The Li-Chao Tree: Algorithm Specification and Analysis

Li Chao

February 17, 2026

Abstract

The Li-Chao tree (LICT) was first introduced in 2012 lecture materials 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 2012 lecture materials. 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 $[m, r]$.

Algorithm 1 and 2 present the insertion and query procedures.

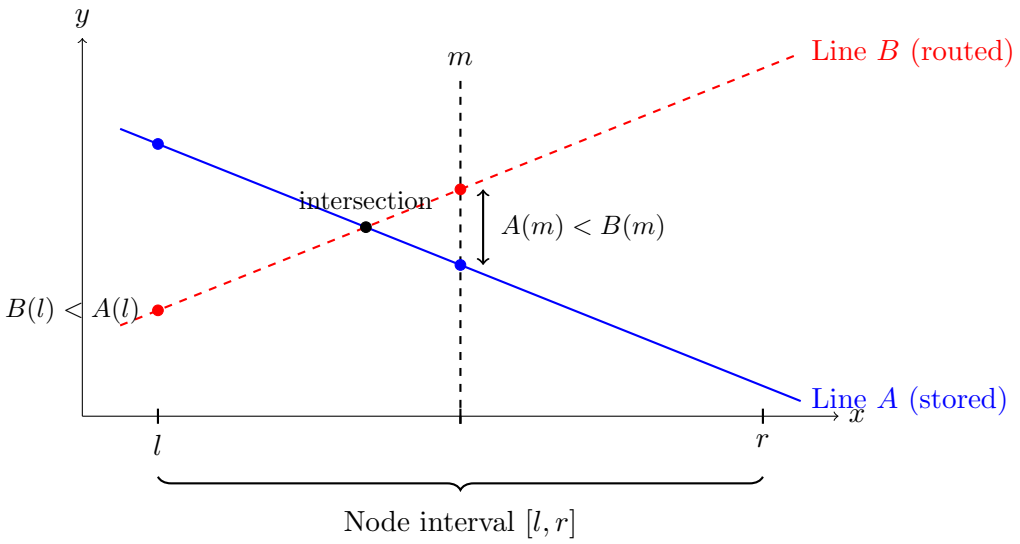

Figure 1: Interval Advantage Line Diagram. The node stores Line A because it achieves the lower value at the midpoint. Line B is routed to the left child where it may be optimal.

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 $l = r$.

Figure 2 illustrates the insertion of a new line $y = 3x - 8$ into a structure already containing $y = x$ and $y = 2x$.

Figure 2 illustrates the insertion of a new line $y = 0.5x + 5$ into a structure already containing $y = x$, $y = 2x$, and $y = 0.6x + 2$.

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

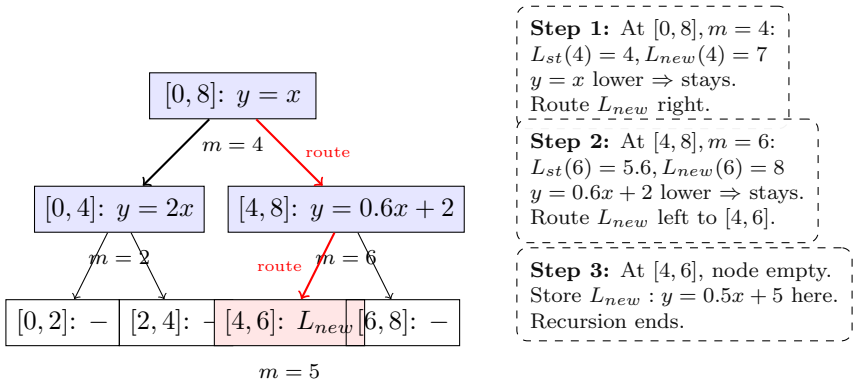

Figure 2: Line Insertion Tree View. Inserting line $y = 0.5x+5$. This example shows how a line is routed through multiple occupied nodes ($[0, 8] \to [4, 8] \to [4, 6]$) based on midpoint comparisons until it finds an empty node.

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 3 shows a query at $x = 1$ following a single root-to-leaf path.

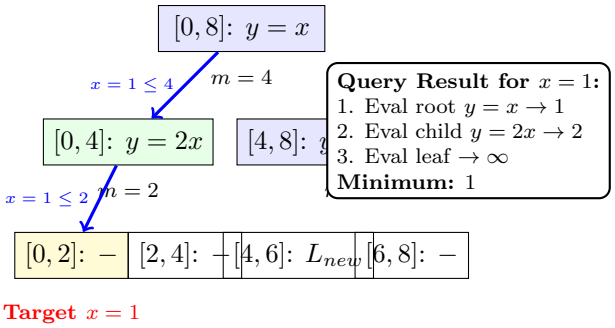

Figure 3: Query Tree View. To query at $x = 1$, we follow the path to the leaf containing 1. At each node along the path, we evaluate the stored line and take the minimum of all results.

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 first establish the geometric intuition underlying the LICT's correctness. The key insight is that two distinct lines intersect at most once. This fundamental property of linear functions ensures that if a line yields a higher value at a midpoint, it can yield a lower value than the stored line on at most one side of the midpoint, never both.

Why this maintains the lower envelope. Consider two lines A and B compared at a node's midpoint m:

- If A achieves a lower value than B at m, but B yields a lower value at one endpoint, then B can only outperform A on the side of m containing that endpoint (since two lines intersect at most once).
- Therefore, B is routed to exactly one child, the side where it might achieve a lower value.

This property guarantees that the suboptimal line need only be propagated down a single child path.

Lemma 1 (Local Optimality Invariant)**.** *For any node v with interval $[l, r]$ and midpoint*

$$m = \left\lfloor \frac{l + r}{2} \right\rfloor,$$

the line stored at v achieves the minimum value at $x = m$ among all lines that have been routed through v.

Proof. We proceed by induction on the sequence of insertions.

Base case: When the first line is inserted, it is stored at the root and all descendants are null. The invariant holds trivially.

Inductive step: Assume the invariant holds before inserting a new line L_{new}. Consider the insertion path from root to leaf. At each node v along this path:

1. Let L_{stored} be the line currently at v.
2. If $L_{\text{new}}(m) \geq L_{\text{stored}}(m)$, we keep L_{stored} and route L_{new} to a child. The invariant at v remains satisfied.
3. If $L_{\text{new}}(m) < L_{\text{stored}}(m)$, we swap the lines, storing L_{new} and routing L_{stored} to a child. The invariant at v is now satisfied by L_{new}.

The routed line proceeds to exactly one child (determined by the intersection point location), so the inductive hypothesis applies to subtrees. By induction, the invariant holds after all insertions. □

Lemma 2 (Query Correctness)**.** *For any query point $x_0 \in X$, the value returned by the query operation equals $\min_i\{k_i x_0 + b_i\}$ over all lines in the structure.*

Proof. Consider any line L in the structure and the query point x_0. Let P be the unique root-to-leaf path whose intervals all contain x_0. Line L follows exactly one root-to-leaf path P_L during insertion.

We claim L is evaluated at x_0 if and only if x_0 lies in the interval where L is optimal among lines routed through some node on P_L. There are two cases:

1. If $P_L = P$, then L is stored at some node on P (specifically, at the deepest node where it achieved minimum at the midpoint), and is evaluated during the query.
2. If $P_L \neq P$, let v be the deepest node common to both paths. Since x_0 is not in the child interval that L was routed to, x_0 lies on the opposite side of the midpoint from where L could potentially be better than the stored line. By the single-intersection property of lines, L cannot be optimal at x_0.

Thus, exactly those lines that could be optimal at x_0 are evaluated, and the minimum over evaluated lines equals the global minimum. □

3.3.2 Complexity Analysis

Lemma 3 (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 4 (Space Complexity)**.** *The LICT stores at most $O(n \log C)$ nodes in the worst case, where n is the number of inserted lines.*

Proof. Each line insertion creates at most one new node per level along its insertion path (when reaching a null child). The path length is

$$O(\log C).$$

With n insertions, the total number of nodes is at most $O(n \log C)$. □

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:
$$\Theta(\sqrt{n}).$$
  - **All on Hull:** Lines $y = i \cdot x - i^2$ for $i \in [1, n]$. 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	LICT	Dynamic CHT
10^5	Random	5.0 ms	3.0 ms
10^5	All on Hull	16.9 ms	16.0 ms
10^6	Random	52.9 ms	31.3 ms
10^6	All on Hull	693.3 ms	882.2 ms
10^7	Random	548.8 ms	326.9 ms
10^7	All on Hull	14702.5 ms	16249.0 ms

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 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:** $k, b \sim U(-C/2, C/2)$
- **Query points:** $x \sim U(-C/2, C/2)$
- **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	Standard LICT	3.1	1.9	5.0
		ZKW LICT	1.8	0.9	2.7
		Dynamic CHT	1.8	1.2	3.0
10^6	Random	Standard LICT	31.5	21.4	52.9
		ZKW LICT	18.2	11.2	29.4
		Dynamic CHT	19.4	11.9	31.3
10^7	Random	Standard LICT	320.1	228.7	548.8
		ZKW LICT	192.3	124.5	316.8
		Dynamic CHT	199.1	127.8	326.9

References

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