A Comprehensive Study of Li-Chao Tree Variants for Dynamic Convex Hull Optimization

Chao Li (chnlich)
Citadel Securities

February 2, 2026

Abstract

The Li-Chao Tree (LICT) is a data structure for dynamically maintaining the lower (or upper) envelope of a set of linear functions, supporting efficient line insertion and point evaluation in $O(\log C)$ time, where C is the coordinate range. Originally proposed by Li Chao at ZJOI 2012 as a solution for online line envelope maintenance, it has since become a fundamental tool in competitive programming and algorithm design, with widespread application to dynamic programming optimization. This paper presents a systematic study of six distinct LICT implementations: Standard (Dynamic), Static, Discrete (Coordinate-Compressed), Iterative, Segment, and Persistent variants. Each variant is optimized for specific use cases, offering trade-offs between memory usage, query speed, and functional capabilities. We provide comprehensive benchmarks comparing these implementations against the Dynamic Convex Hull Trick (CHT) under various workloads, demonstrating that the Iterative LICT achieves competitive performance while the Persistent variant enables novel algorithmic applications. Our implementations are publicly available and serve as a reference for practitioners selecting the appropriate variant for their specific problem constraints.

1 Introduction

A fundamental problem in computational geometry is maintaining the lower (or upper) envelope of a set of linear functions. Given a set of lines $y = kx + m$, we need to support two operations efficiently:

1. **Add Line:** Insert a new line $y = kx + m$ into the structure.
2. **Query:** Given x_0, find the minimum (or maximum) value of y among all lines at $x = x_0$.

This problem arises naturally in various applications, including dynamic programming (DP) optimization, where many DP transitions feature the form:

$$dp[i] = \min_{j<i}\{dp[j] + f(j, i)\} \tag{1}$$

When $f(j, i)$ can be expressed as a linear function $m_j \cdot x_i + b_j$, the problem reduces to querying the minimum value among a set of lines at a specific point x_i.

The Convex Hull Trick (CHT) is the classical solution for this problem. There are two variants:

- **Deque CHT:** For monotonic slope insertions, using a deque achieves $O(1)$ amortized insertion and query.
- **Dynamic CHT:** For arbitrary insertion order, a balanced binary search tree (e.g., `std::multiset`) is required, achieving $O(\log n)$ insertion and query time, where n is the number of lines.

Many practical scenarios require inserting lines in arbitrary order, where the deque CHT cannot be applied. The **Li-Chao Tree** (LICT), introduced by Li Chao at the Zhejiang Olympiad in Informatics (ZJOI) in 2012, was originally designed to solve the online line envelope problem: maintaining the lower envelope of a dynamic set of lines with efficient insertion and query operations. Unlike the Dynamic CHT which maintains the convex hull explicitly using a balanced tree, the LICT takes a divide-and-conquer approach on the query coordinate range, supporting arbitrary line insertions in $O(\log C)$ time, where C is the coordinate range.

While the LICT was invented for general line envelope maintenance, it has found widespread application in dynamic programming optimization and other domains, including:

- Dynamic programming with non-monotonic transitions
- Geometric problems involving line arrangements
- Online algorithms requiring incremental line insertion
- Path optimization on trees with persistence

1.1 Contributions

This work makes the following contributions:

1. We implement and analyze six distinct LICT variants, each targeting specific use cases and constraints.
2. We provide a systematic performance evaluation comparing LICT variants against the Dynamic CHT baseline.
3. We introduce optimizations including iterative insertion, static memory allocation, and full persistence.
4. We present empirical results demonstrating the practical trade-offs between memory usage, preprocessing overhead, and query performance.

2 Related Work

2.1 Convex Hull Trick

The Convex Hull Trick dates back to the early days of competitive programming. There are two fundamentally different approaches:

1. Monotonic CHT (Deque-based): When lines are inserted in order of monotonic slope (either increasing or decreasing), the lower hull can be maintained in a deque. Queries for a monotonic sequence of x values are answered in $O(1)$ amortized time. This is the most common CHT variant taught in competitive programming.

2. Dynamic CHT (Balanced BST-based): When lines can be inserted in arbitrary order, the deque approach fails because removing obsolete lines from the middle of the hull is required. In this case, a balanced binary search tree (e.g., `std::multiset`) is used to maintain the hull. Each insertion and query takes $O(\log n)$ time, where n is the number of lines.

The Dynamic CHT implementation used in our benchmarks (shown in Listing 1) maintains lines in a `std::multiset`, enabling arbitrary insertion and query orders with logarithmic complexity. This is the correct baseline for comparison with LICT, as both handle the general case of arbitrary insertions.

```cpp
struct Line {
    mutable ll k, m, p;  // slope, intercept, intersection point
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
    static const ll inf = LLONG_MAX;
    // Floored division
    ll div(ll a, ll b) {
        return a / b - ((a ^ b) < 0 && a % b);
    }
    // Compute intersection point of lines x and y
    bool isect(iterator x, iterator y) {
        if (y == end()) return x->p = inf, 0;
        if (x->k == y->k)
            x->p = x->m > y->m ? inf : -inf;
        else
            x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    // Add line y = kx + m (for max hull; negate for min)
    void add(ll k, ll m) {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y))
            isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }
    // Query max at point x
    ll query(ll x) {
        if (empty()) return 0;
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};
```

Listing 1: Dynamic CHT Implementation (Balanced BST-based)

2.2 Li-Chao Tree Origins

The Li-Chao Tree was introduced in a lecture by Li Chao at ZJOI 2012 as a data structure for online maintenance of the lower envelope of lines. The original problem addressed was: given a set of lines, support efficient insertion of new lines and querying the minimum y-value at any given x-coordinate. Unlike the CHT, which focuses on explicitly maintaining the convex hull geometry, the LICT takes a divide-and-conquer approach on the query coordinate range, storing the best line at each interval midpoint. This design naturally supports:

- Arbitrary insertion order without performance degradation
- Easy extension to segment trees for line segments
- Persistence through path copying
- Coordinate compression for sparse query sets

2.3 Existing Implementations

Several high-quality LICT implementations exist in the competitive programming community:

- **KACTL** (KTH Algorithm Library): A concise, well-tested implementation used in ICPC competitions.
- **CP-Algorithms**: A widely-read tutorial with reference implementations in C++.
- **Codeforces Community**: Numerous blog posts exploring optimizations and variations.

Our work distinguishes itself by providing a comprehensive suite of variants with detailed performance analysis and use-case guidance.

3 Main Approach

3.1 Standard Li-Chao Tree

The standard LICT is a binary tree where each node represents an interval $[l, r]$ of the query coordinate range. Each node stores the line that is optimal at the midpoint of its interval.

```
void insert(Node* &node, Line new_line, ll l, ll r) {
    if (!node) { node = new Node(new_line); return; }
    ll mid = l + (r - l) / 2;
    bool left_better = new_line.eval(l) < node->line.eval(l);
    bool mid_better = new_line.eval(mid) < node->line.eval(mid);
    if (mid_better) swap(node->line, new_line);
    if (l == r) return;
    if (left_better != mid_better)
        insert(node->left, new_line, l, mid);
    else
        insert(node->right, new_line, mid + 1, r);
}
```

Listing 2: Standard LICT Insertion

The key insight is that if a new line is better at both endpoints, it dominates the existing line entirely. If it's better at exactly one endpoint, the lines must intersect within the interval, and we recursively process the side containing the intersection.

Complexity: Each insertion creates at most $O(\log C)$ nodes, and each query traverses $O(\log C)$ nodes, where C is the coordinate range.

3.2 Static Variant

The Static LICT pre-allocates a fixed-size node pool using a vector instead of dynamic heap allocation. This reduces memory fragmentation and can improve cache locality.

Trade-offs:

- **Advantage:** Predictable memory usage, no malloc overhead during operations.
- **Disadvantage:** Must reserve sufficient space upfront; resizing requires reallocation.

3.3 Discrete (Coordinate-Compressed) Variant

When queries are known in advance (offline scenario), we can compress the coordinate range to only include points that will be queried. This reduces the tree height from $O(\log C)$ to $O(\log Q)$, where Q is the number of unique query coordinates.

Preprocessing:

1. Collect all query coordinates.
2. Sort and deduplicate to create a compressed coordinate array.

3. Map tree nodes to indices in this compressed array.

Trade-offs:

- **Advantage:** Reduced tree height for sparse queries.
- **Disadvantage:** $O(Q \log Q)$ preprocessing overhead; cannot query arbitrary points not in the original set.

3.4 Iterative Variant

The Iterative LICT replaces recursive insertion and query with explicit loops using pointer references. This eliminates function call overhead and stack usage.

```
void add_line(ll k, ll m) {
    Line cur = {k, m};
    Node** ptr = &root;
    ll l = min_x, r = max_x;
    while (true) {
        if (!*ptr) { *ptr = new Node(cur); return; }
        // ... comparison logic ...
        if (lef != midf) { ptr = &node->left; r = mid; }
        else { ptr = &node->right; l = mid + 1; }
    }
}
```

Listing 3: Iterative Insertion Pattern

Trade-offs:

- **Advantage:** No recursion depth limits; slightly faster due to reduced overhead.
- **Disadvantage:** Code is slightly more complex; same asymptotic complexity.

3.5 Segment Li-Chao Tree

The Segment variant supports adding line segments (lines valid only on a subrange $[x_l, x_r]$) rather than infinite lines. This is implemented by decomposing the segment range using a segment tree structure, where each node maintains its own LICT for lines covering that node's range.

Use Cases:

- Problems where constraints apply only to specific intervals.
- Geometric problems involving partial line arrangements.

Complexity: Insertion of a segment takes $O(\log^2 n)$ time, where n is the number of coordinate points.

3.6 Persistent Li-Chao Tree

The Persistent LICT enables querying historical versions of the data structure. Each update creates a new root node while sharing unchanged subtrees with previous versions.

Implementation: When inserting a line into version v to create version v':

1. Copy the nodes along the insertion path.
2. Share unchanged subtrees between versions.
3. Return the index of the new root.

Space Complexity: Each update creates $O(\log C)$ new nodes, making total space $O(Q \log C)$ for Q operations.

Applications:

- Tree path queries: Build a persistent LICT for each node representing the path from root.
- Offline queries: Answer "what was the minimum at time t?" for arbitrary t.

4 Benchmarks

4.1 Experimental Setup

All benchmarks were conducted on a Linux system with the following configuration:

- CPU: x86_64 architecture
- Compiler: g++ with -O2 optimization
- Test sizes: 10^4 to 10^7 operations
- Distributions: Random, Monotonic K (slopes), Monotonic X (queries)

We compare against the Dynamic CHT (balanced BST-based) as the baseline, as it is the only CHT variant that supports arbitrary line insertions like the LICT. The monotonic deque-based CHT is excluded from comparison since it cannot handle arbitrary insertions.

4.2 Results

Table 1: Performance Comparison for 10^6 Operations (Random Distribution)

Algorithm	**Time (ms)**	**Relative Speed**
Dynamic CHT	53	1.00×
Li-Chao (Dynamic)	79	0.67×
Li-Chao (Iterative)	82	0.65×
Li-Chao (Static)	102	0.52×
Li-Chao (Discrete)	374	0.14×

Table 2: Performance Comparison for 10^7 Operations (Random Distribution)

Algorithm	**Time (ms)**	**Relative Speed**
Dynamic CHT	528	1.00×
Li-Chao (Iterative)	816	0.65×
Li-Chao (Dynamic)	819	0.64×
Li-Chao (Static)	898	0.59×
Li-Chao (Discrete)	6512	0.08×

4.3 Analysis

Iterative vs. Dynamic: The Iterative LICT matches or slightly outperforms the Dynamic (recursive) variant by eliminating function call overhead. Both maintain excellent logarithmic scaling, handling 10^7 operations in under one second.

Static Allocation: The Static variant shows slightly worse performance due to vector indexing overhead compared to direct pointer dereferencing. However, it offers better memory predictability.

Discrete Variant: The Discrete LICT is significantly slower (6–7× at 10^7 operations) due to the $O(N \log N)$ preprocessing cost of coordinate sorting and deduplication. This overhead is only justified when:

- The coordinate range C is extremely large ($> 10^{12}$).
- Query points are sparse and known offline.
- The reduced tree height significantly outweighs preprocessing costs.

Comparison with Dynamic CHT: The Dynamic CHT (balanced BST-based) maintains a slight edge (30–35% faster) due to lower constant factors. Both structures handle arbitrary insertions in $O(\log n)$ time. The choice between them often comes down to:

- **Dynamic CHT:** Slightly faster for dense line sets; requires careful handling of floating-point comparisons for intersection points.
- **LICT:** Simpler to implement; works naturally with integer coordinates; easier to extend to segments and persistence.

Note that the monotonic deque CHT achieves $O(1)$ amortized operations but cannot handle arbitrary insertions, so it is not a valid alternative for the general case.

Monotonic Workloads: Under monotonic insertion or query patterns, the performance gap between variants narrows, as cache effects become more pronounced. The Iterative LICT consistently shows the best performance among LICT variants across all tested distributions.

5 Summary

This paper presented a comprehensive analysis of six Li-Chao Tree variants, each optimized for specific problem constraints. Our key findings:

1. **For general use:** The Iterative or Dynamic LICT provides the best balance of performance and flexibility, handling 10^7 operations in under one second.
2. **For known query sets:** The Discrete variant reduces tree height through coordinate compression but requires significant preprocessing overhead.
3. **For line segments:** The Segment LICT extends the structure to support interval-constrained lines with $O(\log^2 n)$ insertion time.
4. **For historical queries:** The Persistent LICT enables versioned access with $O(\log C)$ query time and $O(\log C)$ space per update.
5. **Static allocation:** The Static variant offers predictable memory usage but shows slightly worse performance than pointer-based implementations.

The Li-Chao Tree remains a fundamental data structure for competitive programming and algorithm design, particularly when lines may be inserted in arbitrary order (where the deque-based CHT cannot be used). Its logarithmic guarantees, flexibility in insertion order, and extensibility to persistence and segments make it an essential tool for practitioners.

Key Takeaway: When insertions are monotonic, use the deque CHT for $O(1)$ performance. When insertions are arbitrary, both the Dynamic CHT (balanced BST) and LICT provide $O(\log n)$ operations, with the LICT offering simpler implementation and better extensibility at a modest performance cost.

5.1 Why Use Li-Chao Tree?

Given that the Dynamic CHT is approximately 30–35% faster in our benchmarks, one might wonder why the Li-Chao Tree was invented and why it remains popular. The answer lies in several important factors beyond raw speed:

1. Simplicity and Correctness. The LICT implementation is significantly simpler than the Dynamic CHT. The Dynamic CHT requires:

- Careful handling of intersection point calculations using fractions or floating-point arithmetic.
- Proper handling of degenerate cases (parallel lines, duplicate slopes).
- Complex multiset manipulation to maintain the hull invariant.

In contrast, the LICT uses only integer comparisons of line values at specific points, making it easier to implement correctly and less prone to bugs.

2. Numerical Stability. The Dynamic CHT computes intersection points as $x = \frac{m_2 - m_1}{k_1 - k_2}$, which requires careful handling to avoid precision loss with integer coordinates or floating-point errors. The LICT avoids this entirely by comparing line values directly at integer coordinates.

3. Extensibility. The LICT's divide-and-conquer structure makes it naturally extensible to:

- **Line Segments:** The Segment LICT (Section 4.5) allows adding lines valid only on a subrange.
- **Persistence:** The Persistent LICT (Section 4.6) enables versioned access with minimal modification.
- **Higher Dimensions:** The approach generalizes to higher-dimensional query spaces.

Implementing these extensions in the Dynamic CHT would require significant restructuring of the hull maintenance logic.

4. Deterministic Complexity. The LICT guarantees $O(\log C)$ operations regardless of insertion order. The Dynamic CHT's $O(\log n)$ is amortized and depends on the complexity of the hull structure.

5. Historical Context. At the time of its invention (2012), competitive programming contests often favored simpler, more reliable implementations that could be coded quickly under time pressure. The LICT's straightforward logic made it an attractive alternative to the intricate Dynamic CHT.

References

1. Li Chao. *Lecture at Zhejiang Provincial Olympiad in Informatics (ZJOI 2012).* China, 2012. Original proposal of the data structure.
2. CP-Algorithms. "Li Chao Tree." `https://cp-algorithms.com/geometry/li_chao_tree.html`, 2024.
3. Codeforces. "Li Chao Tree Tutorial," by user *I_LOVE_TIGER*. `https://codeforces.com/blog/entry/51275`, 2017.
4. KACTL (KTH Algorithm Library). "LineContainer." `https://github.com/kth-competitive-programm kactl`, 2024.
5. Cormen, T.H., et al. *Introduction to Algorithms*, 4th Edition. MIT Press, 2022. Chapter on Dynamic Programming and Geometric Data Structures.