Dynamic Computational Geometry

Roberto Tamassia

Department of Computer Science

Brown University

Summary

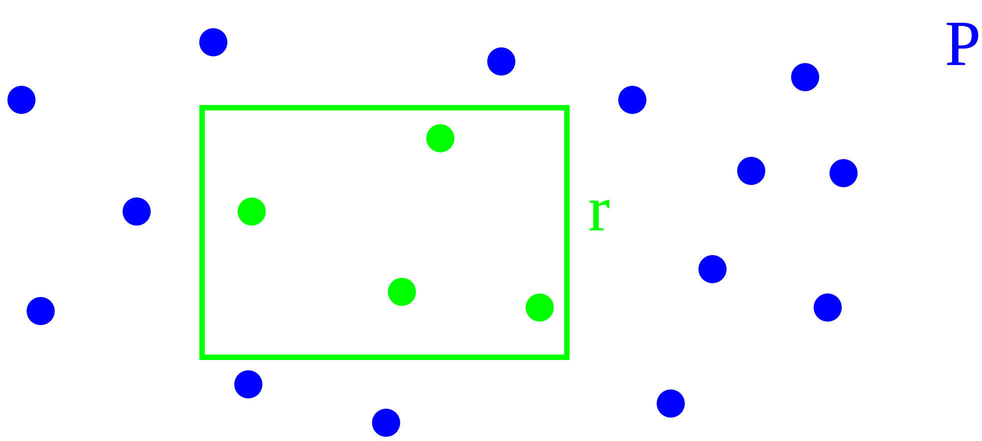
- Range Searching (Range Tree)
- Point Enclosure (Segment Tree)
- Segment Intersection
- Rectangle Intersection
- Point Location with Segment Trees
- Point Location with Dynamic Trees

Reference

- Y.-J. Chiang and R. Tamassia, “Dynamic Algorithms in Computational Geometry,” Technical Report CS-91-24, Dept. of Computer Science, Brown Univ., 1991.

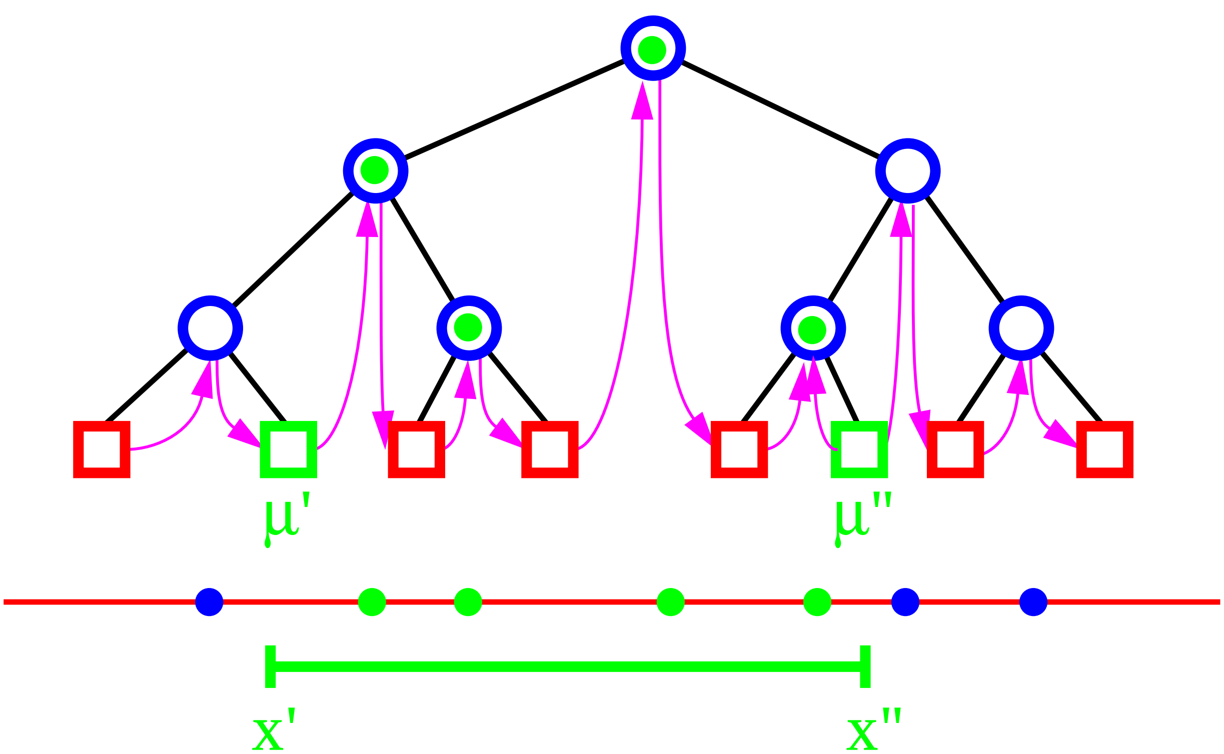
Range Searching

- Set P of points in d-dimensional space E^d
- Range Query: report the points of P contained in a query range r
- Query range:
  - $r = (a_1,b_1) \times (a_2,b_2) \times \ldots \times (a_d,b_d)$
  - d=1 interval
  - d=2 rectangle with sides parallel to axes
- Variations of Range Queries:
  - count points in r
  - if points have associated weights, compute total weight of points in r



One-Dimensional Range Searching

- use a balanced search tree T with internal nodes associated with the points of P
- thread nodes in in-order
- Query for range r = (x',x")
  - search for x' and x" in T, this gives nodes μ' and μ"
  - follow threads from μ' to μ" and report points at internal nodes encountered



Complexity of One-Dimensional Range Searching

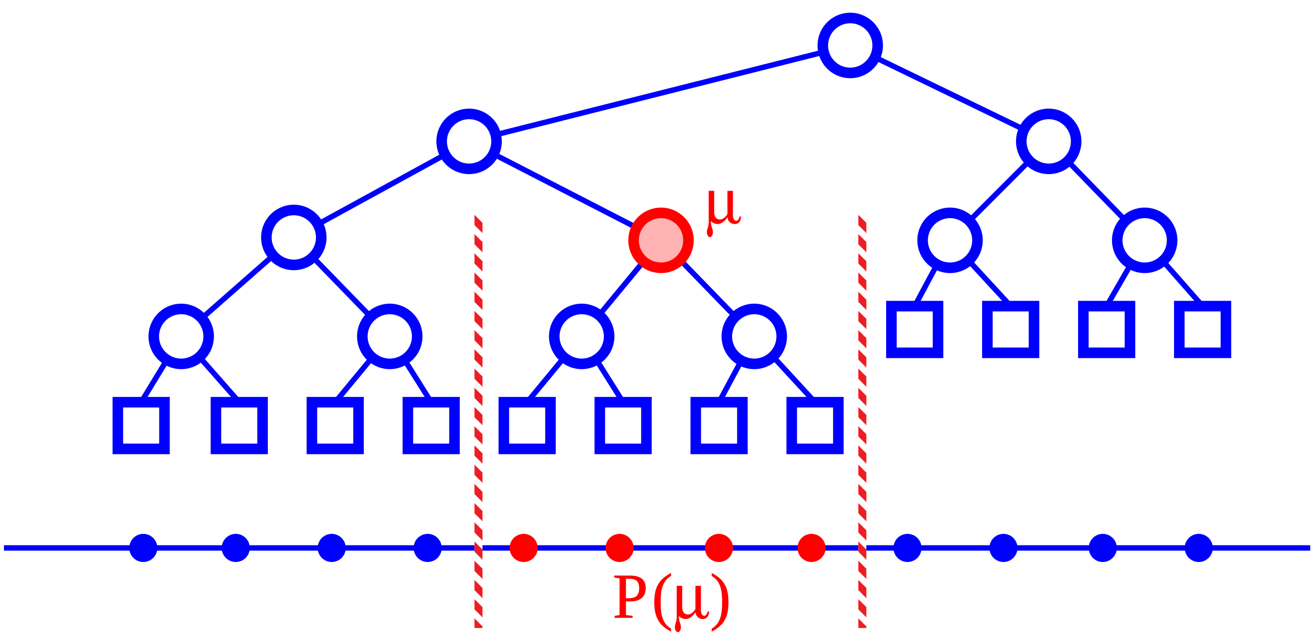
- Space requirement for n points: O(n)
- Query time: O(log n + k), where k is the number of points reported
- Time for insertion or deletion of a point: O(log n).
- Note that thread pointers are not affected by rotations.

Exercises

- * Show how to perform queries without using threads.
- * Show how to perform 1-D range counting queries in time O(log n).
- * Assuming that the points have weights, show how to find the heaviest point in the query range in time O(log n)

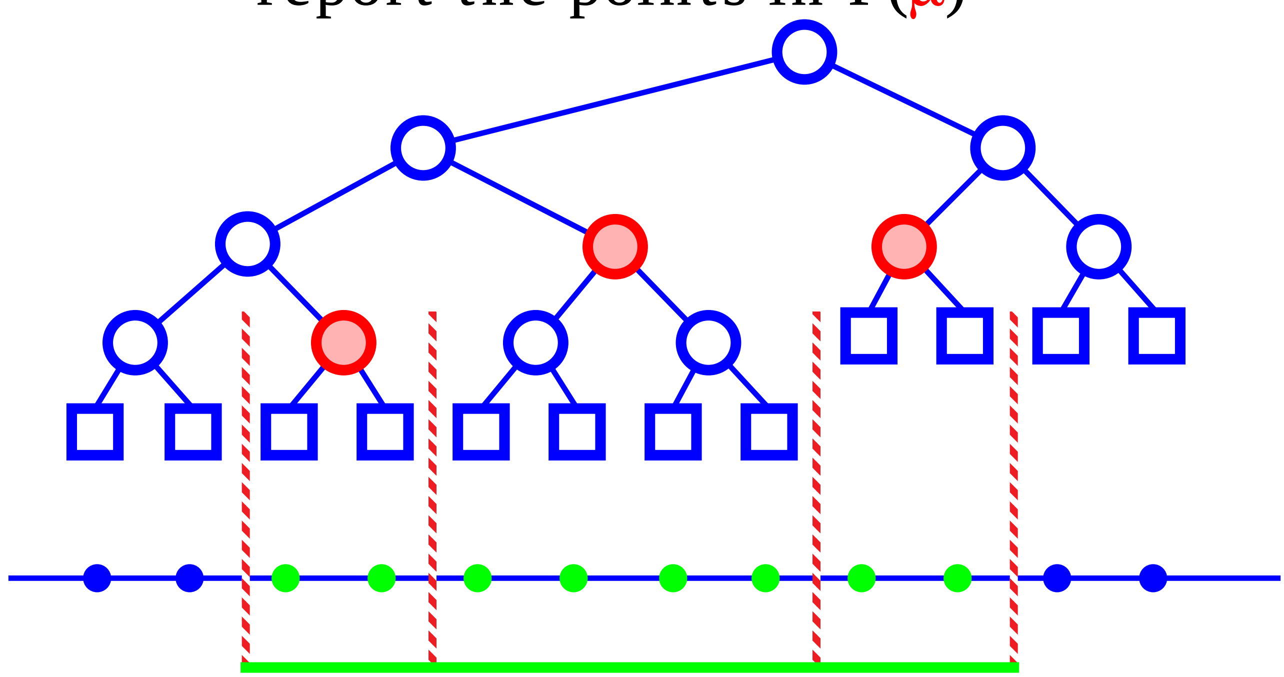
One-Dimensional Range Tree

- Alternative structure for 1-D range searching.
- More complex than a simple balanced search tree.
- Can be extended to higher dimensions.
- Range Tree: balanced search tree T
  - leaves ↔ points, sorted by x-coordinate
  - node μ ↔ subset P(μ) of the points at the leaves in the subtree of μ
- Space for n points: O(n log n).



One-Dimensional Range Queries

- An allocation node μ of T for the query range (x',x") is such that (x',x") contains P(μ) but not P(parent(μ)).
  - the allocation nodes are O(log n)
  - they have disjoint point-sets
  - the union of their point-sets is the set of points in the range (x',x")
- Query Algorithm
  - find the allocation nodes of (x',x")
  - **for each** allocation node μ
    report the points in P(μ)



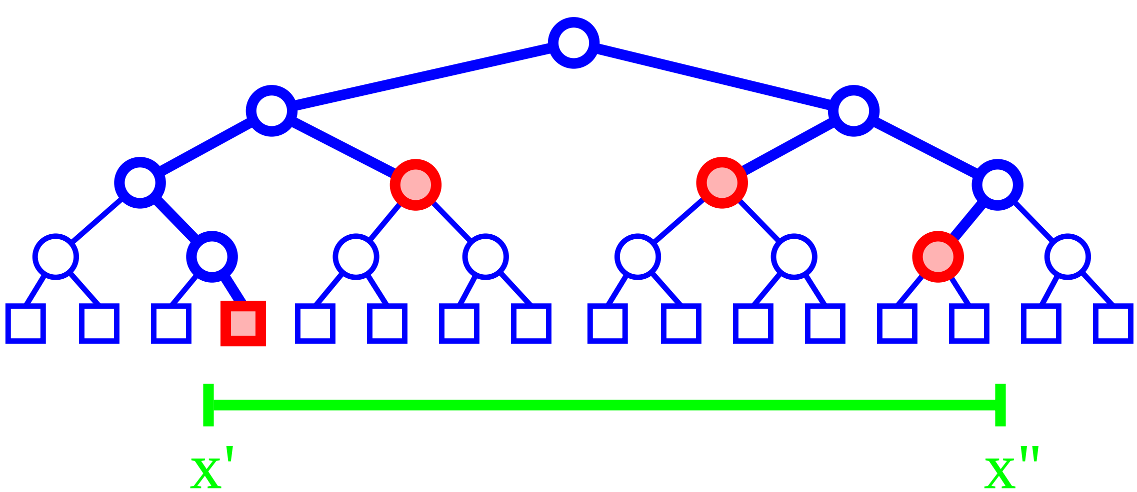
How to Find the Allocation Nodes

- Each node μ of T stores:
  min(μ): smallest x-coordinate in P(μ)
  max(μ): largest x-coordinate in P(μ)
- Find(μ): recursive procedure to mark all the allocation nodes of (x',x") in the subtree of μ

if x' ≤ min(μ) **and** x" ≥ max(μ)
  then mark μ as an allocation node
  else **if** μ is not a leaf **then**

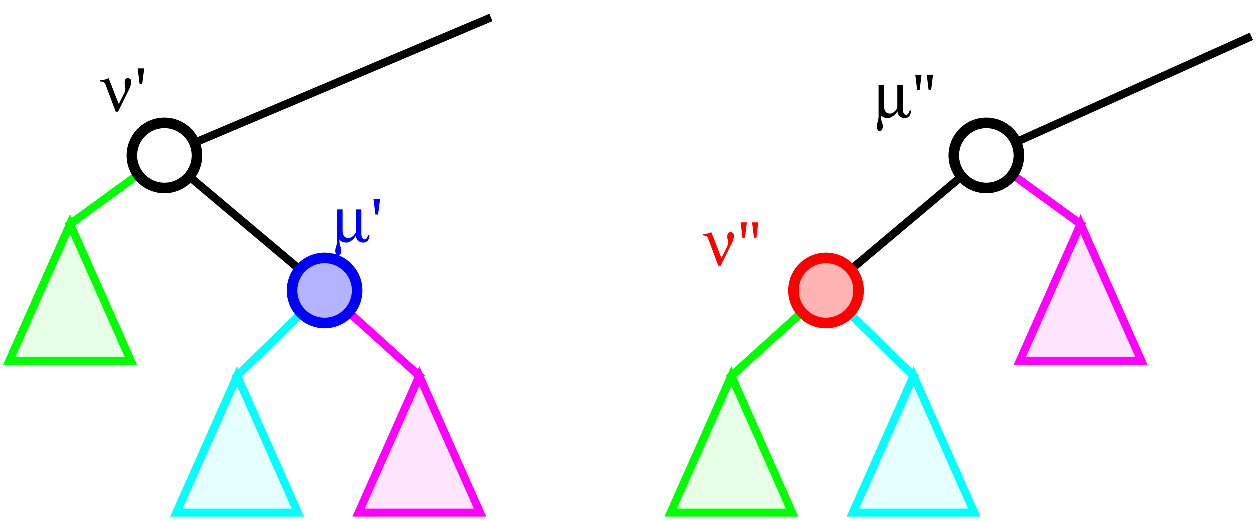
    if x' ≤ max(left(μ))
      then Find(left(μ))

    if x" ≥ min(right(μ))
      then Find(right(μ))



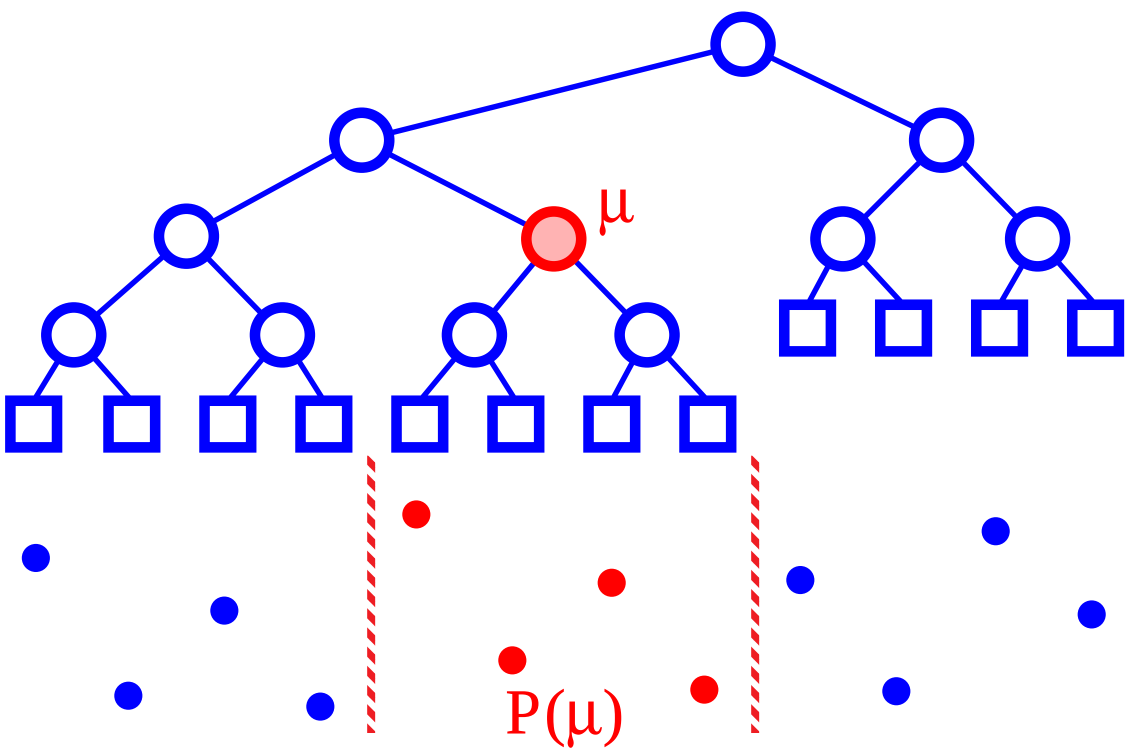
Dynamic Maintenance of the Range Tree

- Algorithm for the insertion of a point p
  - create a new leaf λ for p in T
  - rebalance T by means of rotations
  - **for** each ancestor μ of λ **do**
    insert p in the set P(μ)
- In a rotation, we need to perform split/ splice operations on the point-sets stored at the nodes involved in the rotation.
- We use a red-black tree for T, and balanced trees for the point sets.
- Insertion time: $O(\log^2 n)$. Similarly for deletions.



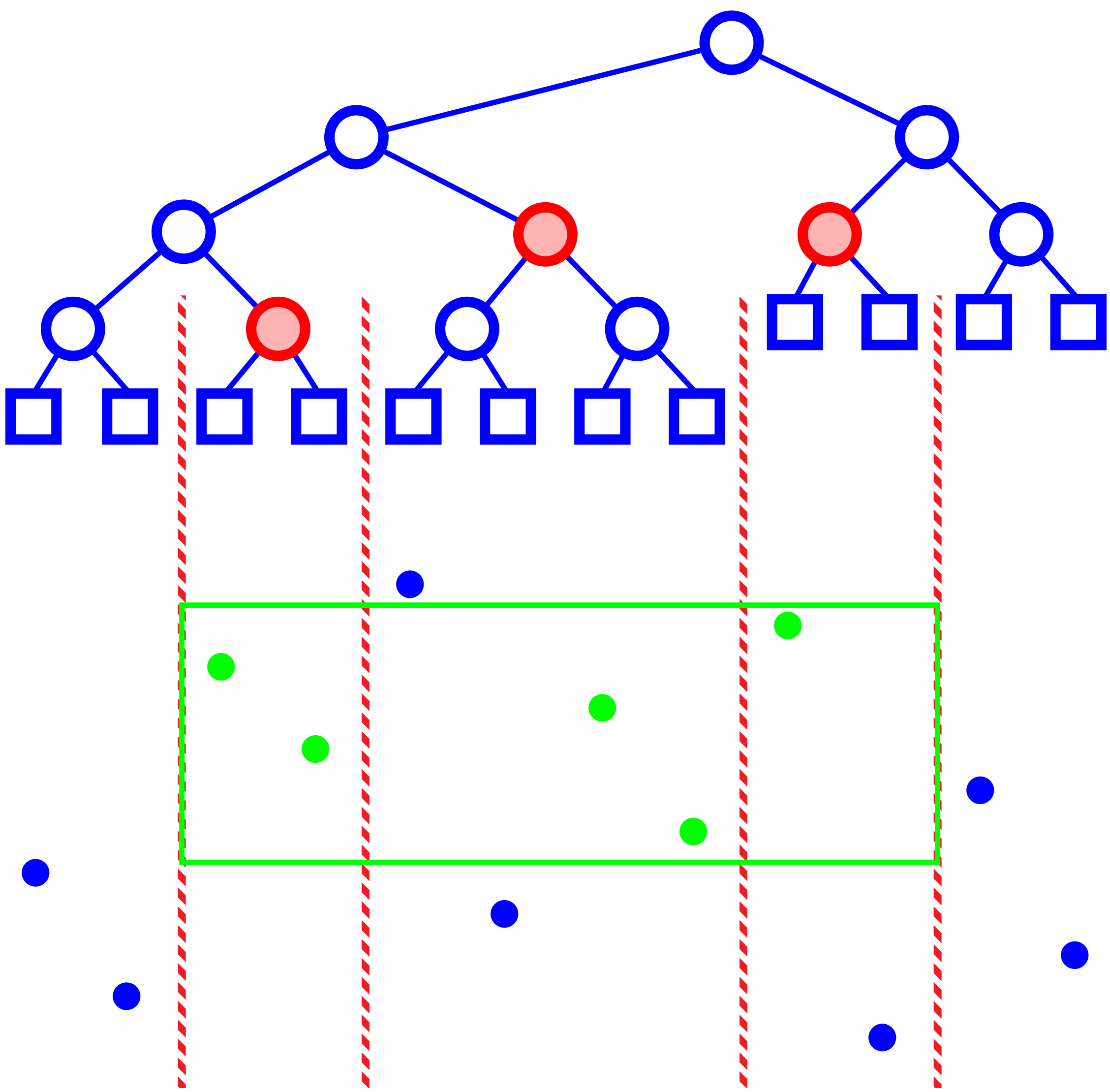
Two-Dimensional Range Searching

- 2-D Range-Tree, a two level structure
- Primary structure: a 1-D range tree T based on the x-coordinates of the points
  - leaves ↔ points, sorted by x-coordinate
  - node μ ↔ subset P(μ) of the points at the leaves in the subtree of μ
- Secondary structure for node μ:
  - Data structure for 1-D range searching by y-coordinate in the set P(μ) (either a 1-D range tree or a balanced tree)



Two-Dimensional Range Queries with the 2-D Range-Tree

- Query Algorithm for range $r = (x',x'') \times (y',y'')$
  - find the allocation nodes of (x',x'')
  - **for each** allocation node μ
    perform a 1-D range query for range (y',y'') in the secondary structure of μ



Space and Query Time

- The space used for n points depends on the secondary data structures:
  - $O(n \log^2 n)$ space with 1-D range trees
  - $O(n \log n)$ with balanced trees
- Query time for a 2-D range query:
  - $O(\log n)$ time to find the allocation nodes
  - Time to perform a 1-D range query at allocation node μ: $O(\log n + k_\mu)$, where k_μ points are reported
  - Total time: $\sum_\mu (\log n + k_\mu) = O(\log^2 n + k)$

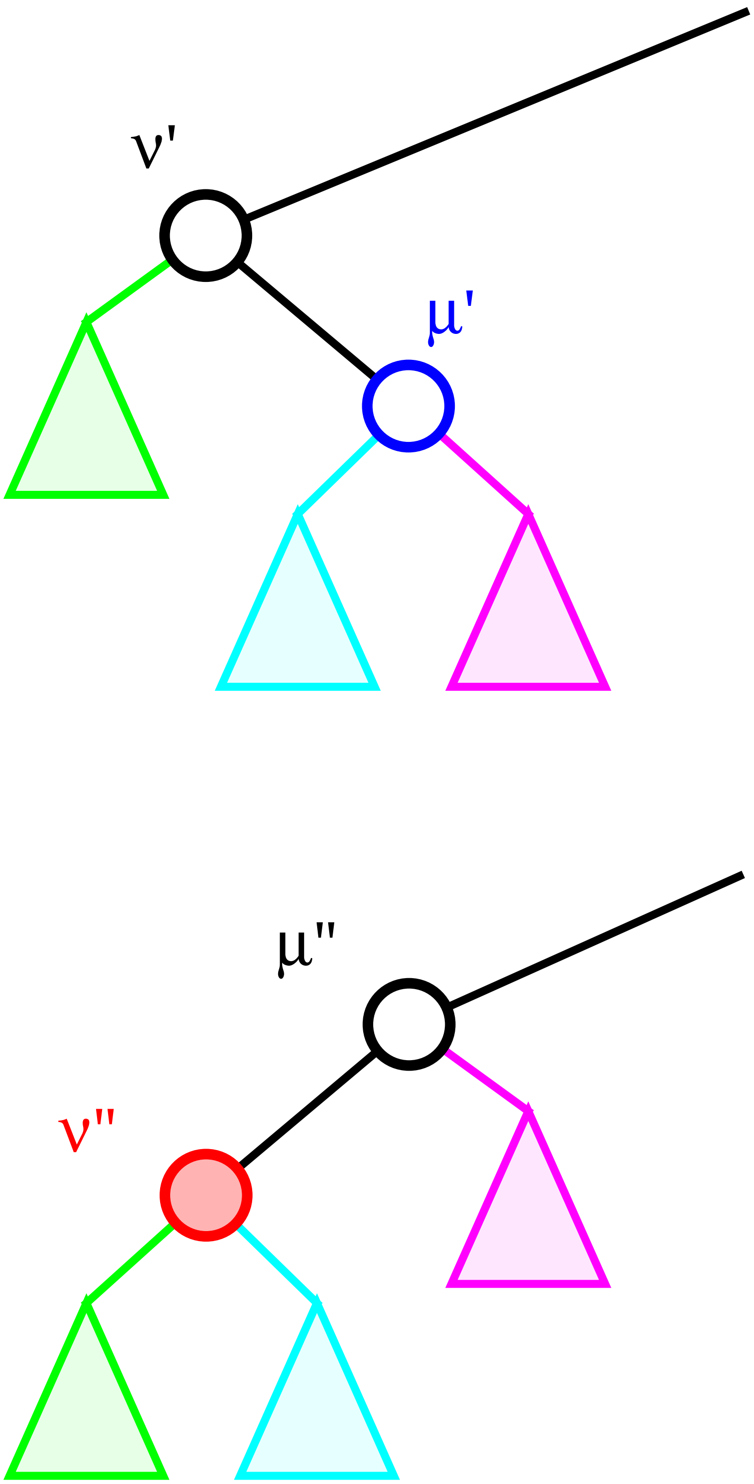
Exercises

- * Show how to perform 2-D range counting queries in time $O(\log^2 n)$.
- ** Give worst-case examples for the space
- *** Extend the range tree to d dimensions: show how to obtain $O(n \log^{d-1} n)$ space and $O(\log^d n + k)$ query time.

Dynamic Maintenance of the Range Tree

- Algorithm for the insertion of a point p
  - create a new leaf λ for p in T
  - rebalance T by means of rotations
  - **for** each ancestor μ of λ **do**
    insert p in the secondary data
    structure of μ
- When performing a rotation, we rebuild from scratch the secondary data structure of the node that becomes child (there seems to be nothing better to do).
- The cost of a rotation at a node μ is $O(|P(\mu)|) = O(\#\text{leaves in subtree of } \mu)$
- By realizing T as a BB[α]-tree, the amortized rebalancing time is O(log n).
- The total insertion time is dominated by the for-loop and is $O(\log^2 n)$ amortized.
- Similar considerations hold for deletion.

Rotation in a 2-D Range Tree



- The secondary data structure of μ" is the same as the one of ν'.
- The secondary data structure of ν" needs to be constructed.
- The secondary data structure of μ' needs to be discarded.

Summary of Two-Dimensional Range Tree

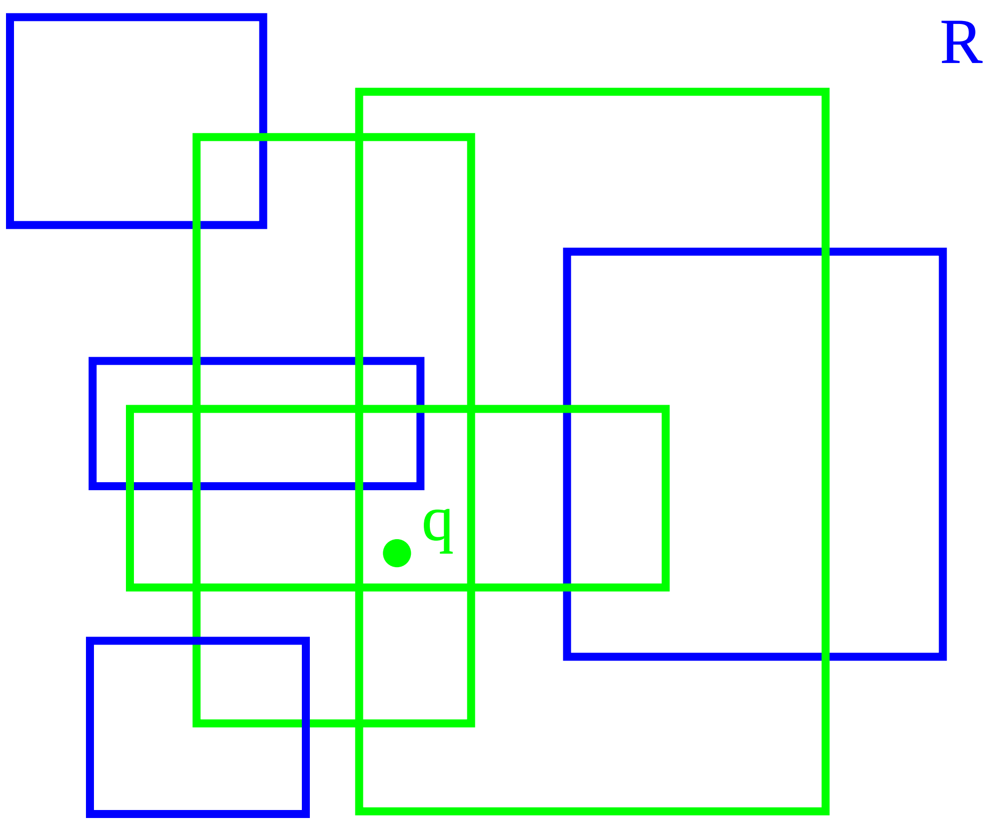
- Two-level tree structure (RR-tree)
- Reduces 2-D range queries to a collection of $O(\log n)$ 1-D range queries
- $O(n \log n)$ space
- $O(\log^2 n + k)$ query time
- $O(\log^2 n)$ amortized update time

Exercise

- *** Modify the range-tree to achieve query time $O(\log n + k)$.

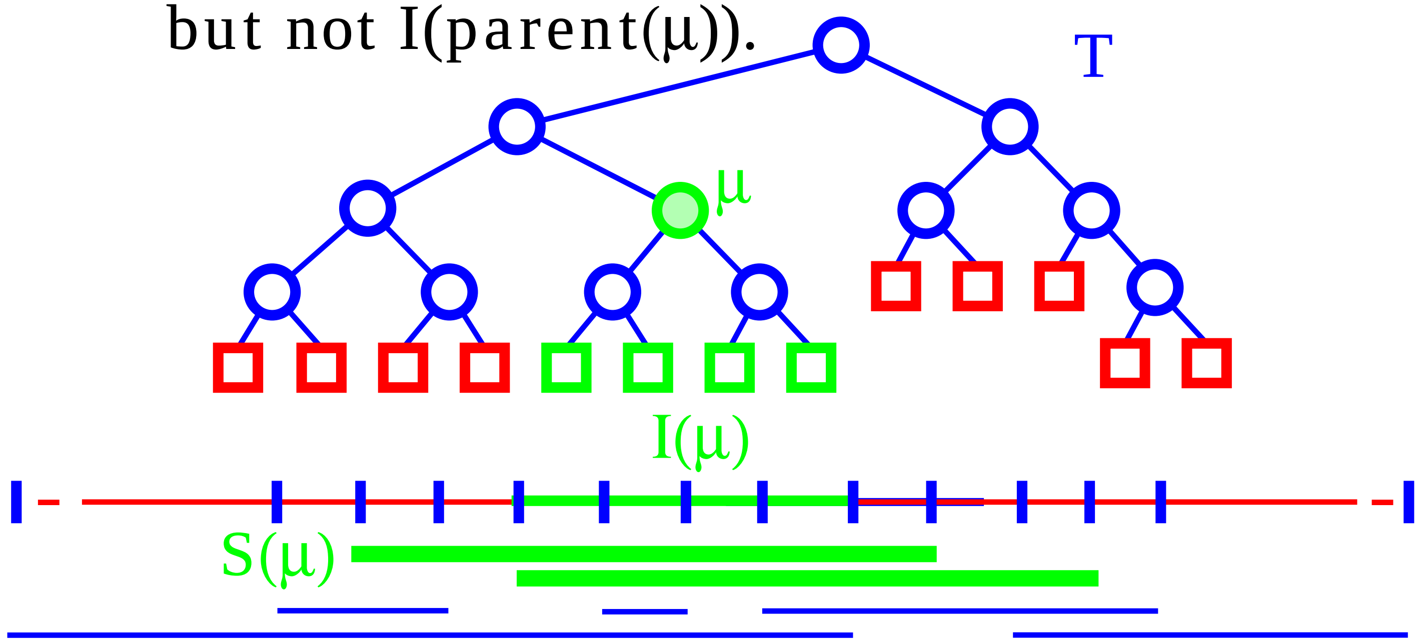
Point Enclosure

- Set R of orthogonal ranges in E^d
- Point Enclosure Query: given a query point q, report the ranges of R containing q.
- Dual of the range searching problem.
- For d=1, R is a set of intervals.
- For d=2, R is a set of rectangles.



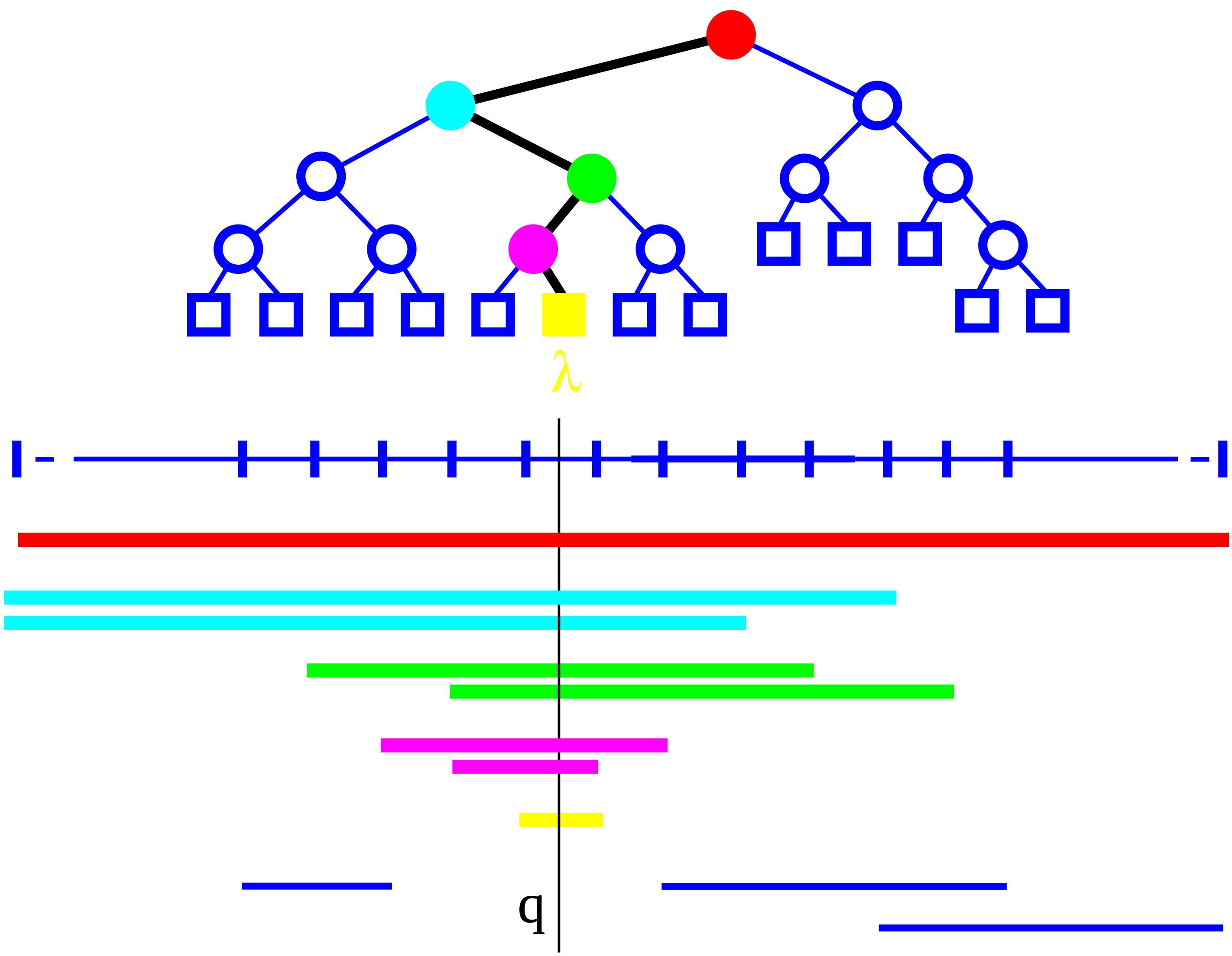
One-Dimensional Point Enclosure

- Let S be a set of segments (intervals), and X the set of segment endpoints plus ±∞.
- Segment-tree T for S: a two-level structure
- Primary structure: balanced tree T for X
  - leaves ↔ elementary intervals induced by the points of X
  - node μ ↔ x-coordinate x(μ) and interval I(μ) formed by the union of the intervals at the leaves in the subtree of μ
- Secondary structure of a node μ:
  - set S(μ) of the segments that contain I(μ) but not I(parent(μ)).



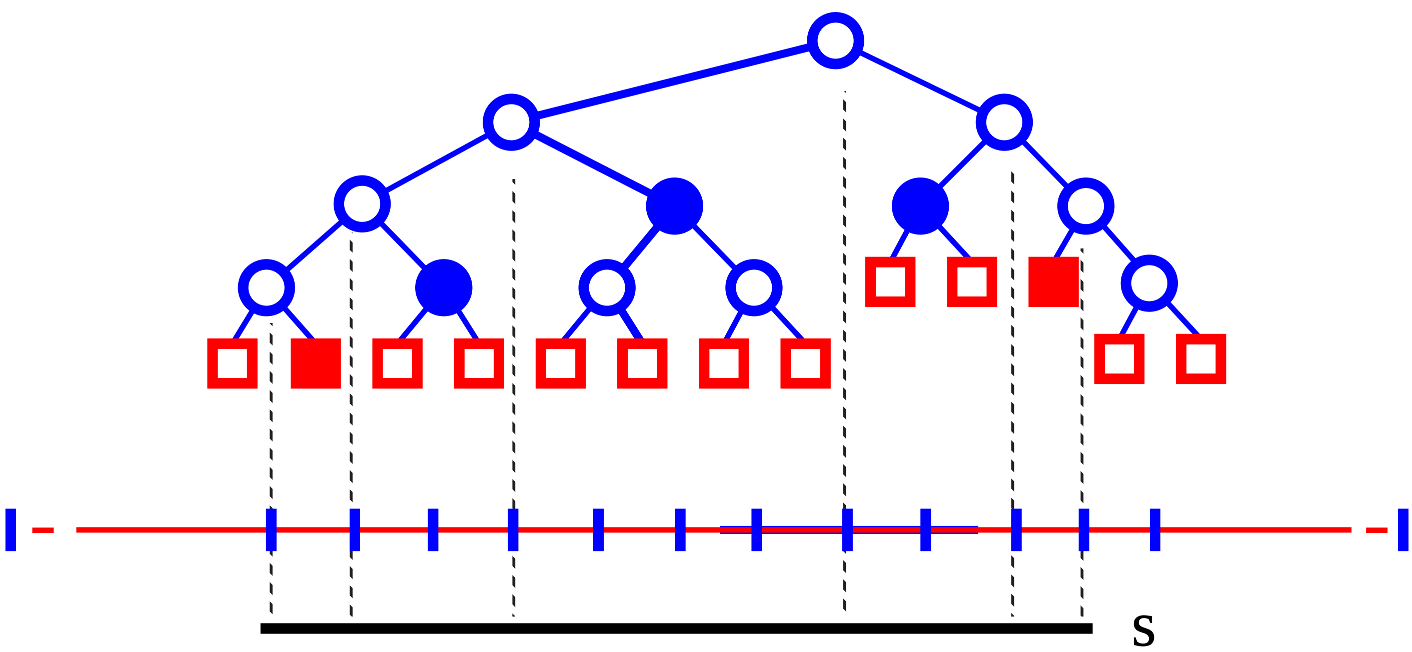
Point Enclosure Queries with the Segment Tree

- Find the elementary interval I(λ) containing the query point q by searching for q in the primary structure of T
- For each node μ in the path from λ to the root, report the segments in S(μ)



Complexity of One-Dimensional Point Enclosure

- A node μ is an allocation node of segment s if S(μ) contains s.
- Each segment s has O(log n) allocation nodes



- Space used by the segment-tree: O(n log n)
- Query time: O(log n + k)

Exercises

- * Show how to perform point enclosure counting queries in O(log n) time using O(n) space.
- ** Discuss special cases that have not been addressed (e.g., a query point is a segment endpoint).
- ** Dynamize the segment tree, i.e., show how to support insertions and deletions of segments.
- ** Give an efficient data structure to perform 1-D segment intersection queries. (Given a set of segments on a line, report the segments intersecting a query segment.)

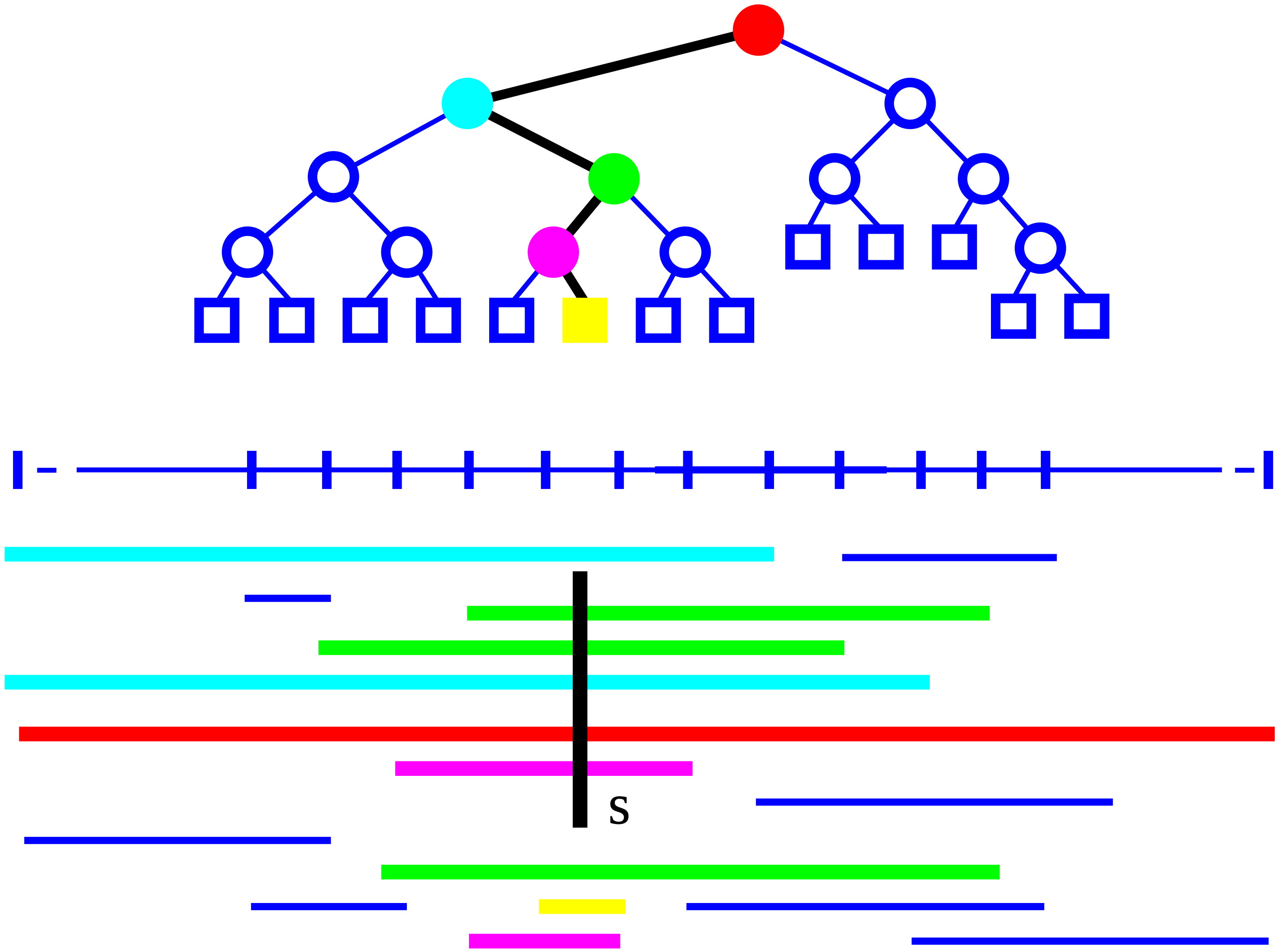
Two-Dimensional Point Enclosure

- We represent a set of rectangles with sides parallel to the axes by means of a two-level structure (SS-tree).
- Primary structure:
  - a segment tree T for the x-intervals of the rectangles of R
- Secondary structure of a node μ:
  - a 1-D point enclosure data structure for the y-intervals of the rectangles in S(μ) (another segment tree)
- Space for n rectangles: $O(n \log^2 n)$
- Query algorithm for point q
  - Locate q in T, this gives a leaf λ whose elementary vertical strip contains q
  - Perform 1-D point enclosure queries in the secondary structures of the nodes on the path from λ to the root
- Query time: $O(\log^2 n + k)$

Orthogonal Segment Intersection

- S: set of n horizontal segments in the plane
- Orthogonal Segment Intersection Query: given a vertical query segment s, report the segments of S intersected by s.
- Two data structures for this problem:
  - SR-tree: the segments of S are stored in an x-based segment-tree T' . The secondary structures support 1-D range searching on the y-coordinate. A segment intersection query corresponds to performing O(log n) 1-D range queries along a root-to-leaf path in T'.
  - RS-tree: the segments of S are stored in a y-based range-tree T''. The secondary structures support 1-D point enclosure queries on the x-coordinate. A segment intersection query corresponds to performing O(log n) 1-D point enclosure queries at the allocation nodes of s in T''.

Example of Querying the SR-Tree

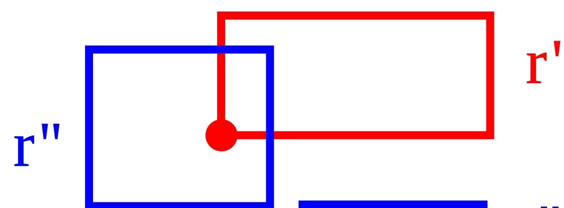


Exercises

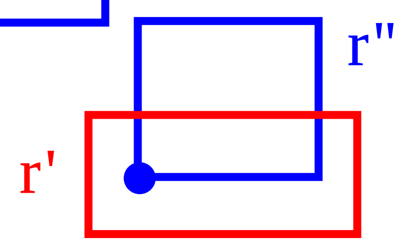
- * Determine the space requirement and query time of the SR-tree and RS-tree.
- ** Dynamize the SR-tree and the RS-tree.
- ** Show how to perform vertical "ray shooting" queries for horizontal segments.

Orthogonal Rectangle Intersection

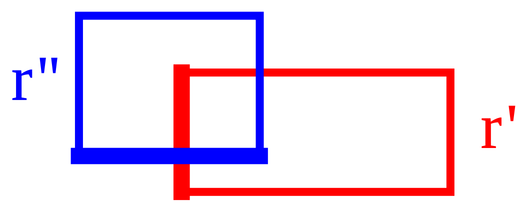
- Let R be a set of n rectangles with sides parallel to the axes
- Orthogonal Rectangle Intersection Query: given a query rectangle r, determine the rectangles of R intersected by r.
- Rectangles r' and r" intersect iff one of the following mutually exclusive cases arises:
  - the bottom-left corner of r' is in r"



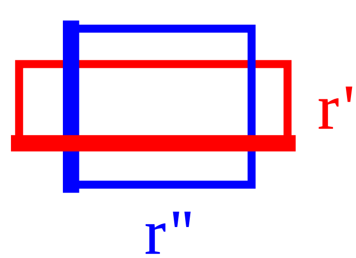
  - the bottom-left corner of r" is in r'



  - the left side of r' intersects the bottom side of r"



  - the left side of r" intersects the bottom side of r'

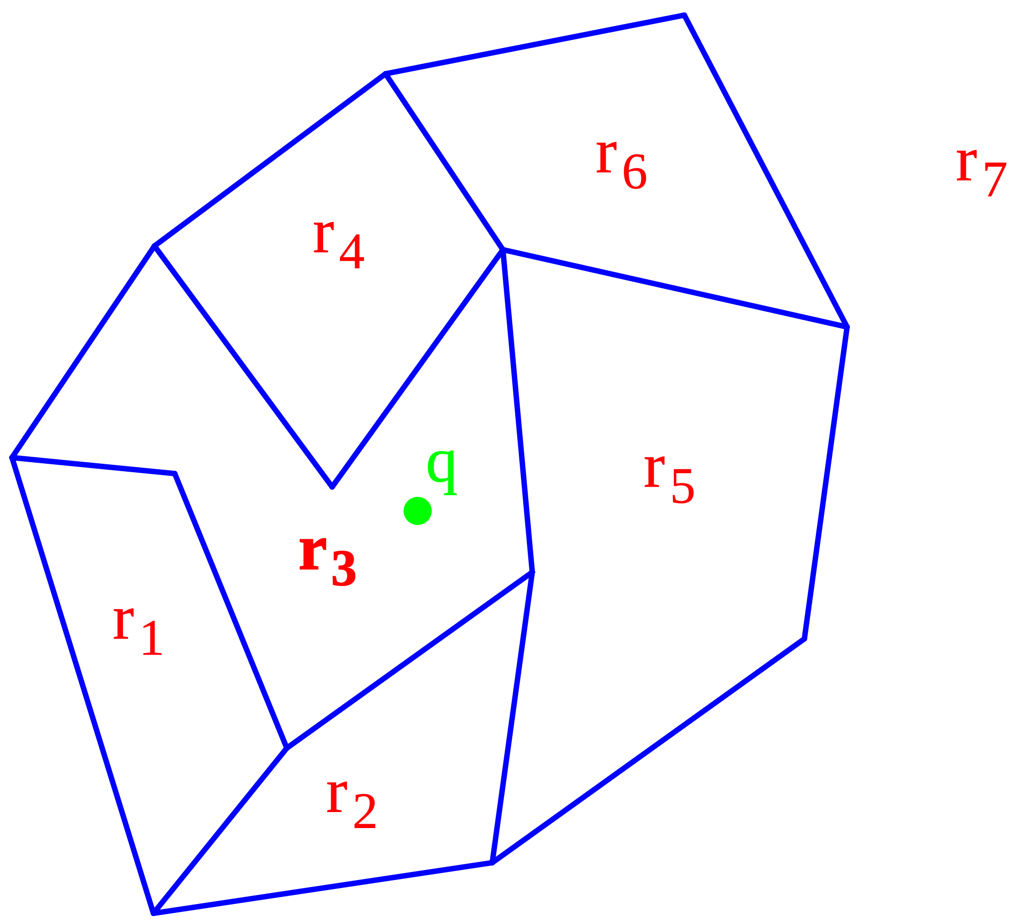


Orthogonal Rectangle Intersection

- We can perform an orthogonal rectangle intersection query as follows:
  - range search query for the bottom-left corners of the rectangles of R contained in r
  - point enclosure query for the rectangles of R containing the bottom-left corner of r
  - orthogonal segment intersection query for the bottom sides of the rectangles of R intersected by the left side of r
  - orthogonal segment intersection query for the left sides of the rectangles of R intersected by the bottom side of r
- We can use a data structure consisting of four components: RR, SS, RS, and RS tress.
- Orthogonal rectangle intersection queries in d dimensions can be performed with a data structure consisting of the d-level trees given by the symbolic expansion of $(R + S)^d$

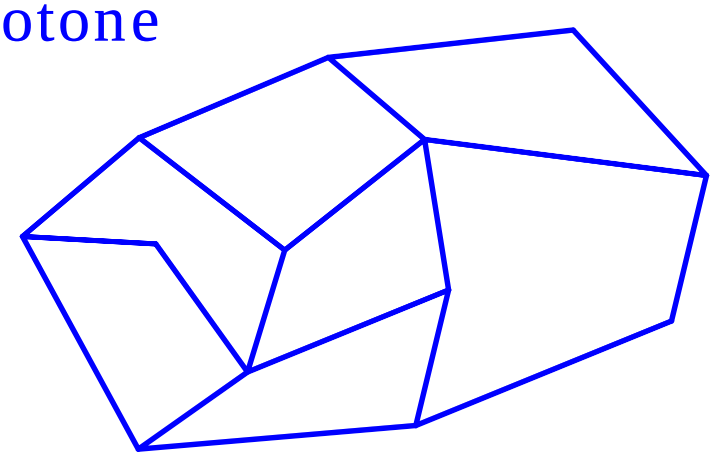
Planar Point Location

- Subdivision S of the plane into polygonal regions, induced by the vertices and edges of a planar graph
- Find the region containing a query point q
- Fundamental two-dimensional searching problem

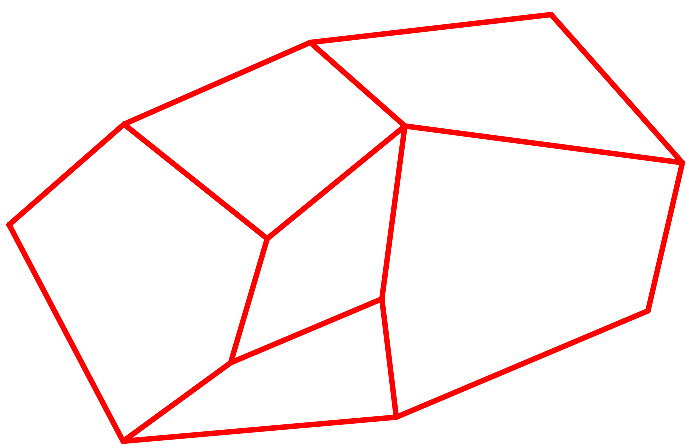


Types of Planar Subdivisions

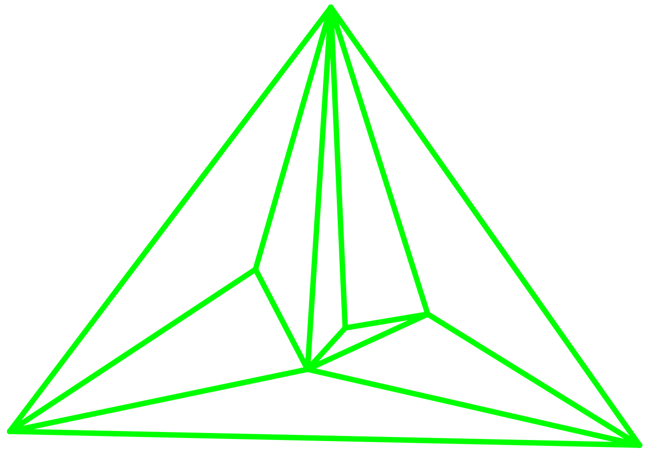
- Monotone



- Convex



- Triangulation



Static Point Location

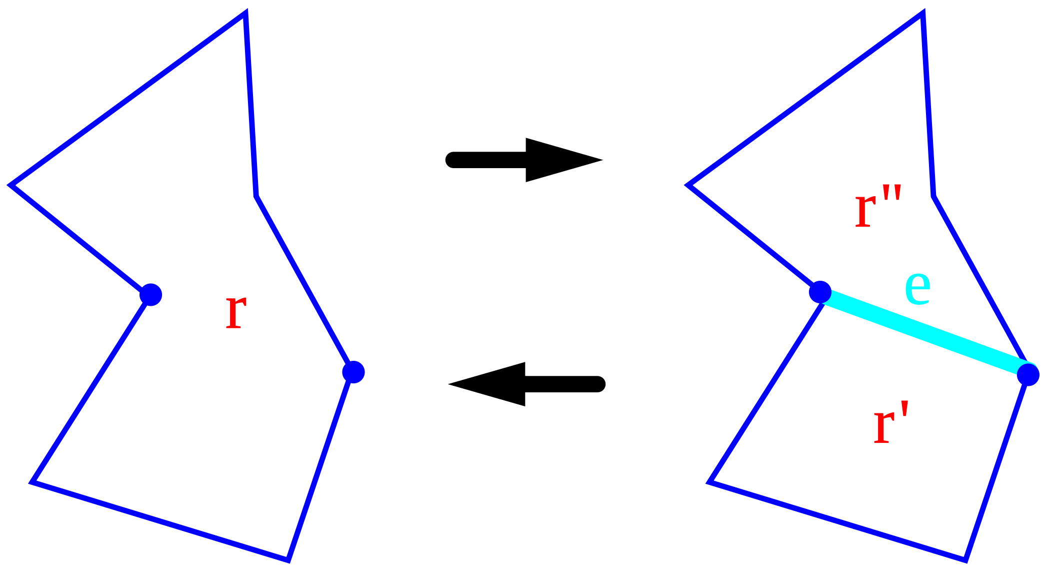
- Preprocess the subdivision
- Answer ***on-line queries***
  (query points are not known in advance)
- Performance measures:
  - space
  - query time
  - preprocessing time

Dynamic Point Location

- Perform an ***on-line*** sequence of intermixed ***queries*** and ***updates*** (insertion and deletion of vertices and edges)
- Performance measures:
  - space
  - query time
  - insertion/deletion time

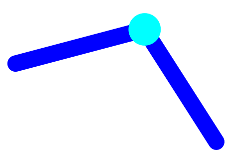
Update Operations for Planar Subdivisions

- Insert/Delete an edge



- Insert/Delete a chain of edges
- Insert/Delete an isolated vertex
- Insert/Delete a vertex on an edge

- Translate a vertex



Best Results for Static Point Location

[Kirkpatrick 83, Edelsbrunner Guibas Stolfi 86, Sarnak Tarjan 86]

- O(n) space
- O(log n) query time
- O(n log n) preprocessing time

Best Results for Dynamic Point Location

[Goodrich Tamassia 91] monotone subdiv.

[Cheng Janardan 90] connected subdiv.

- O(n) space, $O(\log^2 n)$ query time, O(log n) update time

[Preparata Tamassia 89] convex subdiv.

[Chiang Tamassia 91] monotone subdiv.

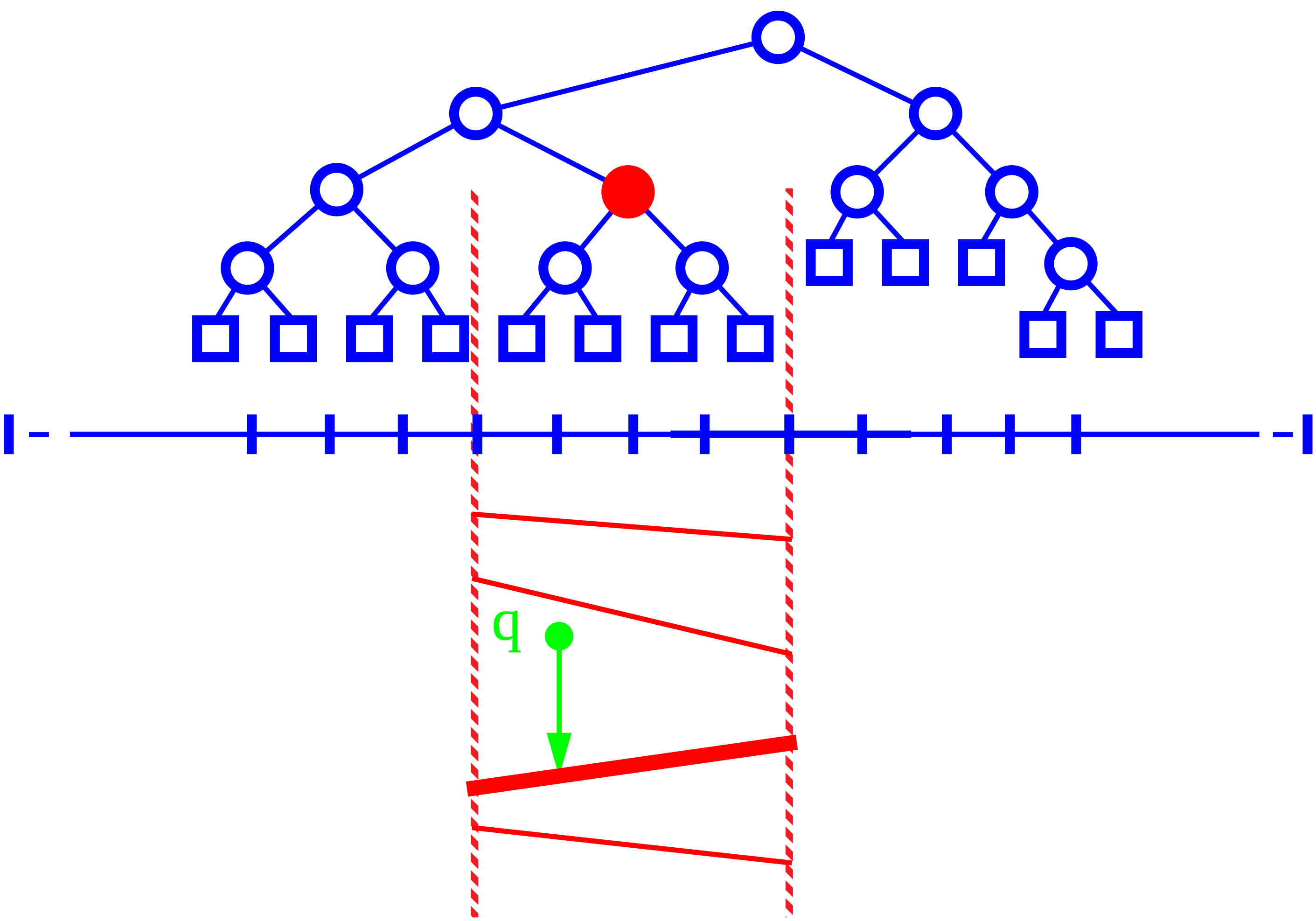
- O(n log n) space, O(log n) query time, $O(\log^2 n)$ amortized update time

[Goodrich Tamassia 91] monotone subdiv.

- O(log n loglog n) query time, O(1) amortized insertion time

Point Location with Segment Trees (Overmars, CG '85)

- Use an SR-tree for the set of edges
- Each edge stores the region above it
- The secondary structures are balanced trees that support down-shooting queries in a vertical "slab"
- O(n log n) space and $O(\log^2 n)$ query time



Exercises

- ** Show how to construct the segment-tree structure for point location in O(n log n) time
- *** Dynamize the data structure
- **** Modify the data structure to achieve O(log n) query time and O(n log n) space in a static environment

Open Problem

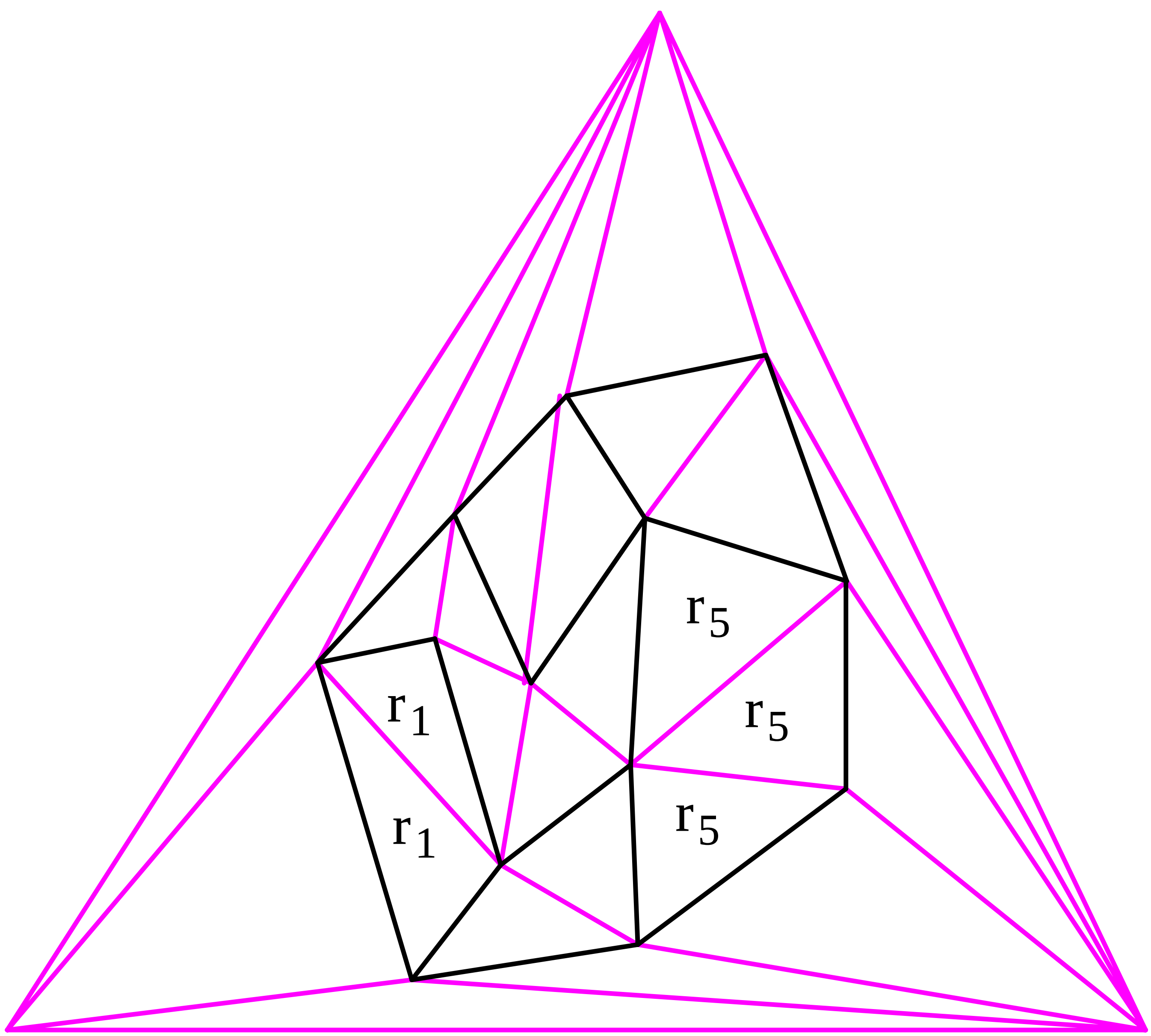
- ***** Modify the data structure to achieve O(log n log log n) query time and polylog upate time in a fully dynamic environment.

Point Location With Dynamic Trees (Goodrich-Tamassia, STOC '91)

- A new method for planar point location, based on interleaving primal and dual spanning trees
- Algorithms are relatively simple and easy to implement
- Optimal static data structure: O(n) space, O(log n) query time
- Efficient fully dynamic data structure for monotone subdivisions: O(n) space, $O(\log^2 n)$ query time, O(log n) update time
- Efficient on-line data structure for insertions: O(log n loglog n) query time, O(1) amortized insertion time
- Improved 3-dimensional point location: O(n log n) space, $O(\log^2 n)$ query time

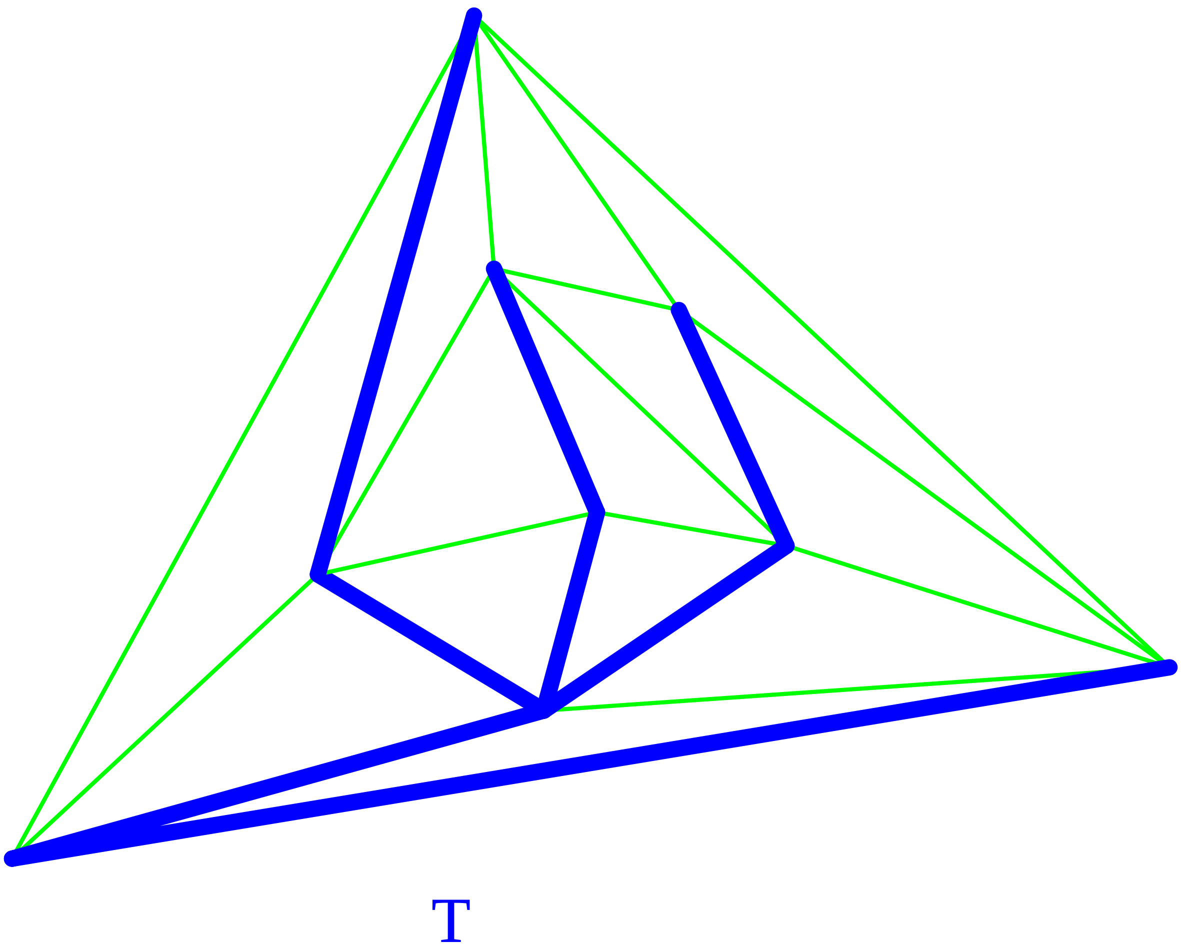
Triangulations

- A subdivision can be refined into a triangulation by adding fictitious edges, plus 3 fictitious vertices



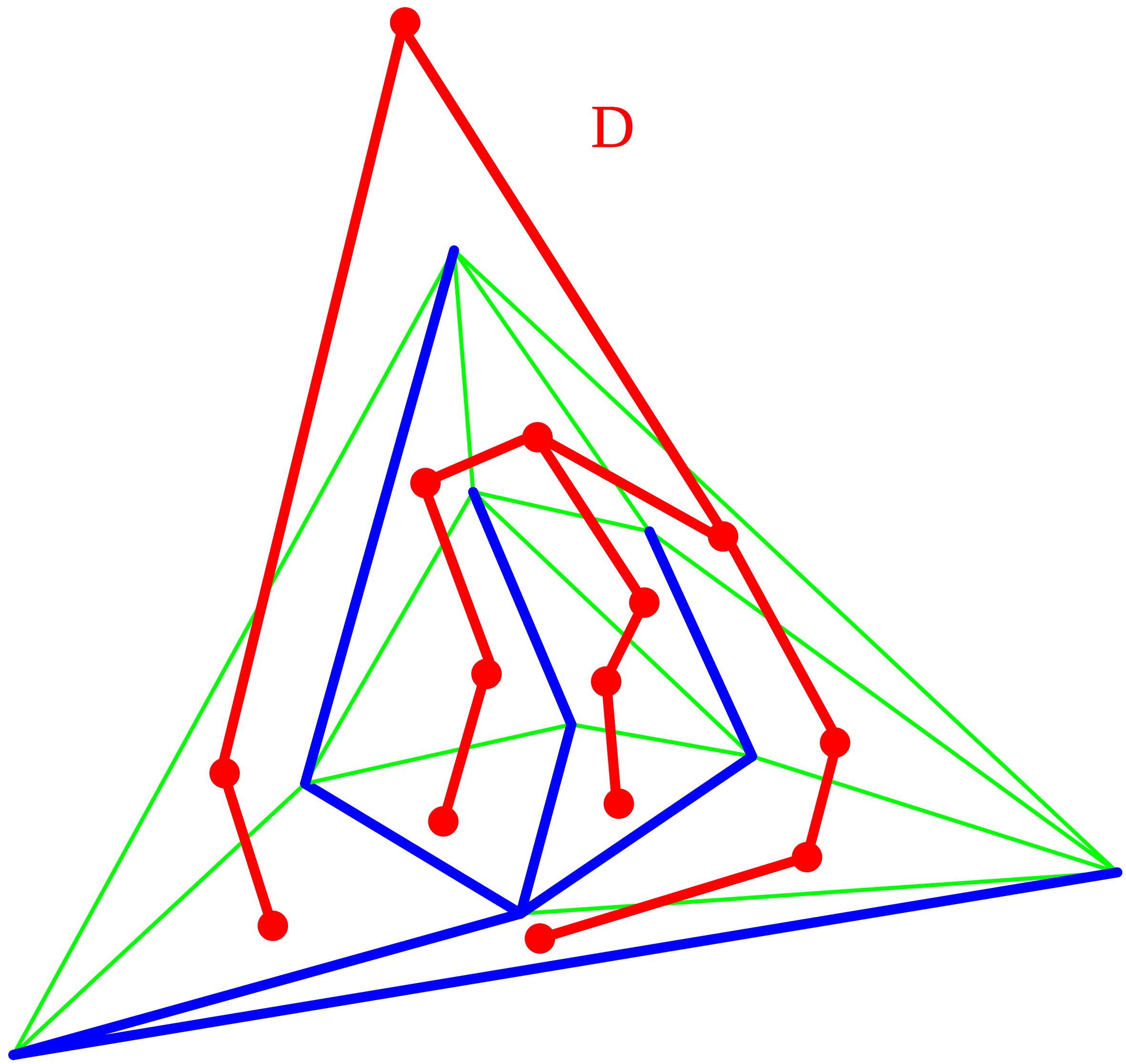
Monotone Spanning Tree

- For each vertex, select an incoming edge (incoming = incident from below)
- This yields a monotone spanning tree T of the subdivision



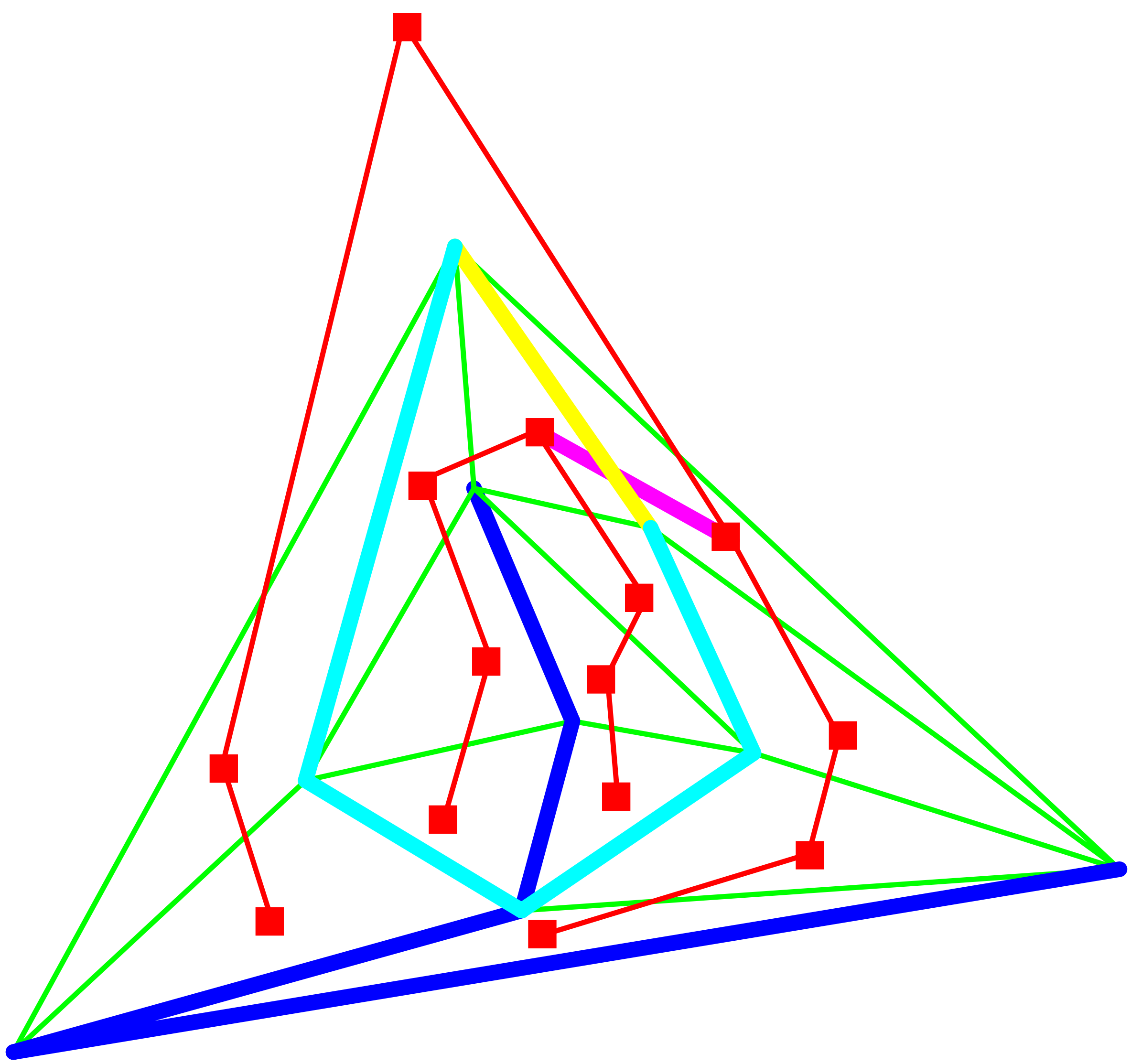
Dual Spanning Tree

- Place a dual node in every region
- For each non-tree edge, draw a dual edge
- This yields a dual tree D



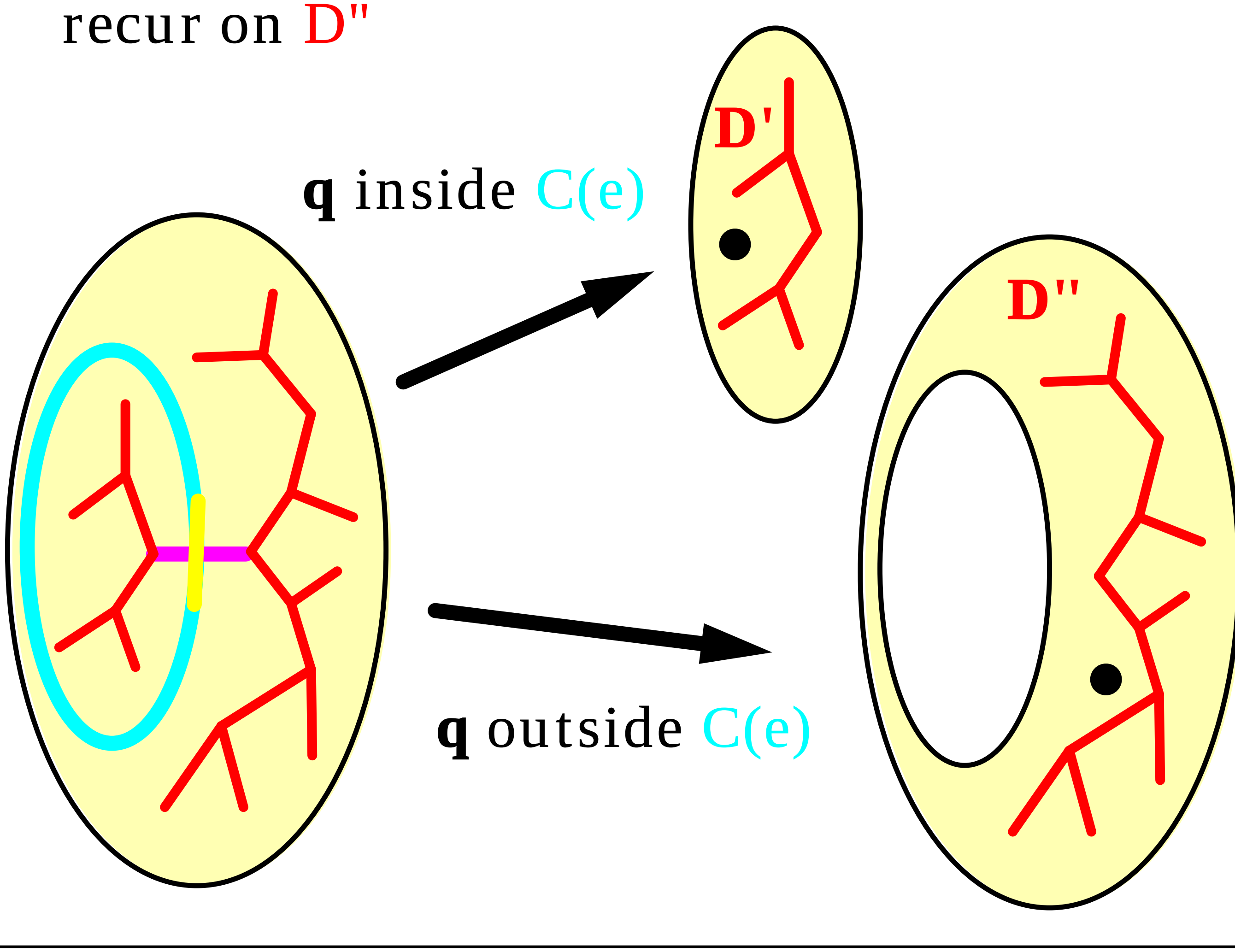
Cycles and Cuts

- Each non-tree edge
  - forms a cycle with T
  - induces a cut in D

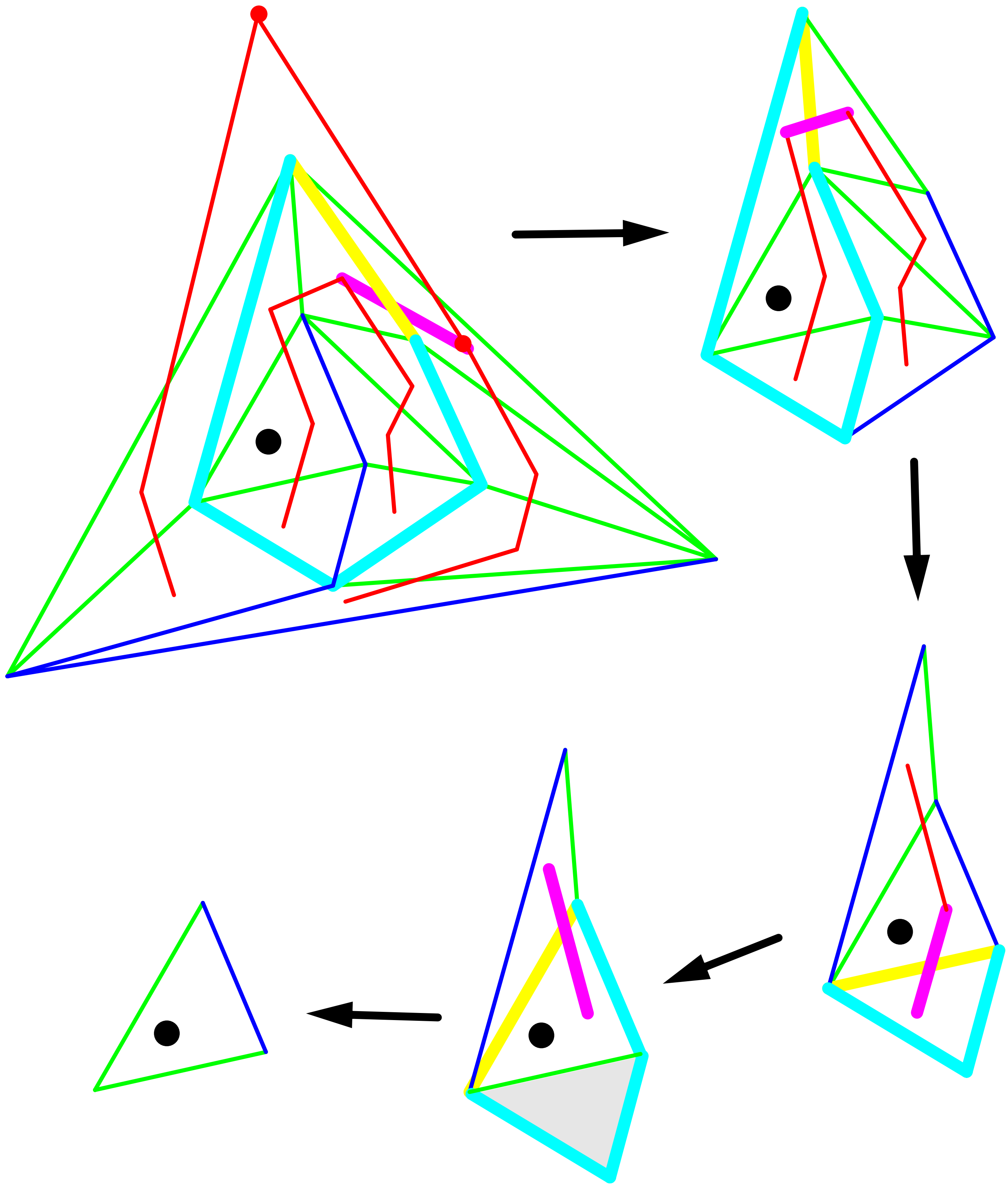


Point Location Algorithm

1. Find a centroid edge e whose cut decomposes D into subtrees D' (internal) and D" (external), each with at most 2/3 of the nodes.
2. Determine if the query point **q** is inside or outside the cycle C(e) induced e
3. If **q** is inside C(e), then recur on D', else recur on D"

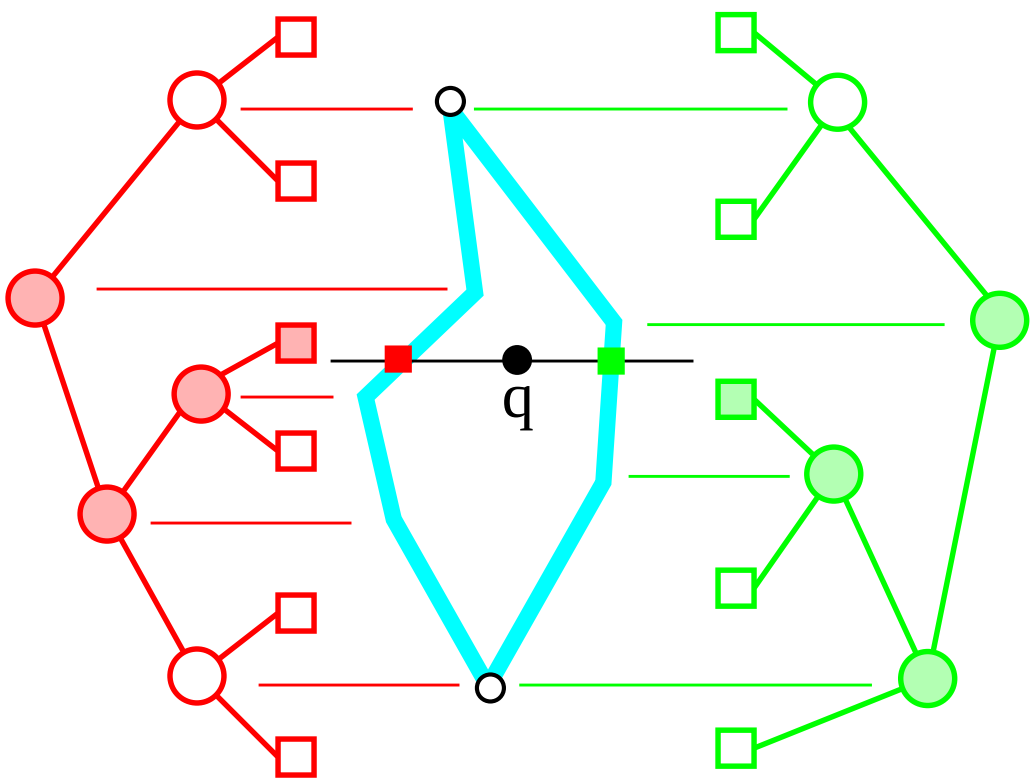


Example



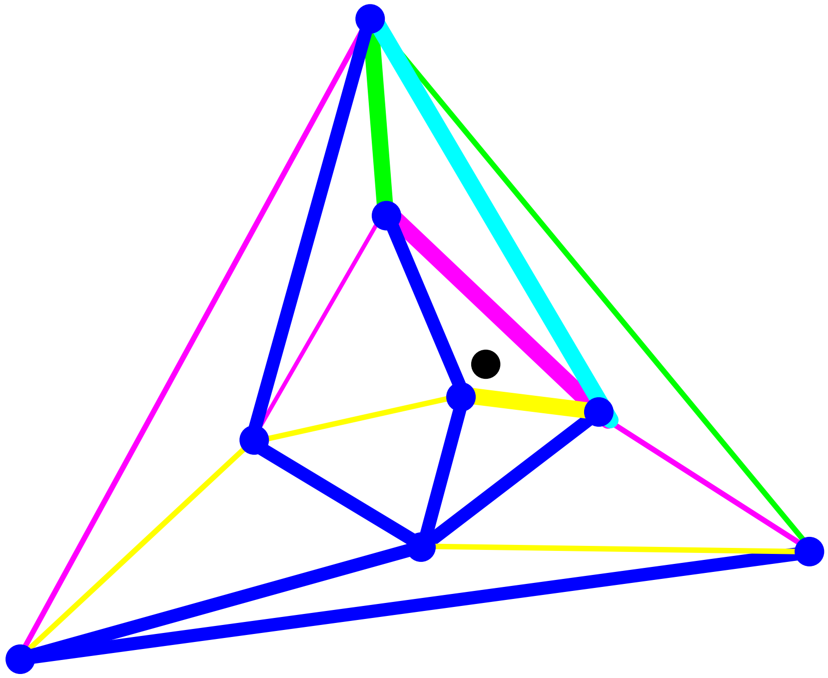
Testing if q is Inside or Outside Cycle C(e)

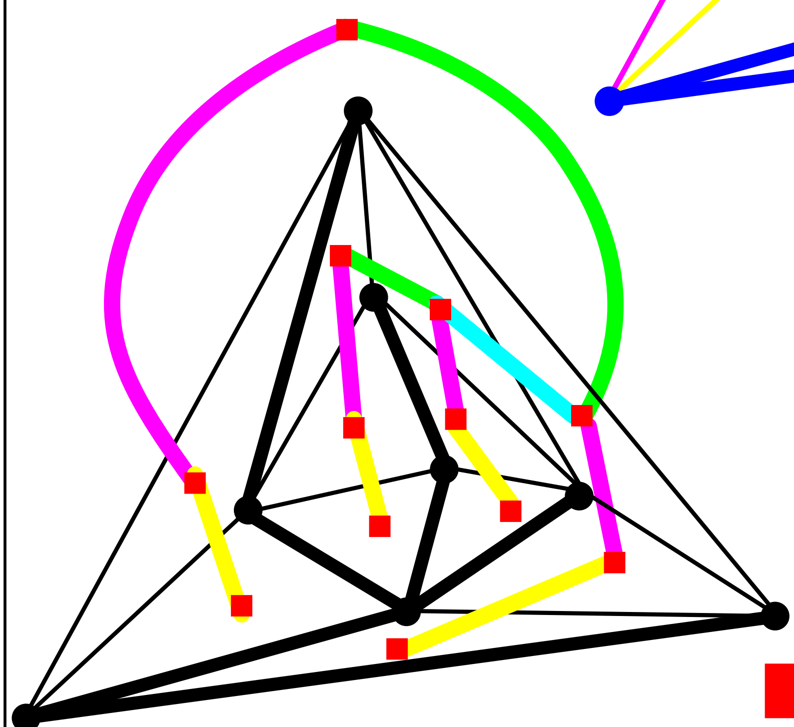
- The boundary of cycle C(e) consists of two monotone chains (L and R)
- We represent each such chain with a balanced tree
- By doing binary search on the y-coordinate of q, we determine the points of L and R in front of q in O(log n) time



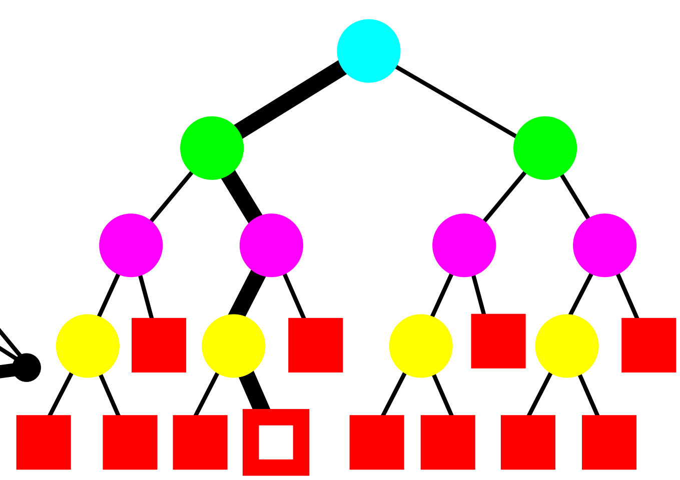
Centroid Decomposition

- Represent the recursive decomposition of the dual tree by means of a binary tree B
- A point location query traverses a root-to-leaf path in B





B



Complexity Analysis

Query Time

- The centroid decomposition tree B has 2n–5 leaves (regions)
- For each node μ of B:
  leaves(μ) < 2/3 leaves(parent(μ))
- The centroid tree has depth O(log n)
- Visiting each node takes O(log n) time
- Query time: $O(\log^2 n)$

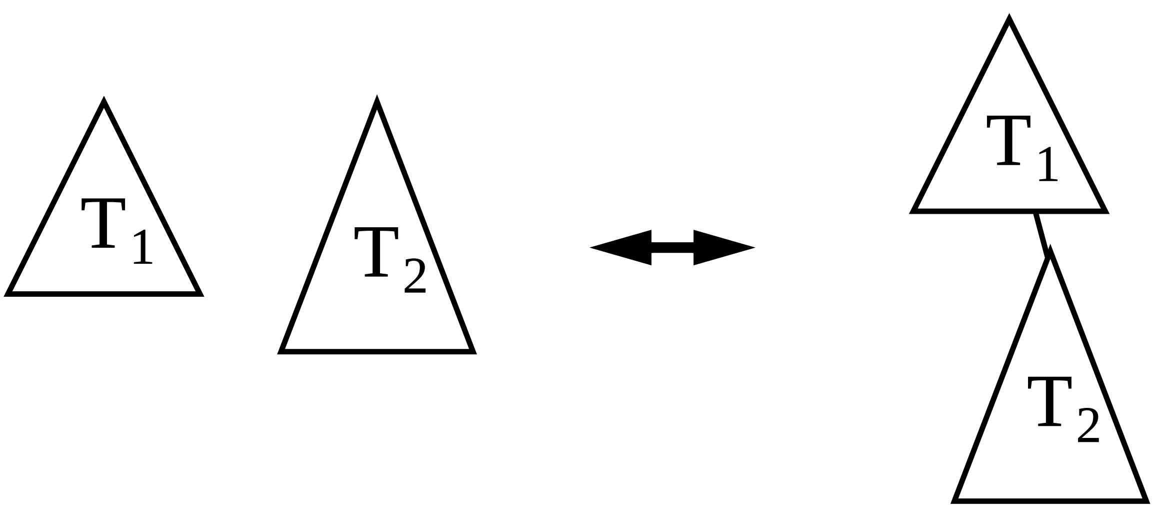
Space

- If we store at each node the corresponding cycle, we use $O(n^2)$ space
- To save space and dynamize the data structure, we use dynamic trees ...

Dynamic Trees

[Sleator Tarjan 1983]

- Data structure to represent a collection of rooted trees
- Operations:
  - Path(v): return the path from v to the root (as a balanced binary tree)
  - Link: join two trees by adding an edge
  - Cut: decompose a tree by removing an edge



Dynamic Trees and Point Location

- use dynamic trees for T and D
  - use D for finding centroid edges
  - use T for retrieving edge chains
- Space: O(n)

Query algorithm

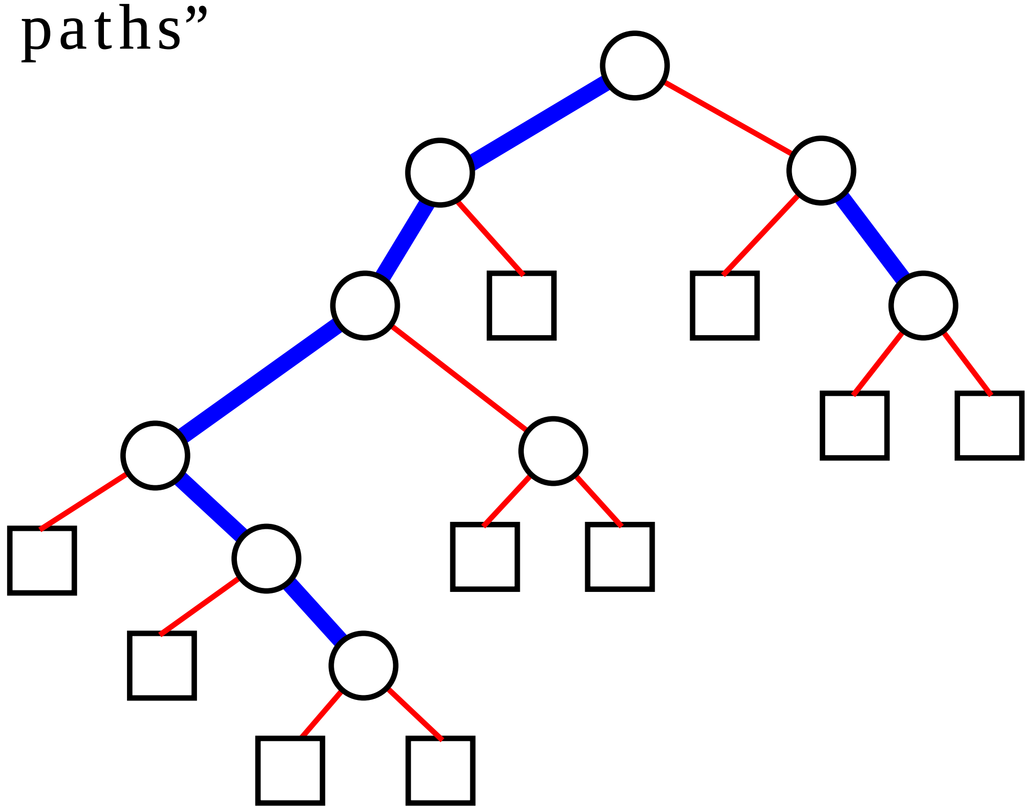
1. If D consists of a single region r, then report r and stop
2. Find a centroid edge e=(u,v)
3. Cut D at edge e into D' (internal) and D" (external)
4. L(e) = Path(u)
5. R(e) = splice(e,Path(v))
6. If **q** is inside, L(e) $\cup$ R(e), then recur on D', else recur on D"

Path Decomposition

- partition the edges into light and heavy:

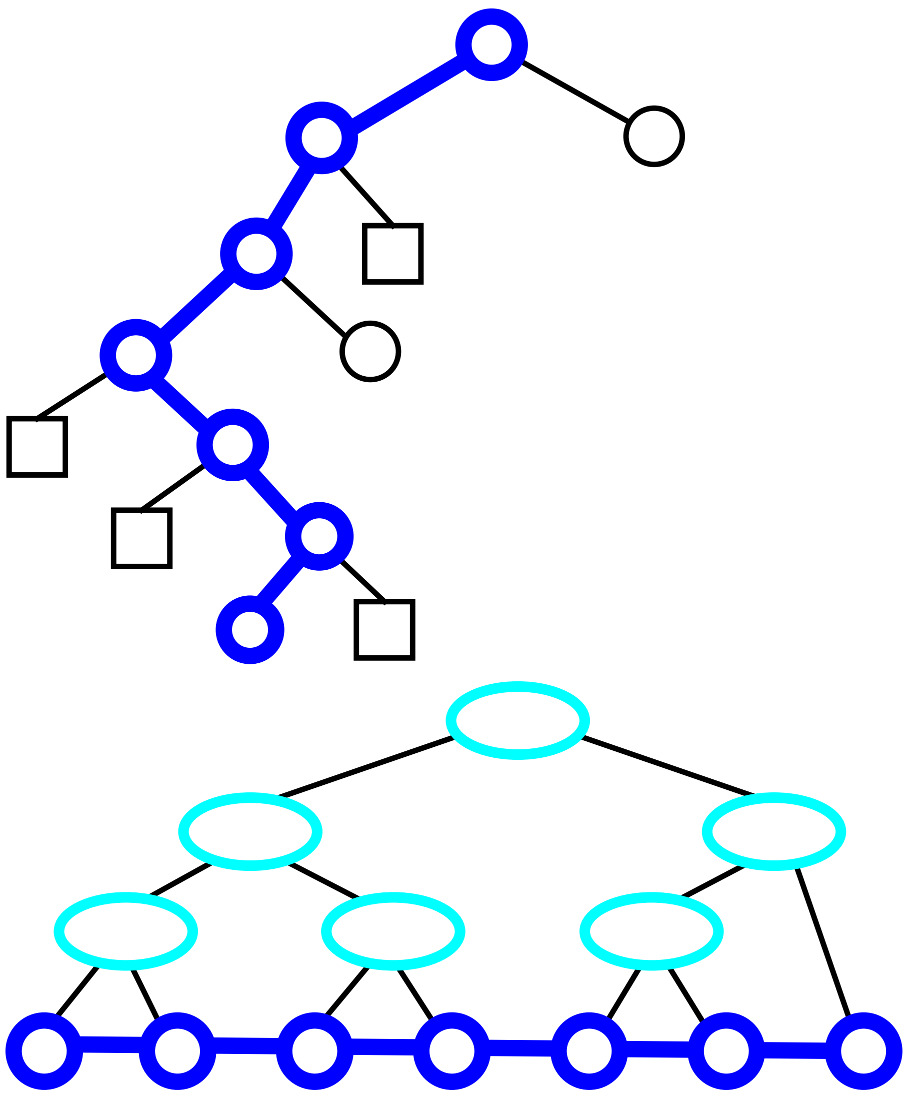
  heavy edge: size(child) > size(parent) / 2

  light edge: size(child) ≤ size(parent) / 2
- heavy edges form disjoint ***solid paths***
- going from a leaf to the root we traverse at most log n light edges
- "removing light edges decomposes an unbalanced tree into a balanced tree of solid paths"



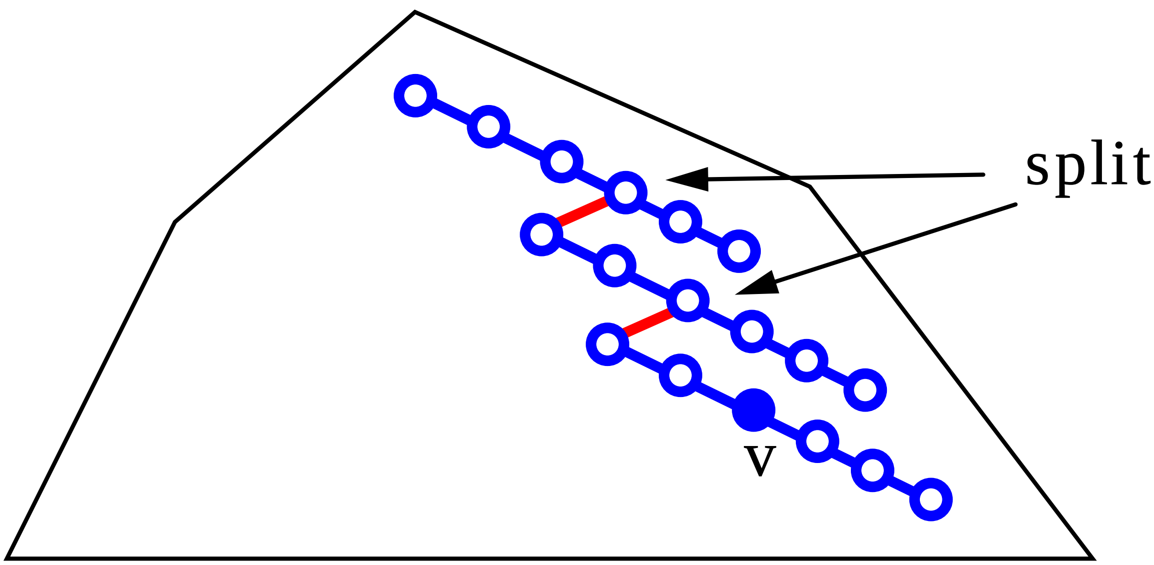
Representing a Solid Path

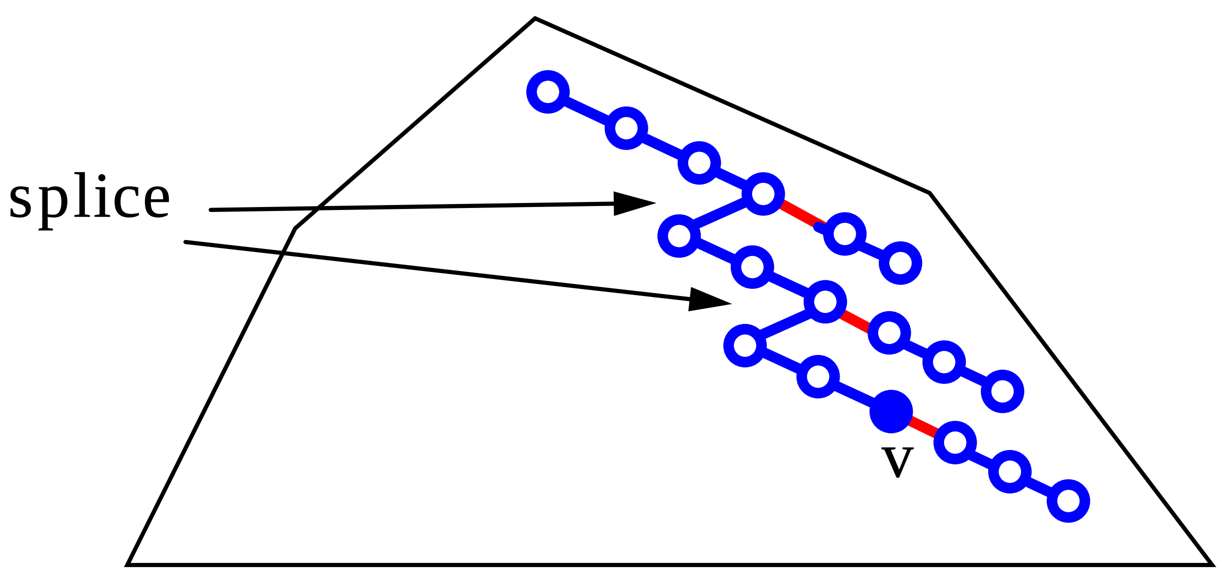
- we represent each solid path P by means of a balanced binary tree, called path-tree
  - leaf ↔ node of P
  - internal node ↔ subpath of P
- solid paths can be split and spliced in time O(log n)



Operation Path(v)

- Construct the path from v to the root by splitting and splicing O(log n) solid paths



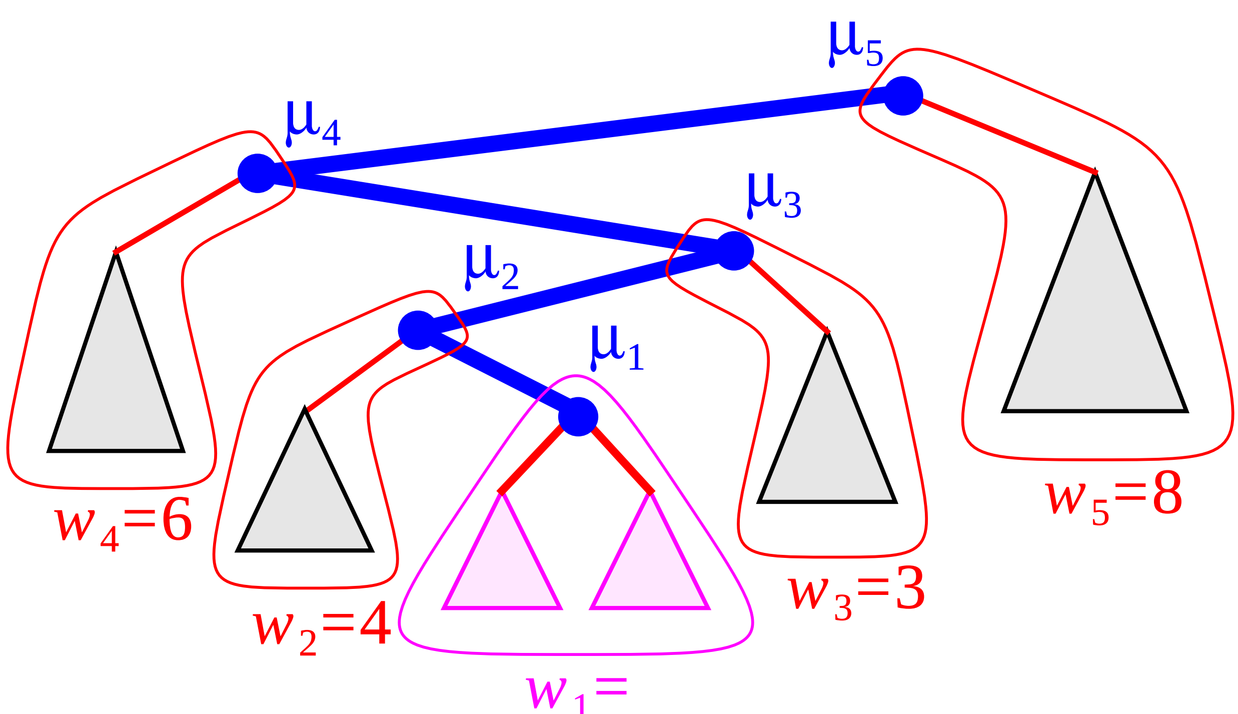


Finding a Centroid Edge

Theorem:

There exists a centroid edge that is either on the solid path P of the root, or is incident to the bottommost node of P

- Case 1: $w_1 < 1 + 2n/3$, centroid edge on P
- Case 2: $w_1 > 1 + 2n/3$, centroid edge incident to μ_1

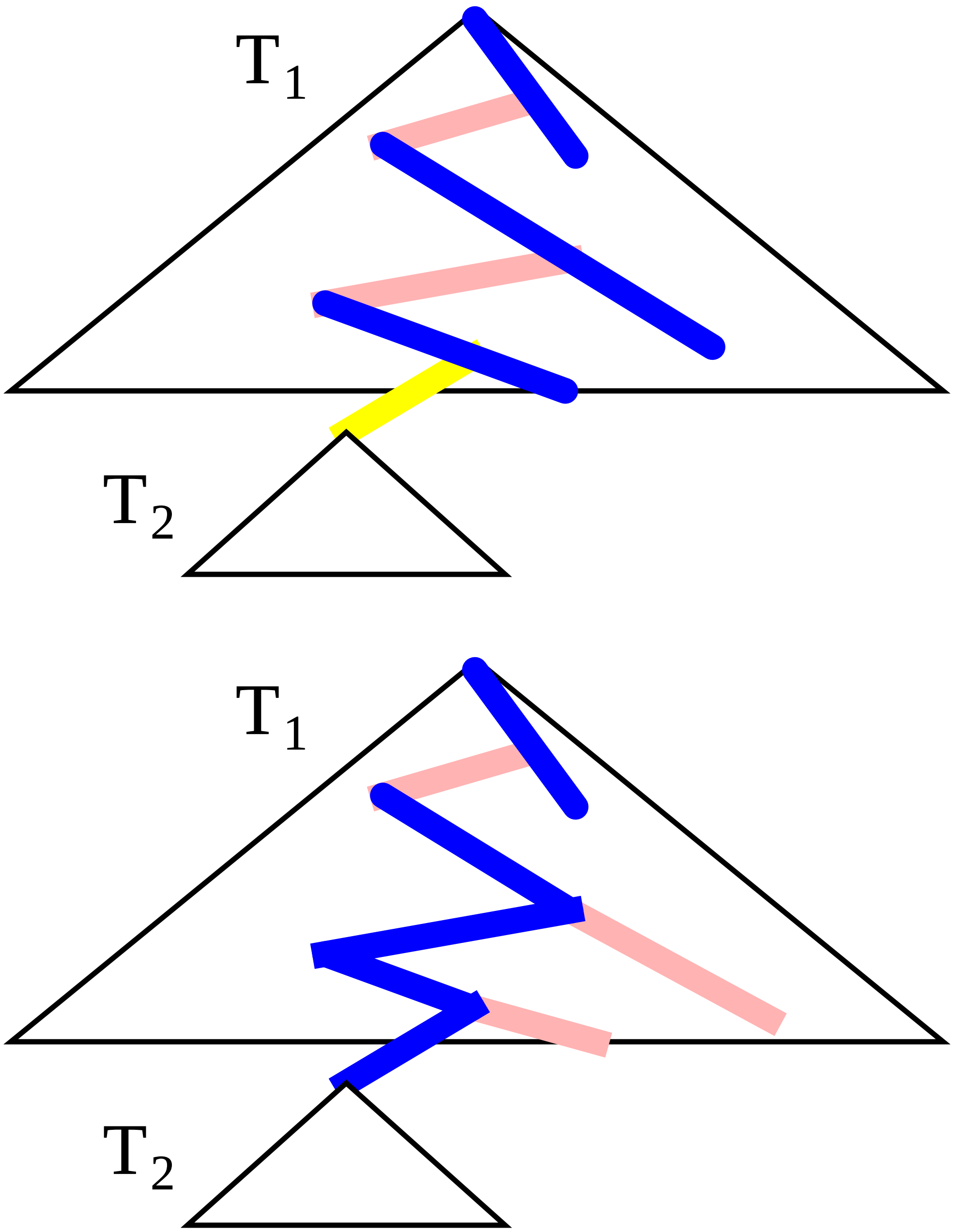


Corollary:

A centroid edge can be found in time O(log n)

Link/Cut Operations

- In a link operation, O(log n) edges may change from light to heavy, thus causing O(log n) split/splice operations on the solid paths. (And similarly for a cut operation.)



Time Complexity of Link/Cut

- Using standard balanced trees (e.g., AVL, red-black) each split/splice operation takes O(log n) time
  - Total time complexity: **$O(\log^2 n)$**

- To improve the update time, use ***biased search trees*** [Bent-Sleator-Tarjan, 85]
  - node μ on a solid path P
  - weight w(μ) = size of child of μ not in P
  - depth of μ-leaf = O(log (W/w(μ))), where W is the total weight
  - Since all the split/splice operations on solid paths are along a root-to-leaf path, the time complexity is now:

$$O(\log(n/w_1)+\log(w_1/w_2)+\ldots+\log(w_{k-1}/w_k))$$

  - Total time complexity: **$O(\log n)$**

Dynamization

- Repertory of update operations for monotone subdivisions:
  - insert/delete an edge
  - expand a vertex into two vertices connected by an edge
  - contract an edge
  - insert/delete a monotone chain
- Use the leftist monotone spanning tree obtained by selecting the lefmost incoming edge of each vertex
- Cannot dynamically maintain a triangulation of the subdivision
- Instead, dynamically maintain a refinement of the subdivision such that the dual tree D has degree at most 3
- An update operation on the subdivision corresponds to performing O(1) link/cut operations on the dynamic trees

Refinement of the Subdivision

- Insert a "comb" that duplicates the left chain of every region. The "comb" is placed infinitesimally close to the left chain
- The refined subdivision is topologically different but geometrically equivalent to the original subdivision.
- In the refined subdivision the dual tree of the leftist monotone spanning tree has degree at most 3.