in the following section. It maintains 2 balanced binary search trees for the upper and the lower hull respectively. We show the algorithms in action on a set of randomly generated points. For our version of Preparata's Algorithm, we also depict handling of three kinds of queries - (a) whether a point lies inside the hull, (b) tangent from a given point and (c) farthest point on the hull in a direction.

4 Formulation, Data Structures and Theoretical Results

4.1 Point location, insertion and tangents

The following notations are used throughout the section.

An ordered set is a set with some intrinsic order. It is denoted here as $(e_0, e_1, ..., e_{n-1})$. This ordered set has n elements and there is some total order among its elements which explains their positions. A proper prefix of an ordered set is the ordered set $(e_0, e_1, ..., e_i)$ for some $0 \leq i < n$. Similarly, a proper suffix of some ordered set is the ordered set $(e_i, e_{i+1}, ..., e_{n-1})$. A prefix or a suffix of the ordered set is then defined as the complement of some proper suffix or proper prefix respectively.

Lowercase alphabets like p denote a point (p_x, p_y) in the Euclidean plane. Uppercase alphabets like S denote a set of points.
$\prec$ denotes the lexicographical ordering of points in $\mathbb{R}^2$ based on their coordinates. $p \prec q$ or $(p_x, p_y) \prec (q_x, q_y)$ iff $p_x < q_x \vee p_x = q_x \wedge p_y < q_y$. $p \preceq q \iff \neg(q \prec p)$.
We also denote common vector operations like vector sum, difference and dot and cross products as follows.

$p \pm q = (p_x \pm q_x, p_y \pm q_y)$

$p \cdot q = p_x q_x + p_y q_y$

$p \times q = p_x q_y - q_x p_y$ (just the magnitude of the cross product)

The commonly used orientation test (or "counter-clockwise test") can be performed using vector cross products. $ccw(u, v, p)$ tests whether point p lies strictly on the left of the line passing through u and v, when the observer is facing v standing at u.

$$ccw(u, v, p) \equiv (v - u) \times (p - u) > 0$$

$CH(S)$ denotes the convex hull region of the point set S. We define upper and lower hulls of S, denoted by $LH(S)$ and $UH(S)$ respectively, as follows.

$$LH(S) \doteq \{p \in S : (p_x, y) \notin CH(S) \forall y < p_y\} \cup max_{\prec}\{S\}$$
$$UH(S) \doteq \{p \in S : (p_x, y) \notin CH(S) \forall y > p_y\} \cup min_{\prec}\{S\}$$

Note that $CH(S)$ spans the complete area of the hull, whereas $LH(S)$ and $UH(S)$ are a subset of vertices of $CH(S) \cap S$.

The upper and lower hulls intersect at exactly two points : $min_{\prec}\{S\}$ and $max_{\prec}\{S\}$. The lower and upper hulls are defined as such, because they partition the line segments formed on $CH(S)$ into two symmetric chains.

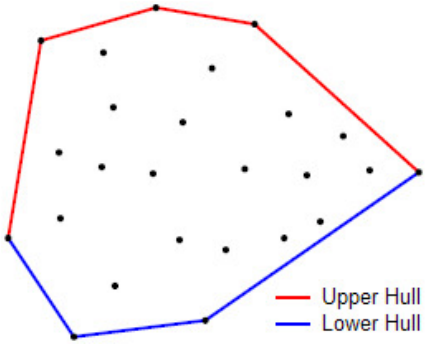

Figure 2: The lower and upper hulls of a set of points.

Let's denote by $-S$ the set $\{(-p_x, -p_y) : (p_x, p_y) \in S\}$, where each point undergoes a centrosymmetric transform $(p_x, p_y) \mapsto (-p_x, -p_y)$

Notice the following relation between the two hulls : $UH(S) = -LH(-S)$

This allows us to just focus the rest of our analysis on the lower hull $LH(S)$ and claim similar results for the upper hull by symmetry. We can hence maintain these hulls separately in a single structure.

We now consider the task of maintaining $LH(S)$ subject to addition of points to S. $LH(S)$ is stored in memory as an ordered set of $n-1$ point pairs $((p_0, p_1), (p_1, p_2), ..., (p_{n-2}, p_{n-1}))$, where each pair corresponds to a line segment $\overline{p_i p_{i+1}}$ on $CH(S)$. The points satisfy the inequalities : $p_0 \preceq p_1 \preceq ... \preceq p_{n-1}$. Furthermore $p_0 = min_{\prec}\{S\}$, and $p_{n-1} = max_{\prec}\{S\}$.

For the addition of a new point p, we consider only three mutually disjoint cases.

1. $p \prec p_0$

   In this case some prefix $((p_0, p_1), ..., (p_{i-1}, p_i))$ of $LH(S)$ is removed, and is replaced by a single element $((p, p_i))$ corresponding to the $\overline{pp_i}$ tangent. The prefix of points corresponds the points that will be in the interior of the newly formed $CH(S \cup \{p\})$. Note that this index i can be found quickly, as will be shown below.

2. $p_{n-1} \prec p$

   This case is similar to the first case, only now we delete some suffix, and replace it

with a similar tangent. Note that in the first two cases, the point p is definitely outside $CH(S)$.

3. $\neg(p \prec p_0 \vee p_{n-1} \prec p)$

   $\neg(p \prec p_0 \vee p_{n-1} \prec p) \iff p_0 \preceq p \preceq p_{n-1} \implies \exists j : 0 \leq j < n-1, p_j \preceq p \preceq p_{j+1}$ Note that this index j can be found quickly, as will be shown below. Now we can check if p can lie in $CH(S)$ by an orientation test : $p \in CH(S) \iff (p_{j+1} - p_j) \times (p - p_j) \geq 0$

   If the point lies outside, we split $LH(S)$ into three parts : $left \doteq ((p_0, p_1), ..., (p_{j-1}, p_j)))$, $mid \doteq ((p_j, p_{j+1}))$ and $right \doteq ((p_{j+1}, p_{j+2}), ..., (p_{n-2}, p_{n-1}))$
   In this sub case, both p_j and p_{j+1} are in the interior of $CH(S \cup \{p\})$. So we remove the segment mid. We then follow the same procedure as in case 2 for $left$ and the one in case 1 for $right$.

With the process described above, we handle point insertion while also computing the tangents. To get the tangents without adding p to S, we simply retrieve the indices, and the corresponding points without deleting / adding any segments. For p to lie in $CH(S)$, case 3 must be applicable to p, and both of the orientation tests in $LH(S)$ and $UH(S)$ must pass.

Cases 1 and 2 give us one tangential point, and case 3 gives us both of them. If a point satisfies the conditions for cases 1 or 2, then it does so for both lower and upper hulls. So, in cases 1 and 2, the other tangent comes from this procedure applied on the upper hull.

To find the tangents, and the mid segment in all of the above operations, we perform binary search on the ordered sequences $LH(S)$, $left$ and $right$, whose details are mentioned in the implementation section.

4.2 Implementation and data structures

We need some additional concepts to implement the operations described in 4.1.

For an ordered set $A = (e_0, e_1, ..., e_{n-1})$, a monotone predicate is a boolean valued function $\phi : A \mapsto \{True, False\}$, satisfying the conditions $\phi(e_i) \implies \phi(e_j) \forall i, j, i \leq j$. This means that there exists a point of split at some index i such that $\phi(e_j) \forall j \geq i$ and $\neg\phi(e_j) \forall j < i$. We can find that point of split quickly, in time $O(log(n))$ using binary search.

The primary advantage of representing the convex hull as two separate chains $LH(S)$ and $UH(S)$ is that (a) they can be represented as ordered sets, and (b) there exist some monotone predicates involving the hulls and any point p such that all of the operations given in section 4.1 can be performed in $O(log(n))$ with binary search.

1. $p \prec p_0$: The deleted prefix and the remaining suffix are the split in $LH(S)$ obtained by $\phi_p((u, v)) \doteq (v - u) \times (p - u) > 0$.

2. $p_{n-1} \prec p$: The deleted suffix and the remaining prefix are the split in $LH(S)$ obtained by $\phi_p((u, v)) \doteq (v - u) \times (p - u) \leq 0$.

3. $p_0 \preceq p \preceq p_{n-1}$: The predicate $\phi_p((u,v)) \doteq p \prec v$ splits $LH(S)$ into $left$ and $mid + right$, while $\psi_p(u,v) \doteq p \preceq u$ splits $mid + right$ into mid and $right$. (+ here denotes concatenation)

We can use balanced binary trees like red-black trees (ref. Guibas and Sedgewick (1978)) or AVL trees (ref. Adelson-Velsky and Landis (1962)) to implement the ordered sets $LH(S)$, $UH(S)$, and any intermediate ordered sets encountered during the above operations. The split and join operations on these ordered sets can be implemented by physically splitting and joining the corresponding binary trees.

The procedure to split a binary tree T by a given monotone predicate ϕ can be abstracted out as in algorithm 3. Here, the symbol T denotes a node in the binary tree, $left(T)$ and $right(T)$ are the two children of T, and ϕ acts on some data stored as part of T. The procedure $SPLIT(T, \phi, L, R)$ splits T by ϕ into L and R, such that $\neg\phi(e) \forall e \in L$ and $\phi(e) \forall e \in R$.

Algorithm 3 SPLIT ORDERED SET

```
1:  procedure SPLIT(T, φ, L, R)
2:      if T = NULL then
3:          L ← NULL, R ← NULL
4:      if φ(T) then
5:          SPLIT(left(T), φ, L, temp)
6:          left(T) ← temp
7:          R ← T
8:      else
9:          SPLIT(right(T), φ, temp, R)
10:         right(T) ← temp
11:         L ← T
```

The $SPLIT(T, \phi, L, R)$ procedure runs in time proportional to the number of nodes visited, which is at most the height of the binary tree rooted at T. Various implementations of balanced binary trees strive to keep the tree height balanced, by maintaining some additional invariants. Our implementation uses randomized binary search trees - treaps, for simplicity of demonstration (ref. Aragon and Seidel (1989)). As this is a randomized data structure, the height of T, and hence the running time is $O(log(n))$ *in expectation*. But with other choices of binary trees, the running time can be made worst case $O(log(n))$ by using the same theory.

For treaps, the join procedure is defined in algorithm 4. Here, the $JOIN(L, R, T)$ procedure takes two trees L and R and joins them to create T so that all of the elements in L occur before R in T. By design, the join process (for treaps) is randomised by using a uniformly random boolean valued hash function $\pi : T^2 \mapsto \{True, False\}$. That is, $\pi(L, R)$ is either true or false, each with equal probability, for all pairs (L, R). The join procedure may be different for other variants of balanced binary trees. All of them have similar running times as splitting T back into L and R, i.e $O(log(n))$ in expectation or in worst case. Also note that the removal of some segments can take different amounts of time over the operations performed, but since each segment is only added and removed at most once, the

running time for deleting the nodes is amortized $O(1)$ over all of the operations. We can overcome this issue by lazily removing the segments by some separate unrelated entity (e.g another process, or some memory management tool.)

Algorithm 4 JOIN ORDERED SETS

```
1:  procedure JOIN(L, R, T)
2:      if L = NULL then
3:          T ← R
4:      if R = NULL then
5:          T ← L
6:      if π(L, R) then
7:          JOIN(L, left(R), temp)
8:          left(R) ← temp
9:          T ← R
10:     else
11:         JOIN(right(L), R, temp)
12:         right(L) ← temp
13:         T ← L
```

Using these procedures as primitive operations, we can implement all of the methods described in 4.1.

4.3 Farthest point queries

The problem of querying the farthest point in a direction can be formulated as follows :

Given a set of points S, and a non zero direction vector v, find the subset of points $argmax_{p \in S}\{v \cdot x\}$ which maximise the dot product with v. For a convex hull $CH(S)$ of some point set S, the output is either a single point, or a line segment on its boundary.

We can solve farthest point queries too in $O(log(n))$, using the techniques discussed in 4.1 and 4.2. We again focus on the lower hull $LH(S)$ of some points. We also restrict the direction vector v to satisfy $v_y < 0 \vee v_y = 0 \wedge v_x > 0$. If it doesn't satisfy this condition, the farthest point or line segment lies on the upper hull.

We split the points in $LH(S)$ by the predicate $\phi_v((p, q)) \doteq q \cdot v \leq p \cdot v$, and look at the first segment (x, y) satisfying ϕ. If $y \cdot v < x \cdot v$, point x is the solution, and if $y \cdot v = x \cdot v$, the line segment $\overline{xy}$ is the solution. If no segment satisfies ϕ, $max_{\prec}\{S\}$ is the solution.

5 Experimental Setup and Results

5.1 Modified Preparata's Algorithm

Through a profiling experiment, we seek to obtain empirical evidence for the expected O(log N) time complexity for the update operation. For this experiment, we take 100000 randomly generated points and record the time taken to update the hull after adding each