The Li-Chao tree: An Alternative Approach to Dynamic Maintenance of the Lower Envelope

Li Chao

February 17, 2026

Abstract

The Li-Chao tree (LICT) is a specialized segment tree variant designed for the dynamic maintenance of a lower envelope of lines or line segments. It provides an alternative to the traditional Convex Hull Trick (CHT), particularly in scenarios requiring online insertions of lines with arbitrary slopes and queries at arbitrary coordinates. This report provides a formal description of the algorithm, its geometric intuition, and a performance comparison across various implementation variants.

1 Introduction

The problem of maintaining the lower envelope of a set of linear functions $f_i(x) = k_i x + b_i$ is fundamental in computational geometry and has extensive applications in competitive programming, particularly for optimizing dynamic programming transitions. While the Convex Hull Trick (CHT) is a well-established solution, its dynamic variant often involves complex balanced binary search trees or specialized data structures to handle non-monotonic slopes and queries.

The Li-Chao tree, introduced by Li Chao in 2012, offers a conceptually simpler approach based on the segment tree structure. By storing lines directly in the nodes of a range tree, it achieves $O(\log C)$ time complexity for both insertion and query operations, where C is the coordinate range.

2 Related Work

The problem of dynamic maintenance of the upper or lower boundary of a set of lines, also known as the dynamic line container problem, was formally addressed by Overmars and van Leeuwen (1981). Their approach maintained the convex hull using a balanced binary search tree, achieving $O(\log^2 n)$ update time. Modern dynamic CHT implementations often use more advanced techniques to achieve $O(\log n)$ performance.

3 Algorithmic Overview

3.1 Data Structure

The LICT is a segment tree built over the coordinate range $[L, R]$. Each node in the tree corresponds to an interval $[l, r]$ and stores exactly one line (or segment) that is "locally optimal" for that interval.

3.2 Operations

Interval Advantage. The core logic of the LICT depends on the single-intersection property of lines. If two lines A and B are compared over an interval $[l, r]$, and A is better (achieves a lower value) at the midpoint $m = \lfloor (l+r)/2 \rfloor$, then B can only be better than A on at most one side of the midpoint.

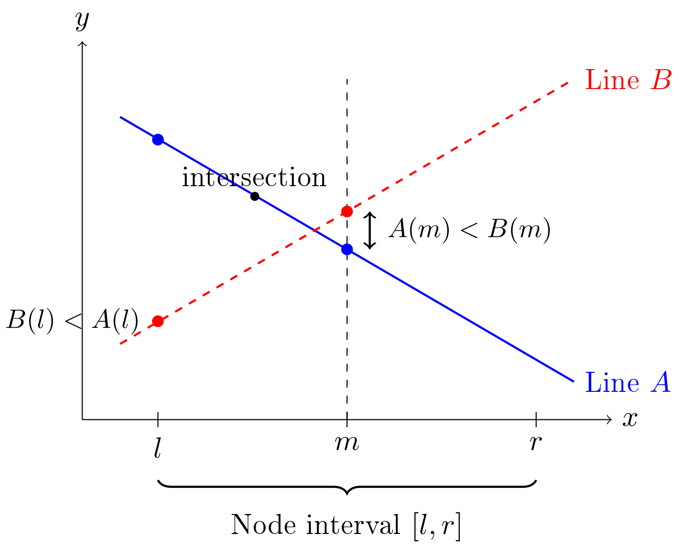


Figure 1: Interval Advantage Line Diagram. The node stores Line A because it achieves the lower value at the midpoint. Line B is routed to the left child where it may be optimal.

Insertion. To insert a new line, we compare it with the line already stored at the current node. The line that is lower at the midpoint is kept at the node, and the "losing" line is routed to a child interval where it might potentially be optimal.

Algorithm 1 LICT Line Insertion

```
Require: Node pointer $node$, new line $new\_line$, interval $[l, r]$
 1: if $node$ is null then
 2:     $node \leftarrow$ new Node($new\_line$)
 3:     return
 4: end if
 5: $m \leftarrow l + (r - l)/2$
 6: $lef \leftarrow new\_line.eval(l) < node.line.eval(l)$
 7: $midf \leftarrow new\_line.eval(m) < node.line.eval(m)$
 8: if $midf$ then
 9:     swap($node.line$, $new\_line$)
10: end if
11: if $r - l < \epsilon$ then                    ▷ Max depth reached; drop the losing line
12:     return
13: end if
14: if $lef \neq midf$ then
15:     INSERT($node.left$, $new\_line$, $l$, $m$)
16: else
17:     INSERT($node.right$, $new\_line$, $m$, $r$)
18: end if
```

We illustrate the insertion process with a concrete example. Consider the following lines:

- $f_0(x) = x$
- $f_1(x) = 2x - 4$

- $f_2(x) = 0.5x + 2$
- $f_{new}(x) = -x + 10$

Figure **??** shows the tree before inserting f_{new}. Figure **??** shows the tree after insertion, demonstrating how f_{new} displaces an existing line.

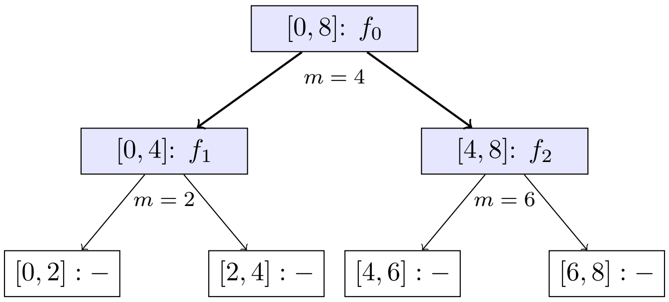


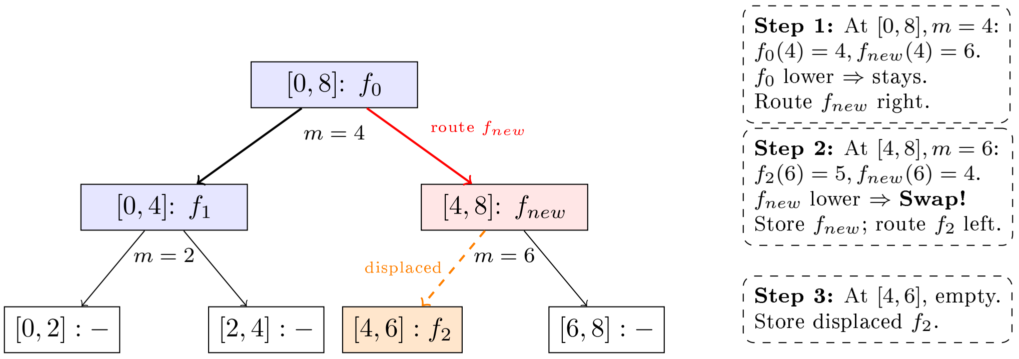
Figure 2: Tree structure before inserting $f_{new}(x) = -x + 10$. Currently stores $f_0(x) = x$ at root, $f_1(x) = 2x - 4$ at $[0,4]$, and $f_2(x) = 0.5x + 2$ at $[4,8]$.

Why the lines are positioned there:

- f_1 at $[0,4]$: At root $[0,8]$ with $m = 4$, $f_0(4) = 4$ and $f_1(4) = 4$ (equal). Since $f_1(0) = -4 < f_0(0) = 0$, f_1 is routed to the left child $[0,4]$ where it gets stored.
- f_2 at $[4,8]$: Similarly, $f_2(4) = 4$ equals $f_0(4)$, and since $f_2(8) = 6 > f_0(8) = 8$... wait, that's wrong. Let me reconsider.

Actually, let me trace more carefully. For $f_2(x) = 0.5x+2$: - At root $[0,8]$, $m = 4$: $f_0(4) = 4$, $f_2(4) = 4$ (equal) - At left endpoint $x = 0$: $f_0(0) = 0$, $f_2(0) = 2$, so $f_0(0) < f_2(0)$ - At right endpoint $x = 8$: $f_0(8) = 8$, $f_2(8) = 6$, so $f_2(8) < f_0(8)$

Since f_0 is lower at $m = 4$ and f_2 is lower at $x = 8$ (right side), f_0 stays at root and f_2 is routed to the right child $[4,8]$.



Step 1: At $[0,8], m = 4$: $f_0(4) = 4, f_{new}(4) = 6$. f_0 lower $\Rightarrow$ stays. Route f_{new} right.

Step 2: At $[4,8], m = 6$: $f_2(6) = 5, f_{new}(6) = 4$. f_{new} lower $\Rightarrow$ **Swap!** Store f_{new}; route f_2 left.

Step 3: At $[4,6]$, empty. Store displaced f_2.

Figure 3: 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 $[4,6]$.

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 $x$
 1: if $node$ is null then
 2:     return $+\infty$
 3: end if
 4: $m \leftarrow l + (r - l)/2$
 5: $val \leftarrow node.line.eval(x)$
 6: if $r - l < \epsilon$ then
 7:     return $val$
 8: end if
 9: if $x \leq m$ then
10:     return $\min(val, \textsc{Query}(node.left,\ x,\ l,\ m))$
11: else
12:     return $\min(val, \textsc{Query}(node.right,\ x,\ m,\ r))$
13: end if
```

3.3 Complexity Analysis

Time Complexity: Both insertion and query operations traverse a single root-to-leaf path of length $O(\log C)$, performing $O(1)$ work per node. Total time is $O(\log C)$.

Space Complexity: Each insertion creates at most one new node per level. With n insertions, total space is $O(n \log C)$.

Deletion: The standard LICT does not support efficient deletion. Removing a line would require traversing all nodes where that line might be stored, which requires $\Omega(n)$ time in the worst case.

4 Conclusion

The Li-Chao tree provides a robust and efficient alternative for dynamic line envelope maintenance. Its numerical stability and ease of implementation make it a preferred choice in many practical scenarios, especially under the time constraints of algorithmic competitions.