Dynamic point set problems in the plane

Sumeet Shirgure

Dynamic convex hulls

Given a *dynamic* set of points in the plane, i.e points are being added and removed; *maintain* the convex hull of these points while answering tangent and farthest point queries.

I give exact and robust C++ implementations of the problem in two settings :

- *Online* : points are just being added and not removed
- *Fully dynamic* : points are being added and removed

Preliminary data structures

Central is the idea of ordered sequences.

E.g a list sorted in ascending order
[1, 4, 6, 7, 9]

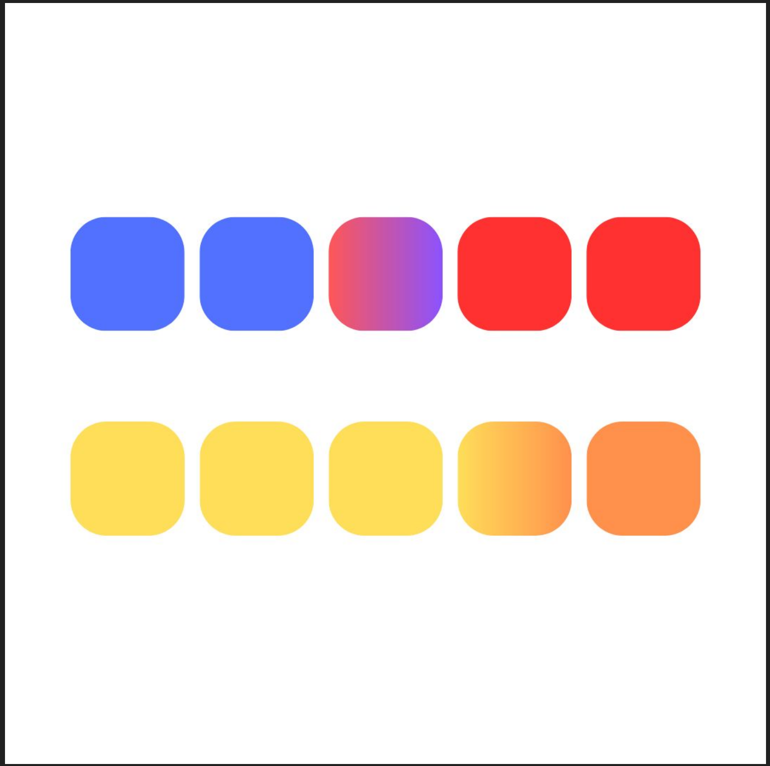
All we need is a total ordering among the elements to define an ordered sequence.

Another idea is that of *monotone predicates*.

Boolean functions that are false for some prefix and true for the rest of the suffix.

(x>5?) : (Blue)[1, 4]   (Red)[6, 7, 9]

(x>6?) : (Yellow)[1,4,6], (Orange)[7, 9]

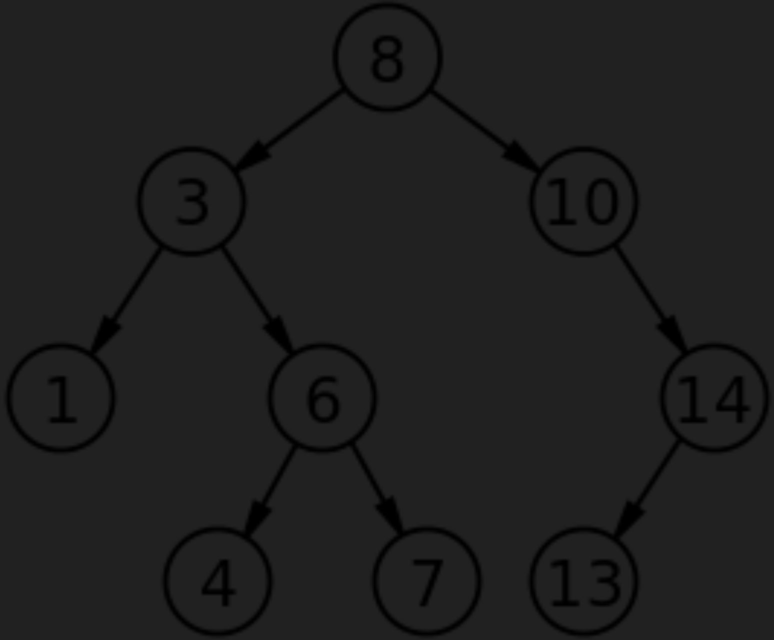


Preliminary data structures

Represent an ordered sequence in memory as a binary search tree.

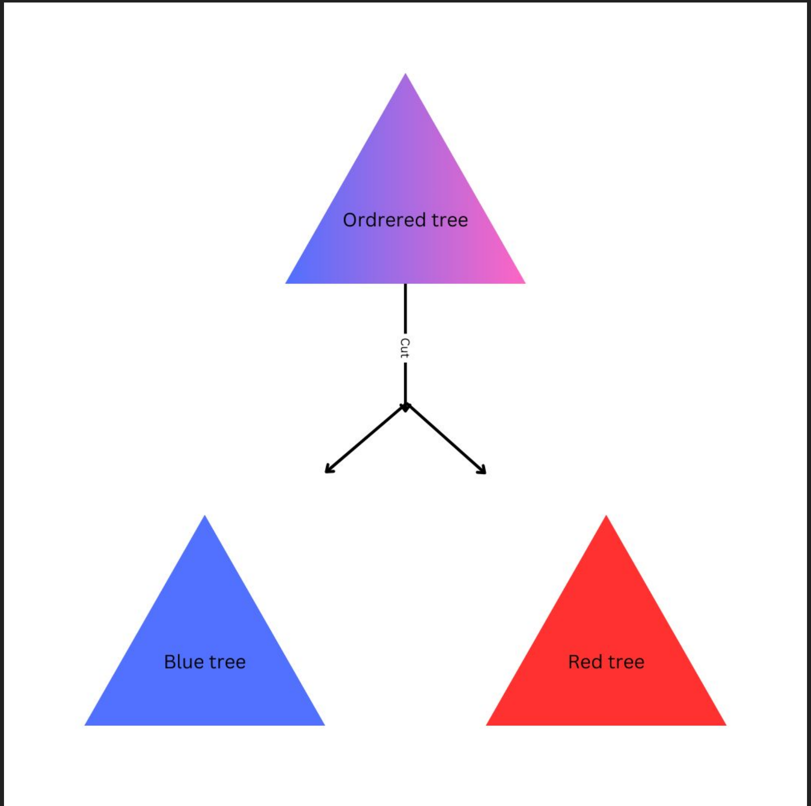
E.g : the tree on the right represents the sequence [1, 3, 4, 6, 7, 8, 10, 13, 14]

Let’s color the split (x>6?)



Preliminary data structures

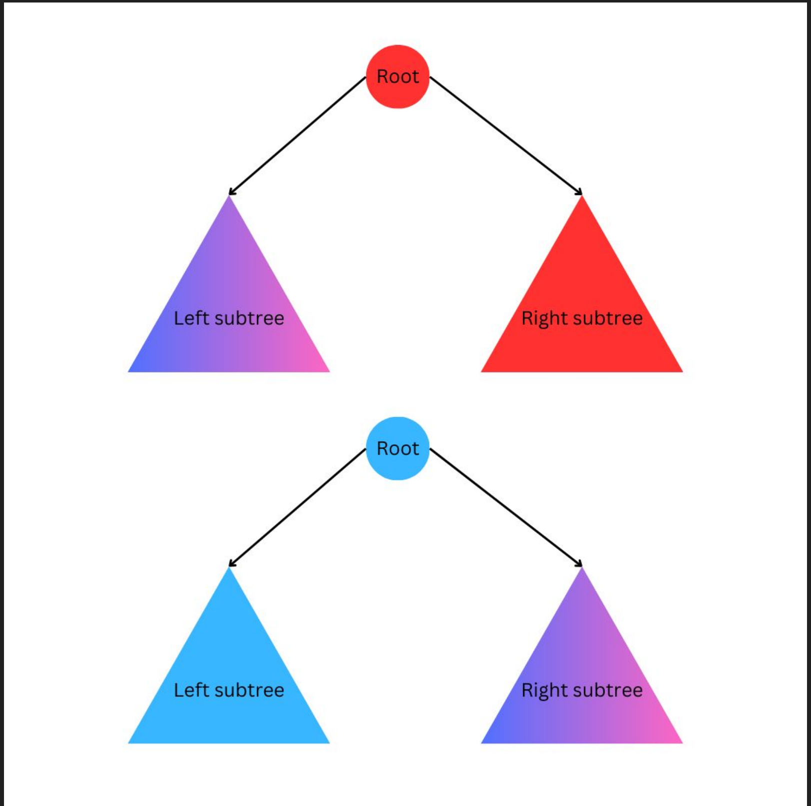
Consider the task of splitting the sequence into the blue prefix and a red suffix.



Preliminary data structures

Consider the task of splitting the sequence into the blue prefix and a red suffix.

Note that if the root is labelled red, so are all the children in the right subtree.

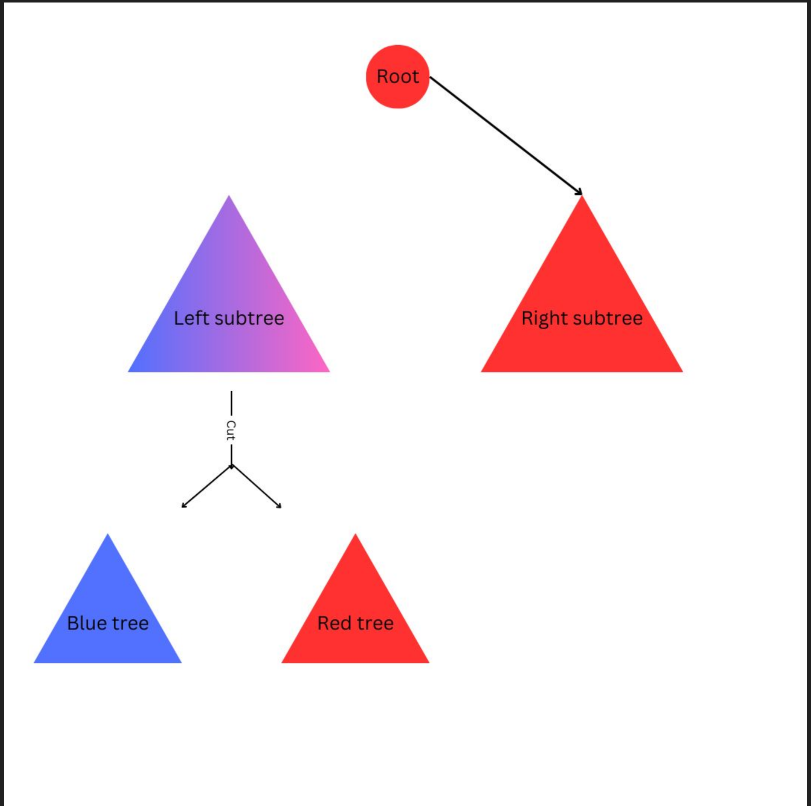


Preliminary data structures

Consider the task of splitting the sequence into the blue prefix and a red suffix.

Note that if the root is labelled red, so are all the children in the right subtree.

If root is colored red, recursively split left subtree and attach the resulting red subtree as the left child of the root.



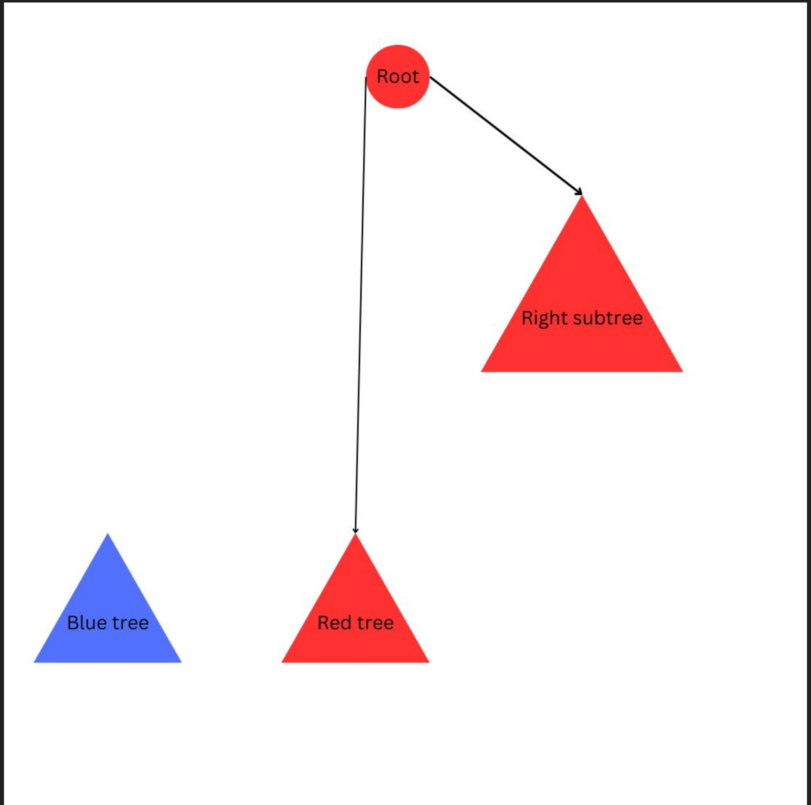
Preliminary data structures

Consider the task of splitting the sequence into the blue prefix and a red suffix.

Note that if the root is labelled red, so are all the children in the right subtree.

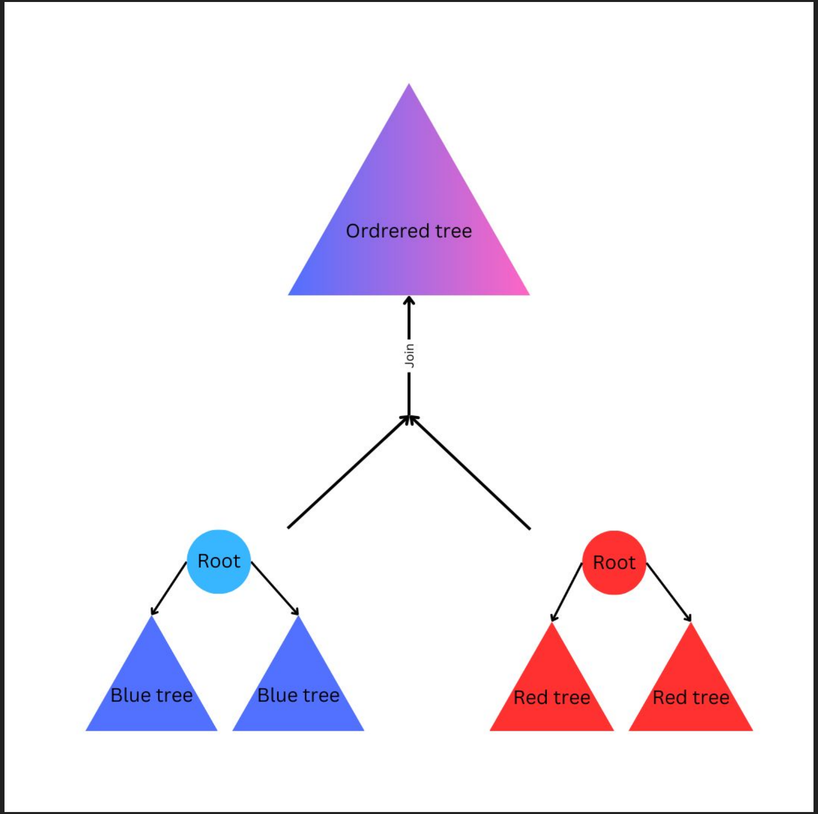
If root is colored red, recursively split left subtree and attach the resulting red subtree as the left child of the root.

Note that we only traverse the depth of the tree. O(h)



Preliminary data structures

For joining two sequences, conveniently colored blue and red, we can do the following :

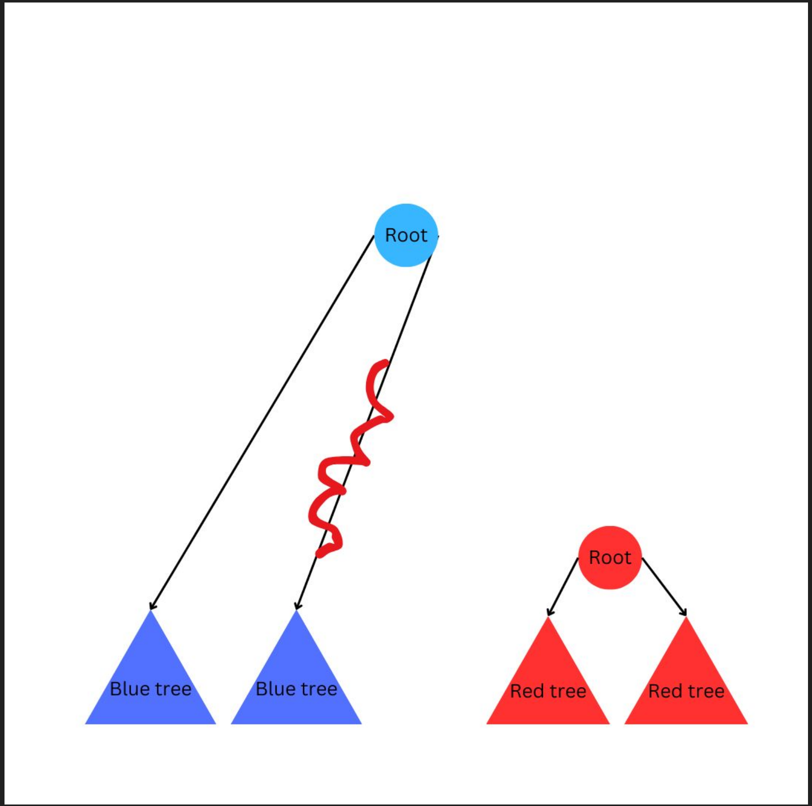


Preliminary data structures

For joining two sequences, conveniently colored blue and red, we can do the following :

Pick one of the two roots randomly.

Make it the new root.



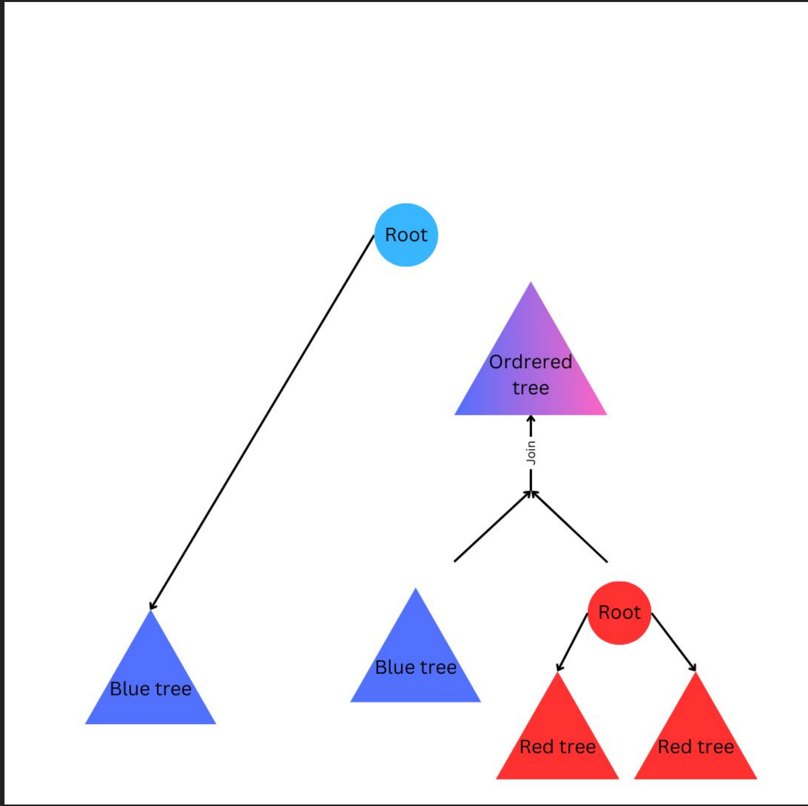
Preliminary data structures

For joining two sequences, conveniently colored blue and red, we can do the following :

Pick one of the two roots randomly.

Make it the new root.

Recursively join the broken subtrees.



Preliminary data structures

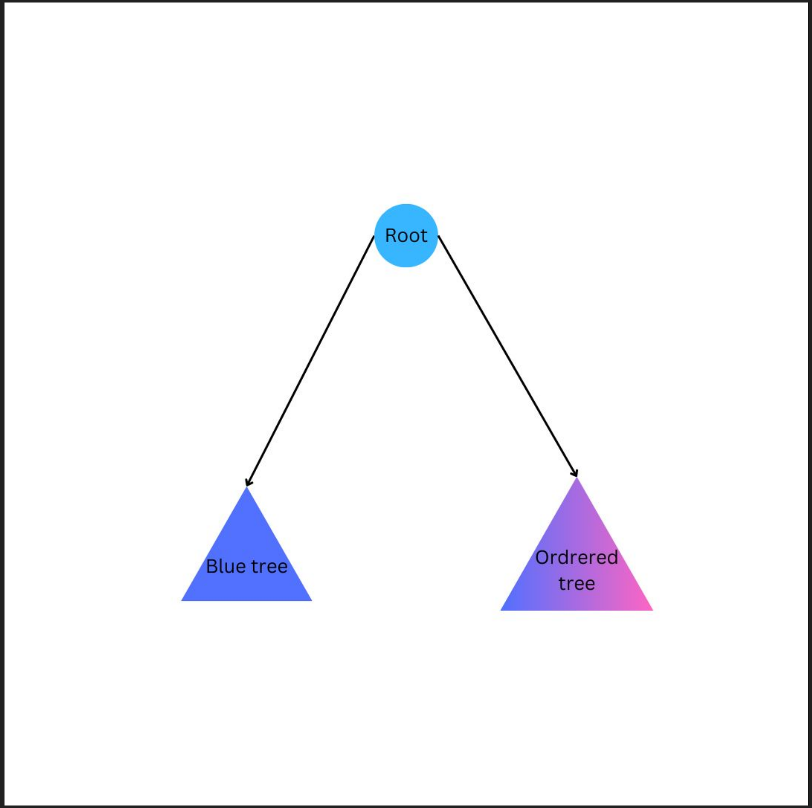
For joining two sequences, conveniently colored blue and red, we can do the following :

Pick one of the two roots randomly.

Make it the new root.

Recursively join the broken subtrees.

Reattach the result as the appropriate child of the root.



Preliminary data structures

For joining two sequences, conveniently colored blue and red, we can do the following :

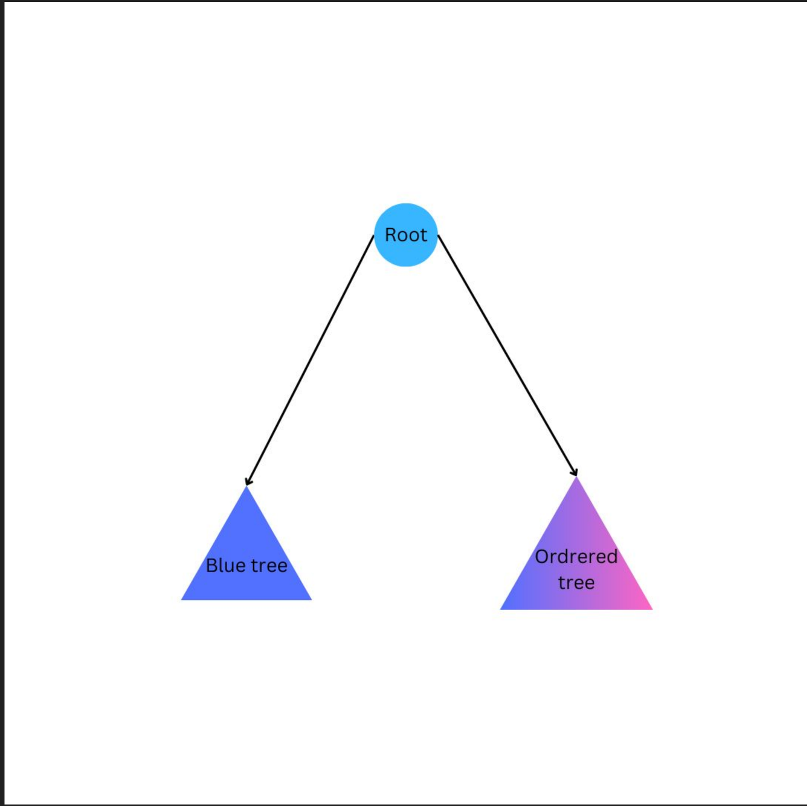
Pick one of the two roots randomly.

Make it the new root.

Recursively join the broken subtrees.

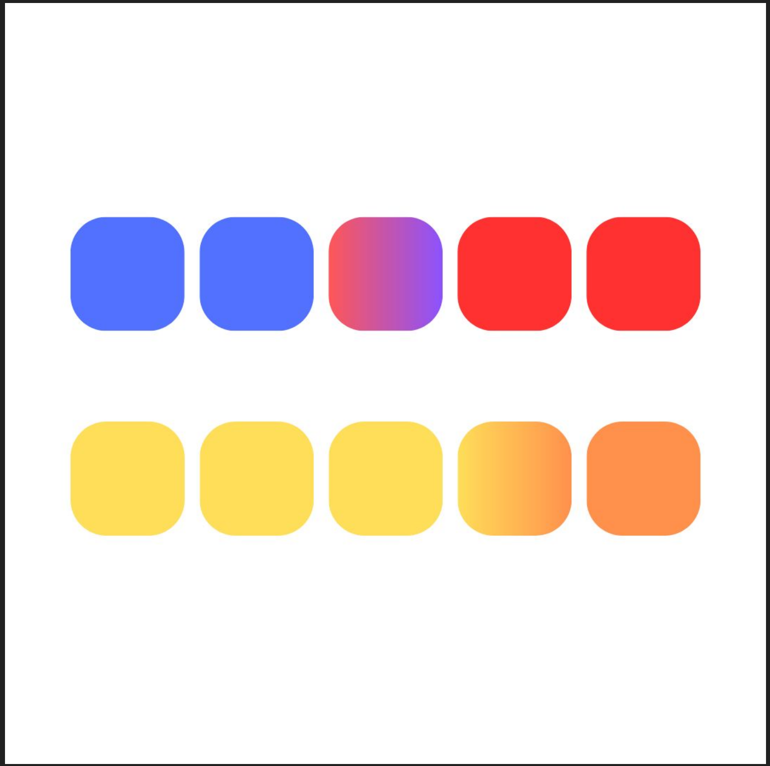
Reattach the result as the appropriate child of the root.

Turns out if we do it randomly enough the tree height is logarithmic.



Preliminary data structures

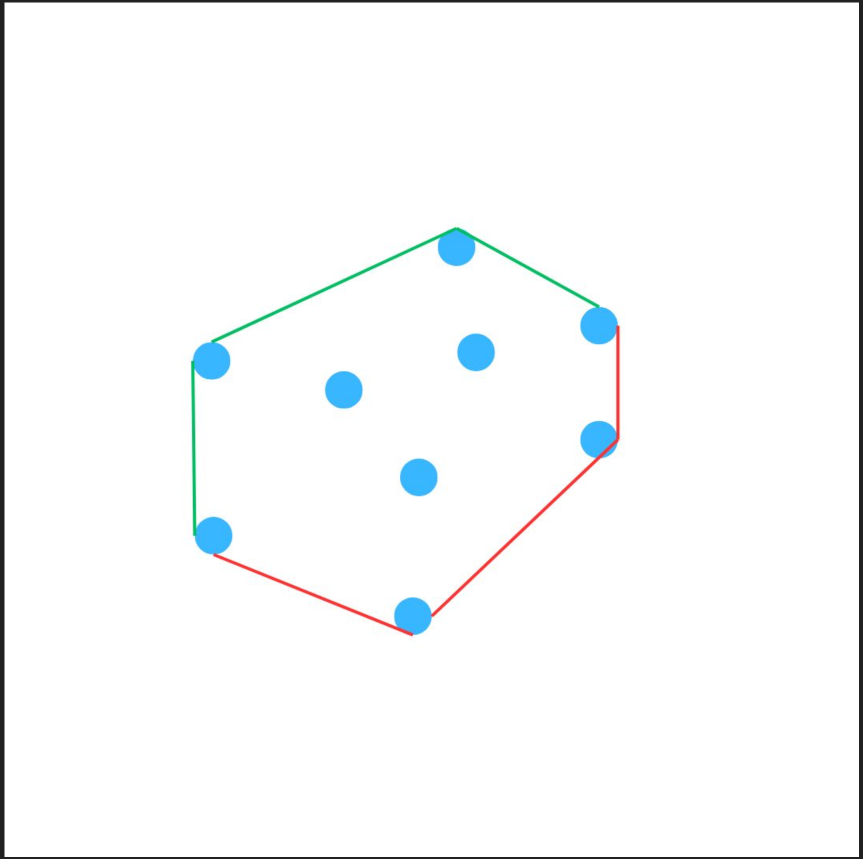
Therefore we have a data structure that allows us to store ordered sequences and split/merge sequences based on arbitrary monotone predicates.



Let’s focus on the lower hull

Red lines are the lower hull.

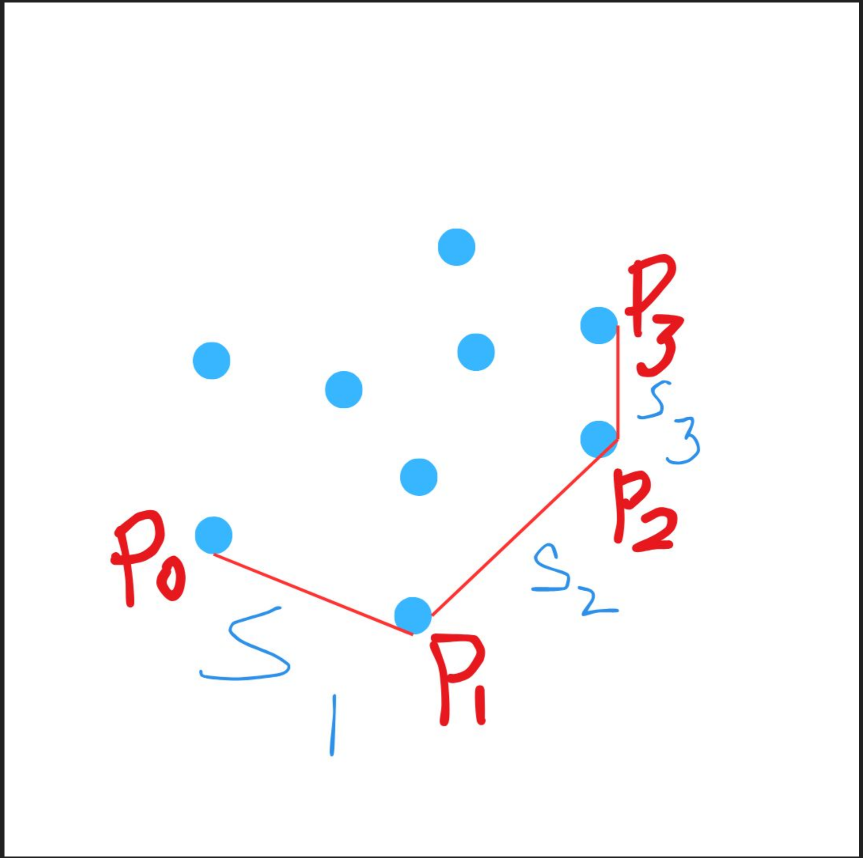
Green lines become red lines when rotated 180 degrees.



Let’s focus on the lower hull

Store the sequence [s1, s2, s3]

[(p0, p1), (p1, p2), (p2, p3)]



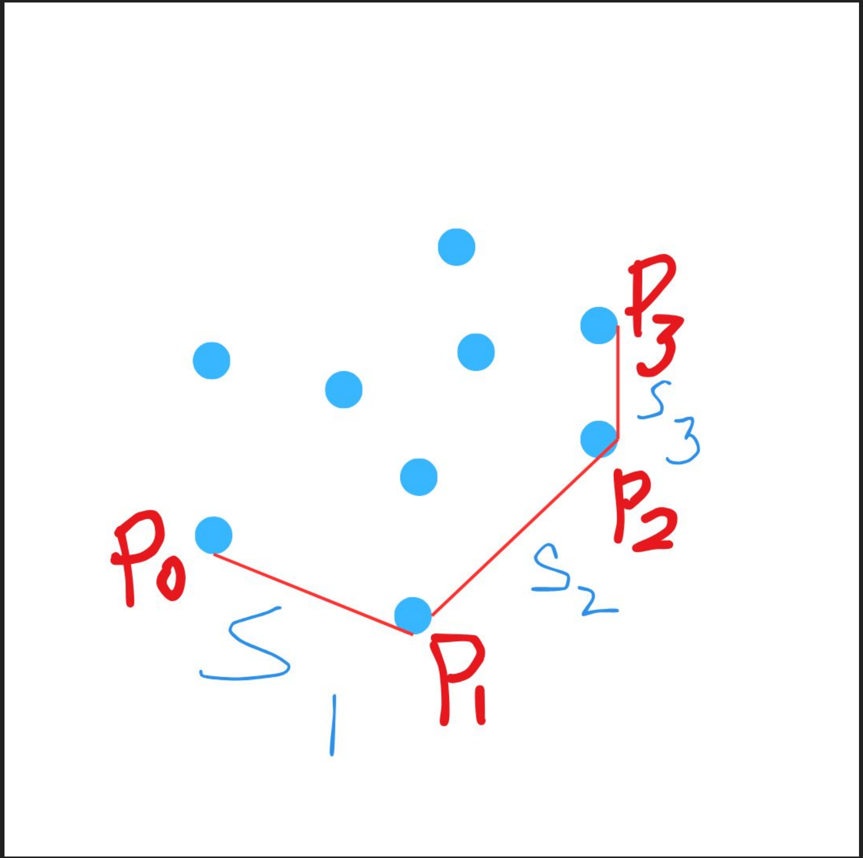
Let’s focus on the lower hull

Store the sequence [s1, s2, s3]

[(p0, p1), (p1, p2), (p2, p3)]

The intrinsic total order is the lexicographical order of the points :

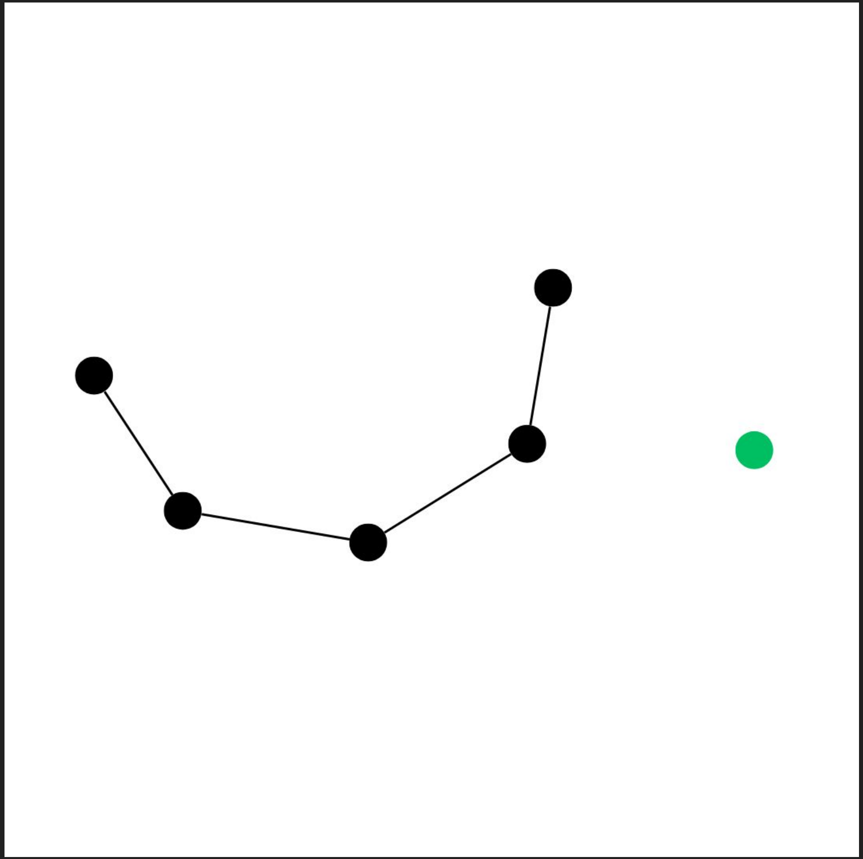
s1 < s2 ⇔ p0 < p1



Adding a new point

Consider the task of adding a new point

Assume that q lies to the right.

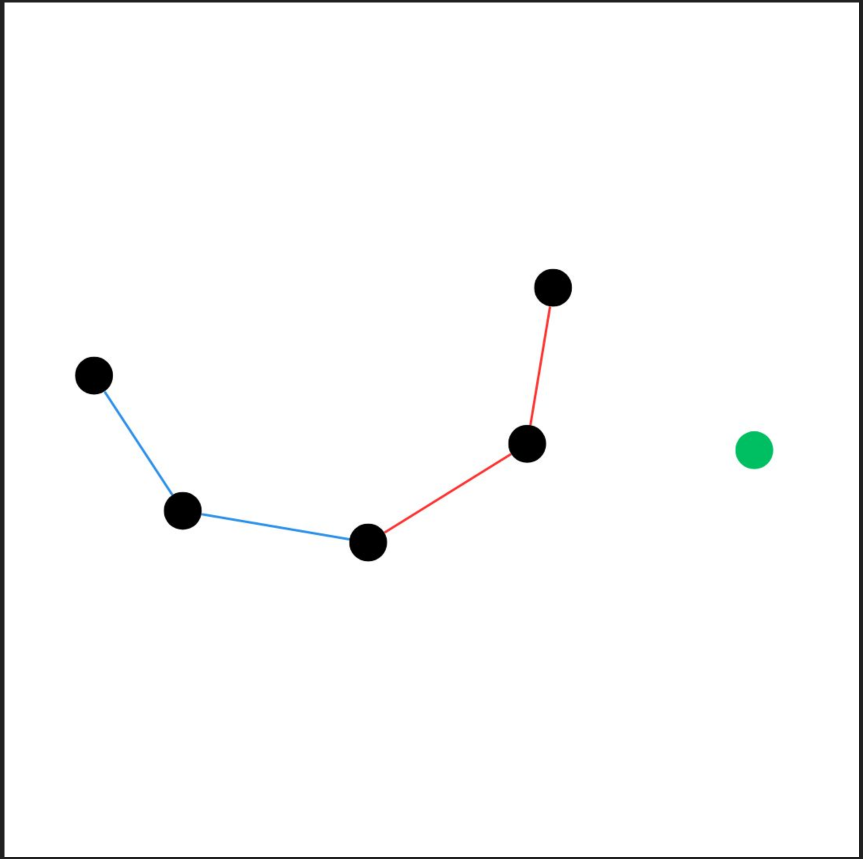


Adding a new point

Consider the task of adding a new point

Assume that q lies to the right.

Color the line segments blue vs. red depending on which side q lies.



Adding a new point

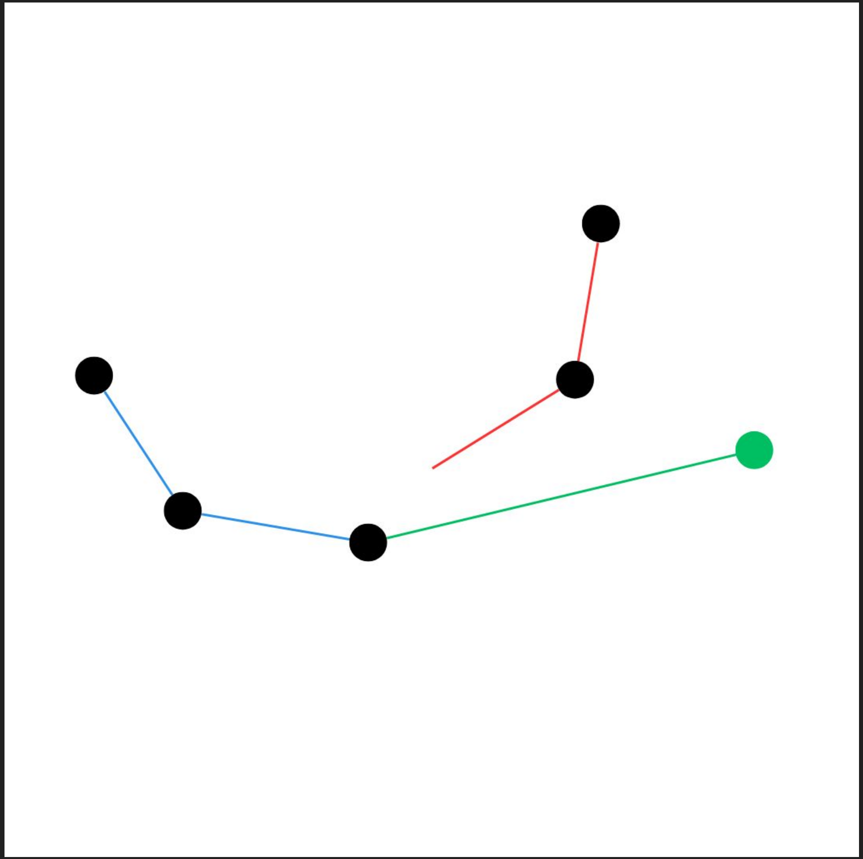
Consider the task of adding a new point

Assume that q lies to the right.

Color the line segments blue vs. red depending on which side q lies.

Note that it forms a monotone predicate

Discard the segment in the interior and construct a new segment.



Adding a new point

Consider the task of adding a new point

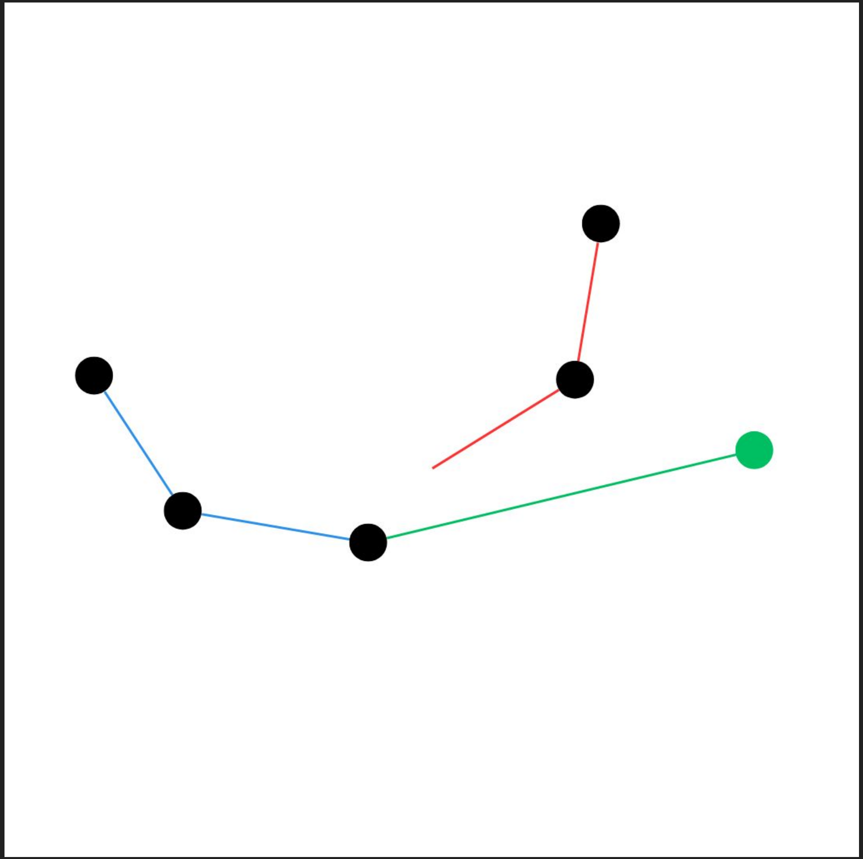
Assume that q lies to the right.

Color the line segments blue vs. red depending on which side q lies.

Note that it forms a monotone predicate

Discard the segment in the interior and construct a new segment.

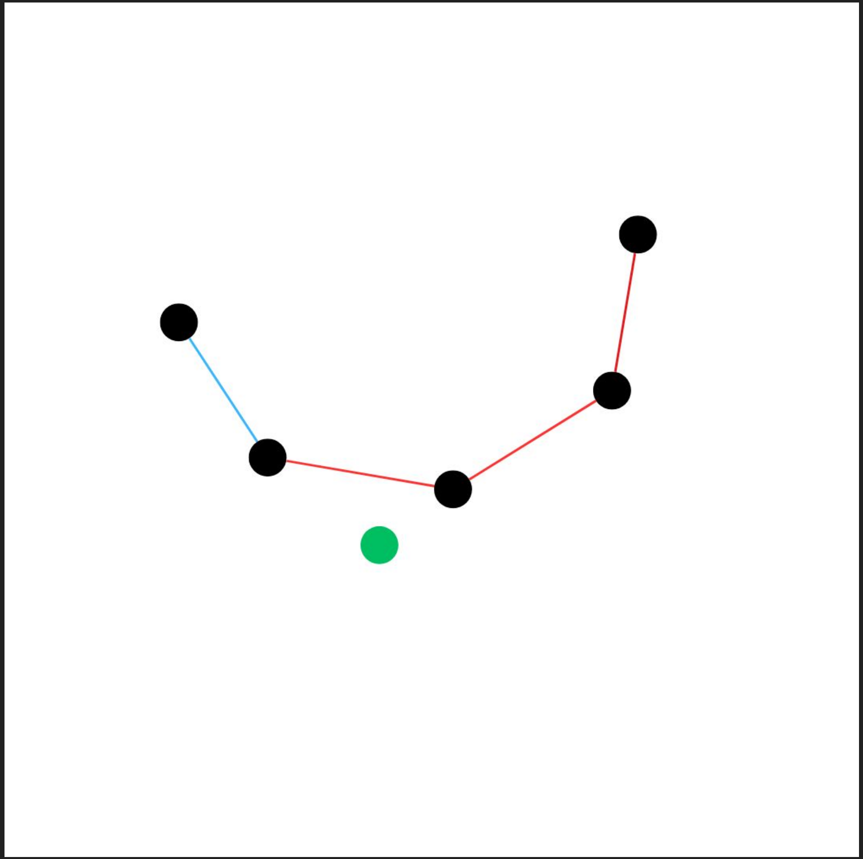
Handle the mirror case symmetrically.



Adding a new point

What if the query point was somewhere in the middle?

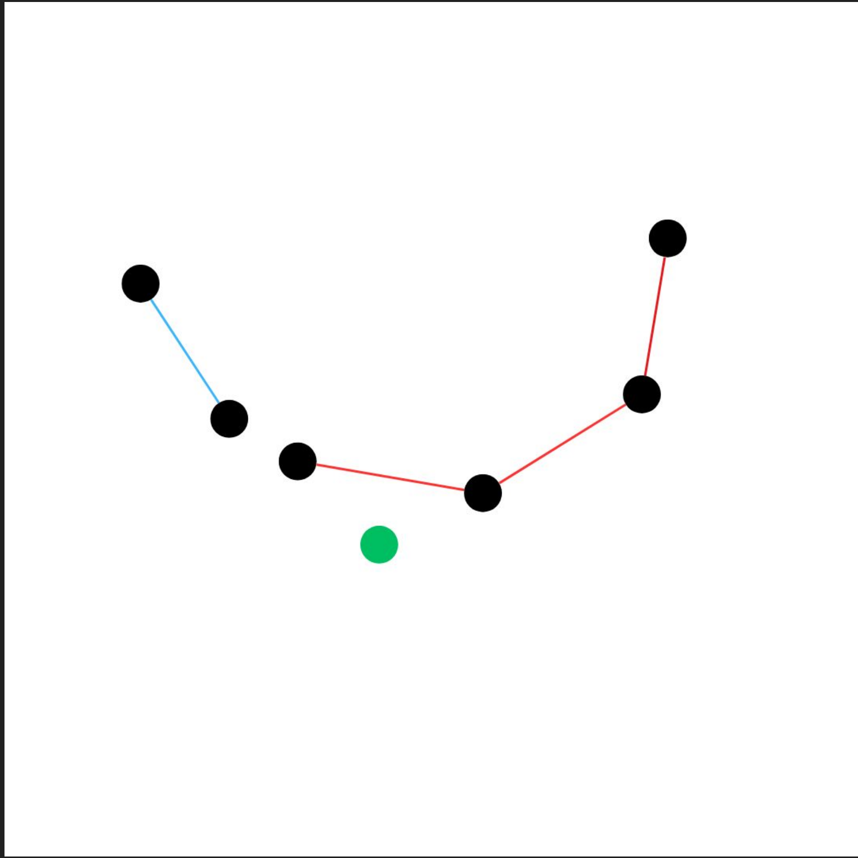
Split by lexicographical ordering



Adding a new point

What if the query point was somewhere in the middle?

Split by lexicographical ordering



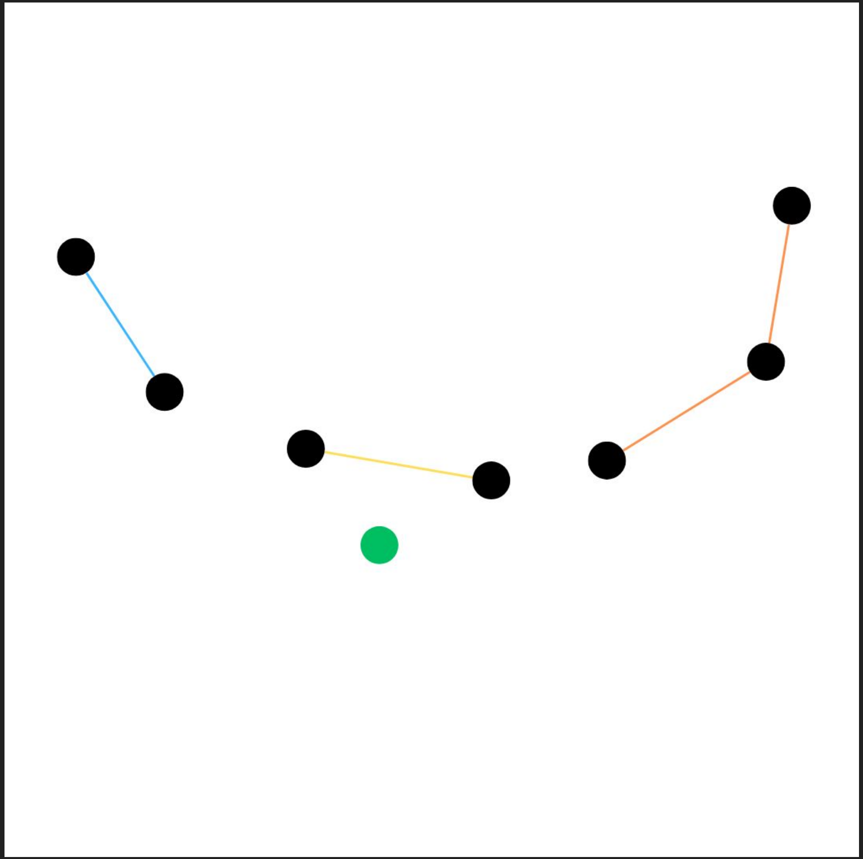
Adding a new point

What if the query point was somewhere in the middle?

Split by lexicographical ordering

There must be a unique segment

(pj, pj+1) that contains q lexicographically



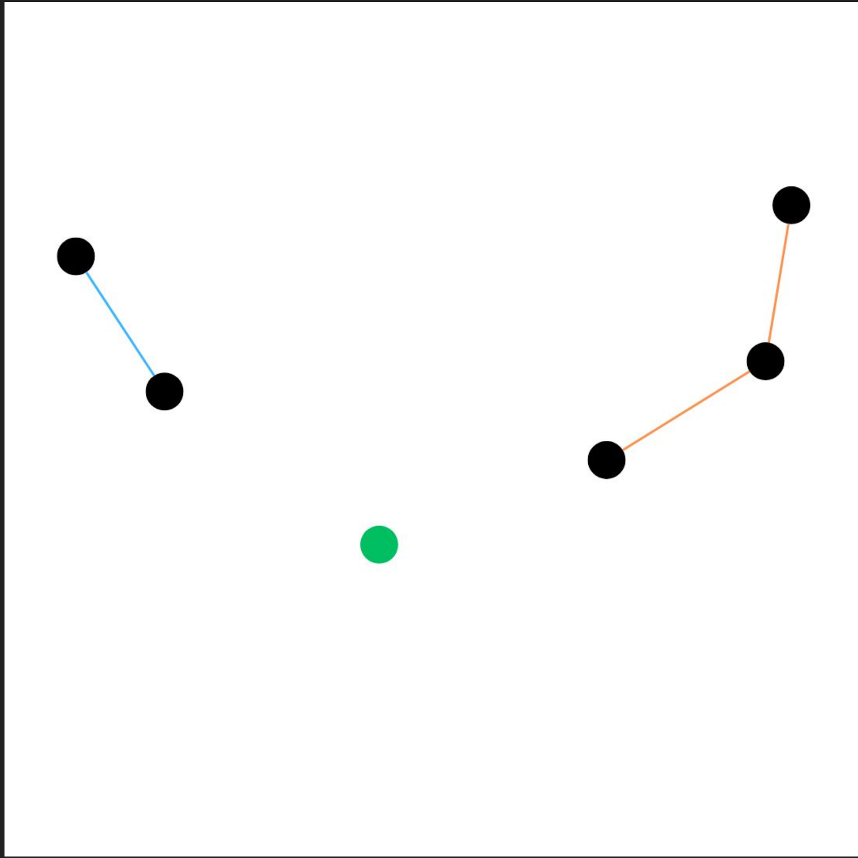
Adding a new point

What if the query point was somewhere in the middle?

Split by lexicographical ordering

There must be a unique segment

(pj, pj+1) that contains q lexicographically



Adding a new point

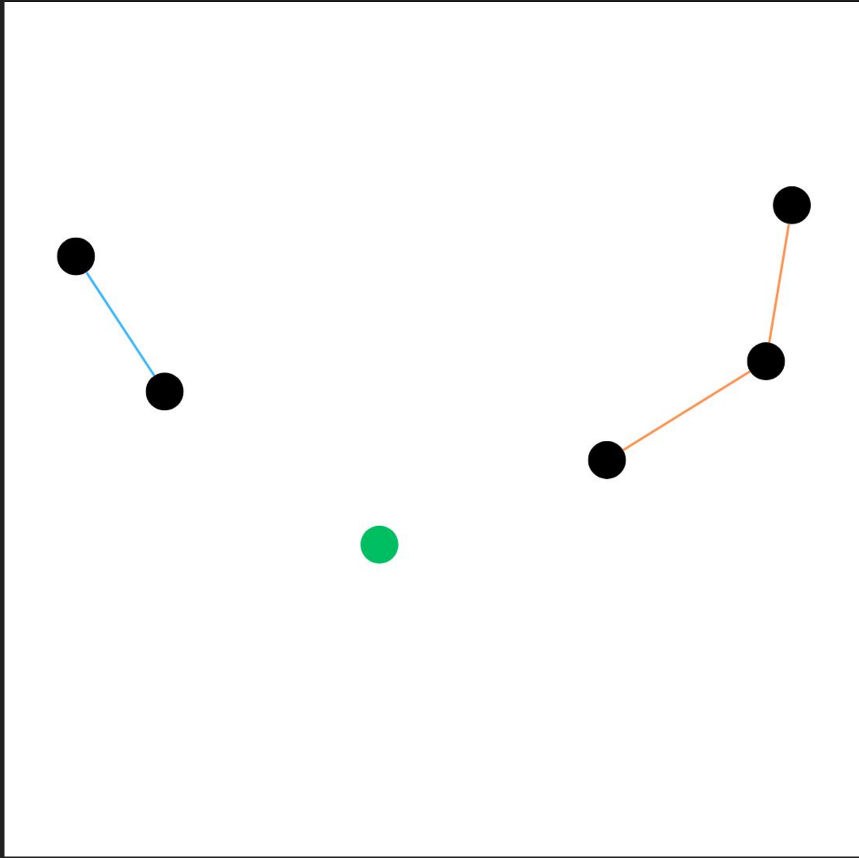
What if the query point was somewhere in the middle?

Split by lexicographical ordering

There must be a unique segment

(pj, pj+1) that contains q lexicographically

Remove that segment and proceed as in the earlier case.



Online hull

We’re halfway through!

Adding points and finding tangents are related and are done with similar tricks.

Dynamic divide and conquer.

Consider the task of finding the minimum of a list of integers.

Humor me for a minute and let’s solve this by divide and conquer.

[A0, A2, …, A(n-1)] → [A0,.. Ai], [A(i+1),...A(n-1)] →left, right → min(left, right)

Dynamic divide and conquer.

Consider the task of finding the minimum of a list of integers.

Humor me for a minute and let’s solve this by divide and conquer.

[A0, A2, …, A(n-1)] → [A0,.. Ai], [A(i+1),...A(n-1)] →left, right → min(left, right)

```
int find_min(int from, int to) {
    if(to - from == 1) return a[from];
    int mid = (from + to) >> 1;
    return min(find_min(from, mid), find_min(mid, to));
} // call find_min(0, n)
```

Dynamic divide and conquer.

Consider the recursion tree of `find_min` : it is a balanced binary tree with each node containing three things : the indices `(from, to)` and the return value `min`

Dynamic divide and conquer.

Consider the recursion tree of `find_min` : it is a balanced binary tree with each node containing three things : the indices `(from, to)` and the return value `min`

The leaves are individual elements of the array.

Dynamic divide and conquer.

Consider the recursion tree of `find_min` : it is a balanced binary tree with each node containing three things : the indices `(from, to)` and the return value `min`

The leaves are individual elements of the array.

Internal nodes are storing the information about the aggregate (minimum) of all the leaves descended from them.

Dynamic divide and conquer.

Consider the recursion tree of `find_min` : it is a balanced binary tree with each node containing three things : the indices `(from, to)` and the return value `min`

The leaves are individual elements of the array.

Internal nodes are storing the information about the aggregate (minimum) of all the leaves descended from them.

The trick is to keep this tree in memory, keep it balanced and perform addition / deletion operations on the leaves like a regular binary search tree.

Dynamic divide and conquer.

So to dynamically maintain the minimum of a set S using “divide and conquer recursion tree” we need a tree T that has the following properties :

1. Approximately height balanced
2. Leaves store data regarding the individual points of S
3. The internal nodes have exactly two children and represent the “conquer” step of the underlying divide and conquer algorithm.
4. Supports addition and deletion of *leaves* in logarithmic time.

One of the key contributions is the first (to my knowledge) open source C++ implementation of exactly such a tree.

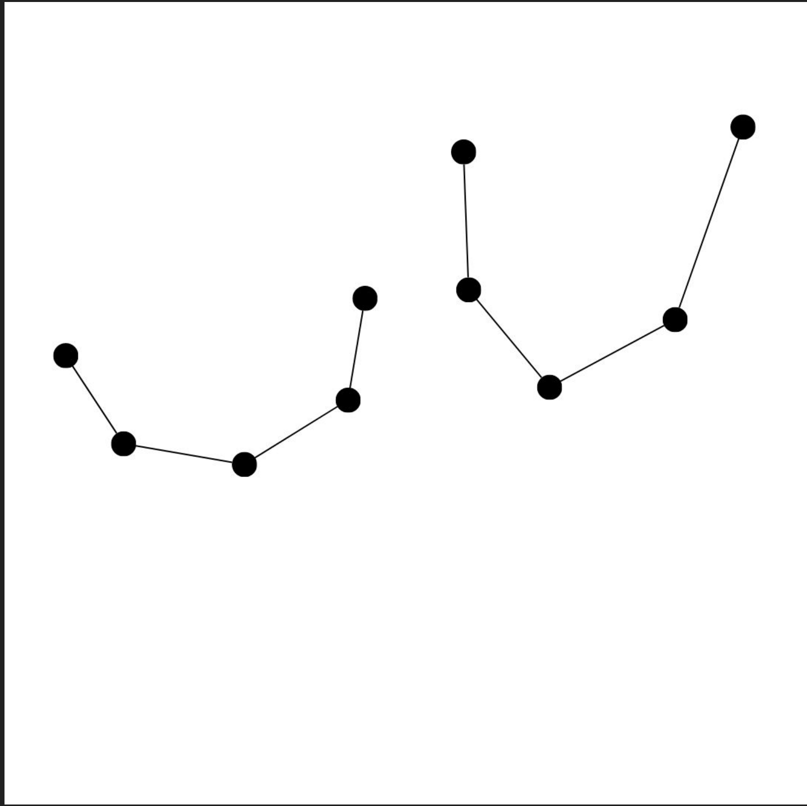
Fully dynamic convex hull in the plane.

Consider a dynamic set S of points in the 2D plane. The idea is to use the “divide and conquer tree” in the following manner :

1. Leaves store individual points in lexicographic order.
2. Internal nodes store the convex hull of all of the leaves reachable from them in the representation discussed earlier.
3. The root node hence stores the convex hull of all the points.

Merging two convex hulls.

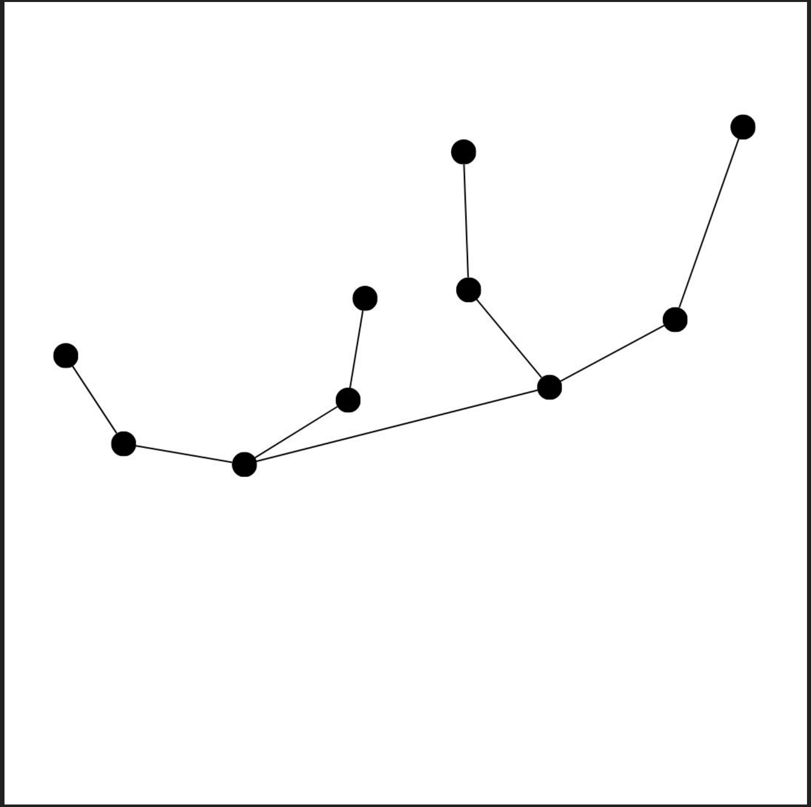
The idea is to find the bridge of two convex hull sequences quickly.



Merging two convex hulls.

The idea is to find the bridge of two convex hull sequences quickly.

Crux of the solution – by Overmars and van Leeuwen



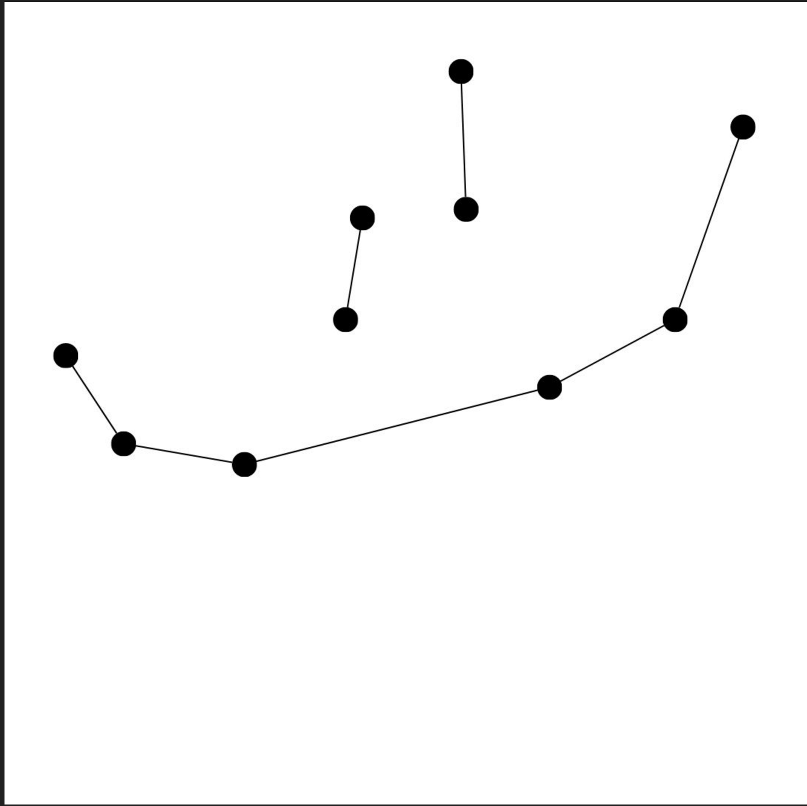
Merging two convex hulls.

The idea is to find the bridge of two convex hull sequences quickly.

Crux of the solution – by Overmars and van Leeuwen

Once we find the bridge, the residual sequences must still be preserved and not discarded.

This is because we may need them after some points are deleted.



Dynamic divide and conquer.

The dynamic divide and conquer trick seems quite powerful. It allows us to convert static problems that can be solved with divide and conquer into dynamic problems with some overhead.

These class of problems are dubbed *decomposable* by Bentley and Saxe.

https://www.sciencedirect.com/science/article/pii/0196677480900152

Dynamic divide and conquer.

Consider finding the pair of points in the plane that are closest to each other.

There exists a standard $O(n \log(n))$ time divide and conquer algorithm.

Dynamic divide and conquer.

Consider finding the pair of points in the plane that are closest to each other.

There exists a standard O(n log(n)) time divide and conquer algorithm.

What happens if we apply dynamic divide and conquer trick to this problem?

Dynamic divide and conquer.

Consider finding the pair of points in the plane that are closest to each other.

There exists a standard O(n log(n)) time divide and conquer algorithm.

What happens if we apply dynamic divide and conquer trick to this problem?

We get a data structure that allows us to dynamically maintain the closest pair of points under point addition and deletion in logarithmic time.

Planar greedy matching

Consider a set of points on a 2D lattice. We wish to find a minimum weight matching of these points under some norm.

Planar greedy matching

Consider a set of points on a 2D lattice. We wish to find a minimum weight matching of these points under some norm.

A greedy approximation is to find a closest pair, match them and keep repeating until we don’t have any points left.

Planar greedy matching

Consider a set of points on a 2D lattice. We wish to find a minimum weight matching of these points under some norm.

A greedy approximation is to find a closest pair, match them and keep repeating until we don’t have any points left.

Trivial approaches are super-linear.

Planar greedy matching

Consider a set of points on a 2D lattice. We wish to find a minimum weight matching of these points under some norm.

A greedy approximation is to find a closest pair, match them and keep repeating until we don't have any points left.

Trivial approaches are super-linear.

We can use the dynamic closest pairs data structure to do the following :

Construct the data structure, find a closest pair, delete the two points from the set in logarithmic time, repeat. Total complexity is still $O(n \log n)$.

Planar greedy matching - WIP

Consider a set of points on a 2D lattice. We wish to find a minimum weight matching of these points under some norm.

A greedy approximation is to find a closest pair, match them and keep repeating until we don't have any points left.

I'm thinking of applying similar ideas to problems that arise in decoding topological surface codes in fault tolerant quantum computation.

Check out updates on my blog! - https://sumeetshirgure.github.io

Thank you for your attention!