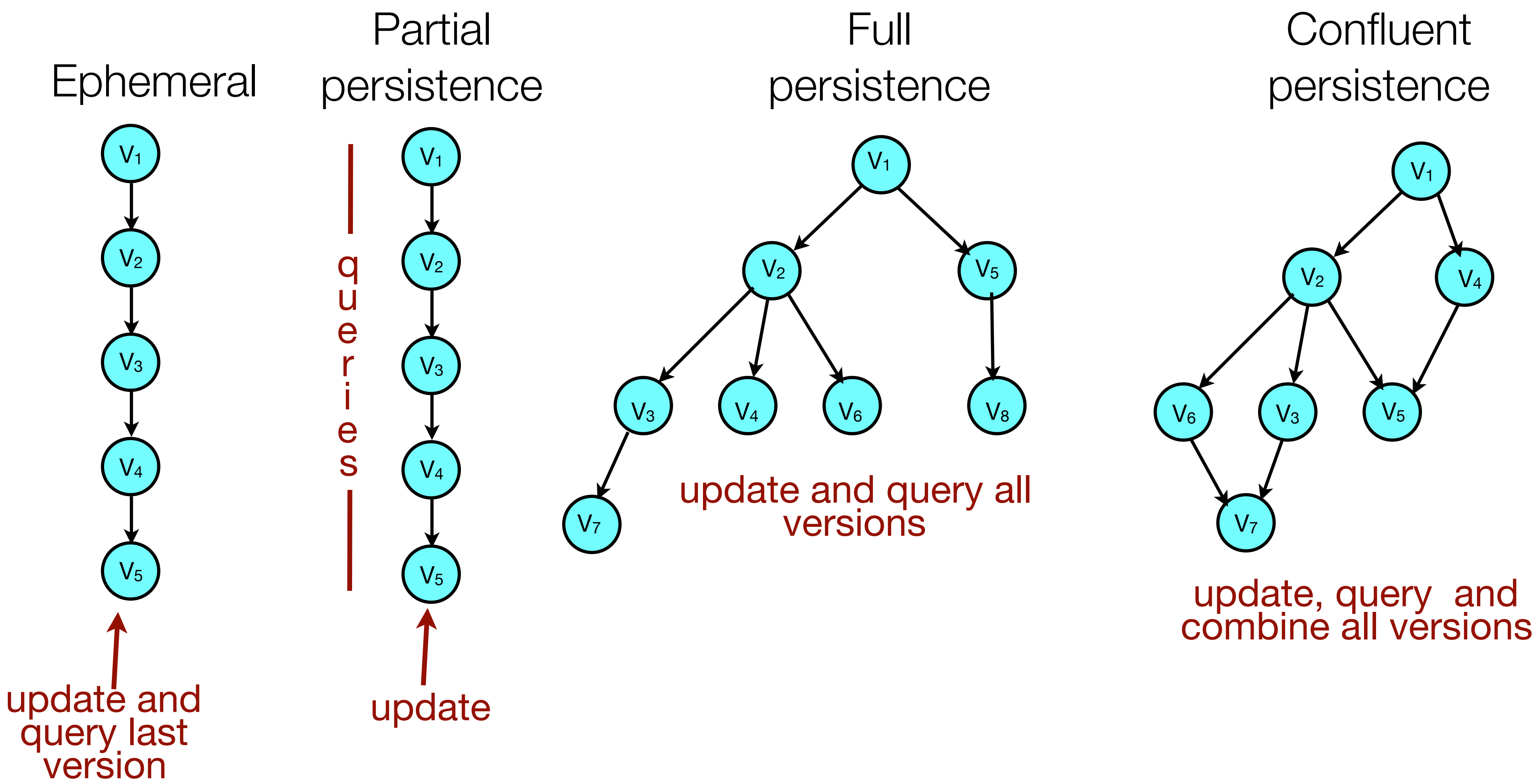
Persistent Data Structures and Planar Point Location

Inge Li Gørtz

Persistent Data Structures



Simple methods for making data structures persistent

- Structure-copying method. Create a copy of the data structure each time it is changed. Slowdown of $\Omega(n)$ time and space *per update* to a data structure of size n.

- Store a log-file of all updates. In order to access version i, first carry out i updates, starting with the initial structure, and generate version i. Overhead of $\Omega(i)$ time per access, $O(1)$ space and time per update.

- Hybrid-method. Store the complete sequence of updates and additionally each k-th version for a suitably chosen k. Result: Any choice of k causes blowup in either storage space or access time.

Overview

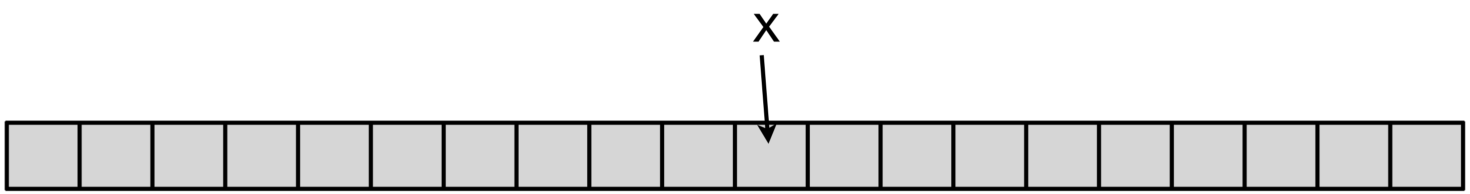
- Partial persistence.
  - Fat node method.
  - Node copying
- Algorithmic applications

Partial Persistence

Fat node method

Fat node method

- Associate set c(x) for each location in memory x.
- c(x)={<t,v>: x modified in version t, x has value v after construction of version t}



A(x): data structure containing c(x)

- Query q(t,x): Find largest version number t’ in t such that t’≤ t. Return value associated with t’ in A(x).
- Update (create new version m): If memory locations $x_1,\ldots,x_k$ modified to the values $v_1,\ldots v_k$: Insert <m,vi> in A(x_i).

Fat node method

- Implementation of $A(x)$:
  - Balanced binary search tree:
    - query $O(\log |c(x)|) = O(\log m)$, m number of versions.
    - Update: $O(1)$
    - Extra space: $O(1)$
  - y-fast trie:
    - query: $O(\log\log m)$
    - update: expected $O(\log\log m)$
    - Extra space: $O(1)$

Fat node method

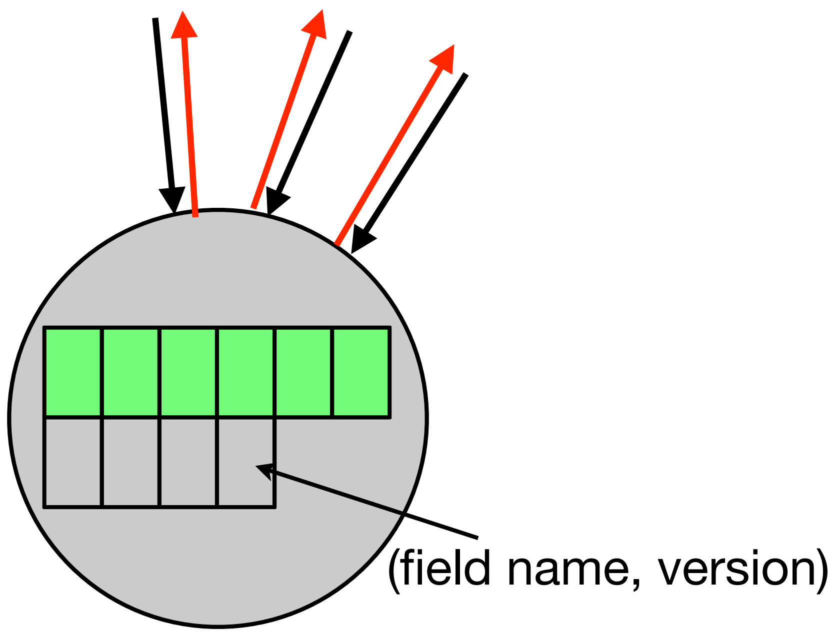
- Driscoll, Sarnak, Sleator, Tarjan, 1989.
  - Any data structure can be made partially persistent with slowdown O(log m) for queries and O(1) for updates. The space cost is O(1) for each ephemeral memory modification.
  - Any data structure can be made partially persistent on a RAM with slowdown O(loglog m) for queries and expected slowdown O(loglog m) for updates. The space cost is O(1) for each ephemeral memory modification

Partial Persistence

Node copying method

Node copying method

- Linked data structure with bounded indegree p, p = O(1).
- Each node has p predecessor pointers + p + 1 extra fields.
- Auxiliary array to keep pointer to root of each version



Partially persistent balanced search trees via node copying

- One extra pointer field in each node enough
- Extra pointers: tagged with version number and field name.
- When ephemeral update allocates a new node you allocate a new node as well.
- When the ephemeral update changes a pointer field:
  - If the extra pointer is empty use it, otherwise copy the node.
  - Try to store pointer to the new copy in its parent.
  - If the extra pointer at the parent is occupied copy the parent…..
- Maintain array of roots indexed by timestamp.

Partially persistent balanced search trees via node copying

- Analysis
  - Time slowdown:
    - access: O(1)
    - updates: O(1) amortized
  - Extra space factor: O(1) amortized
    - O(1) for new nodes also created by ephemeral data structure
    - O(1) amortized space for nodes created when a node is full.

Partially persistent balanced search trees via node copying

- Live nodes: Reachable from latest root.
- Potential function: $\Phi(D_i) =$ #live nodes – #free slots in live nodes
- Amortized cost of of an update = actual cost + change in potential.
- Consider insertion creating k new nodes (k-1 copied):
  - Copied node: old live node had potential 1. New copy has potential 0.
  - Number of live nodes increases by 1.
  - Am. cost $= k + \Phi(D_i) - \Phi(D_{i-1})$

$$= k + (1 - (k - 1))$$

$$= 2$$

Partially Persistent Data Structures

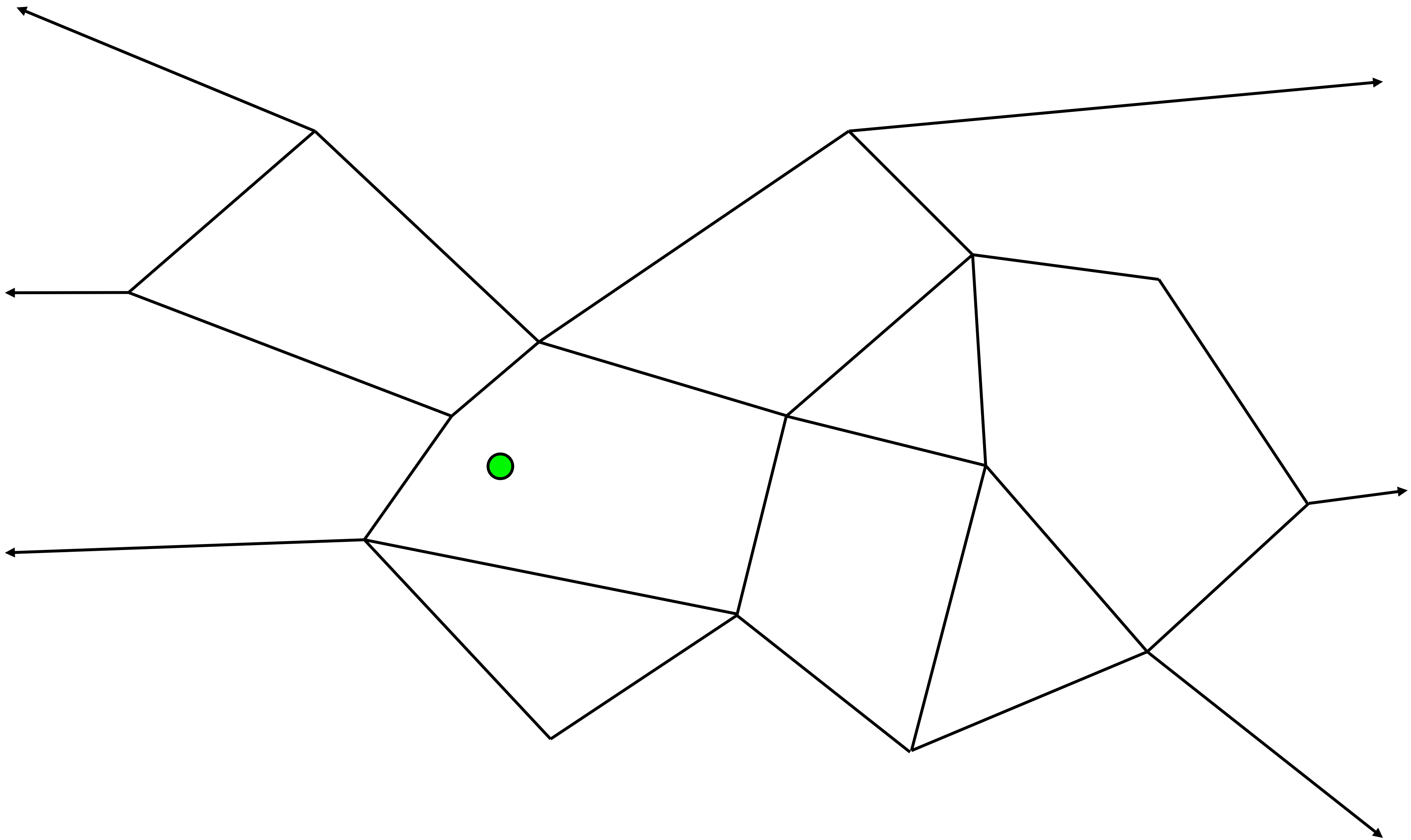
- Driscoll, Sarnak, Sleator, Tarjan, 1989.
  - Any bounded-degree linked data structure can be made partially persistent with (worst-case) slowdown O(1) for queries, amortized slowdown O(1) for updates, and amortized space cost O(1) per memory modification.

Algorithmic Applications

Planar Point Location

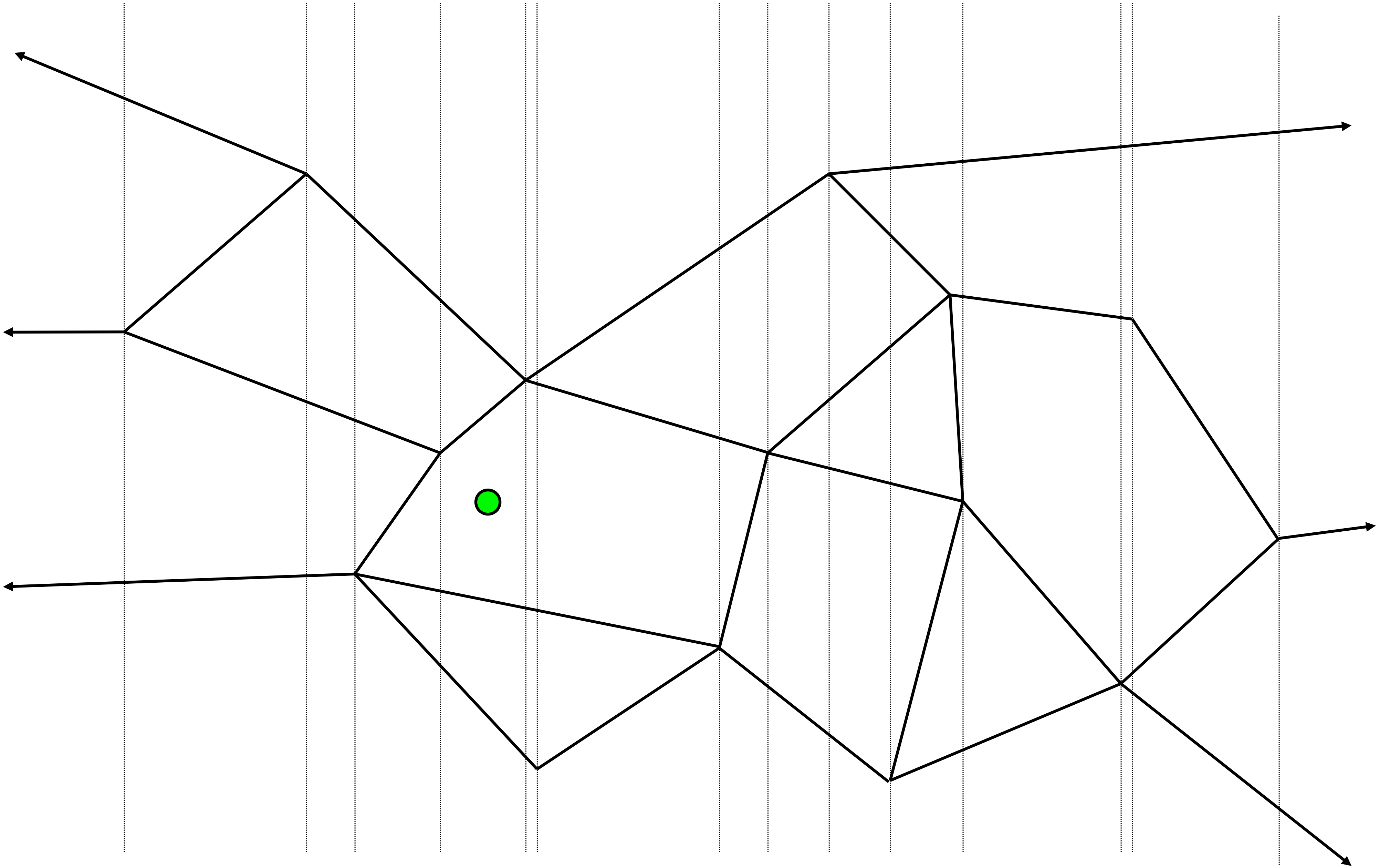
- Planar point location. Euclidean plane subdivided into polygons by n line segments that intersect only at their endpoints.
    - Query: given a query point p determine which polygon that contains p.

Planar point location: Example



From slides by H. Kaplan

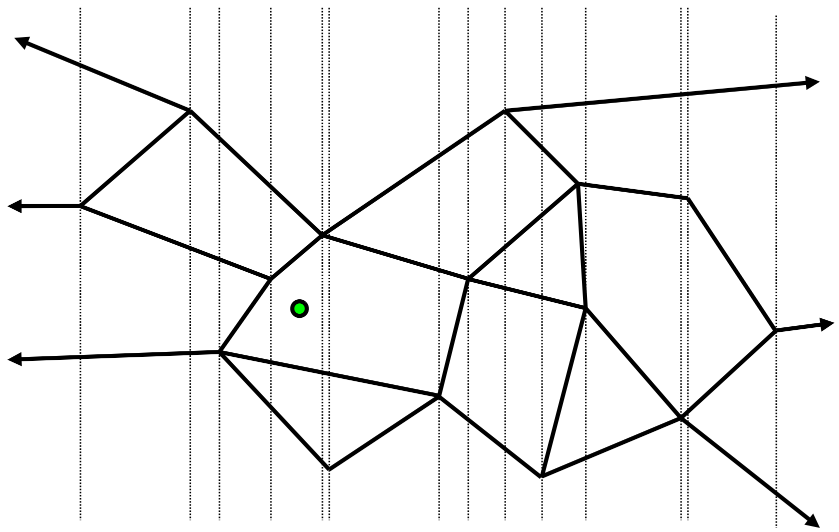
Planar point location: Example



From slides by H. Kaplan

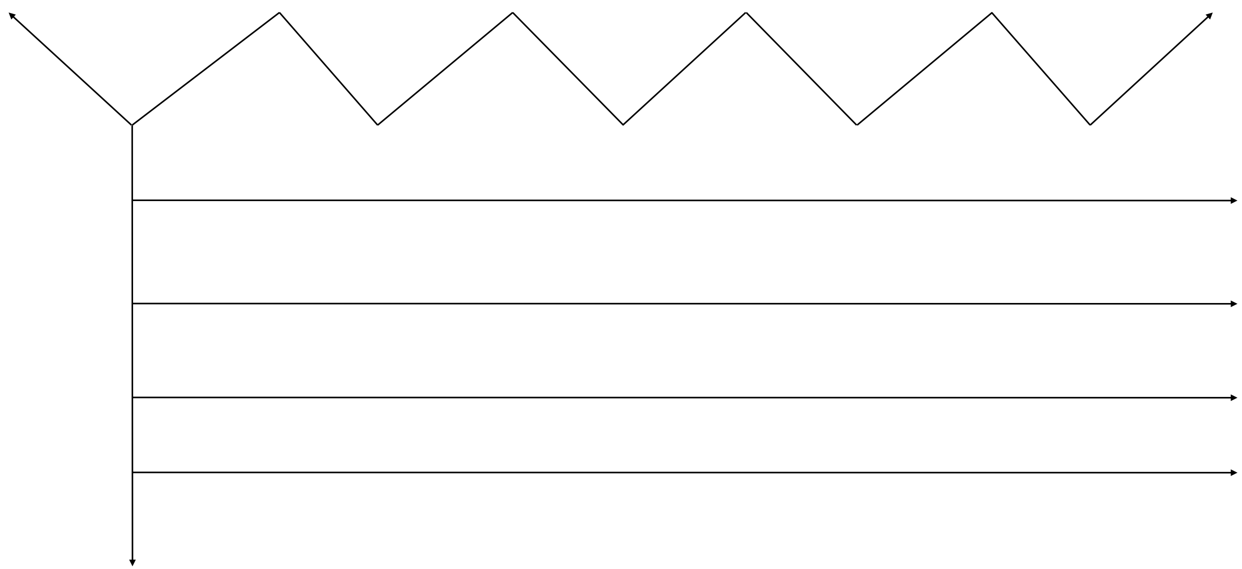
Planar Point Location

- Within each slab the lines are totally ordered.
- Search tree per slab containing the lines at the leaves with each line associate the polygon above it.
- Another search tree on the x-coordinates of the vertical lines.
- query
  - find appropriate slab
  - search the search tree of the slab to find the polygon



Planar Point Location

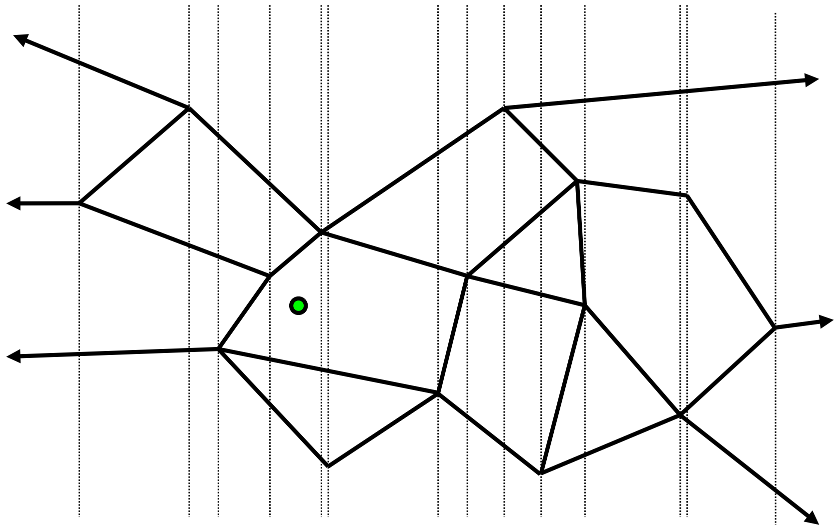
- One search tree for each slab:
    - Query time:
    - Space:



Total # lines O(n), and number of lines in each slab is O(n).

Planar point location: Improve space bound

- Key observation: The lists of the lines in adjacent slabs are very similar.
- Create the search tree for the first slab.
- Obtain the next one by deleting the lines that end at the corresponding vertex and adding the lines that start at that vertex.
- Number of insertions/deletions?
- Use partially persistent search tree.
  x-axis is time.



Planar Point Location

- Sarnak and Tarjan. Sweep line + partially persistent binary search tree:
  - Preprocessing time: O(n log n)
  - Query time: O(log n)
  - Space O(n)

- To get linear space: Balanced binary search tree with worst case O(1) memory modifications per update.