李超线段树：算法规范与分析

李超
chnlich@live.com

2026年3月10日

arXiv:2603.07948v1 [cs.DS] 9 Mar 2026

摘要

李超线段树（LICT）最初在讲义材料 [1] 中作为一种用于动态下包络维护的高效数据结构而提出。此后数年间，它已在竞赛编程社区中得到广泛应用，但同行评审文献中尚未出现正式规范。本文对李超线段树作出权威的形式化定义，既作为其官方规范，也扩展了原始讲义材料 [1]。我们给出完整的算法规范，建立形式化的正确性证明，分析理论复杂度，并提供实证性能评估。LICT 在实现简洁性、数值稳定性以及扩展到持久化和线段等高级变体方面具有独特优势。

1 引言

动态下包络维护是计算几何中的一个基础问题，具有广泛的应用。我们关注李超线段树（LICT），这是一种数据结构，用于维护一组线性函数，同时支持插入和查询操作。问题定义如下：给定一个动态的线性函数集合 $y = kx + b$，支持高效地插入新直线，并查询任意 x 坐标处的最小值（或等价地，最大值）。本文重点讨论最小值查询；最大值查询可通过对直线参数取负来实现。

形式化地说，我们需要一种支持以下两种操作的数据结构：

1. **添加直线：** 将一条新直线 $y = kx + b$ 插入数据结构。
2. **查询：** 给定 x_0，计算
$$\min_i \{k_i x_0 + b_i\}$$
在当前数据结构中的所有直线上取最小值。

对于最小值（或最大值）查询，LICT 为每次操作提供 $O(\log C)$ 的时间复杂度，其复杂度取决于坐标值域大小，而不是插入的直线数量。这里，

$$C = \frac{\text{coord坐标范围}}{\text{精度级别}}$$

表示坐标范围与精度级别的比值。对于范围为 $[0, 10^9]$, $C = 10^9$ 的整数坐标；对于相同范围、精度为 10^{-6}, $C = 10^{15}$ 的情况。

该数据结构除了支持无限直线外，还支持线段（仅在有限区间 $[x_l, x_r]$ 上定义的直线）。

2 相关工作

动态下包络维护有多种方法，它们在时间复杂度、实现复杂度和适用范围限制方面各有取舍。我们将回顾这些方案，为 LICT 提供背景。

几何构型的动态维护在计算几何中已得到广泛研究。

2.1 Overmars 和 van Leeuwen（1981年）

Overmars 和 van Leeuwen [2]提出了动态凸包维护方面的奠基性工作。其数据结构支持在维护下包络的同时对直线进行插入和删除，从而能够高效查询任意点处的最小值。

他们的方法使用平衡二叉搜索树显式维护凸包。每个节点存储一条直线，且树按斜率排序。计算相邻直线之间的交点，以确定凸包数据结构。查询需要 $O(\log\ N)$ 时间，而插入和删除需要 $O(\log^2 N)$ 时间。

2.2 单调凸包技巧

单调凸包技巧适用于直线斜率按单调递增或递减顺序插入的特殊情况 [3]。

当插入的斜率按单调递增（或递减）顺序排列时，基于双端队列的方法可使每次插入的摊销时间达到 $O(1)$，每次查询的摊销时间达到 $O(1)$。这种变体广泛用于动态规划优化，它以增量方式维护凸包，而无需使用平衡树结构。不过，单调性限制了它的适用范围，使其仅适用于直线斜率已知遵循特定顺序的问题。

2.3 动态凸包技巧

对于不受单调性约束的任意插入序列，平衡二叉搜索树会显式维护凸包 [4]。每次插入和查询都需要 $O(\log\ N)$ 的摊销时间。实现复杂度源于需要计算并维护凸包中相邻直线之间的交点。

动态 CHT 需要计算凸包中相邻直线之间的交点 as

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

因此需要谨慎处理精度（接近平行直线时）。

3 李超线段树

本节介绍李超线段树。

3.1 核心思想

李超线段树基于一个关于直线在区间上支配关系的基本观察。考虑定义在区间 $[l, r]$ 上的两条直线:

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

由于两条不同的直线至多相交一次，因此在区间至少一半的范围内，一条直线必定低于另一条。我们存储在中点 $m = \frac{l+r}{2}$ 处占优的直线，并将另一条直线推入包含交点的子节点。被推入的直线只需在原区间至多一半的范围内考虑。这个观察使递归分解成为可能：查询要么由已存储的直线回答，要么递归处理规模严格减半的子问题。

这种区间二分性质带来了对数查询时间，并构成了树结构的基础。

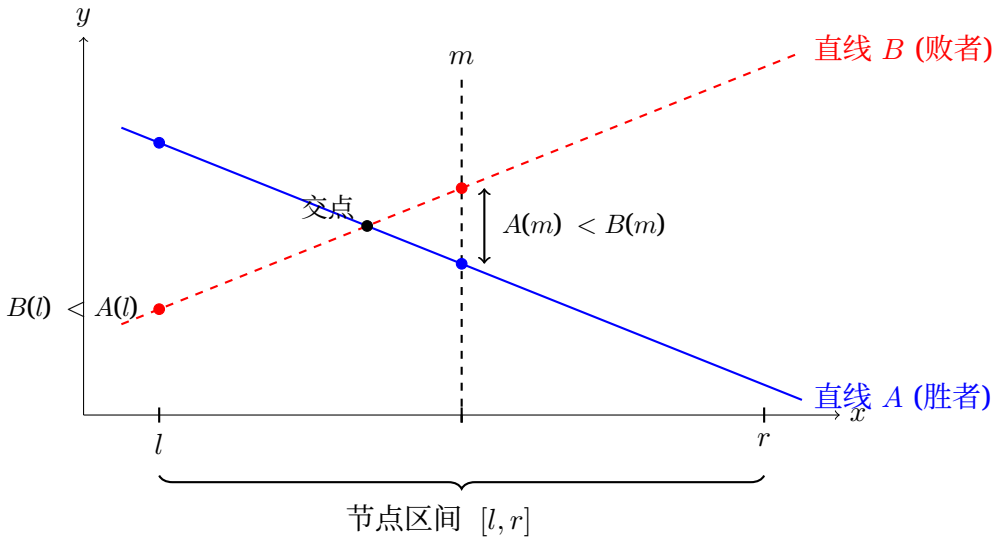


图 1：区间优势直线图。直线 A 在中点处胜出并被存储；直线 B（落败者）被路由到左子节点，在那里它可能是最优的。

请注意，这只是局部最优性。路由到其他分支的直线在中点处可能取得更低的值，因此存储的直线在该坐标处不一定是全局最优的。

3.2 算法方法

定义 1（李超线段树）。 定义域为 $[L, R]$ 的李超线段树 是一棵二叉树，其中：

- 每个节点拥有一个区间 $[l, r] \subseteq [L, R]$，其中点为 $m = (l + r)/2$，并且最多存储一条直线 $f(x) = kx + b$。
- **不变式**。 存储的直线在 m 处取得所有曾路由经过该节点的直线中的最小值。
- **子节点**。 左子节点拥有 $[l, m]$；右子节点拥有 $[m, r]$。子节点在首次插入时按需创建。
- **初始状态**。 空树是在 $[L, R]$ 上的单个空根节点。

LICT 提供了一种截然不同的方法，通过区间细分实现隐式包络维护。该数据结构不显式追踪凸包几何，而是在每个区间维护最佳直线，并递归划分查询范围。

我们首先考虑最简单、最常见的情形：所有被查询的 x坐标都是整数。在这种情况下，如果坐标范围为 $[0, C-1]$，其中 C 是某个正整数，那么树会递归地将该范围二分为子区间 $[l, r]$，其中 $l,\ r \in \mathbb{Z}$。每个叶节点都对应一个整数，因此树的深度为 $\lceil \log_2 C \rceil$，且每次查询都会落在唯一的叶节点上。这种整数情形是

在竞赛编程和许多算法设计应用中最常见的情形，我们在下文的示例中都采用这种情形。

备注（非整数查询）。通过细化离散化，同一数据结构也能处理非整数坐标处的查询。如果查询要求的精度为 ϵ（例如 $\epsilon = 10^{-6}$），可以将坐标范围按 $1/\epsilon$ 进行缩放，从而将每个精度步长视为一个单位。坐标全集大小变为 $C =$ 坐标范围$/\epsilon$，树深度也相应增长至 $\lceil \log_2 C \rceil$。所有算法细节保持不变；变化的只有坐标全集大小。例如，范围为$[0, 10^9]$ 且精度为 10^{-6} 时，得到 $C = 10^{15}$，树深度为 ≈ 50。

树中的每个节点表示一个区间 $[l, r]$，并存储一条直线。

插入。一条直线沿着从根节点到叶节点的单一路径在树中路由。如果当前节点为空，就直接存储新直线并终止路由。否则，两条候选直线会在该节点处进行比较：驻留直线（当前存储的直线）和待插入直线（新直线）。在中点处取值较小的直线

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

被指定为*胜者* 并存储在该节点；*败者* 则继续向下进入相应的子树。

由于胜者已在 m处确定，接下来要判断交点位于哪一半。如果败者在 l 处较低（在 l 处的排序与 m处不同），交点就位于 $[l, m]$，因此败者被路由到左子节点。否则，交点位于 $[m, r]$，败者被路由到右子节点。

当插入到达最大深度（$r - l < \epsilon$，其中 ϵ 是所需精度）时，区间已窄到无法区分任何两个查询点。存储的直线已经能够处理该区间内的所有查询，因此直接丢弃败者直线即可：无需进一步递归。对于整数坐标（$\epsilon = 1$），此条件可简化为 $l = r$。

算法 1 LICT 直线插入

要求：节点指针 $node$**，新直线** $new\ line$**，区间** $[l, r]$**1：如果** $node$ **为空，则 2：** $node \leftarrow$**新建节点(**$new\ line$**) 3：返回 4：结束 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：如果** $midf$**，则 9：交换(**$node.line$**,** $new\ line$**) 10：结束 if 11：如果** $r - l < \epsilon$**，则▷已达到最大深度；丢弃较差的直线 12：返回 13：结束 if 14：如果** $lef \neq midf$ **，则 15：**INSERT**(**$node.left$**,** $new\ line$**,** l**,** m**) 16：否则 17：**INSERT**(**$node.right$**,** $new\ line$**,** m**,** r**) 18：结束 if**

备注（平局处理）。当两条直线在 m处取得相同值时，第 8 行的严格 $<$比较会保留驻留直线，并将新插入直线向下推送。此约定是任意的，且不会影响正确性。重复直线（ k 和 b完全相同）是一种特殊情况：每当它遇到相同的驻留直线时，都不会发生交换，而是将重复直线路由到某个子节点。此过程会在遇到第一个空子节点（此时存储重复直线）或达到最大深度（此时将其丢弃）时终止。在这两种情况下，查询结果都不会改变。

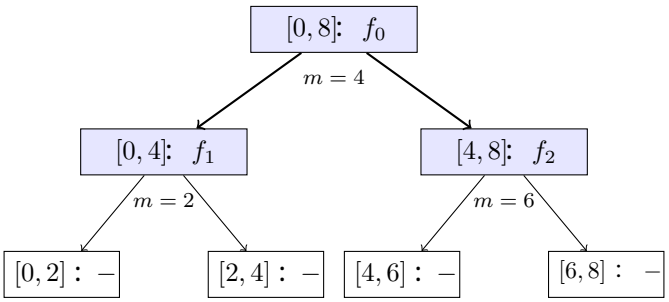


图 2：插入 $f_{new}(x) = -x + 10$ 之前的树结构。当前在根节点存储 $f_0(x) = x$，在 $[0, 4]$，存储 $f_1(x) = 2x - 4$，并在 $[4, 8]$ 存储 $f_2(x) = 0.5x + 2$。

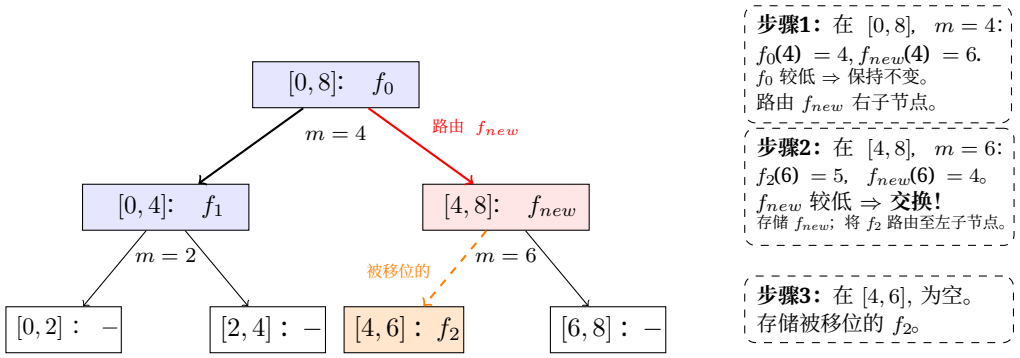


图3：插入 $f_{new}(x) = -x + 10$之后的树结构。在 $[4, 8]$，$f_{new}(6) = 4 < f_2(6) = 5$，因此 f_{new} 交换进入，而 f_2 被移位到 $[4, 6]$。

查询。我们沿路径遍历至 x_0，并评估路径上所有存储的直线。遇到的最小值等于 x_0 处的下包络，因为任何可能在 x_0 处最优的直线都存储在该路径上。

算法 2 LICT 查询

要求： 节点指针 $node$，查询坐标 x，区间 $[l, r]$**确保：** 坐标 x 处的最小值**1: 如果** $node$ **为空则 2: 返回** $+\infty$**3: 结束如果 4:** $m \leftarrow l + (r - l)/2$**5:** $val \leftarrow node.line.eval(x)$ **6: 如果** $r - l < \epsilon$ **则 7: 返回** val**8: 结束如果 9: 如果** $x \leq m$ **则 10: 返回 min(**val**,** QUERY**(**$node.left$**,** x**,** l**,** m**)) 11: 否则 12: 返回 min(**val**,** QUERY**(**$node.right$**,** x**,** m**,** r**)) 13: 结束如果**

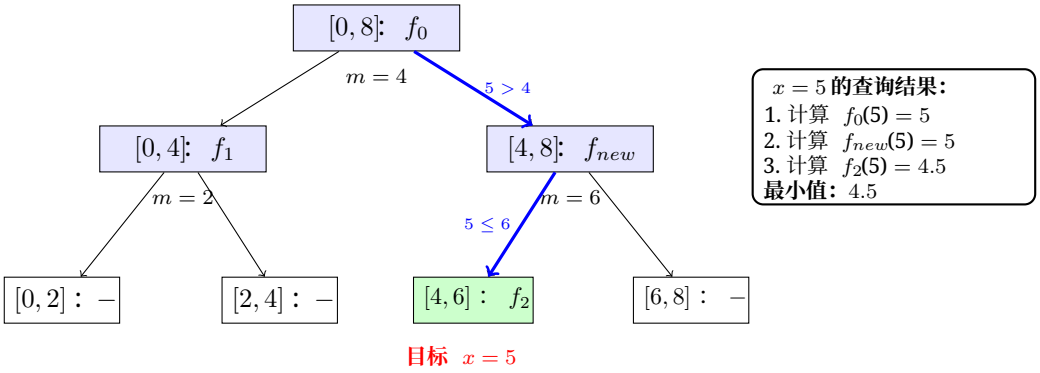


图4：查询树视图。要查询 $x = 5$ 处的值，我们沿着路径 $[0, 8] \to [4, 8] \to [4, 6]$ 前进。我们计算路径上存储的每条直线（f_0、f_{new} 和 f_2）的值，并返回最小值。

3.3 线段扩展

LICT 可自然地从完整直线扩展到线段。要插入定义在$[x_l, x_r]$上的线段，我们将该区间分解为隐式线段树的 $O(\log C)$ 典范区间 。每个典范区间对应一个节点，其范围完全包含在 $[x_l, x_r]$ 内。随后在每个这样的区间上独立调用算法 1，以插入限制在该子范围上的直线。

算法 3 LICT 线段插入

要求： 节点指针 $node$、直线 $line$、节点区间 $[l, r]$、线段边界 $[x_l, x_r]$

如果 $x_r < l$ **或** $r < x_l$ **则**
    返回 ▷ 无重叠
结束如果
如果 $x_l \le l$ **和** $r \le x_r$ **则**
    INSERT($node$, $line$, l, r) ▷ 算法 1
    返回
结束如果
$m \leftarrow l + (r - l)/2$
SEGMENTINSERT($node.left$, $line$, l, m, x_l, x_r)
SEGMENTINSERT($node.right$, $line$, m, r, x_l, x_r)

每次线段插入的总计代价为 $O(\log^2 C)$：区间 $[x_l, x_r]$ 分解为 $O(\log\ C)$ 个典范区间，且每次调用算法 1 耗时 $O(\log C)$。

3.4 理论分析

现在，我们将正式分析 LICT 的正确性和复杂度。**定义 2**（坐标全集）。 LICT 在一个离散域上运行，其中

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

树深度为 $h = \lceil \log_2 C \rceil$。

3.4.1 正确性

我们通过三个结果确立 LICT 的正确性：线性函数的一项基础性质、插入过程中维护的路由不变式，以及关于查询的全局正确性定理。

引理 1（线性支配）。 设 $f\ (x)\ = k_1 x + b_1$ 和 $g\ (x)\ = k_2 x + b_2$ 为线性函数。若 $f\ (a)\ \le g\ (a)$ 且 $f\ (b)\ \le g\ (b)$ 对于 $a \le b$成立，则对于所有 $x \in [a, b]$, $f\ (x)\ \le g\ (x)$ 成立。

证明。 设 $h(x) = f(x) - g(x)$。由于 h 是线性的，且 $h(a) \le 0$ 和 $h(b) \le 0$，对于任意 $x \in [a, b]$:

$$h(x) = h(a) \cdot \frac{b - x}{b - a} + h(b) \cdot \frac{x - a}{b - a} \le 0.$$ □

引理 2（路由支配）。 在区间为 $[l, r]$ 且中点为 m的节点处，如果保留直线 A （$A\ (m)\ \le B\ (m)$），并将直线 B 推送到一个子节点，则在另一个子节点的区间上， A 支配 B 。

证明。 不失一般性，假设将 B 推入左子节点，因此 $B(l) < A(l)$。由于$A(m) \le B(m)$ 且 $B(l) < A(l)$，两条直线在 $[l, m]$处相交。对于所有 $x \in [m, r]$，差值$A(x) - B(x)$ 与在 m处的符号相同，因此 $A(x) \le B(x)$。根据引理 1， A 在$[m, r]$上支配 B 。 □

定理 1 (查询正确性)。 对于任意查询点 x_0，查询操作返回的值等于数据结构中所有直线的最小值 $_i\{k_i x_0 + b_i\}$ 。

证明。 查询会遍历包含 x_0 的唯一根到叶路径 P，并返回存储的直线在 x_0 处的取值中的最小值。由于这些直线是所有插入的直线的子集，因此 query(x_0) $\ge \min_i L_i(x_0)$。接下来只需证明反向不等式。

反设 query(x_0) $>$ min$_i$ L_i(x_0)。设 L^* 是在 x_0 处取得全局最小值的直线。那么， P 上存储的任何直线在 x_0 处的取值都不等于 $\leq L^*(x_0)$；特别地， L^* 没有存储在 P 上的任何节点中。

由于 L^* 从根节点（在 P 上）进入，且在最终树中的 P 上没有存储，因此在某个节点 $w \in P$ 处，L^* 被推送到不包含 x_0 的子节点。根据引理 2，保留在 w 处的直线 S 满足 $S(x_0) \leq L^*(x_0)$，这与假设矛盾。

如果 S 后来被一条新直线 C 从 w 上移位：

- 如果 S 被路由至 x_0 的方向： S 沿着 P 向更深处移动，矛盾仍然成立。
- 如果 S 被路由至远离 x_0 的方向：根据引理 2， $C(x_0) \leq S(x_0) \leq L^*(x_0)$，且 C 位于 $w \in P$。如果 C 也被移位，则重复此过程。

由于移位次数有限，这条链最终会以矛盾告终。 □

引理 3（线段插入正确性）。 *算法 3 能正确维护定义在 $[x_l, x_r]$ 上的线段的下包络。*

证明。 区间 $[x_l, x_r]$ 可分解为隐式线段树的 $O(\log C)$ 个典范区间，每个区间对应一个范围完全位于 $[x_l, x_r]$ 内的节点。对于每个典范区间 $[l_i, r_i]$，算法 1 会插入限制在 $[l_i, r_i]$ 内的直线。根据定理 1 应用于以 $[l_i, r_i]$ 为根的子树，每次此类插入都能正确维护 $[l_i, r_i]$ 内的下包络。由于这些典范区间恰好覆盖 $[x_l, x_r]$ ，且插入不会影响 $[x_l, x_r]$ 之外的节点，因此整体下包络能够得到正确维护。 □

3.4.2 复杂度分析

命题 1（时间复杂度）。 *直线插入和查询操作需要 $O(\log C)$ 时间。线段插入（算法 3）需要 $O(\log^2 C)$ 时间。*

证明。 LICT 是一棵深度为

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

对于插入：新直线恰好沿着一条根到叶路径前进。在每个节点，我们执行：(1) $O(1)$ 次直线求值（$kx + b$）；(2) $O(1)$ 次比较；以及 (3) 可选地交换直线。每项操作耗时 $O(1)$。访问 $O(\log C)$ 个节点时，总时间为 $O(\log C)$。

对于查询：遍历沿一条根到叶路径进行，每个节点的工作量为 $O(1)$（直线求值与比较）。总计时间为 $O(\log C)$。

对于线段插入：线段 $[x_l, x_r]$ 分解为 $O(\log C)$ 个典范区间，每个区间以代价 $O(\log C)$ 调用算法 1。总计时间为 $O(\log C) \times O(\log C) = O(\log^2 C)$。 □

命题 2（空间复杂度）。 *LICT 在最坏情况下最多存储 $O(N)$ 个节点，其中 N 是插入的直线的数量。*

证明。 每次完整直线插入都沿单条根到叶路径进行。在每个已有节点处，算法执行 $O(1)$ 次比较，并恰好递归进入一个子节点。若路径到达空子节点，则创建一个新节点并返回；若路径终止于已有叶节点，则不创建新节点。因此每次插入最多创建一个新节点。经过 N 次插入，节点总数至多为 N，空间为 $O(N)$。

备注。 对于线段插入（算法 3），每条线段会分解为 $O(\log C)$ 个典范区间，每次插入最多创建 $O(\log C)$ 个节点，总共创建 $O(N \log C)$ 个节点。

□

删除。 标准 LICT 不支持高效删除。删除一条直线需要遍历所有可能存储该直线的节点，并根据后代节点重新计算最优直线；在最坏情况下，这需要 $\Omega(N)$ 时间。对于需要删除操作的应用，另一种方法是定期重构树。

LICT 的复杂度取决于坐标范围，而非直线数量。这使其性能不受包络大小影响，始终保持稳定；但当包络相对于插入次数一直较小时，它无法利用这一优势。

4 基准测试

参考实现和基准测试可在 https://github.com/chnlich/ lichao-tree 获取。

为了验证理论复杂度分析，并刻画 LICT 与动态凸包技巧在实际性能上的差异，我们针对不同的问题规模和输入分布进行了系统的实证评估。

4.1 实验设置

所有基准测试均采用以下配置进行。

硬件规格：

- **CPU：**AMD Ryzen 9 3950X 16 核处理器 @ 3.50GHz（Zen 2 微架构）
- **核心配置：**16 核，32 线程（已启用同步多线程）
- **L1 缓存：**512 KiB L1 数据缓存，512 KiB L1 指令缓存
- **L2 缓存：**8MiB
- **L3 缓存：**16 MiB（共享）
- **内存：** 64 GB 动态随机存取存储器
- **平台：** Linux x86 64（WSL2 虚拟化）

软件配置：

- **编译器：** g++ 11.4.0
- **优化标志：** -O3 -std=c++17 -Wall -Wextra
- **操作系统：** Ubuntu 22.04.5 LTS（WSL2，内核 6.6.87）

动态凸包技巧基准采用 KACTL LineContainer [4], ，这是一种基于 std::multiset 的实现，在竞赛编程中得到广泛采用。

实验参数：

- **测试规模：**10_5、10_6和 10_7 次操作
- **坐标范围：**（假设整数精度为 $\epsilon = 1$），其中 $x,\ k,\ b \in [-10^9, 10^9]$ $C = 10^9$
- **随机种子：**42

- **测量协议：**每项测试运行 10 次；报告的时间为平均值。方差较低（各次运行的变异系数为 $< 5\%$）。基准测试在隔离系统上运行，没有其他用户进程，以尽量减少计时噪声。

- **分布：**我们记作

$$X \sim U(a, b)$$

表示随机变量 X 是从区间 $[a, b]$上的连续均匀分布中抽取的。

– 随机： 斜率 $k \sim U(-10^9, 10^9)$，截距 $b \sim U(-10^9, 10^9)$。期望凸包大小：

$$\Theta(\log N).$$

根据点线对偶 [5]，，每条直线 $y = kx + b$映射为对偶点 (k, b)。这些直线的下包络大小等于对偶点集的下凸包大小，而该点集在矩形 $[-10^9, 10^9]^2$中均匀分布。根据雷尼和苏兰克定理 [6]，，凸多边形内 N 个均匀随机点的期望凸包顶点数为 $\Theta(\log\ N)$。

– 全部在凸包上： 约有 $N/2$ 条直线 $y = -(i+1)x + (i+1)^2$，对应 $i \in [0, \lfloor N/2 \rfloor - 1]$，其间穿插约 $N/2$ 次查询。所有插入的直线都会贡献于凸包，因为对偶点 $(-(i+1), (i+1)^2)$位于凸抛物线 $b = k^2$上，因此它们都是下凸包的顶点。

4.2 结果

表 1 展示了性能比较。

表 1：性能比较（时间单位：毫秒）

N	分布	算法	插入（毫秒）	查询（毫秒）	总计（毫秒）
10^5	随机	LICT	4.20	0.78	4.98
10^5	随机	动态凸包技巧	2.21	0.29	2.50
10^5	全部在凸包上	LICT	9.17	7.25	16.42
10^5	全部在凸包上	动态凸包技巧	7.88	7.00	14.88
10^6	随机	LICT	41.63	7.84	49.48
10^6	随机	动态凸包技巧	22.74	2.80	25.53
10^6	全部在凸包上	LICT	245.17	248.77	493.94
10^6	全部在凸包上	动态凸包技巧	294.86	304.05	598.91
10^7	随机	LICT	431.80	78.88	510.68
10^7	随机	动态凸包技巧	226.64	33.48	260.12
10^7	全部在凸包上	LICT	6643.19	7045.61	13688.80
10^7	全部在凸包上	动态凸包技巧	7558.58	7417.51	14976.10

4.3 参数匹配比较：$N = C$ 情形

理论复杂度分析表明，动态凸包技巧的时间复杂度为 $O(\log\ N)$，而 LICT 的时间复杂度为 $O(\log C)$。为了直接比较这两种情形，我们进行了实验，其中直线数量 N 等于坐标全集大小 C。这种配置与竞赛编程中常见的动态规划优化相符，其状态递推形式如下：

$$dp[i] = \min_{0 \le j < i} \{dp[j] + \text{cost}(j, i)\}$$

这里，cost(j,i) 表示关于 i 的线性函数。由于查询（索引 i）位于固定范围 $[0,N]$ 内，坐标全集大小 C实际上等于 N。

在这种静态值域设置下，可以采用迭代线段树 [7]实现（通常称为 ZKW 线段树），以消除递归开销；得益于连续内存布局，它还可能改善缓存局部性。因此，我们在此次比较中加入第三种变体 **ZKW 李超线段树**，以评估这种专门优化相较于标准 LICT 和动态凸包技巧的优势。

- **配置：** $N = C = 10^5$， $N = C = 10^6$，以及 $N = C = 10^7$
- **直线（随机）：** k， $b \sim U(-C/2,\ C/2)$
- **直线（全部在凸包上）：** $y = -(i+1)x + (i+1)^2$对 $i \in [0, N/2-1]$，在插入前打乱顺序
- **查询点：** $x \sim U(-C/2,\ C/2)$（随机），$x \sim U(0,\ N)$（全部在凸包上）
- **操作组合：** $N/2$ 次插入，随后进行 $N/2$次查询

表 2： $N = C$ 情形下的性能（时间单位：毫秒）

N = C	分布	算法	插入（毫秒）	查询（毫秒）	总计（毫秒）
10^5	随机	LICT	2.21	0.94	3.16
10^5	随机	ZKW 李超线段树	2.40	0.58	2.98
10^5	随机	动态凸包技巧	2.23	0.41	2.64
10^5	全部在凸包上	LICT	7.11	5.42	12.54
10^5	全部在凸包上	ZKW 李超线段树	5.32	1.70	7.02
10^5	全部在凸包上	动态凸包技巧	7.17	6.83	14.00
10^6	随机	LICT	26.54	9.46	36.00
10^6	随机	ZKW 李超线段树	27.52	6.59	34.11
10^6	随机	动态凸包技巧	23.57	3.77	27.33
10^6	全部在凸包上	LICT	127.28	177.07	304.35
10^6	全部在凸包上	ZKW 李超线段树	71.79	41.68	113.47
10^6	全部在凸包上	动态凸包技巧	187.77	260.54	448.31
10^7	随机	LICT	311.44	99.31	410.75
10^7	随机	ZKW 李超线段树	324.54	77.98	402.51
10^7	随机	动态凸包技巧	235.24	45.01	280.25
10^7	全部在凸包上	LICT	4022.59	5248.88	9271.48
10^7	全部在凸包上	ZKW 李超线段树	1685.15	1428.63	3113.78
10^7	全部在凸包上	动态凸包技巧	6163.38	6579.40	12742.78

4.4 分析

当 C 大于 N时，尽管 LICT 的理论复杂度为较大的 $O(\log\ C)$，其性能仍与动态凸包技巧相当。$\log C$与 $\log\ N$之间的绝对差异仍然不大（例如，$\log(10^9) \approx 30$ 与 $\log(10^7) \approx 23$），因此，LICT 由于采用无除法比较且无需进行树再平衡而具有较小的常数因子，这足以弥补额外的树深度。

在 $N = C$ 情形下，ZKW 李超线段树变体在稠密凸包输入上明显优于另外两种方法，其中 $N = C = 10^7$ 全点位于凸包上的基准测试的完成速度约为动态凸包技巧的 $4\times$ 倍。

LIC T 的无除法运算确保所有输入都能获得一致的数值精度。Fo r

对于涉及浮点坐标或极端数值范围的应用，LICT 通过避免交点计算，提供了动态凸包技巧所缺乏的稳健性保证。

5 结论

在以下场景中，LICT 是首选：

5.1 优势

需要支持线段。 当问题涉及线段（直线仅在子区间内有效），而非无限直线时，LICT 可自然地实现 $O(\log^2 C)$ 插入。动态凸包技巧也能支持线段，但所需机制要复杂得多。

需要持久化。 在 LICT 中，路径复制 [8] 十分直接：插入只会修改一条根到叶路径上的节点，因此复制这些节点即可创建一个新版本，同时与前一版本共享未修改的子树。由于需要跨版本维护凸包不变量，实现动态凸包技巧的持久化要复杂得多。

直线斜率接近。 处理斜率差异细微的直线时，LICT 的无除法操作可以避免数值不稳定性。

实现时间限制。 在竞赛编程等看重实现速度的场景中，LICT 的简单性具有明显优势。代码量更少，且无需处理几何边界情况，因此能够更快、更有把握地完成实现。

5.2 局限性

LICT 有若干局限性，影响了它的适用范围：

无法高效删除。 如上所述，标准 LICT 不支持删除单条直线。删除一条直线需要遍历所有可能存储该直线的节点，并根据后代节点重新计算最优直线；在最坏情况下，这需要 $\Omega(N)$ 的时间。

依赖坐标范围。 LICT 的复杂度取决于 C，即坐标范围除以精度所得的值。对于精度较高且坐标范围很大的情况（例如，覆盖整个可表示范围的 64 位浮点数）， C 可能会变得大到难以实际应用。

5.3 未来工作

未来研究与开发仍有几个方向：

删除支持。 为 LICT 开发高效的删除机制，将使其能够应用于需要移除直线的动态场景。可行的方法包括采用惰性删除并定期重构，或为节点添加额外的数据结构，以支持高效删除直线。

缓存高效变体。 与基于数组的表示相比，LICT 的基于指针的树结构缓存局部性较差。研究缓存无关或缓存感知变体，有望在不牺牲渐近保证的情况下提升实际性能。

并行实现。 LICT 的树结构天然支持并行查询，但插入本质上是顺序操作。开发支持并行插入且能保持正确性的并发 LICT 变体，将有益于多核应用。

高维扩展。虽然 LICT 可以自然地扩展到更高维度（维护超平面而非直线），但空间和时间复杂度会随维度呈指数级增长

。研究近似变体或降维技术，或许能扩大其适用范围。

致谢

本文是在 Anthropic 的 Claude Code 和 CharlieBot（https://github.com/chnlich/charlie-bot）的协助下完成的。CharlieBot 是一款支持多会话并集成 TEX 编辑功能的智能体框架。

作者感谢以下社区资源为向更广泛的受众传播李超线段树所作的贡献：CP-Algorithms（https://cp-algorithms.com/geometry/li_chao_tree.html）、I LOVE TIGER 在 Codeforces 上发布的教程（https://codeforces.com/blog/entry/ 51275）、KTH 算法竞赛模板库（KACTL，https://github.com/kth-competitive-programming/kactl）以及 OI Wiki（https://oi-wiki.org/ds/li-chao-tree/）。

参考资料

[1] 李超。线段树 [线段树]。讲义，杭州学军中学，中国，2012 年。见 https://github.com/chnlich/lichao-tree/blob/main/doc/slides.ppt。

[2] M.H. Overmars 与 J. van Leeuwen。平面中构型的维护。《计算机与系统科学杂志》，23(2):166–204，1981年。[3] 凸包优化与李超线段树。CP-Algorithms，2024年。URL https://cp-algorithms. com/geometry/convex_hull_trick.html。[4] KTH 算法竞赛模板库（KACTL）。GitHub，2024年。URL https: // github.com/kth-competitive-programming/kactl。[5] Mark de Berg、Otfried Cheong、Marc van Kreveld 与 Mark Overmars。德劳内三角剖分，第191–218页。Springer Berlin Heidelberg，柏林、海德堡，2008年。ISBN 978-3-540-77974-2。doi: 10.1007/978-3-540-77974-2 9。URL https://doi.org/10.1007/ 978-3-540-77974-2_9。.[6]A. R´enyi 与 R. Sulanke。关于凸¨包之 n 随¨机¨选取的点。杂志 *f¨ur* 概率论与相关领域，2(1):75–84，1963年。[7] 张昆伟。统计的力量 [统计学的力量]。讲座幻灯片，清华大学，中国，2010年。介绍了非递归（迭代）线段树，在中国竞赛编程社区中广为人知，称为 ZKW 线段树。[8] J.R. Driscoll、N. Sarnak、D.D. Sleator 与 R.E. Tarjan。使数据结构持久化。《计算机与系统科学杂志》，38(1):86–124，1989年。