinate ranges with fine precision (e.g., 64-bit floating-point values spanning the entire representable range), C can become impractically large.

5.3 Future Work

Several directions for future research and development remain:

Deletion support. Developing an efficient deletion mechanism for the LICT would extend its applicability to dynamic scenarios requiring removal of lines. Potential approaches include lazy deletion with periodic reconstruction or augmenting nodes with additional structure to support efficient line removal.

Cache-efficient variants. The LICT's pointer-based tree structure exhibits poor cache locality compared to array-based representations. Investigating cache-oblivious or cache-aware variants could improve practical performance without sacrificing asymptotic guarantees.

Parallel implementations. The LICT's tree structure naturally supports parallel queries, but insertions are inherently sequential. Developing concurrent LICT variants that support parallel insertions while maintaining correctness would benefit multi-core applications.

Higher-dimensional extensions. While the LICT extends naturally to higher dimensions (maintaining hyperplanes instead of lines), the space and time complexities grow exponentially with dimension. Investigating approximate variants or dimensionality reduction techniques could broaden applicability.

References

[1] Overmars, M.H. and van Leeuwen, J. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166 to 204, 1981.

[2] Li, Chao. 线段树 [Segment trees]. Lecture slides, Hangzhou Xuejun High School, China, 2012. Available at `https://github.com/chnlich/lichao-tree/blob/main/doc/slides.ppt`.

[3] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[4] Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86 to 124, 1989.

[5] Bentley, J.L. Solutions to Klee's rectangle problems. Unpublished manuscript, Carnegie Mellon University, 1977.

[6] Rényi, A. and Sulanke, R. Über die konvexe Hülle von n zufällig gewählten Punkten. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 2(1):75 to 84, 1963.