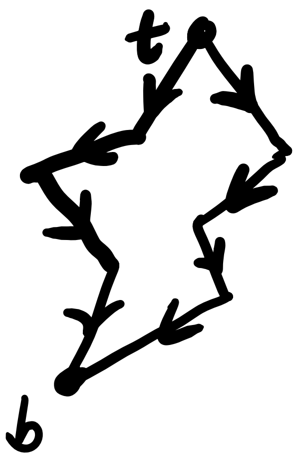
Lecture 5

Last time,
monotone polygons :



both paths from top to bottom are decreasing (lex order)

Algorithm : Triangulate simple polygon :

① Divide it into monotone parts

② Triangulate monotone polygon.

- Last time, did ② time $O(n)$.
- This week, we do ① in time $O(n \log n)$.

Types of vertices vs monotonicity

Start : $v > p, q$ (adjacent vertices) & has polygon below.

End : $v < p, q$ & has polygon above.

Reg : $p < v < q$ or $q < v < p$.

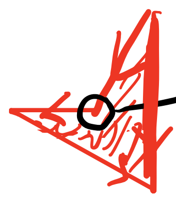
Split : $v > p, q$ & polygon above

Merge : $v < p, q$ & polygon below.

WARNING

- Recall P is y-monotone (monotone wrt y axis) if each horizontal line intersects P in 1 connected component – $\emptyset$, a pt or a segment.

- E-Learning claims P is y-monotone $\Leftrightarrow$ has no split or merge vertices.
- False :

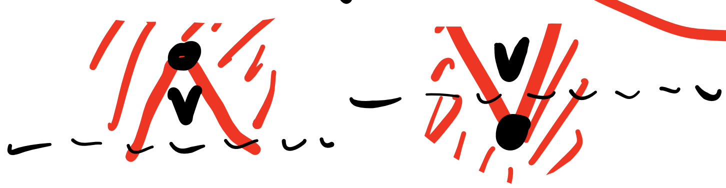


merge vertex but y-monotone.

① In fact P is monotone $\Leftrightarrow$ P contains no split or merge vertices.

&

② P is y-monotone $\Leftrightarrow$ P contains no y-split or y-merge verts.



hor. line splits as drawn

In Fact ① & ② are equivalent :

given P can find small clockwise rotation φ from start such that •

- P is monotone $\Leftrightarrow$ φP is y-monotone
- v is split/merge $\Leftrightarrow$ φP is y-split / y-merge.

Theorem

A simple polygon is monotone $\iff$ it contains no split or merge vertices.

Proof

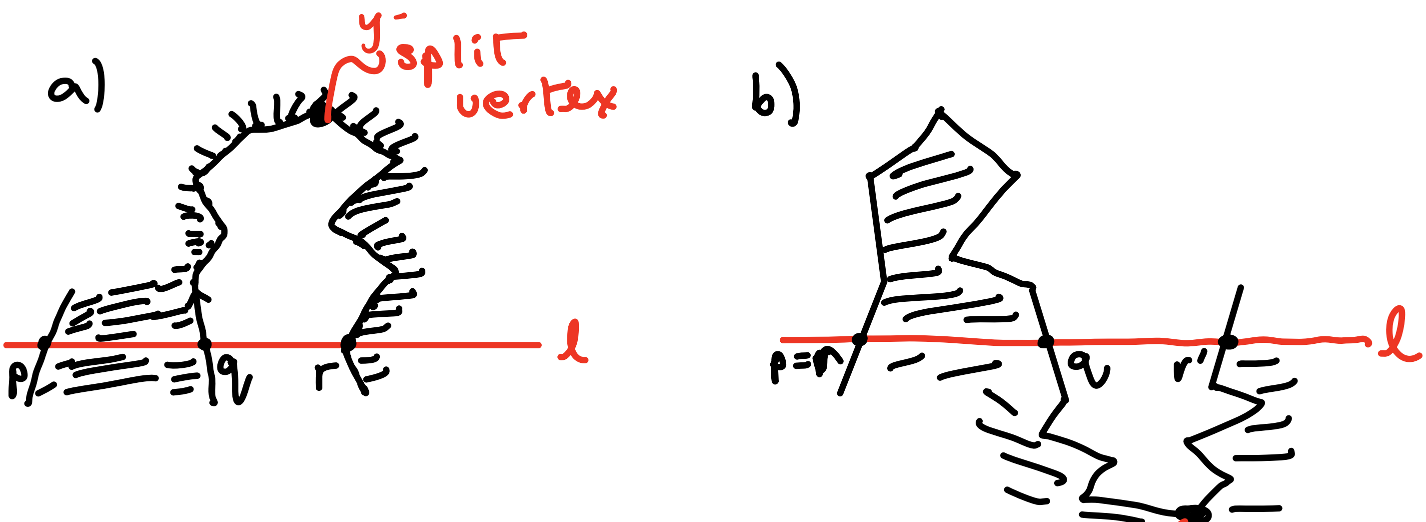
- By above, suffices to prove that P is y-monotone $\iff$ it contains no y-split or y-merge vertices.
- If P contains a y-split vertex 

the line ℓ splits it into 2 components $\Rightarrow$ not y-monotone. The y-merge vertex case is similar.

- Conversely, suppose P is not y-monotone, so there is horizontal line ℓ which intersects P in more than one connected component.

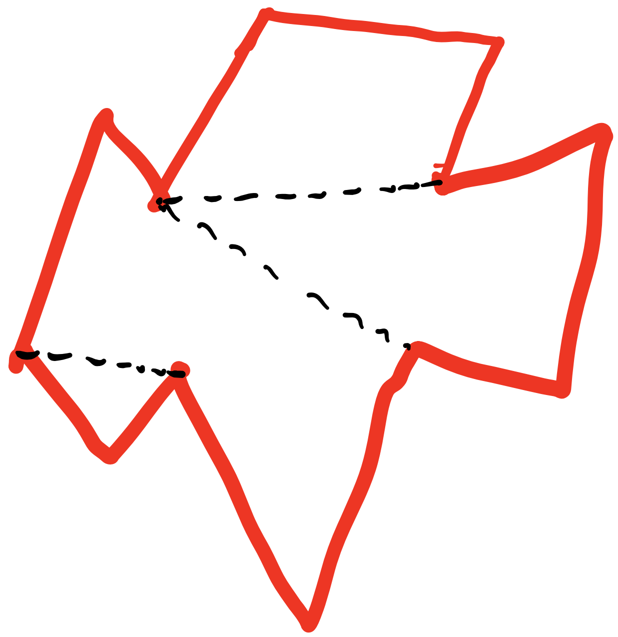
Can assume leftmost component of $P \cap \ell$ is a segment, not a point (else, move ℓ slightly vertically).

- Let p be left pt & q right point of segment
- Starting at q, follow boundary of P so P lies to left of boundary.
- Then at a point r, boundary of P intersects ℓ again.
- Two cases: a) $p \neq r$ & b) $p = r$.



- In case a) highest vertex between q, r is y-split.
- In case b), follow boundary from q in opposite direction & let r' be intersection point.
- The lowest vertex between q & r' is then a y-merge vertex. $\square$

Given the above, we can break simple polygons into monotone ones by removing split & merge vertices.



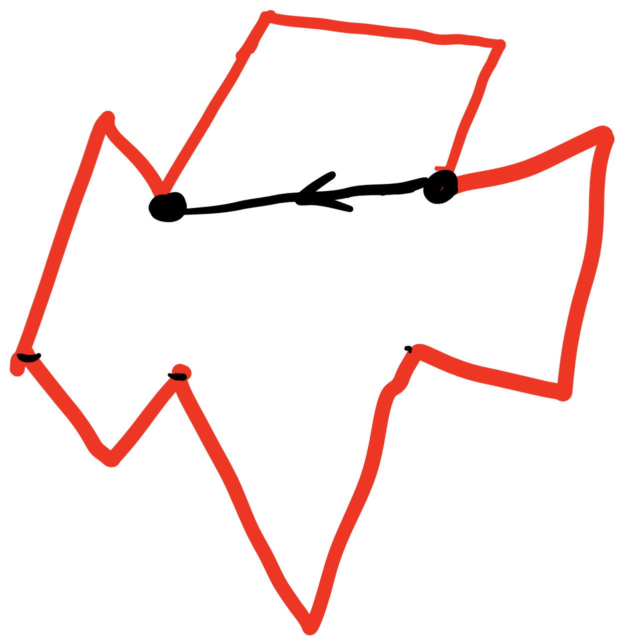
Idea: at merge vertex, draw a line downwards to a vertex

- At split vertex, draw a line upwards to a vertex.

Qn: To which vertices, do we draw lines?

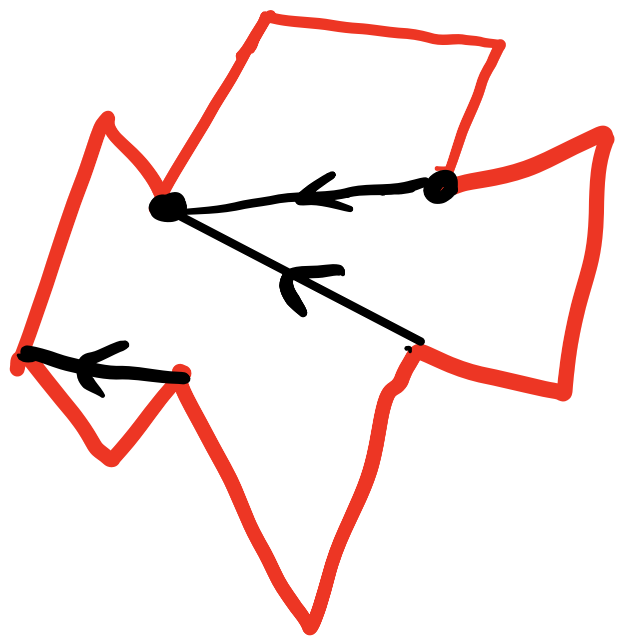
Naturally, we use a sweep-line algorithm from top to bottom.

Given the above, we can break simple polygons into monotone ones by removing split & merge vertices.



Idea: at merge vertices, draw line downwards to another vertex.

Given the above, we can break simple polygons into monotone ones by removing split & merge vertices.



Idea: at merge vertices, draw line downwards to another vertex.

at split vertex, draw line upwards to a vertex.

Q) To which vertices, do we draw lines?

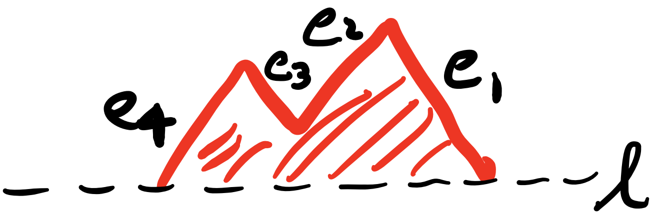
- Algorithm is a sweep-line algorithm.

Data structures

- Polygon stored in a DCEL D
- Event queue Q (bal. bin. tree as usual)

stores vertices of polygon in lex order.

• Bal bin tree T stores edges intersecting sweepline & having polygon to their right



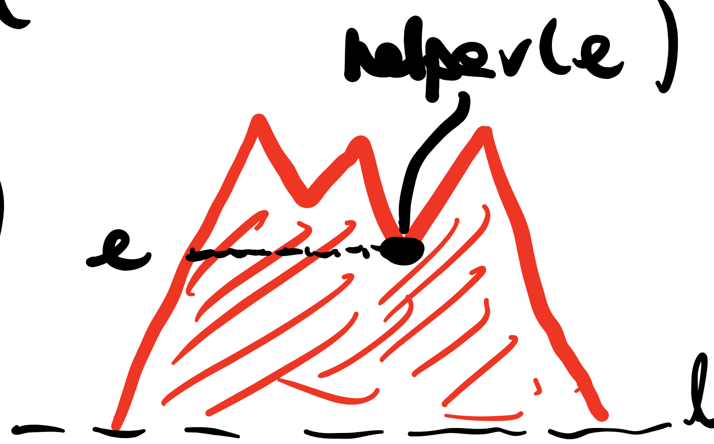
• At l_1, $T = \{e_4, e_2\}$

- Also, with each edge e in T we store a vertex p = helper(e):

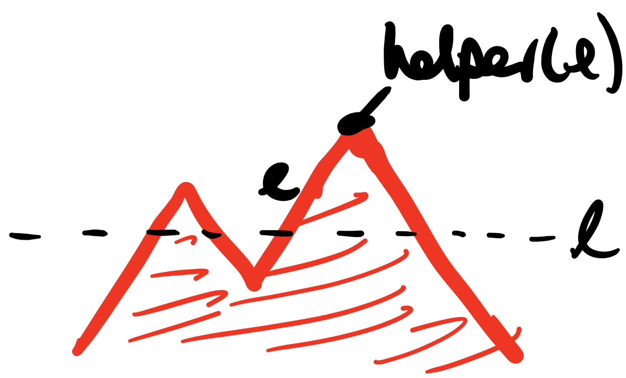
• helper(e) lies above l

• horizontal segment between e & helper(e) belongs to P.

• helper(e) is lex. least vertex with these properties.



• It may be the case that helper(e) is its upper endpoint.



Overview of algorithm

When sweepline passes vertex, we

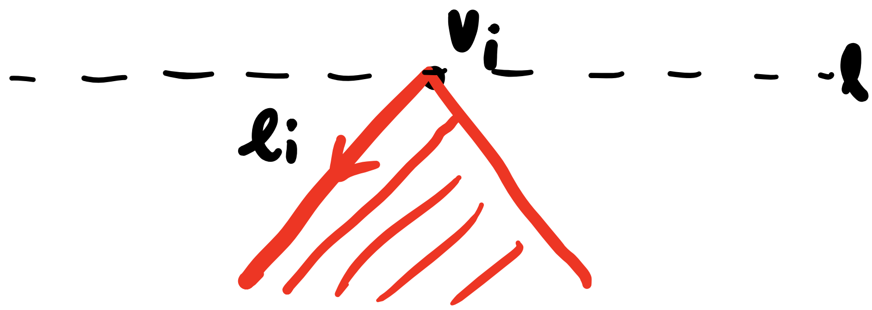
- connect a vertex with helper of edge in DCEL
- add edges & their helpers to T
- remove - - - - - - - - - - from T
- change helpers of some edges in T

Also, we use anticlockwise enumeration of vertices & edges

beginning from the top (calculated using DCEL)

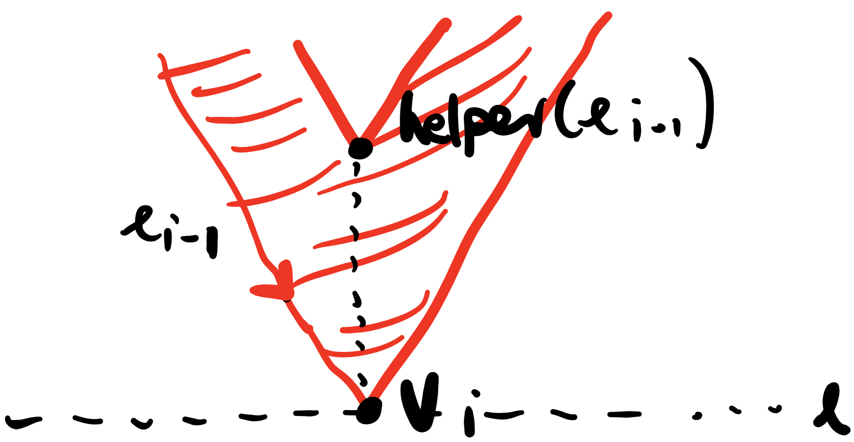
Cases to handle : types of vertex

Start



- Add e_i to T.
- Set $helper(e_i) = v_i$

End

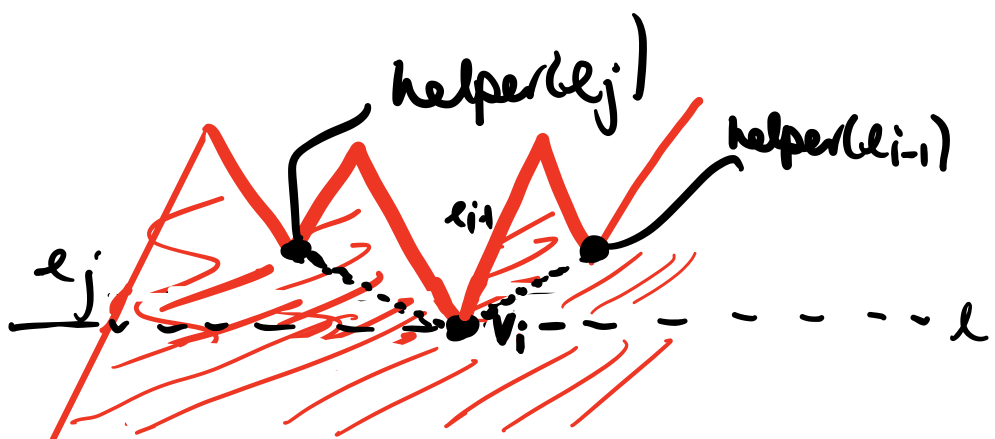


- If $helper(e_{i-1})$ is merge, add edge from v_i to it in D.
- Remove e_{i-1} from T.

Split

- Search T for closest edge e_j to left of v_i.
- Add edge from v_i to helper(e_j)
- Add e_i to T.
- Set helper(e_i) = v_i , helper(e_j) = v_i .

Merge

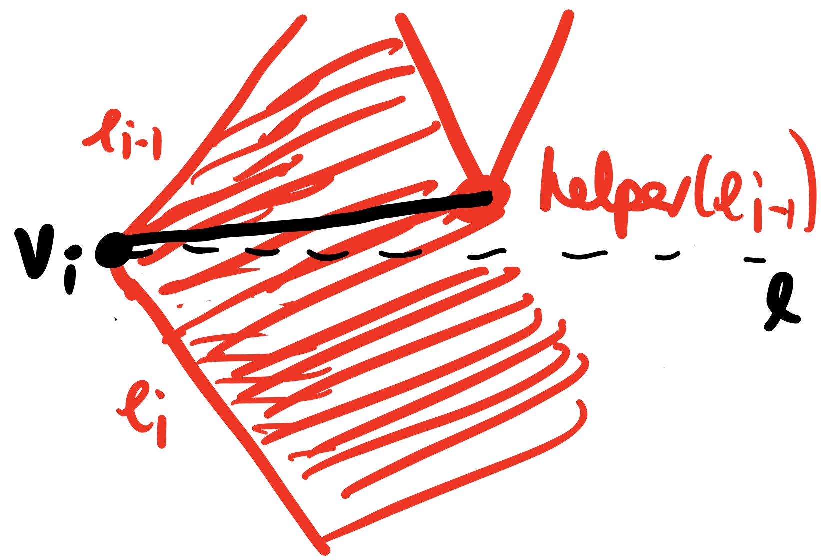


- If helper(e_{i-1}) is merge, add edge to v_i in D.
- If helper(e_j) is merge, add edge to v_i in D.
- Set helper(e_j) = v_i ,

- Delete e_{i-1} from T.

Regular vertex

Case 1: P to right of v_i

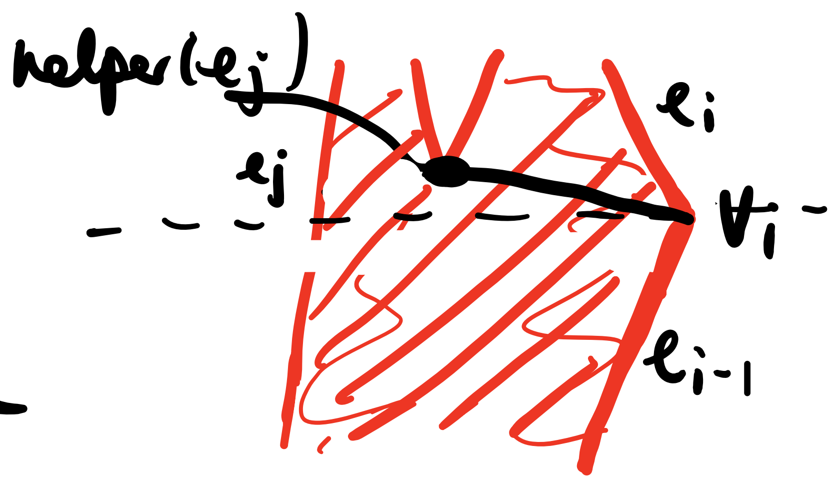


- If helper(e_{i-1}) is merge, draw a line from it to v_i.

- Delete e_{i-1} from T.
- Insert e_i into T, with helper(e_i) = v_i)

Else:

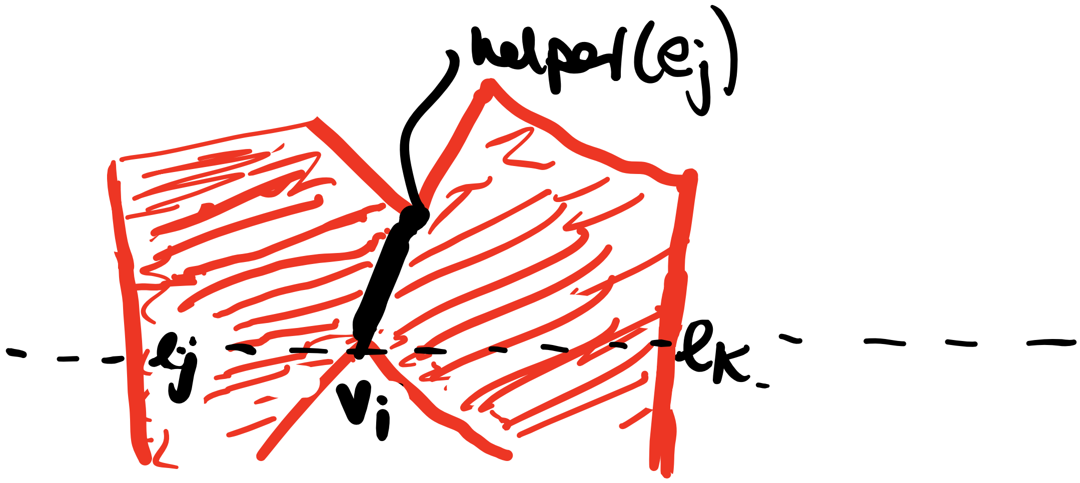
P to left of v_i



- Search T for edge e_j to left of v_i.
- If helper(e_j) is merge, draw line from it to v_i.
- Set helper(e_j) = v_i

Why does the algorithm work?

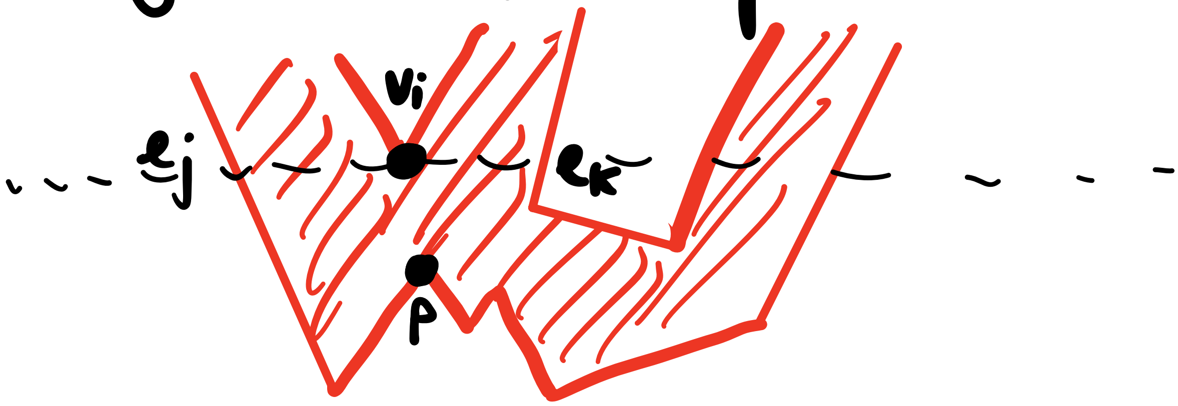
① Getting rid of split vertices:



- Consider split vertex v_i.

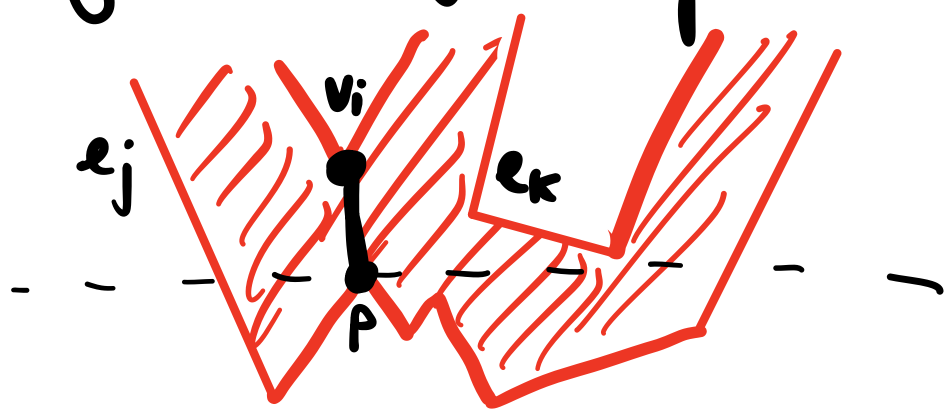
When sweepline passes v_i, it is connected to helper(e_j), the lowest vertex between left & right neighbours.

① Getting rid of merge vertices:



- Consider merge vertex v_i with left & right neighbours e_j, e_k.
  - When sweepline passes v_i, we set helper$(e_j) = v_i$.

① Getting rid of merge vertices:



- Consider merge vertex v_i with left & right neighbours e_j, e_k
  - When sweepline passes v_i, we set $\text{helper}(e_j) = v_i$.
- At max vertex p between e_j & e_k & below v_i, we add an edge from p to $v_i = \text{helper}(e_j)$.

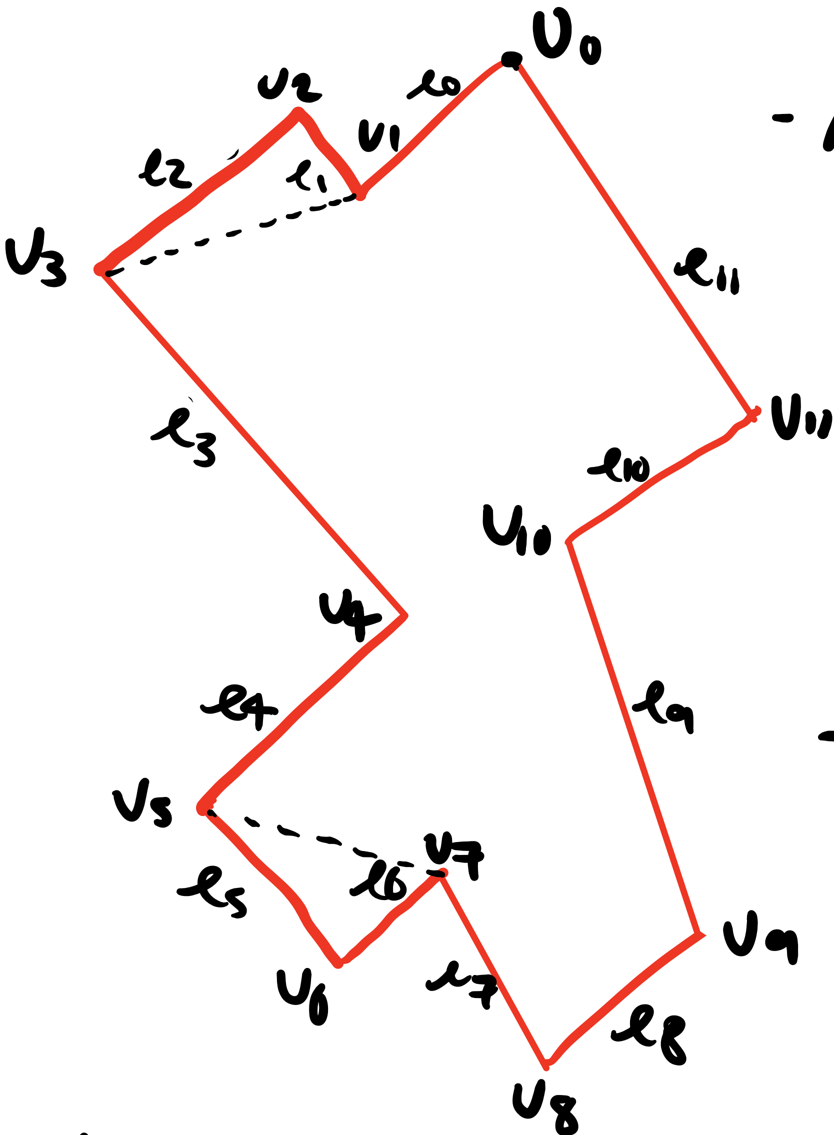
Complexity

- $O(n \log n)$ - order vertices in Q
- $O(n)$ calc. anticlockwise order

- Each event involves searching, rebalancing tree - time $O(\log n)$ - plus constant time operations:
  updating helpers
  adding edges
  (at most 1 or 2)

- Therefore complexity is
  $O(n \log n) + O(n) + O(n \log n)$
  $= \underline{O(n \log n)}$.

Example



- At v_0, add e_0 to T & $h(e_0) = v_0$.
- At v_2, also start, add e_2 to T, set $h(e_2) = v_2$.
- At v_1, $h(e_0) = v_0$ not merge, $h(e_2) = v_2$ not merge. Do nothing. Change $h(e_2) = v_1$.
- At v_3, reg. vertex, $h(e_2) = v_1$ merge. Add line v_1 to v_3. Remove e_2. Add e_3.
- At v_{11}, $h(e_3) = v_3$ so do nothing.
- At v_{10}, change $h(e_3) = v_{10}$.
- At v_4, remove e_3 & add e_4.
- At v_5, rem. e_4 & add e_5.
- At v_7 split, $h(e_5) = v_5$ so add line v_5 to v_7. Ch $h(e_5) = h(v_7)$. Add e_7.
- At v_9, do nothing.
- At v_6, remove e_5. At v_8, remove e_7. $\square$