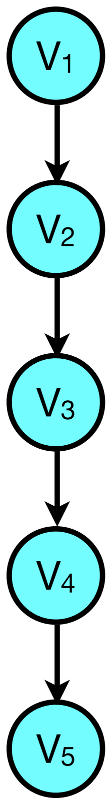
Persistent Data Structures and Planar Point Location

Inge Li Gørtz

Persistent Data Structures

Ephemeral



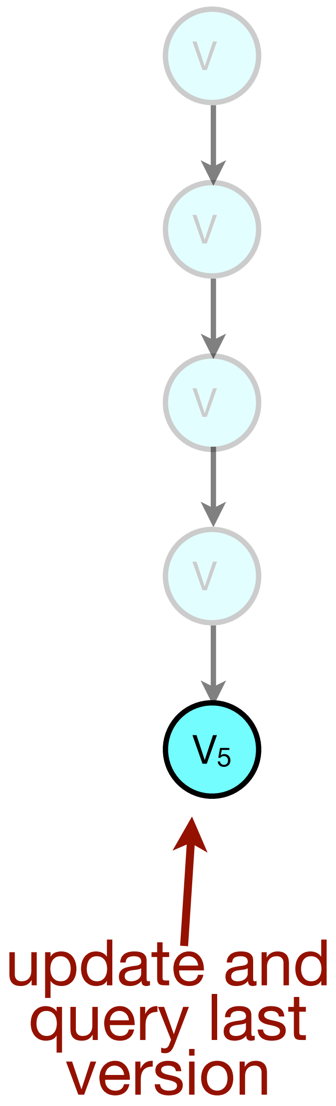
Partial persistence

Full persistence

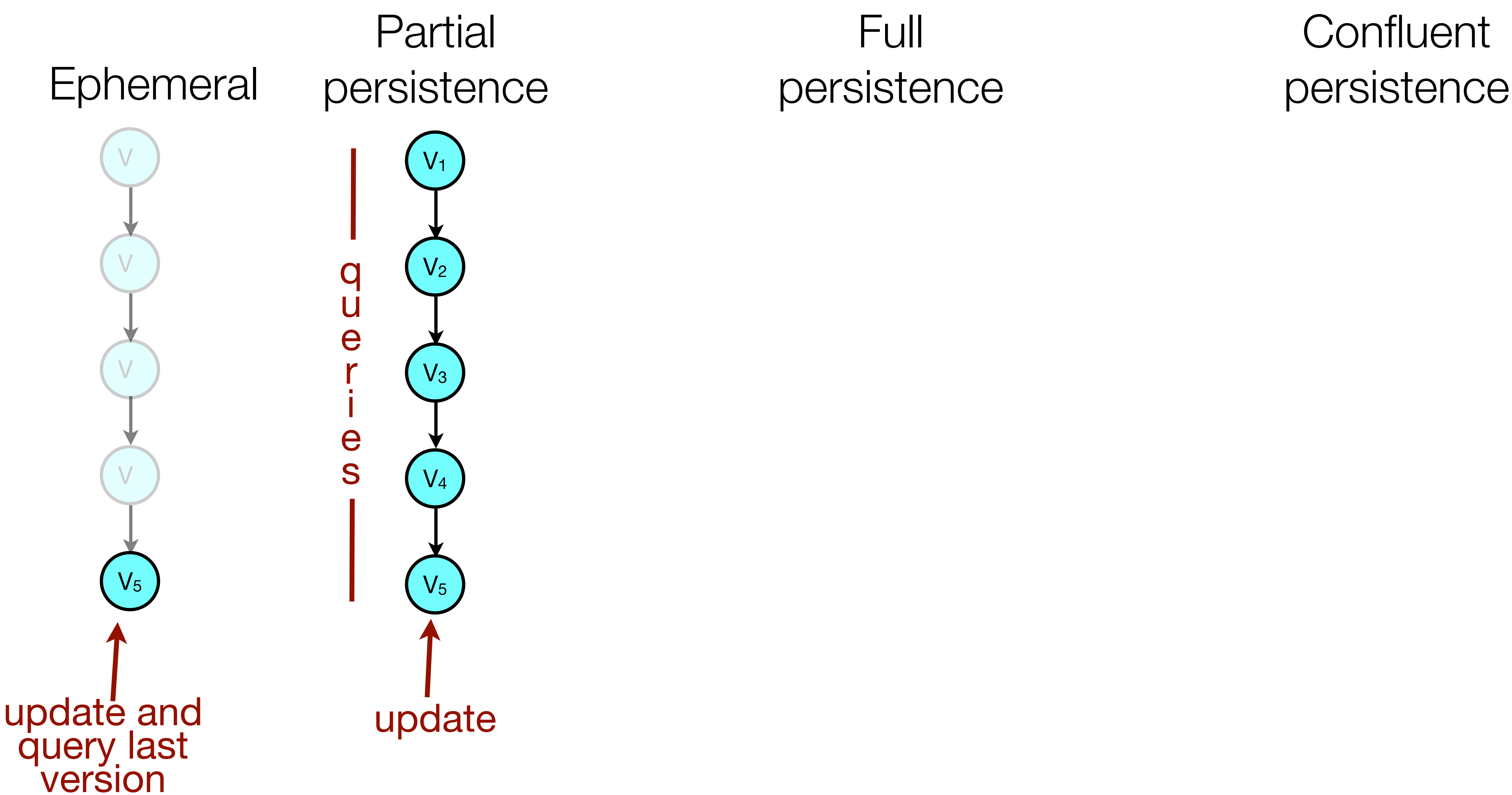
Confluent persistence

Persistent Data Structures

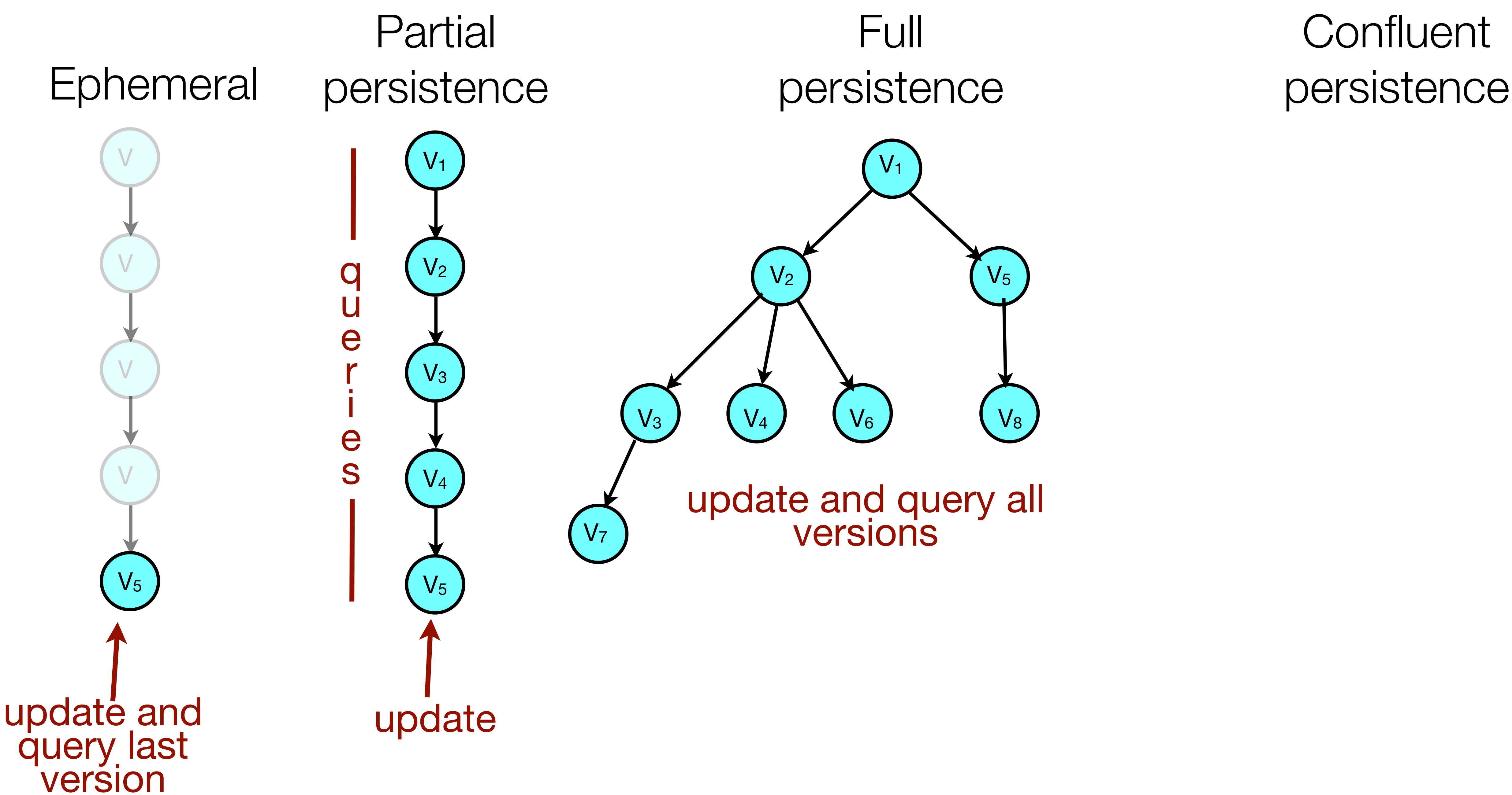
Ephemeral	Partial persistence	Full persistence	Confluent persistence



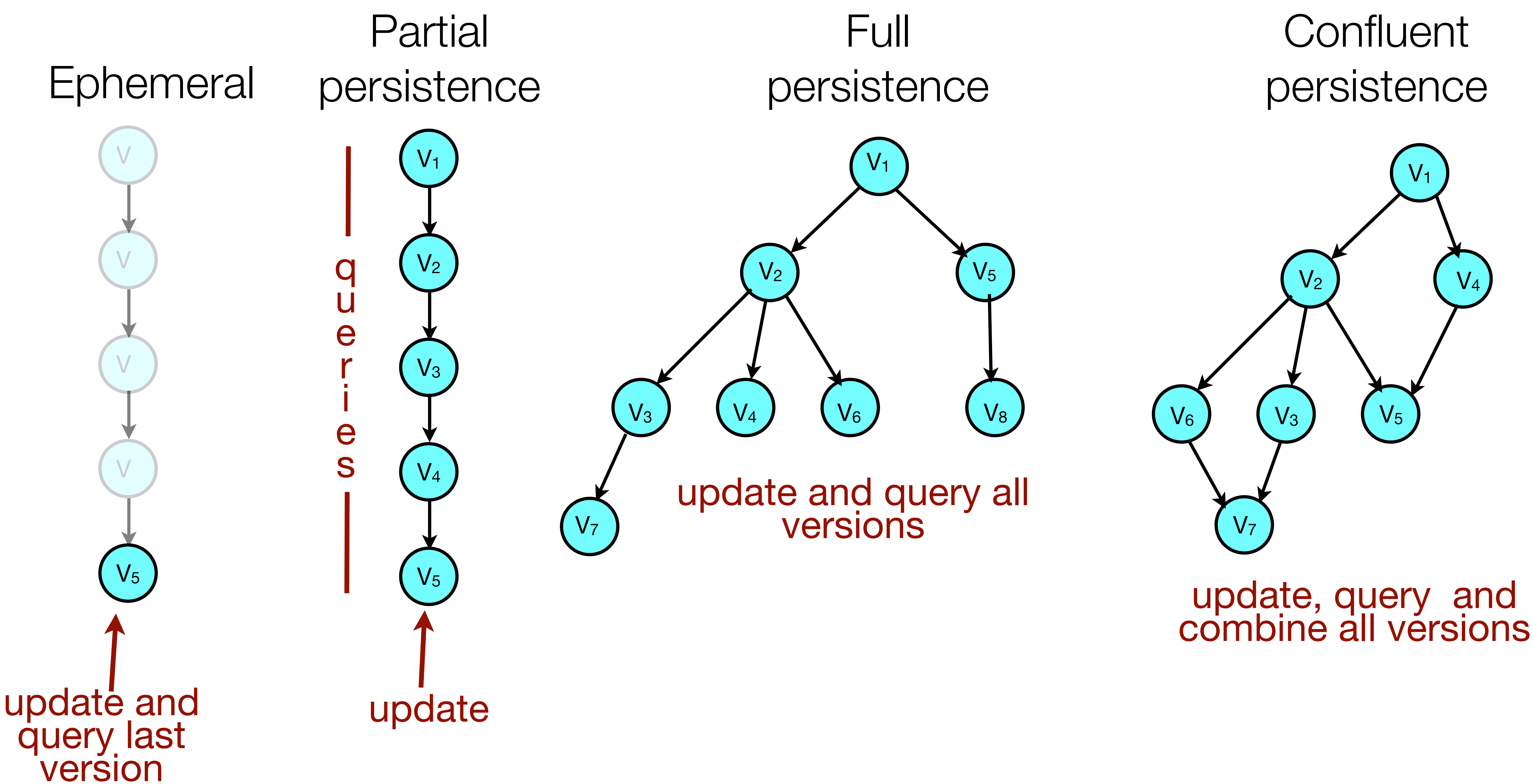
Persistent Data Structures



Persistent Data Structures



Persistent Data Structures



Simple methods for making data structures persistent

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.

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.

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

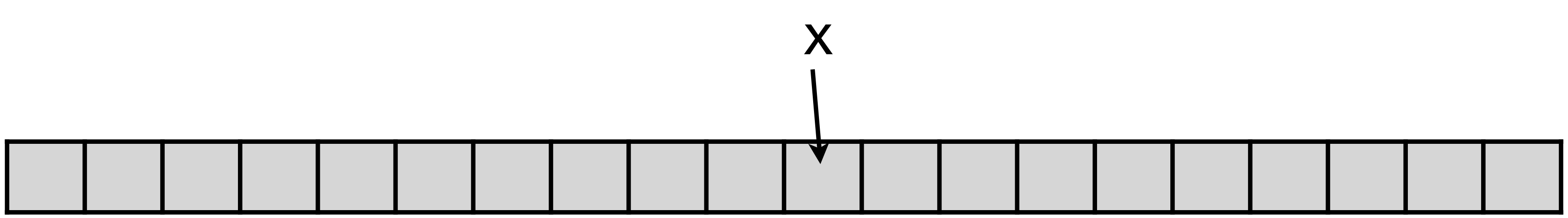
- Partial persistence.
  - Fat node method.
  - Node copying
- Full persistence. Main idea.
- Algorithmic applications

Partial Persistence

Fat node method

Fat node method

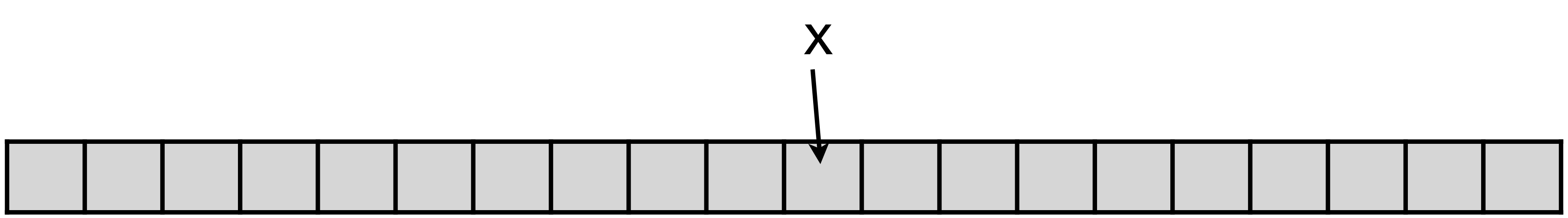
- Associate set c(x) for each location in memory x.



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

Fat node method

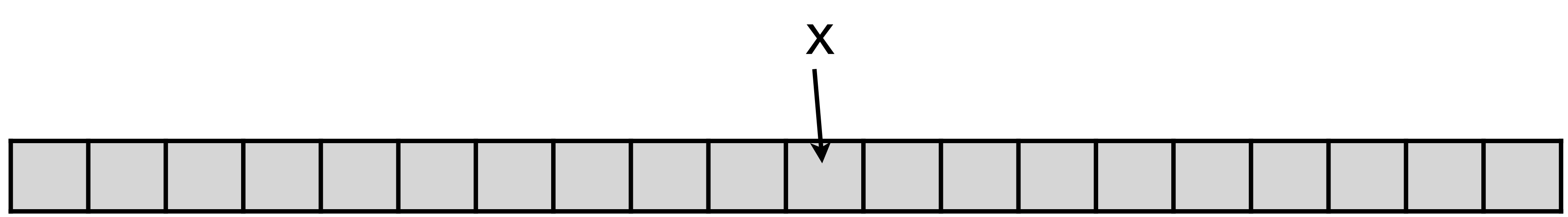
- Associate set c(x) for each location in memory x.
- c(x)={<t,v>: x modified in version t, x has value v after construction of version t}



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

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}

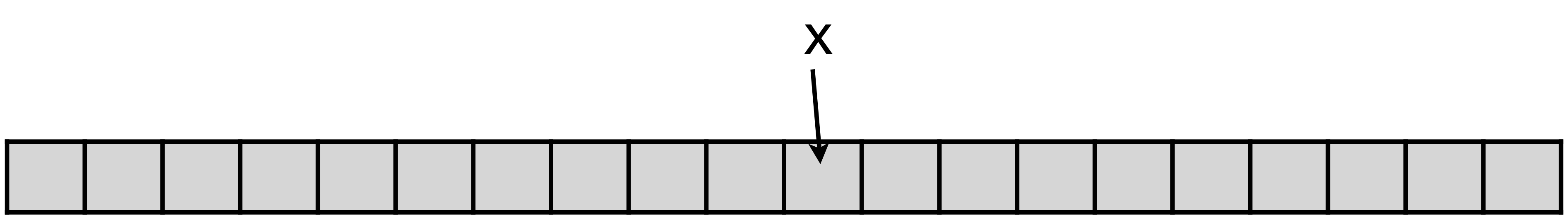


D(x): data structure containing c(x)

- Query q(t,x): Find largest version number t’ in t such that t’≤ t. Return value associated with t’ in D(x).

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}



D(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 D(x).
- Update (create new version m): If memory locations $x_1,...,x_k$ modified to the values $v_1,...v_k$: Insert <m,v_i> in D(x_i).

Fat node method

- Implementation of D(x):

Fat node method

- Implementation of D(x):
  - Balanced binary search tree:

Fat node method

- Implementation of $D(x)$:
    - Balanced binary search tree:
        - query $O(\log |c(x)|) = O(\log m)$, m number of versions.

Fat node method

- Implementation of D(x):
  - Balanced binary search tree:
    - query O(log |c(x)|) = O(log m), m number of versions.
    - Update: O(1)

Fat node method

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

Fat node method

- Implementation of D(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:

Fat node method

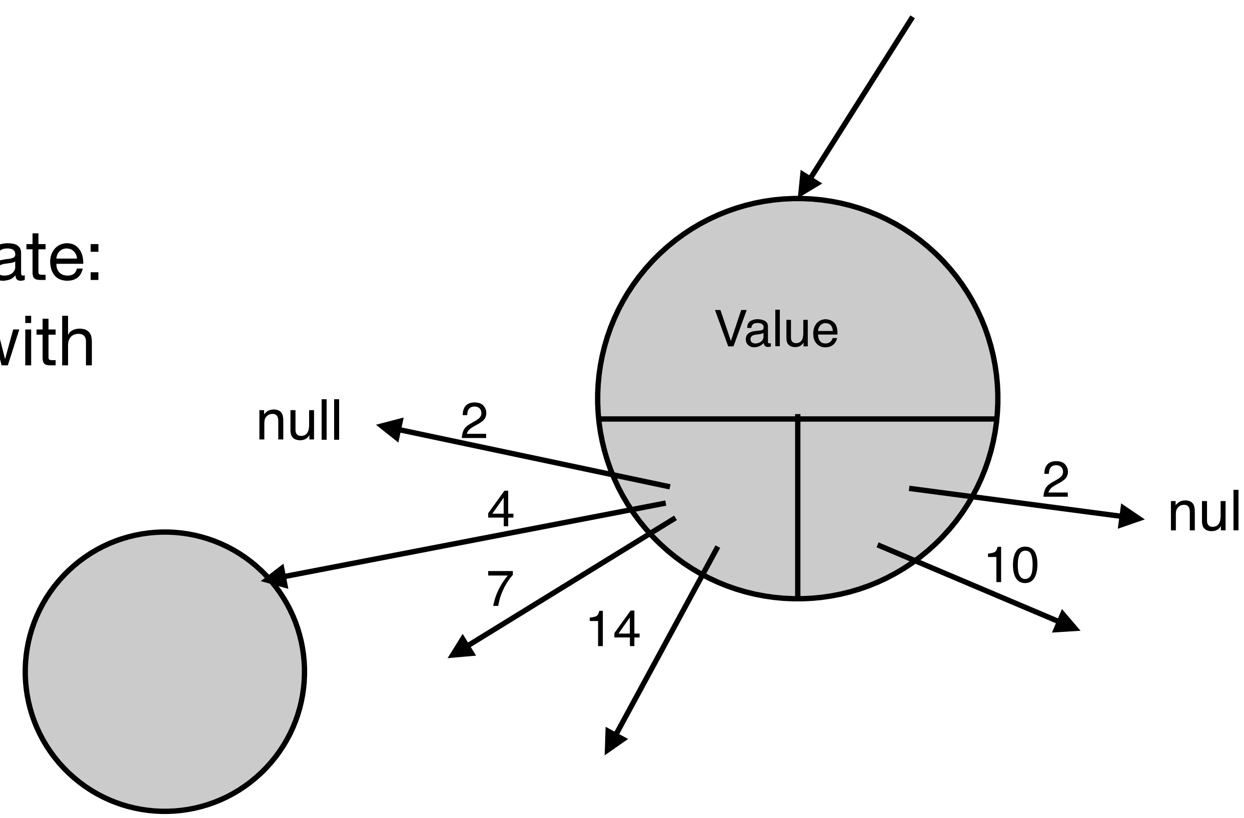
- Implementation of D(x):
  - Balanced binary search tree:
    - query O(log |c(x)|) = O(log m), m number of versions.
    - Update: O(1)
    - Extra space: O(1)
  - y-fast trie:
    - query: O(loglog m)

Fat node method

- Implementation of $D(x)$:
  - 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: $O(1)$

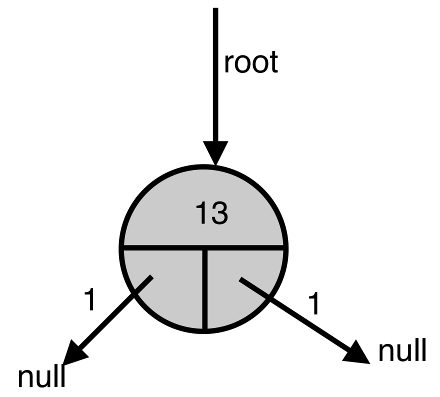
Fat node method

- Linked data structures:
  - each pointer field store many time value pairs.
  - new node created by ephemeral update: create new node and mark all fields with version i.
  - Auxiliary array keep pointer to root of each version.

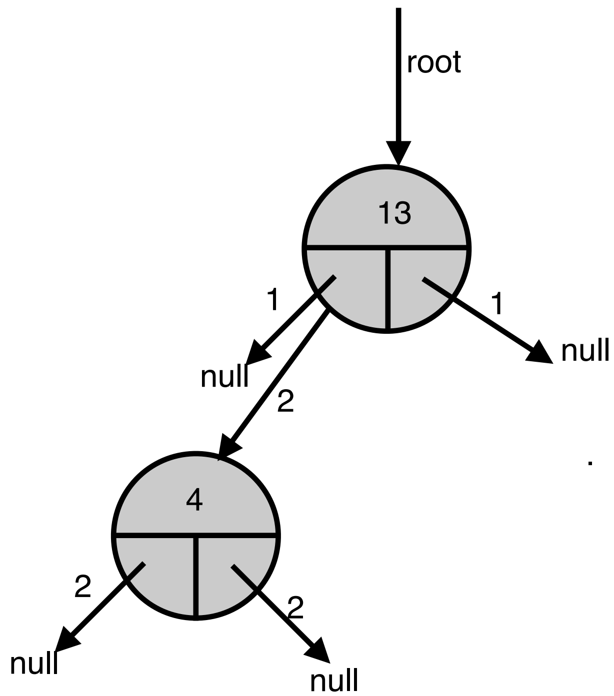


Fat node method example

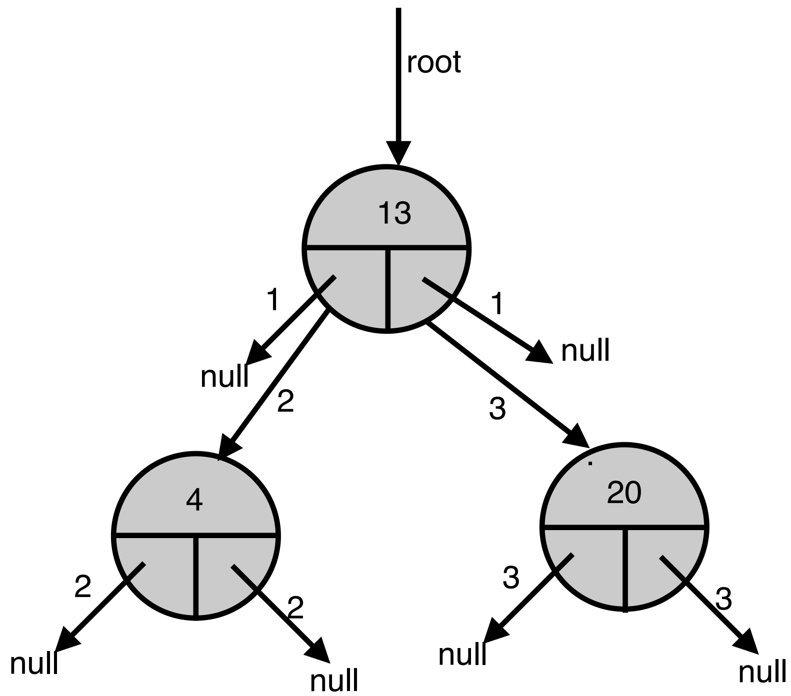
Fat node method example



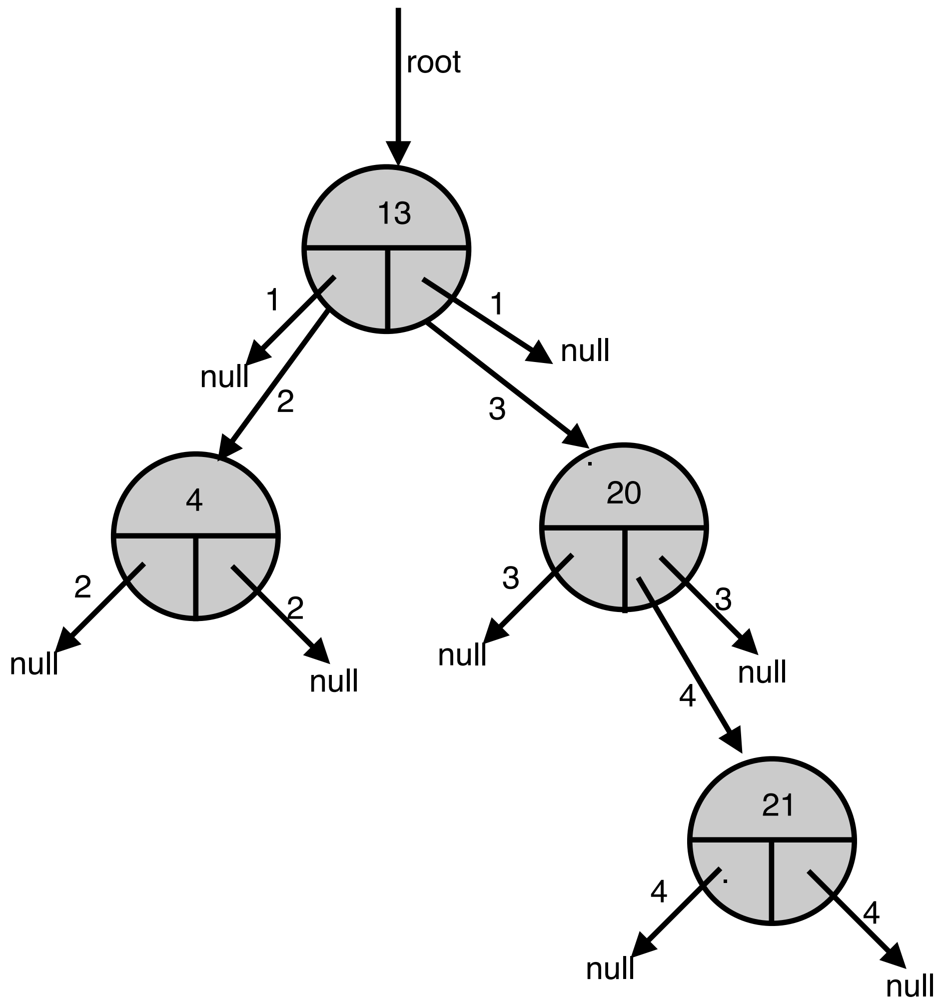
Fat node method example



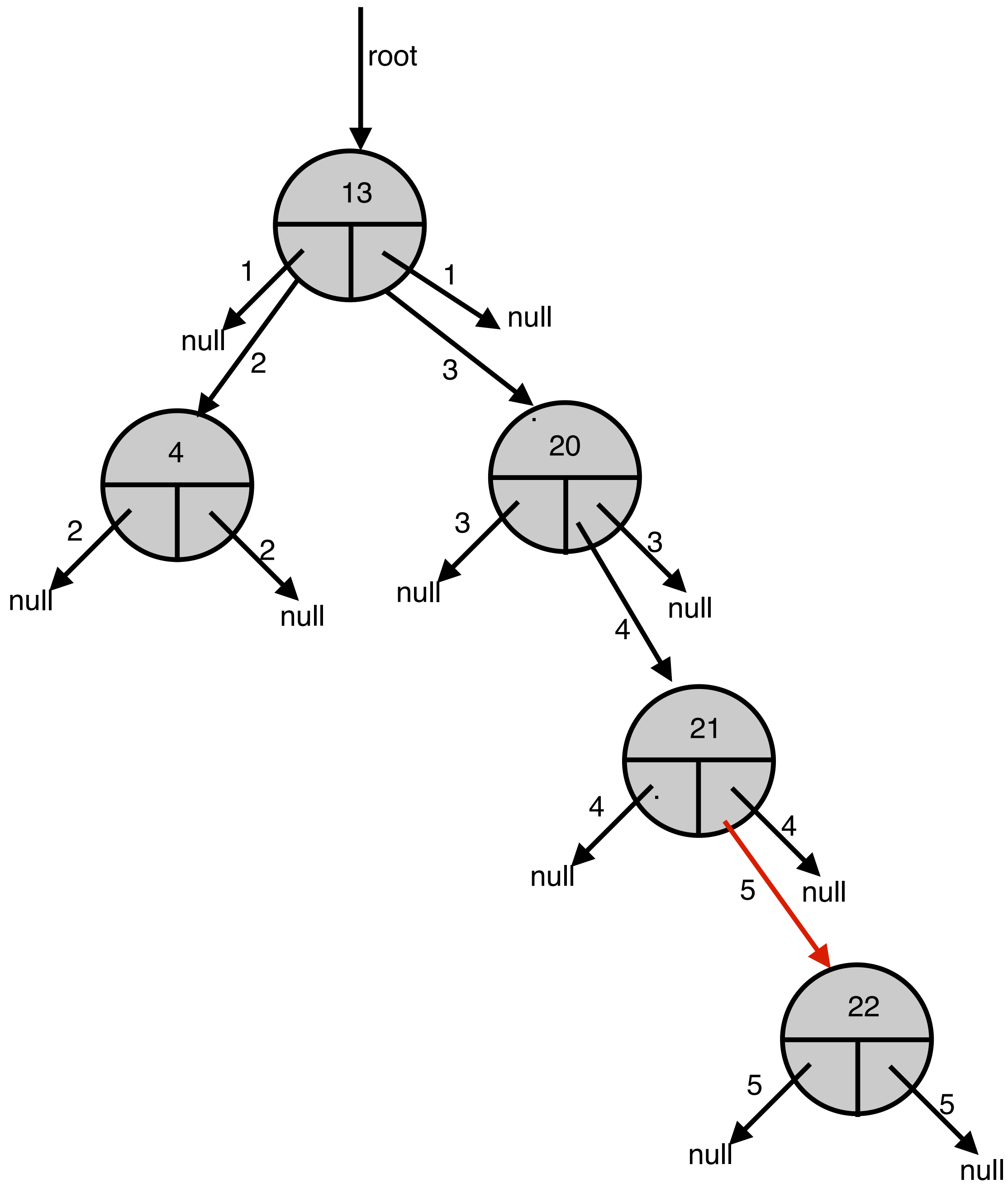
Fat node method example



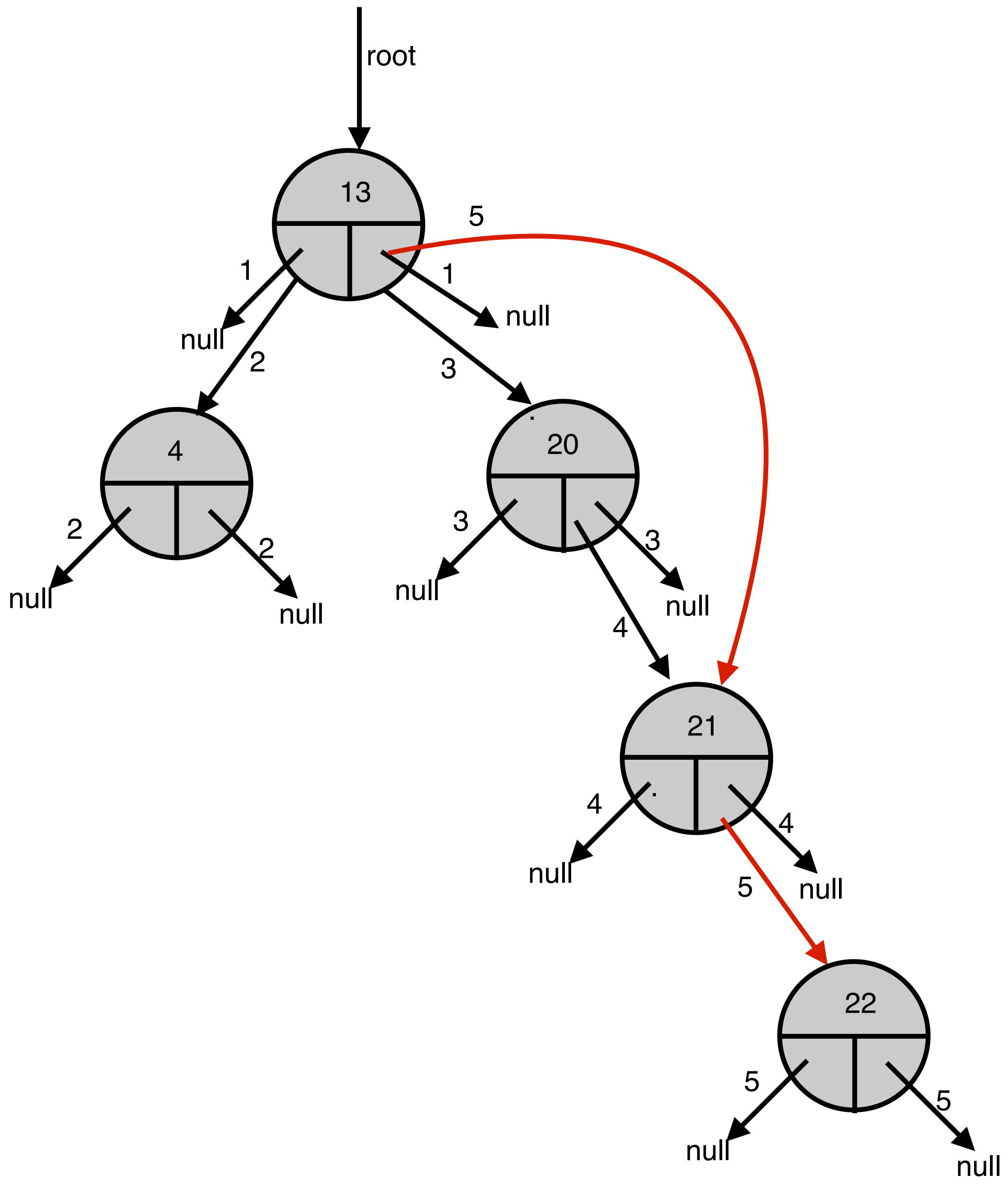
Fat node method example



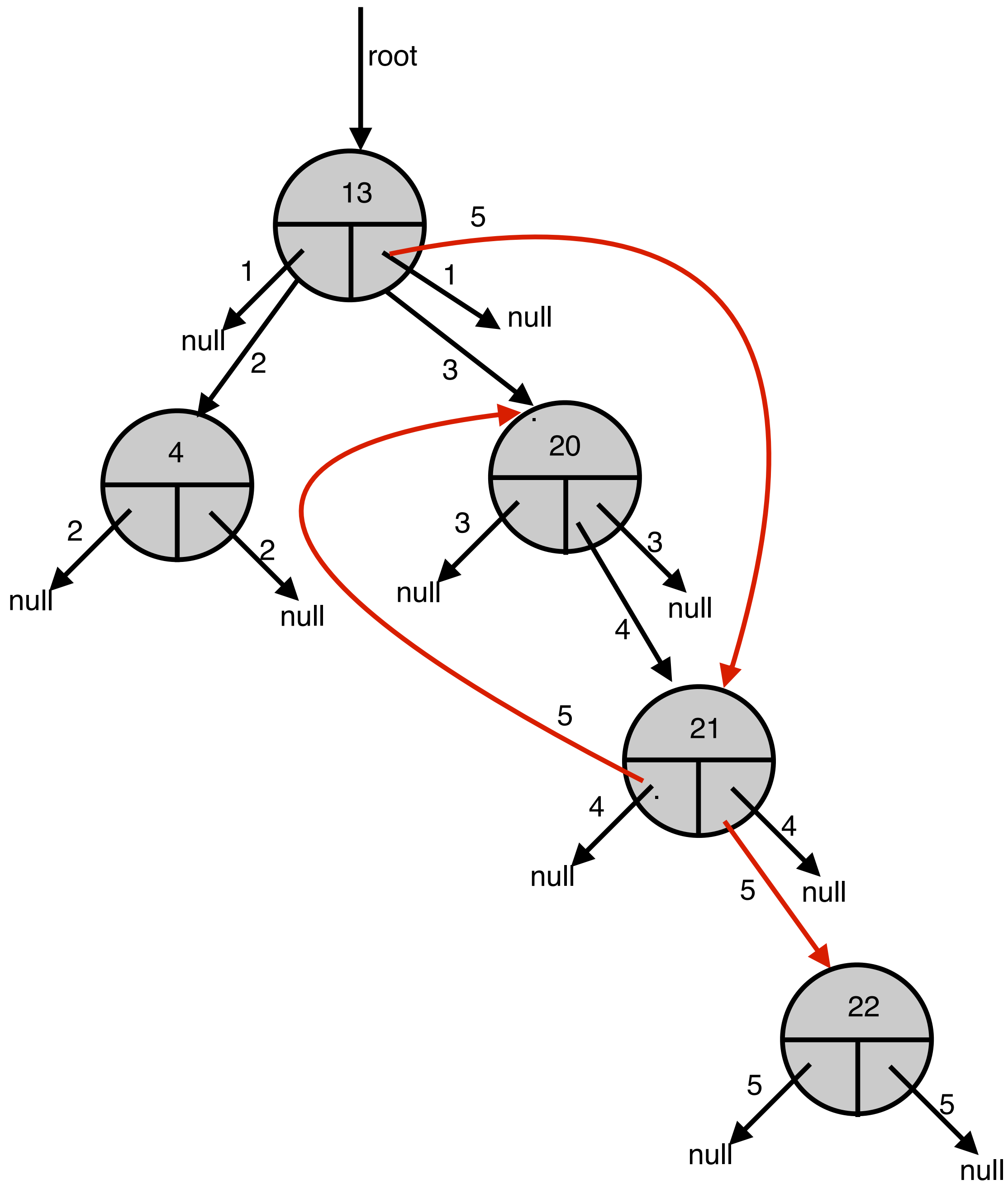
Fat node method example



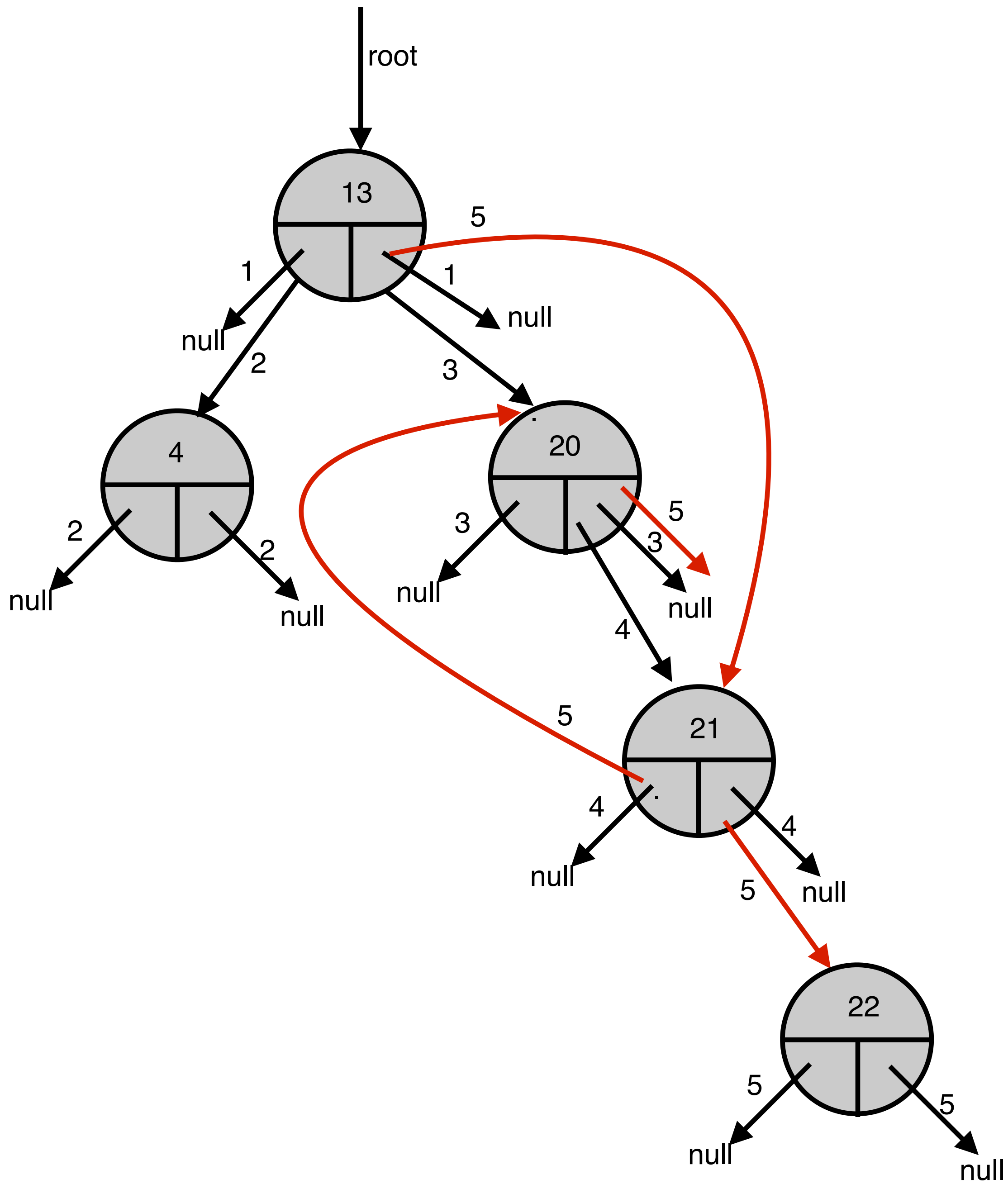
Fat node method example



Fat node method example



Fat node method example



Fat node method

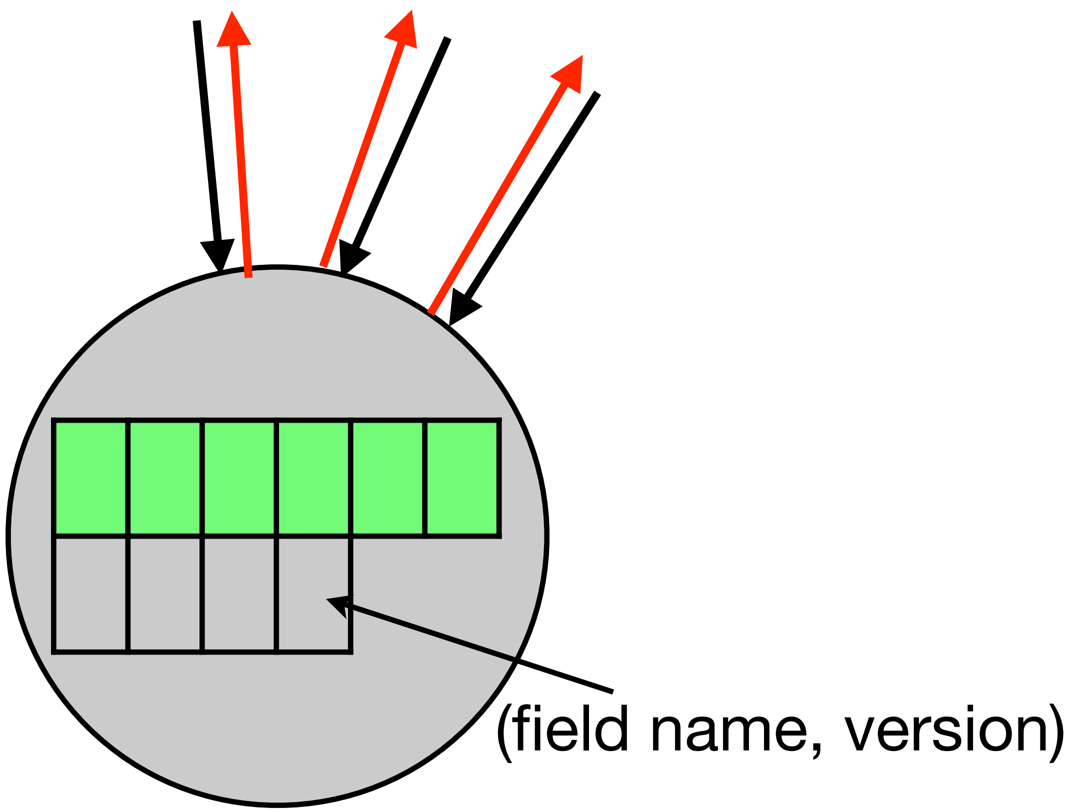
- Driscoll, Sarnak, Sleator, Tarjan, 1989.
  - Any data structure can be made partially persistent with slowdown O(log m) for queries and O(1) for updates. The space cost is O(1) for each ephemeral memory modification.
  - Any data structure can be made partially persistent on a RAM with slowdown O(loglog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



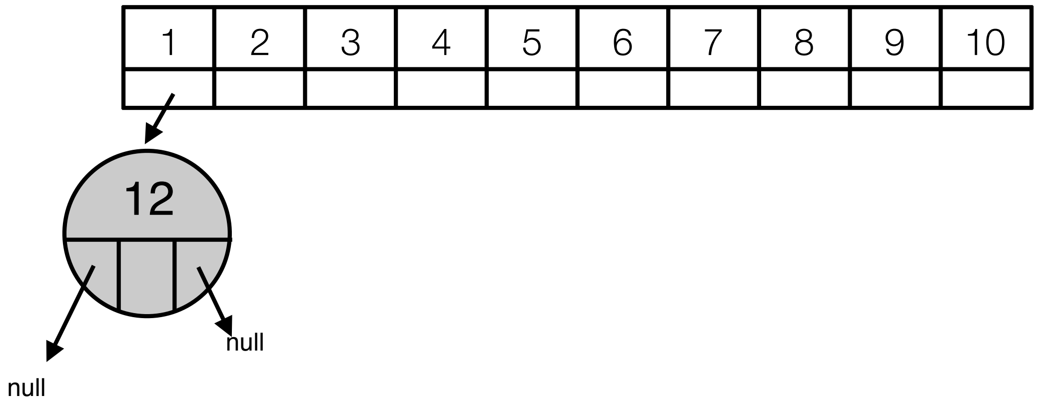
Partially persistent balanced search trees via node copying

- 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.

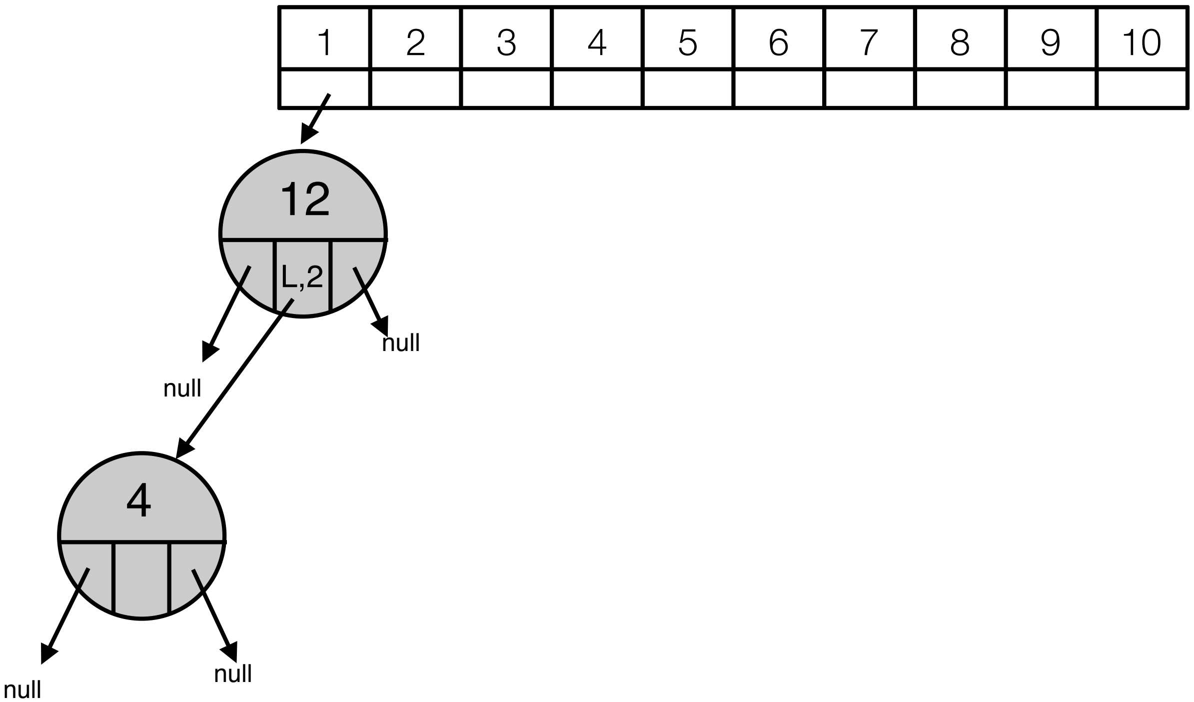
Node copying example

1	2	3	4	5	6	7	8	9	10

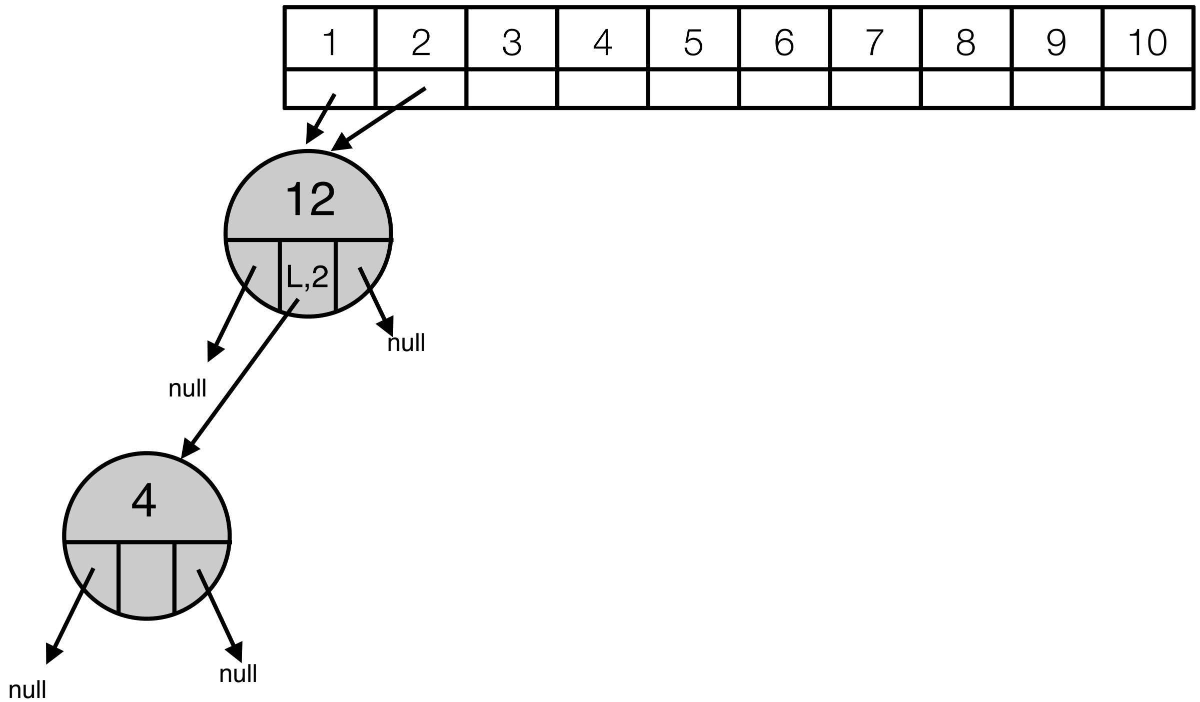
Node copying example



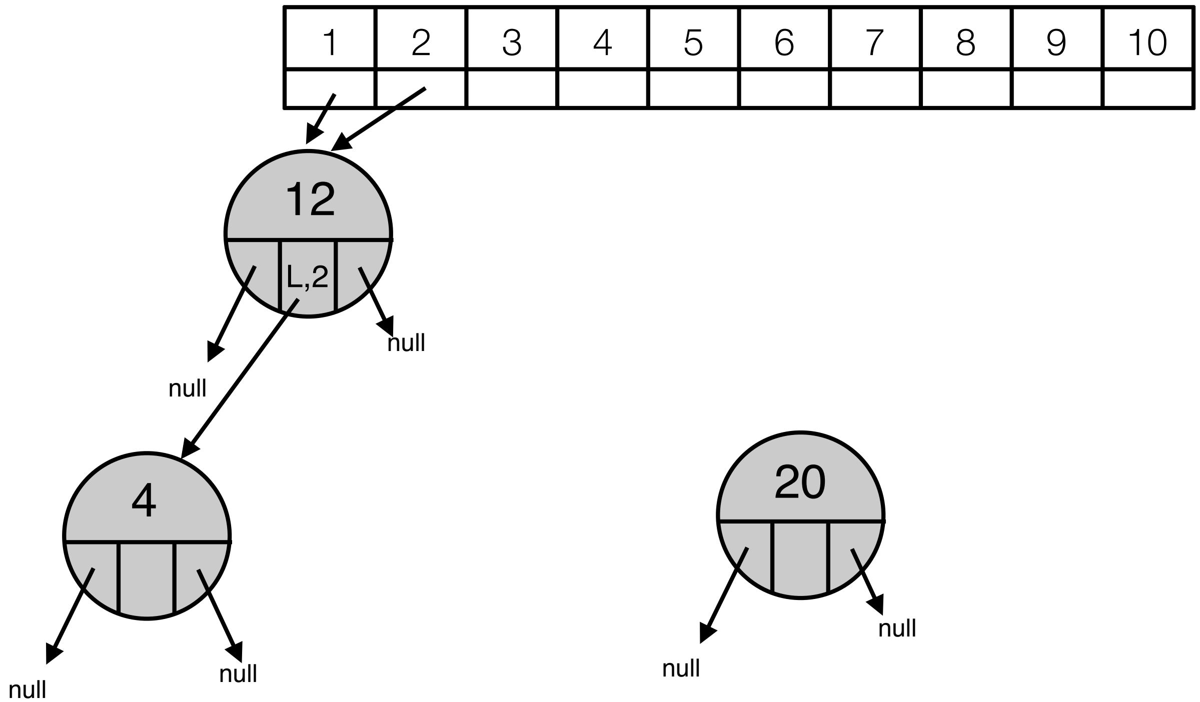
Node copying example



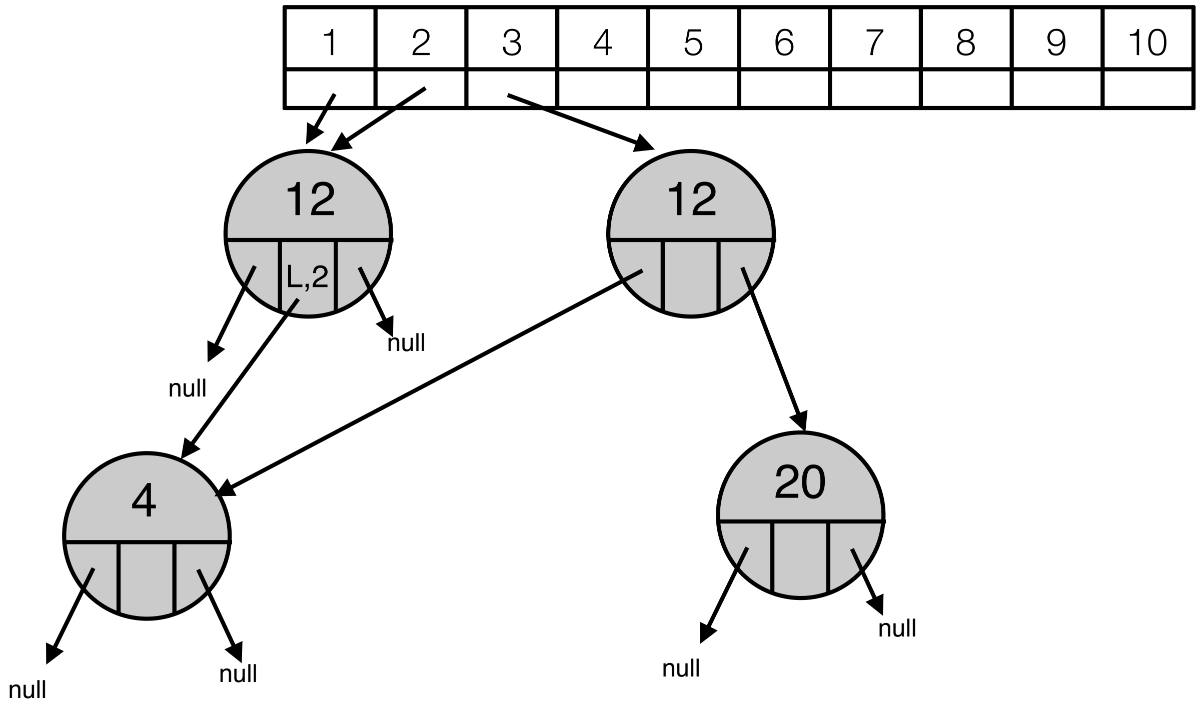
Node copying example



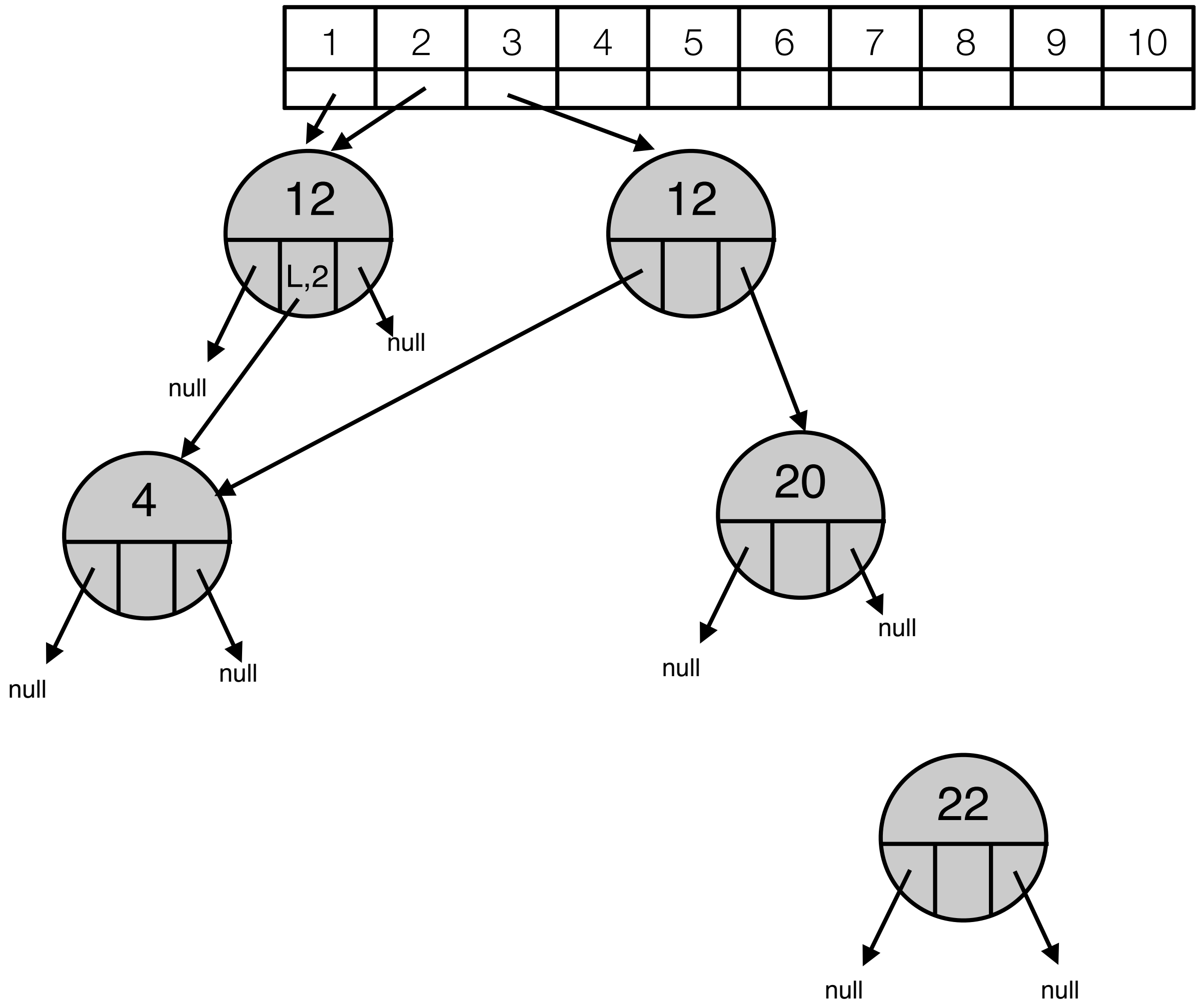
Node copying example



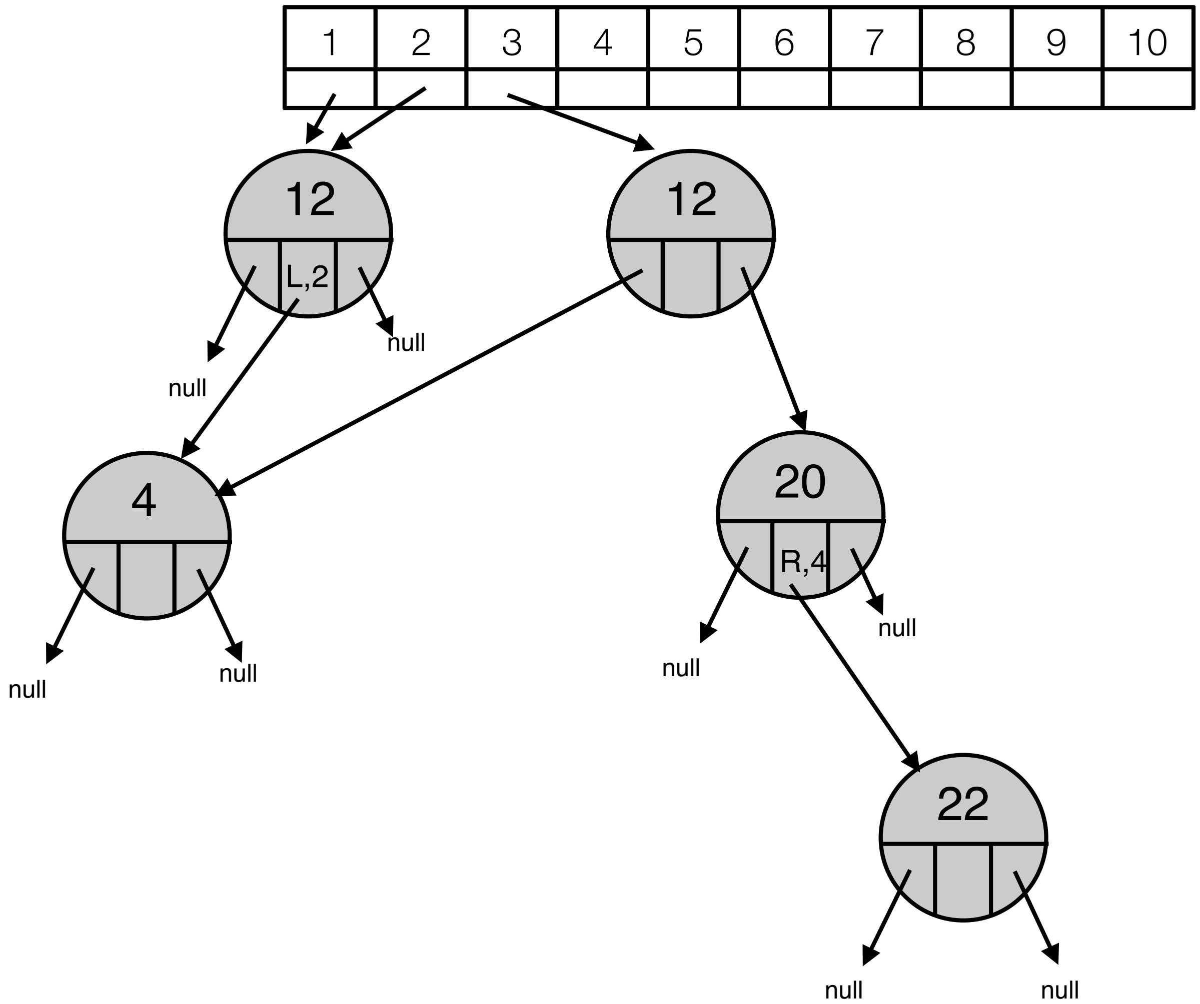
Node copying example



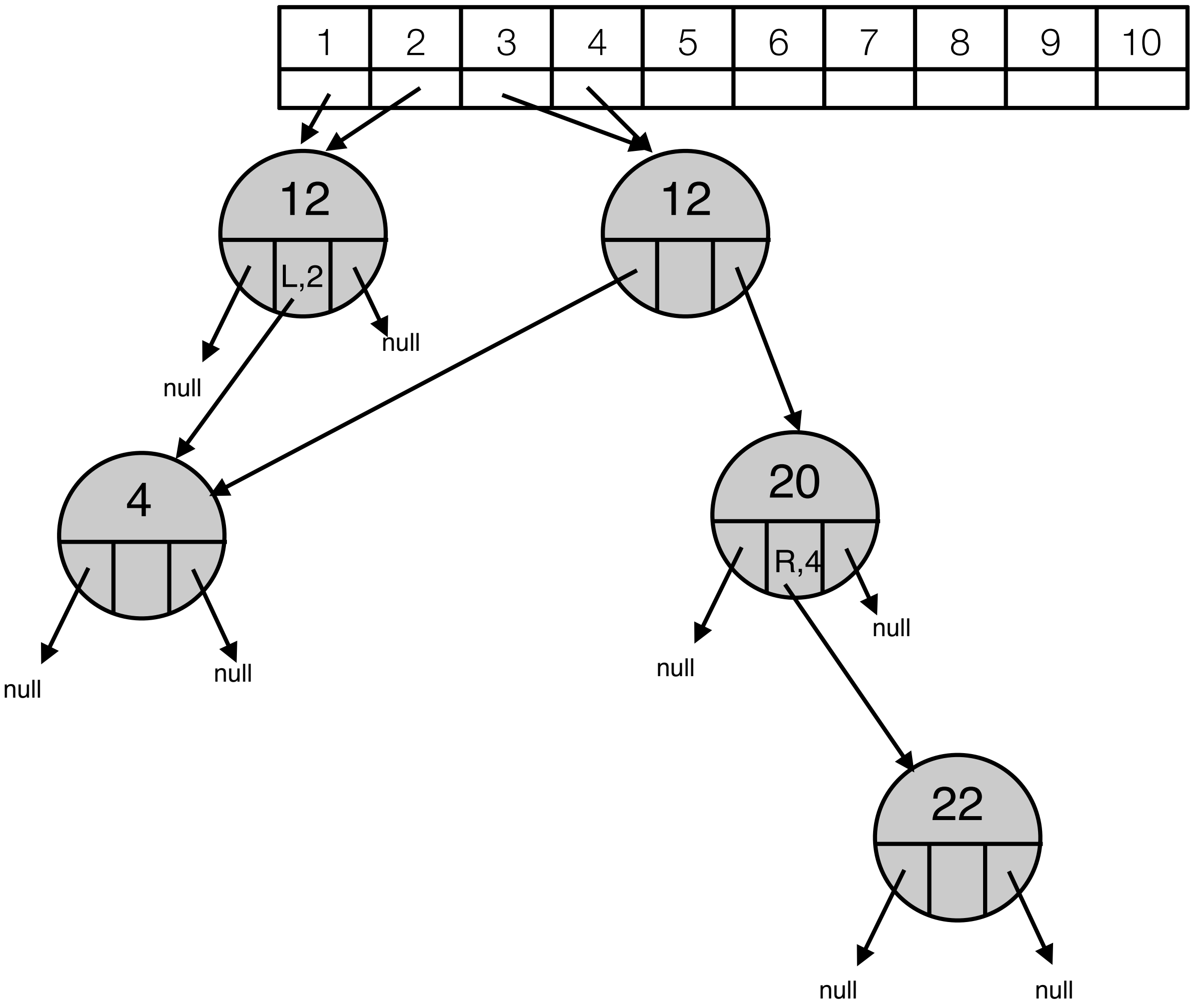
Node copying example



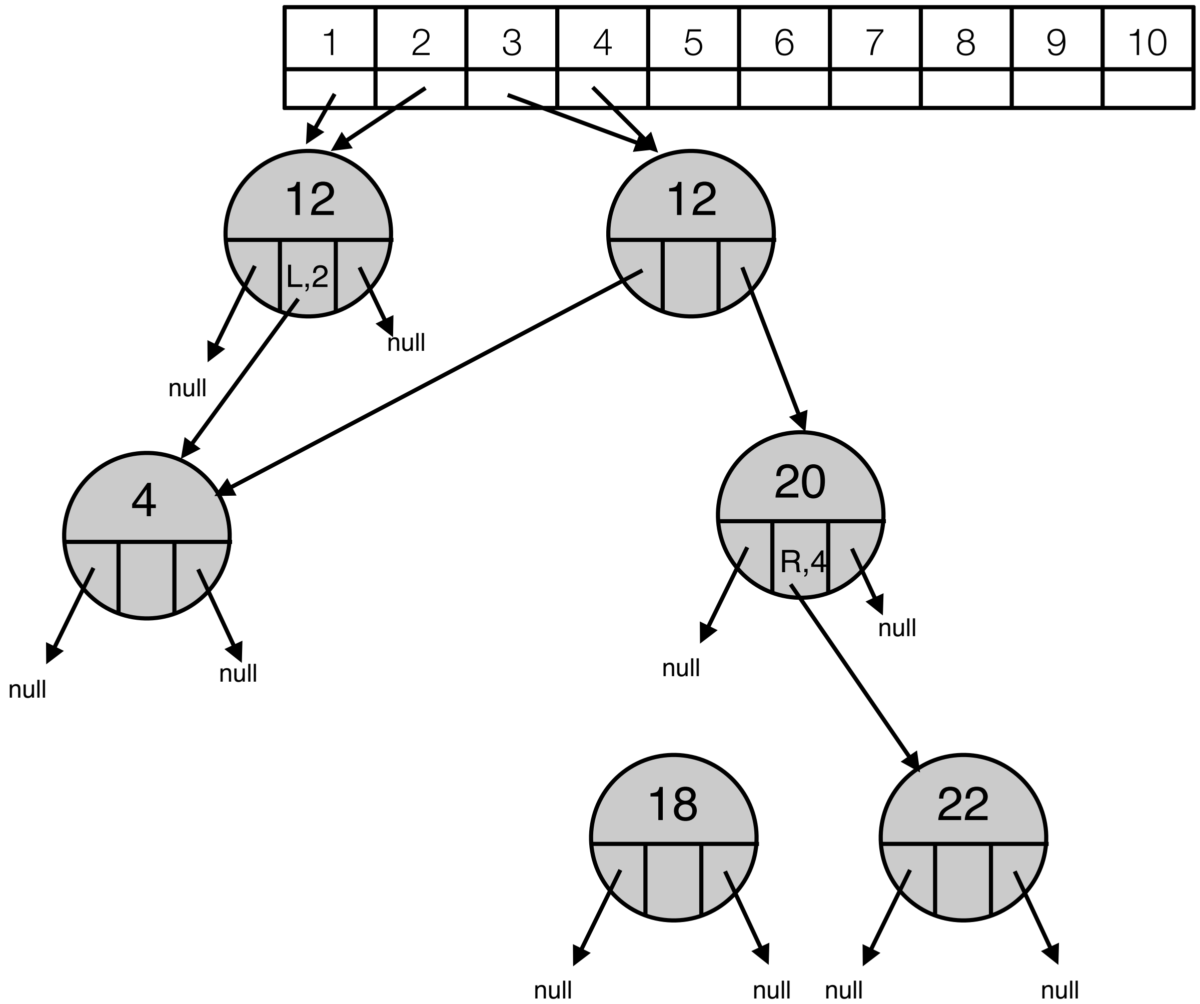
Node copying example



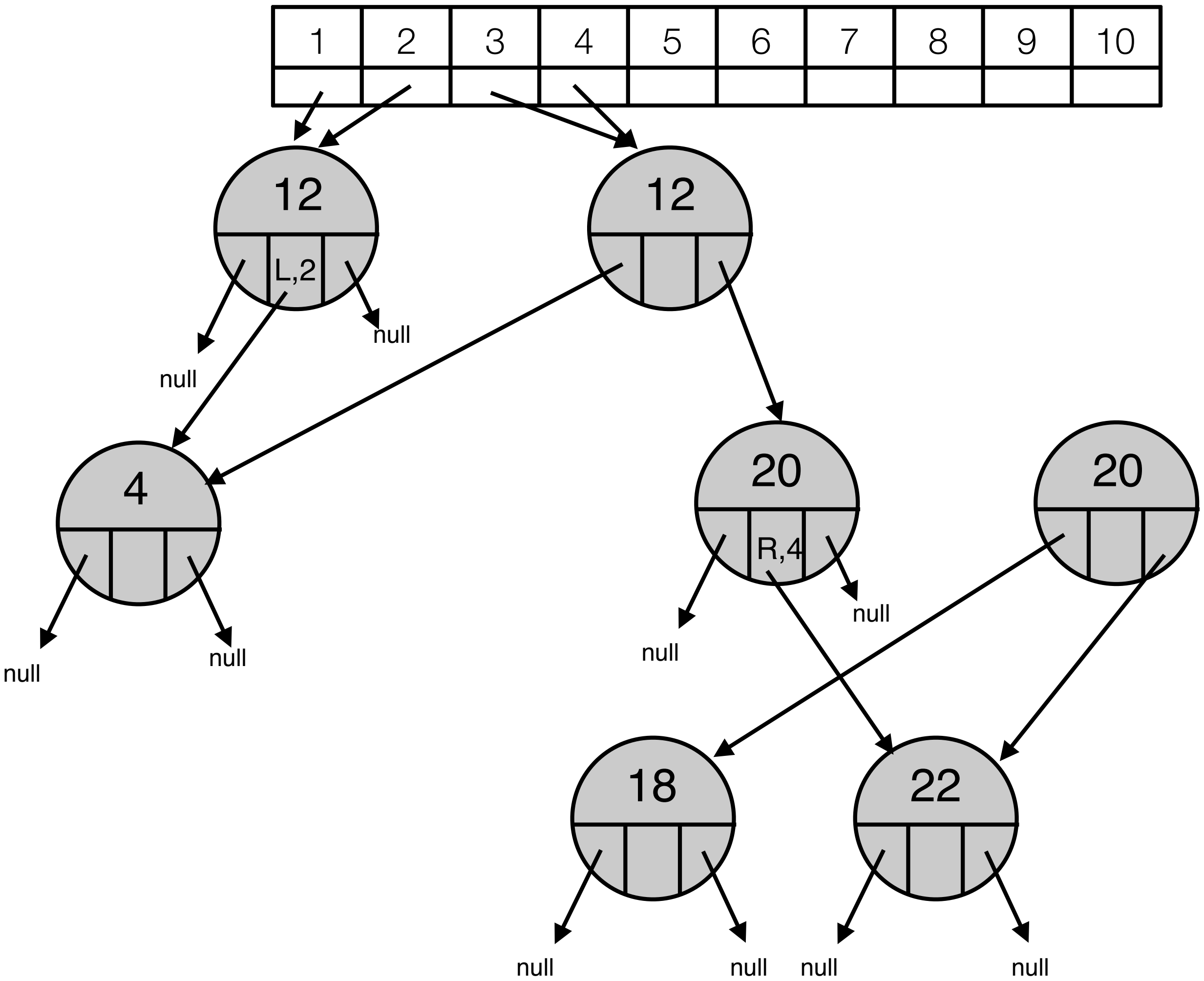
Node copying example



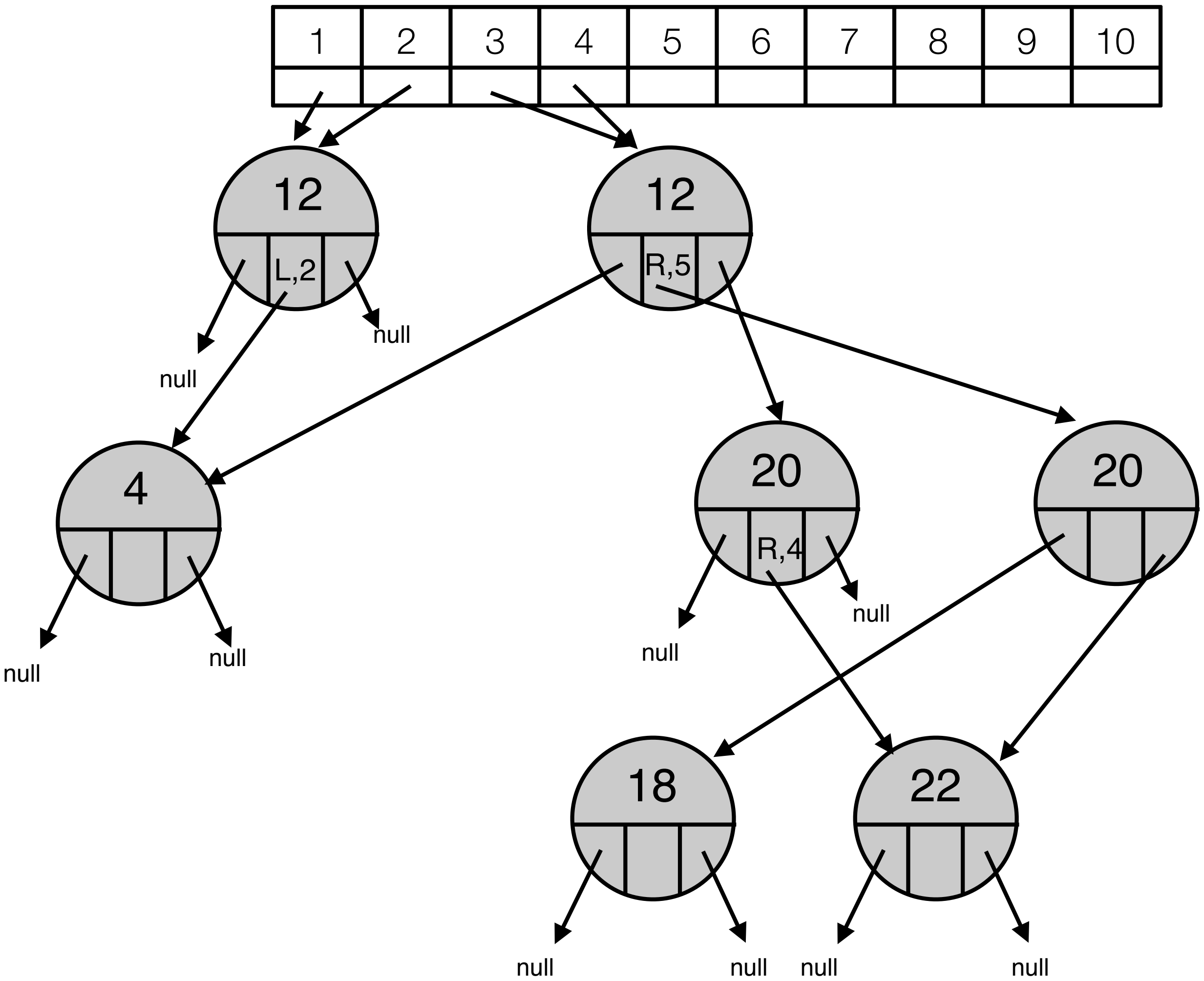
Node copying example



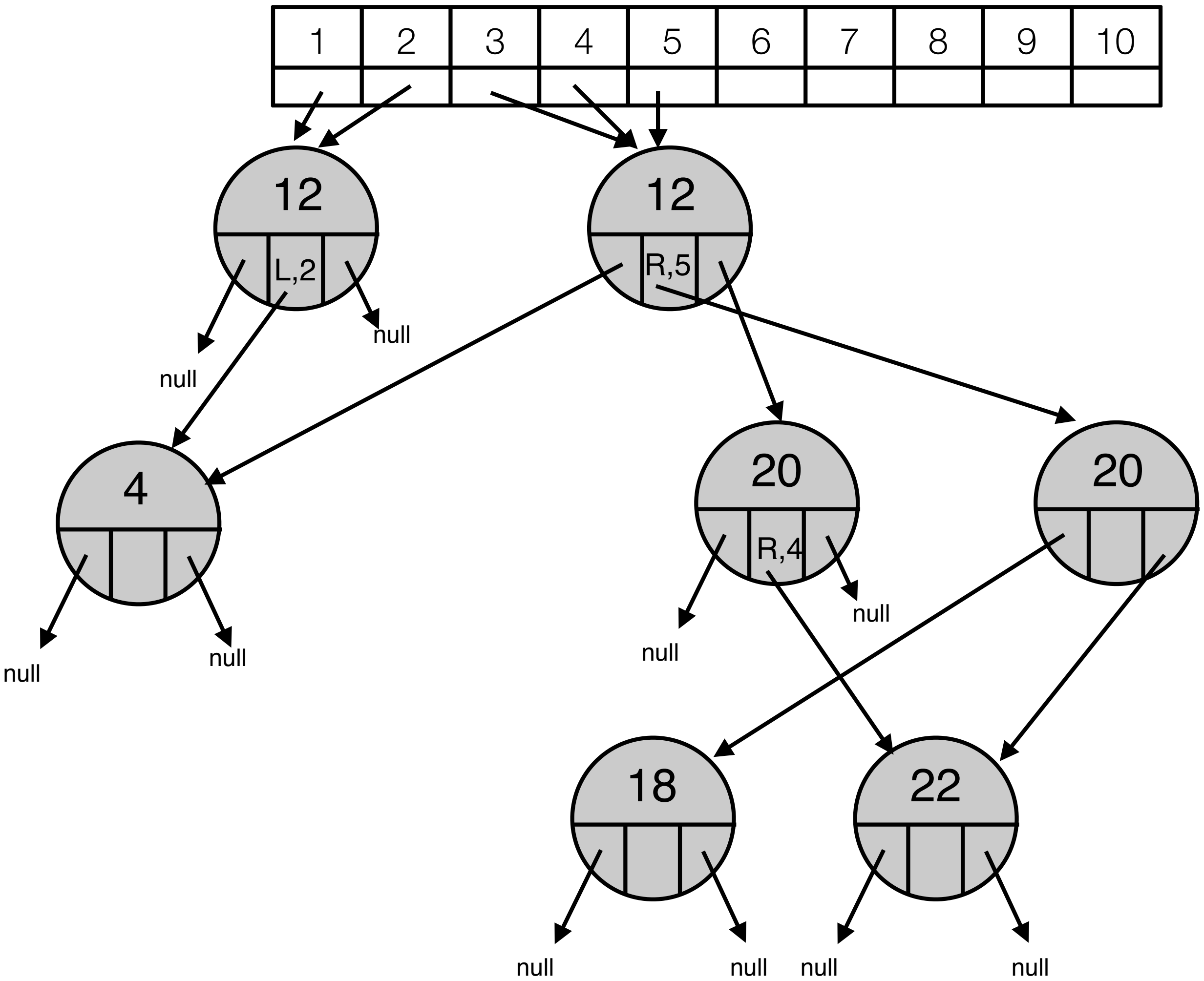
Node copying example



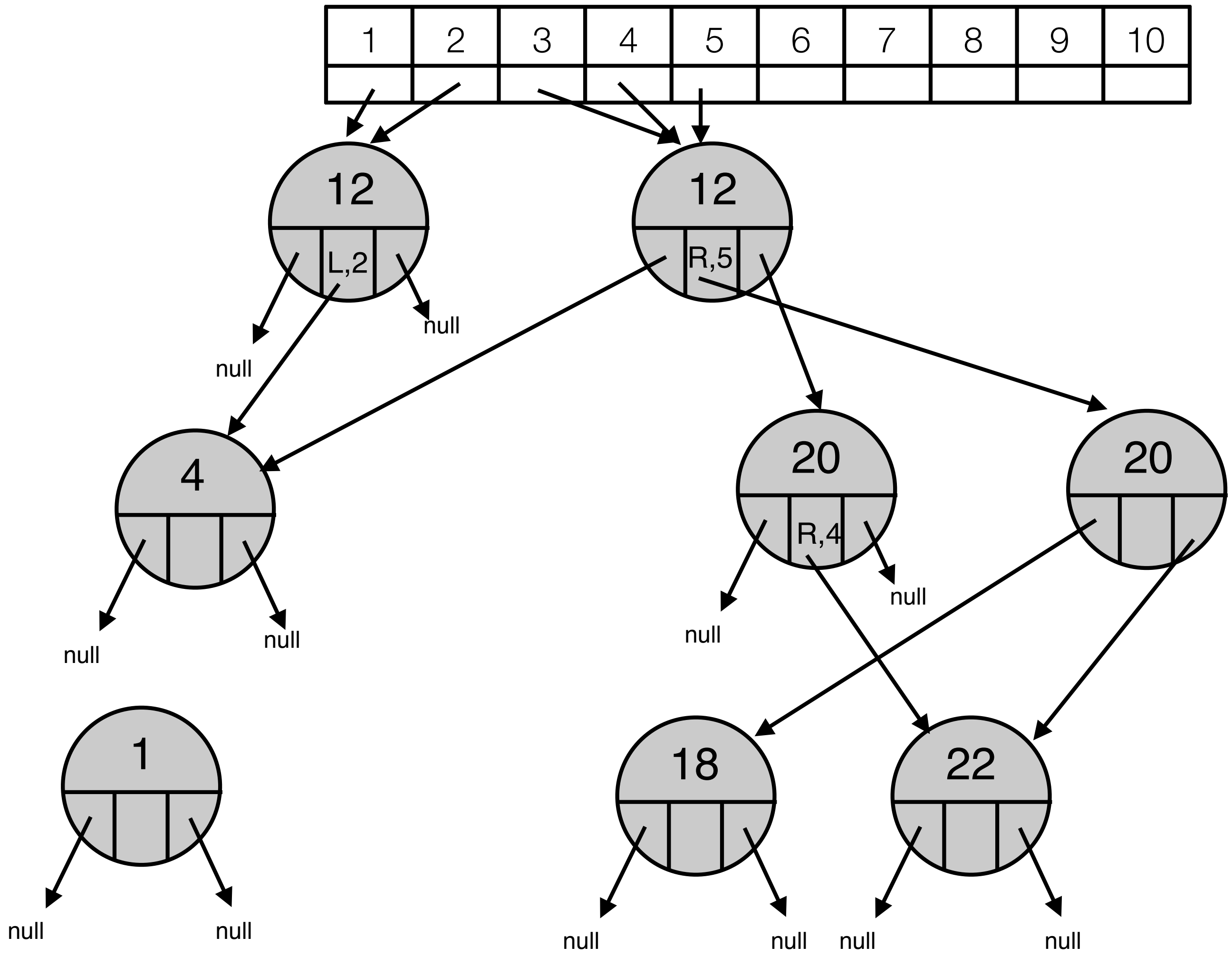
Node copying example



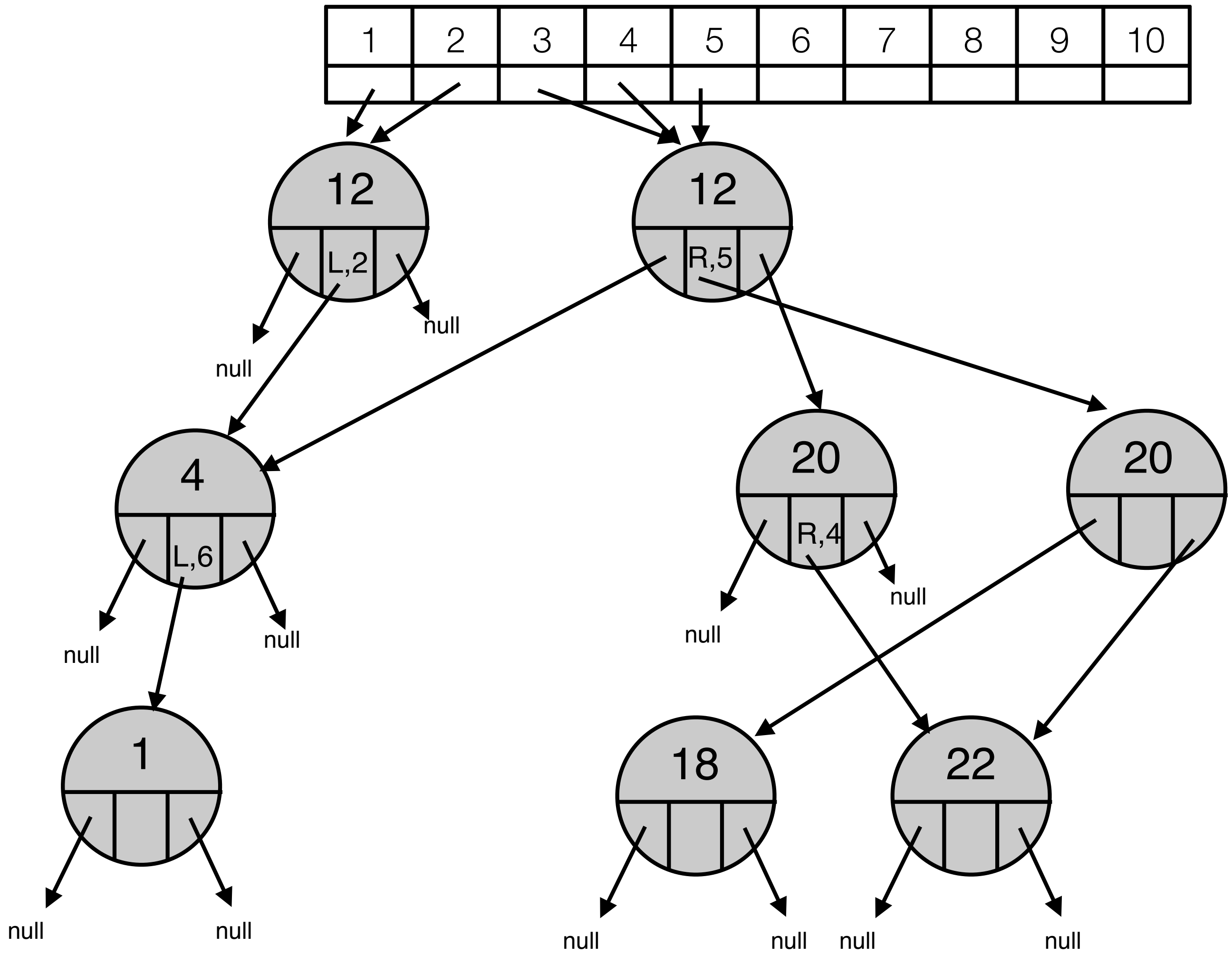
Node copying example



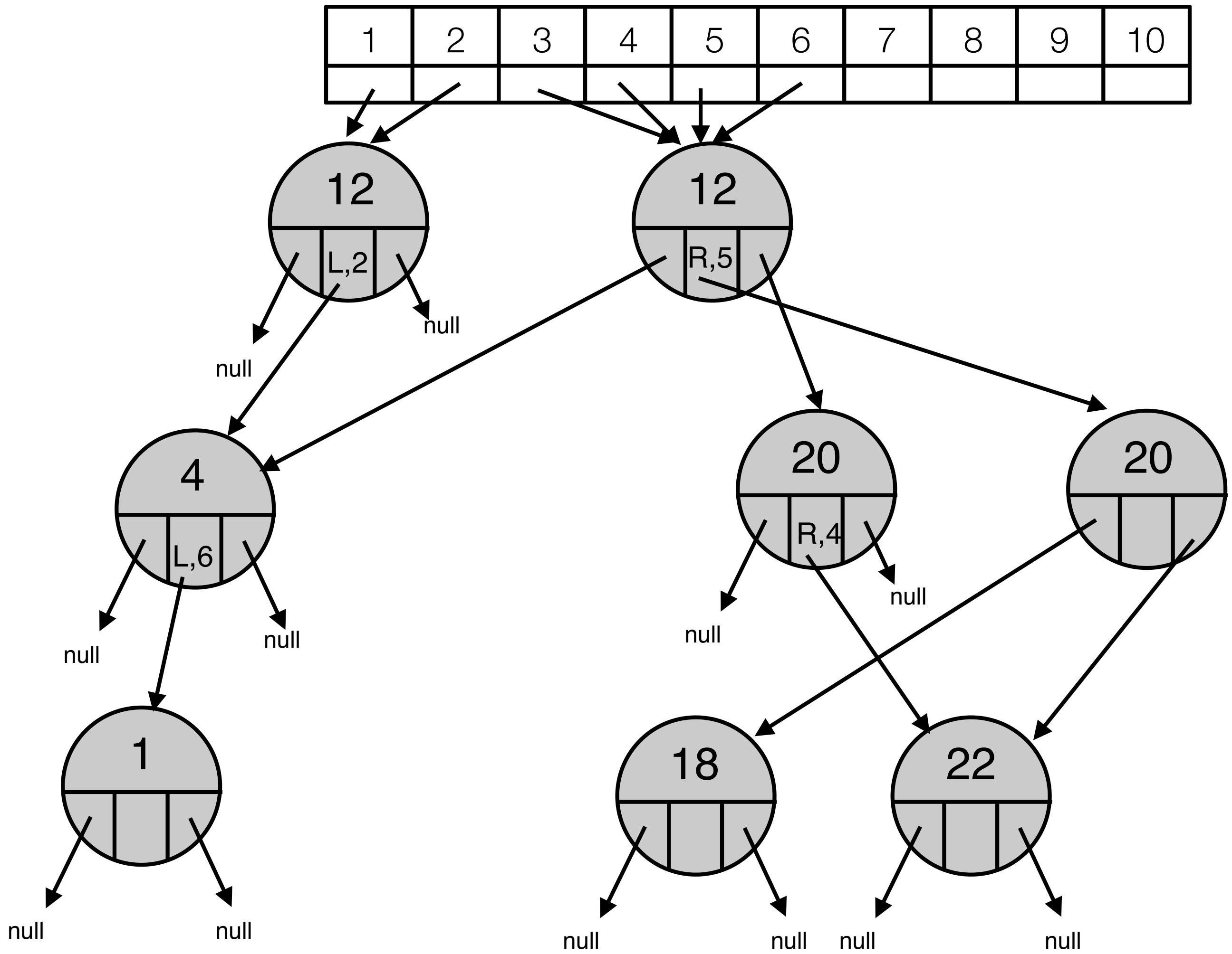
Node copying example



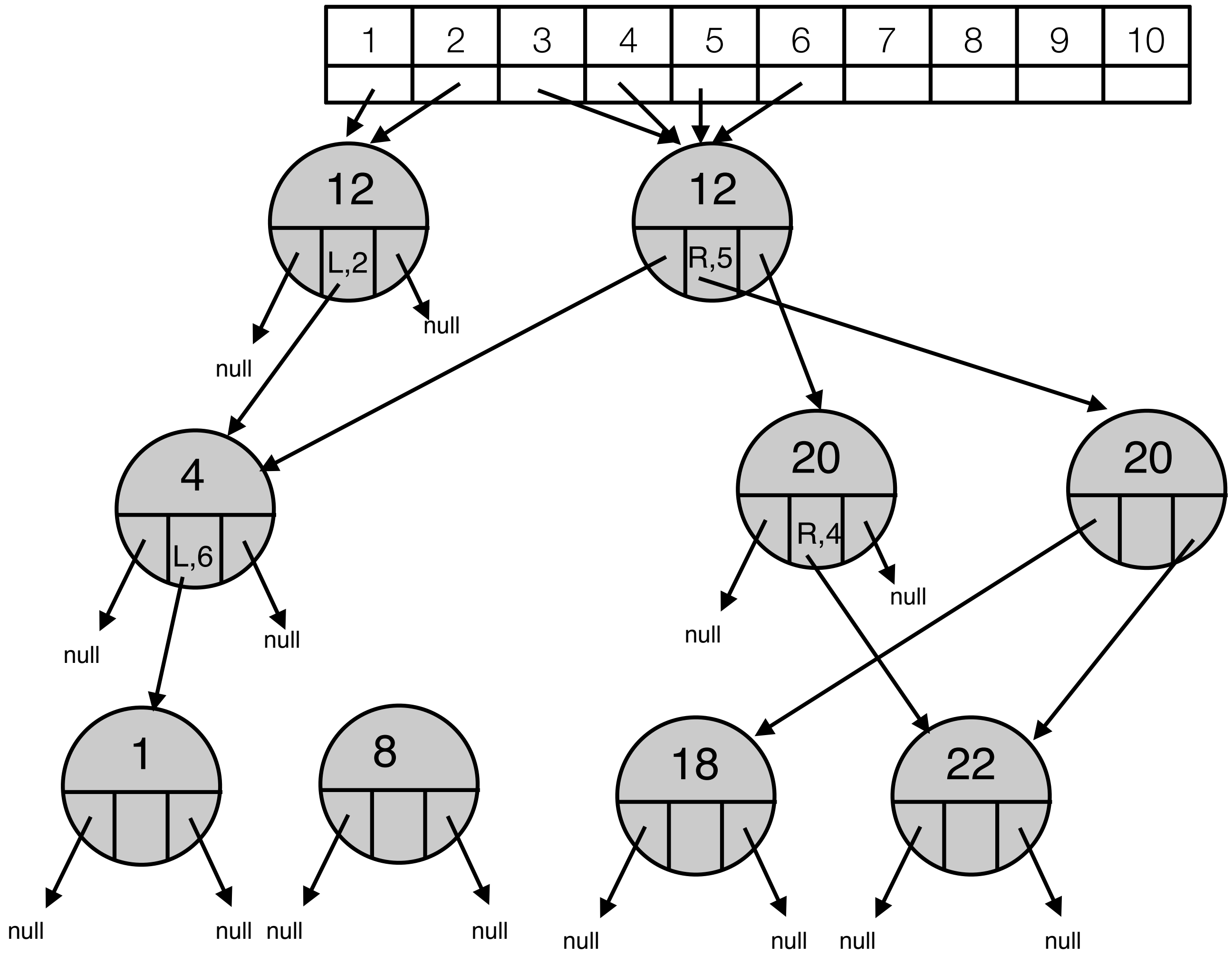
Node copying example



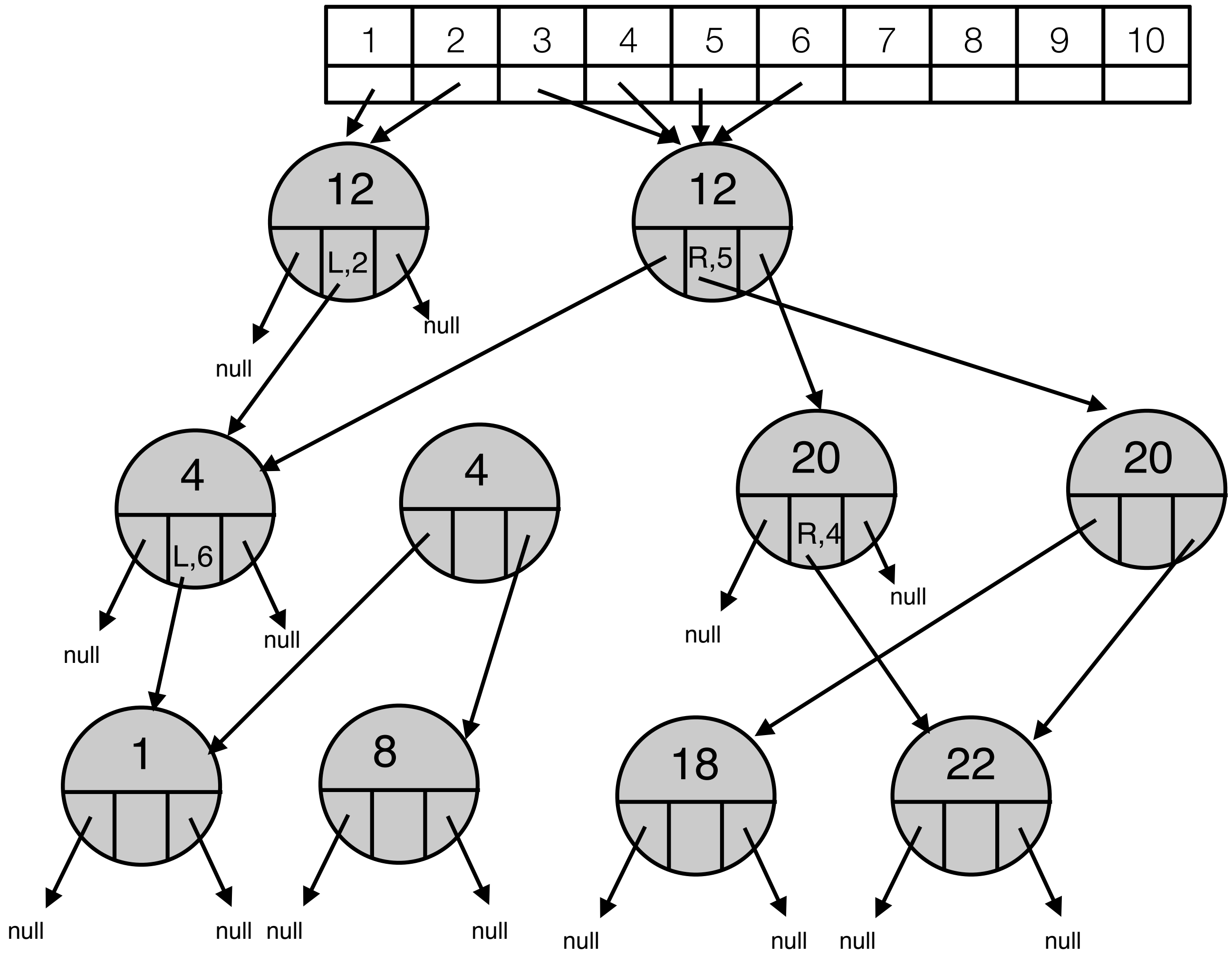
Node copying example



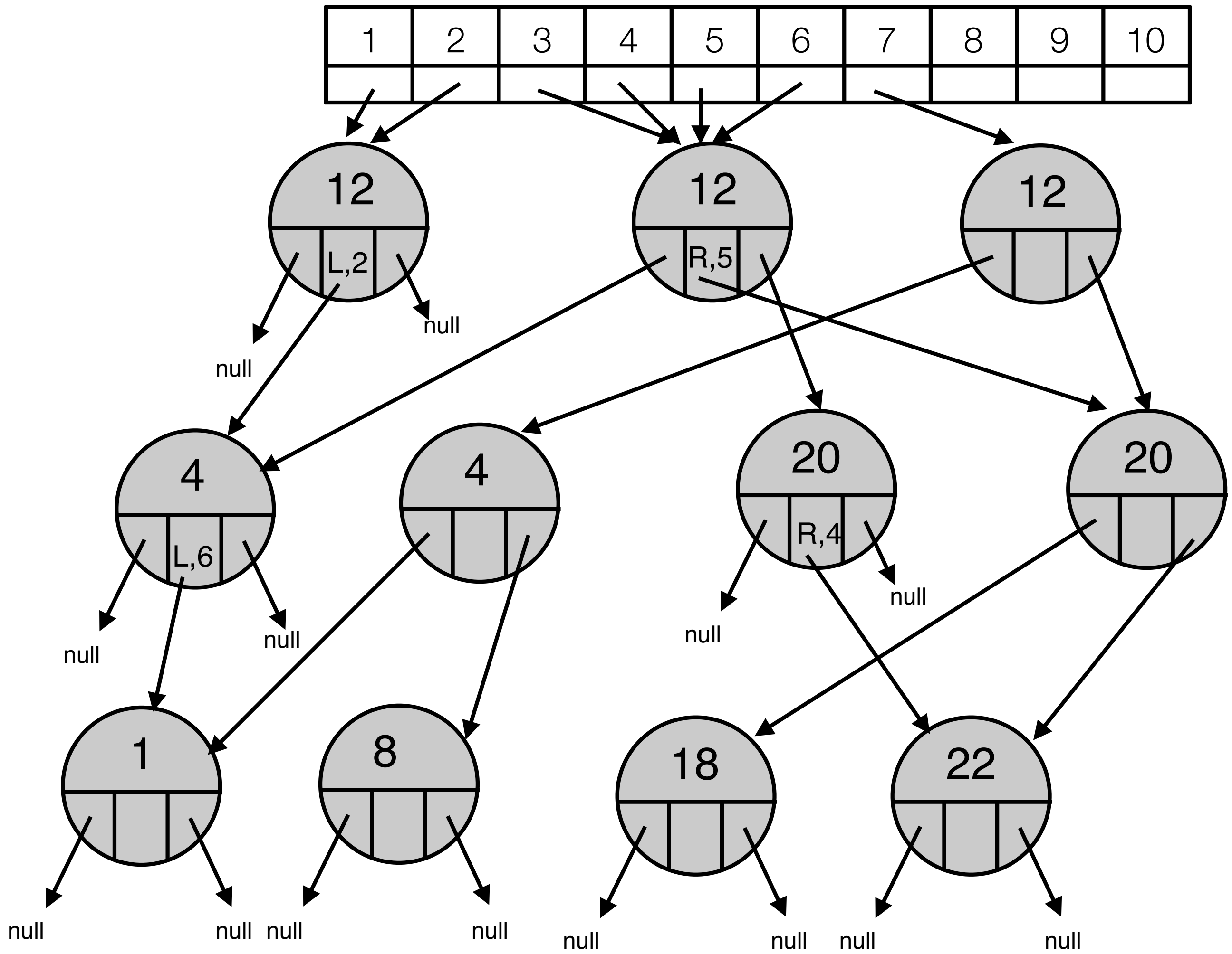
Node copying example



Node copying example



Node copying example



Partially persistent BST with node copying

- Analysis:
  - Time slowdown:
    - access: O(1)
    - updates: O(1) amortized
  - Extra space: 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. Proof uses potential analysis (next time).

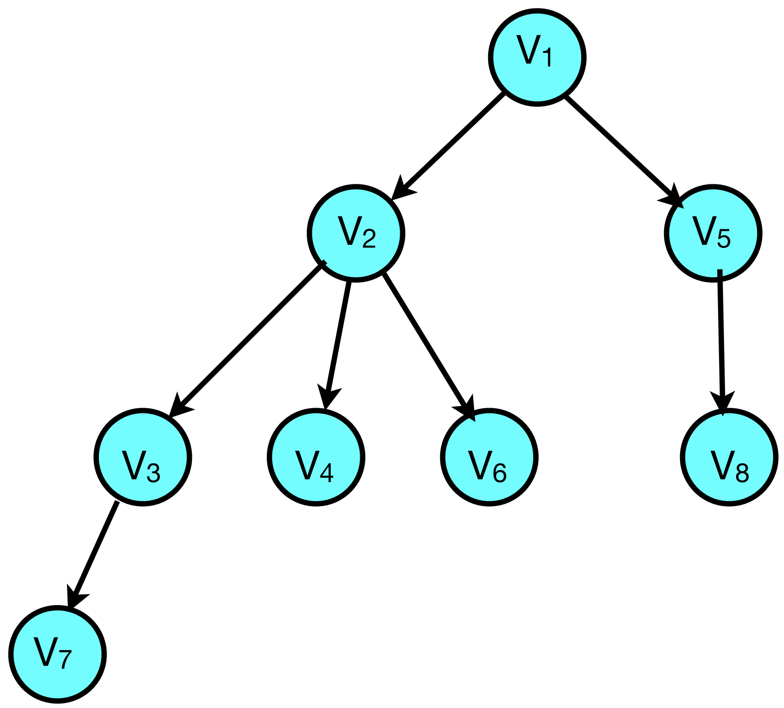
Partially Persistent Data Structures

- 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.

Full Persistence

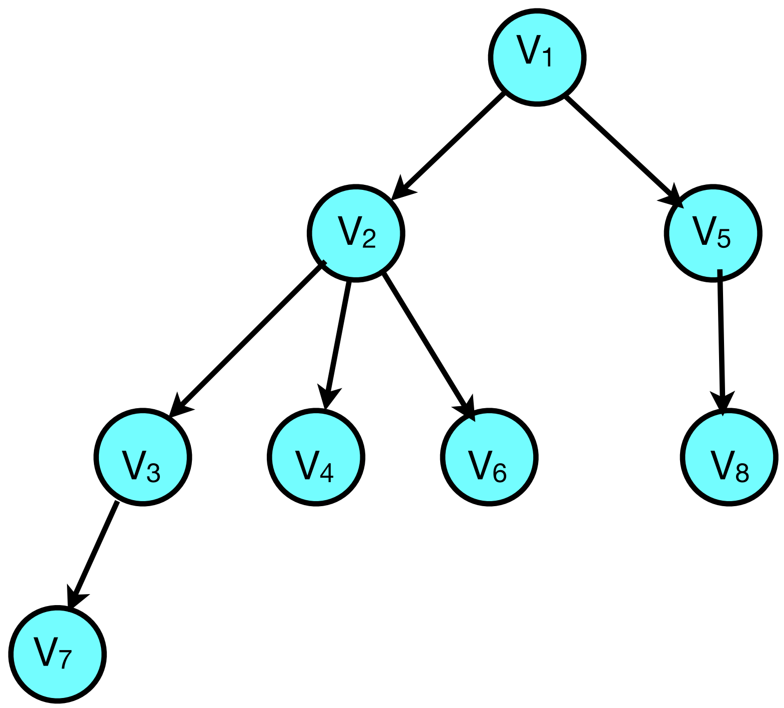
Fat node method

Version tree



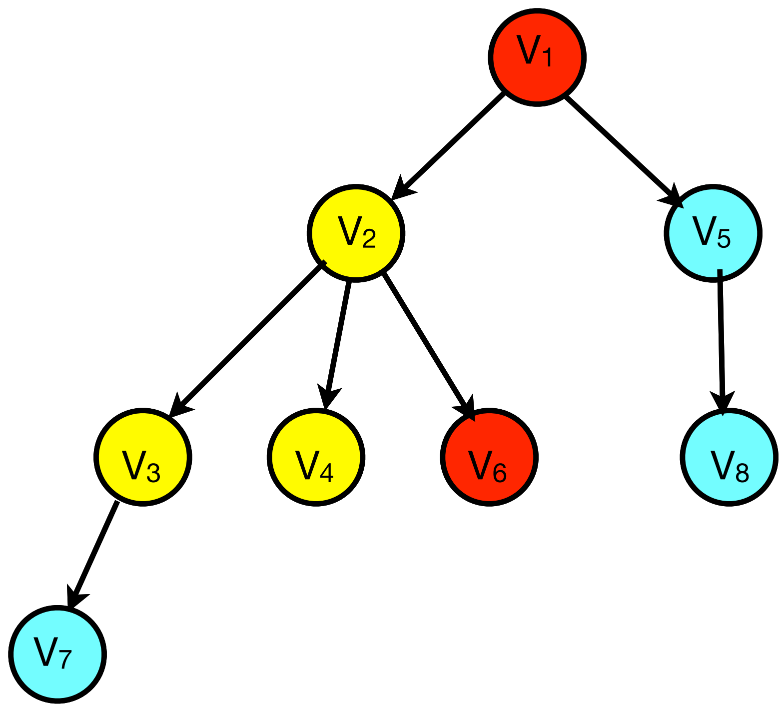
Version tree

- Version tree: partial order.



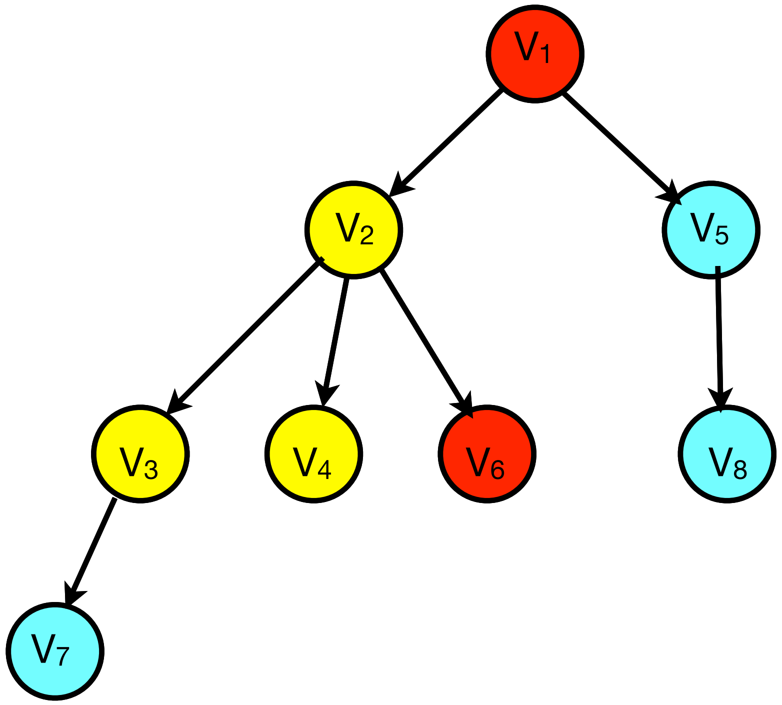
Version tree

- Version tree: partial order.



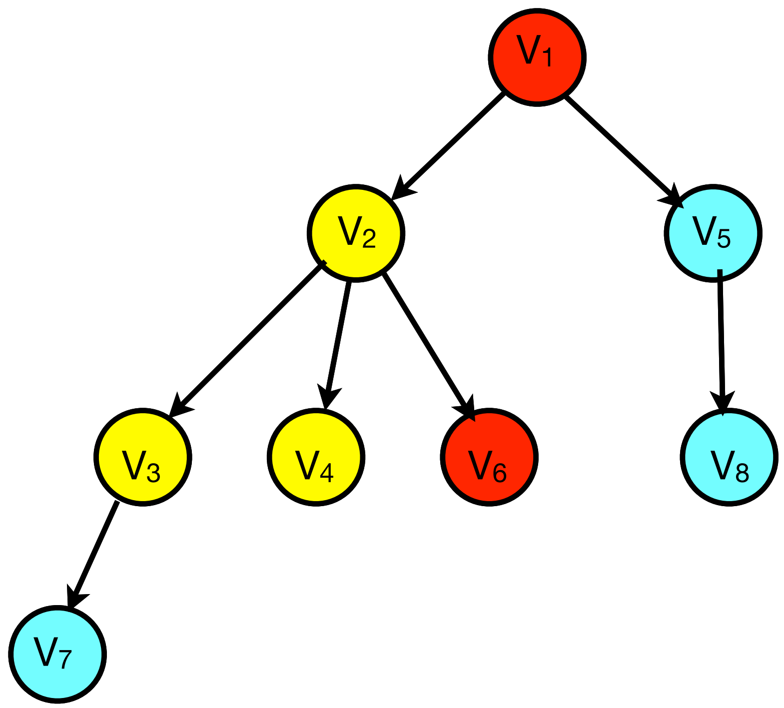
Version tree

- Version tree: partial order.
- Tree color problem:



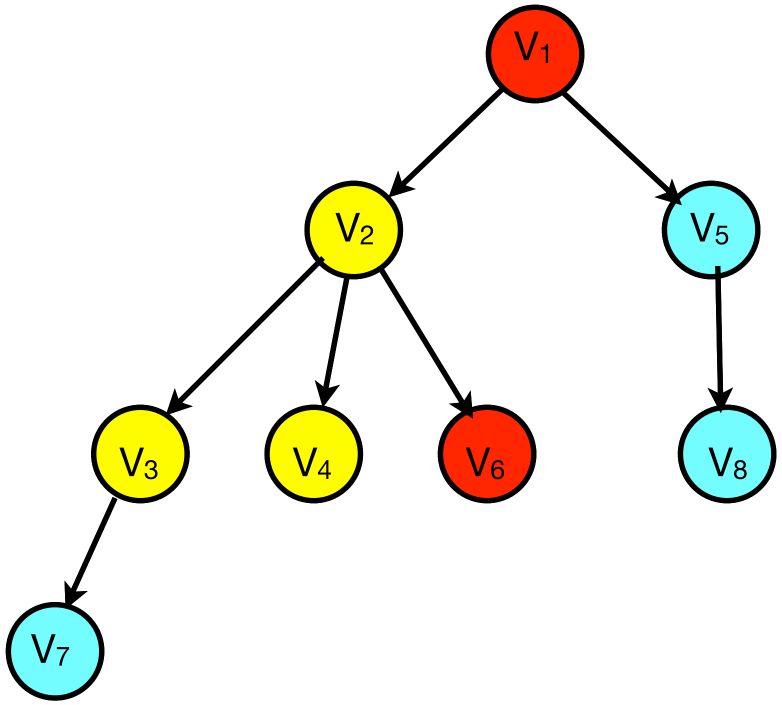
Version tree

- Version tree: partial order.
- Tree color problem:
  - AddLeaf(v, c): Add leaf u as child of v, with color(u)=c.



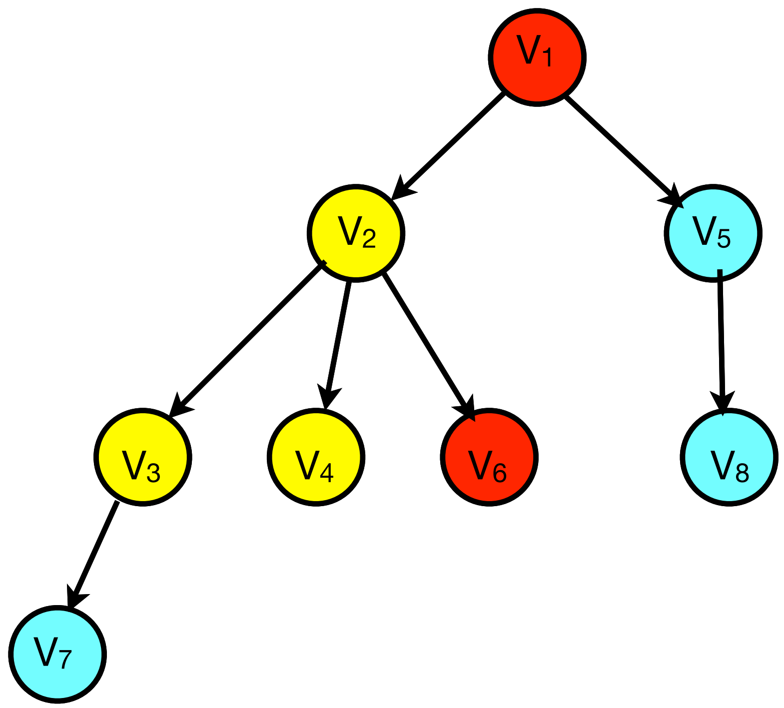
Version tree

- Version tree: partial order.
- Tree color problem:
  - AddLeaf(v, c): Add leaf u as child of v, with color(u)=c.
  - Lookup(v, c): Find nearest ancestor of v with color c.



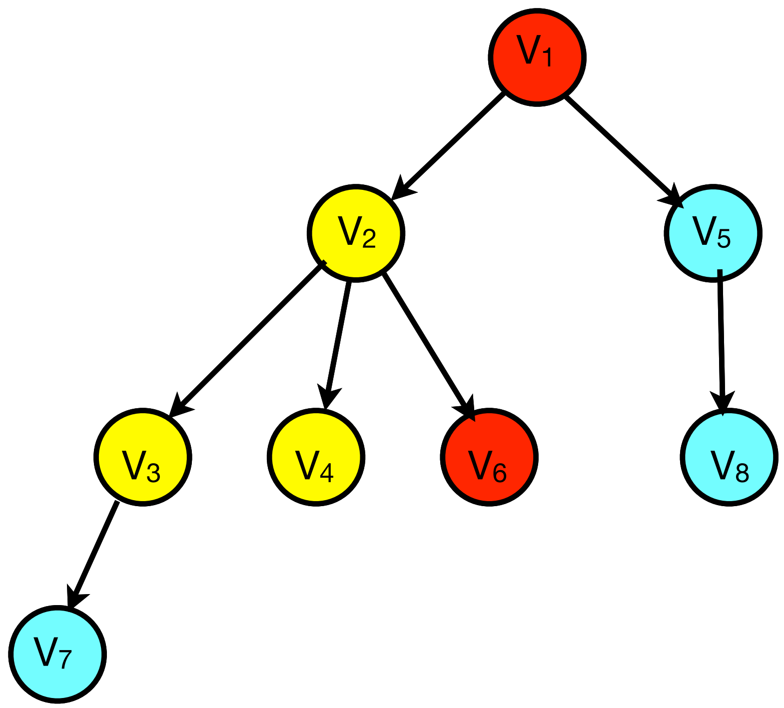
Version tree

- Version tree: partial order.
- Tree color problem:
    - AddLeaf(v, c): Add leaf u as child of v, with color(u)=c.
    - Lookup(v, c): Find nearest ancestor of v with color c.
- Fully persistent array:



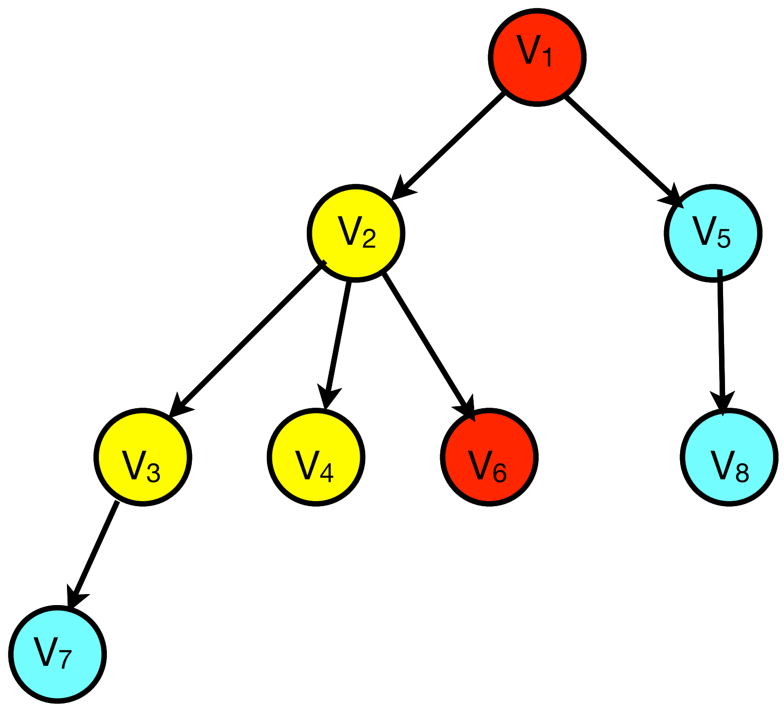
Version tree

- Version tree: partial order.
- Tree color problem:
  - AddLeaf(v, c): Add leaf u as child of v, with color(u)=c.
  - Lookup(v, c): Find nearest ancestor of v with color c.
- Fully persistent array:
  - Store(A, i, x, t): Set A[i]=x at time t
    ~ AddLeaf(t, i), value v = x.



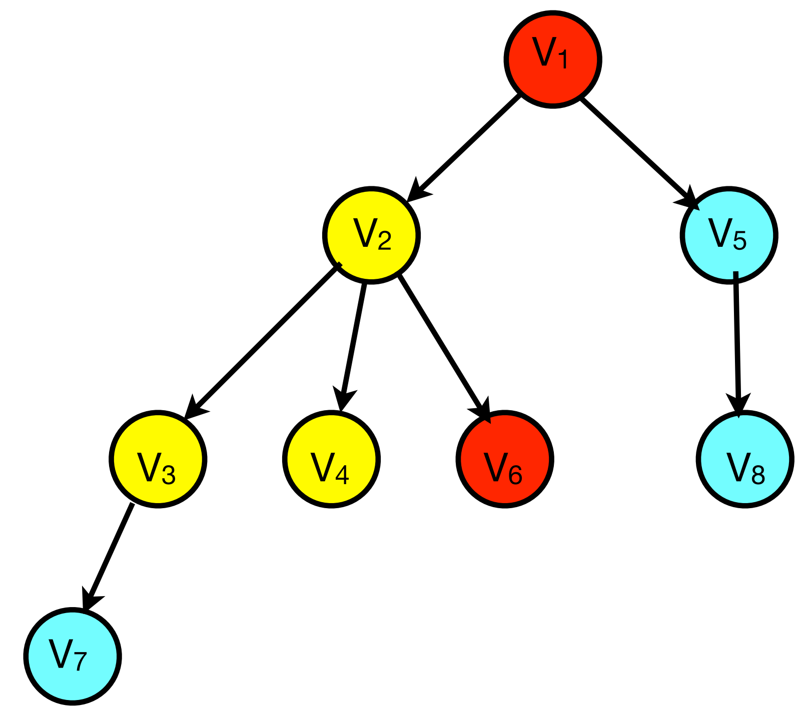
Version tree

- Version tree: partial order.
- Tree color problem:
  - AddLeaf(v, c): Add leaf u as child of v, with color(u)=c.
  - Lookup(v, c): Find nearest ancestor of v with color c.
- Fully persistent array:
  - Store(A, i, x, t): Set A[i]=x at time t
    ~ AddLeaf(t, i), value v = x.
  - Access(A, i, t): Lookup value A[i] at time t
    ~ Lookup(t, i)



Version tree and version list

- Euler tour: $L(T) = (v_1, v_2, v_3, v_7, v_{7'}, v_{3'}, v_4, v_{4'}, v_6, v_{6'}, v_{2'}, v_5, v_8, v_{8'}, v_{5'}, v_{1'})$
- Partition list for each color:
  - $L(1) = (v_1, v_2, v_3, v_7, v_{7'}, v_{3'}, v_4, v_{4'}), (v_6, v_{6'}), (v_{2'}, v_5, v_8, v_{8'}, v_{5'}, v_{1'})$
  - $L(2) = (v_1), (v_2), (v_3, v_7, v_{7'}, v_{3'}), (v_4, v_{4'}), (v_6, v_{6'}, v_{2'}, v_5, v_8, v_{8'}, v_{5'}, v_{1'})$
  - $L(3) = (v_1, v_2, v_3, v_7, v_{7'}, v_{3'}, v_4, v_{4'}, v_6, v_{6'}, v_{2'}), (v_5), (v_8, v_{8'}), (v_{5'}, v_{1'})$



- Predecessor data structure for each color to find right sublist.
- *Maintaining order in a list problem:* O(1) time.

Fully Persistent Data Structures

- Driscoll, Sarnak, Sleator, Tarjan, 1989.
  - Any data structure can be made fully persistent with slowdown O(log m) for both queries and updates. The space cost is O(1) for each ephemeral memory modification.
  - Any bounded-degree linked data structure can be made fully persistent with (worst-case) slowdown O(1) for queries, amortized slowdown O(1) for updates, and amortized space cost O(1) per memory modification.
- Dietz, 1989. Any data structure can be made fully 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.

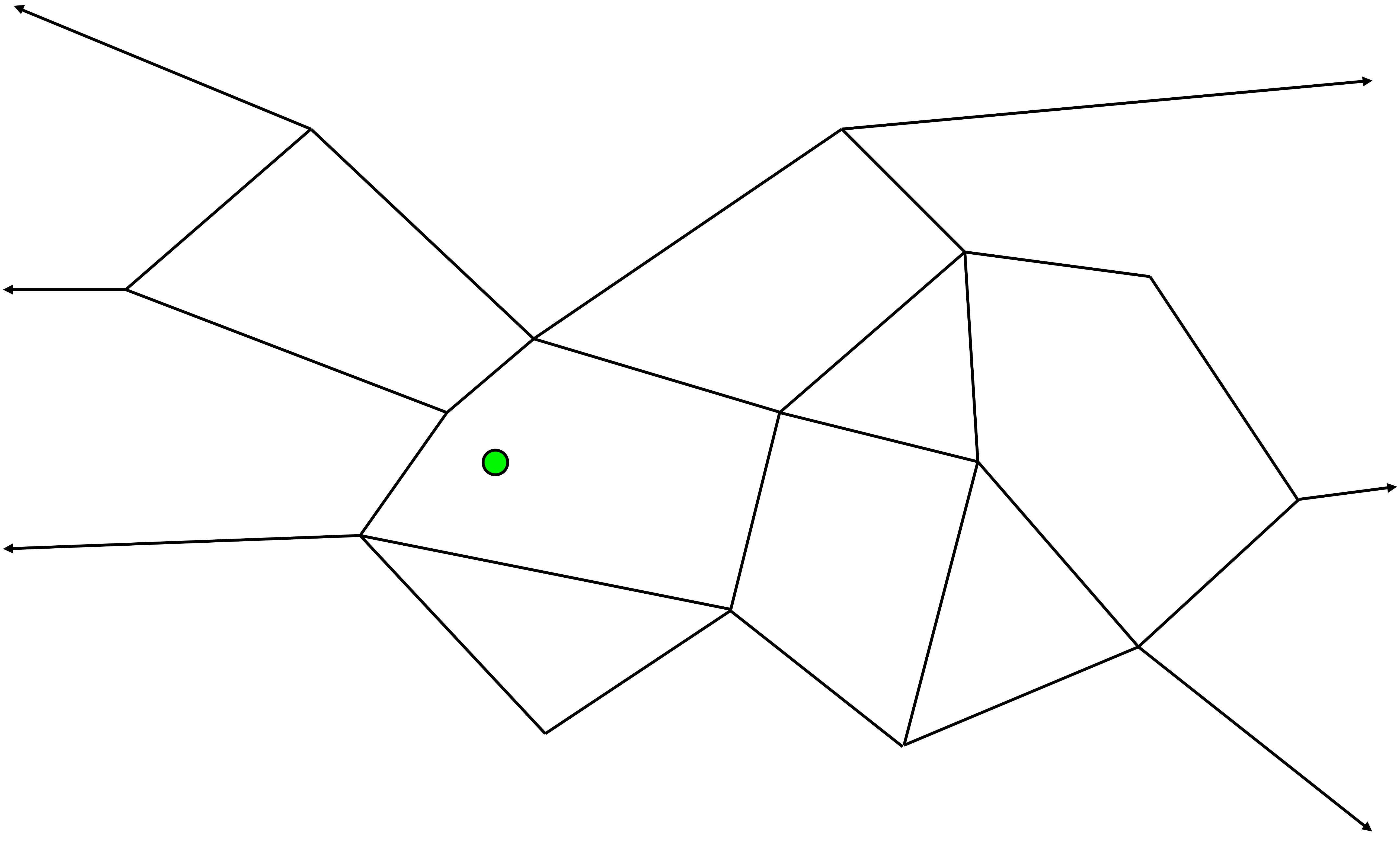
Algorithmic Applications

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.

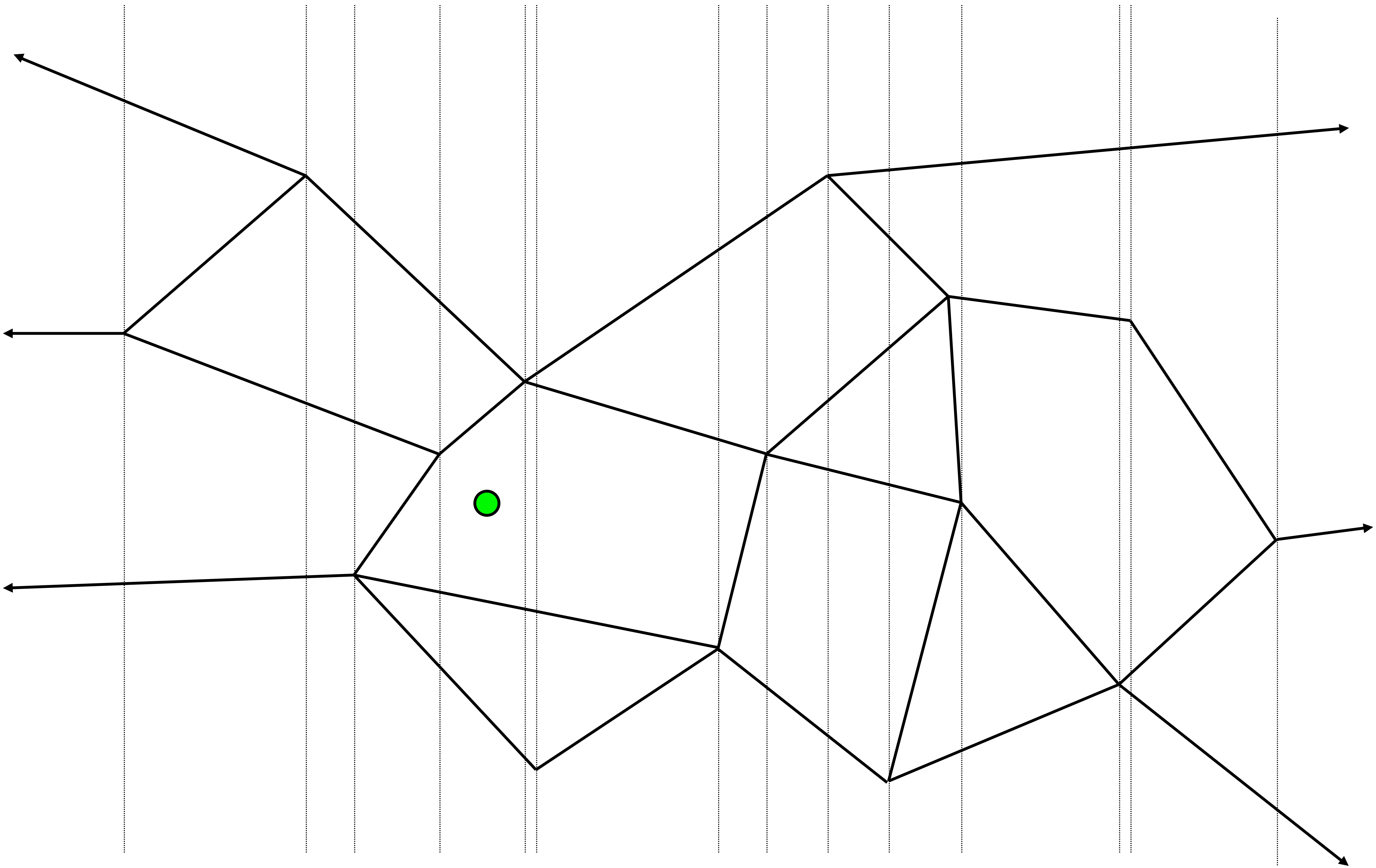
- Measure algorithm by three parameters:
    - Preprocessing time
    - Query time
    - Space

Planar point location: Example



From slides by H. Kaplan

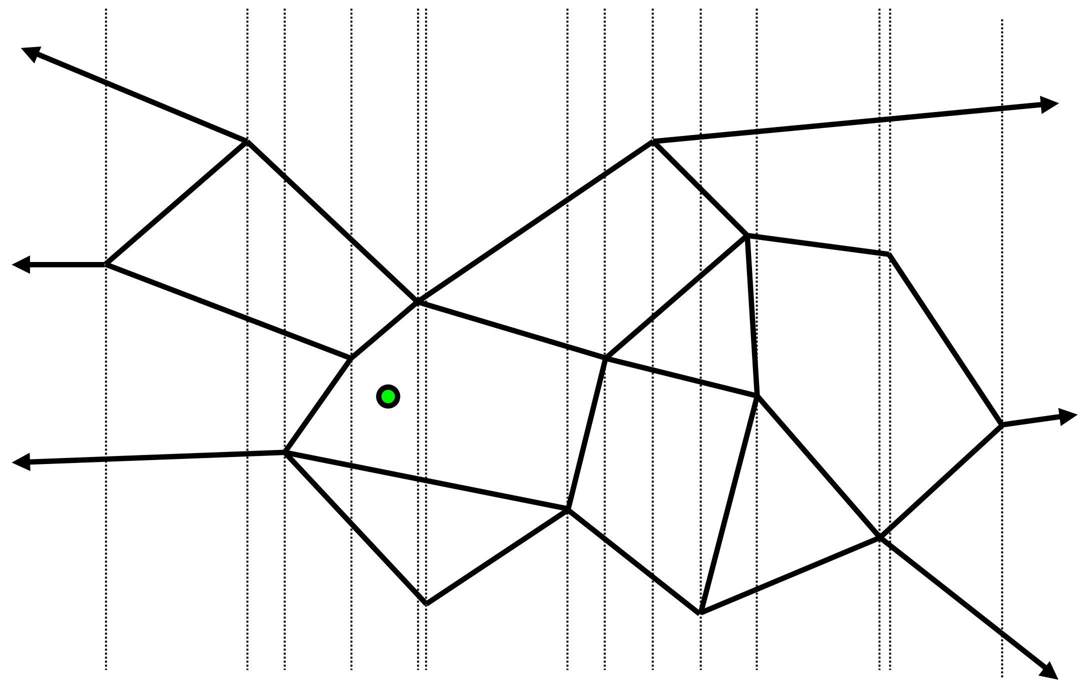
Planar point location: Example



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:

Planar Point Location

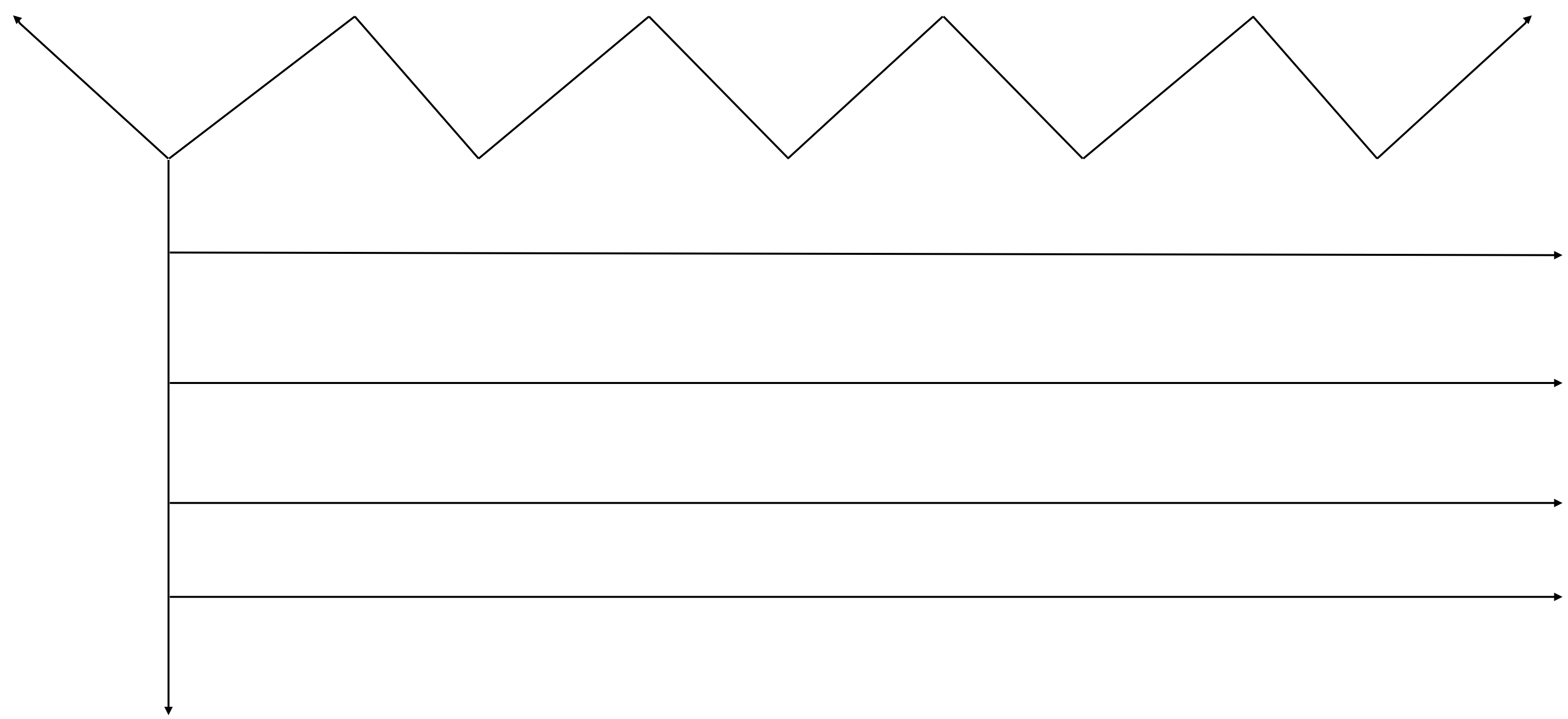
- One search tree for each slab:
    - Query time:
        - O(log n)
    - Space:

Planar Point Location

- One search tree for each slab:
  - Query time:
    - $O(\log n)$
  - Space:
    - $\Omega(n^2)$

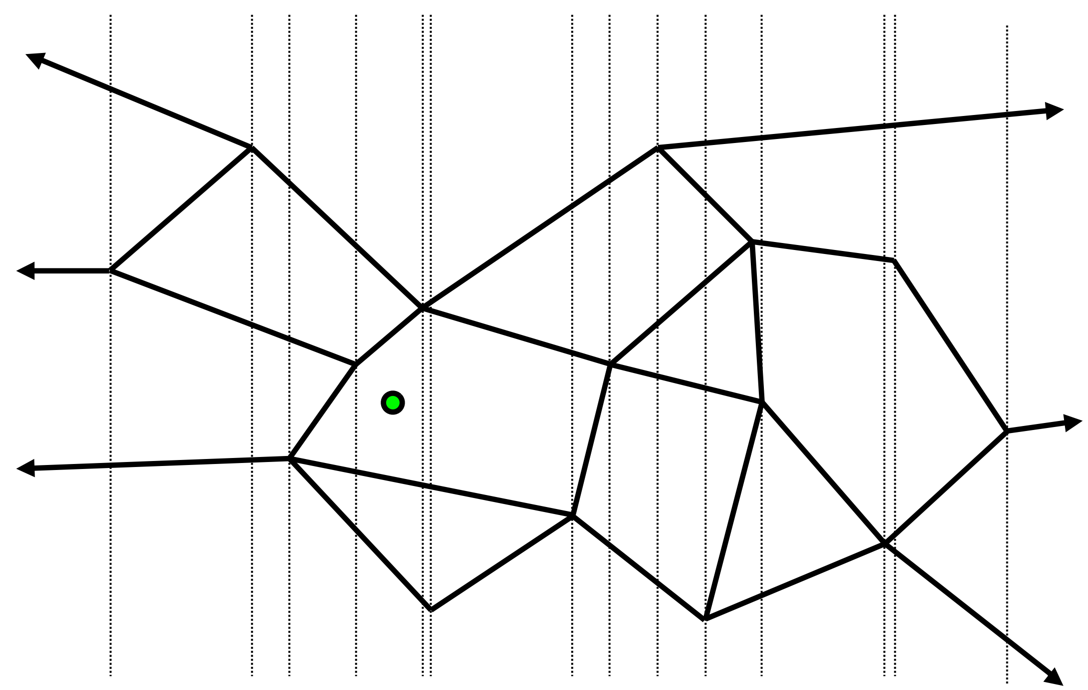
Planar Point Location

- One search tree for each slab:
  - Query time:
    - $O(\log n)$
  - Space:
    - $\Omega(n^2)$



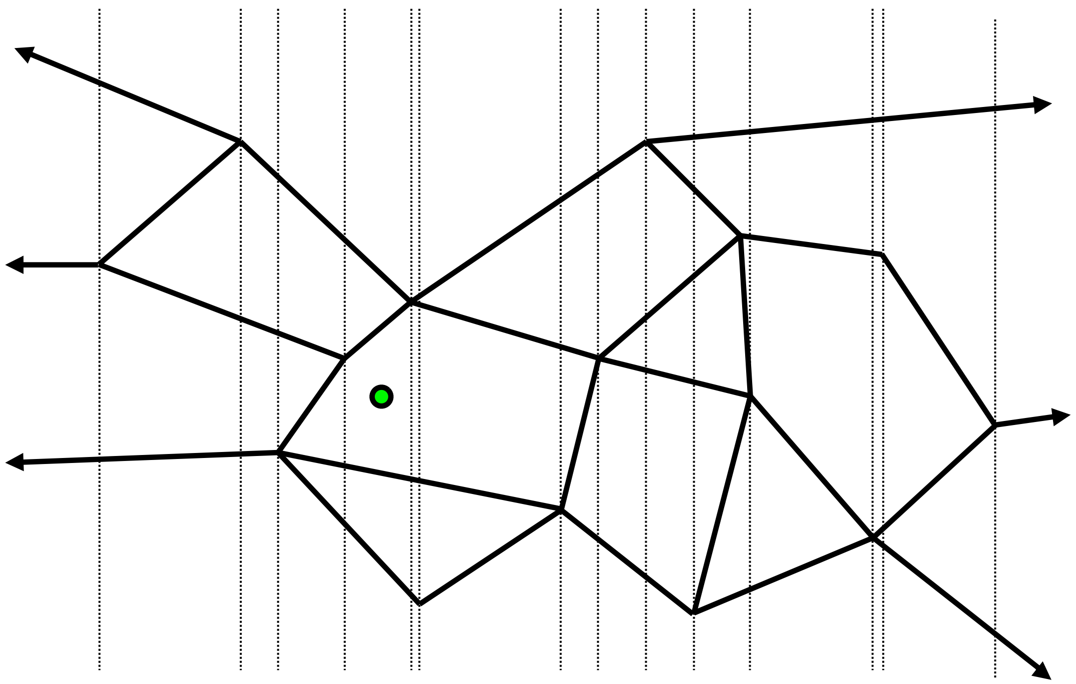
Total # lines O(n), and number of lines in each slab is O(n).

Planar point location: Improve space bound



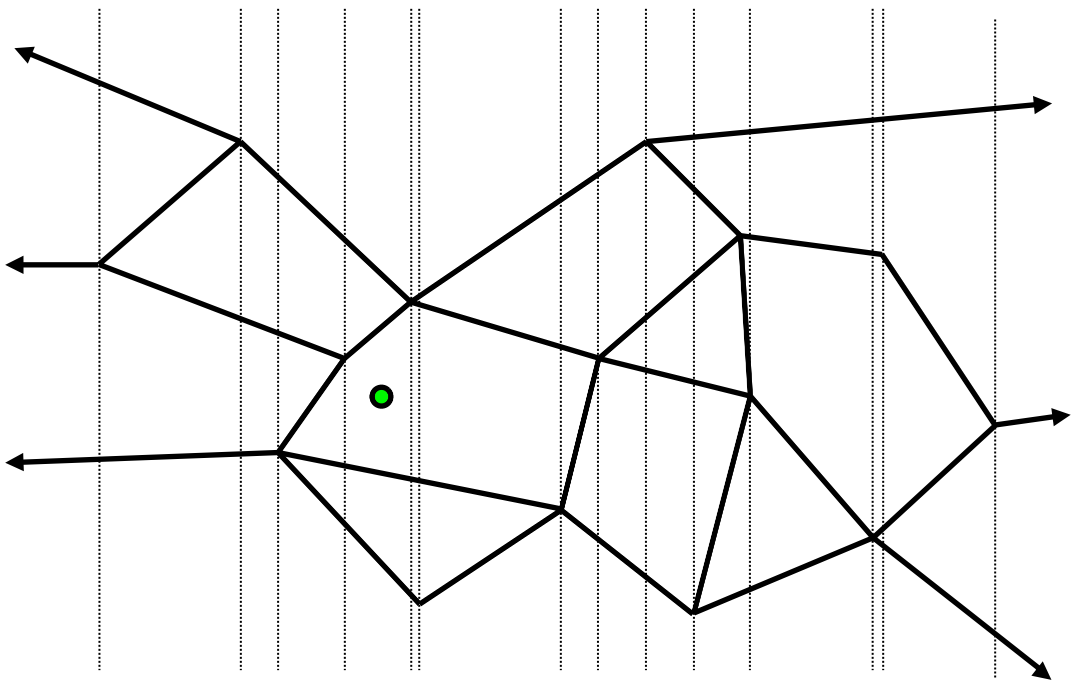
Planar point location: Improve space bound

- Key observation: The lists of the lines in adjacent slabs are very similar.



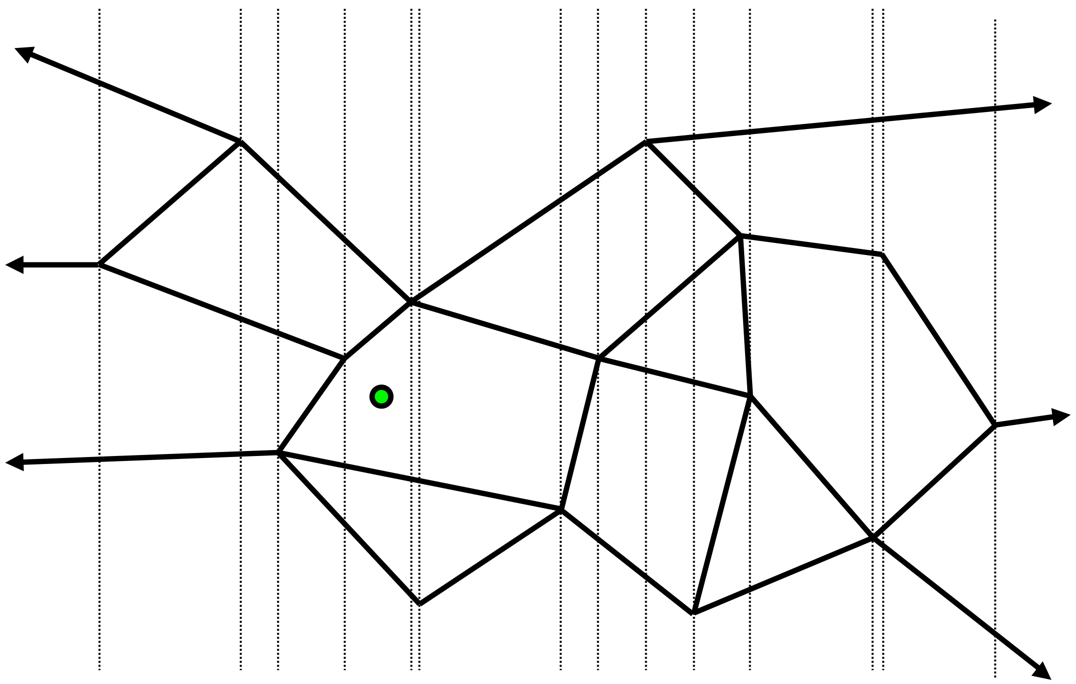
Planar point location: Improve space bound

- Key observation: The lists of the lines in adjacent slabs are very similar.
- Create the search tree for the first slab.



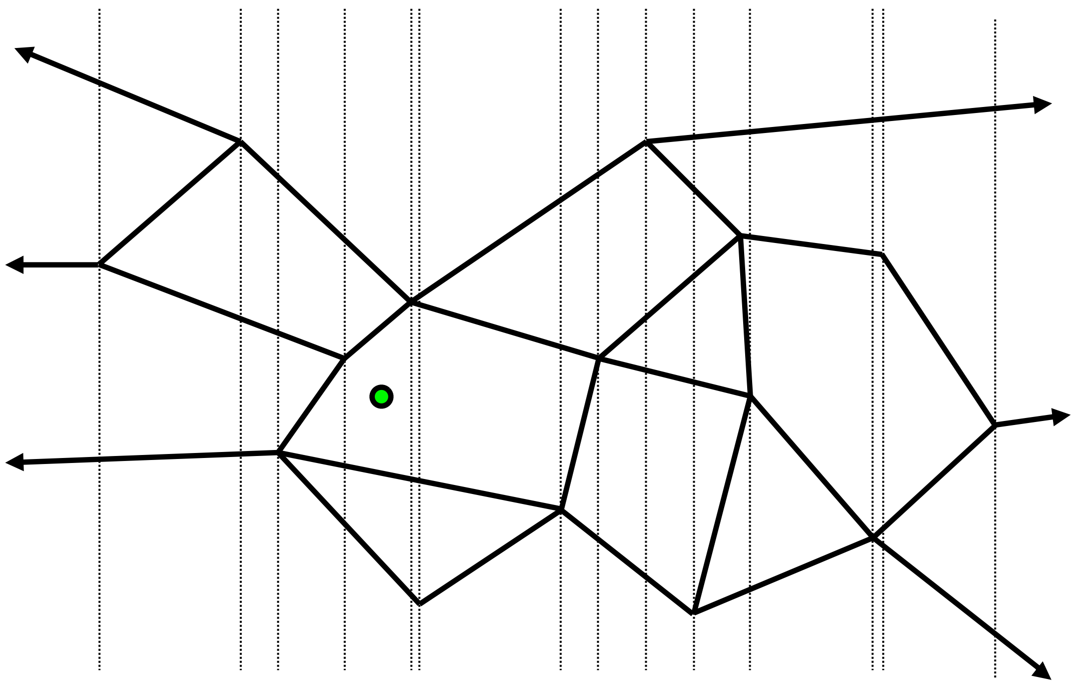
Planar point location: Improve space bound

- 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.



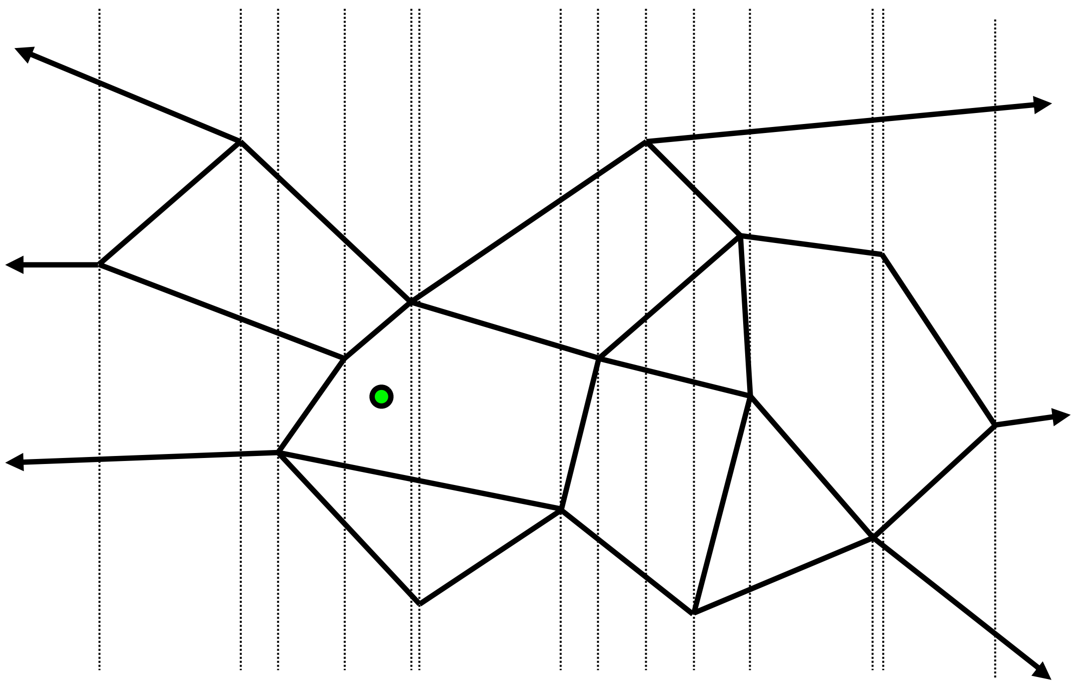
Planar point location: Improve space bound

- 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?



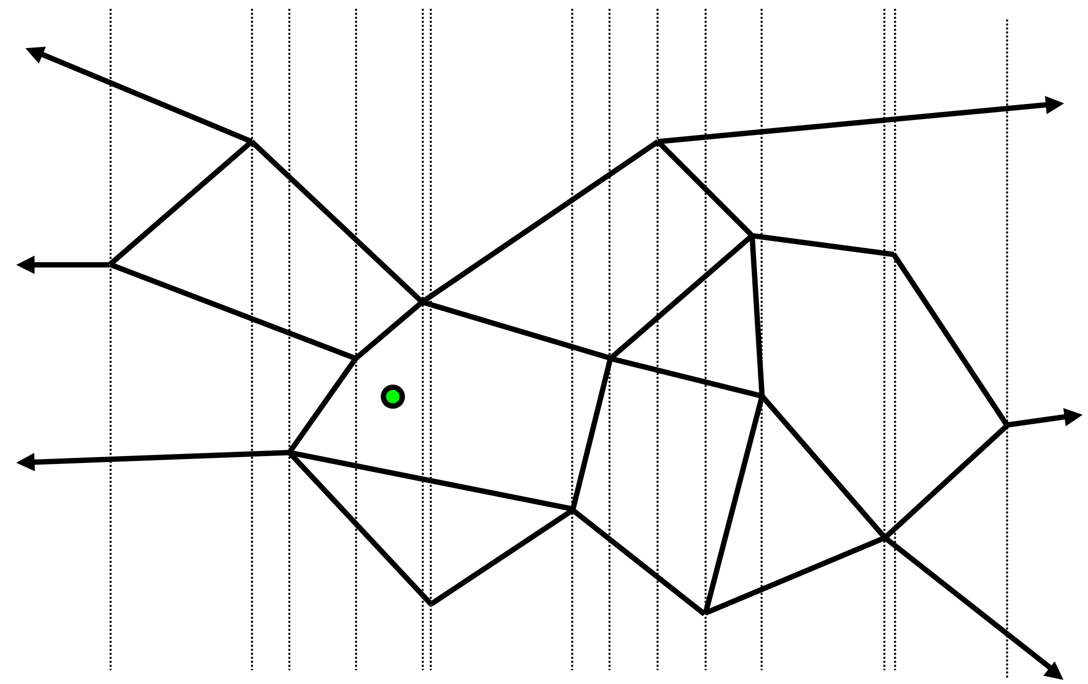
Planar point location: Improve space bound

- 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? 2n



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? 2n
- 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.