Dynamic Convex Hulls for Simple Paths*

Bruce Brewer ✉ ⓘ
Kahlert School of Computing, University of Utah, Salt Lake City, UT 84112, USA

Gerth Stølting Brodal ✉ ⓘ
Department of Computer Science, Aarhus University, Aabogade 34, 8200 Aarhus N, Denmark

Haitao Wang ✉ ⓘ
Kahlert School of Computing, University of Utah, Salt Lake City, UT 84112, USA

──── **Abstract** ────

We consider the planar dynamic convex hull problem. In the literature, solutions exist supporting the insertion and deletion of points in poly-logarithmic time and various queries on the convex hull of the current set of points in logarithmic time. If arbitrary insertion and deletion of points are allowed, constant time updates and fast queries are known to be impossible. This paper considers two restricted cases where worst-case constant time updates and logarithmic time queries are possible. We assume all updates are performed on a deque (double-ended queue) of points. The first case considers the monotonic path case, where all points are sorted in a given direction, say horizontally left-to-right, and only the leftmost and rightmost points can be inserted and deleted. The second case assumes that the points in the deque constitute a simple path. Note that the monotone case is a special case of the simple path case. For both cases, we present solutions supporting deque insertions and deletions in worst-case constant time and standard queries on the convex hull of the points in $O(\log n)$ time, where n is the number of points in the current point set. The convex hull of the current point set can be reported in $O(h + \log n)$ time, where h is the number of edges of the convex hull. For the one-sided monotone path case, where updates are only allowed on one side, the reporting time can be reduced to $O(h)$, and queries on the convex hull are supported in $O(\log h)$ time. All our time bounds are worst case. In addition, we prove lower bounds that match these time bounds, and thus our results are optimal. For a quick comparison, the previous best update bounds for the simple path problem were amortized $O(\log n)$ time by Friedman, Hershberger, and Snoeyink [SoCG 1989].

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases Dynamic convex hull, convex hull queries, simple paths, path updates, deque

1 Introduction

Computing the convex hull of a set of n points in the plane is a classic problem in computational geometry. Several algorithms can compute the convex hull in $O(n \log n)$ time in the static setting. For example, Graham's scan [17] and Andrew's vertical sweep [2]. Andrew's algorithm can construct the convex hull in $O(n)$ time if the points are already sorted by either the x-coordinates or the y-coordinates. This has been generalized by Graham and Yao [18] and by Melkman [28] to construct the convex hull of a simple path in $O(n)$ time. Output-sensitive algorithms of $O(n \log h)$ time have also been achieved by Kirkpatrick and Seidel [26] and by Chan [9], where h is the size of the convex hull.

---

* A preliminary version of this paper appeared in *Proceedings of the 40th International Symposium on Computational Geometry (SoCG 2024)* [4].
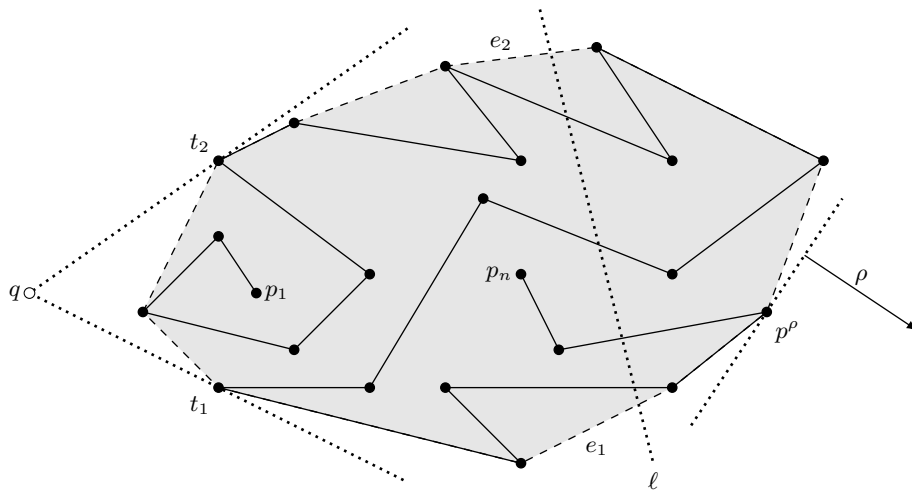
■ **Figure 1** The convex hull (dashed) of a simple path $p_1, \ldots, p_n$ (solid). Three types of convex hull queries are shown (dotted): the tangent points t_1 and t_2 with a query point q outside the convex hull; the extreme point p^ρ in direction ρ; and the two convex hull edges (bridges) e_1 and e_2 intersecting a line ℓ.

Overmars and van Leeuwen [30] studied the problem in the dynamic context where points can be inserted and deleted. Since a single point insertion and deletion can imply a linear change in the number of points on the convex hull, it is not desirable to report the entire convex hull explicitly after each update. Instead, one maintains a representation of the convex hull that can be queried. Overmars and van Leeuwen support the insertion and deletion of points in $O(\log^2 n)$ time, where n is the number of points stored. They maintain the points in sorted order in one dimension as the elements of a binary tree and bottom-up maintain a hierarchical decomposition of the convex hull. Since the convex hull is maintained explicitly as essentially a linked list at the root, the h points on the hull can be reported in $O(h)$ time, and queries on the convex hull can be supported in $O(\log n)$ time using appropriate binary searches. Some examples of convex hull queries are (see Figure 1): Determine whether a point q is outside the convex hull, and if yes, compute the tangents (i.e., find the tangent points) of the convex hull through q. Given a direction ρ, compute an extreme point on the convex hull along ρ. Given a line ℓ, determine whether ℓ intersects the convex hull, and if yes, find the two edges (bridges) on the convex hull intersected by ℓ. Tangent and extreme point queries are examples of *decomposable* queries, which are queries whose answers can be obtained in constant time from the query answers for any constant number of subsets that form a partition of the point set. In contrast, bridge queries are not decomposable.

Chan [10] improved the update (insertion/deletion) time to amortized $O(\log^{1+\varepsilon} n)$, for any $\varepsilon > 0$, by not maintaining an explicit representation of the convex hull. Tangent and extreme point queries are supported in $O(\log n)$ time, allowing the h points on the convex hull to be reported in $O(h \log n)$ time. The bridge query time was increased to $O(\log^{3/2} n)$. The update time was subsequently improved to amortized $O(\log n \log \log n)$ by Brodal and Jacob [6] and Kaplan, Tarjan, Tsioutsiouliklis [25], and to amortized $O(\log n)$ by Brodal and Jacob [7]. Chan [11] improved the time for bridge queries to $2^{O\left(\sqrt{\log \log n \log \log \log n}\right)} \log n$, with the same amortized update time. It is known that sub-logarithmic update time and logarithmic query time are not possible. For example, to achieve $O(\log n)$-time extreme point queries, an amortized update time of $\Omega(\log n)$ is necessary [6].

In this paper, we consider the dynamic convex hull problem for restricted updates, where we can achieve worst-case constant update time and logarithmic query time. In particular, we assume that the points are inserted and deleted in a *deque* (double-ended queue) and that they are geometrically restricted. We consider two restrictions: The first is the monotone path case, where all points in the deque are sorted in a given direction, say horizontally left-to-right, and only the leftmost and rightmost points can be inserted and deleted. The second case allows the points to form a simple path (i.e., a path that does not intersect itself), where updates are restricted to both ends of the path. The simple path problem was previously studied by Friedman, Hershberger, and Snoeyink [16], who supported deque insertions in amortized $O(\log n)$ time, deletions in amortized $O(1)$ time, and queries in $O(\log n)$ time. Bus and Buzer [8] considered a special case of the problem where insertions only happen to the "front" end of the path and deletions are only on points at the "rear" end. Based on the algorithm in [28], which can compute the convex hull of a simple path in linear time, they achieved $O(1)$ amortized update time to support $O(h)$-time hull reporting. However, hull queries were not considered in [8]. Wang [35] recently considered a special monotone path case where updates are restricted to queue-like updates, i.e., insert a point to the right of the point set and delete the leftmost point of the point set. Wang called these *window-sliding* updates and achieved amortized constant time updates, hull queries in $O(\log h)$ time,[1] and hull reporting in $O(h)$ time.

1.1 Our results

We present dynamic convex hull data structures for both the monotone path and the simple path variants. For both problems, we support deque insertions and deletions in worst-case constant time. We can answer extreme point, tangent, and bridge queries in $O(\log n)$ time, and we can report the convex hull in $O(h + \log n)$ time. For the *one-sided* monotone case, where updates are only allowed on one side, the reporting time can be reduced to $O(h)$, and convex hull queries are supported in $O(\log h)$ time. That is, they are only dependent on the current hull size and independent of the number of points in the set. In addition, we show that these time bounds are the best possible by proving matching lower bounds. The previous and new bounds for the various restricted versions of the dynamic convex hull problem are summarized in Table 1.

Our results are obtained by a combination of several ideas. To support deque updates, we partition the deque into left and right parts and treat these parts as two independent stack problems. Queries then need to compose the convex hull information from both the stack problems. This strategy has previously been used by Friedman, Hershberger, and Snoeyink [16] and by Wang [35]. To support deletions in the stack structures, we store rollback information when performing insertions. When one of the stacks becomes nearly empty, we repartition the deque into two new stacks of balanced sizes. To achieve worst-case bounds, the repartition is done with incremental global rebuilding ahead of time [29]. To achieve worst-case insertion time, we perform incremental merging of convex hull structures, where we exploit the fact that the convex hulls of two horizontally separated sets can be combined in worst-case $O(\log n)$ time [30] and that the convex hulls of a bipartition of a simple path can be combined in $O(\log^2 n)$ time [19]. To reduce the query bounds for the one-sided monotone path problem to be dependent on h instead of n, we adopt ideas from

---

[1] The runtime was $O(\log n)$ in the conference paper but was subsequently improved to $O(\log h)$ in the arXiv version https://arxiv.org/abs/2305.08055.

■ **Table 1** Known and new results for dynamic convex hull on paths. O_A are amortized time bounds. – denotes operation is not supported. For an update, h denotes the maximum size of the hull before and after the update. DL = delete left, IR = insert right, etc.

Reference	DL	IL	IR	DR	Queries	Reporting
No geometric restrictions						
Preparata [31] + rollback	–	–	$O(\log h)$	$O(\log h)$	$O(\log h)$	$O(h)$
Monotone path						
Andrews' sweep [2]	–	$O_A(1)$	$O_A(1)$	–	$O(\log h)$	$O(h)$
Wang [35]	$O_A(1)$	–	$O_A(1)$	–	$O(\log h)$	$O(h)$
New (Theorem 5)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(h + \log n)$
New (Theorem 6)	–	–	$O(1)$	$O(1)$	$O(\log h)$	$O(h)$
Simple path						
Friedman et al. [16]	$O_A(1)$	$O_A(\log n)$	$O_A(\log n)$	$O_A(1)$	$O(\log n)$	–
Bus and Buzer [8]	$O_A(1)$	–	$O_A(1)$	–	–	$O(h)$
New (Theorem 7)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(h + \log n)$

Sundar's priority queue with attrition [33]. In particular, we partition the stack of points into four lists (possibly with some interior "redundant" points removed), of which three lists are in convex position, and three lists have size $O(h)$. We believe this idea is interesting in its own right as, to our knowledge, this is the first time Sundar's approach has been used to solve a geometric problem.

1.2   Other related work

Andrew's static algorithm [2] is an incremental algorithm that explicitly maintains the convex hull of the points considered so far. It can add the next point to the right and left of the convex hull in amortized constant time.

Preparata [31] presented an insertion-only solution maintaining the convex hull in an AVL tree [1] that supports the insertion of an arbitrary point in $O(\log h)$ time, queries on the convex hull in $O(\log h)$ time, and reporting queries in $O(h)$ time. For the *stack* version of the dynamic convex hull problem, where updates form a stack, a general technique to support deletions is by having a stack of rollback information, i.e., the changes performed by the insertions. The time bound for deletions will then match the time bound for insertions, provided that insertion bounds are worst-case. Applying this idea to [31], we have a stack dynamic convex hull solution with $O(\log h)$-time updates. Note that these time bounds hold for arbitrary new points inserted without geometric restrictions. The only limitation is that updates form a stack.

Hershberger and Suri [23] considered the offline version of the dynamic convex hull problem, assuming the sequence of insertions and deletions is known in advance, supporting updates in amortized $O(\log n)$ time. Hershberger and Suri [22] also considered the semi-dynamic deletion-only version of the problem, supporting initial construction and a sequence of n deletions in $O(n \log n)$ time.

Given a simple path of n vertices, Guibas, Hershberger, and Snoeyink [19] considered the problem of processing the path into a data structure so that the convex hull of a query subpath (specified by its two ends) can be (implicitly) constructed to support queries on the convex hull. Using a compact interval tree, they gave a data structure of $O(n \log \log n)$ space with $O(\log n)$ query time. The space was recently improved to $O(n)$ by Wang [36]. There are also other problems in the literature regarding convex hulls for simple paths. For example,

Hershberger and Snoeyink [21] considered the problem of maintaining convex hulls for a simple path under split operations at certain extreme points, which improves the previous work in [13].

Notation. We define some notation that will be used throughout the paper. For any compact subset R of the plane (e.g., R is a set of points or a simple path), let $\mathcal{H}(R)$ denote the convex hull of R and let $|\mathcal{H}(R)|$ denote the number of vertices of $\mathcal{H}(R)$. We also use ∂R to denote the boundary of R.

For a dynamic set P of points, we define the following five operations: INSERTRIGHT: Insert a point to P that is to the right of all of the points of P; DELETERIGHT: Delete the rightmost point of P; INSERTLEFT: Insert a point to P that is to the left of all the points of P; DELETELEFT: Delete the leftmost point of P; HULLREPORT: Report the convex hull $\mathcal{H}(P)$ (i.e., output the vertices of $\mathcal{H}(P)$ in cyclic order around $\mathcal{H}(P)$). We also use STANDARDQUERY to refer to standard queries on $\mathcal{H}(P)$. This includes all decomposable queries like extreme point queries and tangent queries. It also includes certain non-decomposable queries like bridge queries. Other queries, such as deciding if a query point is inside $\mathcal{H}(P)$, can be reduced to bridge queries.

We define the operations for the dynamic simple path π similarly. For convenience, we call the two ends of π the *rear end* and the *front end*, respectively. Therefore, instead of "left" and "right", we use "rear" and "front" in the names of the update operations. This means that we have the following four updates: INSERTFRONT, DELETEFRONT, INSERTREAR, and DELETEREAR, in addition to HULLREPORT and STANDARDQUERY as above.

Outline. We present our algorithms for the monotone path problem in Section 2 and for the simple path problem in Section 3. In Section 4, we prove lower bounds that match our results in the previous two sections.

2 The monotone path problem

In this section, we study the monotone path problem where updates occur only at the extremes in a given direction, say, the horizontal direction. That is, given a set of points $P \subset \mathbb{R}^2$, we maintain the convex hull of P, denoted by $\mathcal{H}(P)$, while points to the left and right of P may be inserted into P and the rightmost and leftmost points of P may be deleted from P. Throughout this section, we let n denote the size of the current set P and $h = |\mathcal{H}(P)|$. For ease of exposition, we assume that no three points of P are collinear.

If updates are allowed at both sides (resp., at one side), we denote it the *two-sided* (resp. *one-sided*) problem. We call the structure for the two-sided problem the "deque convex hull", where we use the standard abbreviation deque to denote a double-ended queue (according to Knuth [27, Section 2.2.1], E. J. Schweppe introduced the term deque). The one-sided problem's structure is called the "stack convex hull".

In what follows, we start with describing a "stack tree" in Section 2.1, which will be used to develop a "deque tree" in Section 2.2. We will utilize the deque tree to implement the deque convex hull in Section 2.3 for the two-sided problem. The deque convex hull, along with ideas from Sundar's priority queues with attrition [33], will be used for constructing the stack convex hull in Section 2.4 for the one-sided problem.

2.1   Stack tree

Suppose P is a set of n points in $\mathbb{R}^2$ sorted from left to right. Consider the following operations on P (assuming $P = \emptyset$ initially). (1) INSERTRIGHT; (2) DELETERIGHT; (3) TREERETRIEVAL: Return the root of a balanced binary search tree (BST) that stores all points of the current P in the left-to-right order. We have the following lemma.

▶ **Lemma 1.** *Let P be an initially empty set of points in $\mathbb{R}^2$ sorted from left to right. There exists a "Stack Tree" data structure $ST(P)$ for P supporting the following operations:*
1. *INSERTRIGHT: $O(1)$ time.*
2. *DELETERIGHT: $O(1)$ time.*
3. *TREERETRIEVAL: $O(\log n)$ time, where n is the current size of P.*

   In what follows, we describe the tree's structure and then discuss the operations. Unless otherwise stated, P refers to the current point set, and $n = |P|$.

Remark.   It should be noted that the statement of Lemma 1 is not new. Indeed, one can simply use a finger search tree [5, 20, 34] to store P to achieve the lemma (in fact, TREE-RETRIEVAL can even be done in $O(1)$ time). We propose a stack tree as a new implementation for the lemma because it can be applied to our dynamic convex hull problem. When we use the stack tree, TREERETRIEVAL will be used to return the root of a tree representing the convex hull of P; in contrast, simply using a finger search tree cannot achieve the goal (the difficulty is how to efficiently maintain the convex hull to achieve constant time update). Our stack tree may be considered a framework for Lemma 1 that potentially finds other applications as well.

Structure of the stack tree.   The structure of the stack tree is illustrated in Figure 2 to the right of the line ℓ. The stack tree $ST(P)$ consists of a sequence of trees T_i for $i = 0, 1, \ldots, \lceil \log \log n \rceil$. Each T_i is a balanced BST storing a contiguous subsequence of P such that for any $j < i$, all points of T_i are to the left of each point of T_j. The points of all T_i form a partition of P. We maintain the invariant that $0 \le |T_i| \le 2^{2^{i+1}}$, and for $i \ge 1$, $|T_i|$ is a multiple of 2^{2^i}, where $|T_i|$ represents the number of points stored in T_i.

   In order for our construction to achieve worst-case constant time insertions, the joining of two trees is performed incrementally over subsequent insertions. Specifically, we apply the *recursive slowdown* technique of Kaplan and Tarjan [24], where every 2^{i+1}-th insertion, $i \ge 1$, performs delayed incremental work toward joining T_{i-1} with T_i, if such a join is deemed necessary.

Remark.   The critical observation of our algorithm is that because the ranges of the trees do not overlap, we can join adjacent trees T_i and T_{i+1} to obtain (the root) of a new balanced BST that stores all points in $T_i \cup T_{i+1}$ in $O(\log(|T_i| + |T_{i+1}|))$ time. Later in the paper, we generalize this idea to horizontally neighboring convex hulls which can be merged in $O(\log(|\mathcal{H}(T_i)| + |\mathcal{H}(T_{i+1})|))$ time [30] and to convex hulls over consecutive subpaths of a simple path which can be merged in $O(\log |\mathcal{H}(T_i)| \cdot \log |\mathcal{H}(T_{i+1})|)$ time [19]. In fact, the stack tree may be applied to solve other problems with polylogarithmic joining time so that TREERETRIEVAL (or other operations) can be performed efficiently.

InsertRight.   Suppose we wish to insert into P a point p that is to the right of all points of P.
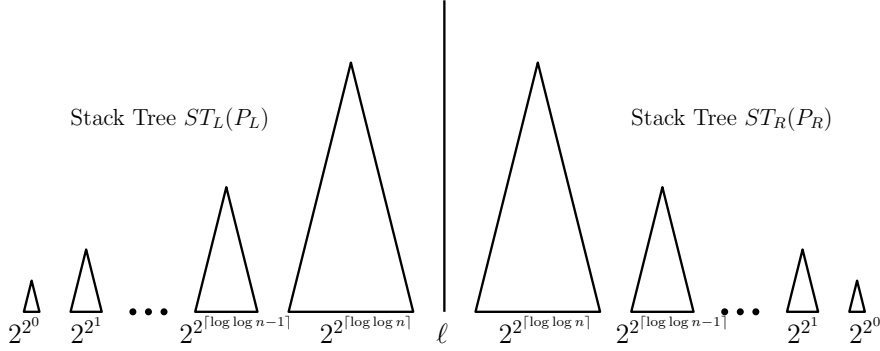
Figure 2 Illustrating a deque tree, comprising two stack trees separated by the vertical line ℓ.

We start with inserting p into the tree T_0, which takes $O(1)$ time as $|T_0| = O(1)$. Next, we perform $O(1)$ delayed incremental work on a tree T_i for a particular index i. To determine i, we maintain a counter N that is a binary number. Initially, $N = 1$, and it is an invariant that $N = 1 + n$ (recall that n is the size of the current point set P). For each insertion, we increment N by one and determine the index i of the digit that flips from 0 to 1, indexed from the right where the rightmost digit has index 0. Note that there is exactly one such digit. Then, if $i \geq 1$, we perform incremental work on T_i (i.e., joining T_{i-1} with T_i). To find the digit i in $O(1)$ time, we represent N by a sequence of ranges, where each range represents a contiguous subsequence of digits of 1's in N. For example, if N is 101100111, then the ranges are $[0, 2], [5, 6], [8, 8]$. After N is incremented by one, N becomes 101101000, and the ranges become $[3, 3], [5, 6], [8, 8]$. Therefore, based on the first two ranges in the range sequence, one can determine the digit that flips from 0 to 1 and update the range sequence in $O(1)$ time (note that this can be easily implemented using a linked list to store all ranges without resorting to any bit tricks).

After i is determined, we perform incremental work on T_i as follows. We use a variable n_j to maintain the size of each tree T_j, i.e., $n_j = |T_j|$. For each tree T_j, with $j \geq 1$, we say that T_j is "blocked" if there is an incremental process for joining the previous T_{j-1} with T_j (more details to be given later) and "unblocked" otherwise (T_0 is always unblocked). If T_i is blocked, then the incremental process joining the previous T_{i-1} with T_i will complete within time linear in the height of T_i, which is $O(2^i)$, since $|T_i| \leq 2^{2^{i+1}}$. We perform the next c steps for the process for a sufficiently large constant c. If the joining process is completed within the c steps, we set T_i to be unblocked.

Next, if T_i is unblocked and $n_i \geq 2^{2^{i+1}}$ (in this case we prove in Observation 2 that n_i must be exactly equal to $2^{2^{i+1}}$), then our algorithm maintains the invariant that T_{i+1} must be unblocked, which will be proved in Lemma 3. In this case, we first set T_{i+1} to be blocked, and then we start an incremental process to join T_i with T_{i+1} without performing any actual steps. For reference purpose, let T_i' refer to the current T_i and let T_i start over from $\emptyset$. Using this notation, we are actually joining T_i' with T_{i+1}. Although the joining process has not been completed, we follow the convention that T_i' is now part of T_{i+1}; hence, we update $n_{i+1} = n_{i+1} + n_i$. Also, since T_i is now empty, we reset $n_i = 0$. This finishes the work due to the insertion of p.

▶ **Observation 2.** **1.** *If $n_i \geq 2^{2^{i+1}}$, then $n_i = 2^{2^{i+1}}$.*
2. *It holds that $n_i = 0$ or $2^{2^i} \leq n_i \leq 2^{2^{i+1}}$ for $i \geq 1$, and $n_0 \leq 4$.*

Proof. According to our algorithm, T_0 is never blocked and $n_0 = 4$ whenever $N = 1 + 4k$ for $k = 1, 2, 3, \ldots$, where n_0 is reset to 0 (this is because whenever $n_0 \geq 4$, T_1 must be unblocked,

which is proved in Lemma 3). Hence, $n_0 \leq 4$ always holds. In the following, we prove the case for $i \geq 1$.

Observe that starting from $n_i = 0$, whenever n_i increases, it increases by exactly the amount 2^{2^i}. Hence, whenever $n_i \geq 2^{2^{i+1}}$ for the first time, it always holds that $n_i = 2^{2^{i+1}}$. According to our algorithm, whenever $n_i \geq 2^{2^{i+1}}$, n_i is reset to 0 (this is because whenever $n_i \geq 2^{2^{i+1}}$, T_{i+1} must be unblocked, which is proved in Lemma 3). The observation thus follows.   ◄

The following lemma proves the algorithm invariant mentioned above.

▶ **Lemma 3.**
1. *If $n_0 \geq 4$, then T_1 must be unblocked.*
2. *If $i \geq 1$ and $n_i \geq 2^{2^{i+1}}$ right after the process of joining T_{i-1} with T_i is completed, then T_{i+1} must be unblocked.*

Proof. Recall that N equals one plus the number of insertions performed, i.e., N can be considered a timer.

Proof of the first lemma statement.   We have $n_0 = 4$ exactly when $N = 1 + 4k$ for $k = 1, 2, 3, \ldots$, where n_0 is reset to 0, an incremental join at T_1 is initiated, and T_1 becomes blocked. But for the next insertion, i.e., when $N = 2 + 4k$ for $k = 1, 2, 3, \ldots$, we have always $n_0 = 1$ and all the incremental $O(1)$ work at T_1 is processed if we make c large enough, and T_1 becomes unblocked and remains unblocked until $n_0 = 4$ again. This proves the first lemma statement.

Proof of the second lemma statement.   We assume that the process of joining T_{i-1} with T_i is completed due to the insertion of p and $n_i \geq 2^{2^{i+1}}$.

Let N' be the most recent timer where T_{i+1} changed status from unblocked to blocked. Hence, at N', $T_i = \emptyset$, and the process of joining T_i' with T_{i+1} started at N' and T_{i+1} has been blocked due to that process. We argue below that by the time N, the incremental process for joining T_i' with T_{i+1} has already been completed.

Indeed, the joining process takes $O(2^{i+1})$ time as the height of T_{i+1} is $O(2^{i+1})$. Whenever the $(i+1)$-th bit of the counter N flipped from 0 to 1, c steps of the joining process will be performed. Note that the $(i+1)$-th bit of the counter N flipped from 0 to 1 every 2^{i+2} insertions. Hence, for a sufficiently large constant c, the joining process will be completed within $2^{i+1} \cdot 2^{i+2} = 2^{2i+3}$ insertions. If we make c larger, then we can say that the joining process will be completed within 2^{2i} insertions.

On the other hand, at N', we have $T_i = \emptyset$. We argue that after 2^{2i} insertions, the size of T_i cannot be larger than or equal to $2^{2^{i+1}}$. Indeed, all points of T_i are originally from T_j for all $0 \leq j < i$. At N', the total number of points in all trees T_j for $0 \leq j < i$ is no larger than $2 \cdot 2^{2^i}$ by Observation 2(2). Hence, after 2^{2i} insertions, the total number of points that can be inserted into T_i is at most $2 \cdot 2^{2^i} + 2^{2i}$, which is smaller than $2^{2^{i+1}}$ for $i \geq 1$.

The above implies that by the time N when $n_i \geq 2^{2^{i+1}}$, the incremental process for joining T_i' with T_{i+1} has already been completed, and therefore, T_{i+1} cannot be blocked. This proves the second lemma statement.   ◄

As we only perform $O(1)$ incremental work, the total time for inserting p is $O(1)$.

DeleteRight. To perform DELETERIGHT, we maintain a stack that records the changes made on each insertion. To delete a point p, p must be the most recently inserted point, and thus all changes made due to the insertion of p are at the top of the stack. To perform the deletion, we pop the stack and roll back all the changes made during the insertion of p.

TreeRetrieval. To perform TREERETRIEVAL, we start by completing all incremental joining processes. Then, we join all trees T_i in their index order. This results in a single BST T storing all points of P. In applications, we usually need to perform binary searches on T, after which we need to continue processing insertions and deletions on P. To this end, when constructing T as above, we maintain a stack that records the changes we have made. Once we are done with queries on T, we use the stack to roll back the changes and return the stack tree to its original form right before the TREERETRIEVAL operation.

For the time analysis, it takes $O(2^i)$ steps to finish an incremental joining process for each T_i (i.e., merging T_{i-1} with T_i). In total, it takes $O\left(\sum_{i=1}^{\lceil \log \log n \rceil} 2^i\right) = O(\log n)$ time to finish all such processes. Next, joining all trees T_i in their index order takes $O\left(\sum_{i=0}^{\lceil \log \log n \rceil} 2^i\right) = O(\log n)$ time in total. Consequently, the stack used to record changes made during the operation has size $O(\log n)$ because it stores $O(\log n)$ changes. Rolling back all changes in the stack thus takes $O(\log n)$ time as well. Therefore, TREERETRIEVAL can be performed in $O(\log n)$ time.

2.2 Deque tree

We now introduce the deque tree, which is built upon stack trees. We have the following lemma, where TREERETRIEVAL is defined in the same way as in Section 2.1.

▶ **Lemma 4.** *Let P be an initially empty set of points in $\mathbb{R}^2$ sorted from left to right. There exists a "Deque Tree" data structure $DT(P)$ for P supporting the following operations:*
1. *INSERTRIGHT: $O(1)$ time.*
2. *DELETERIGHT: $O(1)$ time.*
3. *INSERTLEFT: $O(1)$ time.*
4. *DELETELEFT: $O(1)$ time.*
5. *TREERETRIEVAL: $O(\log n)$ time, where n is the size of the current set P.*

As with Lemma 1, the statement of Lemma 4 is not new because we can also use a finger search tree [5, 20] to achieve it. Here, we propose a different implementation for solving our dynamic convex hull problem.

The deque tree $DT(P)$ is built from two stack trees $ST_L(P_L)$ and $ST_R(P_R)$ from opposite directions, where P_L and P_R refer to the subsets of P to the left and right of a vertical dividing line ℓ, respectively (see Figure 2). To insert a point to the left of P, we insert it to $ST_L(P_L)$. To delete the leftmost point of P, we delete it from $ST_L(P_L)$. For insertion/deletion on the right side of P, we use $ST_R(P_R)$. For TREERETRIEVAL, we perform TREERETRIEVAL operations on both $ST_L(P_L)$ and $ST_R(P_R)$, which produces two balanced BSTs; then, we join these two trees into a single one. The time complexities of all these operations are as stated in the lemma.

To make this idea work, we need to make sure that neither $ST_L(P_L)$ nor $ST_R(P_R)$ is empty when n is sufficiently large (if $n < c$ for a constant c, then we can do everything by brute force). To this end, we apply incremental global rebuilding [29, Section 5.2.2], where we dynamically adjust the dividing line ℓ. Specifically, we maintain an invariant that $|P_L|$ and $|P_R|$ differ by at most a factor of 4, i.e., $\frac{1}{4} \le \frac{|P_L|}{|P_R|} \le 4$. To achieve this, when the condition

$\frac{1}{2} \leq \frac{|P_L|}{|P_R|} \leq 2$ is first violated after an update, we set ℓ' to be the vertical line partitioning the current P into two equal-sized subsets. Once ℓ' is set, we begin building $ST_L(P'_L)$ on the subset P'_L of points of P to the left of ℓ' and $ST_R(P'_R)$ on the subset P'_R of points to the right of ℓ'. It takes $O(n)$ time to build $ST_L(P'_L)$ and $ST_R(P'_R)$, and there will be $\Omega(n)$ updates (e.g., at least $\frac{n}{6}$ updates, where n is the size of $|P|$ when the rebuilding procedure starts) between the time when the condition $\frac{1}{2} \leq \frac{|P_L|}{|P_R|} \leq 2$ is first violated and the time when the condition $\frac{1}{4} \leq \frac{|P_L|}{|P_R|} \leq 4$ is first violated, so we can perform $O(1)$ incremental work to progressively build $ST_L(P'_L)$ and $ST_R(P'_R)$ for the next $O(n)$ updates (e.g., the next $\frac{n}{6}$ updates). As soon as the above second condition is met, if ever, we replace $ST_L(P_L)$ and $ST_R(P_R)$ with $ST_L(P'_L)$ and $ST_R(P'_R)$. This returns us to the state where $\frac{1}{2} \leq \frac{|P_L|}{|P_R|} \leq 2$ (in fact, when the rebuilding terminates, we have a stronger guarantee $\frac{2}{3} \leq \frac{|P_L|}{|P_R|} \leq \frac{3}{2}$).

2.3   Two-sided monotone path dynamic convex hull

We can tackle the two-sided monotone path dynamic convex hull problem using the deque tree. Suppose P is a set of n points in $\mathbb{R}^2$. In addition to the operations INSERTRIGHT, DELETERIGHT, INSERTLEFT, DELETELEFT, HULLREPORT, as defined in Section 1, we also consider the operation HULLTREERETRIEVAL: Return the root of a BST of height $O(\log h)$ that stores all vertices of the convex hull $\mathcal{H}(P)$ (so that binary search based operations on $\mathcal{H}(P)$ can all be supported in $O(\log h)$ time). We will prove the following theorem.

▶ **Theorem 5.** *Let $P \subset \mathbb{R}^2$ be an initially empty set of points, with $n = |P|$ and $h = |\mathcal{H}(P)|$. There exists a "Deque Convex Hull" data structure $DH(P)$ of $O(n)$ space that supports the following operations:*
1. *INSERTRIGHT: $O(1)$ time.*
2. *DELETERIGHT: $O(1)$ time.*
3. *INSERTLEFT: $O(1)$ time.*
4. *DELETELEFT: $O(1)$ time.*
5. *HULLTREERETRIEVAL: $O(\log n)$ time.*
6. *HULLREPORT: $O(h + \log n)$ time.*

Remark.   The time complexities of the four update operations in Theorem 5 are obviously optimal. We will show in Section 4 that the other two operations are also optimal. In particular, it is not possible to reduce the time of HULLTREERETRIEVAL to $O(\log h)$ or reduce the time of HULLREPORT to $O(h)$ (but this is possible for the one-sided case as shown in Section 2.4).

The deque convex hull is a direct application of the deque tree from Section 2.2. We maintain the upper hull and lower hull of $\mathcal{H}(P)$ separately. In the following, we only discuss how to maintain the upper hull, as maintaining the lower hull is symmetric. By slightly abusing the notation, let $\mathcal{H}(P)$ refer to the upper hull only in the following discussion.

We use a deque tree $DT(P)$ to maintain $\mathcal{H}(P)$. The $DT(P)$ consists of two stack trees ST_L and ST_R. Each stack tree is composed of a sequence of balanced search trees; each such tree T_i stores left-to-right the points of the convex hull $\mathcal{H}(P')$ for a contiguous subsequence P' of P. We follow the same algorithm as the deque tree with the following changes. During the process of joining T_{i-1} with T_i, our task here becomes merging the two hulls stored in the two trees. To perform the merge, we first compute the upper tangent of the two hulls. This can be done in $O(\log(|T_{i-1}| + |T_i|))$ time using the method of Overmars and van Leeuwen [30]. Then, we split the tree T_{i-1} into two portions at the tangent point; we do the same for T_i. Finally,

we join the relevant portions of the two trees into a new tree that represents the merged hull of the two hulls. The entire procedure takes $O(\log(|T_{i-1}| + |T_i|))$ time. This time complexity is asymptotically the same as joining two trees T_{i-1} and T_i as described in Section 2.1, and thus we can still achieve the same performances for the first five operations as in Lemma 4; in particular, to perform HULLTREERETRIEVAL, we simply call TREERETRIEVAL on the deque tree. Finally, for HULLREPORT, we first perform HULLTREERETRIEVAL to obtain a tree representing $\mathcal{H}(P)$. Then, we perform an in-order traversal on the tree, which can output $\mathcal{H}(P)$ in $O(h)$ time. Thus, the total time for HULLREPORT is $O(h + \log n)$.

2.4 One-sided monotone path dynamic convex hull

We now consider the one-sided monotone problem. Suppose P is a set of n points in $\mathbb{R}^2$. Consider the following operations on P (assume that $P = \emptyset$ initially): INSERTRIGHT, DELETE-RIGHT, HULLTREERETRIEVAL, HULLREPORT, as in Section 2.3. Applying Theorem 5, we can perform HULLTREERETRIEVAL in $O(\log n)$ time and perform HULLREPORT in $O(h + \log n)$ time. We will prove the following theorem, which reduces the HULLTREERETRIEVAL time to $O(\log h)$ and reduces the HULLREPORT time to $O(h)$.

▶ **Theorem 6.** *Let $P \subset \mathbb{R}^2$ be an initially empty set of points, with $n = |P|$ and $h = |\mathcal{H}(P)|$. There exists a "Stack Convex Hull" data structure $SH(P)$ of $O(n)$ space that supports the following operations:*
1. *INSERTRIGHT: $O(1)$ time.*
2. *DELETERIGHT: $O(1)$ time.*
3. *HULLTREERETRIEVAL: $O(\log h)$ time.*
4. *HULLREPORT: $O(h)$ time.*

The main idea to prove Theorem 6 is to adapt ideas from Sundar's algorithm in [33] for priority queue with attrition as well as the deque convex hull data structure from Section 2.3. As in Section 2.3, we maintain the upper and lower hulls of $\mathcal{H}(P)$ separately. In the following, we only discuss how to maintain the upper hull. By slightly abusing the notation, let $\mathcal{H}(P)$ refer to the upper hull only in the following discussion.

2.4.1 Structure of the stack convex hull

For any two disjoint subsets P_1 and P_2 of P, we use $P_1 \prec P_2$ to denote the case where all points of P_1 are to the left of each point of P_2. Our data structure maintains four subsets $A_1 \prec A_2 \prec A_3 \prec A_4$ of P. Each A_i, $1 \leq i \leq 3$, is a convex chain, but this may not be true for A_4. Further, the following invariants are maintained during the algorithm. See Figure 3.

1. Vertices of $\mathcal{H}(P)$ are all in $\bigcup_{i=1}^{4} A_i$.
2. A_1 is a prefix of the vertices of $\mathcal{H}(P)$ sorted from left to right.
3. $A_1 \cup A_2$ and $A_1 \cup A_3$ are both convex chains.
4. BIAS: $|A_1| \geq |A_3| + 2 \cdot |A_4|$.

The partition of the sequences of points into four subsequences and the above invariants are strongly inspired by Sundar's construction [33] for a priority queue supporting INSERT and DELETEMIN in worst-case constant time, and where insertions have the side-effect of deleting all elements larger than the inserted element from the priority queue.

The convex chains A_1, A_2, and A_3 are stored as finger search trees so that updates at both ends can be supported in $O(1)$ time [5, 20, 34]. The set A_4 is stored using the deque convex hull structure of Theorem 5.
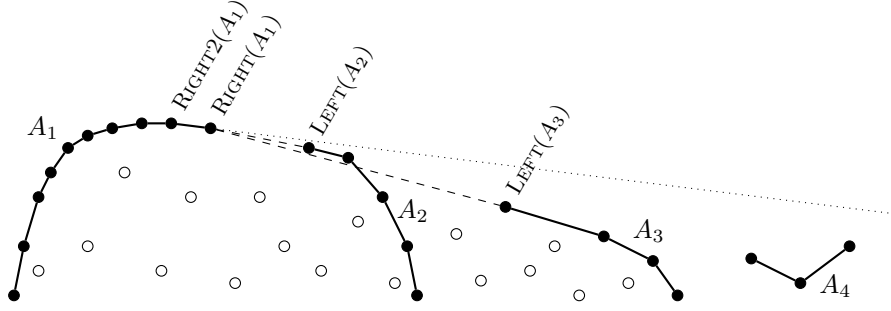
Figure 3 Stack convex hull. Black nodes are nodes in $A_i \subset P$, white nodes are $P \setminus (A_1 \cup A_2 \cup A_3 \cup A_4)$. There cannot be nodes above the dotted line through $\text{RIGHT2}(A_1)$ and $\text{RIGHT}(A_1)$, since $A_1 \subset \mathcal{H}(P)$. The dashed lines show that $A_1 \cup A_2$ and $A_1 \cup A_3$ are convex chains.

For $1 \le i \le 4$, we let $\text{LEFT}(A_i)$ and $\text{RIGHT}(A_i)$ denote the leftmost and rightmost points of A_i, respectively, and let $\text{LEFT2}(A_i)$ and $\text{RIGHT2}(A_i)$ denote the second leftmost and second rightmost point of A_i, respectively (see Figure 3). For any three points $p_1 \prec p_2 \prec p_3$, we use $p_1 \to p_2 \to p_3$ to refer to the traversal from p_1 to p_2 and then to p_3; we often need to determine whether $p_1 \to p_2 \to p_3$ makes a left turn or right turn.

2.4.2 InsertRight and DeleteRight

The basic idea is to perform insertions into A_4 (possibly after emptying A_1, A_3 and A_4, see Algorithm 1). This may cause the BIAS invariant to be violated since $|A_4|$ increases by one. To restore it, we will perform the BIAS procedure (Algorithm 2) twice. Running the BIAS procedure once will increase $|A_1| - |A_3| - 2 \cdot |A_4|$ by at least one, unless $A_3 = A_4 = \emptyset$ (in which case the BIAS invariant trivially holds). This is done by one of the following operations: (1) remove a point from A_3; (2) move a point from A_4 to A_3; (3) move a point from A_2 to A_1; (4) move a point from A_3 to A_1.

Algorithm 1 INSERTRIGHT

Input: A_1, A_2, A_3, A_4, and p, a point to be inserted
1 **if** $|A_1| \ge 2$ *and* $RIGHT2(A_1) \to RIGHT(A_1) \to p$ *makes a left turn* **then**
2 $\quad$ Set A_2 to refer to A_1
3 $\quad$ Set A_1, A_3, and A_4 to empty
4 INSERTRIGHT(A_4, p)
5 Run the BIAS procedure twice $\quad$ // Restore BIAS invariant

---

Note that the BIAS procedure preserves invariants 1, 2, and 3. To see this, notice that Lines 1-5 of Algorithm 2 are essentially performing steps of Graham's scan, adding the leftmost point of A_4 to the convex chain $A_1 \cup A_3$. When A_4 is empty, Lines 6-13 are trying to empty A_2 and A_3 by: (1) deleting $\text{LEFT}(A_3)$ if $\text{LEFT}(A_3) \notin \partial\mathcal{H}(P)$, identified by $\text{LEFT}(A_2) \to \text{LEFT}(A_3) \to \text{LEFT2}(A_3)$ making a left turn; (2) moving $\text{LEFT}(A_2)$ to A_1 if $\text{LEFT}(A_2) \in \partial\mathcal{H}(P)$ and $A_1 \cup \{\text{LEFT}(A_2)\} \cup A_3$ is a convex chain; (3) deleting all points of A_2 if $A_2 \cap \partial\mathcal{H}(P) = \emptyset$.

The BIAS procedure takes $O(1)$ time because the representations of each A_i allow insertion and deletion at both ends in $O(1)$ time. With the BIAS procedure in hand, we perform the INSERTRIGHT operation as shown in Algorithm 1. Notice that the check on Line 1 ensures

---

◼ **Algorithm 2** BIAS

Input: A_1, A_2, A_3, and A_4

1 **if** $|A_4| > 0$ **then**

2   **if** ($|A_3| \geq 2$ *and* $RIGHT2(A_3) \to RIGHT(A_3) \to LEFT(A_4)$ *is a left turn) or* ($|A_3| = 1$ *and* $|A_1| \geq 1$ *and* $RIGHT(A_1) \to RIGHT(A_3) \to LEFT(A_4)$ *is a left turn*) **then**

3          $\lfloor$ DELETERIGHT(A_3)

4   **else**

         `// Delete the leftmost point from ` A_4 ` and insert it to the right of ` A_3 `.`

5          $\lfloor$ INSERTRIGHT$(A_3,$ DELETELEFT$(A_4))$

6 **else if** $|A_3| > 0$ **then**

7   **if** $|A_2| > 0$ *and* $|A_3| \geq 2$ *and* $LEFT(A_2) \to LEFT(A_3) \to LEFT2(A_3)$ *is a left turn* **then**

8          $\lfloor$ DELETELEFT(A_3)

9   **else if** $|A_2| > 0$ *and* ($|A_1| = 0$ *or* $RIGHT(A_1) \to LEFT(A_2) \to LEFT(A_3)$ *is a right turn*) **then**

10          $\lfloor$ INSERTRIGHT$(A_1,$ DELETELEFT$(A_2))$

11   **else**

12   Set A_2 to empty

13   INSERTRIGHT$(A_1,$ DELETELEFT$(A_3))$

---

that Invariant 2 is preserved. If the condition is true, then the invariants are preserved trivially. If the condition is false, then inserting p to A_4 does not violate Invariant 2. In both cases, two calls to the BIAS procedure ensure that the BIAS invariant is preserved. Due to the way we implement each A_i, INSERTRIGHT can be done in $O(1)$ time.

To handle DELETERIGHT, we use a stack to record changes made during each insertion. To delete a point p, we use the stack to roll back the changes done during the insertion of p (these changes must be at the top of the stack). Since each INSERTRIGHT takes $O(1)$ time, there are $O(1)$ changes due to the insertion of p, and thus DELETERIGHT can be accomplished in $O(1)$ time as well.

2.4.3   HullTreeRetrieval

We wish to (implicitly) construct a BST of height $O(\log h)$ to represent the upper hull $\mathcal{H}(P)$. Recall that $h = |\mathcal{H}(P)|$. The second and fourth invariants together imply that $|A_1|$, $|A_3|$, and $|A_4|$ are all bounded by $O(h)$. Recall that for each A_i, $1 \leq i \leq 3$, A_i is a convex chain stored by a finger search tree, denoted by T_i. Since $|A_1|, |A_3| = O(h)$, the heights of both T_1 and T_3 are $O(\log h)$. For A_4, it is represented by a deque convex hull data structure. Since $|A_4| = O(h)$, by Theorem 5, we can obtain in $O(\log h)$ time a tree T_4 of height $O(\log h)$ to store $\mathcal{H}(A_4)$. For A_2, the only guaranteed upper bound for its size is $O(n)$. Hence, the height of T_2 is $O(\log n)$, instead of $O(\log h)$. By our algorithm invariants, $\mathcal{H}(P)$ is the upper hull of $A_1 \cup A_2 \cup A_3 \cup \mathcal{H}(A_4)$. As $A_1 \prec A_2 \prec A_3 \prec A_4$, we can compute $\mathcal{H}(P)$ by merging all four convex chains from left to right. This can be done in $O(\log n)$ time by using the trees T_i, $1 \leq i \leq 4$, and the binary search method of Overmars and van Leeuwen [30] for finding the upper tangents.

We now reduce the time to $O(\log h)$. The $O(\log n)$ time of the above algorithm is due to the fact that the height of T_2 is $O(\log n)$ instead of $O(\log h)$. This is because our algorithm invariants do not guarantee $|A_2| = O(h)$. Therefore, to have an $O(\log h)$-time algorithm, we need a clever way to merge A_2 with the other convex chains.

We first merge T_3 and T_4 to obtain in $O(\log h)$ time a new tree T_{34} of height $O(\log h)$ to represent the upper hull $\mathcal{H}(A_3 \cup A_4)$. Since A_1 is part of $\mathcal{H}(P)$ by our algorithm invariant, we have the following two cases for $\mathcal{H}(P)$: (1) no point of A_2 is on $\partial\mathcal{H}(P)$; (2) at least one point of A_2 is on $\partial\mathcal{H}(P)$. In the first case, $\mathcal{H}(P)$ can be obtained by simply merging T_1 and T_{34} in $O(\log h)$ time. In the second case, let $\overline{p_2 p_3}$ be the upper tangent between A_2 and $\mathcal{H}(A_3 \cup A_4)$ with $p_2 \in A_2$ and $p_3 \in \partial\mathcal{H}(A_3 \cup A_4)$. Then the vertices of $\mathcal{H}(P)$ from left to right are: all points of A_1, points of A_2 from $\text{Left}(A_2)$ to p_2, and points of $\mathcal{H}(A_3 \cup A_4)$ from p_3 to the right end. Hence, to compute $\mathcal{H}(P)$ in the second case, the key is to compute the tangent $\overline{p_2 p_3}$. We show below that this can be done in $O(\log h)$ time by an "exponential search" on A_2 and using the method of Overmars and van Leeuwen [30], in the following referred to as the OvL algorithm.

Note that which of the above two cases happens can be determined in $O(\log h)$ time. Indeed, we can first merge T_1 and T_{34} to obtain a tree T_{134} of height $O(\log h)$ representing the upper hull $\mathcal{H}(P \setminus A_2)$. Then, due to our algorithm invariants, the first case happens if and only if $\text{Left}(A_2)$ is below $\mathcal{H}(P \setminus A_2)$, which can be determined in $O(\log h)$ time using the tree T_{134}.

We now describe an algorithm to compute the common tangent $\overline{p_2 p_3}$ in the second case. Let $\mathcal{H}_{34} = \mathcal{H}(A_3 \cup A_4)$. Applying the OvL algorithm directly with T_2 and T_{34} will take $O(\log(|A_2| + |A_3 \cup A_4|)) = O(\log n)$ time to find $\overline{p_2 p_3}$. We next reduce the time to $O(\log h)$.

The OvL algorithm uses the binary search strategy. In each iteration, it picks two candidate points $p_2' \in A_2$ and $p_3' \in \mathcal{H}_{34}$ (initially, p_2' is the middle point of A_2 and p_3' is the middle point of $\mathcal{H}_{34}$), and in $O(1)$ time, the algorithm can determine at least one of the three cases: (1) $p_2 = p_2'$ and $p_3 = p_3'$; (2) p_2 is to the left or right of p_2'; (3) p_3 is to the left or right of p_3'. In the first case, the algorithm stops. In the second case, half of the remaining portion of A_2 is pruned and p_2' is reset to the middle point of the remaining portion of A_2 (but p_3' does not change). The third case is processed analogously.

When we apply the OvL algorithm, the way we set p_3' is the same as described above. However, for p_2', we set it by following the exponential search strategy using T_2 (or the standard finger search with a finger at the leftmost leaf of T_2 [5, 20, 34]). Specifically, we first reset p_2' to the leftmost node of T_2 and then continue the search on its parent and so on, until the first time we find p_2 is left of p_2', at which point we search downwards on T_2. This will eventually find $\overline{p_2 p_3}$.

We claim that the runtime is $O(\log h)$. Indeed, since the height of T_{34} is $O(\log h)$, the time the algorithm spends on resetting p_3' in the entire algorithm is $O(\log h)$. To analyze the time the algorithm spends on resetting p_2', following the standard finger search (with a finger at the leftmost leaf of the tree) or exponential search analysis, the number of times the algorithm resets p_2' is $O(\log m)$, where m is the number of points of T_2 to the left of p_2. Observe that all points of T_2 to the left of p_2 are on $\mathcal{H}(P)$; thus, we have $m \leq h$. Hence, the runtime of the algorithm for computing the tangent $\overline{p_2 p_3}$ is $O(\log h)$.

After $\overline{p_2 p_3}$ is computed, we split T_2 at p_2 and obtain a tree T_2' of height $O(\log h)$ representing all points of T_2 left of p_2. Then, we merge T_2' with T_1 and T_{34} to finally obtain a tree of height $O(\log h)$ representing $\mathcal{H}(P)$ in $O(\log h)$ time. The total time of the algorithm is $O(\log h)$.

As before, we use a stack to record changes due to the above algorithm for constructing the tree representing $\mathcal{H}(P)$. Once we finish the queries using the tree, we roll back the changes we have made by popping the stack. This also takes $O(\log h)$ time.

2.4.4 HullReport

To output the convex hull $\mathcal{H}(P)$, we first perform HULLTREERETRIEVAL to obtain a tree of height $O(\log h)$ that stores $\mathcal{H}(P)$ in $O(\log h)$ time. Then, using this tree, we can output $\mathcal{H}(P)$ in additional $O(h)$ time. Hence, the total time for reporting $\mathcal{H}(P)$ is $O(h)$.

3 The simple path problem

In this section, we consider the dynamic convex hull problem for a simple path. Let π be a simple path of n vertices in the plane. Unless otherwise stated, a "point" of π always refers to a vertex of it (this is for convenience and also to be consistent with the terminology in Section 2). For ease of discussion, we assume that no three vertices of π are collinear.

For any subpath π' of π, let $|\pi'|$ denote the number of vertices of π', and $\mathcal{H}(\pi')$ the convex hull of π', which is also the convex hull of all vertices of π'.

We designate the two ends of π as the *front end* and the *rear end*, respectively. We consider the following operations on π: INSERTFRONT, DELETEFRONT, INSERTREAR, DELETEREAR, STANDARDQUERY, and HULLREPORT, as defined in Section 1. The following theorem summarizes the main result of this section.

▶ **Theorem 7.** *Let $\pi \subset \mathbb{R}^2$ be an initially empty simple path, with $n = |\pi|$ and $h = |\mathcal{H}(\pi)|$. There exists a "Deque Path Convex Hull" data structure $PH(\pi)$ of $O(n)$ space that supports the following operations:*
1. *INSERTFRONT: $O(1)$ time.*
2. *DELETEFRONT: $O(1)$ time.*
3. *INSERTREAR: $O(1)$ time.*
4. *DELETEREAR: $O(1)$ time.*
5. *STANDARDQUERY: $O(\log n)$ time.*
6. *HULLREPORT: $O(h + \log n)$ time.*

Remark. We will show in Section 4 that all these bounds are optimal even for the "one-sided" case. In particular, it is not possible to reduce the time of HULLTREERETRIEVAL to $O(\log h)$ or reduce the time of HULLREPORT to $O(h)$. This is why we do not consider the one-sided simple path problem separately. For answering standard queries, our algorithm first constructs four BSTs representing convex hulls of four (consecutive) subpaths of π whose union is π and then uses these trees to answer queries. The heights of the two trees for the two middle subpaths are $O(\log n)$ while the heights of the other two are $O(\log \log n)$. This means that all decomposable queries can be answered in $O(\log n)$ time. We show that certain non-decomposable queries can also be answered in $O(\log n)$ time, such as bridge queries. We could further enhance the data structure so that a BST of height $O(\log h)$ that represents $\mathcal{H}(\pi)$ can be obtained in $O(\log n \log \log n)$ time; this essentially performs the HULLTREERETRIEVAL operation (as defined in Section 2.3) in $O(\log n \log \log n)$ time.

In what follows, we prove Theorem 7. One crucial property we rely on is that the convex hulls of two subpaths of a simple path intersect at most twice and thus have at most two common tangents as observed by Chazelle and Guibas [12]. Let π_1 and π_2 be two consecutive subpaths of π. Suppose we have two BSTs representing $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$, respectively. Compared to the monotone path problem, one difficulty here (we refer to it as the "path-challenge") is that we do not have an $O(\log n)$-time algorithm to find the common tangents between $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$ and thus merge the two hulls. The best algorithm we
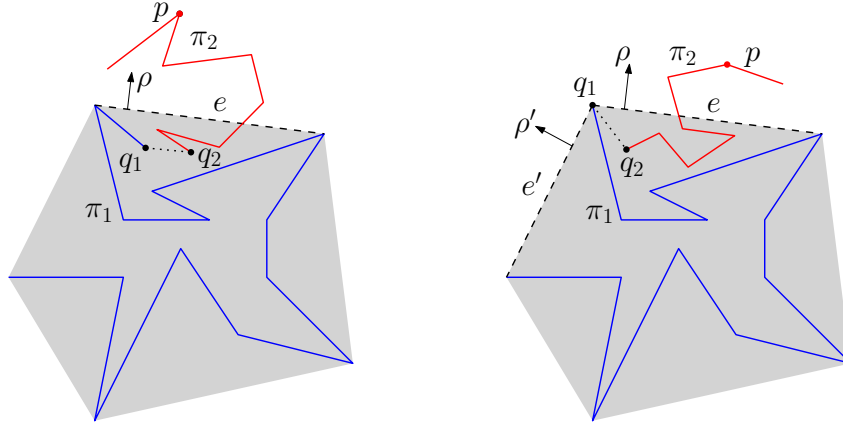
Figure 4 Illustrating the two cases in the proof of Lemma 8. The grey region in each case is $\mathcal{H}(\pi_1)$. ρ is the normal of e and ρ' is the normal of e'.

have takes $O(\log^2 n)$ time by a nested binary search, assuming that we have two "helper points": a point on each convex hull that is outside the other convex hull [19].

It is tempting to apply the deque convex hull idea of Theorem 5 (i.e., instead of considering the points in left-to-right order, we consider the points in the "path order" along π). We could get the same result as in Theorem 5 except that the HULLTREERETRIEVAL operation now takes $O(\log^2 n)$ time and HULLREPORT takes $O(h + \log^2 n)$ time due to the path-challenge. Our main effort below is to achieve $O(\log n)$ time for STANDARDQUERY and $O(h + \log n)$ time for HULLREPORT.

Before presenting our data structure, we introduce in Section 3.1 several basic lemmas that will be frequently used later.

3.1   Basic lemmas

The following lemma was given in [19], and we sketch the proof here to make the paper more self-contained. Later, we will need to modify the algorithm for other purposes.

▶ **Lemma 8.** (Guibas, Hershberger, and Snoeyink [19, Lemma 5.1])   *Let π_1 and π_2 be two consecutive subpaths of π. Suppose the convex hull $\mathcal{H}(\pi_i)$ is stored in a BST of height $O(\log |\pi_i|)$, for $i = 1, 2$. We can do the following in $O(\log(|\pi_1| + |\pi_2|))$ time: Determine whether $\mathcal{H}(\pi_2)$ is completely inside $\mathcal{H}(\pi_1)$ and if not find a "helper point" $p \in \partial\mathcal{H}(\pi_2)$ such that $p \in \partial\mathcal{H}(\pi_1 \cup \pi_2)$ and $p \notin \partial\mathcal{H}(\pi_1)$.*

Proof. We sketch the proof here. See [19, Lemma 5.1] for the details. Since π_1 and π_2 are consecutive subpaths of π, let $q_1 \in \pi_1$ and $q_2 \in \pi_2$ be consecutive vertices of π.

If q_1 is in the interior of $\mathcal{H}(\pi_1)$, then q_1 is in a single "bay" with a hull edge e such that: if $\mathcal{H}(\pi_2)$ is inside $\mathcal{H}(\pi_1)$, then π_2 must be inside the bay; otherwise, the path $\overline{q_1 q_2} \cup \pi_2$ must cross e (see the left of Figure 4). The edge e can be computed in $O(\log |\pi_1|)$ time by binary search since indices of vertices of $\mathcal{H}(\pi_1)$ form a bimodal sequence [19]. Let ρ be the normal of e toward outside $\mathcal{H}(\pi_1)$. We compute the most extreme vertex p of $\mathcal{H}(\pi_2)$ along ρ, which takes $O(\log |\pi_2|)$ time. Then, $\mathcal{H}(\pi_2)$ is inside $\mathcal{H}(\pi_1)$ if and only if p is in the same side of e as $\mathcal{H}(\pi_1)$. If p is on the opposite side of e as $\mathcal{H}(\pi_1)$, then p is a helper point as defined in the lemma statement.

If $q_1 \in \partial\mathcal{H}(\pi_1)$, then q_1 is a common vertex of two edges e and e' of H_1 (see the right of Figure 4), then we can determine whether $\mathcal{H}(\pi_2)$ is inside $\mathcal{H}(\pi_1)$ and if not find a helper

point on $\partial\mathcal{H}(\pi_2)$, using the normals of e and e', in a similar way to the above. The total time is still $O(\log(|\pi_1| + |\pi_2|))$. ◀

The following lemma is also from [19] and uses Lemma 8. We again sketch the proof.

▶ **Lemma 9.** (Guibas, Hershberger, and Snoeyink [19, Section 2]) *Let π_1 and π_2 be two consecutive subpaths of π. Suppose the convex hull $\mathcal{H}(\pi_i)$ is stored in a BST of height $O(\log |\pi_i|)$, $i = 1, 2$. We can compute a BST of height $O(\log(|\pi_1| + |\pi_2|))$ that stores the convex hull of $\pi_1 \cup \pi_2$ in $O(\log |\pi_1| \cdot \log |\pi_2|)$ time.*

Proof. We first use Lemma 8 to check whether one of the two convex hulls $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$ contains the other. If yes, then we simply return the tree of the larger convex hull. Otherwise, we compute a helper point $p_1 \in \partial\mathcal{H}(\pi_1)$ and a helper point $p_2 \in \partial\mathcal{H}(\pi_2)$ by Lemma 8. Using the two helper points, the two common tangents of $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$ can be computed in $O(\log |\pi_1| \cdot \log |\pi_2|)$ time by a nested binary search [19]. Consequently, a tree of height $O(\log(|\pi_1| + |\pi_2|))$ that stores $\mathcal{H}(\pi_1 \cup \pi_2)$ can be obtained in $O(\log |\pi_1| \cdot \log |\pi_2|)$ time from the two trees for $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$. ◀

The following lemma provides a basic tool for answering bridge queries.

▶ **Lemma 10.** *Let $H_1, H_2, \ldots,$ be a collection of $O(1)$ convex polygons with a total of n vertices, each represented by a BST or an array so that binary search on each convex polygon can be supported in $O(\log n)$ time. Let H be the convex hull of all these convex polygons. We can answer the following queries in $O(\log n)$ time each.*
1. *Bridge query: Given a query line ℓ, determine whether ℓ intersects H, and if yes, find the edges of H that intersect ℓ.*
2. *Given a query point p, determine whether $p \in H$, and if yes, further determine whether $p \in \partial H$.*

Proof. It is not difficult to see that the second query can be reduced to a bridge query (e.g., we can apply a bridge query on the vertical line ℓ through p, and the answer to the query on p can be obtained based on the bridge query outcome). In what follows, we focus on the bridge query. Without loss of generality, we assume that ℓ is vertical.

We first find the leftmost vertex p_1 of H, which can be done in $O(\log n)$ time by finding the leftmost vertex of each H_i. Similarly, we find the rightmost vertex p_2 of H. Then, ℓ intersects H if and only if ℓ is between p_1 and p_2. Assuming that ℓ is between them, we next compute the edges of H that intersect ℓ. For ease of discussion, we assume that ℓ does not contain any vertex of any convex hulls. This means that ℓ intersects exactly one edge of the upper hull of H and intersects exactly one edge of the lower hull. We only discuss how to find the edge on the upper hull, denoted by e^*, since the edge on the lower hull can be found similarly.

Observe that there are two cases for e^*: (1) The two vertices of e^* are from the same polygon H_k; (2) the two vertices of e^* are from two different polygons H_i and H_j, respectively. In the first case, e^* is the edge of the upper hull of H_k intersecting ℓ. In the second case, e^* is the edge of the upper hull of $\mathcal{H}(H_i \cup H_j)$ that intersects ℓ. Based on this observation, our algorithm works as follows.

To address the first case, for each H_k, we can compute the edge e_k that intersects ℓ (if it exists) in $O(\log n)$ time using a bridge query on H_k. To address the second case, for each pair (H_i, H_j), we will show how to compute the edge e_{ij} of the upper hull of $\mathcal{H}(H_i \cup H_j)$ intersecting ℓ (if it exists) in $O(\log n)$ time. Then, based on the above observation, among the edges e_k for all H_k and the edges e_{ij} for all pairs (H_i, H_j), e^* is the one whose intersection

with ℓ is the highest. This means that e^* can be found in $O(\log n)$ time as there are $O(1)$ convex hulls. It remains to show how to compute e_{ij} for any pair (H_i, H_j) in $O(\log n)$ time.

Let D_i be the data structure (an array or a BST) storing H_i. Let H_i^l and H_i^r denote the portions of H_i to the left and right of ℓ, respectively. Regardless of whether D_i is an array or a BST, binary search on each of H_i^l and H_i^r can be supported in $O(\log n)$ time by D_i. Define D_j, H_j^l, and H_j^r similarly.

If H_i^l and H_j^r are both non-empty, let e_{ij}^1 be the upper common tangent of H_i^l and H_j^r. If H_j^l and H_i^r are both non-empty, let e_{ij}^2 be the upper common tangent of H_j^l and H_i^r. If H_i intersects ℓ, let e_i be the edge on the upper hull of H_i intersecting ℓ. If H_j intersects ℓ, let e_j be the edge on the upper hull of H_j intersecting ℓ. Observe that e_{ij} exists if at least one of e_{ij}^1, e_{ij}^2, e_i, and e_j exists, in which case it is whichever one has the highest point of intersection with ℓ. Hence, to compute e_{ij}, it suffices to compute e_{ij}^1, e_{ij}^2, e_i, and e_j. Because H_i^l and H_j^r are separated by ℓ, e_{ij}^1 can be computed in $O(\log n)$ time with the data structures D_i and D_j using the binary search algorithm of Overmars and van Leeuwen [30]. The edge e_{ij}^2 can be computed similarly. Computing e_i and e_j takes $O(\log n)$ time by performing bridge queries on H_i and H_j. Finally, we can determine e_{ij} from e_{ij}^1, e_{ij}^2, e_i, and e_j in constant time by comparing their intersections with ℓ. As previously argued, because we can compute e_{ij} in $O(\log n)$ time for any pair (H_i, H_j), we can answer each bridge query in $O(\log n)$ time. ◄

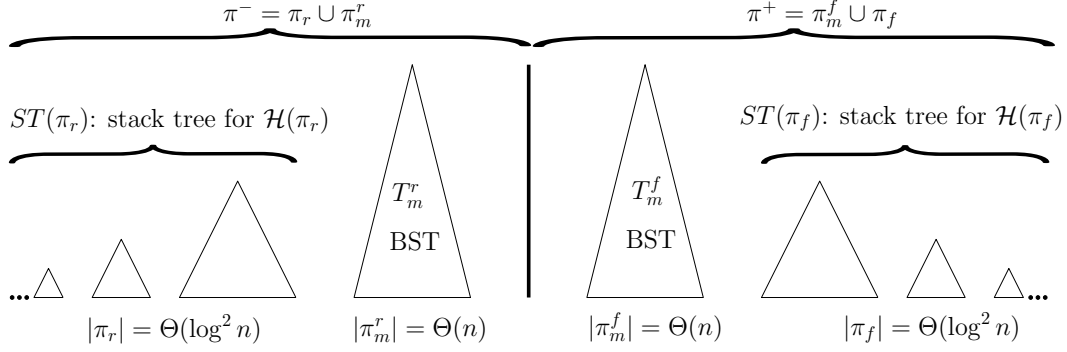## 3.2 Structure of the deque path convex hull $PH(\pi)$

For convenience, we assume that $|\pi|$ is always greater than a constant number (e.g., $|\pi| > 8$). We partition π into four (consecutive) subpaths π_r, π_m^r, π_m^f, and π_f in the order from the rear to the front of π so that π_r and π_f contain the rear and front ends, respectively. Further, let $\pi^+ = \pi_f \cup \pi_m^f$ and $\pi^- = \pi_r \cup \pi_m^r$. Our algorithm maintains the following invariants.

1. $\frac{1}{4} \leq |\pi^+|/|\pi^-| \leq 4$.
2. $|\pi_f| = O(\log^2 |\pi^+|)$ and $|\pi_r| = O(\log^2 |\pi^-|)$.

Note that the invariants imply that $|\pi_f|, |\pi_r| = O(\log^2 n)$ and $|\pi_m^f|, |\pi_m^r| = \Theta(n)$, where $n = |\pi|$. The first invariant resembles the partition of P by a dividing line ℓ in our deque tree in Section 2.2. As with the deque tree, in order to maintain the first invariant, when $1/2 \leq |\pi^+|/|\pi^-| \leq 2$ is first violated, we pick a new center to partition π and start to build a deque path convex hull data structure by incrementally inserting points around this new center toward both front and rear ends. As will be seen later, since each insertion takes $O(1)$ time, it takes $O(n)$ time to build this new data structure. Thus, we can spread the incremental work over the next $\Theta(n)$ updates so that only $O(1)$ incremental work on each update is incurred to maintain this invariant.

We use a stack tree $ST(\pi_f)$ to maintain the convex hull $\mathcal{H}(\pi_f)$, with the algorithm in Lemma 9 for merging two hulls of two consecutive subpaths. More specifically, we consider the vertices of π_f following their order along the path (instead of left-to-right order as in Section 2.1) with insertions and deletions only at the front end. Whenever we need to join two neighboring trees, we merge the two hulls of their corresponding subpaths by Lemma 9.[2]

---

[2] The analysis is slightly different since we use a merging algorithm of bigger complexity. More specifically, merging the hulls represented by the trees T_i and T_{i+1} now takes $O(\log^2 |T_i|)$ time, which is $O(2^{2i+2})$. Nevertheless, we can still guarantee that the current incremental joining process for T_{i+1} is completed before the next joining process starts by slightly modifying the proof of Lemma 3. More specifically, by making the constant c slightly larger, the joining process can be completed within 2^{3i-1} insertions. After 2^{3i-1} insertions, the total number of points that can be inserted into T_i is at most $2 \cdot 2^{2^i} + 2^{3i-1}$, which is still smaller than $2^{2^{i+1}}$ for $i \geq 1$.

Figure 5 A schematic view of the deque path convex hull data structure $PH(\pi)$.

Due to the second invariant, merging all trees of $ST(\pi_f)$ takes $O(\log^2 \log n)$ time, after which we obtain a single tree of height $O(\log \log n)$ that represents $\mathcal{H}(\pi_f)$. Similarly, we build a stack tree $ST(\pi_r)$ for $\mathcal{H}(\pi_r)$ but along the opposite direction of the path. See Figure 5 for an illustration.

Define $n^+ = |\pi^+|$. In order to maintain the second invariant, when π_f is too big due to insertions, we will split off a subpath of length $\Theta(\log^2 n^+)$ from π_f and concatenate it with π_m^f. When π_f becomes too small due to deletions, we will split off a portion of π_m^f of length $\Theta(\log^2 n^+)$ and merge it with π_f; but this split is done implicitly using the rollback stack for deletions. Consequently, we need to build a data structure for maintaining π_m^f so that the above concatenate operation on π_m can be performed in $O(\log^2 n^+)$ time (this is one reason why the bound for π_f in the second invariant is set to $O(\log^2 n^+)$). We process π^- in a symmetric way. The way we handle the interaction between π_m^f and π_f (as well as their counterpart for π^-) is one main difference from our approach for the two-sided monotone path problem in Section 2.3; again this is due to the path-challenge.

Our data structure for π_m^f is simply a balanced BST T_m^f, which stores the convex hull $\mathcal{H}(\pi_m^f)$. In particular, we will use T_m^f to support the above concatenation operation (denoted by CONCATENATE) in $O(\log^2 n)$ time. For reference purpose, this is summarized in the following lemma, which is an immediate application of Lemma 9.

▶ **Lemma 11.** *Given a BST of height $O(\log |\tau|)$ representing a simple path τ of length $O(\log^2 n)$ such that the concatenation of π_m^f and τ is still a simple path, we can perform the following* CONCATENATE *operation in $O(\log^2 n)$ time[3]: Obtain a new tree T_m^f of height $O(\log n)$ that represents the convex hull $\mathcal{H}(\pi_m^f)$, where π_m^f is the new path after concatenating with τ.*

Similarly, we use a balanced BST T_m^r to store the convex hull $\mathcal{H}(\pi_m^r)$. We have a similar lemma to the above for the CONCATENATE operation on π_m^r.

The four trees $ST(\pi_r)$, T_m^r, T_m^f, and $ST(\pi_f)$ constitute our deque path convex hull data structure $PH(\pi)$ for Theorem 7; see Figure 5. In the following, we discuss the operations.

---

[3] Applying Lemma 9 can obtain a time complexity of $O(\log n \log \log n)$, but $O(\log^2 n)$ is sufficient for our purpose.

3.3   Standard queries

For answering a decomposable query σ, we first perform a TREERETRIEVAL operation on $ST(\pi_f)$ to obtain a tree T_f that represents $\mathcal{H}(\pi_f)$. Since $|\pi_f| = O(\log^2 n)$, this takes $O(\log^2 \log n)$ time as discussed before. We do the same for $ST(\pi_r)$ to obtain a tree T_r for $\mathcal{H}(\pi_r)$. Recall that the tree T_m^f stores $\mathcal{H}(\pi_m^f)$ while T_m^r stores $\mathcal{H}(\pi_m^r)$. We perform query σ on each of the above four trees. Based on the answers to these trees, we can obtain the answer to the query σ for $\mathcal{H}(\pi)$ because σ is a decomposable query. Since the heights of T_f and T_r are both $O(\log \log n)$, and the heights of T_m^f and T_m^r are $O(\log n)$, the total query time is $O(\log n)$.

If σ is a bridge query, we apply Lemma 10 on the above four trees. The query time is $O(\log n)$.

3.4   Insertions and deletions

We now discuss the updates. INSERTFRONT and DELETEFRONT will be handled by the data structure for π^+, i.e., T_m^f and $ST(\pi_f)$, while INSERTREAR and DELETEREAR will be handled by the data structure for π^-.

InsertFront.   Suppose we insert a point p to the front end of π. We first perform the insertion using the stack tree $ST(\pi_f)$. To maintain the second invariant, we must handle the interaction between the largest tree T_k of $ST(\pi_f)$ and the tree T_m^f. Recall that $n^+ = |\pi^+|$ and $n^+ = \Theta(n)$.

According to the second invariant and the definition of the stack tree $ST(\pi_f)$, we have $|T_k| = O(\log^2 n^+)$, and we can assume a constant c such that the total size of all trees of $ST(\pi_f)$ smaller than T_k is at most $c \cdot \log^2 n^+$. We set the size of T_k to be $(c+1) \cdot \log^2 n^+$. During the algorithm, whenever $|T_k| > (c+1) \cdot \log^2 n^+$ and there is no incremental process of joining T_{k-1} with T_k, we let $T_k' = T_k$ and let $T_k = \emptyset$, and then start to perform an incremental CONCATENATE operation to concatenate T_k' with T_m^f. The operation takes $O(\log^2 n^+)$ time by Lemma 11. We choose a sufficiently large constant c_1 so that each CONCATENATE operation can be finished within $c_1 \cdot \log^2 n^+$ steps. For each INSERTFRONT in future, we run c_1 steps of this CONCATENATE algorithm. This means that within the next $\log^2 n^+$ INSERTFRONT operations in future, the CONCATENATE operation will be completed. If there is an incremental CONCATENATE operation (that is not completed), then we say that T_m^f is *dirty*; otherwise, it is *clean*.

If T_m^f is dirty, an issue arises during a STANDARDQUERY operation. Recall that during a STANDARDQUERY operation, we need to perform queries on $\mathcal{H}(\pi_m^f)$ by using the tree T_m^f. However, if T_m^f is dirty, we do not have complete information for T_m^f. To address this issue, we resort to persistent data structures [14, 32]. Specifically, we use a persistent tree for T_m^f so that if there is an incremental CONCATENATE operation, the old version of T_m^f can still be accessed (we call it the "clean version"). A partially persistent tree suffices for our purpose [14, 32]. After the CONCATENATE is completed, we designate the new version of T_m^f as clean and the old version as dirty; in this way, at any time, there is only one clean version we can refer to. During a STANDARDQUERY operation, we can perform queries on the clean version of T_m^f. Similarly, during the query, if there is an incremental CONCATENATE process, T_k' is also dirty, and we need to access its clean version (i.e., the version right before T_k' started the CONCATENATE operation). To solve this problem, before we start CONCATENATE, we make another copy of T_k', denoted by T_k''. After CONCATENATE is completed, we make T_k'' refer to null. The above strategy causes additional $O(\log^2 n^+)$ time,

i.e., update the persistent tree T_m^f and make a copy T_k''. To accommodate this additional cost, we make the constant c_1 large enough so that all these procedures can be completed within the next $\log^2 n^+$ INSERTFRONT operations.

Recall that once we are about to start a CONCATENATE operation for T_k', T_k becomes empty. We argue that CONCATENATE will be completed before another CONCATENATE operation starts. To this end, it suffices to argue that the current CONCATENATE will be completed before $|T_k|$ becomes larger than $(c+1) \cdot \log^2 n^+$ again. Indeed, we know that the current CONCATENATE will be finished within the next $\log^2 n^+$ INSERTFRONT operations. On the other hand, recall that the number of points in all trees of $ST(\pi_f)$ smaller than T_k is at most $c \cdot \log^2 n^+$. Since all points of T_k come from those smaller trees plus newly inserted points from the INSERTFRONT operations, within the next $\log^2 n^+$ INSERTFRONT operations, $|T_k|$ cannot be larger than $(c+1) \cdot \log^2 n$. Therefore, there cannot be two concurrent CONCATENATE operations from T_k to T_m^f.

DeleteFront. To perform a DELETEFRONT operation, i.e., delete the front vertex p of π, we keep a stack of changes to our data structure $PH(\pi)$ due to the INSERTFRONT operations. When deleting p, p must be the most recently inserted point at the front end, and thus, the changes to the data structure due to the insertion of p must be at the top of the stack. We roll back these changes by popping the stack.

InsertRear and DeleteRear. Handling updates at the rear end is the same, but using T_m^r and $ST(\pi_r)$ instead. We omit the details.

3.5 Reporting the convex hull $\mathcal{H}(\pi)$

We show that the convex hull $\mathcal{H}(\pi)$ can be reported in $O(h + \log n)$ time.

As in the algorithm for STANDARDQUERY, we first obtain in $O(\log n)$ time the four trees T_f, T_r, T_m^f, and T_m^r representing $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_r)$, $\mathcal{H}(\pi_m^f)$, and $\mathcal{H}(\pi_m^r)$, respectively. Then, we can merge these four convex hulls using Lemma 9 in $O(\log^2 n)$ time and compute a BST $T(\pi)$ representing $\mathcal{H}(\pi)$. Finally, we can output $\mathcal{H}(\pi)$ by traversing $T(\pi)$ in additional $O(h)$ time. As a result, $\mathcal{H}(\pi)$ can be reported in $O(h + \log^2 n)$ time. In what follows, we reduce the time to $O(h + \log n)$. To this end, we first enhance our data structure $PH(\pi)$ by having it maintain the common tangents of the convex hulls of the two middle subpaths π_m^f and π_m^r during updates.

3.5.1 Enhancing the data structure $PH(\pi)$

We now enhance our data structure $PH(\pi)$ described in Section 3.2. We have our enhanced $PH(\pi)$ maintain the common tangents between the convex hulls of the two middle subpaths of π, i.e., $\mathcal{H}(\pi_m^f)$ and $\mathcal{H}(\pi_m^r)$. Before describing how to maintain these common tangents, we first explain why they are useful. Let π_m denote the concatenation of π_m^f and π_m^r.

Suppose the common tangents of $\mathcal{H}(\pi_m^f)$ and $\mathcal{H}(\pi_m^r)$ are available to us. Then, we can compute in $O(\log n)$ time a tree T_m representing $\mathcal{H}(\pi_m)$ by merging $\mathcal{H}(\pi_m^f)$ and $\mathcal{H}(\pi_m^r)$ using their trees T_m^f and T_m^r. Consequently, we can do the following. (1) With the three trees T_f, T_m, and T_r, representing $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_m)$, and $\mathcal{H}(\pi_r)$, respectively, we can report $\mathcal{H}(\pi)$ in $O(h + \log n)$ time with an algorithm to be given later in Section 3.5.2. (2) By further merging $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_m)$, and $\mathcal{H}(\pi_r)$ using their trees, we can obtain a single tree of height

$O(\log n)$ representing $\mathcal{H}(\pi)$, which takes $O(\log n \log \log n)$ time by Lemma 9 since the heights of both T_f and T_r are $O(\log \log n)$ and the height of T_m is $O(\log n)$.

We proceed to describe how to maintain the common tangents of $\mathcal{H}(\pi_m^f)$ and $\mathcal{H}(\pi_m^r)$. To do so, we modify our algorithms for the update operations so that whenever π_m^f or π_m^r is changed (either due to the CONCATENATE operation or the rollback of the operation, referred to as an *inverse* CONCATENATE), we recompute the common tangents. By Lemma 9, their common tangents can be computed in $O(\log^2 n)$ time. The main idea is to incorporate the tangent-computing algorithm into the CONCATENATE operation. Once an incremental CONCATENATE operation is finished, say, on π_m^f, we immediately start computing the new common tangents. We consider this tangent-computing procedure part of the CONCATENATE operation. As each CONCATENATE operation takes $O(\log^2 n)$ time, incorporating the $O(\log^2 n)$-time tangent-computing algorithm as above does not change the time complexity of CONCATENATE asymptotically. However, we do have an issue with this idea. Recall that during DELETE-FRONT operations, we roll back the changes incurred by INSERTFRONT operations, and thus, each inverse CONCATENATE is done by the rollback. After an inverse CONCATENATE is completed, π_m^f loses a subpath of length $\Theta(\log^2 n^+)$, and we need to recompute the common tangents. However, in this case, rollback cannot recompute the common tangents; thus, we explicitly run the tangent-computing algorithm during rollback. This means we must find an effective way to incorporate this additional step into the rollback process so that each DELETEFRONT operation still takes $O(1)$ worst-case time. For reference purpose, we call this the *rollback issue*.

We now elaborate on how to modify our update algorithms. We only discuss INSERTFRONT and DELETEFRONT since the other two updates are similar.

InsertFront. We follow notation in Section 3.4, e.g., T_m^f, $ST(\pi_f)$, n^+, T_k, T_k', T_{k-1}, c, c_1, etc. During the algorithm, as before, whenever $|T_k| > (c+1) \cdot \log^2 n^+$ and there is no incremental process of joining T_{k-1} with T_k, we let $T_k' = T_k$, $T_k = \emptyset$, and start an incremental CONCATENATE to concatenate T_k' with T_m^f. Recall that the incremental CONCATENATE in Section 3.4 will be completed within the next $\log^2 n^+$ INSERTFRONT operations. Here, to complete CONCATENATE, we do the following three steps in the next $\log^2 n^+$ INSERTFRONT operations (note that these are "net" INSERTFRONT operations, i.e., the number of INSERT-FRONT operations minus the number of DELETEFRONT operations happened in the future): (1) For each of the next $\frac{1}{3} \cdot \log^2 n^+$ INSERTFRONT operations, we push a "token" into our rollback stack, where a token could be a special symbol. Intuitively, a token represents a time credit we can use during the rollback to perform an incremental tangent-computing procedure. This is our mechanism to address the aforementioned rollback issue. (2) In each of the subsequent $\frac{1}{3} \cdot \log^2 n^+$ INSERTFRONT operations, we perform the next $3c_1$ steps of the actual CONCATENATE operation by Lemma 11 (including the extra step of making a copy T_k'' from T_k' first as discussed in Section 3.4). (3) Start an incremental tangent-computing algorithm to find the common tangents between this new $\mathcal{H}(\pi_m^f)$ (using the new T_m^f after the current CONCATENATE is completed in the above step (2)) and $\mathcal{H}(\pi_m^r)$ (using the clean version of its tree T_m^r), and in each of the subsequent $\frac{1}{3} \cdot \log^2 n^+$ INSERTFRONT operations, perform the next c_2 steps of the tangent-computing algorithm, for a sufficiently large constant c_2 so that the algorithm will be finished within $\frac{c_2}{3} \cdot \log^2 n^+$ steps (such a constant c_2 exists because the tangent-computing algorithm runs in $O(\log^2 n)$ time and $n^+ = \Theta(n)$). Once all three steps are completed, we consider the current incremental CONCATENATE completed and set T_k'' to null (meaning that the subpath stored in T_k' now officially becomes part of π_m^f). However, we mark the current version of T_m^f clean after step (2) is completed. Since it still takes $\log^2 n^+$

INSERTFRONT operations to complete the new incremental CONCATENATE operation, the prior argument in Section 3.4 that the current CONCATENATE will be completed before the next CONCATENATE starts still applies here.

DeleteFront. We perform the rollback process for each DELETEFRONT operation as before but with the following changes. Whenever an "inverse" CONCATENATE is completed (i.e., the above step (2)), then according to our new incremental CONCATENATE, the rollback stack contains $\frac{1}{3} \cdot \log^2 n^+$ tokens which will be popped by the next $\frac{1}{3} \cdot \log^2 n^+$ to-be-deleted points (and the history of changes of the next inverse CONCATENATE is stored below all these tokens). When we encounter the first of these tokens, we mark the current T_m^f clean and start a tangent-computing algorithm to find the common tangents between this new $\mathcal{H}(\pi_m^r)$ and the current clean version of T_m^r for $\mathcal{H}(\pi_m^r)$. Then, for each of the subsequent $\frac{1}{3} \cdot \log^2 n^+$ tokens popped out of the rollback stack, we perform the next c_2 steps of the tangent-computing algorithm (c_2 was already defined above in such a way that the algorithm will be finished within $\frac{c_2}{3} \cdot \log^2 n^+$ steps). We consider the inverse CONCATENATE operation completed once the tangent-computing algorithm is finished.

Obtaining the common tangents. We argue that the common tangents of $\mathcal{H}(\pi_m^f)$ and $\mathcal{H}(\pi_m^r)$ can be accessed at any time. Consider the time t after an update operation. Each of T_m^f and T_m^r has a clean version at t. Suppose the current clean version T_m^f (resp., T_m^r) became clean at time t_1 (resp., t_2). Without loss of generality, we assume $t_1 \geq t_2$, i.e., T_m^f became clean later than T_m^r did. Hence, once T_m^f became clean at t_1, a tangent-computing algorithm started to compute common tangents between the clean versions of T_m^f and T_m^r.

At the time t, each of T_m^f and T_m^r may or may not have an incomplete incremental CONCATENATE operation or an incomplete inverse CONCATENATE operation. If T_m^f has an incomplete operation, we let $\hat{T}_m^f$ denote its version before the operation started. Define $\hat{T}_m^r$ similarly. We use T_m^f and T_m^r to refer to their current clean versions at t. Depending on whether T_m^f and/or T_m^r has an incomplete operation, there are four cases.

1. If neither T_m^f nor T_m^r has an incomplete operation at t, then the tangent-computing algorithm for T_m^f and T_m^r has been finished (otherwise the operation would be incomplete at t). Thus, their common tangents are available at t.

2. If T_m^f has an incomplete operation while T_m^r does not, then the common tangents between T_m^r and $\hat{T}_m^f$ must be available. Indeed, as above, the one of T_m^r and $\hat{T}_m^f$ that became clean later must start a tangent-computing algorithm to compute their common tangents and the algorithm must have finished before t since otherwise at least one of them must have an incomplete operation at t, which is not true.
   Since T_m^f still has an incomplete operation, $\hat{T}_m^f$ actually represents the current subpath π_m^f. More specifically, if the incomplete operation is CONCATENATE, then it is concatenating a new subpath τ to π_m^f. As the operation is incomplete, we still consider τ part of π_f (instead of part of π_m^f), and τ is stored at T_k''. If the incomplete operation is an inverse CONCATENATE, then it is removing a subpath τ from π_m^f. As the operation is incomplete, we still consider τ as part of π_f, which is stored in $\hat{T}_m^f$. Therefore, the common tangents of T_m^r and $\hat{T}_m^f$ are what we need.

3. If T_m^r has an incomplete operation while T_m^f does not, this case is symmetric to the above second case and can be treated likewise.

4. If both T_m^f and T_m^r have incomplete operations, then by a similar argument to the above, the common tangents between $\hat{T}_m^r$ and $\hat{T}_m^f$ must be available, and their common tangents are what we need.

3.5.2   Algorithm for HullReport

Suppose we have the three trees T_f, T_m, and T_r, representing $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_m)$, and $\mathcal{H}(\pi_r)$, as discussed above. We now describe our algorithm for reporting $\mathcal{H}(\pi)$ using the three trees. Depending on whether one of the three convex hulls $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_r)$, and $\mathcal{H}(\pi_m)$ contains another or both of the other two, there are a number of cases.

1. If $\mathcal{H}(\pi_m)$ contains both $\mathcal{H}(\pi_f)$ and $\mathcal{H}(\pi_r)$, then $\mathcal{H}(\pi_m)$ is $\mathcal{H}(\pi)$ and thus we can simply report $\mathcal{H}(\pi_m)$, which can be done in $O(h)$ time using the tree T_m. We can apply Lemma 8 to determine this case. More specifically, since π_m and π_f are consecutive subpaths of π, applying Lemma 8 with T_m and T_f can determine whether $\mathcal{H}(\pi_m)$ contains $\mathcal{H}(\pi_f)$ in $O(\log n)$ time. Similarly, whether $\mathcal{H}(\pi_m)$ contains $\mathcal{H}(\pi_r)$ can be determined in $O(\log n)$ time. Therefore, we can report $\mathcal{H}(\pi)$ in $O(h + \log n)$ time in this case.

2. If $\mathcal{H}(\pi_m)$ contains $\mathcal{H}(\pi_r)$ but not $\mathcal{H}(\pi_f)$, and $\mathcal{H}(\pi_f)$ does not contain $\mathcal{H}(\pi_m)$, then we will present an algorithm later to report $\mathcal{H}(\pi)$ in $O(h + \log n)$ time. As in the first case, we can determine whether this case happens in $O(\log n)$ time using Lemma 8.

3. If $\mathcal{H}(\pi_m)$ contains $\mathcal{H}(\pi_f)$ but not $\mathcal{H}(\pi_r)$, and $\mathcal{H}(\pi_r)$ does not contain $\mathcal{H}(\pi_m)$, then this is a case symmetric to the above second case and thus can be treated likewise.

4. If $\mathcal{H}(\pi_r)$ contains both $\mathcal{H}(\pi_m)$ and $\mathcal{H}(\pi_f)$, then $\mathcal{H}(\pi_r)$ is $\mathcal{H}(\pi)$ and thus we can simply report $\mathcal{H}(\pi_r)$, which can be done in $O(h)$ time using the tree T_r. We can determine whether this case happens in $O(\log n)$ time using Lemma 8. Indeed, we first apply Lemma 8 on $\mathcal{H}(\pi_r)$ and $\mathcal{H}(\pi_m)$ to determine whether $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m)$ in $O(\log n)$. If yes, we must determine whether $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_f)$. Note that this time we cannot apply Lemma 8 directly as π_r and π_f are not consecutive. To overcome this issue, since we already know that $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m)$, it suffices to know whether $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m \cup \pi_f)$. For this, we can slightly modify the algorithm in Lemma 8 since π_r and $\pi_m \cup \pi_f$ are two consecutive subpaths of π. Specifically, the algorithm needs to solve a subproblem that is to compute the most extreme point p of $\mathcal{H}(\pi_m \cup \pi_f)$ along a direction ρ that is the normal of an edge of $\mathcal{H}(\pi_r)$. We cannot solve this subproblem by applying the algorithm in the proof of Lemma 8 since we do not have a tree to represent $\mathcal{H}(\pi_m \cup \pi_f)$. Instead, we can compute p by first computing the most extreme point of $\mathcal{H}(\pi_m)$ along ρ in $O(\log n)$ time using the tree T_m and computing the most extreme point of $\mathcal{H}(\pi_r)$ along ρ in $O(\log \log n)$ time using the tree T_r, and then return the more extreme point along ρ among the two points. Therefore, we can determine whether $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m \cup \pi_f)$ in $O(\log n)$ time.

5. If $\mathcal{H}(\pi_f)$ contains both $\mathcal{H}(\pi_m)$ and $\mathcal{H}(\pi_r)$, this is a case symmetric to the above fourth case and thus can be treated likewise.

6. The remaining case: $\mathcal{H}(\pi_m)$ contains neither $\mathcal{H}(\pi_f)$ nor $\mathcal{H}(\pi_r)$, and neither $\mathcal{H}(\pi_f)$ nor $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m)$. We will present an $O(h + \log n)$ time algorithm. This is the most general case, and the algorithm is also the most complicated. Note that it is possible that one of $\mathcal{H}(\pi_f)$ and $\mathcal{H}(\pi_r)$ contains the other; however, we cannot apply Lemma 8 to determine that since π_f and π_r are not consecutive subpaths of π.

It remains to present algorithms for the above Case 2 and Case 6.

Algorithm for Case 2.   In Case 2, $\mathcal{H}(\pi_m)$ contains $\mathcal{H}(\pi_r)$ but not $\mathcal{H}(\pi_f)$, and $\mathcal{H}(\pi_f)$ does not contain $\mathcal{H}(\pi_m)$. One may consider the algorithm for this case a "warm-up" for the more complicated algorithm of Case 6 given later. In this case, $\mathcal{H}(\pi)$ is the convex hull of $\mathcal{H}(\pi_f)$ and $\mathcal{H}(\pi_m)$, and thus it suffices to merge $\mathcal{H}(\pi_f)$ and $\mathcal{H}(\pi_m)$ by computing their common tangents. Note that computing the common tangents can easily be done in $O(\log n \log \log n)$
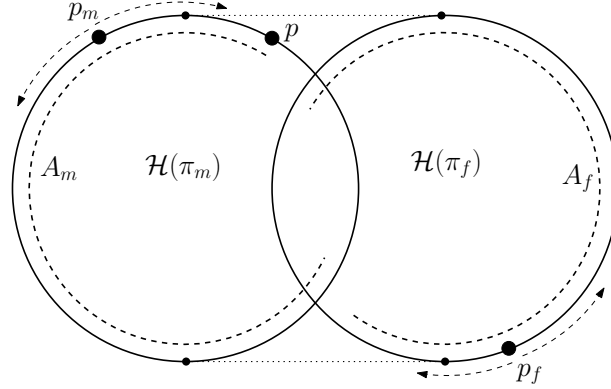
Figure 6 Illustrating the walking procedure for Case 2. The dashed arcs illustrate A_m and A_f on $\partial\mathcal{H}(\pi_m)$ and $\partial\mathcal{H}(\pi_f)$, respectively

time by a nested binary search [19] since the height of T_m is $O(\log n)$ and the height of T_f is $O(\log \log n)$. After the common tangents are computed, $\mathcal{H}(\pi)$ can be output in additional $O(h)$ time. The total time of this algorithm is $O(h + \log n \log \log n)$. In what follows, we present an improved algorithm achieving $O(h + \log n)$ time.

First, we apply Lemma 8 to compute a helper point $p_m \in \partial\mathcal{H}(\pi_m)$ such that $p_m \in \partial\mathcal{H}(\pi)$ and $p_m \notin \mathcal{H}(\pi_f)$, and also compute a helper point $p_f \in \partial\mathcal{H}(\pi_f)$ such that $p_f \in \partial\mathcal{H}(\pi)$ and $p_f \notin \mathcal{H}(\pi_m)$. This takes $O(\log n)$ time using the trees T_m and T_f by Lemma 8.

Next, starting from p_m, we perform the following *walking procedure*. We walk clockwise on $\partial\mathcal{H}(\pi_m)$ with a "step size" $\log n$ (see Figure 6). The idea is to identify a portion of $\partial\mathcal{H}(\pi_m)$ of size at most $\log n$ that contains a common tangent point. To this end, using the tree T_m, we find in $O(\log n)$ time a point p on $\partial\mathcal{H}(\pi_m)$ such that the portion $\partial\mathcal{H}(\pi_m)[p_m, p]$ from p_m clockwise to p along $\partial\mathcal{H}(\pi_m)$ contains $\log n$ vertices. Then, we determine whether $p \in \partial\mathcal{H}(\pi)$. Since $\mathcal{H}(\pi)$ is the convex hull of $\mathcal{H}(\pi_m)$ and $\mathcal{H}(\pi_f)$, we can apply Lemma 10 to determine whether $p \in \partial\mathcal{H}(\pi)$ in $O(\log n)$ time.

- If $p \notin \partial\mathcal{H}(\pi)$, then $\partial\mathcal{H}(\pi_m)[p_m, p]$ must contain a tangent point. In this case, starting from p_m, we repeat the same walking procedure counterclockwise until we find another boundary portion of $\mathcal{H}(\pi)$ that contains a tangent point. Let A_m denote the set of vertices that have been traversed during the above clockwise and counterclockwise walks (see Figure 6; note that all vertices in each step, e.g., all vertices in $\partial\mathcal{H}(\pi_m)[p_m, p]$, are included in A_m). Observe that A_m has the following *key property*: Vertices of $\mathcal{H}(\pi)$ on $\partial\mathcal{H}(\pi_m)$ are all in A_m and A_m contains at most $2\log n$ vertices that are not on $\partial\mathcal{H}(\pi)$. This means that $|A_m| = O(h + \log n)$.
- If $p \in \partial\mathcal{H}(\pi)$, then we keep walking clockwise as above until we make a step where $p \notin \partial\mathcal{H}(\pi)$ or we pass over the starting point p_m again. In the first case, we find a portion of $\log n$ vertices on $\partial\mathcal{H}(\pi)$ that contain a tangent point. Then, we repeat the same walk procedure counterclockwise around $\mathcal{H}(\pi)$. We can still obtain a set A_m of vertices of $\partial\mathcal{H}(\pi)$ with the above key property. In the second case, we realize that one of the walking steps contains two common tangent points. In this case, the observation is that all but at most $\log n$ vertices of $\mathcal{H}(\pi_m)$ are also vertices of $\mathcal{H}(\pi)$; we define A_m as the set of all vertices of $\mathcal{H}(\pi_m)$, and thus the key property still holds on A_m.

In either case, we have found a subset A_m of vertices of $\mathcal{H}(\pi_m)$ with the above key property. Since $|A_m| = O(h + \log n)$, the walking procedure computes A_m in $O(h + \log n)$
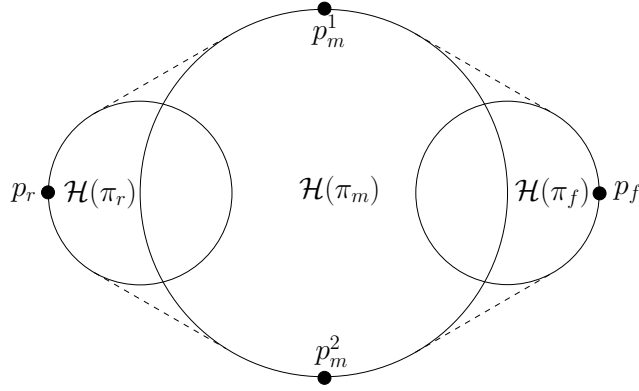
Figure 7 Illustrating the convex hull $\mathcal{H}(\pi)$ in Case 6.

time because each step takes $O(\log n)$ time and each step either completes the traversal of A_m or traverses $\log n$ points of A_m (therefore the number of steps is $O(1 + h/\log n)$). Also, since all vertices of A_m are on $\partial\mathcal{H}(\pi_m)$, we can sort them from left to right in linear time. Using the same strategy, starting from p_f on $\mathcal{H}(\pi_f)$, we can also find a subset A_f of sorted vertices of $\mathcal{H}(\pi_f)$ with a similar key property: Vertices of $\mathcal{H}(\pi)$ on $\partial\mathcal{H}(\pi_f)$ are all in A_f and A_f contains at most $2\log n$ vertices that are not on $\partial\mathcal{H}(\pi)$ (see Figure 6). Notice that the convex hull of $A_m \cup A_f$ is exactly $\mathcal{H}(\pi)$. As each of A_m and A_f is already sorted, their convex hull can be computed in $O(|A_m| + |A_f|)$ time, which is $O(h + \log n)$ since both $|A_m|$ and $|A_f|$ are bounded by $O(h + \log n)$.

In summary, we can report $\mathcal{H}(\pi)$ in $O(h + \log n)$ time in Case 2.

Algorithm for Case 6. We now describe the algorithm for Case 6, in which $\mathcal{H}(\pi_m)$ contains neither $\mathcal{H}(\pi_f)$ nor $\mathcal{H}(\pi_r)$, and neither $\mathcal{H}(\pi_f)$ nor $\mathcal{H}(\pi_r)$ contains $\mathcal{H}(\pi_m)$. The algorithm, which extends our walking algorithm for Case 2, becomes more involved.

First, observe that $\mathcal{H}(\pi_r)$ can have at most one maximal boundary portion appearing on the boundary of $\mathcal{H}(\pi)$. To see this, notice that $\mathcal{H}(\pi)$ is the convex hull of $\mathcal{H}(\pi_r)$ and $\mathcal{H}(\pi_m \cup \pi_f)$. As π_r and $\pi_m \cup \pi_f$ are two consecutive subpaths of π, there are at most two common tangents between $\mathcal{H}(\pi_r)$ and $\mathcal{H}(\pi_m \cup \pi_f)$, implying that $\mathcal{H}(\pi_r)$ has at most one maximal boundary portion appearing on $\partial\mathcal{H}(\pi)$. Similarly, $\mathcal{H}(\pi_f)$ has at most one maximal boundary portion appearing on $\partial\mathcal{H}(\pi)$.

However, $\mathcal{H}(\pi_m)$ can have at most two maximal boundary portions appearing on $\partial\mathcal{H}(\pi)$. To see this, first of all, as argued above, $\mathcal{H}(\pi_m)$ has at most one maximal boundary portion appearing on $\partial\mathcal{H}(\pi_m \cup \pi_r)$. Note that $\mathcal{H}(\pi)$ is the convex hull of $\mathcal{H}(\pi_m \cup \pi_r)$ and $\mathcal{H}(\pi_f)$. Since $\mathcal{H}(\pi_f)$ has at most one maximal boundary portion appearing on $\partial\mathcal{H}(\pi)$, we obtain that $\mathcal{H}(\pi_m)$ has at most two maximal boundary portions appearing on $\partial\mathcal{H}(\pi)$.

Consequently, the boundary of $\mathcal{H}(\pi)$ has at most four maximal portions, each of which belongs to one of the three convex hulls $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_m)$, or $\mathcal{H}(\pi_f)$, and these maximal portions are connected by at most four edges of $\mathcal{H}(\pi)$ each of which is a common tangent of two hulls of S (see Figure 7). Suppose we know four vertices (p_r, p_m^1, p_m^2, p_f), one from each of these four boundary portions of $\mathcal{H}(\pi)$. Then, we can construct $\mathcal{H}(\pi)$ in $O(h + \log n)$ time by an algorithm using a walking procedure similar to that in Case 2, as follows.

Let $p_r \in \partial\mathcal{H}(\pi_r) \cap \partial\mathcal{H}(\pi)$. Starting from p_r, we perform a walking procedure by making reference to $\mathcal{H}(\pi)$. More specifically, we walk clockwise on $\partial\mathcal{H}(\pi_r)$ with a step size $\log n$. Let p_r' be the point on $\partial\mathcal{H}(\pi_r)$ such that there are $\log n$ vertices on $\partial\mathcal{H}(\pi_r)$ clockwise from

p_r to p_r'. We determine whether p_r' is on $\partial\mathcal{H}(\pi)$, which can be done in $O(\log n)$ time using the three trees T_m, T_f, and T_r by Lemma 10. If yes, then we continue the walk until after a step in which we either pass over p_r or find that p_r' is not on $\partial\mathcal{H}(\pi)$. In the former case, we let A_r be the set of all vertices of $\mathcal{H}(\pi_r)$. In the latter case, we make a counterclockwise walk on $\partial\mathcal{H}(\pi_r)$ starting from p_r again and let A_r be the subset of vertices that have been traversed before we stop the walking procedure.

We compute A_m^1, A_m^2, and A_f similarly. Also, each of A_r, A_m^1, A_m^2, and A_f can be sorted since all vertices of each lie on the boundary of a single convex hull. As in the algorithm for Case 2, we have the following *key property*: All vertices of $\mathcal{H}(\pi)$ are in $A = A_r \cup A_m^1 \cup A_m^2 \cup A_r$, and A contains at most $O(\log n)$ points that are not on $\partial\mathcal{H}(\pi)$. Therefore, $|A| = O(h + \log n)$ holds. Hence, $\mathcal{H}(\pi)$ is the convex hull of A and can be computed in $O(h + \log n)$ time since A can be sorted in $O(|A|)$ time and $|A| = O(h + \log n)$.

In light of the above algorithm, to report $\mathcal{H}(\pi)$, it suffices to find the (at most) four points p_r, p_m^1, p_m^2, and p_f. In particular, we will find a point p_r on the maximal portion of $\partial\mathcal{H}(\pi_r)$ (if it exists) appearing in $\partial\mathcal{H}(\pi)$, a point p_f on the maximal portion of $\partial\mathcal{H}(\pi_f)$ (if it exists) appearing in $\partial\mathcal{H}(\pi)$, and points p_m^1 and p_m^2 on the two maximal portions of $\partial\mathcal{H}(\pi_m)$ (if they exist) appearing in $\partial\mathcal{H}(\pi)$; in particular, if $\partial\mathcal{H}(\pi_m)$ only has one maximal portion in $\partial\mathcal{H}(\pi)$, then we will only compute p_m^1 and we say that p_m^2 does not exist.

We start with finding a helper point $\tilde{p}_r \in \partial\mathcal{H}(\pi_r)$ with $\tilde{p}_r \in \partial\mathcal{H}(\pi_r \cup \pi_m)$ using Lemma 8. We also find a helper point $\tilde{p}_f \in \partial\mathcal{H}(\pi_f)$ with $\tilde{p}_f \in \partial\mathcal{H}(\pi_m \cup \pi_f)$. Depending on whether $\tilde{p}_r \in \partial\mathcal{H}(\pi)$ and $\tilde{p}_f \in \partial\mathcal{H}(\pi)$ (determined using Lemma 10), there are four cases.

6a If $\tilde{p}_r \in \partial\mathcal{H}(\pi)$ and $\tilde{p}_f \in \partial\mathcal{H}(\pi)$. In this case, we have found $p_r = \tilde{p}_r$ and $p_f = \tilde{p}_f$, and it remains to compute p_m^1 and p_m^2.

6b If $\tilde{p}_r \in \partial\mathcal{H}(\pi)$ and $\tilde{p}_f \notin \partial\mathcal{H}(\pi)$. In this case, we claim that $\partial\mathcal{H}(\pi_m)$ has at most one maximal portion appearing on $\partial\mathcal{H}(\pi)$. Indeed, assume to the contrary that $\partial\mathcal{H}(\pi_m)$ has two maximal portions appearing on $\partial\mathcal{H}(\pi)$; see Figure 7. Then, the two portions of $\partial\mathcal{H}(\pi)$ from $\partial\mathcal{H}(\pi_m)$ are separated by the portion from $\partial\mathcal{H}(\pi_f)$ and the portion from $\partial\mathcal{H}(\pi_r)$. This means that every point of $\partial\mathcal{H}(\pi_f)$ on $\partial\mathcal{H}(\pi_m \cup \pi_f)$ must be on $\partial\mathcal{H}(\pi)$. As $\tilde{p}_f \in \partial\mathcal{H}(\pi_f)$ and $\tilde{p}_f \in \partial\mathcal{H}(\pi_m \cup \pi_f)$ by definition, we obtain $\tilde{p}_f \in \partial\mathcal{H}(\pi)$, a contradiction. Since $\partial\mathcal{H}(\pi_m)$ has at most one maximal portion appearing on $\partial\mathcal{H}(\pi)$, p_m^2 does not exist. As $\tilde{p}_r \in \partial\mathcal{H}(\pi)$, we have found $p_r = \tilde{p}_r$. Therefore, it remains to compute p_m^1 and p_f.

6c If $\tilde{p}_r \notin \partial\mathcal{H}(\pi)$ and $\tilde{p}_f \in \partial\mathcal{H}(\pi)$. This case is symmetric to Case 6b and can be treated likewise.

6d If $\tilde{p}_r \notin \partial\mathcal{H}(\pi)$ and $\tilde{p}_f \notin \partial\mathcal{H}(\pi)$. We can reduce this case to Case 6a as follows. When computing $\tilde{p}_r$ by Lemma 8, $\tilde{p}_r$ was found to be the most extreme point of $\mathcal{H}(\pi_r \cup \pi_m)$ along some direction ρ. Let p_f' be the most extreme point of $\mathcal{H}(\pi_f)$ in the direction ρ. As $\tilde{p}_r \notin \partial\mathcal{H}(\pi)$ and $\tilde{p}_r$ is the most extreme point along ρ in $\mathcal{H}(\pi_r \cup \pi_m)$, p_f' must be more extreme along ρ than $\tilde{p}_r$, and we can conclude that $p_f' \in \partial\mathcal{H}(\pi)$. Using the same strategy with respect to $\tilde{p}_f$, we can find a point $p_r' \in \partial\mathcal{H}(\pi_r)$ with $p_r' \in \partial\mathcal{H}(\pi)$. With p_r' and p_f' playing the roles of $\tilde{p}_r$ and $\tilde{p}_f$, respectively, we are in Case 6a.

We now discuss our algorithm for Case 6a, where we know p_r and p_f, and it remains to find p_m^1 and p_m^2. Using the walking procedure, we can compute A_r and A_f. Next, we compute $H = \mathcal{H}(A_r \cup A_f)$ in $O(h + \log n)$ time since A_r and A_f can be sorted in $O(h + \log n)$ time, as described previously. Note that H contains two *special edges*: Each of them connects a vertex of A_r and a vertex of A_f, i.e., these are common tangents between the convex hulls $\mathcal{H}(A_r)$ and $\mathcal{H}(A_f)$; let e_1 and e_2 denote these two edges, respectively. Let ρ_1 be the direction normal to e_1 toward the outside of H. Let ρ_2 be defined similarly for e_2. Notice
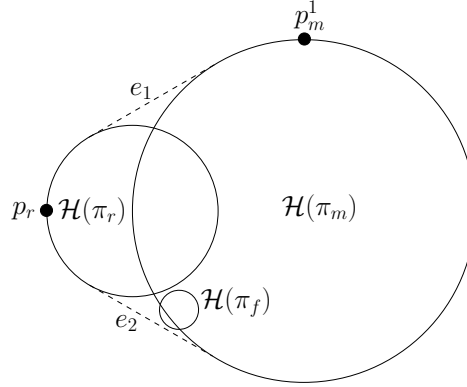
Figure 8 Illustrating an example in which π_f does not contribute to $\partial\mathcal{H}(\pi)$, i.e., $\pi_f \subset \mathcal{H}(\pi_r \cup \pi_m)$.

that we can find p_m^1 and p_m^2, if they exist, by finding the most extreme points of $\mathcal{H}(\pi_m)$ in the directions ρ_1 and ρ_2, respectively. That is, we compute $\tilde{p}_m^1$ as the most extreme point in $\mathcal{H}(\pi_m)$ in the direction ρ_1. Then, if $\tilde{p}_m^1 \in \mathcal{H}(\pi)$, then we let $p_m^1 = \tilde{p}_m^1$. Otherwise, no such p_m^1 exists. Likewise for p_m^2 and ρ_2. The correctness of this strategy for computing p_m^1 and p_m^2 follows from the observation that because $\partial\mathcal{H}(\pi_r)$ and $\partial\mathcal{H}(\pi_f)$ each have one maximal portion on $\partial\mathcal{H}(\pi)$, any maximal portion of $\partial\mathcal{H}(\pi_m)$ on $\partial\mathcal{H}(\pi)$ will connect on one side to a point from $\partial\mathcal{H}(\pi_r)$ and on the other to a point from $\partial\mathcal{H}(\pi_f)$. As described earlier, we can finally compute $\mathcal{H}(\pi)$ in $O(h + \log n)$ time using the points p_r, p_m^1, p_m^2, and p_f

We now describe our algorithm for Case 6b where we know p_r and it remains to compute p_m^1 and p_f. Using the modification of Lemma 8 described in Case 4, we can find a point $p_{mf} \in \partial\mathcal{H}(\pi_m \cup \pi_f)$ with $p_{mf} \in \partial\mathcal{H}(\pi)$ and $p_{mf} \notin \partial\mathcal{H}(\pi_r)$. If $p_{mf} \in \partial\mathcal{H}(\pi_f)$, then by setting $p_f = p_{mf}$, we can reduce this case to Case 6a (i.e., p_f plays the role of $\tilde{p}_f$). Otherwise, $p_{mf} \in \partial\mathcal{H}(\pi_m)$, and we let $p_m^1 = p_{mf}$. It remains to compute p_f. Using the walking procedure, we can compute A_r and A_m^1. Then, we compute $H = \mathcal{H}(A_r \cup A_m^1)$. As in the above case, we can find e_1 and e_2, the two special edges of H. Let ρ_1 and ρ_2 be the outward normal directions from e_1 and e_2, respectively. We compute $\tilde{p}_f^1$ and $\tilde{p}_f^2$, the most extreme points of $\mathcal{H}(\pi_f)$ in the directions ρ_1 and ρ_2, respectively. If $\tilde{p}_f^1$ or $\tilde{p}_f^2$ is on $\partial\mathcal{H}(\pi)$, then we let p_f be this point. If neither is, then there is no p_f (i.e., π_f does not contribute to $\partial\mathcal{H}(\pi)$; see Figure 8). The correctness of this strategy for computing p_f follows from the observation that because $\mathcal{H}(\pi_r)$ and $\mathcal{H}(\pi_m)$ each have one maximal portion on $\partial\mathcal{H}(\pi)$, the maximal portion of $\mathcal{H}(\pi_f)$ on $\mathcal{H}(\pi)$ (if it exists) will connect on one side to a point from $\partial\mathcal{H}(\pi_r)$ and on the other to a point from $\partial\mathcal{H}(\pi_m)$. As described earlier, we can finally compute $\mathcal{H}(\pi)$ in $O(h + \log n)$ time using the points p_r, p_m^1, and p_f.

This completes the algorithm for reporting the convex hull $\mathcal{H}(\pi)$. The runtime of the algorithm is $O(h + \log n)$.

4   Lower bounds

In this section, we prove lower bounds, which justify the optimality of our solutions to both the monotone path problem and the simple path problem.

The following theorem gives a lower bound for the two-sided monotone path problem by a reduction from the set inclusion problem [3]. Note that the lower bound is even applicable to the amortized cost. It implies that our result in Theorem 5 is the best possible.
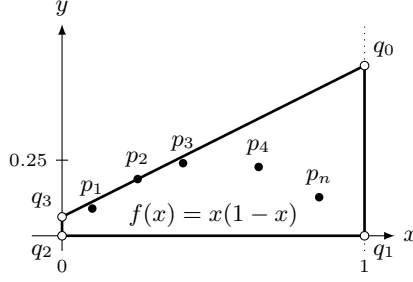
Figure 9 Illustrating the lower bound reduction.

▶ **Theorem 12.** *The following lower bounds hold for the two-sided monotone path dynamic convex hull problem.*

1. *At least one of the operations* INSERTLEFT, DELETELEFT, INSERTRIGHT, DELETERIGHT, *and* HULLREPORT *is required to take* $\Omega(\log n)$ *amortized time under the algebraic decision tree model, regardless of the value h, where n is the size of the point set P and $h = |\mathcal{H}(P)|$.*
2. *Similarly, at least one of the operations* INSERTLEFT, DELETELEFT, INSERTRIGHT, DELETERIGHT, *and* STANDARDQUERY *is required to take* $\Omega(\log n)$ *amortized time, where* STANDARDQUERY *could be (but is not limited to) any of the following queries: extreme point queries, tangent queries, bridge queries.*

Proof. We begin with proving the first theorem statement. The second one can be proven similarly.

We use a reduction from the set inclusion problem. Let $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ be two sets of n real numbers each, such that the numbers of B are distinct and sorted, i.e., $b_1 < b_2 < \cdots < b_n$. Without loss of generality, we assume that all numbers of A and B are larger than 0 and smaller than 1. The *set inclusion* problem is to determine whether $A \subseteq B$. The problem requires $\Omega(n \log n)$ time to solve under the algebraic decision tree model [3].

We create an instance of the two-sided monotone path dynamic convex hull problem as follows. Define a function $f(x) = x(1-x)$. For each number $b_i \in B$, we create a point $p_i = (b_i, f(b_i))$ in $\mathbb{R}^2$. Since $b_1 < b_2 < \cdots < b_n$, the points p_i in their index order are sorted from left to right. We show below that whether $A \subseteq B$ can be determined using $O(n)$ of the five operations in the first theorem statement, which will prove the first theorem statement.

First, starting from $P = \emptyset$, we perform n INSERTRIGHT operations for $p_1, p_2, \ldots, p_n$ in this order. Then, for each $a_i \in A$, we show that whether $a_i \in B$ can be determined using $O(1)$ operations. The function f at the point $(x, f(x))$ has a tangent ℓ_x with slope $1 - 2x$ that intersects the vertical lines for $x = 0$ and $x = 1$ in the points $q_3(x) = (0, x^2)$ and $q_0(x) = (1, (1-x)^2)$. By definition, if x is the x-coordinate of the point p_i of P (i.e., $x = b_i$), then ℓ_x is tangent to $\mathcal{H}(P)$ at p_i (see Figure 9, where the tangent goes through the point p_2).

Define $q_0 = q_0(a_i)$, $q_1 = (1, 0)$, $q_2 = (0, 0)$, and $q_3 = q_3(a_i)$. Note that all points of P are below or on the segment $\overline{q_0 q_3}$. We perform the following operations in order: INSERTRIGHT(q_0), INSERTRIGHT(q_1), INSERTLEFT(q_2), INSERTLEFT(q_3), and HULLREPORT. According to the above discussion, if $a_i \in B$, then ℓ_{a_i} must be tangent to $\mathcal{H}(P)$ at the point $p \in P$ whose x-coordinate is equal to a_i. This implies that the HULLREPORT operation will report the following five points: q_i, $0 \le i \le 3$, and p (we could resolve the collinear situation by a standard symbolic perturbation [15,37]). On the other hand, if $a_i \notin B$, then all points of P will be strictly below the line ℓ_{a_i} and thus only the four points q_i, $0 \le i \le 3$, will

be reported by HULLREPORT. Therefore, by checking the output of HULLREPORT, we are able to tell whether $a_i \in B$. If $a_i \notin B$, then we know $A \nsubseteq B$. Otherwise, we continue to check the next number of A, but before doing so, we perform the following operations to delete the four points q_i: DELETELEFT(q_3), DELETELEFT(q_2), DELETERIGHT(q_1), DELETERIGHT(q_0).

The above shows that whether $A \subseteq B$ can be determined using $O(n)$ of the four update operations and the HULLREPORT operation. This proves the first theorem statement.

To prove the second theorem statement, we can follow the same reduction but replace the operation HULLREPORT with a corresponding query operation.

Consider the extreme point queries. After we insert q_i, $0 \leq i \leq 3$, as above, we perform an extreme point query with a direction ρ being the normal of $\overline{q_0 q_3}$. Then, $a_i \in B$ if and only if an extreme point of $\mathcal{H}(P)$ returned by the query has x-coordinate equal to a_i. This means that whether $A \subseteq B$ can be determined using $O(n)$ of the four update operations as well as the extreme point queries.

The tangent queries and the bridge queries can be argued similarly (e.g., by asking a tangent query at q_0 or a bridge query for the vertical line $x = a_i$). ◄

For the simple path problem, we have the following lower bound, which is even applicable to the one-sided case in which insertions and deletions only happen at one end of the path (say, INSERTFRONT and DELETEFRONT). It implies that our result in Theorem 7 is the best possible.

▶ **Theorem 13.** *The following lower bounds hold for the one-sided simple path dynamic convex hull problem.*

1. *At least one of the operations INSERTFRONT, DELETEFRONT, and REPORTHULL is required to take $\Omega(\log n)$ amortized time under the algebraic decision tree model, regardless of the value h, where n is the number of the vertices of the simple path π and $h = |\mathcal{H}(\pi)|$.*
2. *Similarly, at least one of the operations INSERTFRONT, DELETEFRONT, and STANDARD-QUERY is required to take $\Omega(\log n)$ amortized time, where STANDARDQUERY could be (but is not limited to) any of the following queries: extreme point queries, tangent queries, bridge queries.*

Proof. We follow a similar proof to Theorem 12 by a reduction from the set inclusion problem. Define A, B, and p_i, $1 \leq i \leq n$, in the same way as in the proof of Theorem 12.

Starting from $\pi = \emptyset$, we perform n INSERTFRONT operations for $p_1, p_2, \ldots, p_n$ in this order. Clearly, π is a simple path. Then, for each $a_i \in A$, we can determine whether $a_i \in B$ using $O(1)$ operations as follows. Define q_i, $0 \leq i \leq 3$, in the same way as before. We perform the following operations in order: INSERTFRONT(q_0), INSERTFRONT(q_1), INSERTFRONT(q_2), INSERTFRONT(q_3), and HULLREPORT. Since all numbers in A and B are larger than 0 and smaller than 1, π is still a simple path after the above insertions. Following the same analysis as before, whether $a_i \in B$ can be determined based on the output of HULLREPORT. Afterward, we perform the DELETEFRONT operations in the inverse order of the above insertions. This means that whether $A \subseteq B$ can be determined using $O(n)$ operations. This proves the first theorem statement.

The second theorem statement can be proved similarly, following the same argument as in Theorem 12. ◄

Data Availability Statement   Data sharing is not applicable to this article as no datasets were generated or analysed during the current study.

───── **References** ─────

1 Georgii Maksimovich Adel'son-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Doklady Akademii Nauk*, 146(2):263–266, 1962.

2 A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979. `doi:10.1016/0020-0190(79)90072-3`.

3 Michael Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 1983. `doi:10.1145/800061.808735`.

4 Bruce Brewer, Gerth Stølting Brodal, and Haitao Wang. Dynamic Convex Hulls for Simple Paths. In *40th International Symposium on Computational Geometry (SoCG 2024)*, pages 24:1–24:15, 2024. `doi:10.4230/LIPIcs.SoCG.2024.24`.

5 Gerth Stølting Brodal. Finger search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. URL: `https://www.cs.au.dk/~gerth/papers/finger05.pdf`.

6 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log \log n)$ update time. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 57–70, 2000. `doi:10.1007/3-540-44985-X_7`.

7 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002. `doi:10.1109/SFCS.2002.1181985`.

8 Norbert Bus and Lilian Buzer. Dynamic convex hull for simple polygonal chains in constant amortized time per update. In *Proceedings of the 31st European Workshop on Computational Geometry (EuroCG)*, 2015.

9 Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996. `doi:10.1007/BF02712873`.

10 Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *Journal of the ACM*, 48:1–12, 2001. `doi:10.1145/363647.363652`.

11 Timothy M. Chan. Three problems about dynamic convex hulls. *Int. J. Comput. Geom. Appl.*, 22(4):341–364, 2012. `doi:10.1142/S0218195912600096`.

12 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986. `doi:10.1007/BF01840441`.

13 David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Algorithmica*, 10:1–23, 1993. `doi:10.1007/BF01908629`.

14 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989. `doi:10.1016/0022-0000(89)90034-2`.

15 Herbert Edelsbrunner and Ernst P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990. `doi:10.1145/77635.77639`.

16 Joseph Friedman, John Hershberger, and Jack Snoeyink. Efficiently planning compliant motion in the plane. *SIAM Journal on Computing*, 25:562–599, 1996. `doi:10.1145/73833.73854`.

17 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972. `doi:10.1016/0020-0190(72)90045-2`.

18 Ronald L. Graham and F. Frances Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4:324–331, 1983. `doi:10.1016/0196-6774(83)90013-5`.

19 Leonidas Guibas, John Hershberger, and Jack Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry and Applications*, 1:1–22, 1991. `doi:10.1142/S0218195991000025`.

20 Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC)*, pages 49–60, 1977. `doi:10.1145/800105.803395`.

21   John Hershberger and Jack Snoeyink. Cartographic line simplification and polygon CSG formula in $O(n \log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998. `doi:10.1016/S0925-7721(98)00027-3`.

22   John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992. `doi:10.1007/BF01994880`.

23   John Hershberger and Subhash Suri. Off-line maintenance of planar configurations. *Journal of Algorithms*, 21(3):453–475, 1996. `doi:10.1006/jagm.1996.0054`.

24   Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 93–102, 1995. `doi:10.1145/225058.225090`.

25   Haim Kaplan, Robert E. Tarjan, and Kostas Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 836–844, 2001. `doi:10.5555/365411.365793`.

26   David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986. `doi:10.1137/0215021`.

27   Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

28   Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25(1):11–12, 1987. `doi:10.1016/0020-0190(87)90086-X`.

29   Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. `doi:10.1007/BFB0014927`.

30   Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981. `doi:10.1016/0022-0000(81)90012-X`.

31   Franco P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22:402–405, 1979. `doi:10.1145/359131.359132`.

32   Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986. `doi:10.1145/6138.6151`.

33   Rajamani Sundar. Worst-case data structures for the priority queue with attrition. *Information Processing Letters*, 31:69–75, 1989. `doi:10.1016/0020-0190(89)90071-9`.

34   Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985. `doi:10.1016/S0019-9958(85)80034-6`.

35   Haitao Wang. Dynamic convex hulls under window-sliding updates. In *Proceedings of the 18th Algorithms and Data Structures Symposium (WADS)*, pages 689–703, 2023. `doi:10.1007/978-3-031-38906-1_46`.

36   Haitao Wang. Algorithms for subpath convex hull queries and ray-shooting among segments. *SIAM Journal on Computing*, 53:1132–1161, 2024. `doi:10.1137/21M145118X`.

37   Chee-Keng Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Journal of Computer and System Sciences*, 40:2–18, 1990. `doi:10.1016/0022-0000(90)90016-E`.