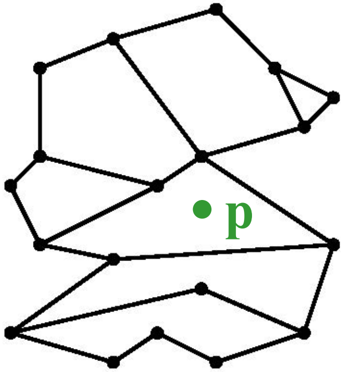
Computational Geometry



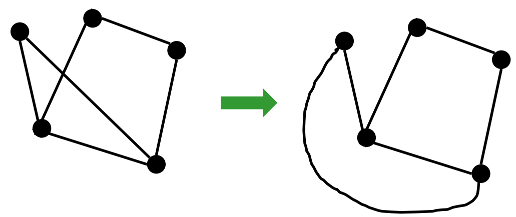
Planar Point Location

Michael Goodrich

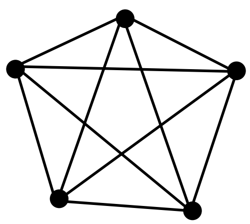
with some slides from Carola Wenk and Olivier Pirson

Planar Subdivision

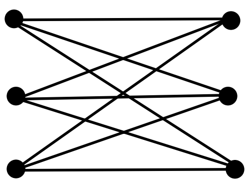
- Let $G=(V,E)$ be an undirected graph.
- G is planar if it can be embedded in the plane without edge crossings.



planar

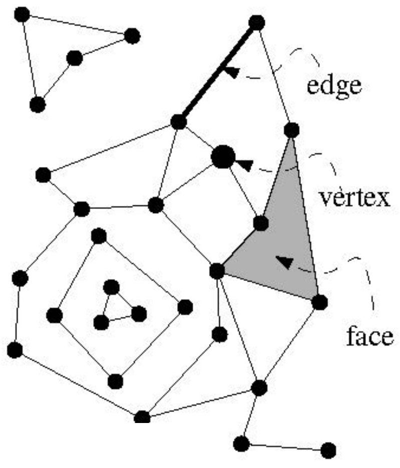


K_5, not planar



$K_{3,3}$, not planar

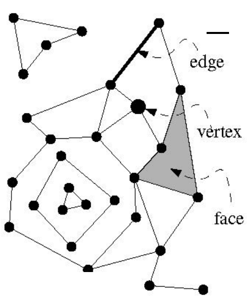
- A planar embedding (=drawing) of a planar graph G induces a **planar subdivision** consisting of vertices, edges, and faces.

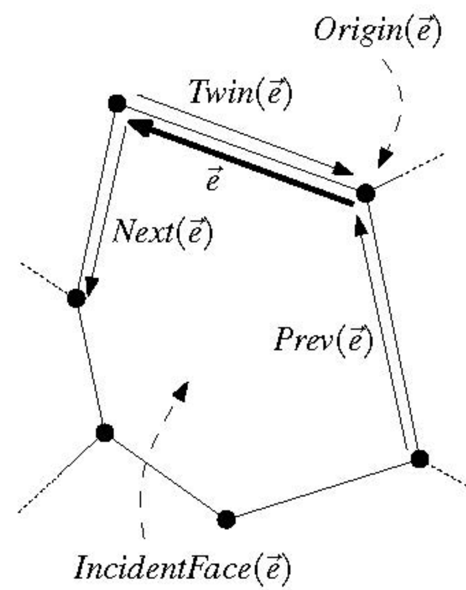


Review: Doubly-Connected Edge List

- The **doubly-connected edge list (DCEL)** is a popular data structure to store the geometric and topological information of a planar subdivision.
  - It contains records for each face, edge, vertex
  - (Each record might also store additional application-dependent attribute information.)
  - It should enable us to perform basic operations needed in algorithms, such as walk around a face, or walk from one face to a neighboring face

- The DCEL consists of:
  - For each vertex *v*, its coordinates are stored in **Coordinates(*v*)** and a pointer **IncidentEdge(*v*)** to a half-edge that has *v* as its origin.
  - Two oriented **half-edges** per edge, one in each direction. These are called **twins**. Each of them has an **origin** and a **destination**. Each half-edge *e* stores a pointer **Origin(*e*)**, a pointer **Twin(*e*)**, a pointer **IncidentFace(e)** to the face that it bounds (on its left), and pointers **Next (e)** and **Prev(e)** to the next and previous half-edge on the boundary of **IncidentFace(e)**.
  - For each face *f*, **OuterComponent(*f*)** is a pointer to some half-edge on its outer boundary (null for unbounded faces). It also stores a list **InnerComponents(*f*)** which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.





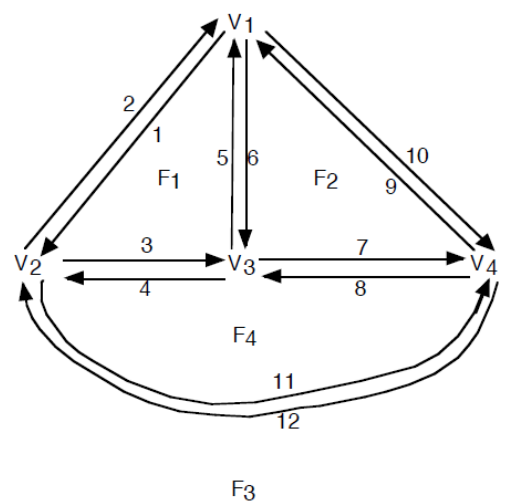
Review: Doubly-Connected Edge List

- For each vertex *v*, its coordinates are stored in **Coordinates(*v*)** and a pointer **IncidentEdge(*v*)** to a half-edge that has *v* as its origin.
- Two oriented **half-edges** per edge, one in each direction. These are called **twins**. Each of them has an **origin** and a **destination**. Each half-edge *e* stores a pointer **Origin(*e*)**, a pointer **Twin(*e*)**, a pointer **IncidentFace(e)** to the face that it bounds (on its left), and pointers **Next (e)** and **Prev(e)** to the next and previous half-edge on the boundary of **IncidentFace(e)**.
- For each face *f*, **OuterComponent(*f*)** is a pointer to some half-edge on its outer boundary (null for unbounded faces). It also stores a list **InnerComponents(*f*)** which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.

Vertex #	Cooordinates	Incident Edge#
1	0 0 0	1
2	1 0 0	2
3	0 1 0	4
4	0 0 1	8

Face #	Edge
1	1
2	6
3	10 (OuterComponent)
4	11

Edge #	Origin (Tail)	Twin	Incident Face	Next	Prev
1	1	2	1	3	5
2	2	1	3	10	12
3	2	4	1	5	1
4	3	3	4	11	8
5	3	6	1	1	3
6	1	5	2	7	9
7	3	8	2	9	6
8	4	7	4	4	11
9	4	10	2	6	7
10	1	9	3	12	2
11	2	12	4	8	4
12	4	11	3	2	10



DCEL example from: http://ranger.uta.edu/~weems/NOTES5319/DCEL.pdf

Complexity of a Planar Subdivision

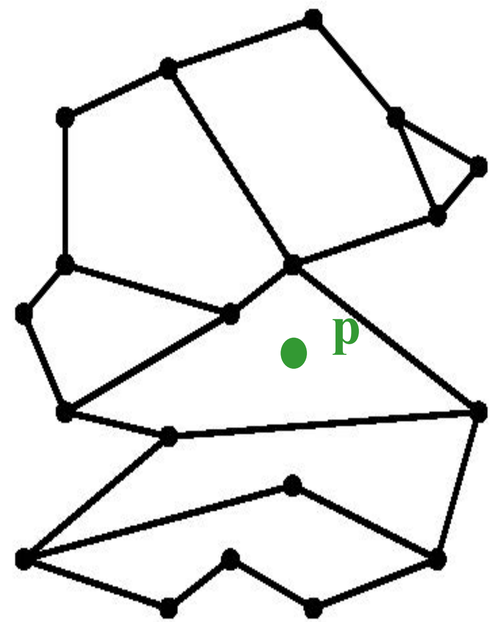
- The complexity of a planar subdivision is:
  #vertices + #edges + #faces = $n_v + n_e + n_f$
- Euler's formula for planar (embedded) graphs:
  1) $n_v - n_e + n_f \geq 2$
  2) $n_e \leq 3n_v - 6$

  Proof that *2)* follows from *1)*:
  Count edges. Every face is bounded by ≥ 3 edges.
  Every edge bounds ≤ 2 faces.
  $\Rightarrow 3n_f \leq 2n_e \Rightarrow n_f \leq 2/3 n_e$
  $\Rightarrow 2 \leq n_v - n_e + n_f \leq n_v - n_e + 2/3\ n_e = n_v - 1/3\ n_e$
  $\Rightarrow 2 \leq n_v - 1/3\ n_e$

- Hence, the complexity of a planar subdivision is $O(n_v)$, i.e., linear in the number of vertices.

Point Location

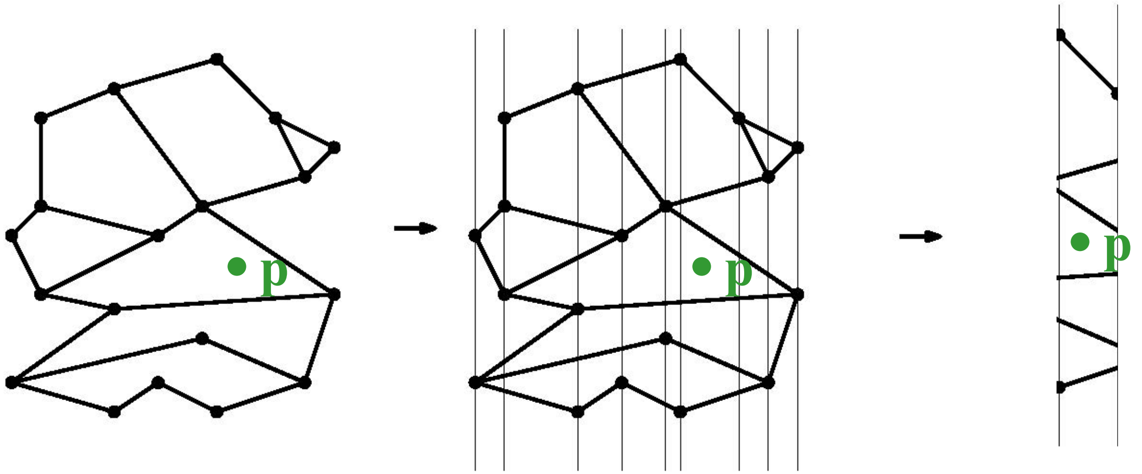
- **Point location task:**
  Preprocess a planar subdivision to efficiently answer **point-location queries** of the type: Given a point $p=(p_x,p_y)$, find the face it lies in.



- **Important metrics:**
  1. Time complexity for preprocessing = time to construct the data structure
  2. Space needed to store the data structure
  3. Time complexity for querying the data structure

Slab Method

- **Slab method:**
  Draw a vertical line through each vertex. This decomposes the plane into slabs.



- In each slab, the vertical order of the line segments remains constant.
- If we know in which slab ***p*** lies, we can perform binary search, using the sorted order of the segments in the slab.
- Find slab that contains ***p*** by binary search on x among slab boundaries.
- A second binary search in thr slab determines the face containing ***p***.
- Search complexity $O(\log n)$, but space complexity $O(n^2)$.

Coherence

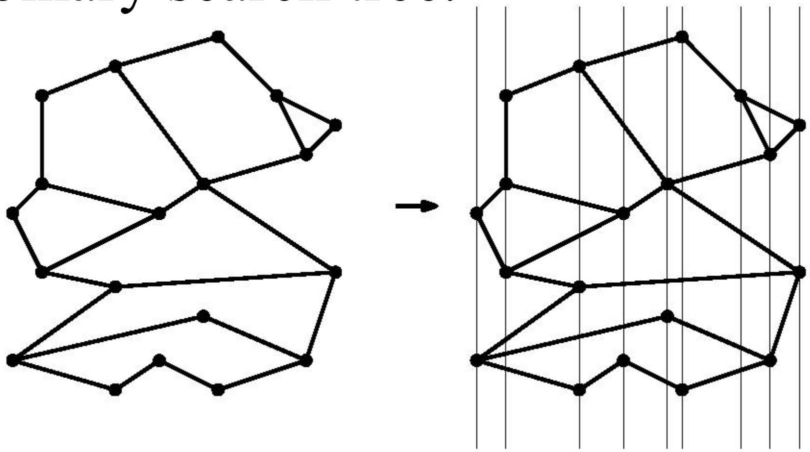
- Data for close-by objects or time snapshots is often similar
  - A common theme in a lot of computer science

- We can often achieve efficiencies in time and/or space by exploiting this similarity

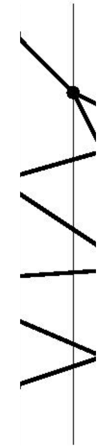
Image by Kyprianidis and Kang. Used for educational purposes only.

Revisiting the Slab Method

- **Consider a plane-sweep of a planar subdivision:**
- Sweep a vertical line, stopping at each vertex as an event.
- Maintain the edges intersecting the sweep line in a balanced binary search tree.



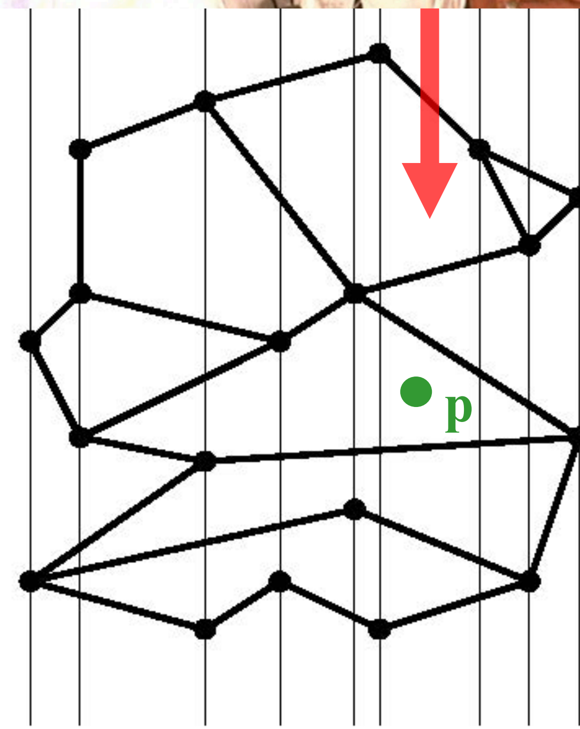
An event:



- From one event to the next, the vertical order of the line segments changes little.
- The only changes occur for the segments we add or remove because their endpoints occur at that event.
- The slab method essentially stores a complete snapshot of the binary search tree that exists at each event
- This is wasteful of time and space, because of the coherence of the binary search tree from one event to the next.

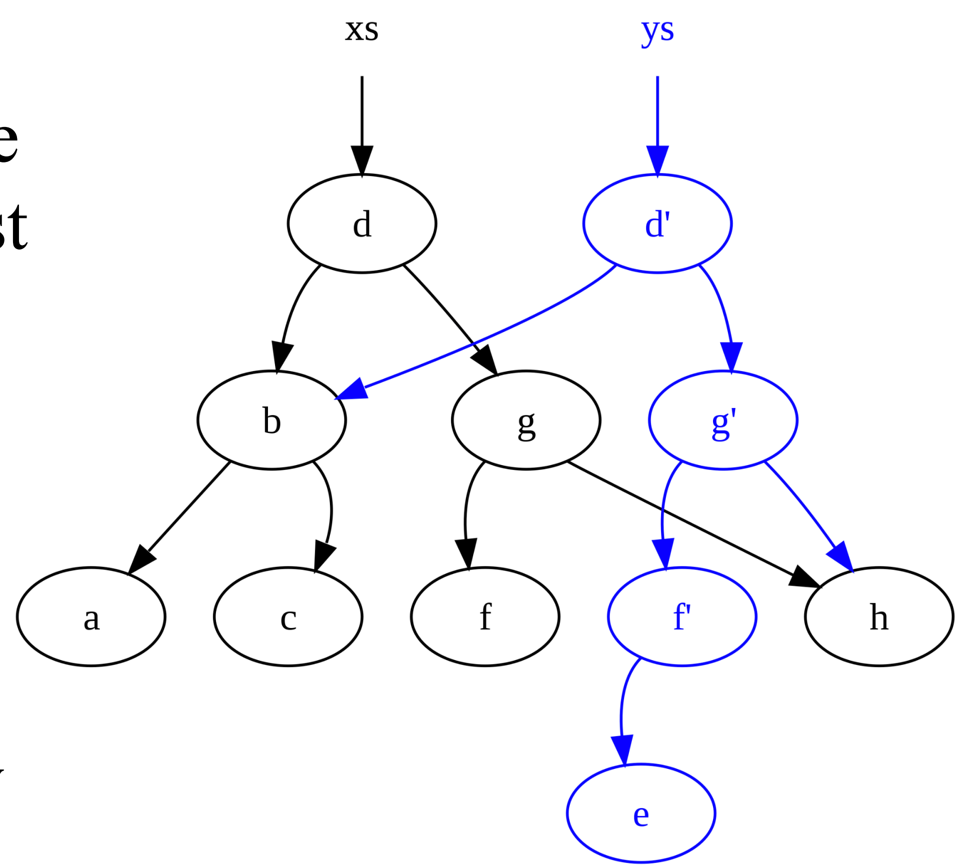
BACK IN TIME

- What if we could answer a point location query by **going back in time** to the place in the plane sweep when we crossed the query point, *p*?
- We could just do a search for *p* in our binary search tree that existed at that time.



Persistent Data Structures

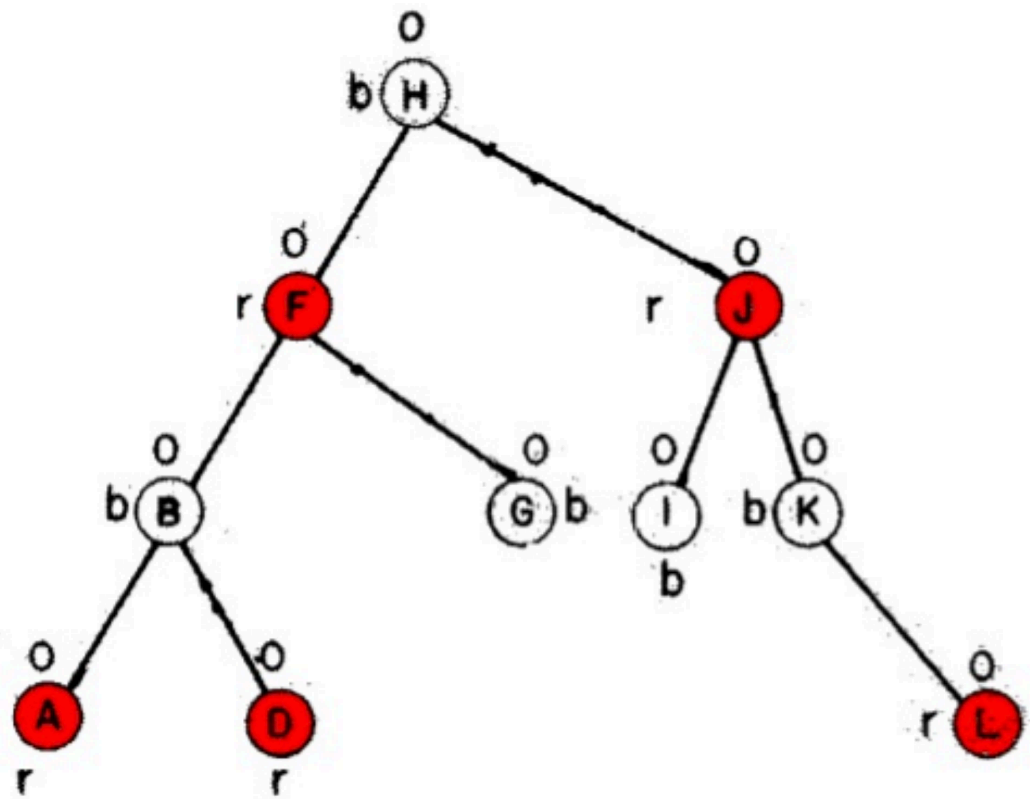
- A data structure is **persistent** if it allows one to perform queries on past versions.
- One way to do this for binary search trees is by **path copying**.
  - Leave the old nodes unchanged and create new nodes for the nodes that change.



An Example of Path Copying

Persistent red–black tree with **path copying**.

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.

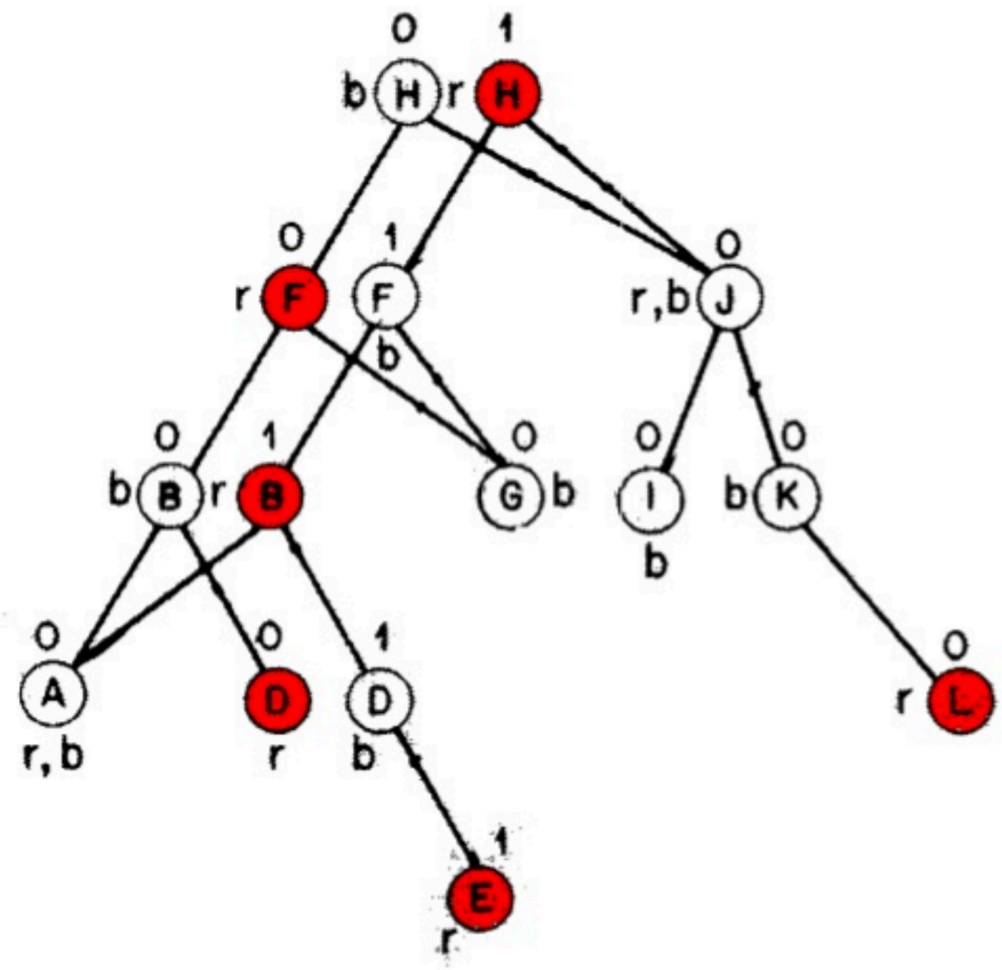


An Example of Path Copying

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)

Persistent red–black tree with **path copying**.

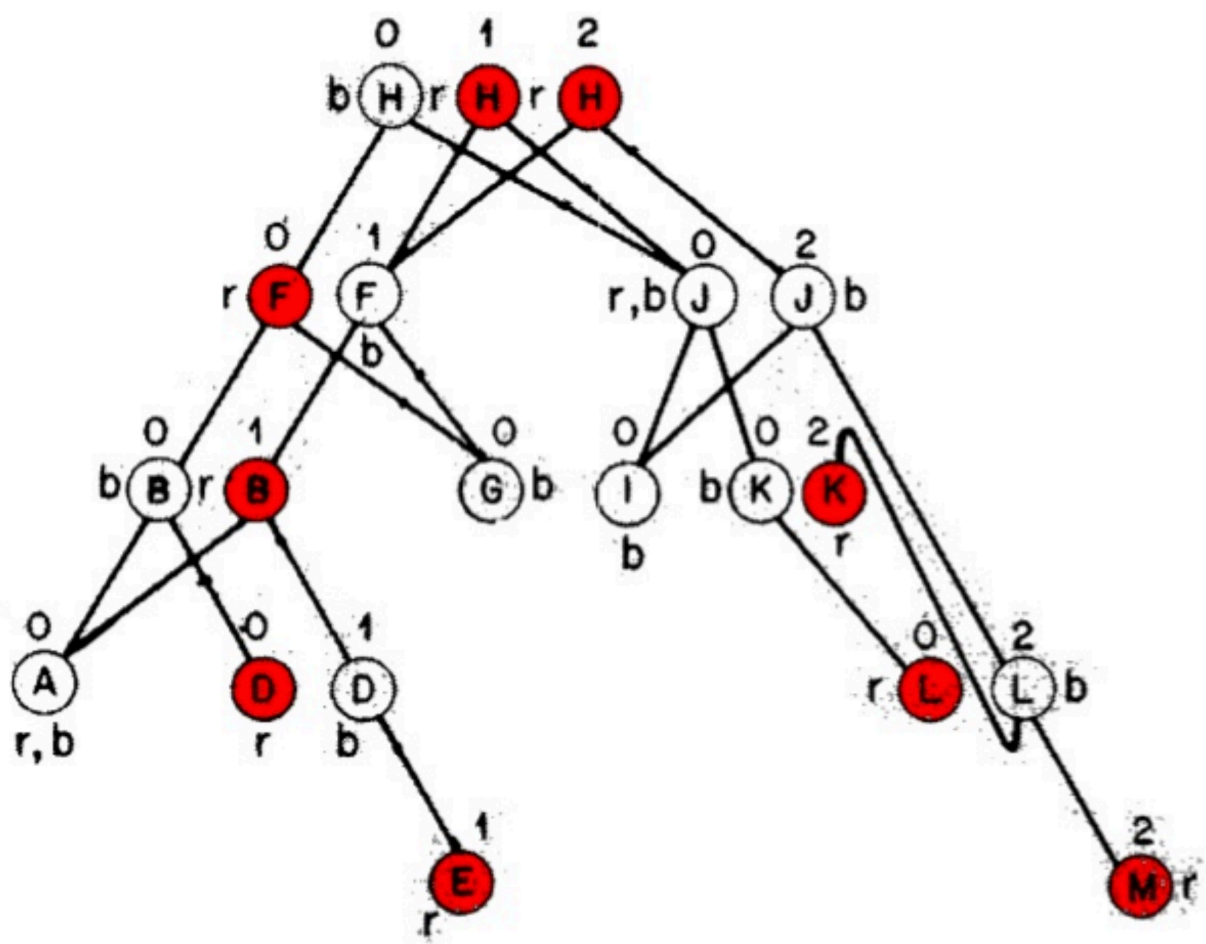


An Example of Path Copying

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)
- Add M, in the time 2.

Persistent red–black tree with **path copying**.



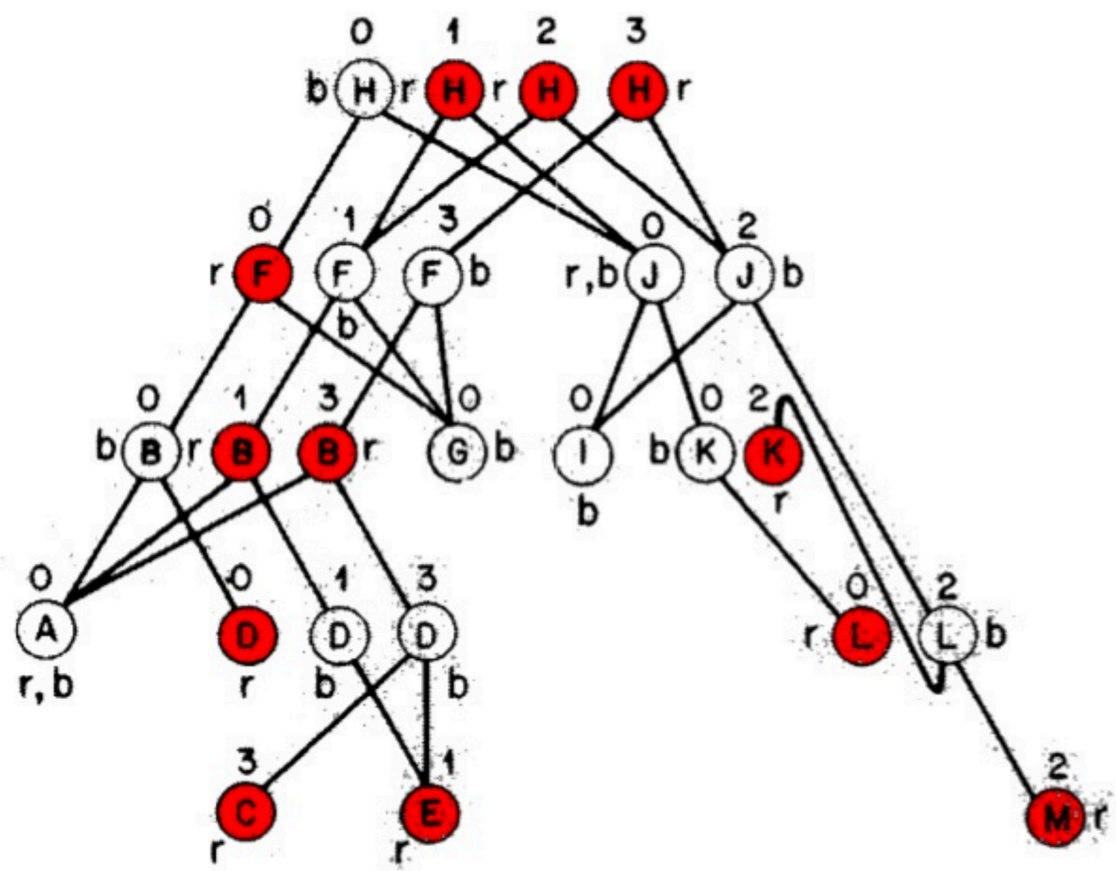
An Example of Path Copying

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)
- Add M, in the time 2.
- Add C, in the time 3.

We have preserved the O(log n) **complexity of operations**.

Persistent red–black tree with **path copying**.



Analysis of Path Copying

- Since each event in the plane sweep takes O(log n) time, we create O(log n) new nodes for each event. Thus, the total space and preprocessing time is O(n log n).
- To perform a point-location query:
  - We first do a binary search of the root nodes to determine the root that was active for the x-coordinate of the query point, p.
  - Then we do a binary search in this tree for the y-coordinate of p, locating the face that contains p.
  - Query time: O(log n)
- Sarnak and Tarjan show how to get the total space down to O(n) – **read** their paper at the notes site.