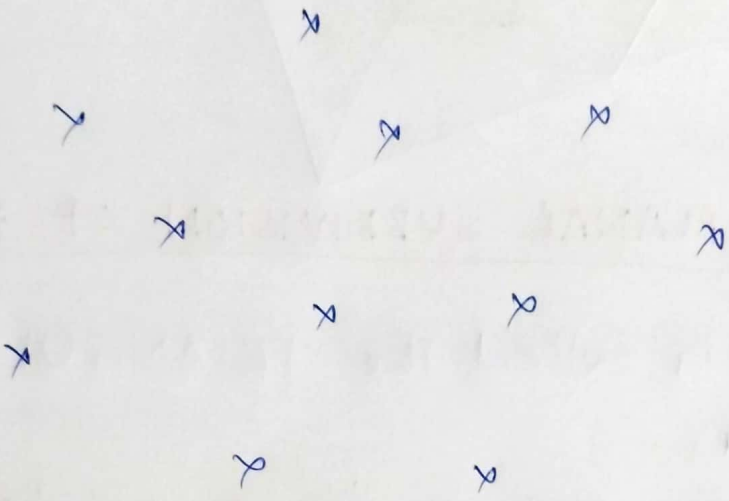
COMPUTATIONAL GEOMETRY

Lec 12

Lets consider a few points in 2-dimensional plane. Here, we take some arbitrary set of points for the sake of example.



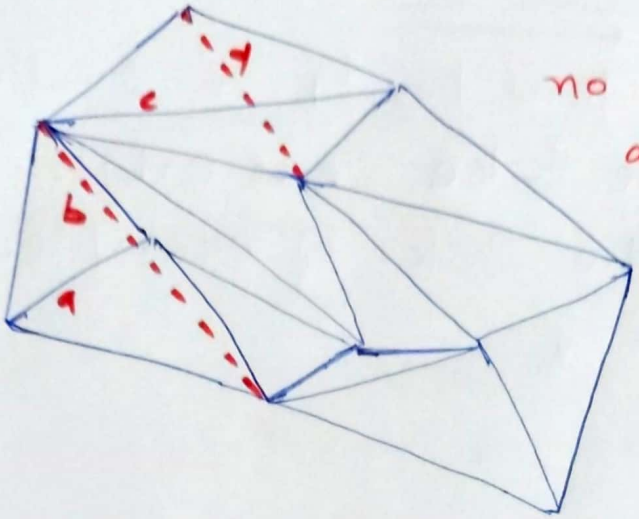
We can triangulate these points in many different ways.

Before going to triangulation, let us first look at the definition of triangulation. But before that, we need to define Maximal Planar Subdivision of P (some set of points)

Add as many edges among the vertices of the set of points P such that no two edges are intersecting, i.e., after adding all these edges, the graph obtained must be planar & we should add maximum

Scanned with CamScanner

number of edges while satisfying this condition.



no edge can be added without destroying the planarity

MAXIMUM PLANAR SUBDIVISION OF P

This is what is called the TRIANGULATION of these points.

Hence, triangulation of set of points P is nothing but the Maximum Planar Subdivision of P.

In fact, given a point set, it can be triangulated in many different ways

Eg:- remove a, add b
remove c, add d, etc.

TYPES OF TRIANGULATION

① Delaunay triangulation

② Greedy triangulation
(greedy strategy → add edges with smaller edge length)
Sort all possible edges b/w two distinct

Scanned with CamScanner

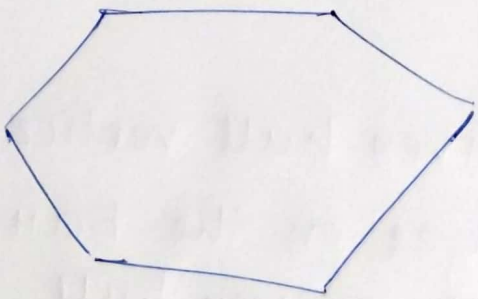
vertices based on edge length.
Iterate over the sorted list & add the current edge if it doesn't destroy planarity, else skip it.

(8) Minimum weighted Triangulation

triangulation which has minimum weight.
(people haven't been able to find any polynomial time algorithm nor have been able to show that it is NP complete)

Weight of a triangulation: Sum of all edge lengths in a triangulation is called weight of a triangulation.

Let us look at a special case of min weighted triangulation in which we have been given the vertices of a convex hull.

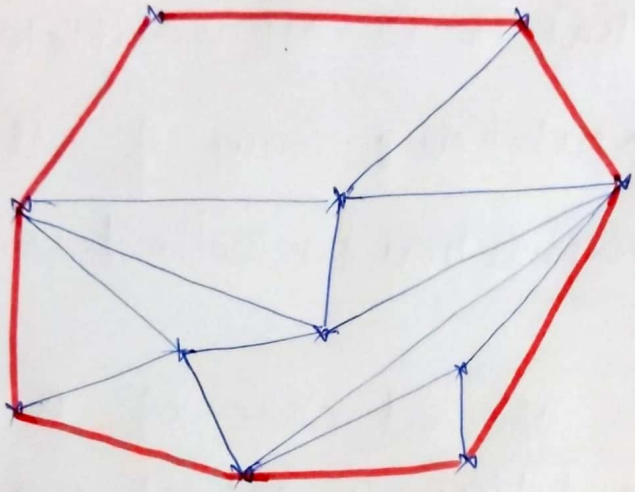


In this special case, min. weighted triangulation can be formed in polynomial time using Dynamic Programming.

But if the n points are in general position, whether we can find min. weighted triangulation in polynomial time is still an open question.

Scanned with CamScanner

NOTE: We can even bound the no. of faces & no. of edges in a triangulation. Whatever may be the triangulation, the boundary edges which are a part of the convex hull always present in all the triangulations of this point set.



Now, we look at the number of triangles that are there.

Let us assume

k = # convex hull vertices

= # edges on the boundary of convex hull.

No. of triangles

$$m = 2n - 2 - k$$

No. of edges

$$e = 3n - 3 - k$$

Scanned with CamScanner

These formulae can be easily obtained using Euler's formula & properties of triangulation.

no. of triangles

$n_f = m + 1$

(Every triangle represents one face & 1 is added to account for the outer face)

$\Delta \longrightarrow$ 3 edges

unbounded face has k edges (outer face)

no. of edges

$\frac{3m + k}{2}$

all edges except at the boundary are counted twice (for two triangles).

To make every edge be counted twice, we add k (number of boundary edges) to this number

$$n_v - n_e + n_f = 2$$

$$n - \frac{(3m+k)}{2} + m + 1 = 2$$

$$2n - (3m+k) + 2m = 2$$

$$\Rightarrow \boxed{m = 2n - k - 2}$$

$$\Rightarrow e = \frac{3(2n-2) - 2k}{2}$$

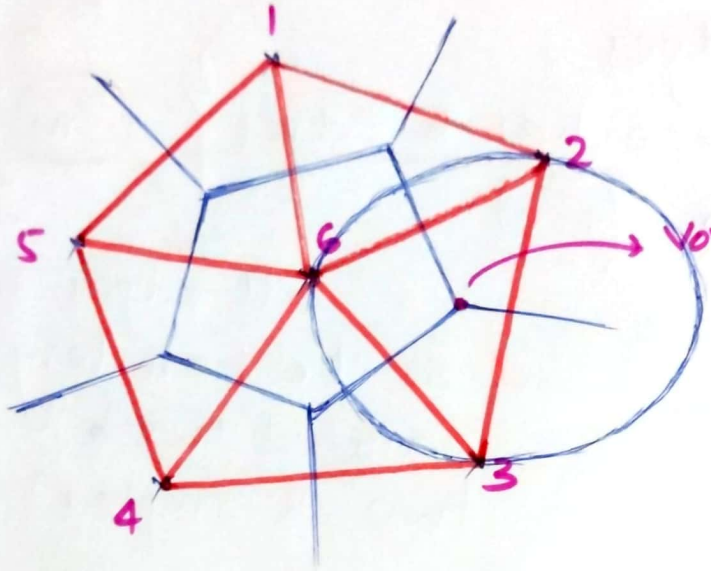
$$= 3(n-1) - k \quad \Rightarrow \boxed{e = 3n - 3 - k}$$

Scanned with CamScanner

Hence, any triangulation with k convex hull vertices has $(2n-2-k)$ triangles & $(3n-3-k)$ edges.

DELAUNAY TRIANGULATION

(a dual graph of voronoi diagram)



Points are in general position (no 4 points are cocircular)

The graph drawn in red lines is Delaunay Triangulation.

We can define Delaunay Triangulation as a dual graph of voronoi diagram or we can also define it directly from the point set (independently from voronoi diagram).

If you take any triangle, lets say $\Delta 236$.

If you consider a circle passing through these 3 points, there won't be any other point inside this circle.

Here we make the assumption that the points are in general position (No 4 points are cocircular).

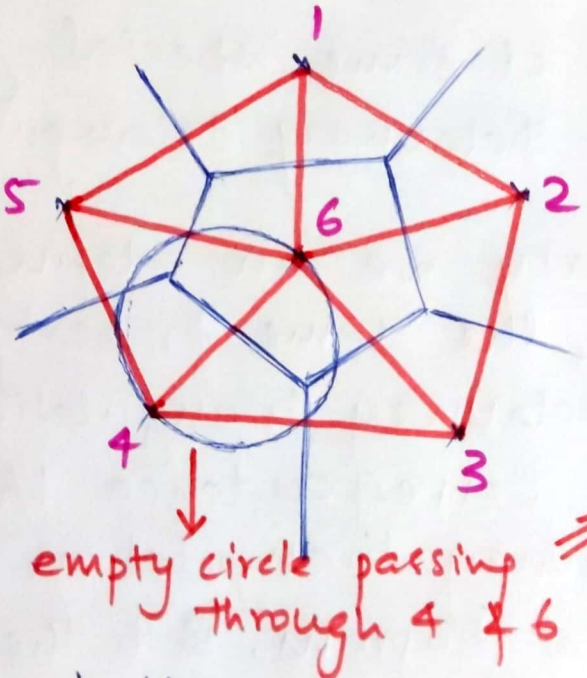
Scanned with CamScanner

$\Delta 236$ is in Delaunay Triangulation iff circle passing through 2, 3, 6 is empty (No other point from set P is inside the circle).

If any point from the point set lies inside the circle, the voronoi diagram will be different from its current form & the associated Delaunay triangulation will also change to a new form in which the above property will be satisfied by every triangle.

Now, let's talk about edges.

edge $\overline{v_i v_j} \in$ Delaunay Triangulation iff $\exists$ an empty circle passing through v_i & v_j.



NOTE: Voronoi diagram was defined independent of Delaunay Triangulation & Delaunay Triangulation was defined independent of voronoi diagram. Both are defined in two different time frames

Scanned with CamScanner

independent of each other.
Later, researchers found the relation b/w these 2.

People generally compute Delaunay Triangulation instead of voronoi diagram because for computing voronoi diagram, we have to compute voronoi vertices. So when we do these calculations in floating point arithmetic, some rounding off & truncation errors take place. So, to avoid these floating point errors, computing Delaunay triangulation is preferred instead of voronoi diagram.
In Delaunay triangulation, we are adding edges b/w existing vertices. No new vertices are created. only thing to be decided is which edge is to be added.
Now let us look at some special graphs with respect to Delaunay triangulation.

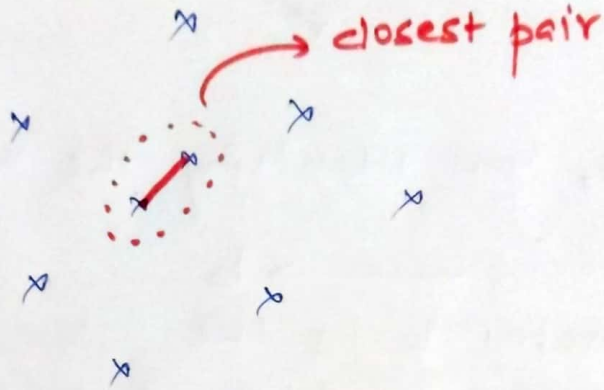
Whatever property we can define in a voronoi diagram, the same property can be defined in a Delaunay triangulation. There is a one to one correspondence b/w the properties of Delaunay triangulation & properties of voronoi diagram. Both the graphs are dual of the other one.
Voronoi diagram is defined in terms of the neighbourhood of some site.

Scanned with CamScanner

Delaunay triangulation & voronoi diagram capture a lot of neighbourhood properties.

for eg:- Closest Pair (CP)

Given n points in a plane, which two points are the closest to each other



In case of multiple pairs having the same minimum distance b/w them, all of them become the closest pairs.

NOTE:- CLOSEST PAIR MUST BE PART OF THE DELAUNAY TRIANGULATION

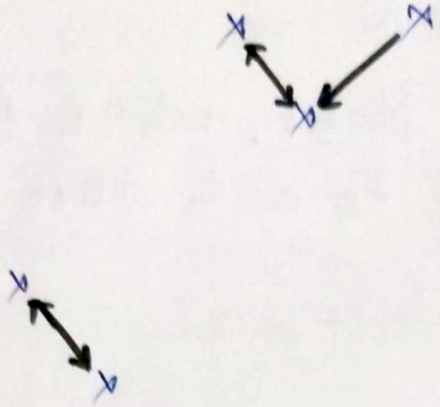
Approaches to finding the closest pair

(1) O(n log n) DIVIDE & CONQUER approach, discussed in CLRS

(2) Compute Delaunay triangulation. Then check every edge of Delaunay triangulation for closest pair.

Scanned with CamScanner

Nearest Neighbourhood Graph (NNG)



DIRECTED GRAPH OF NEAREST NEIGHBOURS

for each vertex, you identify the nearest neighbour.

$a \longrightarrow b$

denotes b is the nearest neighbour of a.

Closest Pair $\subseteq$ is a subgraph of Nearest Neighbourhood Graph

Nearest Neighbourhood Graph is a subgraph of $\subseteq$ Delaunay Triangulation

Euclidean Minimum Spanning Tree (EMST)

Consider n points in a plane.

Scanned with CamScanner

weight of each edge is the distance b/w its endpoints.

Using these edge weights, find the minimum Spanning Tree.

Euclidean Minimum Spanning Tree $\subseteq$ Delaunay Triangulation.

is a subgraph of

$$CP \subseteq NNG \subseteq EMST \subseteq GG \subseteq RN \subseteq DT$$

Connected graph

Gabriel Graph

Relative Neighbourhood Graph

All these graphs capture some neighbourhood properties (closeness). All these things are part of Delaunay Triangulation.

$EMST \subseteq DT$

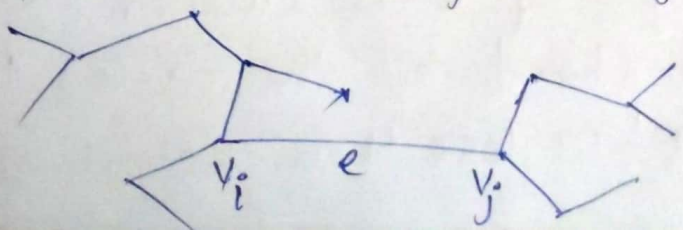
Proof: We will prove this using Proof by Contradiction technique.

Let us assume this is not true.

$\Rightarrow$ $\exists$ edge $e \in EMST$ but $\notin DT$

Remove this edge from EMST

Let v_i & v_j be the endpoints of edge e



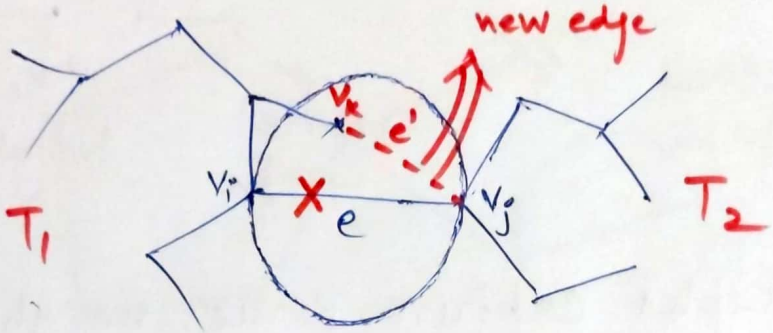
Scanned with CamScanner

When we remove e from EMST, the EMST will become disconnected

Edge e does not belong to Delaunay Triangulation.

⇒ There is no empty circle passing through v_i & v_j (or else e will be a part of DT)

Construct a circle with $v_i v_j$ as a diameter.
This circle will be non-empty.
There exists some point inside this circle.



That point will either be connected to left subtree T_1 (on the side of v_i) or to the right subtree T_2 (on the side of v_j). Here, WLOG we assume it to be connected to T_1.
Let that point be v_k.
If we remove e & introduce an edge b/w v_k & v_j

$$T' = T - e \cup \{e'\}$$

Now let us look at the weight of T'
Since the length of e' is smaller (Diameter is the largest chord of the circle) than e, weight of T' is less than weight of T.

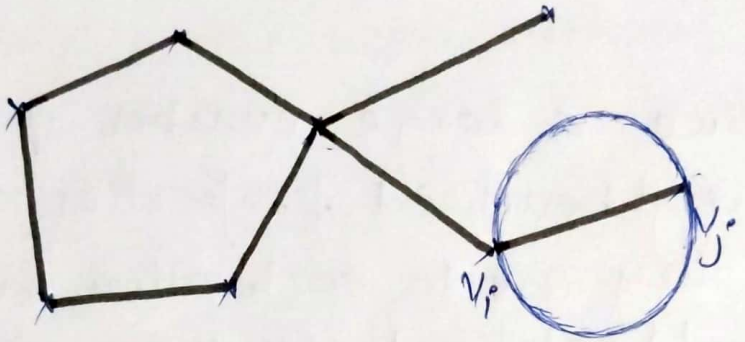
Scanned with CamScanner

This contradicts our initial assumption that T is an Euclidean Minimum Spanning Tree. Hence, e must be a part of DT.

Hence, EMST $\subseteq$ DT

Proved by contradiction

Similarly, we can also define other graphs such as Gabriel Graph (GG)



Given n points, we add an edge b/w 2 points if the circle drawn taking the line segment joining the 2 points as diameter is empty.

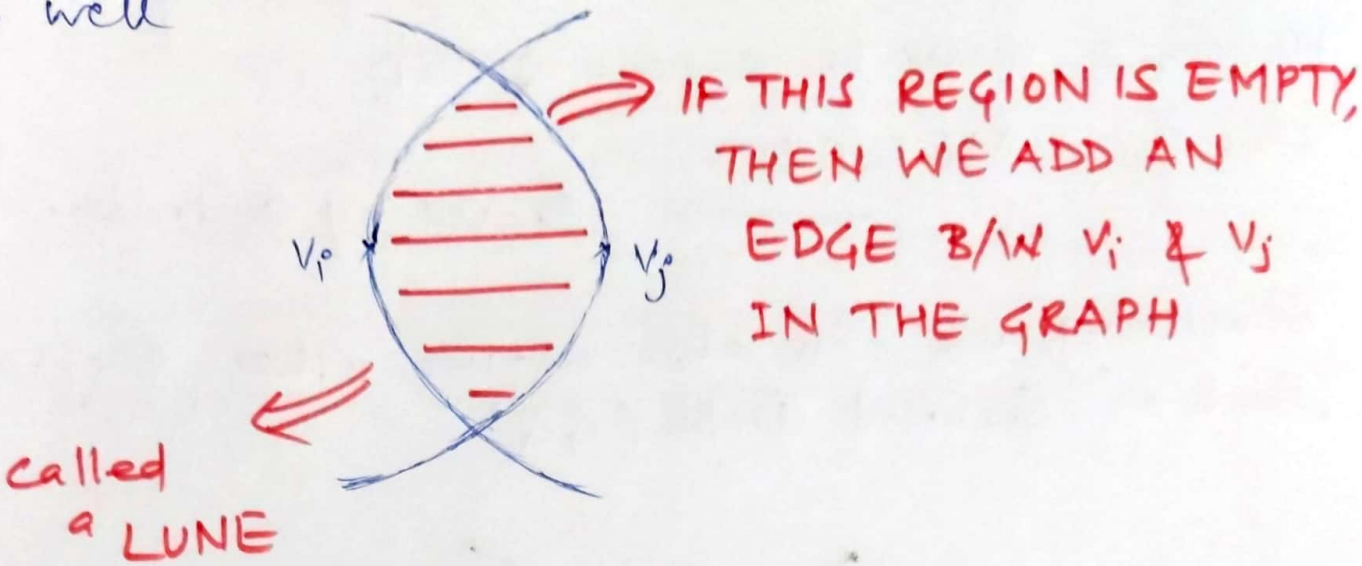
Relative Neighbourhood Graph (RNG)

In the case of a Gabriel graph, we are defining the neighbourhood of 2 points as a circle with line segment joining the two points as diameter.

In relative neighbourhood graph, we choose 1 point as center & draw a circular arc passing through the other point with the line segment joining them as radius.

Scanned with CamScanner

Similarly, we do this for the other point as well



We can define a large number of graphs based on neighbourhood properties of vertices (how close they are to each other by some neighbourhood relation). All these graphs are part of Delaunay Triangulation.

Many other graphs are there.

Different parameterized graphs are there. You can have different parameters you'll get a different graph. Infinitely many no. of graphs can be defined for each value of the parameter. These are called β-skeleton & α-graphs.

β-skeleton has <u>two definitions</u>

- CIRCLE BASED
- LUNE BASED

Scanned with CamScanner

All these graphs have relation with DT. For some values of β and some values of α, these graphs are subgraphs of DT.
All these graphs are defined in terms of a neighbourhood relation. You take some neighbourhood region b/w two vertices. If that region is empty, add that edge in the graph.
How you define the neighbourhood changes these various graphs.
Large number of graphs are there which are a part of DT & capture nice neighbourhood properties. These graphs are used in many applications like Face Recognition, Pattern Recognition, Shape Analysis. This is why DT is a very powerful object to consider in various neighbourhood properties in image processing & analysis.

Scanned with CamScanner

COMPUTATIONAL GEOMETRY

Lec 13

If we look at a map, it is a 2D plane where countries are divided into states & states are divided into districts, so on & so forth. There are multiple regions in the map with clearly defined boundaries. We need to identify in which region a location is present. Once we identify the region, all the attributes of the region are applicable to the location also.

So, here the key problem to solve is Given x, y coordinates (defining a location), identify in which region this coordinate lies. Once we identify this, we can get useful information like address etc.

this is called **POINT LOCATION**.

A formal way of stating the problem is

"Locate the point within a given planar subdivision".

Plane $\xrightarrow[\text{into}]{\text{divided}}$ Regions (Planar subdivision)

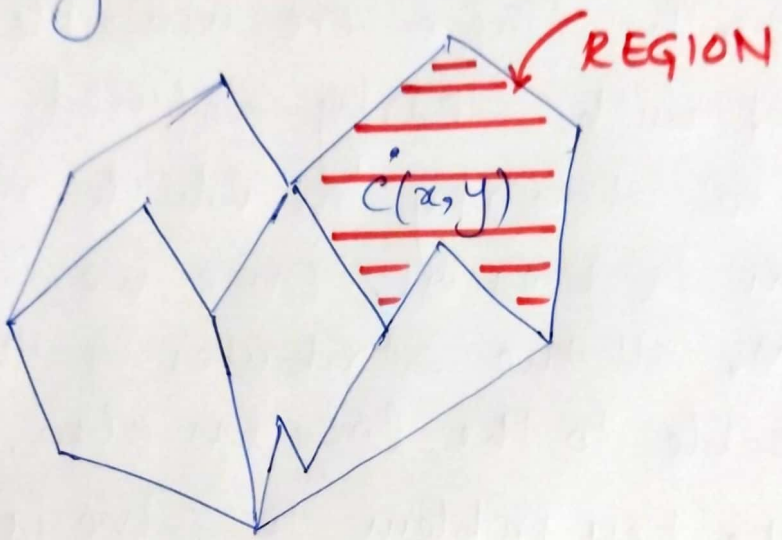
Location → find the region in which this location lies.

Scanned with CamScanner

POINT LOCATION

We will see how this can be done geometrically.

The state boundaries & district boundaries are curved lines but for simplicity, we take straight line boundaries here



We took an arbitrary diagram.

The plane is divided into regions. These regions are defined by straight line edges. We can approximate curved boundaries using straight line edges.

We have been given the above planar subdivision & a point by means of x & y coordinates.

We need to identify that the point C belongs to the region R.

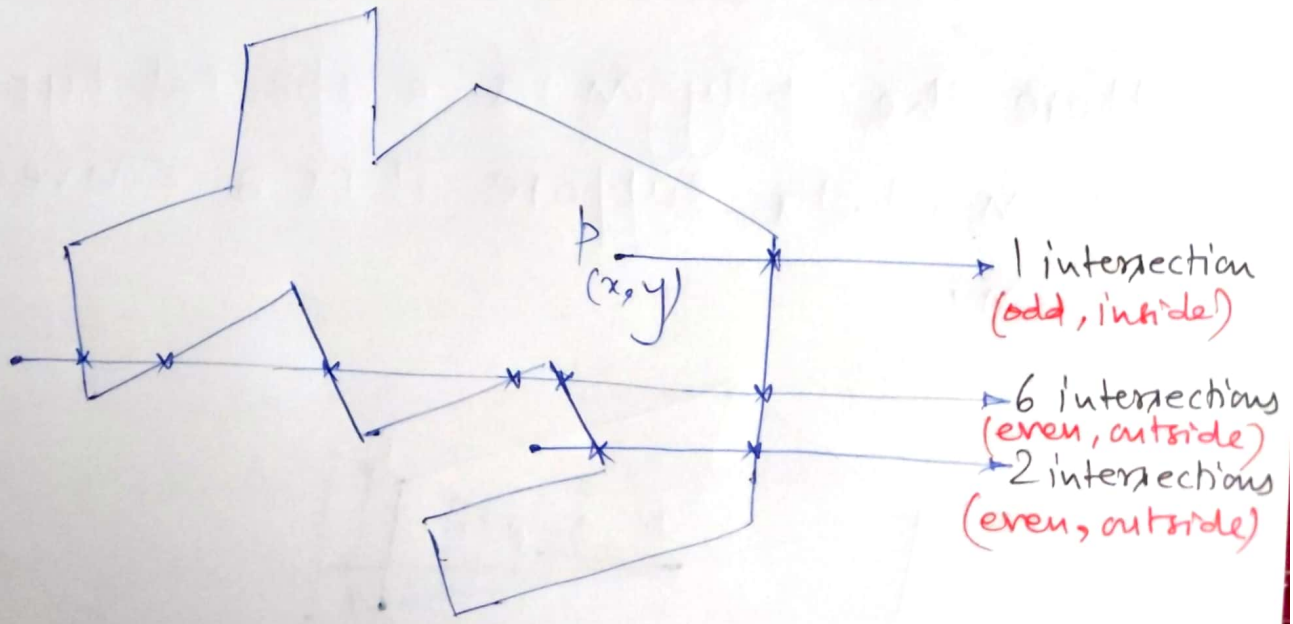
In case of Voronoi diagram also, when we needed to find which resource center is the closest resource center, there also,

Scanned with CamScanner

we have to identify the region in which this point lies. When we find the region, we find the resource center associated to that region & that will be the closest one for the given point.

So, in the case of voronoi diagram also, we have to look at the point to identify the closest resource center. Point location is a very crucial problem in many applications In this course, we address this using a geometric approach.

We will start by taking a simple polygon initially instead of a complex planar subdivision (previous diagram).



Scanned with CamScanner

How can we identify if the given query point is inside or outside the polygon.

We do this by starting a horizontal ray in the +ve X dir^n from that point & then check its intersection with each edge.

This is the standard **RAY CASTING ALGORITHM**.

We have discussed this previously in the course.

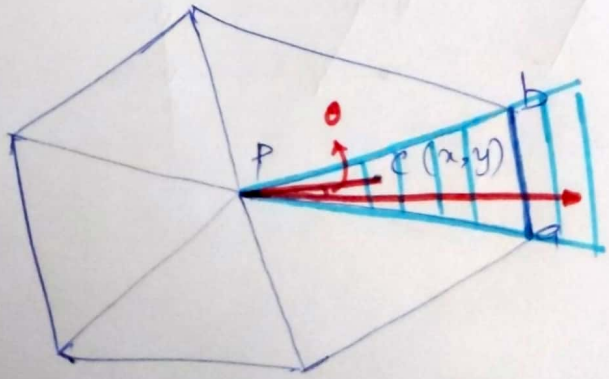
If number of intersections is odd, the point is inside the polygon.

If number of intersections are even, the point is outside the polygon.

Time complexity of this algorithm is

$O(n)$ ← **LINEAR**

Suppose the polygon is a special type of polygon. Let's suppose it is a convex polygon.



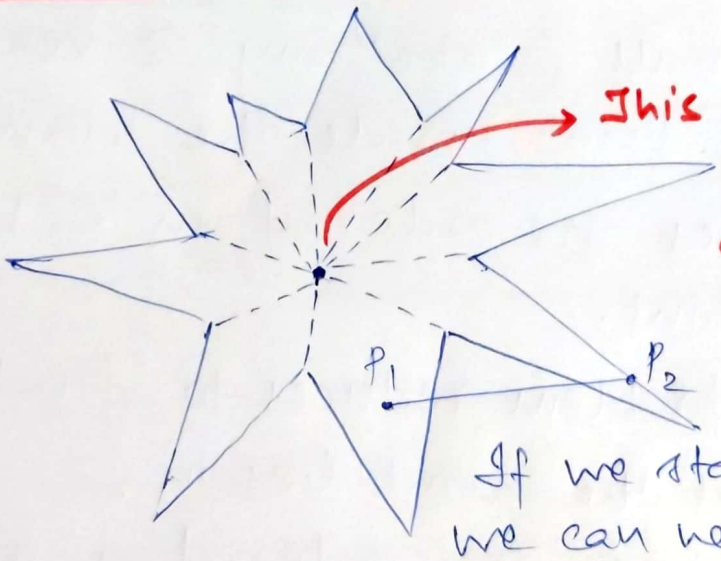
Scanned with CamScanner

In this case too, we can use the RAY CASTING ALGORITHM to find whether the point is inside or outside in O(n) time complexity.
But, can we do anything better than O(n).
Here, we need to use some special property of convex polygons.
We have the vertices of the convex hull of these points.
Let us suppose that the vertices are given in COUNTER CLOCKWISE order.
We take an arbitrary point p inside the convex hull (take any 3 vertices & take any point inside the triangle).
The vertices are ordered in CCW dirn wrt this point.
Now, the problem reduces to finding the sector that the point lies in.
All the vertices are ordered angularly wrt the inside point.
Given a query point, find angle θ that the line joining it to the inside point makes with the +ve X axis.
Search for this angle among the vertices (Lower bound & upper bound) using Binary Search.

Scanned with CamScanner

This will give you two consecutive points a & b b/w which this angle lies.
This will give you the sector in which the point lies in logarithmic time ($O(\log n)$).
Now, once we identify the sector, we can tell if the point lies inside the polygon or not in constant time (check if point lies inside $\triangle Pab$ or not).
Hence, overall complexity is $O(\log n)$.

STAR-SHAPED POLYGON



If we stand at P_1, we can never see P_2 b'coz the wall will obstruct its field of vision.

Mathematically, it can be defined as follows. If you can find one point inside the polygon such that you can see the entire boundary of the polygon from this point, then the polygon is called a

Scanned with CamScanner

"STAR SHAPED POLYGON."

If we make a complete 360° rotation around that point, we can see the entire boundary. If we knew this point, we can find out if a point is inside the polygon or not in logarithmic time.

We can find this point in O(n lg n) TIME COMPLEXITY using something similar to convex hull problem namely.

"Half Planes Intersection problem"

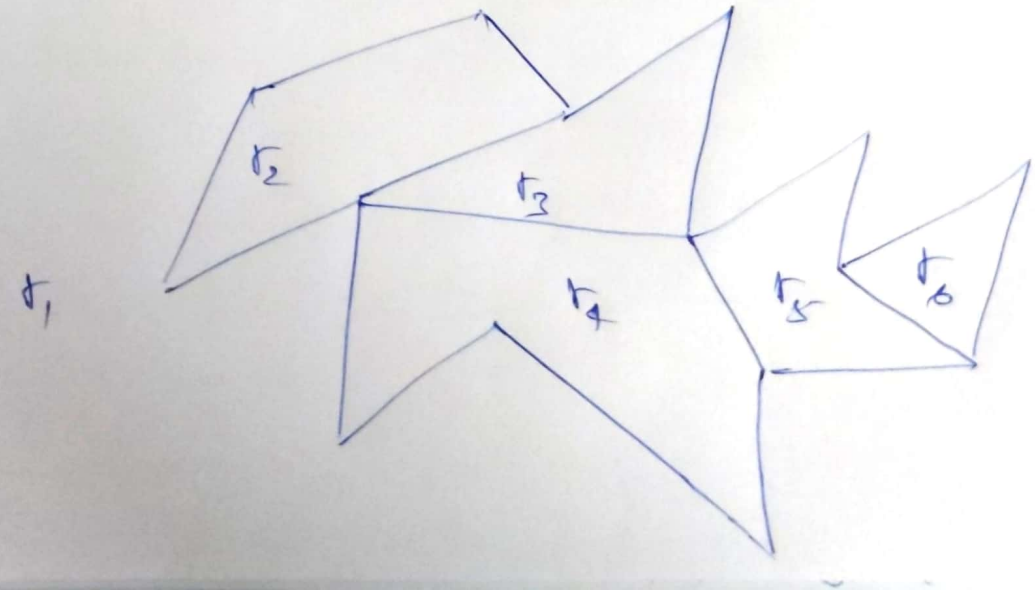
⇓

If we solve this problem, we can find all points inside the polygon such that entire boundary is visible from them.

This is known as "kernel of the polygon"

Coming back to our original problem of planar st. line partitioning.

Let us take one more example

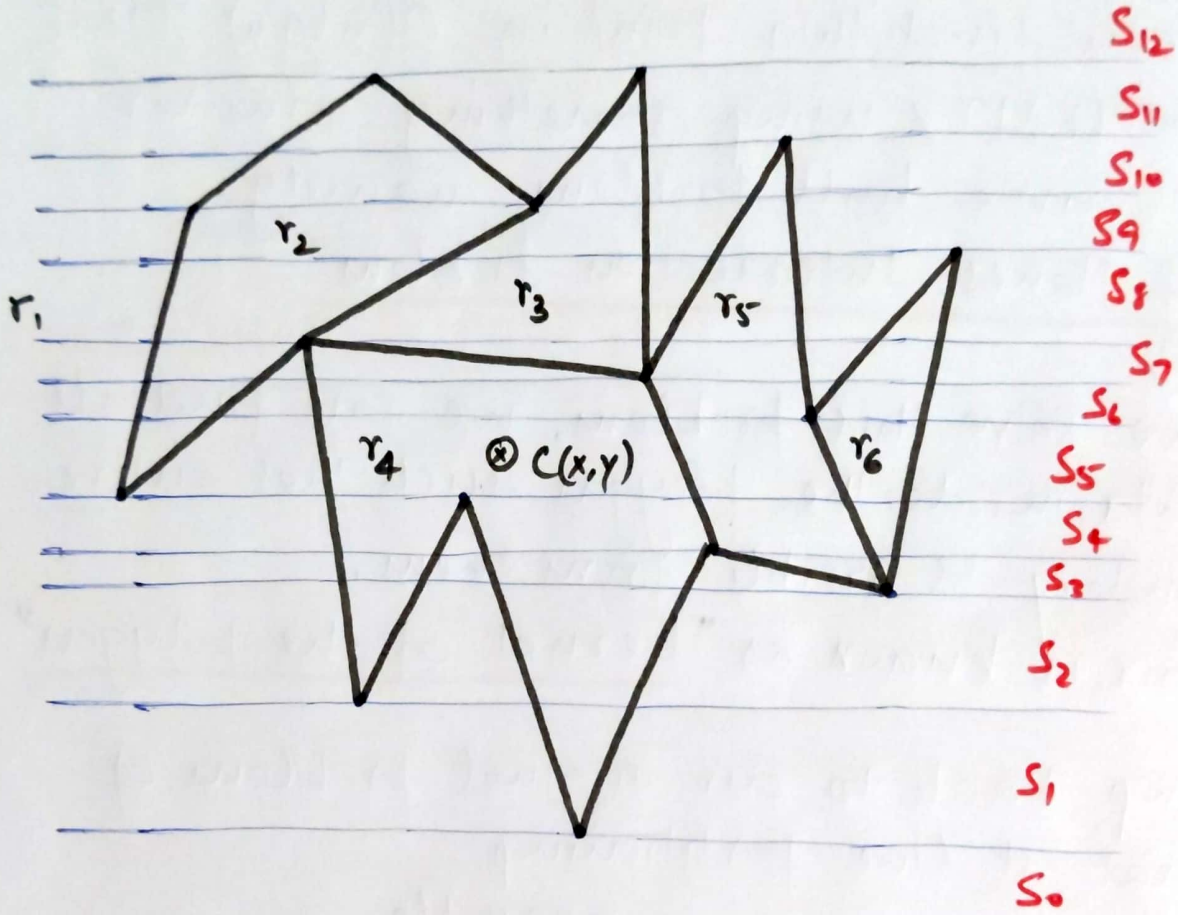


Scanned with CamScanner

Given a point, we should be able to tell that the point belongs to which region. This can be solved by an easy method named SLAB METHOD.

SLAB METHOD

Draw a horizontal line through each vertex.



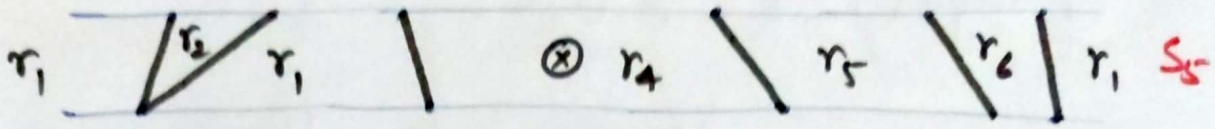
Given a query point (C(x, y)), how ~~quic~~ quickly can we identify that the query point belongs to which slab (S_5 in this instance)

We can do this in O(log n) using Binary Search. We binary search the y coordinate of query point in the y coordinate of slabs (topmost / bottommost).

Scanned with CamScanner

Once we have the slab, how can we identify the region in which the point lies?

We draw the slab separately here.



Here also, we can apply Binary Search.

We can tell whether the point is to the left or to the right with respect to every edge.

This is what is required for Binary Search.

Using binary search, we can find two edges b/w which the point lies in logarithmic time.

Using the two edges, we can tell the region in which the point lies in $O(1)$ time.

Hence, using Slab method, we can determine the region in which point lies in $O(\log n)$

LOGARITHMIC TIME

$Q : O(\log n)$

To get the slabs & all the information, we have to preprocess this planar straight line ~~graph~~ embedded graph into slabs & we have to store information about each slab.

$P : O(n \log n)$

(using Sweep Line algorithm)

Scanned with CamScanner

Here, we have a planar straight line embedded graph

- no. of vertices (V) : n
- no. of edges (e) : $O(n)$ — because it is a planar graph
- no. of faces (f) : $O(n)$

→ ALL ARE LINEAR IN n

Complexity of planar straight line embedded graph

$V + e + f = O(n)$

$S : O(n^2)$

↳ n vertices, we draw a horizontal line through each. Therefore, a maximum of $n+1$ slabs can be there. In each slab, $O(n)$ edges can be present in the worst case.

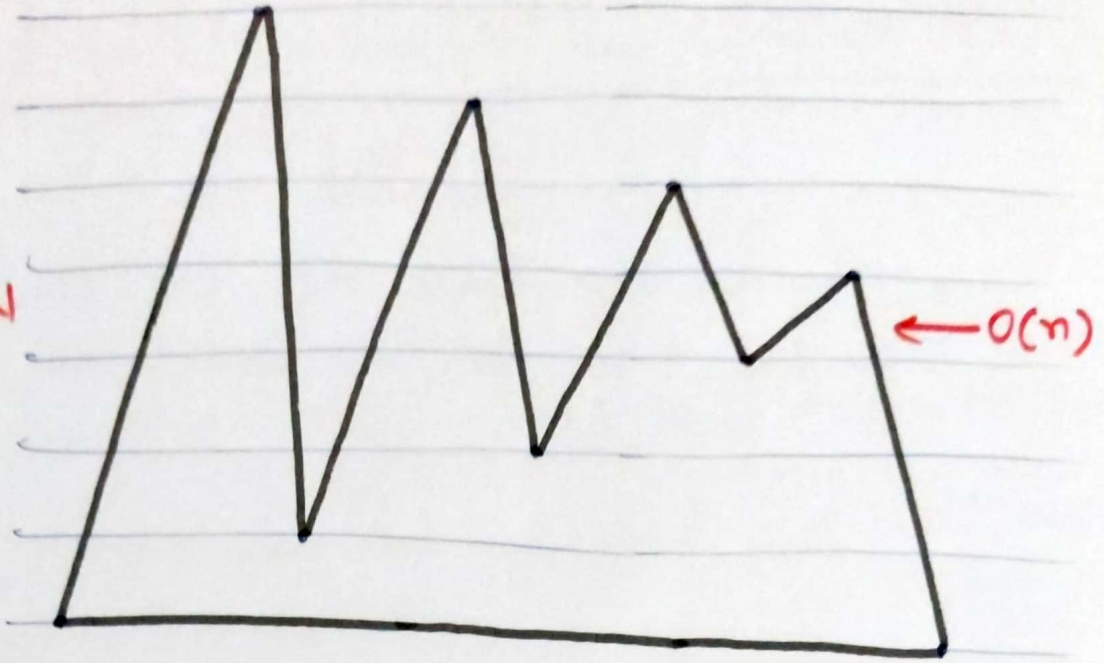
So, total complexity of all slabs together = $O(n^2)$

Hence, $S = O(n^2)$.

Is this a tight upper bound or a crude upper bound?

Can we find an example of a planar straight line embedded graph where $S = O(n^2)$?

Scanned with CamScanner



For this algorithm,

Query Time is very good
Preprocessing Time is also very good.

(for applying Binary Search in 1D, we had to sort the numbers initially taking $O(n \log n)$ time which is same as here. Hence, we cannot do better than this $O(n \log n)$ preprocessing time)

The only issue is $O(n^2)$ space

That is the crucial part

There are various methods to improve on this. But in terms of query time & preprocessing time, it is the best method.

Scanned with CamScanner

COMPUTATIONAL GEOMETRY

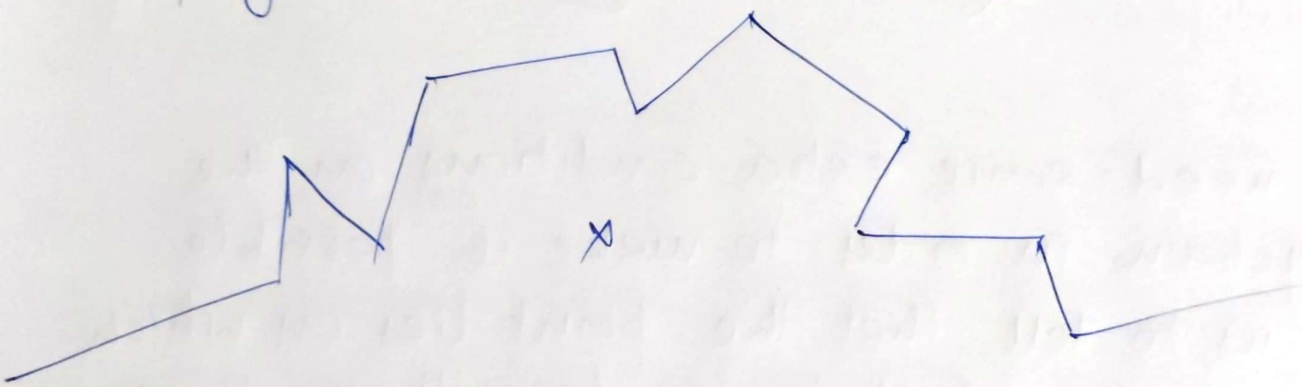
Lec 14

In the last class, we discussed about the SLAB METHOD. In this class, we will discuss another thing called the CHAIN METHOD.

Without giving any background, we will move directly to the solution.

Suppose if we have a chain.

Chain can be anything, for eg:- polygonal chain or a polyline.



Given a query point.

How quickly can we say that this point is on which side of the chain.

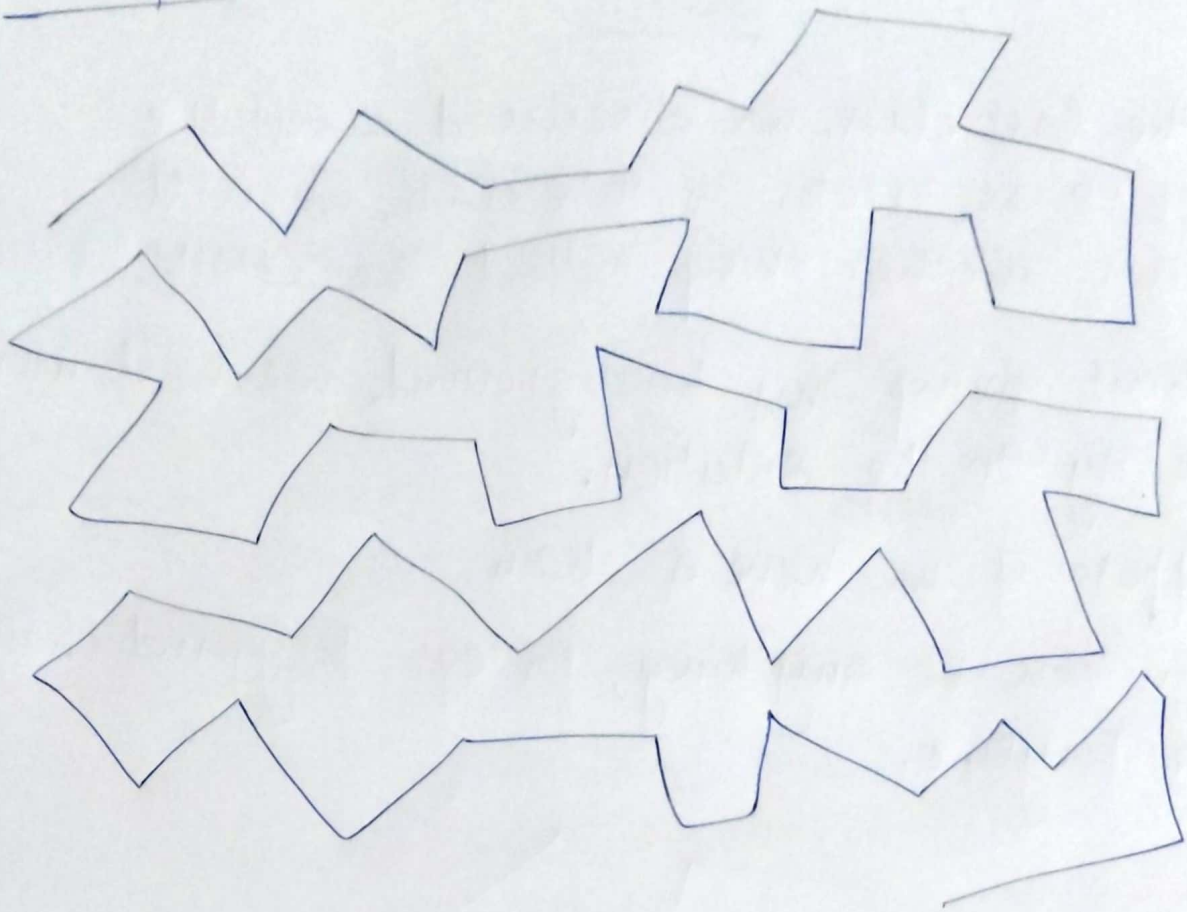
If we extend its last edges to infinity, then this polyline/chain divides the plane into 2 halves.

Here, we only mentioned a polygonal chain. There isn't any specific property of this chain.

Scanned with CamScanner

It can be arbitrary.

For eg:-



We need some extra conditions on the polychain in order to make it possible for us to tell that the point lies on which side of the polychain in logarithmic time using Binary Search.

For that, we take up "MONOTONE CHAIN."

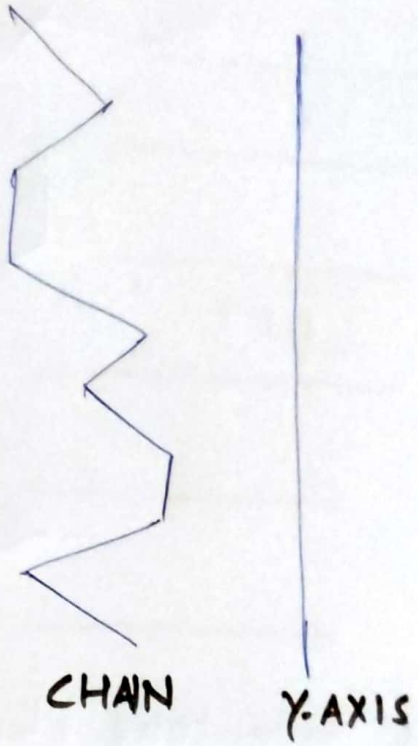
MONOTONE CHAIN

Here, we take chains monotone wrt y axis.

The figure of a y-monotone chain is given on the next page

Scanned with CamScanner

This chain is y monotone because if we take a line orthogonal to y-axis (parallel to x-axis), there is only 1 intersection with this chain. Of course, last edge is extended to infinity on both ends.



Such a chain is called monotone chain wrt y axis.

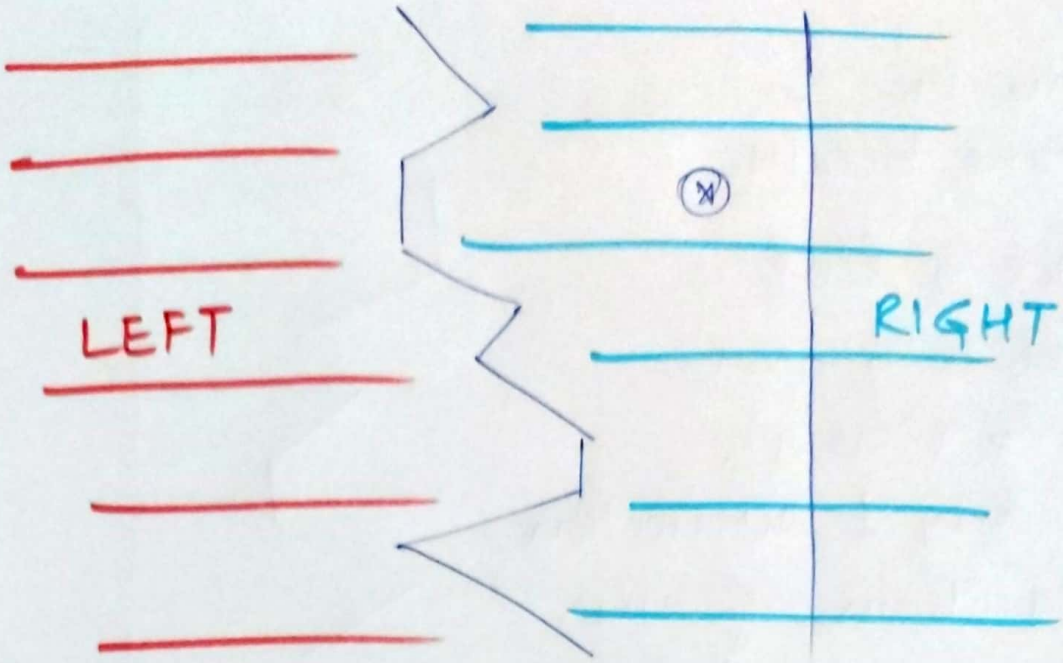
Of course, we can have any line, need not be y axis. We can have an arbitrary line & define a monotone chain wrt that line. Let the arbitrary line be l.

If we take any line orthogonal to l & it has only 1 intersection with the chain, such a chain is called a monotone chain wrt line l.

For simplicity & convenience, here we take line l to be y-axis. It can be any line.

Aim: If a y-monotone chain & a query point is given to us, we need to find whether the point lies to the left or to the right of the chain.

Scanned with CamScanner



This monotone chain divides the plane into 2 halves & is monotone wrt y axis. So, we can define left & right sides of this chain.

Hence, the problem reduces to how quickly can we tell whether the query point is to the left or to the right of the chain.

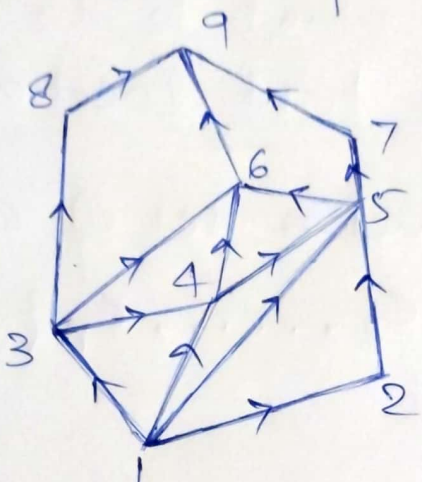
This can be done in $O(\log p)$ time where p is the number of line segments in chain.

Every line segment has an upper & lower endpoint. The line segments can be sorted based on their upper/lower endpoints & then we can use binary search to find the line segment which intersects with the horizontal line passing through query point. Let the y coordinate of query point be y'. Then we need to compute the x coordinate of

Scanned with CamScanner

the line at y' & compare it with x-coordinate of query point to tell whether point lies to the left or to the right.

Now, how can we extend this to some arbitrary planar straight line graph. Lets consider an example.



This is a special type of graph.

We order the vertices from bottom to top. If two points have same y-coordinate, we can order them from left to right or right to left. In our example, all points have distinct y coordinates.

Each edge is going from lower index to higher index. We give a direction to each edge.

For every point except the first & last points, we have incoming edges as well as outgoing edges.

Scanned with CamScanner

Such a graph is called regular planar straight line graph.

Every planar st. line graph which follows
Every vertex except first & last has incoming & outgoing edges
is called regular planar st. line graph.

On this graph, we want to do chain method of planar point location.

→ We want to find a set of chains

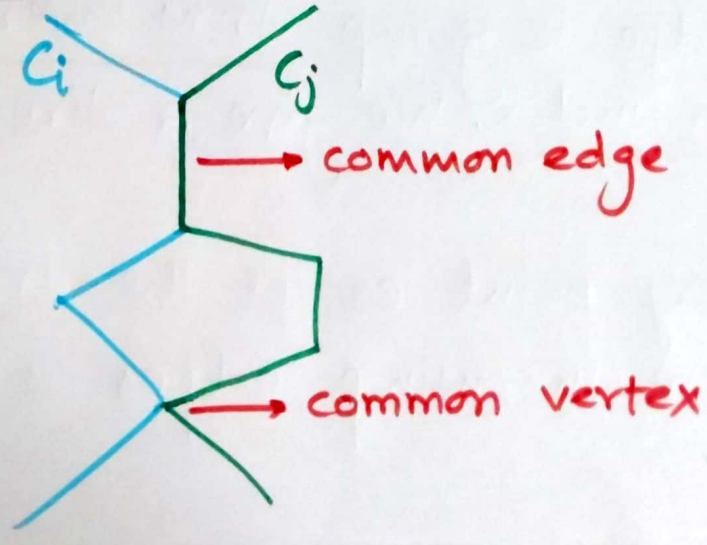
$$\{C_1, C_2, \ldots, C_r\}$$

such that

① $\bigcup_{i=1}^{r} C_i = G$ ← given graph

② Take any 2 chains C_i & C_j.
One chain should be completely on the right side of another.
No crossing b/w chains is allowed.

For eg:-

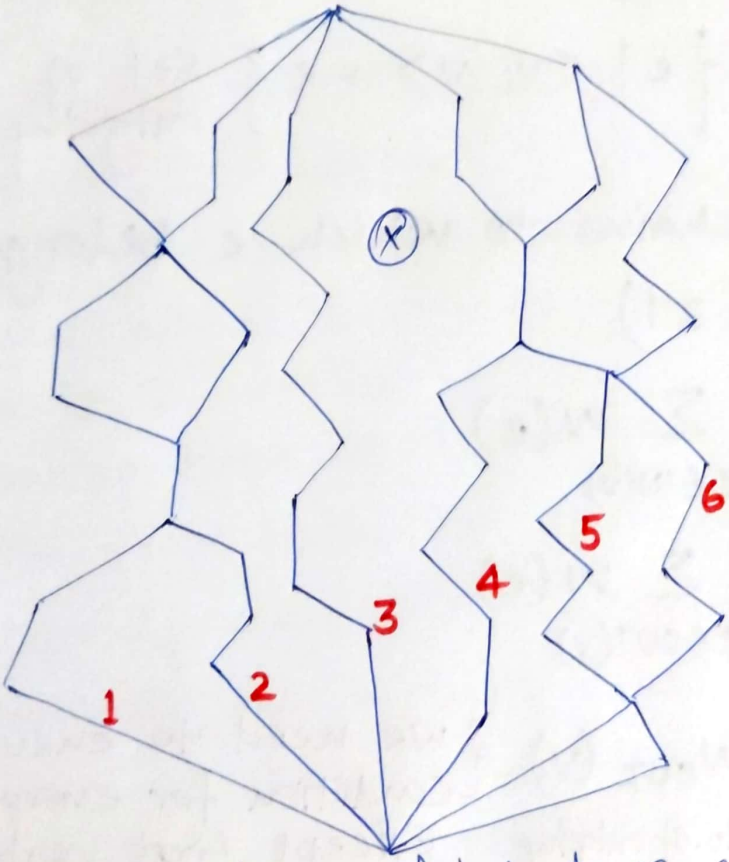


Scanned with CamScanner

This means that every edge of the graph must be a part of some chain.

If we have such a set of chains then we can use 2 level Binary Search to determine location of query point.

NOTE:- The chains start & end at starting (first vertex) & ending (last vertex) points respectively.



We identify b/w which two chains does the query point lie.

We can do this in $O(\log p \log r)$ time complexity using Binary Search.

where p = maximum no. of line segments in a chain

r = no. of chains

Scanned with CamScanner

Order these chains from left to right & then use Binary Search to determine location of point wrt middle chain of current interval & proceed accordingly.

Now the question is, how to find these chains.

For that, we use some notations,

$$IN(V_i) = \{e \mid \langle v\ V_i \rangle = e\}$$ Set of incoming edges

$$OUT(V_i) = \{e \mid \langle V_i\ v \rangle = e\}$$ Set of outgoing edges

$W(e)$ = # chains to which e belongs.

$(W(e) \geq 1)$

$$W_{IN}(V) = \sum_{e \in IN(V)} W(e)$$

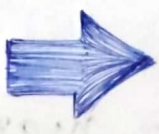
$$W_{OUT}(V) = \sum_{e \in OUT(V)} W(e)$$

$W_{IN}(V) = W_{OUT}(V)$ { we need to ensure this condition for every vertex except first vertex & last vertex }

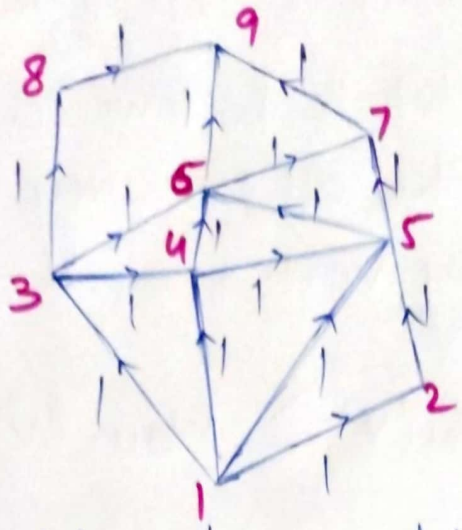
No. of incoming chains

No. of outgoing chains

DIAGRAM ON NEXT PAGE



Scanned with CamScanner



Firstly, we will assign weight of each edge as 1. Then, we will scan the bottom vertex & gradually move to the top.

- We simply ignore the 1st vertex
- Move to 2nd vertex.
- Incoming edge wt = 1 = outgoing edge wt
- So, no need to do anything.
- Move to 3rd vertex
- Incoming edge wt = 1, outgoing edge wt = 3

In the first pass, we ignore cases where Incoming edge wt < outgoing edge wt. We simply make sure that

$$W_{IN}(v) \leq W_{OUT}(v)$$

in the first pass.

- Move to vertex 4
- 2 incoming 2 outgoing
- Move to vertex 5
- 3 incoming edges, 2 outgoing edges Condition not satisfied.

Add $W_{IN}(v) - W_{OUT}(v)$ to leftmost outgoing edge

Scanned with CamScanner

- Move to next vertex & repeat the procedure.

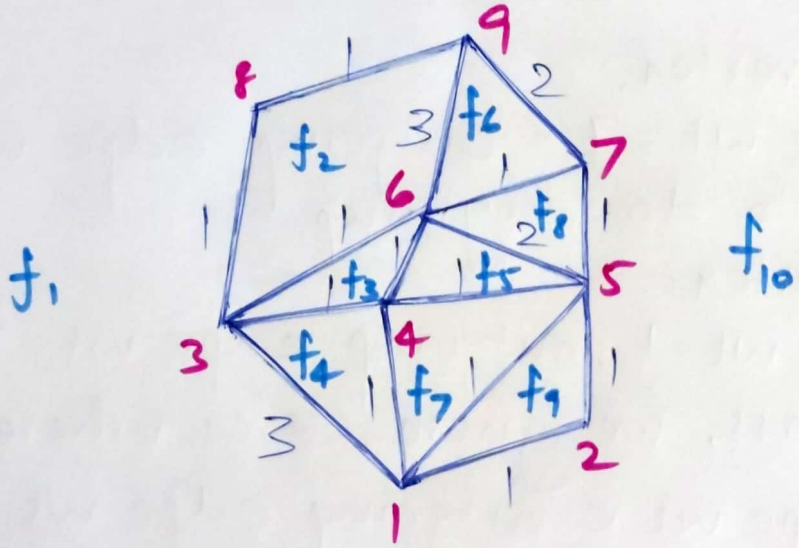
Now move from top to bottom in 2nd pass & try to balance the other way around for each & every vertex excluding first & last one.

$$W_{IN}(v) \geq W_{OUT}(v)$$

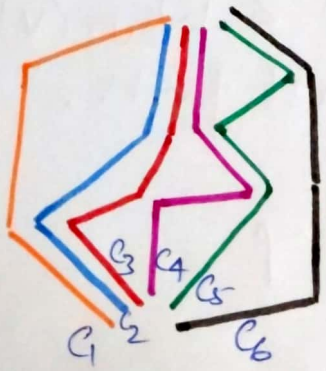
After pass 2,

$$W_{IN}(v) = W_{OUT}(v)$$

FINAL RESULT

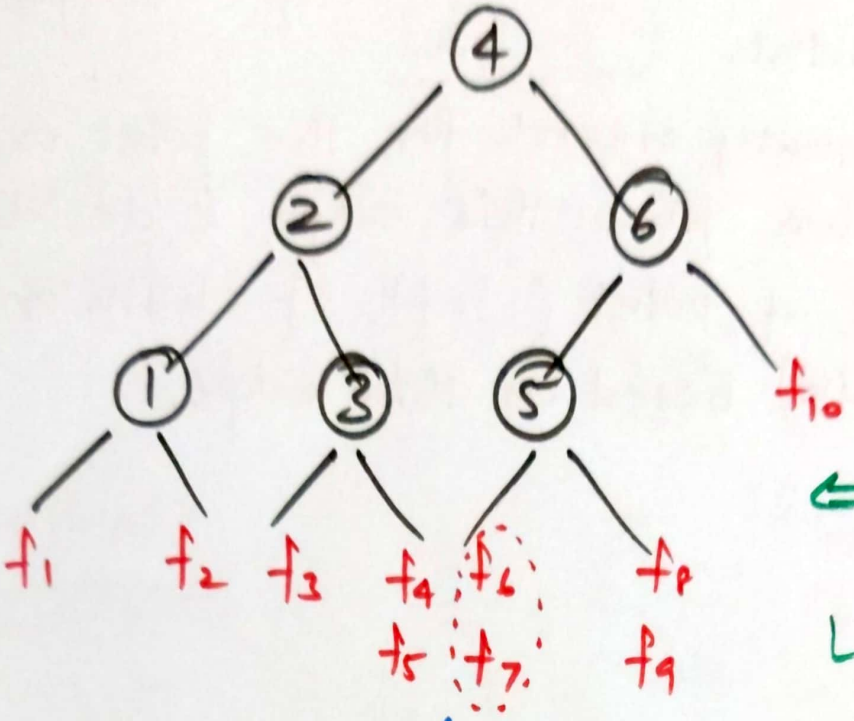


CHAINS



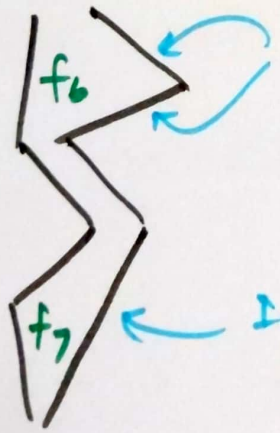
Scanned with CamScanner

BST of chains



⇐ USE THIS TO FIND LOCATION OF A POINT

can be distinguished based on the edge that we used to determine the location of the point



if this was the deciding edge then f_6

If this edge was the deciding edge then f_7

Scanned with CamScanner

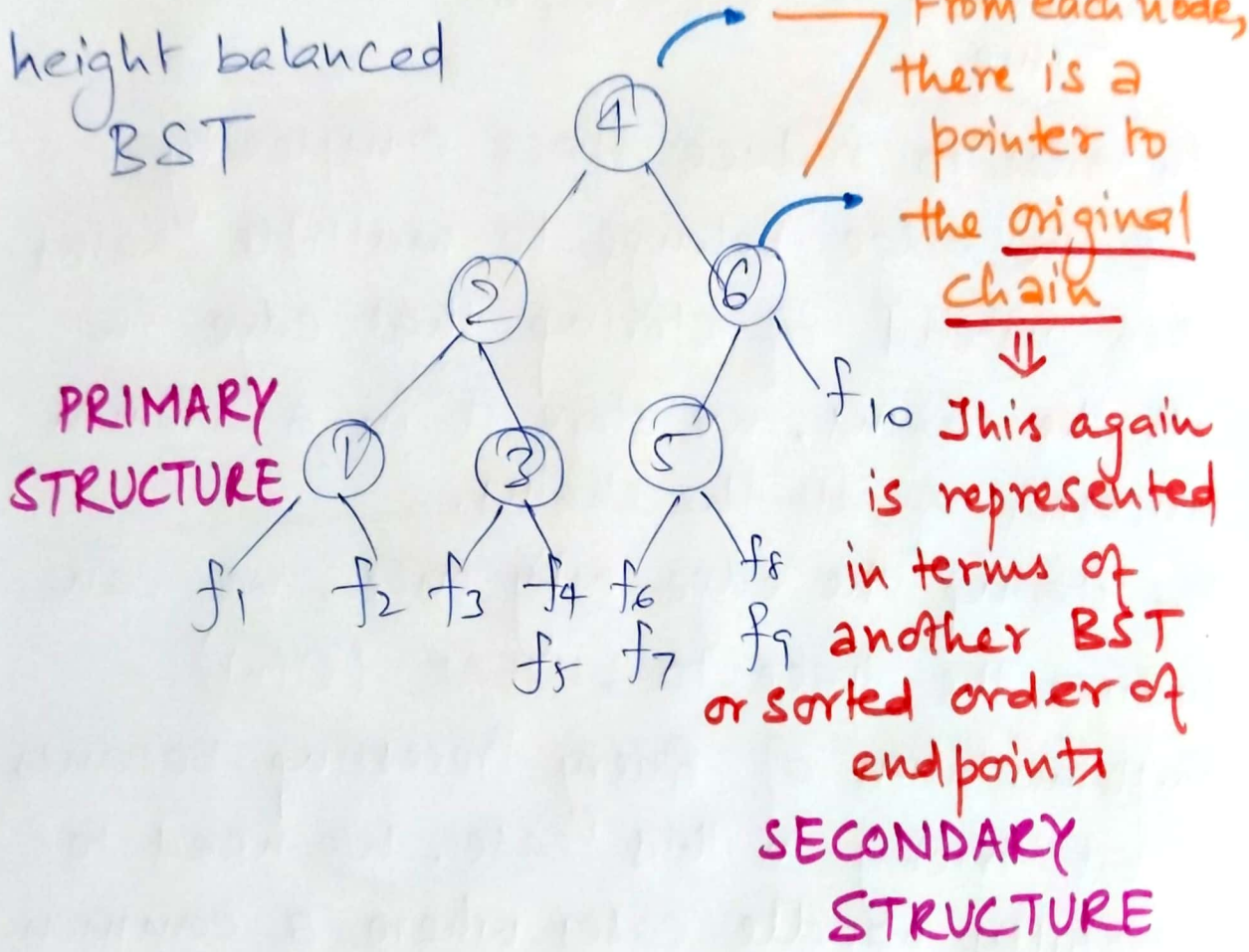
Deciding edge is the edge that intersects with the horizontal line passing through the query point.

When we binary search for the point on the chain, we find this edge & decide the location of point (left of chain or right of chain) based on this edge.

Scanned with CamScanner

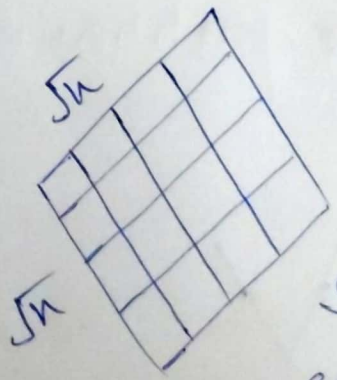
COMPUTATIONAL GEOMETRY

Lec 15



In worst case, p & r can be $O(n)$.

WORST CASE EXAMPLE



GRID

In this case, there are $\frac{n}{2}$ chains each containing $\frac{n}{2}$ edges.

To store all these chains,
we require $O(n)$ space.

Scanned with CamScanner

Query Time = $O(\log^2 n)$

Space Complexity = $O(n^2)$

Preprocessing Time = $O(n \log n)$

In order to reduce space complexity, if an edge belongs to multiple chains then instead of storing that edge in all the chains, we store it in a common ancestor of all the chains.

By storing the edge only once, we can reduce the space to **LINEAR** ($O(n)$).

Implementation of Query Processing becomes a bit tricky in this case. We need to carefully handle cases where a common edge has been listed in a node & its shared with some of its descendents.

The only part that we need to address here is that the given planar partitioning may not be regular.

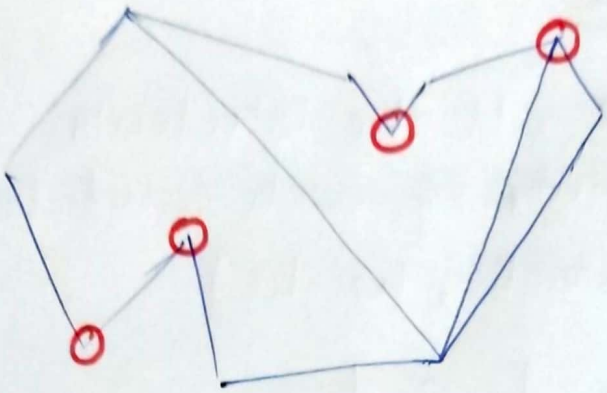
There might be some vertices with no outgoing edges

or

There might be some vertices with no incoming edges

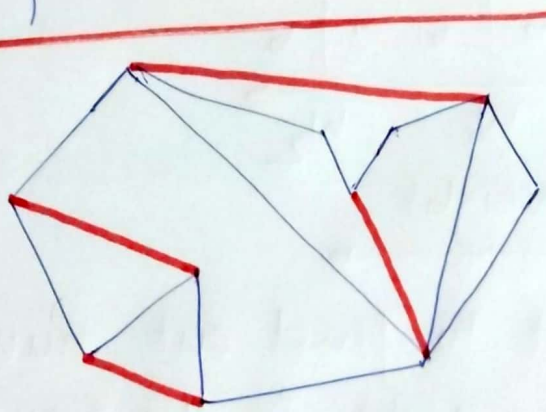
Scanned with CamScanner

Ex:-



O = non regular vertices.

For this, we add a few edges & make this regular.



— : newly added edge

REGULAR

Now the question is, how to add these edges.
For each non regular vertex, we have to identify to which vertex this edge needs to be added.

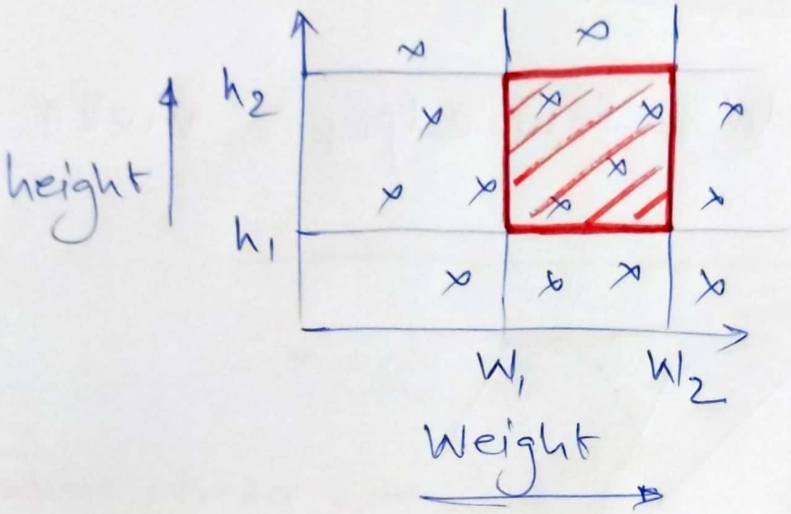
Can be done in $O(n \log n)$ TIME COMPLEXITY using a Sweep line algorithm.

Scanned with CamScanner

ORTHOGONAL RANGE SEARCH

Database Search

Suppose we take all the students of the class & plot points of each candidate as (weight, height)



Suppose we want to find out number of students having weight in the range $[w_1, w_2]$ & height is in the range $[h_1, h_2]$.

So basically, we are looking at a rectangle. Since this rectangle is orthogonal / parallel to axes, this is called **ORTHOGONAL RANGE QUERY**.

If we take a 3rd parameter, we can have a 3-dimensional orthogonal ~~range~~ range query.

Scanned with CamScanner

There are two types of queries:-

① # students within this rectangle
(COUNTING QUERY)

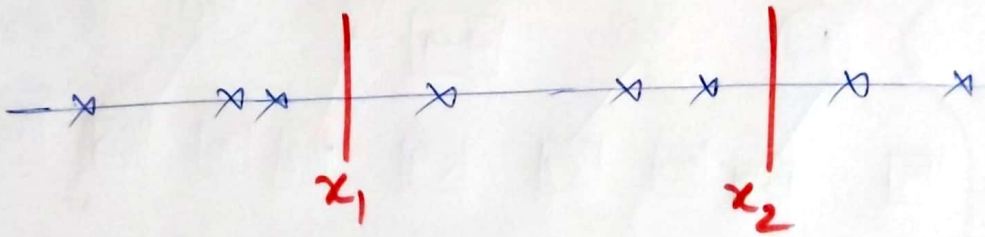
② List all the points within the rectangle
(LISTING QUERY)

Query Time:- COUNTING $O(\log n)$ LISTING $O(\log n + k)$

k → OUTPUT SIZE

Let us first have a look at a simpler version.

1 DIMENSIONAL ORTHOGONAL QUERY



Given:-

n numbers

Query:-

$[x_1, x_2]$

Output:-

Number of points in this range.

Sort them & then use Binary Search

S: $O(n)$ P: $O(n \log n)$ Q: $O(\log n)$ COUNTING
$O(\log n + k)$ LISTING

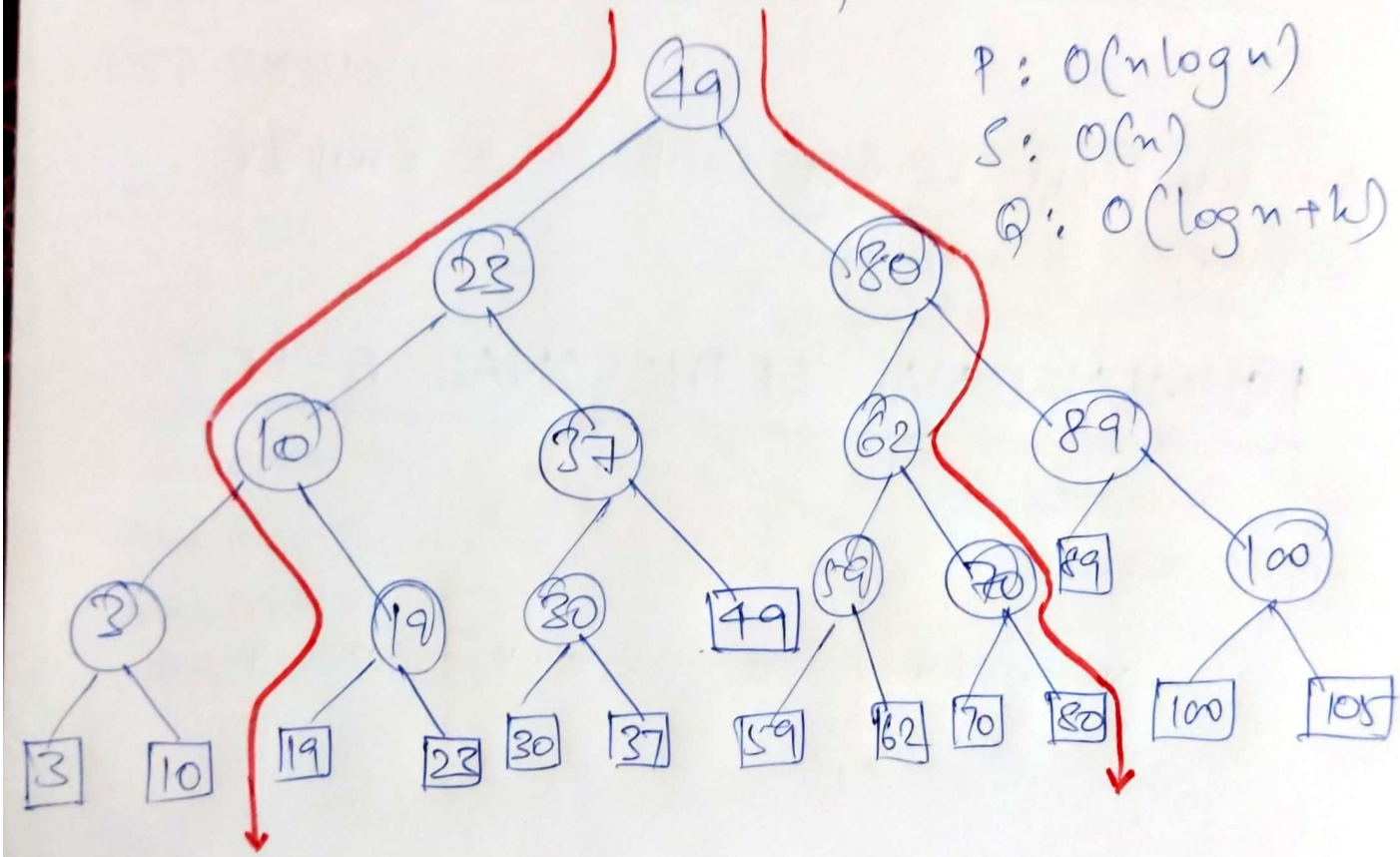
Scanned with CamScanner

This is pretty simple but the problem with this is that it cannot be extended to 2D.

We look at another method of solving the problem.

By using Binary Search Tree (BST)

Let us look at an example.



P: O(n log n)
S: O(n)
Q: O(log n + k)

This is a variation of BST.

Storing data in leaf nodes.

Internal nodes aid in deciding whether to go left or to go right while searching a particular value.

Every internal node contains the value of the largest element in its left subtree.

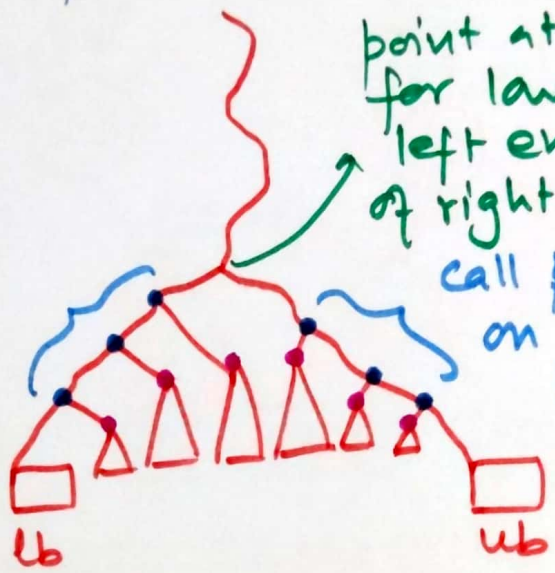
Scanned with CamScanner

Suppose we are given a query

[18, 77]

We find the lower bound of left end & upper bound of right end in the BST.

Then, we need to print all the elements in the range [lb, ub).



Since no. of elements inside a tree
= O(number of leaf nodes)

no. of nodes visited in subtree of purple vertices = O(no. of leaf nodes)
= O(output size)
= O(k)

Hence, time complexity remains
O(log n + k)

Scanned with CamScanner

For counting query, we can keep count of leaf nodes at internal nodes & directly add them instead of traversing over the subtree & counting. This removes the factor of k & hence time complexity becomes $O(\log n)$.

IMP.

Before orthogonal range query, we had discussed POINT LOCATION & the query time for CHAIN METHOD came out to be $O(\log^2 n)$.

How to get a logarithmic time complexity?

P: $O(n \log n)$ ⟶ very good

S: $O(n)$ ⟶ very good

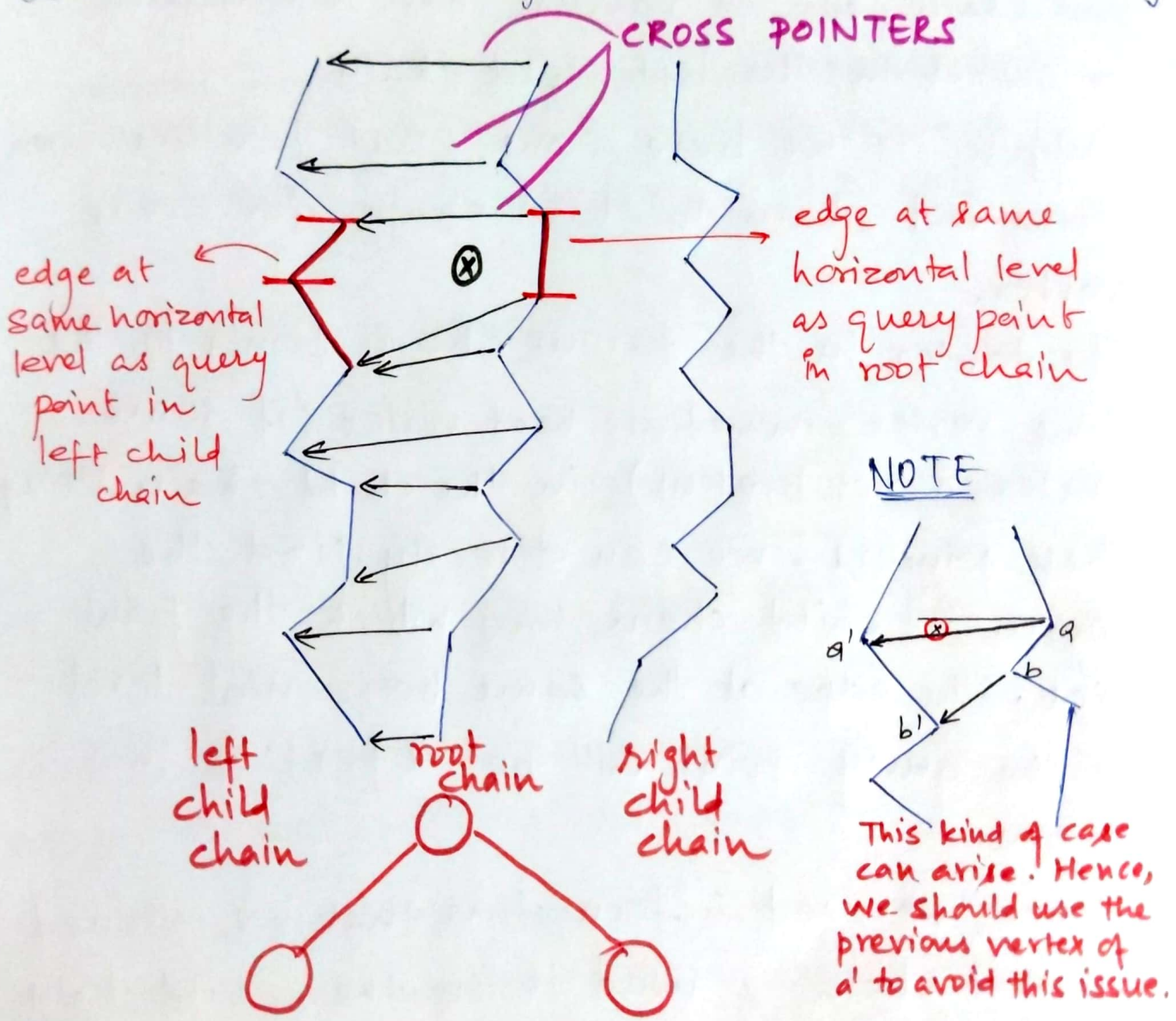
So, only query time of $O(\log^2 n)$ is NON-OPTIMAL. There is a method that takes $O(\log n)$ query time with $O(n \log n)$ preprocessing time & $O(n)$ storage.

If you look at CHAIN METHOD, we have a primary Data Structure (BST) where each node represents a chain. We are doing Binary Search in these chains & accordingly move to left or right. If we move left, we again do binary search on this chain from the beginning. If somehow,

Scanned with CamScanner

we could apply the information that resulted after binary searching previous chain to this chain, we wouldn't have to binary search from the beginning of this subchain again & it would be more efficient.

Let us look at a diagram for better understanding.



At the root, we search wrt the root chain. Whether the query point lies to the left or to the right by binary searching the edge which is at the same horizontal level as the query point.

Scanned with CamScanner

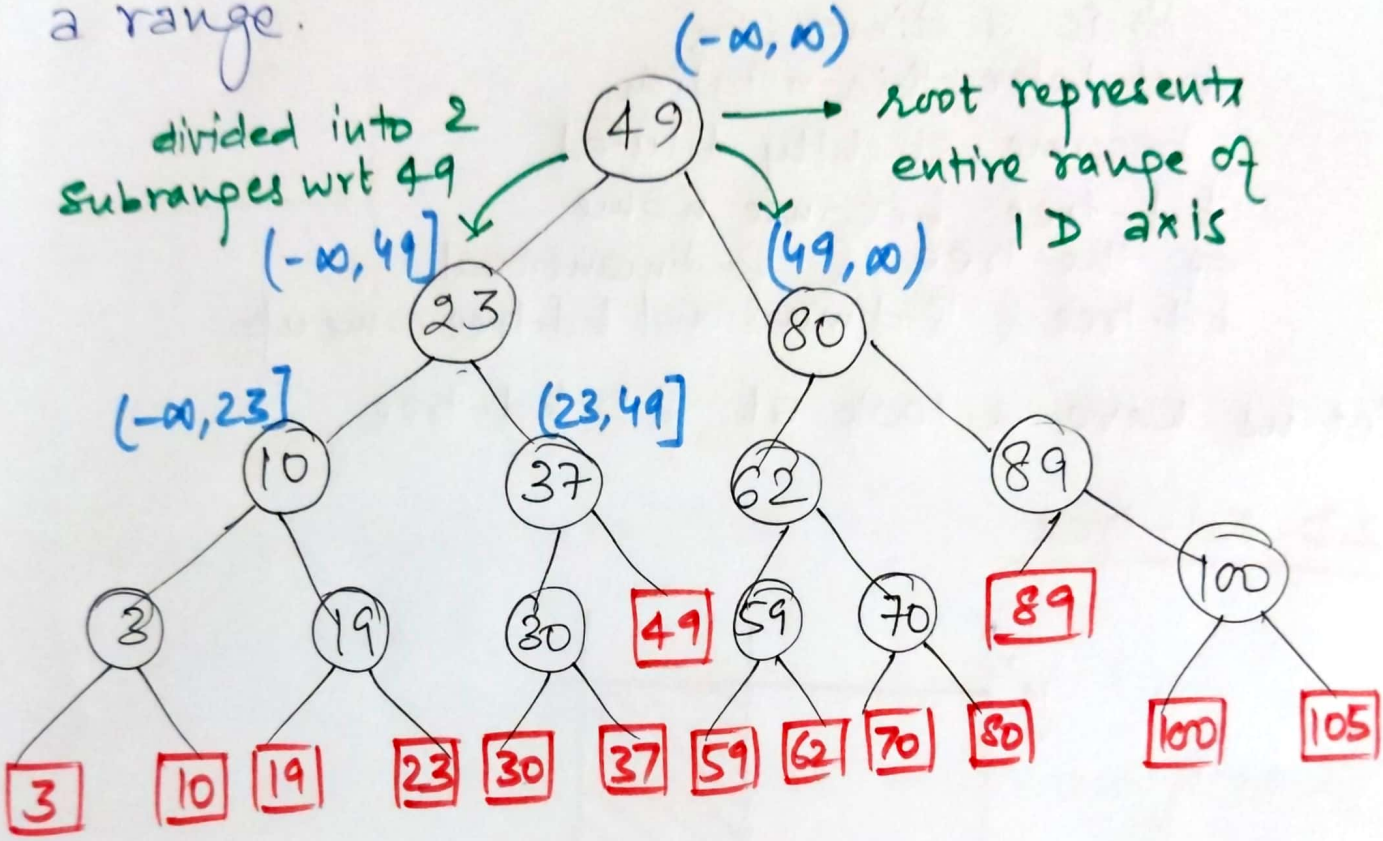
In our example, the point is to the left. Hence, next we will check left child chain.
We will binary search for the edge in this chain at the same horizontal level as the query point.
So, whatever search that we did on the root chain, we do not use that information to search in the left child chain.
Suppose if we have some cross pointers from root chain to child chain for every vertex.
The pointer in the parent chain points to the same vertex / immediate next vertex (if same vertex is not present) in the child chain. Using these pointers, we can directly find the region in child chain b/w which the point lies. The edge at the same horizontal level as the query point will be a part of this region.
Hence, we used information that we attained in root chain & used it in the child chain.
This avoids a binary search in the child chain.
Due to this, we drop a log factor & hence time complexity becomes $O(\log n + k)$.

Scanned with CamScanner

COMPUTATIONAL GEOMETRY

Lec 16

We can look at the Binary Search Tree in another way. Every node represents a range.



Given a range, look at intersection of range at root with given range.

If they intersect & range of current node doesn't lie completely within query range
↳ search on the left & right subtrees recursively.

If they intersect & range of current node lies completely within query range
↳ no need of going deeper recursively
output all leaf nodes within the subtree as all points belong to query range

Scanned with CamScanner

If they do not intersect
↳ Do nothing, return.

EXTENSION to 2-DIMENSION

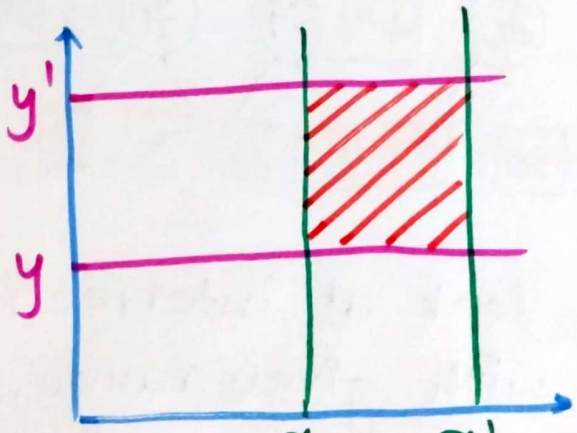
kd-tree

↓

(k) dimension tree
↳ is a dimension
but later, this notation became slightly diluted
kd-tree became name of the tree & 2-dimensional kd-tree & 3-dimensional kd-tree came up.

Let us have a look at 2D kd-tree.

2D kd-tree

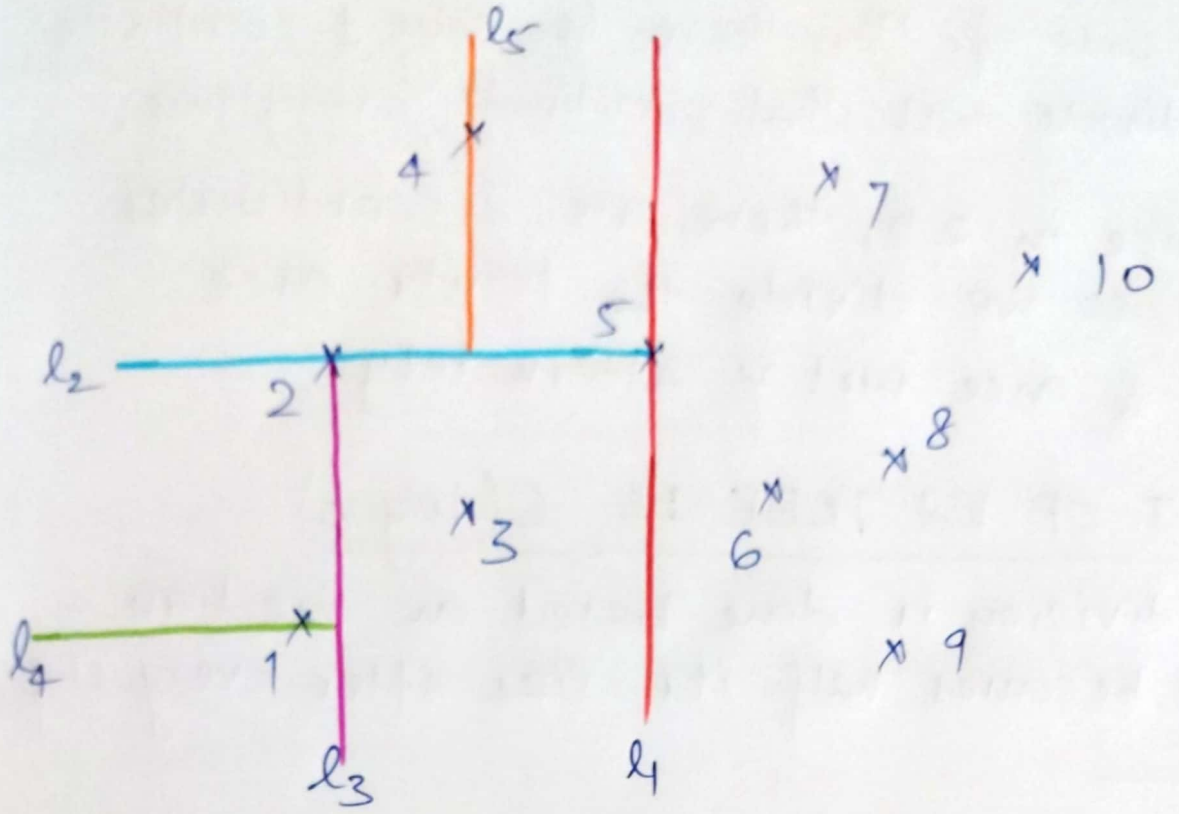


x-range $[x, x']$ × y-range $[y, y']$

Query input

We want to report all the points within this range.
We need to combine these 2 1-D queries to answer the 2D query.

Scanned with CamScanner



VERTICAL & HORIZONTAL ALTERNATELY

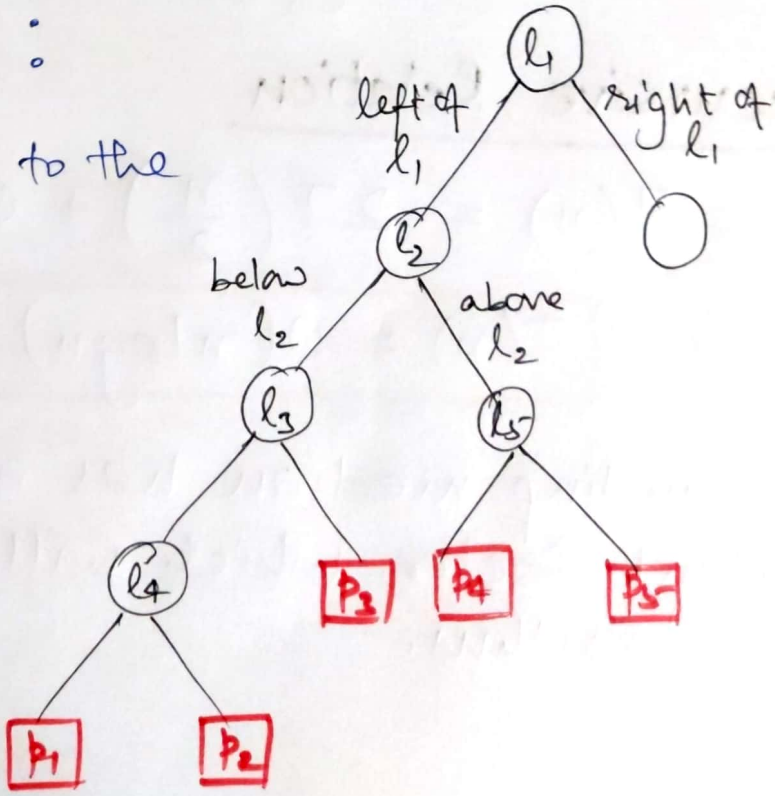
Divide the plane into two regions wrt vertical line l_1.
Divide the left half wrt horizontal line l_2
Divide the bottom half wrt horizontal line l_3

⋮

We do the same to the right half too.

This is called

"2D kd TREE"



Scanned with CamScanner

In the case of 1D, there is only 1 coordinate & we divide wrt that coordinate everytime.

But since in 2D, there are 2 coordinates x & y so we divide the points once wrt x & once wrt y alternately.

HEIGHT OF kd TREE IS $O(\log n)$

Since division is done based on median. So list becomes half its size after every step.

Finding median of n points

$\Downarrow$

$O(n)$

Dividing them into two halves wrt median

$\Downarrow$

$O(n)$

Recursive Relation

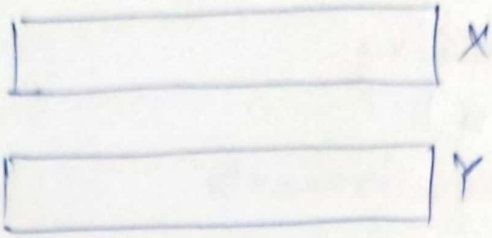
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$\Rightarrow \boxed{T(n) = O(n \log n)}$$

But finding median has a very high constant factor which will slow down our algorithm.

Scanned with CamScanner

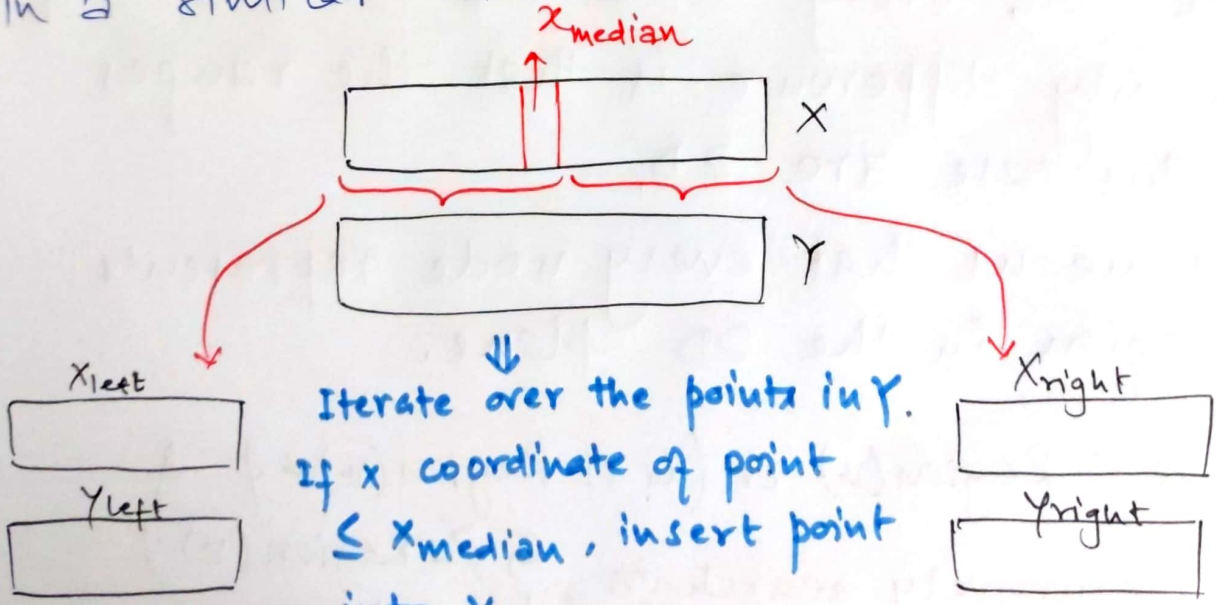
Hence, we will try to avoid it.
We will do this by maintaining two arrays.



First one contains points sorted by x coordinate

Second one contains points sorted by y coordinate.

Here, we will see how are points broken into 2 subarrays wrt median x coordinate (vertical line). Sorting wrt median y coordinate (horizontal line) can be done in a similar manner.



TIME COMPLEXITY = $O(n)$

Scanned with CamScanner

Hence, total time complexity remains same.
The arrays were sorted only once at the beginning.

→ Preprocessing
P: $O(n \log n)$
→ Space requirements
S: $O(n)$

Q = ?
→ Query Time

Now how to calculate answer to our queries using this Data Structure?

At the start of the lecture, we had seen that every node represents a range.
Here also, every node represents a range.
The only difference is that the ranges in this case are 2D.
This means that every node represents a region in the 2D plane.

If $Q \subseteq Region(v)$ or $(Q \cap Region(v) \neq \phi$ & $Q \not\supseteq Region(v))$
recursively search on left & right subtrees

If $Q \supseteq Region(v)$
no need to search further
output all the points inside this region as all of them belong to query region.

Scanned with CamScanner

If $Q \cap Region(v) = \phi$

Return.

No point of this region belongs to query.

$Q : O(\sqrt{n} + k)$

↑ We will see how in the next lecture

Beauty of kd-tree is that it can be extended to higher dimensions also.

$P : O(d \cdot n \log n)$

$S : O(dn)$

$Q : O(n^{1-\frac{1}{d}} + k)$

Scanned with CamScanner

COMPUTATIONAL GEOMETRY

Lec 18

NOTE: Lec 17 was disrupted due to an issue in MS Teams. The only discussion that was done was revision of Lec 16. Hence, ~~sh~~ skipping Lec 17 in Notes.

---

partial intersection

① If $Q \subseteq Region(v)$ or ($Q \cap Region(v) \neq \phi$ & $Q \not\supseteq Region(v)$) ④

recursively search on left & right subtrees.

② If $Q \supseteq Region(v)$

no need to search further
output all points inside this region as all of them belong to query region

③ If $Q \cap Region(v) = \phi$

Return.
No point of this region belongs to query.

We focus on this part here.

We want to put a bound on the number of regions that our query region boundary can partially intersect.

For this, we will find out the maximum number of regions that a straight line can intersect.

A single edge of the boundary will intersect

Scanned with CamScanner

with maximum this number of regions.
Multiply that by 4 to get maximum number of regions that our query region boundary can intersect.

We aren't interested in the second case as the time complexity associated with it cannot exceed $O(k)$.

we simply output all leaf nodes in the subtree

Complexity $\geq O$(# nodes in the subtree)
$= O$(# leaf nodes in the subtree)

Hence, summing over all such subtrees

Complexity $= O$(output size)
$= O(k)$

Case 1 is also not important because it only recurses on either the left subtree or the right subtree. The moment it recurses on both, from there onwards we will get case 4. Case 1 only works on 1 subtree (left or right) before splitting

↳ recursion goes on both left subtree & right subtree.

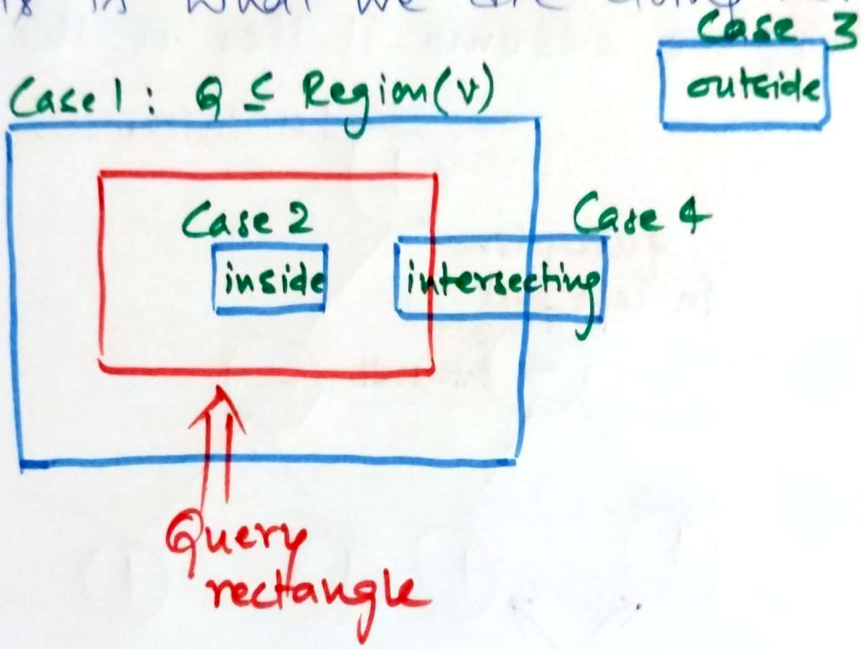
After this, we start getting case 4 & do not get case 1

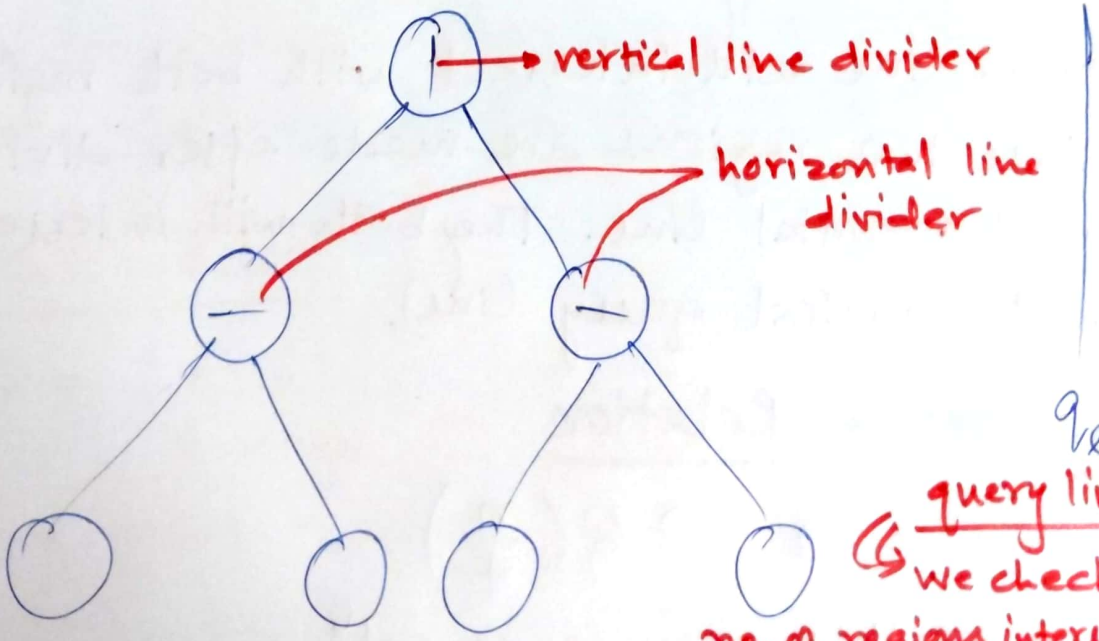
Scanned with CamScanner

Hence, complexity of case 1 depends on height of tree since it only descends on 1 branch.

$\Rightarrow O(\log n)$

To govern overall time complexity, we need to find out number of occurences of case 4. This is what we are doing here.





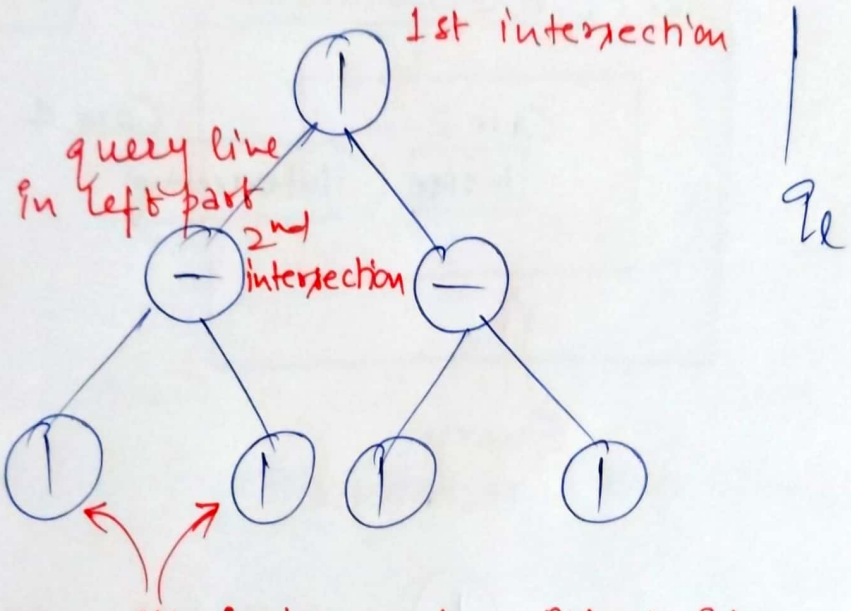
we check no. of regions intersecting with this line & multiply the result by 4 to estimate no. of regions intersecting with Query region.

Scanned with CamScanner

Since the root represents the entire plane, the query line will definitely intersect with this region.

The query line can either be on the left or the right part of the plane divided by the line (parallel to y-axis) passing through median point. It cannot be both.

WLOG, we assume it lies in the left part.



Query line will intersect with both regions (these two regions are made after division by horizontal line. They both will intersect with vertical query line).

Recurrence Relation

$$Q(n) = 2 + 2Q\left(\frac{n}{4}\right)$$

From this recurrence rel^n,

$$Q(n) = O(\sqrt{n})$$

Scanned with CamScanner

Hence, overall complexity becomes

$$Q : O(\sqrt{n} + k)$$
$$S : O(n)$$
$$P : O(n \log n)$$

This can be extended to higher dimensions also.

$$P : O(nd \log n)$$
$$S : O(nd)$$
$$Q : O(n^{1-\frac{1}{d}} + k)$$

The query time is $O(\sqrt{n} + k)$. This is higher than logarithmic time complexity. Let us look at another way of building the tree called as "RANGE TREE"

RANGE TREE

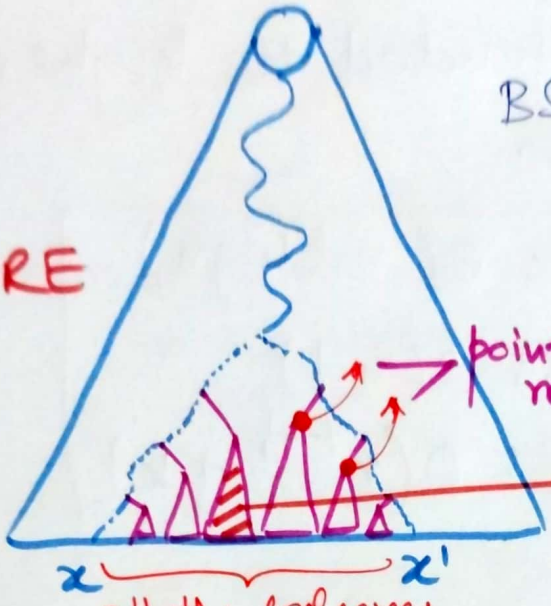
Query $[x : x'] \times [y : y']$

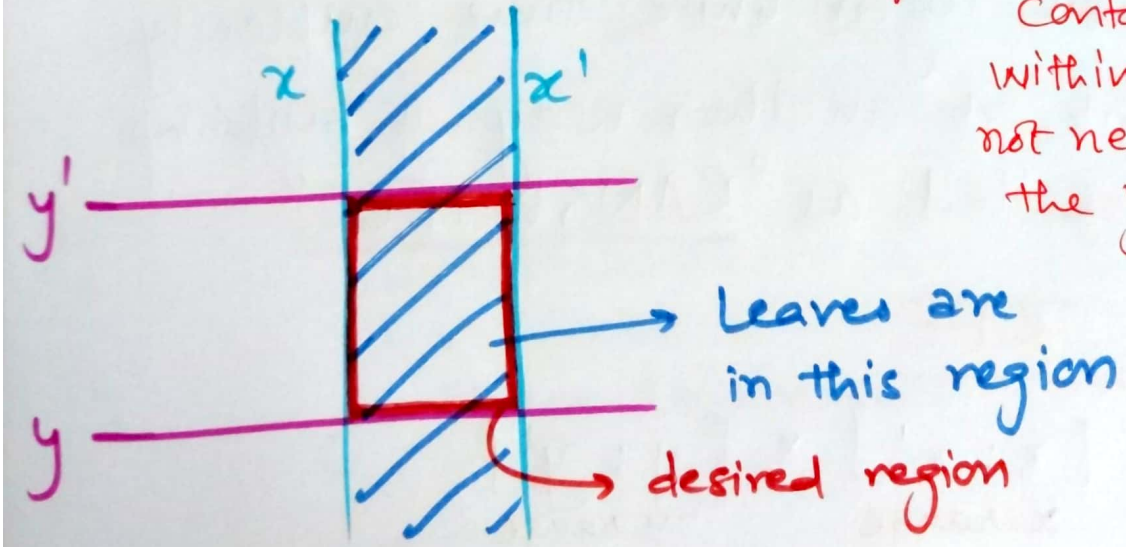
x range y range

We can treat them as 2 1-dimensional ranges. In kd-tree, we combine these 2 1-D ranges, we alternately divided points with their x coordinate & y coordinate.

Scanned with CamScanner

In range tree, we won't do like that. We will build a BST purely based on x coordinates. For the time being, forget about y coordinates. Treat them as 2 independent 1D ranges.



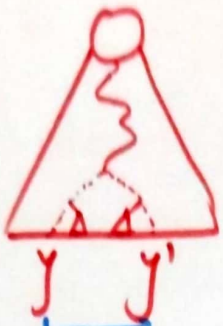


If we want to extract the points from this that fall within the y range, then what should we do?

Scanned with CamScanner

Build another BST taking the leaf nodes inside the chosen subtree. Build the BST based on y coordinate.

SECONDARY DATA STRUCTURE



searching for yrange in BST

all points within this region belong to the query region

For every node in Primary Data structure, build a BST containing the leaf nodes in that subtree based on y coordinate & attach a pointer to this BST to the node in the primary Data Structure.

This structure is known as a Range Tree.

Preprocessing
$P: O(n \log n)$

Space Requirement
$S: O(n \log n)$

Query Time
$Q: O(\log^2 n + k)$

can be further improved to $\log n$.

Scanned with CamScanner

COMPUTATIONAL GEOMETRY

Lec 19

Q: $O(\log^2 n + k)$ → output complexity

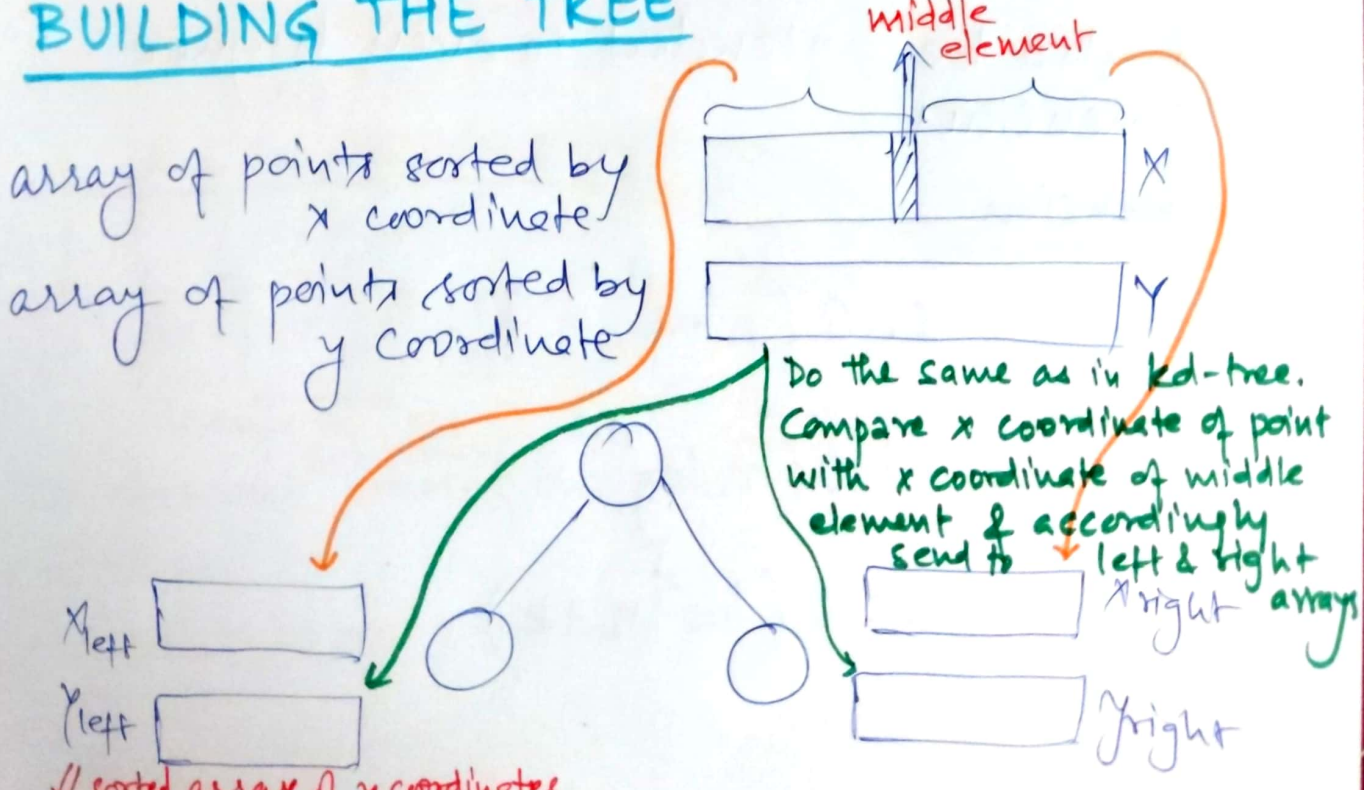
There are $O(\log n)$ nodes on the search path in the primary data structure. For every node, we will search the y range in its corresponding BST which will incur $O(\log n)$ per node.
Hence, total time complexity becomes

$$O(\log^2 n).$$

P: $O(n \log n)$

S: $O(n \log n)$

BUILDING THE TREE



This can be used as a secondary DS. Optionally, you can also build a BST on these values

Scanned with CamScanner

Repeat the same procedure for the left & right subtrees.

making x_{left}, y_{left}, x_{right}, y_{right}

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$\Rightarrow \boxed{T(n) = O(n \log n)}$$

By this diagram, we can also see that the space complexity is $O(n \log n)$

The same thing can be extended to 3 dimensions also.

Primary DS based on x coordinate
Secondary DS based on y coordinate
Tertiary DS based on z coordinate.

Can also be extended to even higher dimensions

d dimension

$$S: O(n \log^{d-1} n)$$

$$P: O(n \log^{d-1} n)$$

$$Q: O(\log^{d} n + k)$$

Scanned with CamScanner

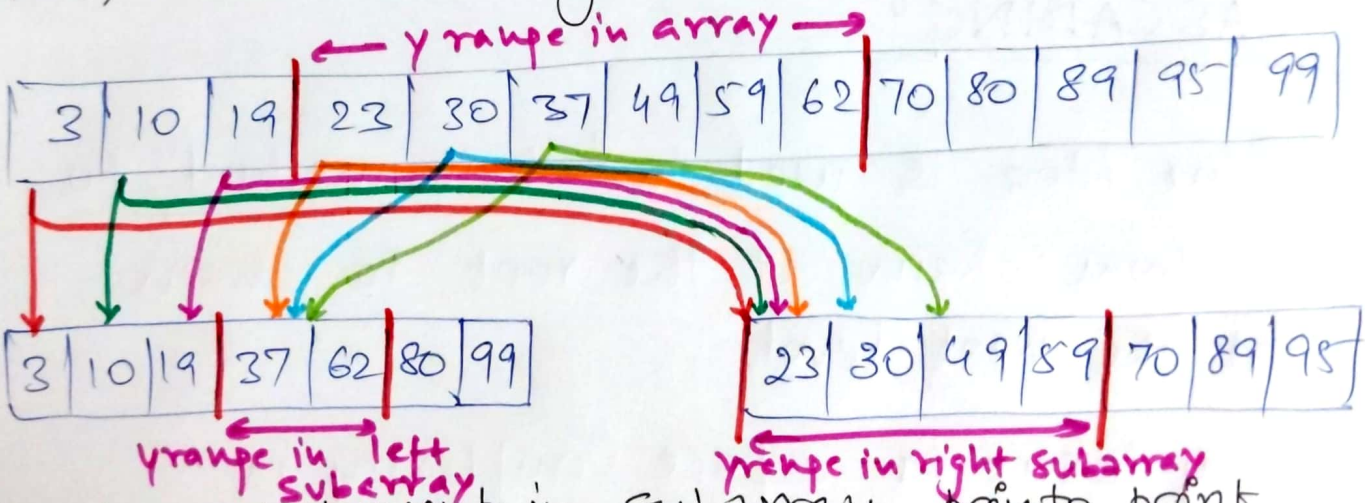
Coming back to the 2D case,

Query time can be improved to

$O(\log n + k)$

In the primary DS, we search for x range
In the secondary DS, we search for y range from the beginning for every BST.
If we can somehow use the range that we searched in previous BST to obtain information pertaining to this BST, we can lower the time complexity.
Let us look how can we do so.

Here, we take array as SECONDARY DS.



If element present in subarray, ~~point to~~ point to that.
else point to next higher element in subarray.

y range [20, 65]

Scanned with CamScanner

We have to search for γ & γ' only at the root. After that, we use pointers to get ranges in left & right subarrays in constant ($O(1)$) time

Due to this, we get away with 1 log factor.

$$T.C. = O(\underbrace{\log n}_{\text{PRIMARY DS}} + \underbrace{\log n}_{\text{1 binary search in SECONDARY DS}} + k)$$

d dimension

$$O(\log^{d-1} n + k)$$

THIS IDEA IS KNOWN AS "FRACTIONAL CASCADING"

Same idea is used in "chain method" to go from chain at the root to chain at the next level.

Only issue is Space Complexity is

$O(n \log n)$

NON-LINEAR

Scanned with CamScanner

We achieved optimal query time at the cost of non-linear space complexity.
We achieved time complexity same as in 1 dimensional Binary Search in 2D case.
We cannot do better than this.

There are various results

when Query Time ↓ then Space complexity ↑

when Query Time ↑ then Space complexity ↓

THERE IS TRADE-OFF BETWEEN QUERY TIME & SPACE COMPLEXITY.

We do not pay much attention to preprocessing as it is done only once.

Scanned with CamScanner

EXTRA

Regularising a non-regular PSLG

This is a 2 pass algorithm. First we correct vertices with no outgoing edges. Then we correct edges with no incoming edges. The first pass is a sweep line algorithm from top to bottom & the second is a sweep line algorithm from bottom to top. The second is a mirror image of the first.

Observation

If vertex v comes in between two adjacent edges e_1 & e_2 and v_1 & v_2 are the upper endpoints of e_1 & e_2 respectively, and $y(v_1) \leq y(v_2)$ then vv_1 does not cross any edge. This edge can be added if v needs an edge in that direction in order for the graph to be regular.

Algorithm

Given graph G

PASS 1:-

$Q = \phi$ (Event Queue) → Min. heap of points sorted by decreasing y coordinate

$L = \phi$ (Sweep Line Status) → Balanced BST of edges sorted by x coordinate

Scanned with CamScanner

Add all points from G into Q.

while $Q \neq \phi$

P = Extract_Max(Q) → edges going up

If p has outgoing edges then

remove all outgoing edges of P from L

Else

e_1 = left adjacent edge of p in L

e_2 = right adjacent edge of p in L

If $e_1 \neq \phi$ or $e_2 \neq \phi$ then (not start vertex)

p_1 = top endpoint of e_1 (If $e_1 \neq \phi$)

p_2 = top endpoint of e_2 (If $e_2 \neq \phi$)

$p' = \underset{y}{\arg\min}(p_1, p_2)$

Add edge $e = pp'$ to the graph G

~~Add~~

Add Incoming edges of p into L.

↳ going down from p

PASS 2:

Similar to pass 1 except use min heap instead of max heap, use maximum y coordinate instead of minimum, use bottom endpoint instead of top, use outgoing instead of incoming & vice versa

Scanned with CamScanner

EXTRA

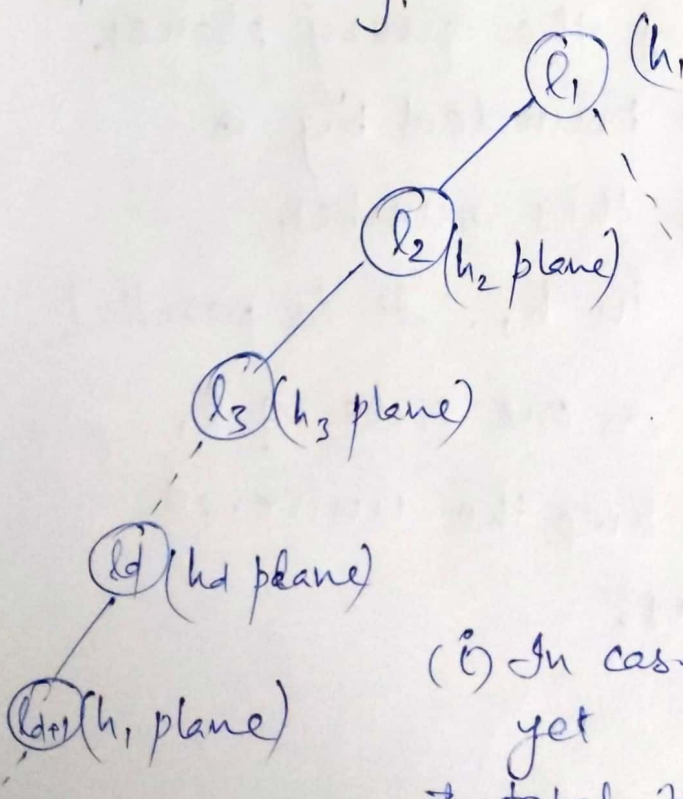
kd-tree higher dimension analysis

(d dimensions)

In d dimensions, we have d classes of orthogonal hyperplanes such that each class contains hyperplanes ~~are~~ parallel to one of the axes & two hyperplanes of different classes are orthogonal.

Classes $\{h_1, h_2, \ldots, h_d\}$

In a d dimension kd tree, first partitioning is done wrt a plane l_1 from h_1, then l_2 from h_2, then l_3 from h_3 etc. Back to h_1 eventually.



3 types of queries

(i) $\theta \subseteq$ Region (v)
(ii) $\theta \supseteq$ Region (v)
(iii) $\theta \cap$ Region (v) $= \phi$
(iv) $\theta \cap$ Region (v) $\neq \phi$

(i) In case (i), no split has occured yet
⇒ total time complexity of all cases of case (i) is $O(\log n)$

Scanned with CamScanner

(ii) In case (ii), all leaf nodes of region belong to query ⇒ Total time = $O(k)$
where k = output size

(iii) These regions are skipped. One at each level maybe ⇒ Total time = $O(\log n)$ (worst case)

(iv) This is complex.

① Query Time

In order to compute TC for (iv), we need a bound on the number of regions intersected by the query set (since split occur here).

For this, consider the number of regions intersected by one of the query planes. The total number is bounded by a constant multiple of this number.

This query plane is in h_1. It is parallel to l_1 ⇒ it lies only on one side of l_1.

It starts by intersecting the universe once into two regions.

Q_{h_i} definition:-

$Q_{h_i}(n)$ is the number of regions intersected

Scanned with CamScanner

by the query plane in a region containing n points to be bisected by a plane of class h_i

$$\theta_{h_1}(n) = 1 + \theta_{h_2}\left(\frac{n}{2}\right)$$

Lies in only one region

intersects only one region since parallel

Next region is intersected by line of h_2

$$\theta_{h_2}(n) = 2 + 2\,\theta_{h_3}\left(\frac{n}{2}\right)$$

Intersects 2 regions

Need to check max intersections of both regions

$$\Rightarrow \theta_{h_3}(n) = 2 + 2\,\theta_{h_4}\left(\frac{n}{2}\right)$$

$$\theta_{h_4}(n) = 2 + 2\,\theta_{h_5}\left(\frac{n}{2}\right)$$

$$\theta_{h_d}(n) = 2 + 2\,\theta_{h_1}\left(\frac{n}{2}\right)$$

After h_d, we get h_1 again

$$\Rightarrow \theta_{h_1}(n) = 1 + \theta_{h_2}\left(\frac{n}{2}\right)$$

$$= 1 + 2 + 2\theta_{h_3}\left(\frac{n}{4}\right)$$

$$= 1 + 2 + 4 + 4\,\theta_{h_4}\left(\frac{n}{8}\right)$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{d-2}\,\theta_{h_d}\left(\frac{n}{2^{d-1}}\right)$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{d-1} + 2^d\,\theta_{h_1}\left(\frac{n}{2^d}\right)$$

Scanned with CamScanner

$$= 2^d - 1 + 2^{d-1} \theta_{h_1}\left(\frac{n}{2^d}\right)$$

MASTER THEOREM

$$2^d - 1 = O\left(n^{\frac{d-1}{d}}\right) \longrightarrow \log_{2^d}(2^d - 1)$$ (small o)

$$\Rightarrow \theta_{h_1}(n) = O\left(n^{\frac{d-1}{d}}\right)$$

$$\Rightarrow T.C. = O\left(n^{1-\frac{1}{d}}\right) \quad [\text{FOR } 4^{th} \text{ CASE}]$$

$$\Rightarrow T.C = \underbrace{O(\log n)}_{I} + \underbrace{O(k)}_{II} + \underbrace{O(\log n)}_{III} + \underbrace{O\left(n^{1-\frac{1}{d}}\right)}_{IV}$$

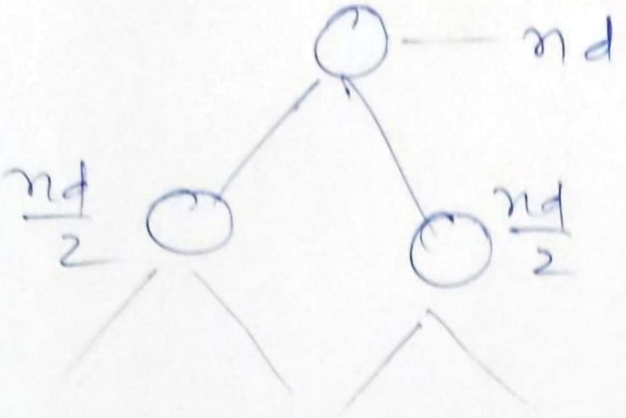
$$= O\left(n^{1-\frac{1}{d}} + k\right)$$

(2) Preprocessing Time

At each node, given $X_1, X_2, X_3, \ldots, X_d$ (arrays sorted by each dimension containing points referred to by that node), or median wrt some i^{th} dimension is found & corresponding partitions for 2 new sets of arrays $X_1', X_2', \ldots, X_d'$ & $X_1'', X_2'', \ldots, X_d''$ are found (for child nodes)

These partitions take $\theta(nd)$ time given n points represented by a node.

Scanned with CamScanner



$\frac{nd}{2} \times 2$

$\frac{nd}{4} \times 4$

$\vdots$

$\frac{n}{2^{\log n}} d \times 2^{\log n}$

$nd \log n$

$\Rightarrow P = O(nd \log n)$

(8) Space required

Leaves store the points. Each point has d coordinates

$\Rightarrow$ Space required $= O(nd)$

$\Rightarrow$ S: $O(nd)$

P: $O(nd \log n)$

Q: $O(n^{1-\frac{1}{d}})$

Scanned with CamScanner