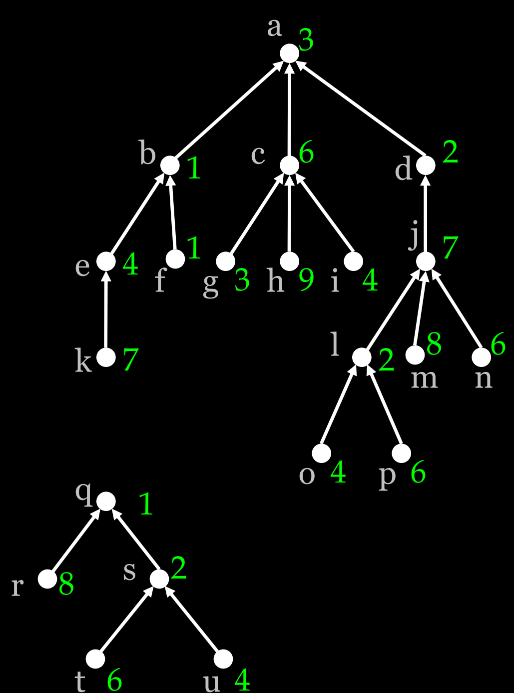
Dynamic Trees

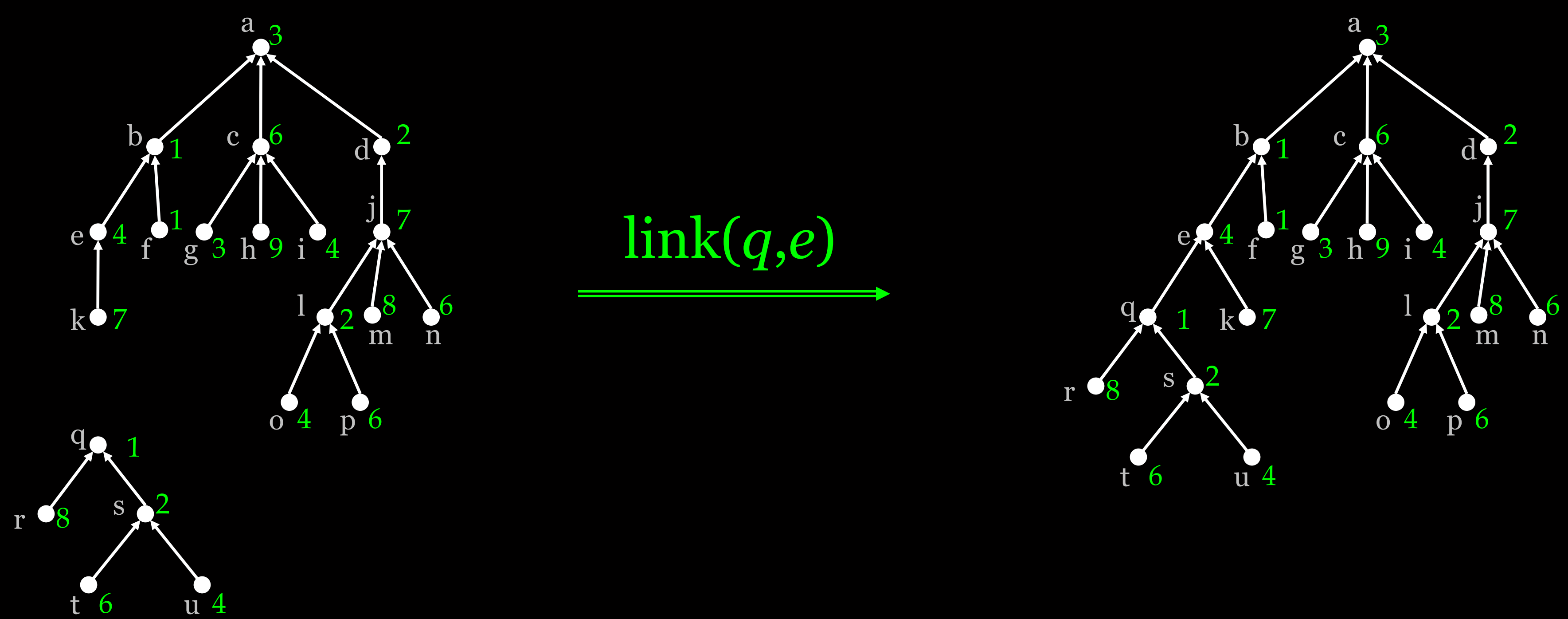
- Goal: maintain a forest of rooted trees with costs on vertices.
  - Each tree has a root, every edge directed towards the root.
- Operations allowed:
  - link(v,w): creates an edge between v (a root) and w.
  - cut(v,w): deletes edge (v,w).
  - findcost(v): returns the cost of vertex v.
  - findroot(v): returns the root of the tree containing v.
  - findmin(v): returns the vertex w of minimum cost on the path from v to the root (if there is a tie, choose the closest to the root).
  - addcost(v,x): adds x to the cost every vertex from v to root.

Dynamic Trees

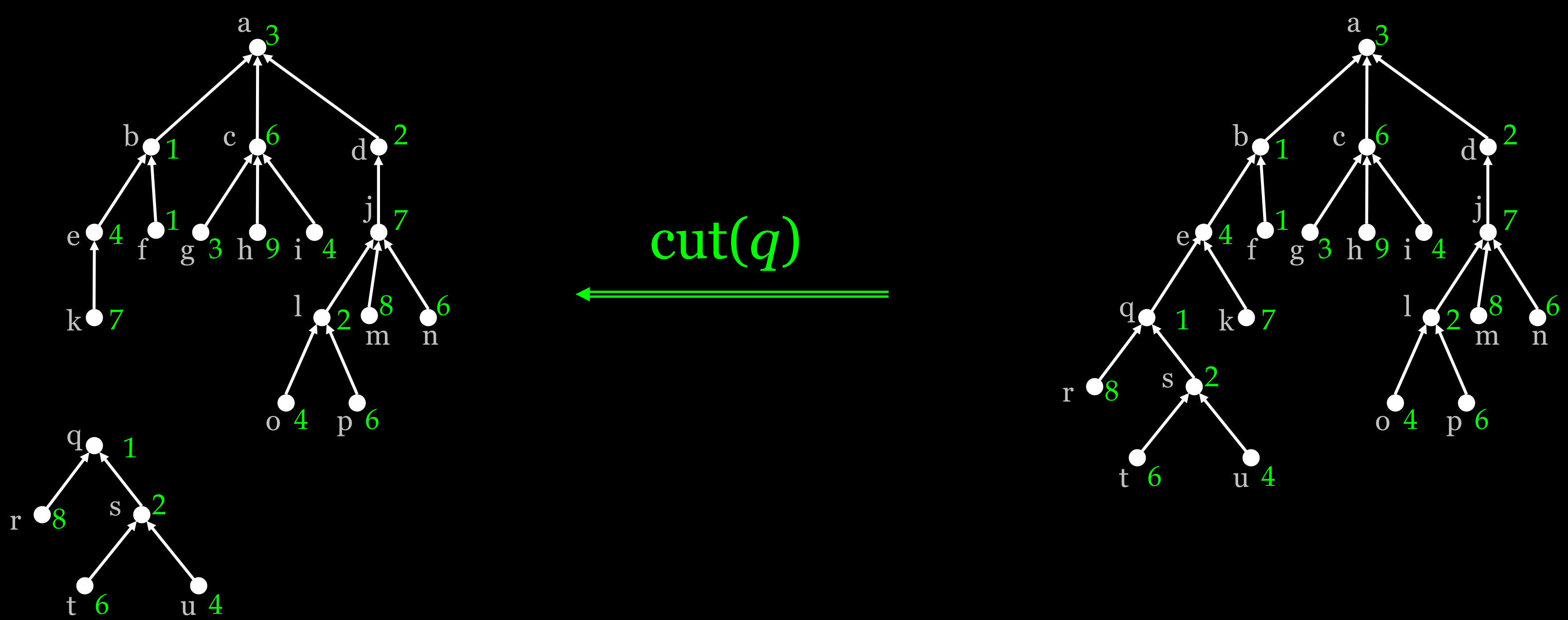
- An example (two trees):



Dynamic Trees

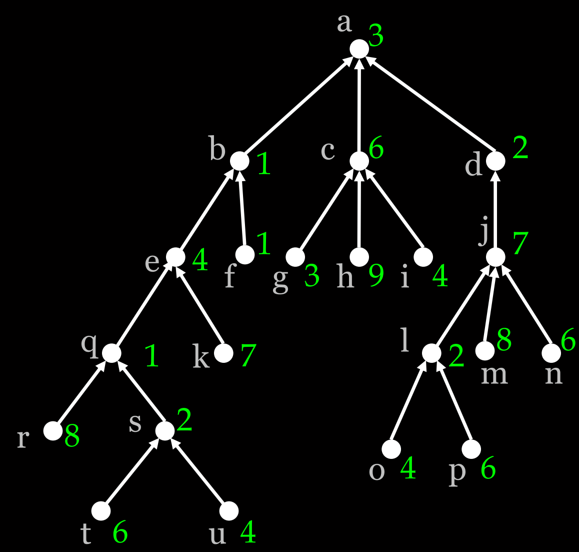


Dynamic Trees

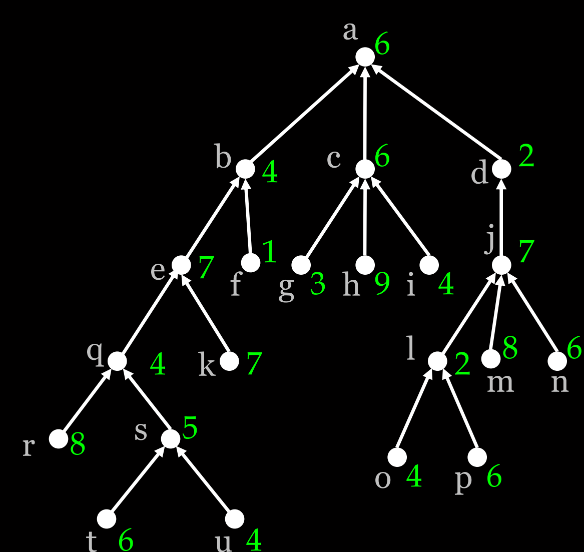


Dynamic Trees

- findmin(s) = b
- findroot(s) = a
- findcost(s) = 2



- addcost(s,3)

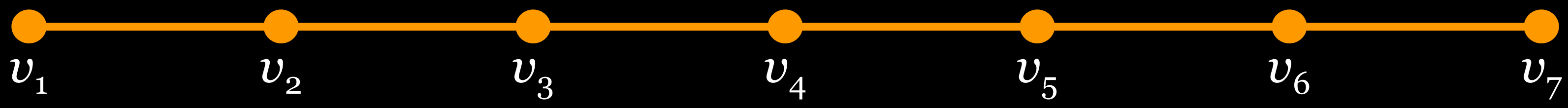


Obvious Implementation

- A node represents each vertex;
- Each node x points to its parent $p(x)$:
  - cut, split, findcost: constant time.
  - findroot, findmin, addcost: linear time on the size of the path.
- Acceptable if paths are small, but O(n) in the worst case.
- Cleverer data structures achieve O(log n) for all operations.

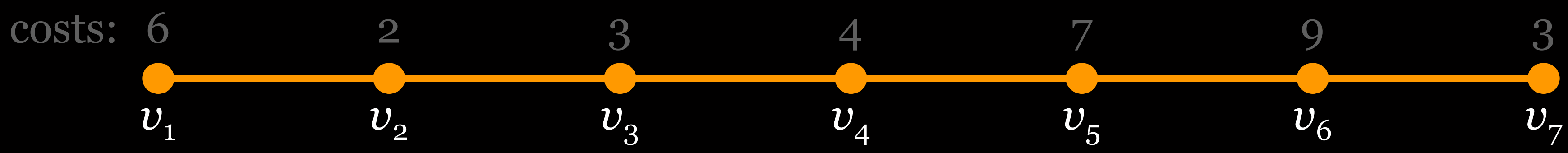
Simple Paths

- We start with a simpler problem:
  - Maintain set of paths subject to:
    - split: cuts a path in two;
    - concatenate: links endpoints of two paths, creating a new path.
  - Operations allowed:
    - findcost(v): returns the cost of vertex v;
    - addcost(v,x): adds x to the cost of vertices in path containing v;
    - findmin(v): returns minimum-cost vertex path containing v.



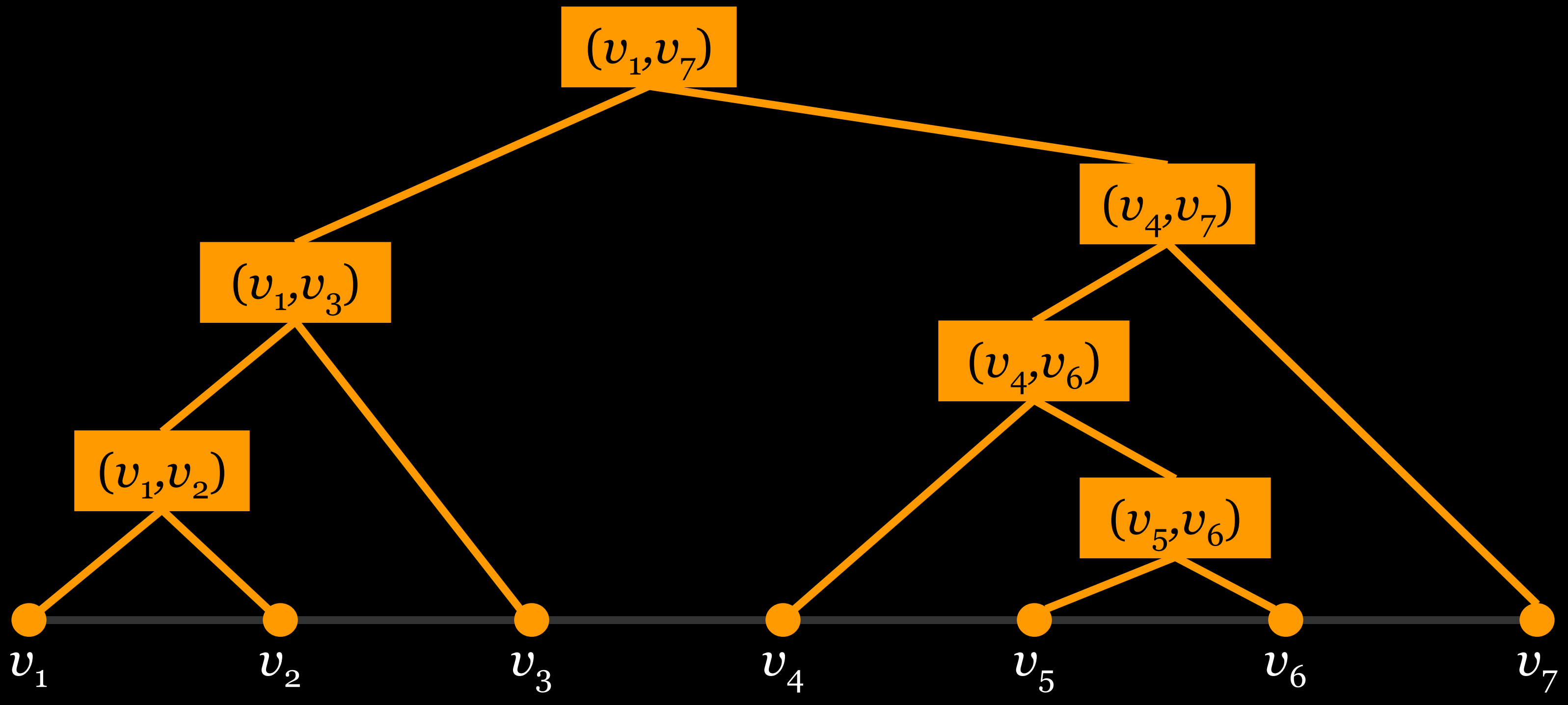
Simple Paths as Lists

- Natural representation: doubly linked list.
  - Constant time for findcost.
  - Constant time for concatenate and split if endpoints given, linear time otherwise.
  - Linear time for findmin and addcost.
- Can we do it O(log n) time?



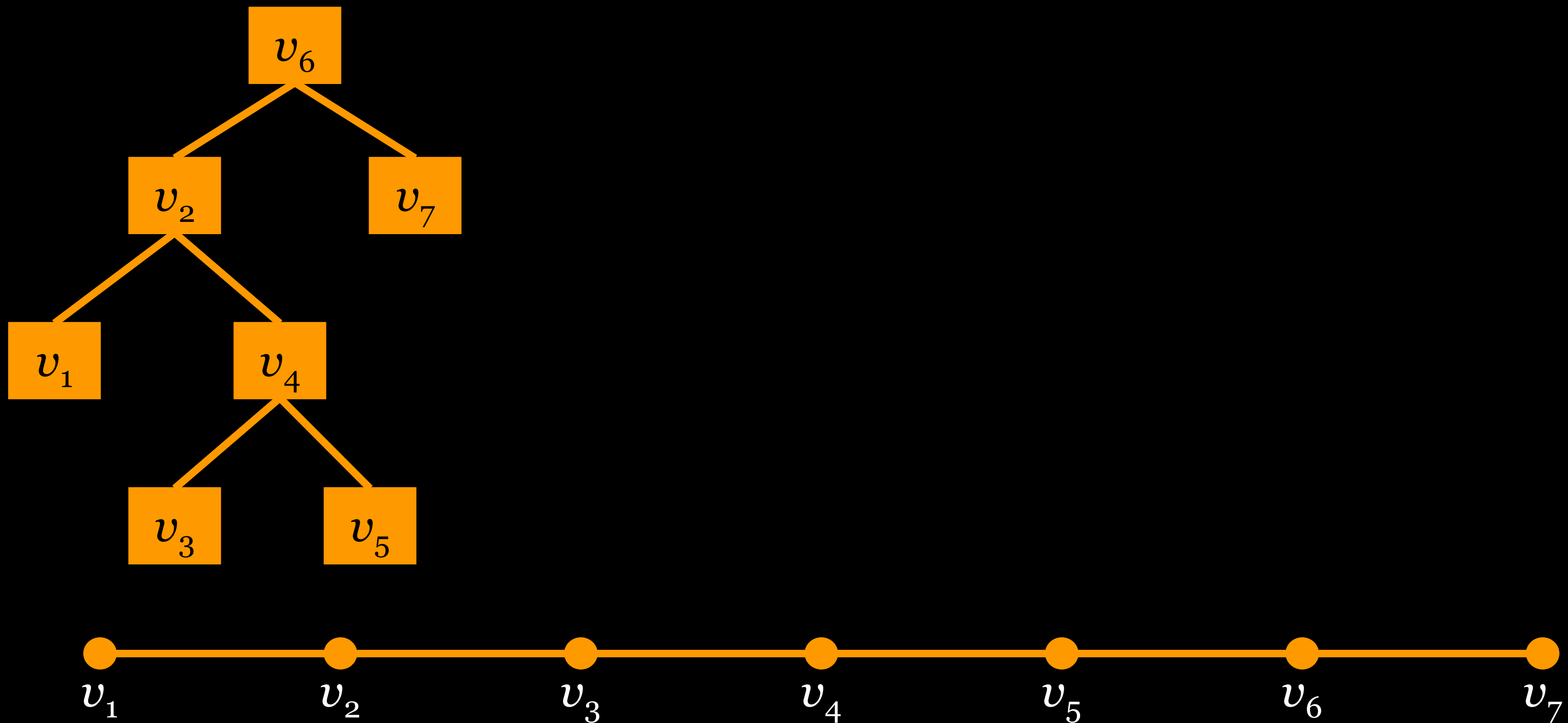
Simple Paths as Binary Trees

- Alternative representation: balanced binary trees.
  - Leaves: vertices in symmetric order.
  - Internal nodes: subpaths between extreme descendants.



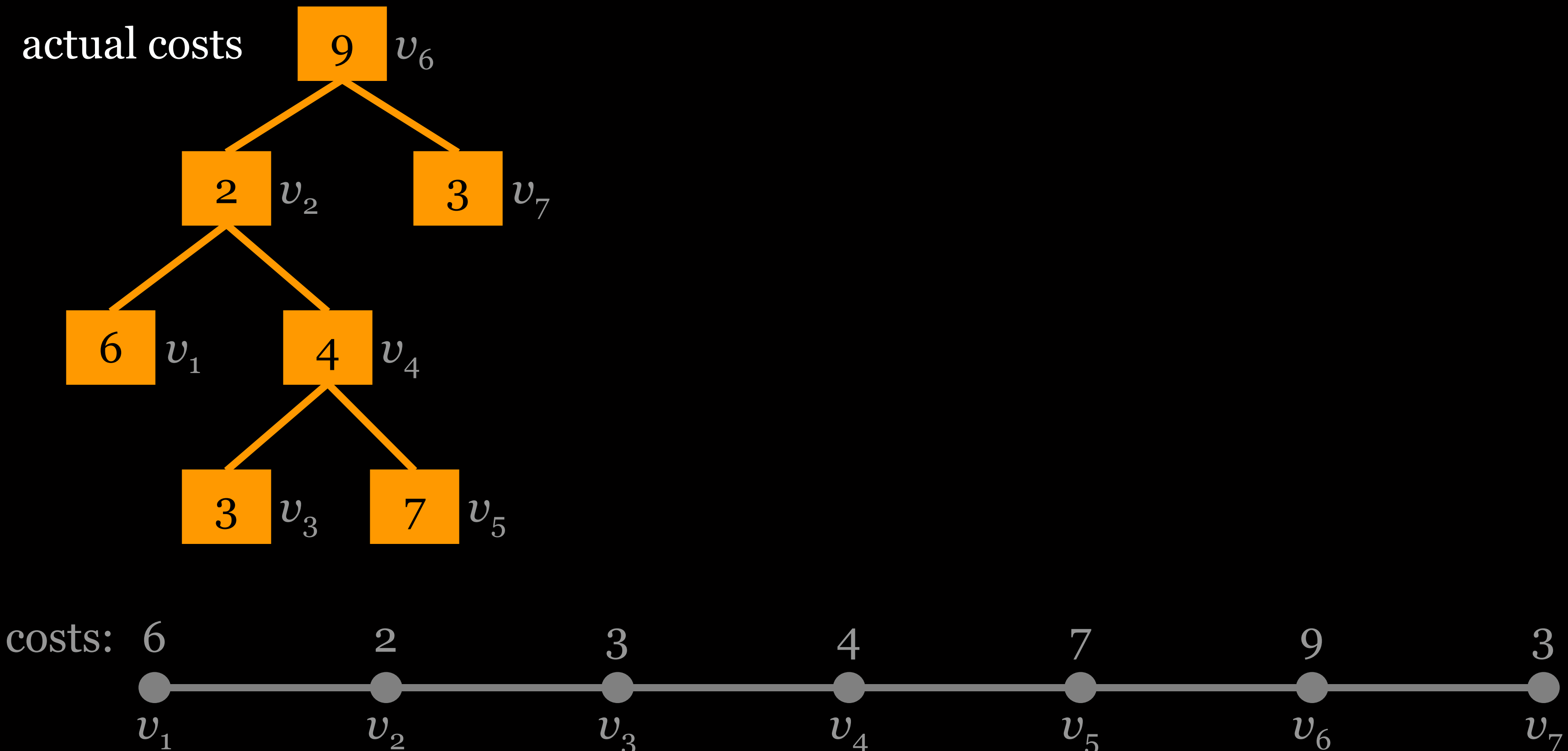
Simple Paths as Binary Trees

- Compact alternative:
  - Each internal node represents both a vertex and a subpath:
    - subpath from leftmost to rightmost descendant.



Simple Paths: Maintaining Costs

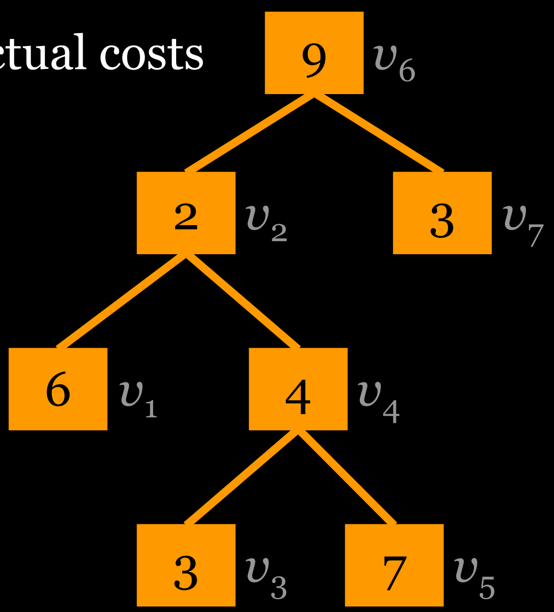
- Keeping costs:
  - First idea: store cost(x) directly on each vertex;
  - Problem: addcost takes linear time (must update all vertices).



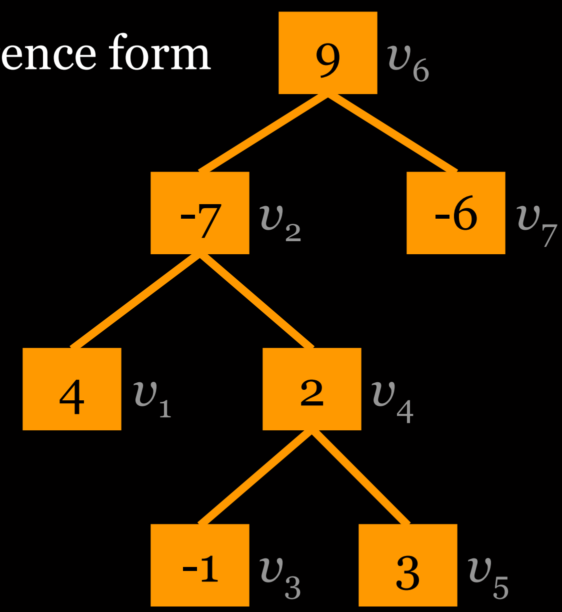
Simple Paths: Maintaining Costs

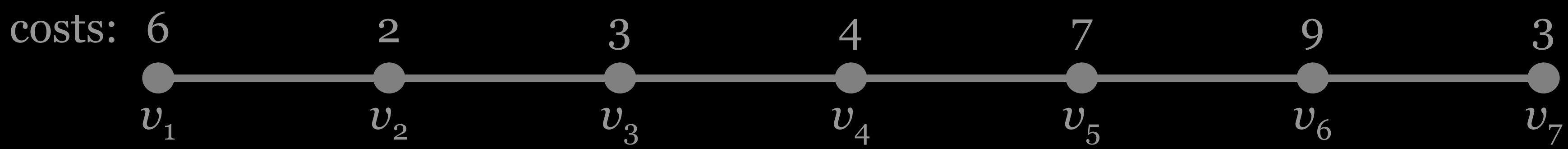
- Better approach: store $\Delta cost(x)$ instead:
  - Root: $\Delta cost(x) = cost(x)$
  - Other nodes: $\Delta cost(x) = cost(x) - cost(p(x))$

actual costs



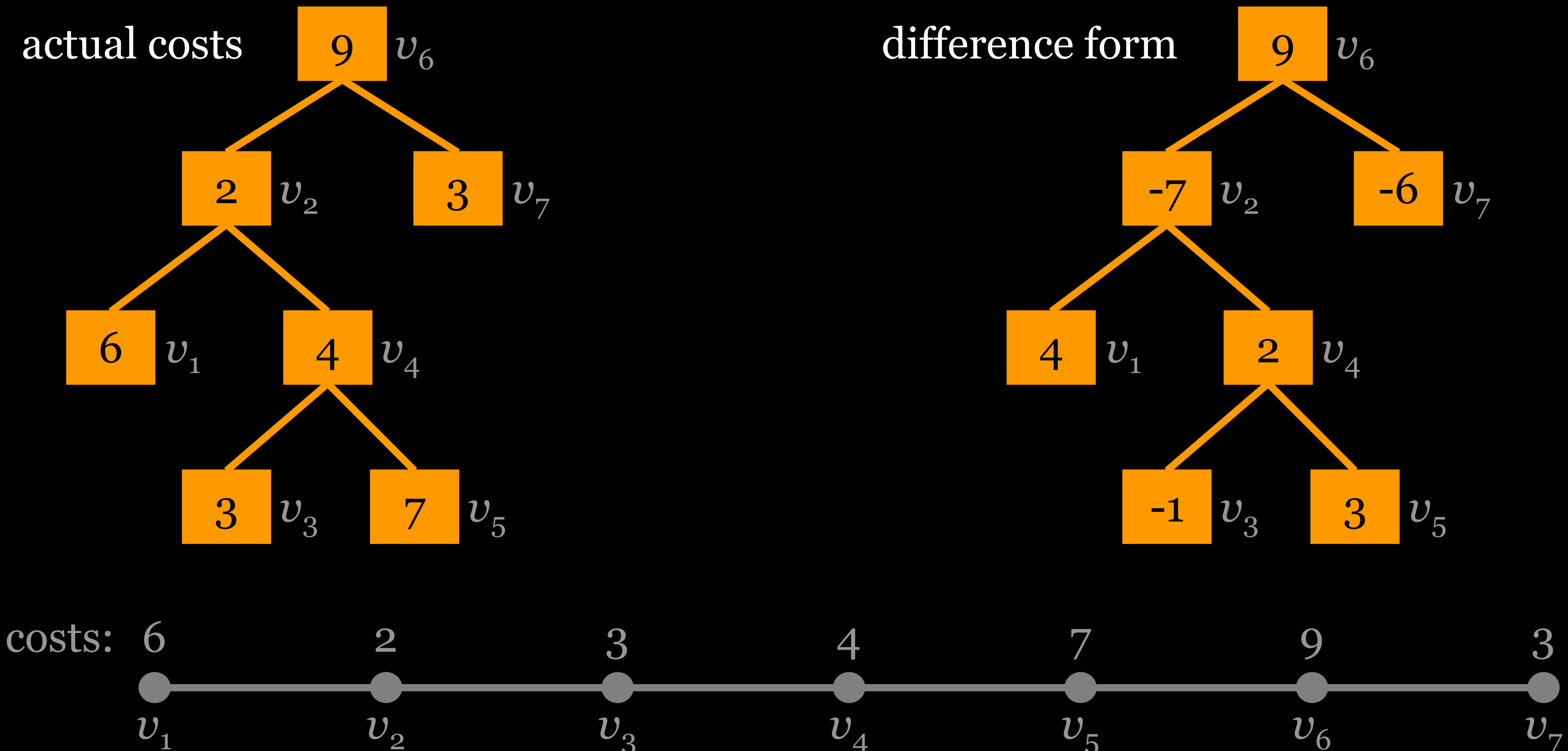
difference form





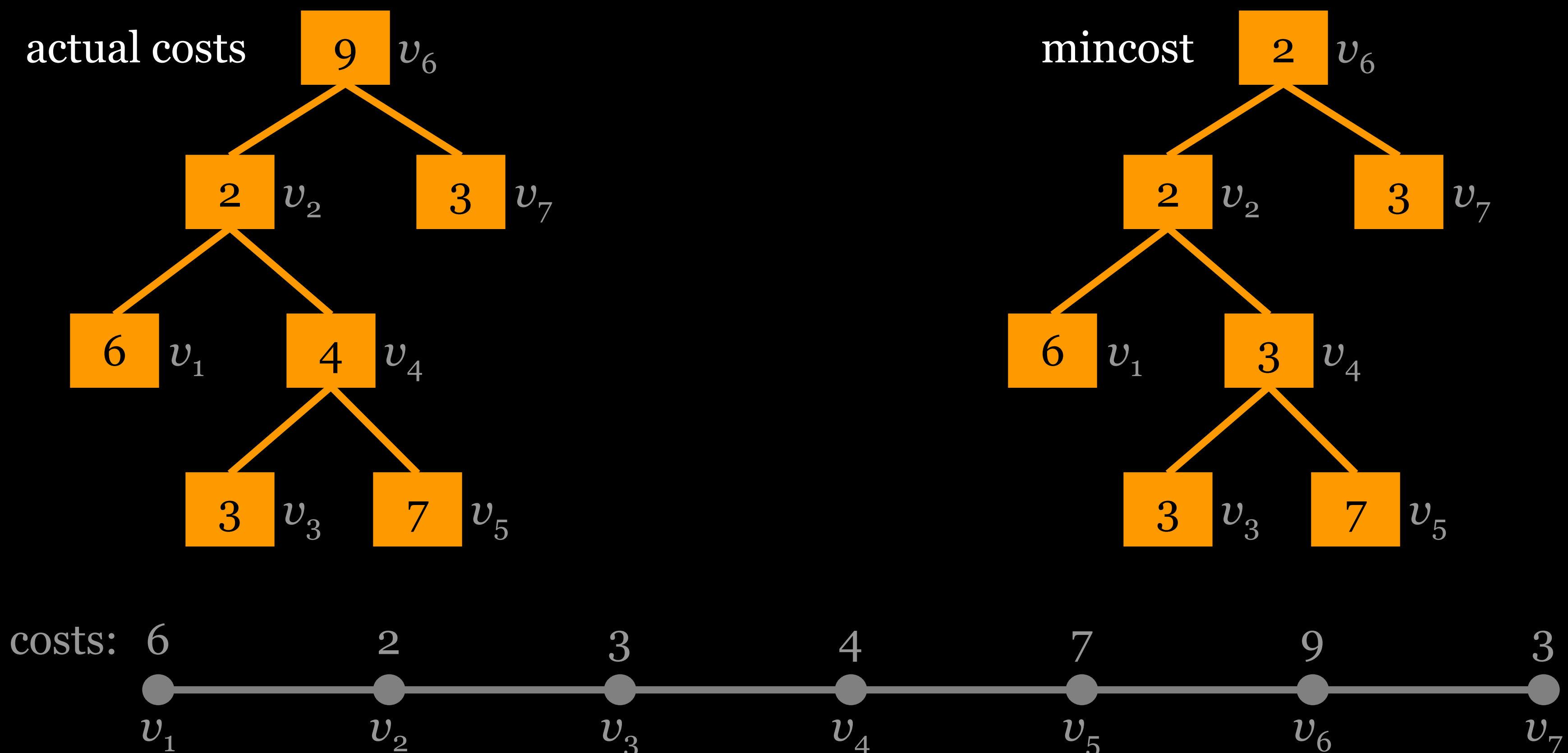
Simple Paths: Maintaining Costs

- Costs:
  - addcost: constant time (just add to root)
  - Finding cost(x) is slightly harder: O(depth(x)).



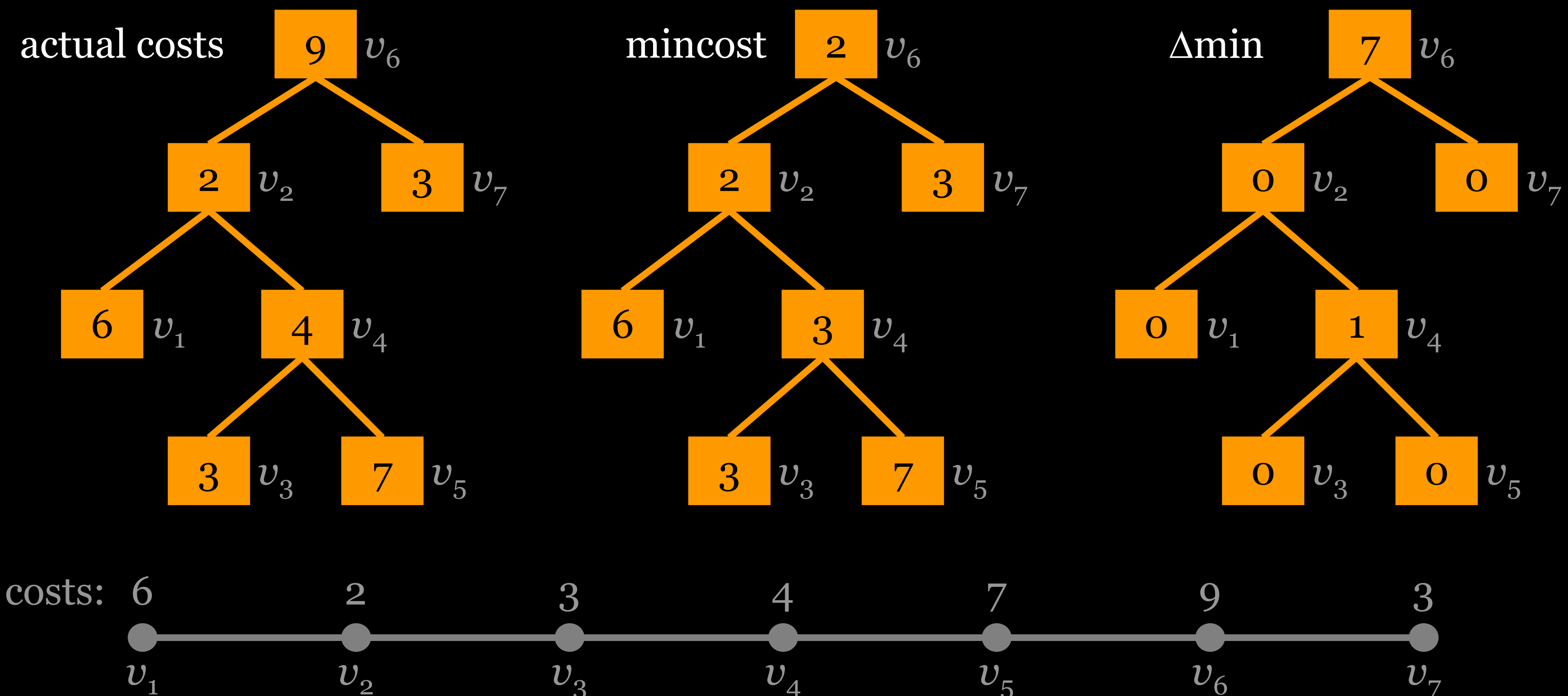
Simple Paths: Finding Minima

- Still have to implement findmin:
  - Store mincost(x), the minimum cost on subpath with root x.
    - findmin runs in O(log n) time, but addcost is linear.



Simple Paths: Finding Minima

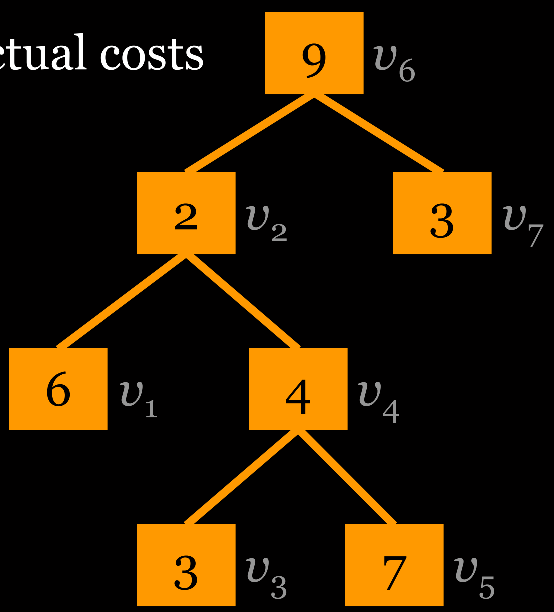
- Store $\Delta\min(x) = \text{cost}(x) - \text{mincost}(x)$ instead.
  - findmin still runs in O(log n) time, addcost now constant.



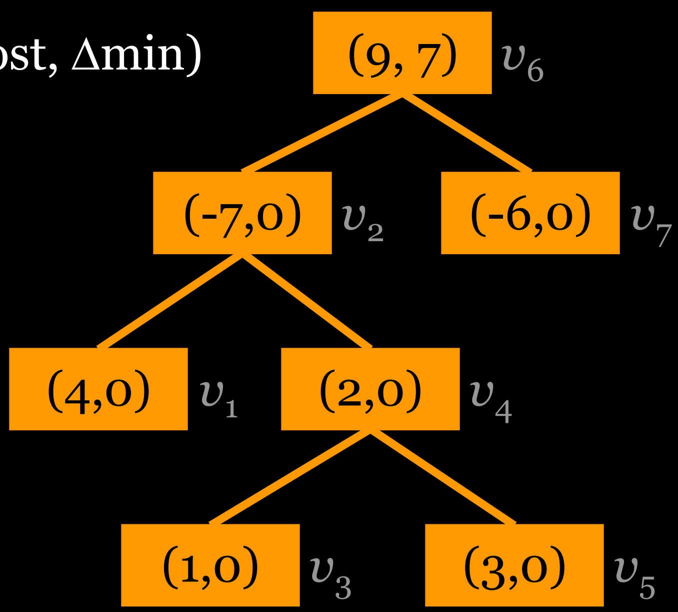
Simple Paths: Data Fields

- Final version:
  - Stores $\Delta\min(x)$ and $\Delta\text{cost}(x)$ for every vertex

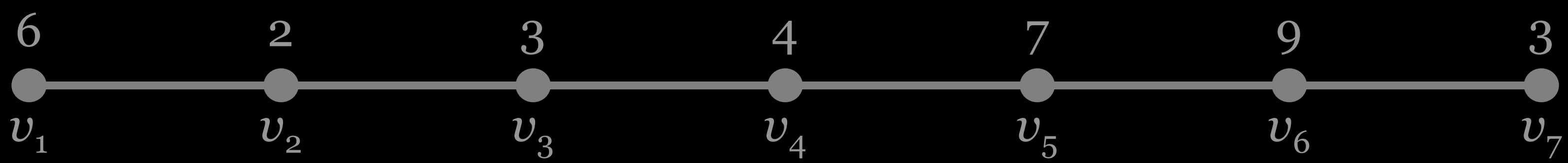
actual costs



(Δcost, Δmin)



costs:

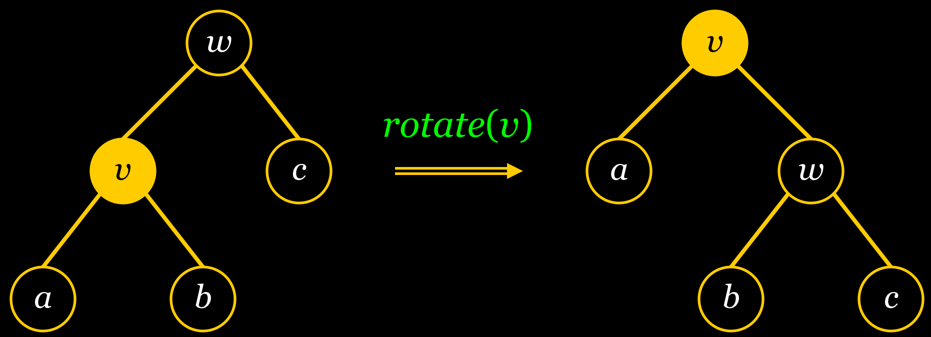


Simple Paths: Structural Changes

- Concatenating and splitting paths:
  - Join or split the corresponding binary trees;
  - Time proportional to tree height.
  - For balanced trees, this is $O(\log n)$.
    - Rotations must be supported in constant time.
    - We must be able to update Δmin and Δcost.

Simple Paths: Structural Changes

- Restructuring primitive: *rotation*.



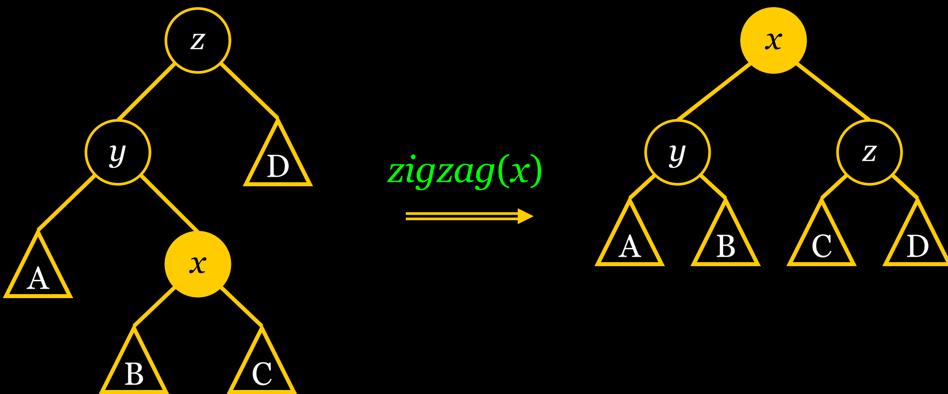
- Fields are updated as follows (for left and right rotations):
  - $\Delta cost'(v) = \Delta cost(v) + \Delta cost(w)$
  - $\Delta cost'(w) = -\Delta cost(v)$
  - $\Delta cost'(b) = \Delta cost(v) + \Delta cost(b)$
  - $\Delta min'(w) = \max\{0, \Delta min(b) - \Delta cost'(b), \Delta min(c) - \Delta cost(c)\}$
  - $\Delta min'(v) = \max\{0, \Delta min(a) - \Delta cost(a), \Delta min'(w) - \Delta cost'(w)\}$

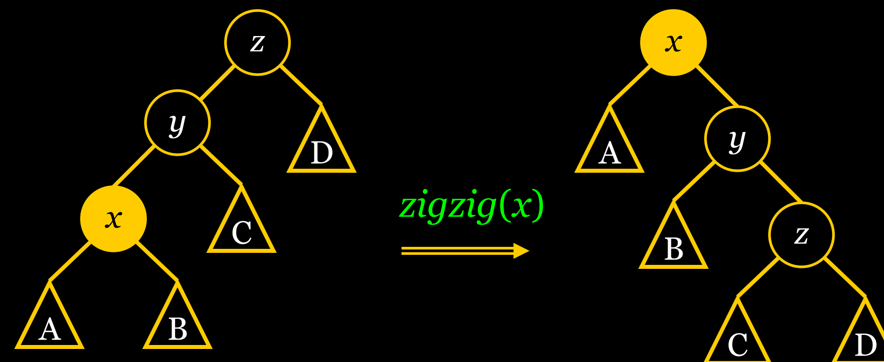
Splaying

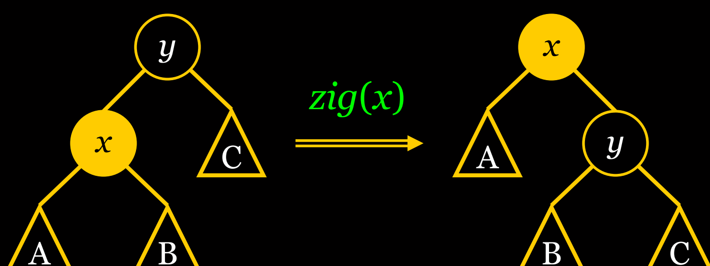
- Simpler alternative to balanced binary trees: splaying.
  - Does not guarantee that trees are balanced in the worst case.
  - Guarantees $O(\log n)$ access in the amortized sense.
  - Makes the data structure much simpler to implement.
- Basic characteristics:
  - Does not require any balancing information;
  - On an access to v, splay on v:
    - Moves v to the root;
    - Roughly halves the depth of other nodes in the access path.
  - Based entirely on rotations.
- Other operations (insert, delete, join, split) use splay.

Splaying

- Three restructuring operations:

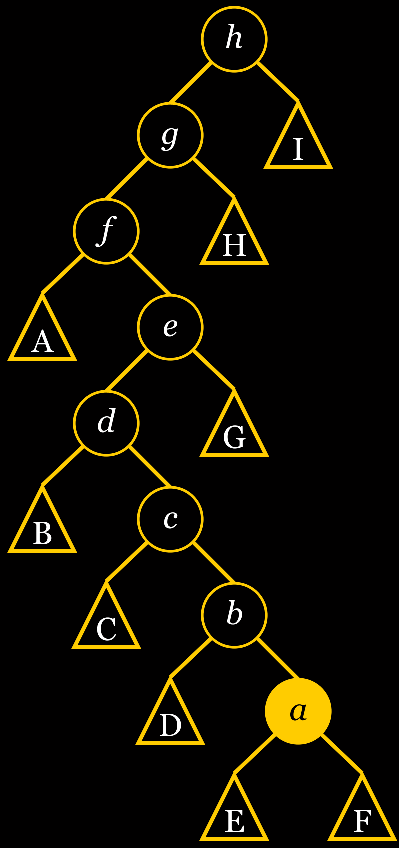




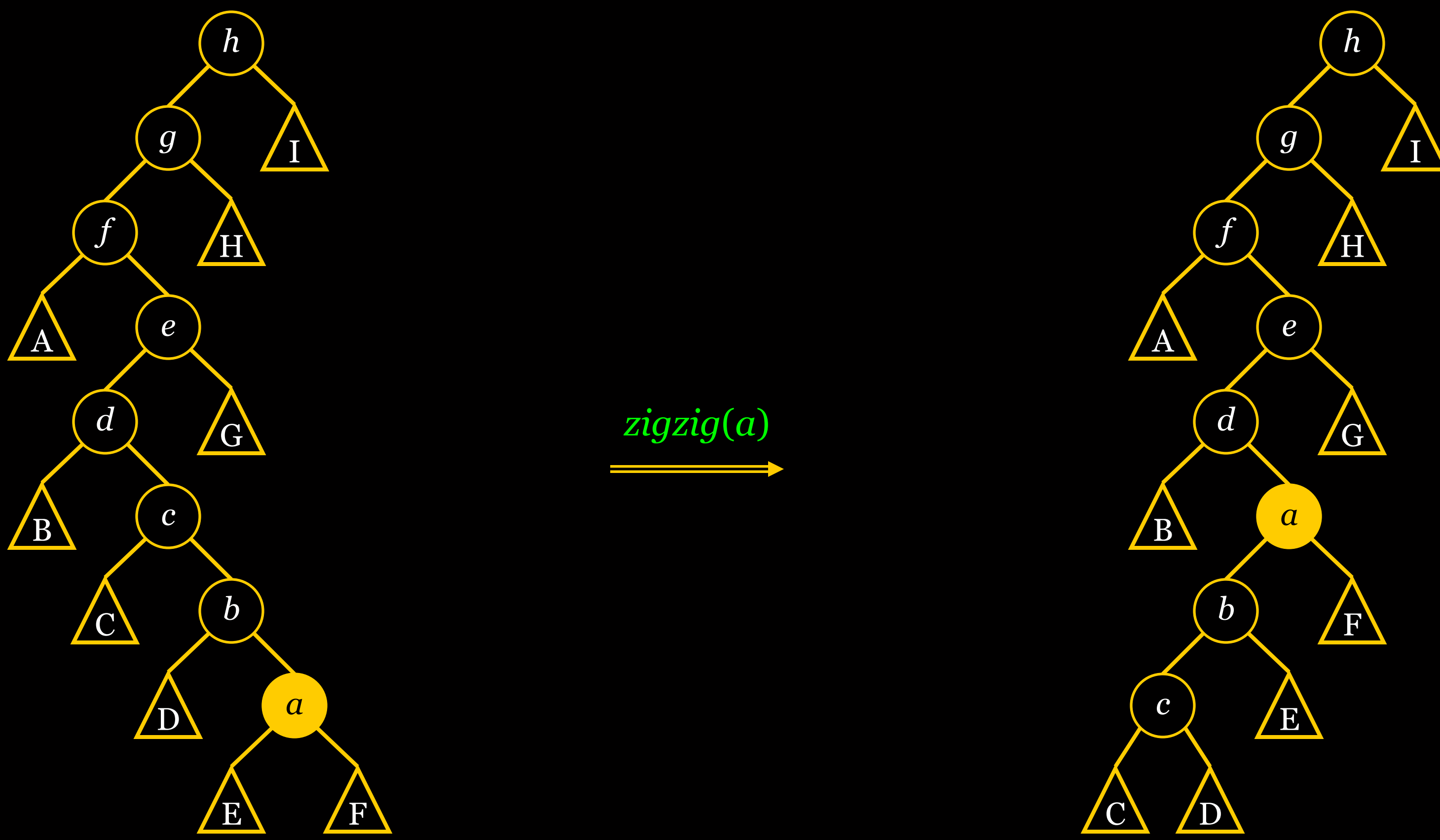


(only happens if y is root)

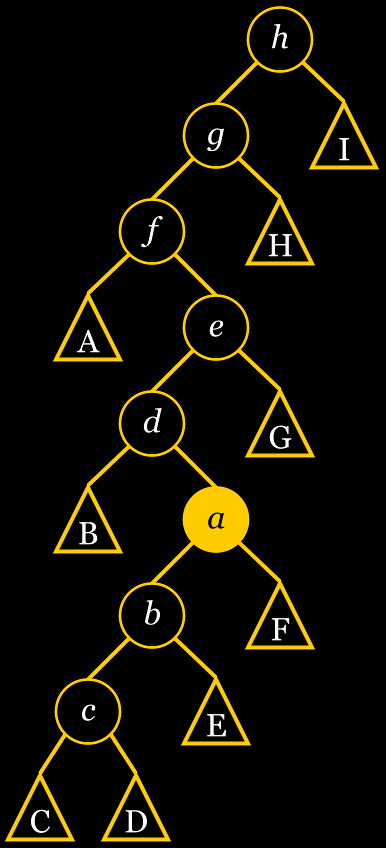
An Example of Splaying



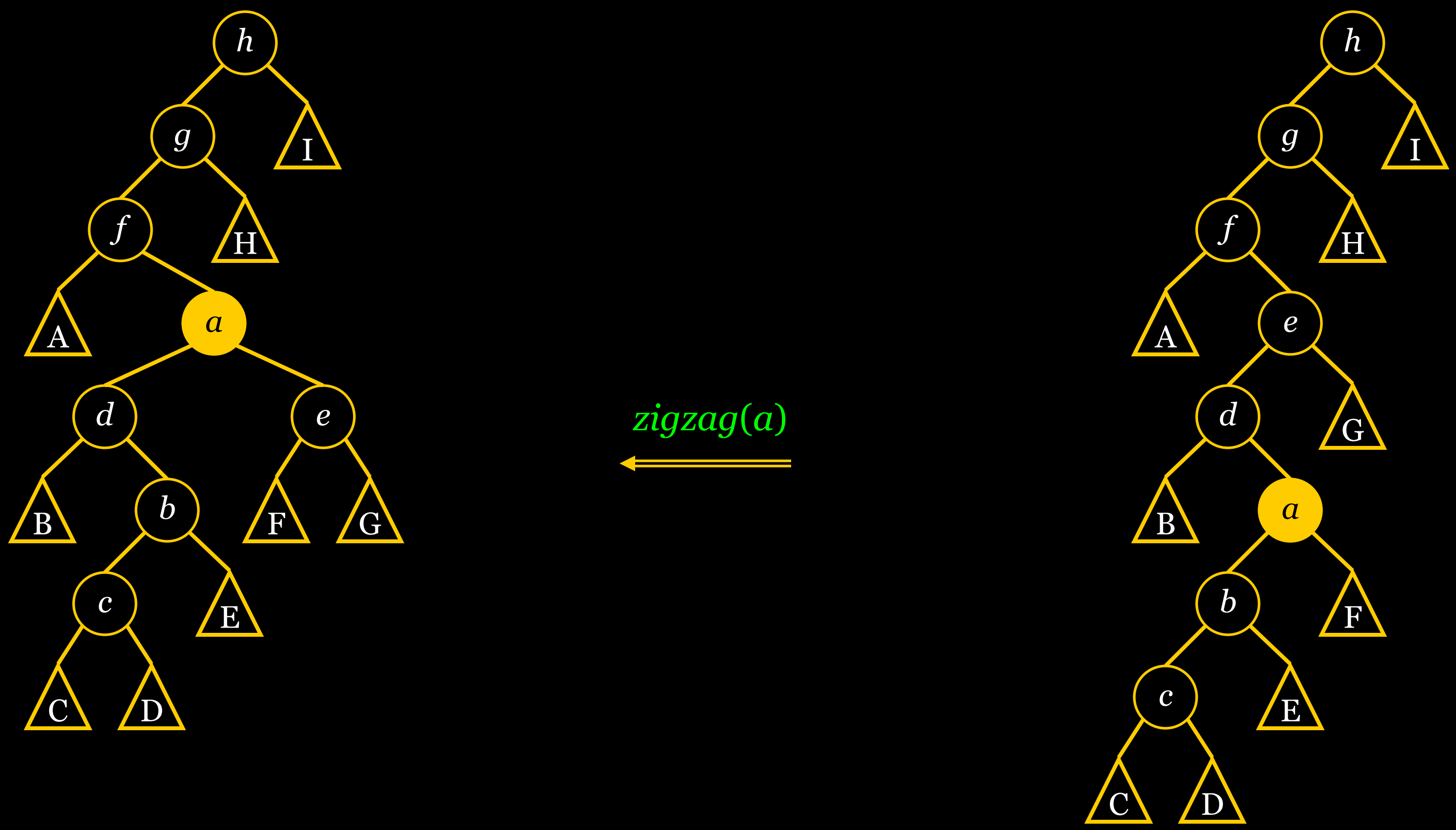
An Example of Splaying



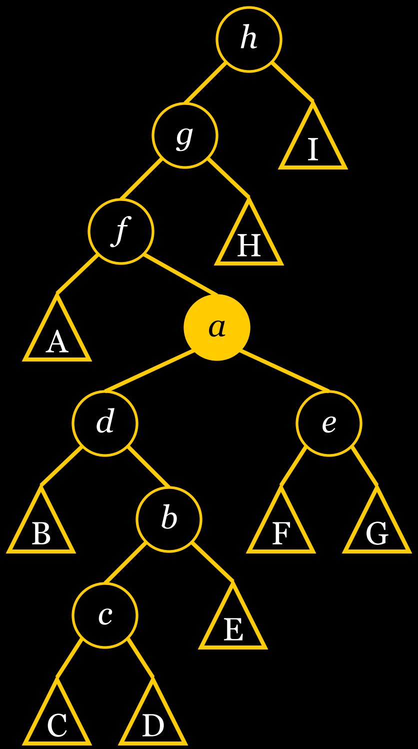
An Example of Splaying



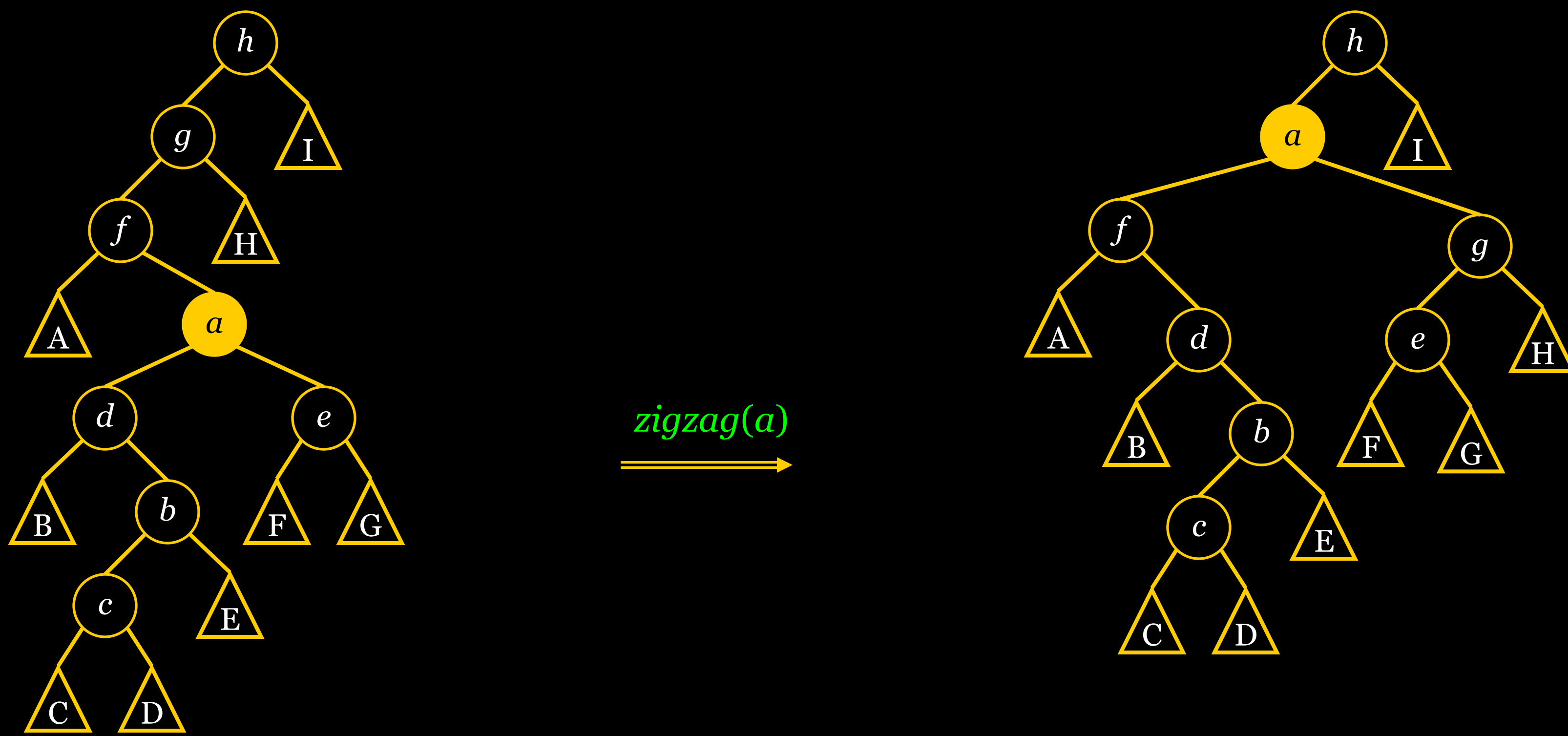
An Example of Splaying



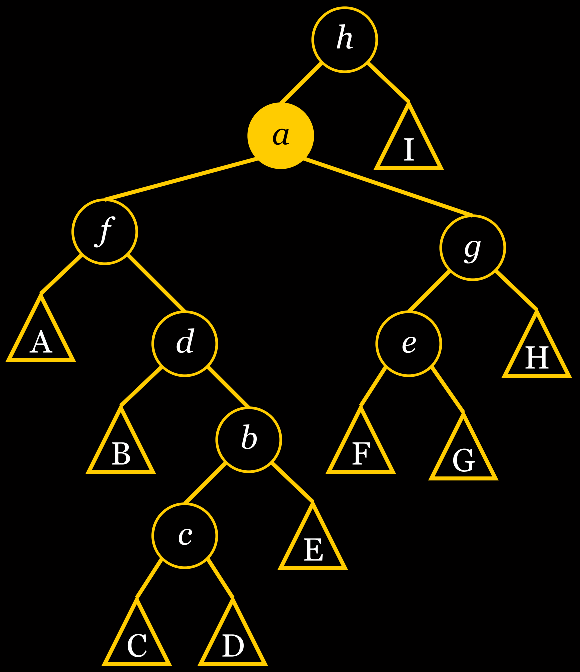
An Example of Splaying



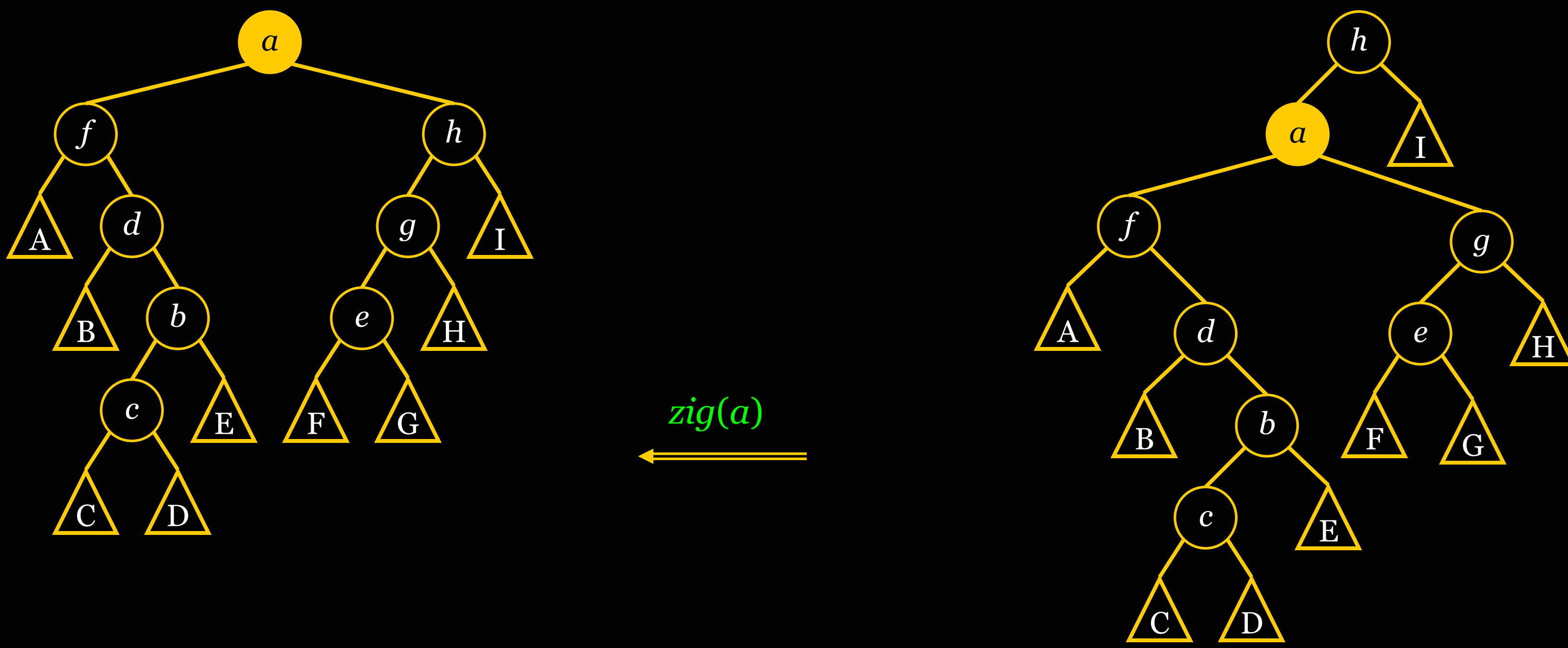
An Example of Splaying



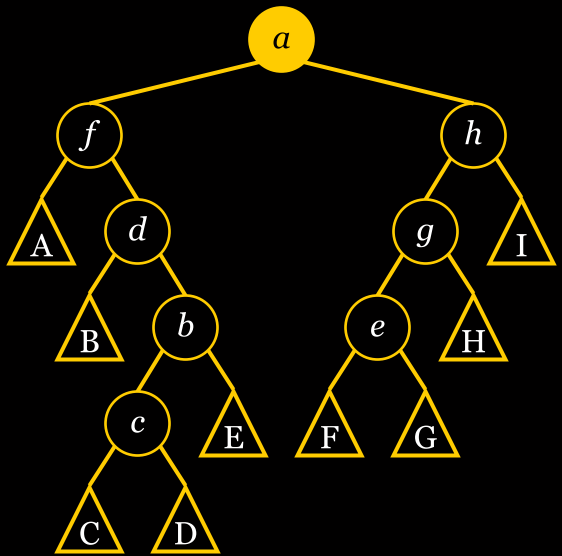
An Example of Splaying



An Example of Splaying

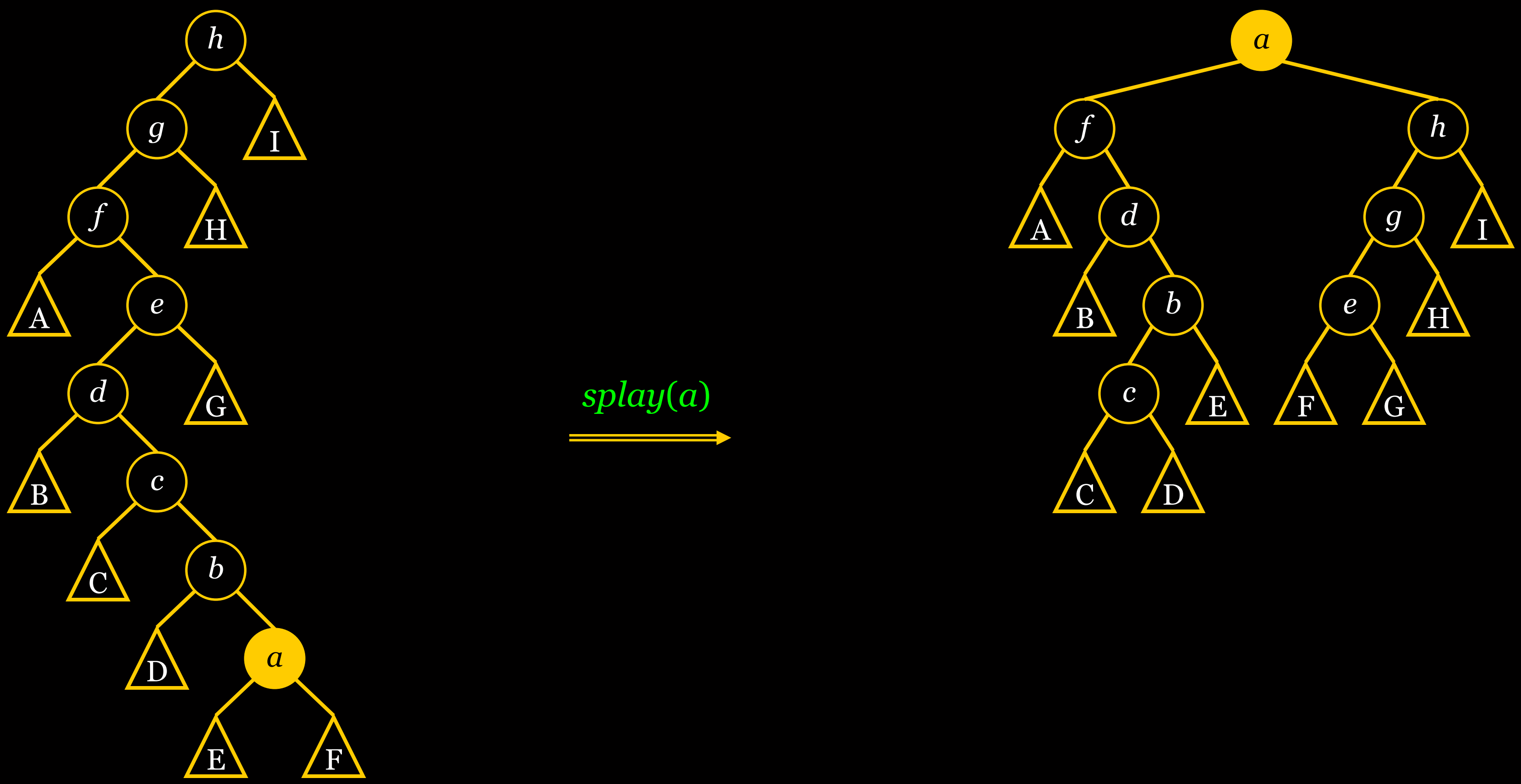


An Example of Splaying



An Example of Splaying

- End result:



Amortized Analysis

- Bounds the running time of a sequence of operations.
- Potential function Φ maps each configuration to real number.
- Amortized time to execute each operation:
  - $a_i = t_i + \Phi_i - \Phi_{i-1}$
    - a_i: amortized time to execute i-th operation;
    - t_i: actual time to execute the operation;
    - Φ_i: potential after the i-th operation.
- Total time for m operations:

$$\Sigma_{i=1..m}\, t_i = \Sigma_{i=1..m}(a_i + \Phi_{i-1} - \Phi_i) = \Phi_0 - \Phi_m + \Sigma_{i=1..m}\, a_i$$

Amortized Analysis of Splaying

- Definitions:
  - $s(x)$: size of node x (number of descendants, including x);
    - At most n, by definition.
  - $r(x)$: rank of node x, defined as $\log s(x)$;
    - At most $\log n$, by definition.
  - Φ_i: potential of the data structure (twice the sum of all ranks).
    - At most $O(n \log n)$, by definition.
- Access Lemma [ST85]: *The amortized time to splay a tree with root t at a node x is at most*

$$6(r(t)-r(x)) + 1 = O(\log(s(t)/s(x))).$$

Proof of Access Lemma

- Access Lemma [ST85]: *The amortized time to splay a tree with root t at a node x is at most*

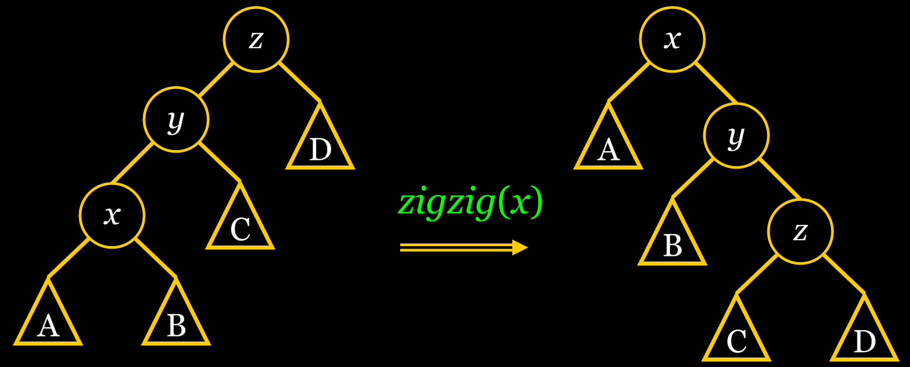
$$6(r(t)-r(x)) + 1 = O(\log(s(t)/s(x))).$$

- Proof idea:
  - $r_i(x)$ = rank of x after the i-th splay step;
  - a_i = amortized cost of the i-th splay step;
  - $a_i \leq 6(r_i(x)-r_{i-1}(x)) + 1$ (for the zig step, if any)
  - $a_i \leq 6(r_i(x)-r_{i-1}(x))$ (for any zig-zig and zig-zag steps)
  - Total amortized time for all k steps:

$$\begin{aligned}\textstyle\sum_{i=1..k} a_i &\leq \textstyle\sum_{i=1..k\text{-}1} [6(r_i(x)-r_{i-1}(x))] + [6(r_k(x)-r_{k-1}(x)) + 1] \\ &= 6r_k(x) - 6r_0(x) + 1\end{aligned}$$

Proof of Access Lemma: Splaying Step

- Zig-zig:



Claim: $a \leq 6\,(r'(x) - r(x))$

$t + \Phi' - \Phi \leq 6\,(r'(x) - r(x))$

$2 + 2(r'(x)+r'(y)+r'(z)) - 2(r(x)+r(y)+r(z)) \leq 6\,(r'(x) - r(x))$

$1 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \leq 3\,(r'(x) - r(x))$

$1 + r'(y) + r'(z) - r(x) - r(y) \leq 3\,(r'(x) - r(x))$ since $r'(x) = r(z)$

$1 + r'(y) + r'(z) - 2r(x) \leq 3\,(r'(x) - r(x))$ since $r(y) \geq r(x)$

$1 + r'(x) + r'(z) - 2r(x) \leq 3\,(r'(x) - r(x))$ since $r'(x) \geq r'(y)$

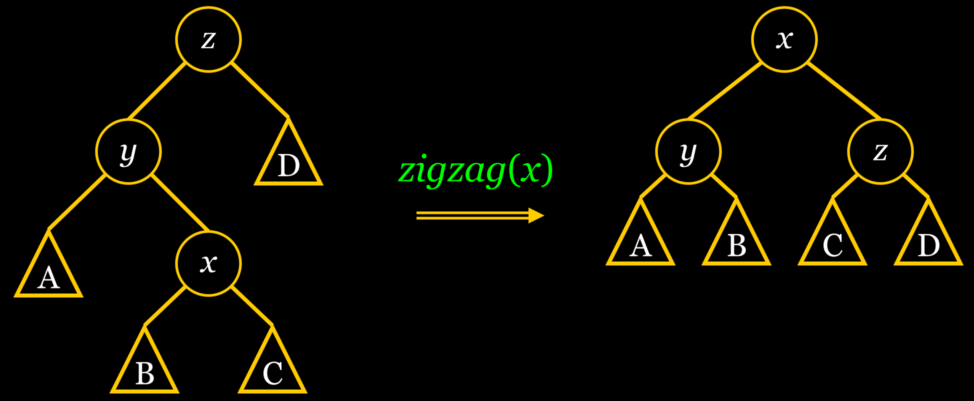
$(r(x) - r'(x)) + (r'(z) - r'(x)) \leq -1$ rearranging

$\log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \leq -1$ definition of rank

TRUE because $s(x)+s'(z)<s'(x)$: both ratios are smaller than 1, at least one is at most 1/2.

Proof of Access Lemma: Splaying Step

- Zig-zag:



Claim: $a \leq 4\,(r'(x) - r(x))$

$t + \Phi' - \Phi \leq 4\,(r'(x) - r(x))$

$2 + (2r'(x)+2r'(y)+2r'(z)) - (2r(x)+2r(y)+2r(z)) \leq 4\,(r'(x) - r(x))$

$2 + 2r'(y) + 2r'(z) - 2r(x) - 2r(y) \leq 4\,(r'(x) - r(x))$, since $r'(x) = r(z)$

$2 + 2r'(y) + 2r'(z) - 4r(x) \leq 4\,(r'(x) - r(x))$, since $r(y) \geq r(x)$

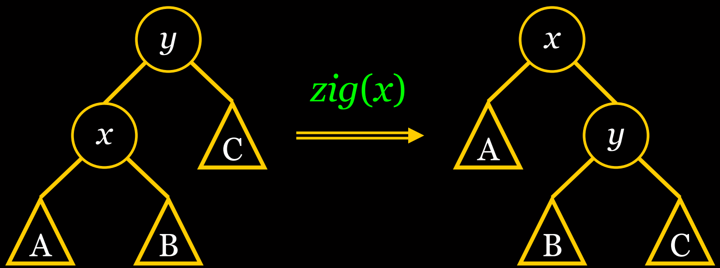
$(r'(y) - r'(x)) + (r'(z) - r'(x)) \leq -1$, rearranging

$\log(s'(y)/s'(x)) + \log(s'(z)/s'(x)) \leq -1$ definition of rank

TRUE because $s'(y)+s'(z)<s'(x)$: both ratios are smaller than 1, at least one is at most 1/2.

Proof of Access Lemma: Splaying Step

- Zig:



(only happens if y is root)

Claim: $a \le 1 + 6\,(r'(x) - r(x))$

$t + \Phi' - \Phi \le 1 + 6\,(r'(x) - r(x))$

$1 + (2r'(x) + 2r'(y)) - (2r(x) + 2r(y)) \le 1 + 6\,(r'(x) - r(x))$

$1 + 2\,(r'(x) - r(x)) \le 1 + 6\,(r'(x) - r(x))$, since $r(y) \ge r'(y)$

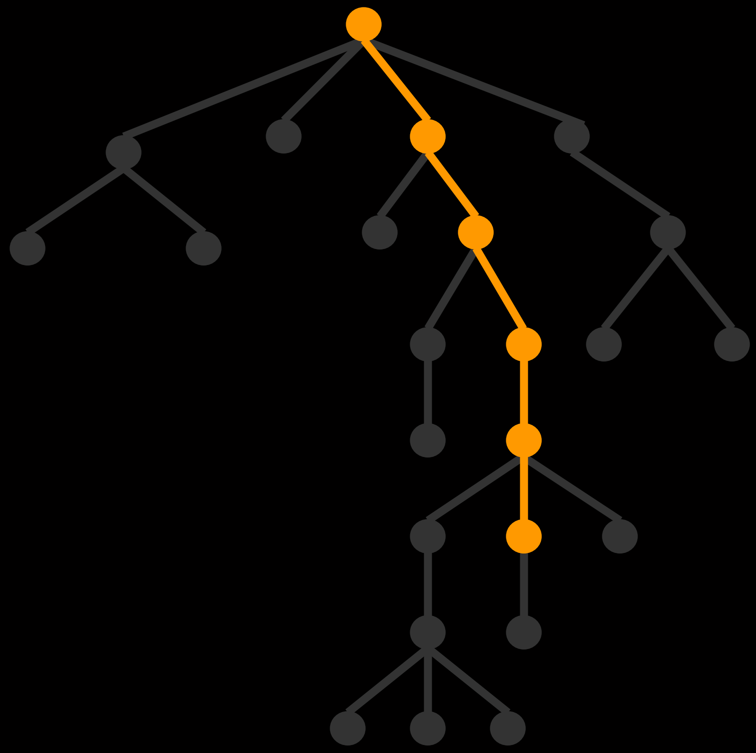
TRUE because $r'(x) \ge r(x)$.

Splaying

- To sum up:
  - No rotation: $a = 1$
  - Zig: $a \leq 6\ (r'(x) - r(x)) + 1$
  - Zig-zig: $a \leq 6\ (r'(x) - r(x))$
  - Zig-zag: $a \leq 4\ (r'(x) - r(x))$
  - Total amortized time at most $6\ (r(t) - r(x)) + 1 = O(\log n)$
- Since accesses bring the relevant element to the root, other operations (insert, delete, join, split) become trivial.

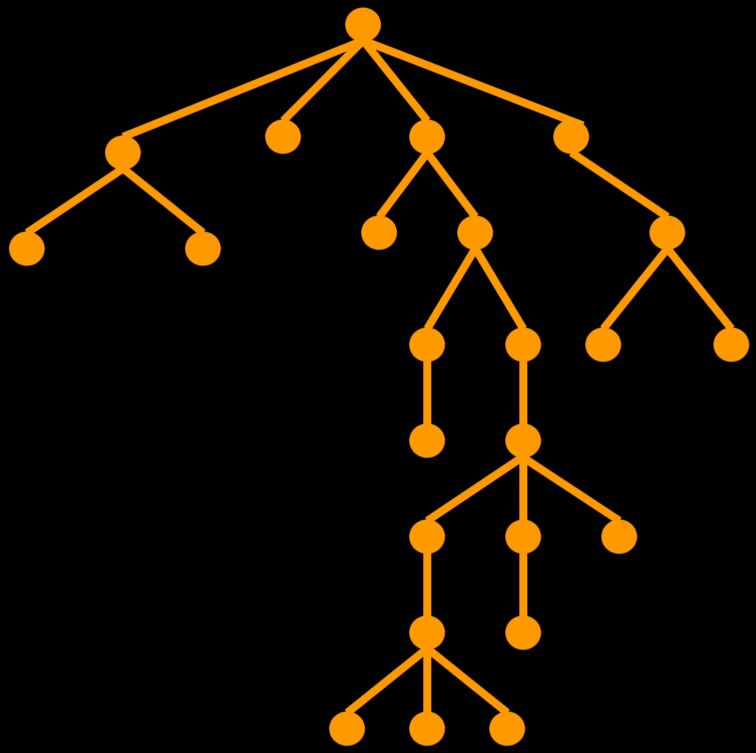
Dynamic Trees

- We know how to deal with isolated paths.
- How to deal with paths within a tree?



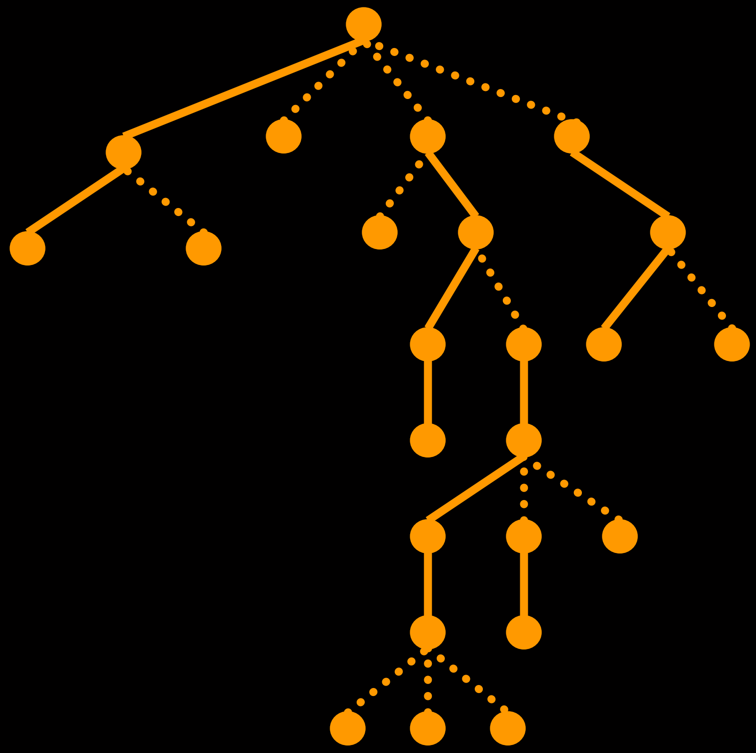
Dynamic Trees

- Main idea: partition the vertices in a tree into disjoint solid paths connected by dashed edges.



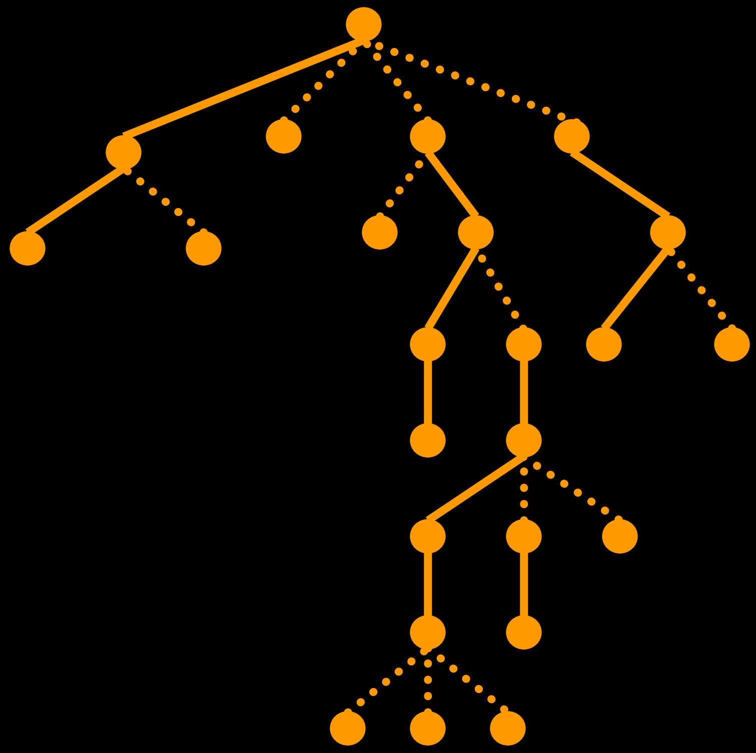
Dynamic Trees

- Main idea: partition the vertices in a tree into disjoint solid paths connected by dashed edges.



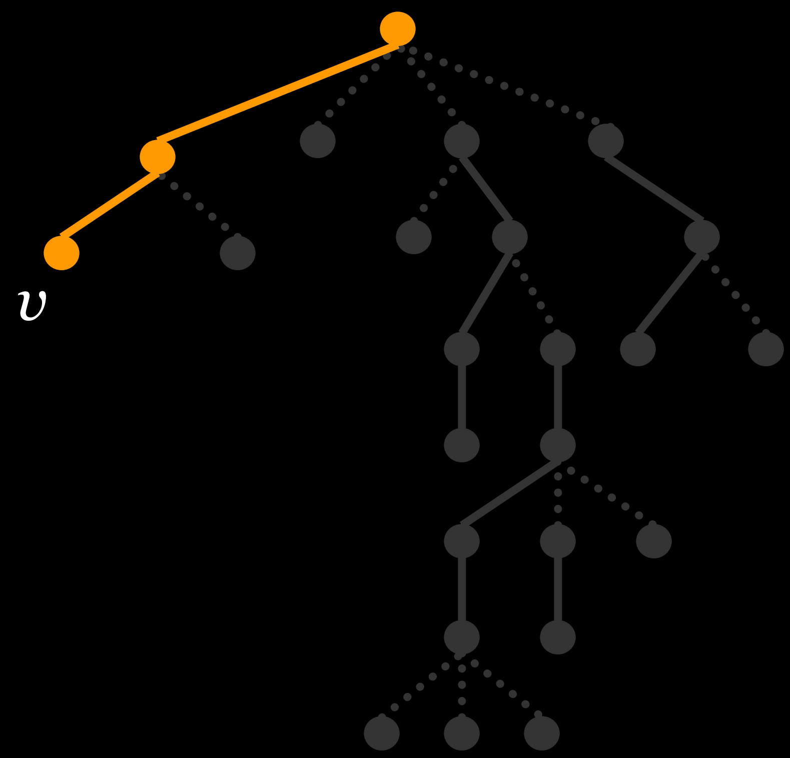
Dynamic Trees

- A vertex v is exposed if:
  - There is a solid path from v to the root;
  - No solid edge enters v.



Dynamic Trees

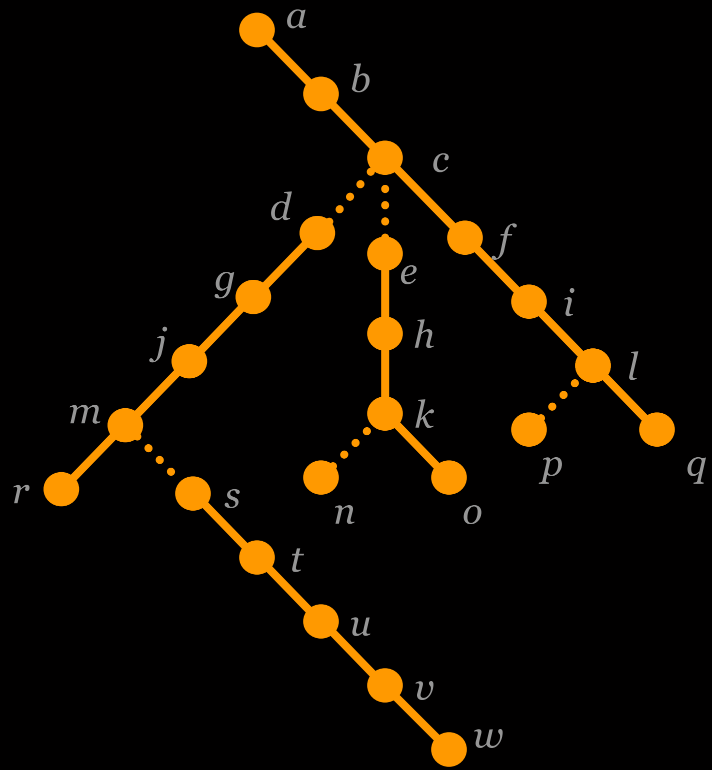
- A vertex v is exposed if:
  - There is a solid path from v to the root;
  - No solid edge enters v.
- It is unique.



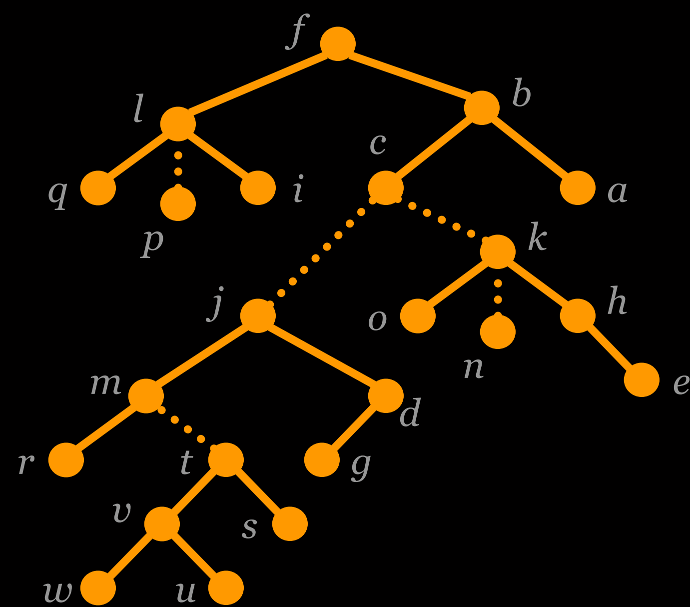
Dynamic Trees

- Solid paths:
  - Represented as binary trees (as seen before).
  - Parent pointer of root is the outgoing dashed edge.
  - Hierarchy of solid binary trees linked by dashed edges: "virtual tree".
- "Isolated path" operations handle the exposed path.
  - The solid path entering the root.
  - Dashed pointers go up, so the solid path does not "know" it has dashed children.
- If a different path is needed:
  - expose(v): make entire path from v to the root solid.

Virtual Tree: An Example



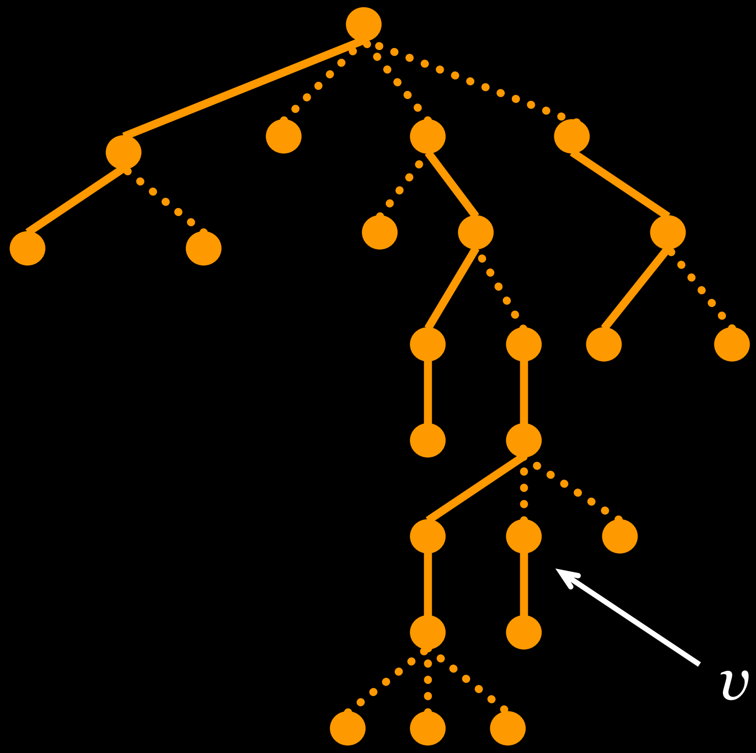
actual tree



virtual tree

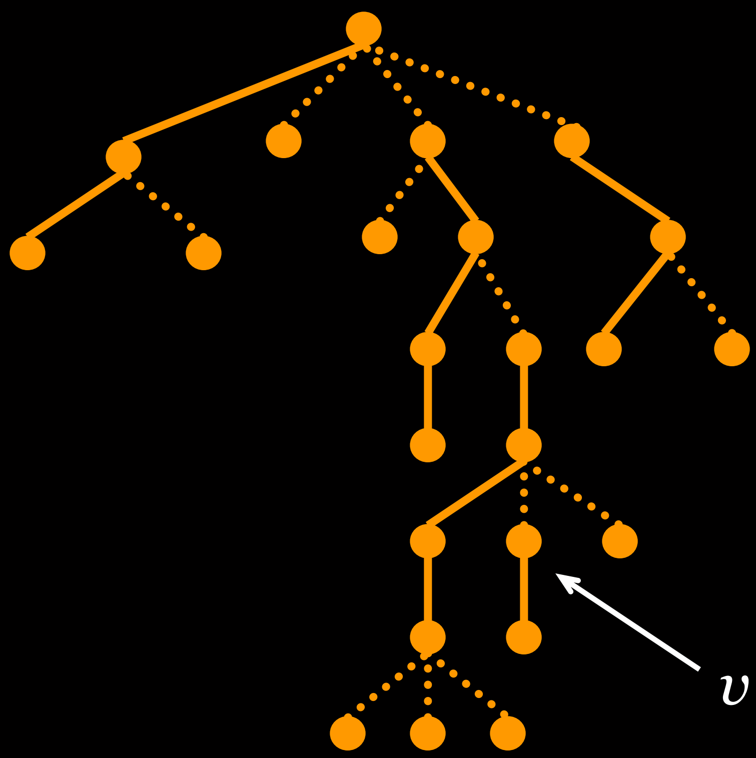
Dynamic Trees

- Example: expose(v)



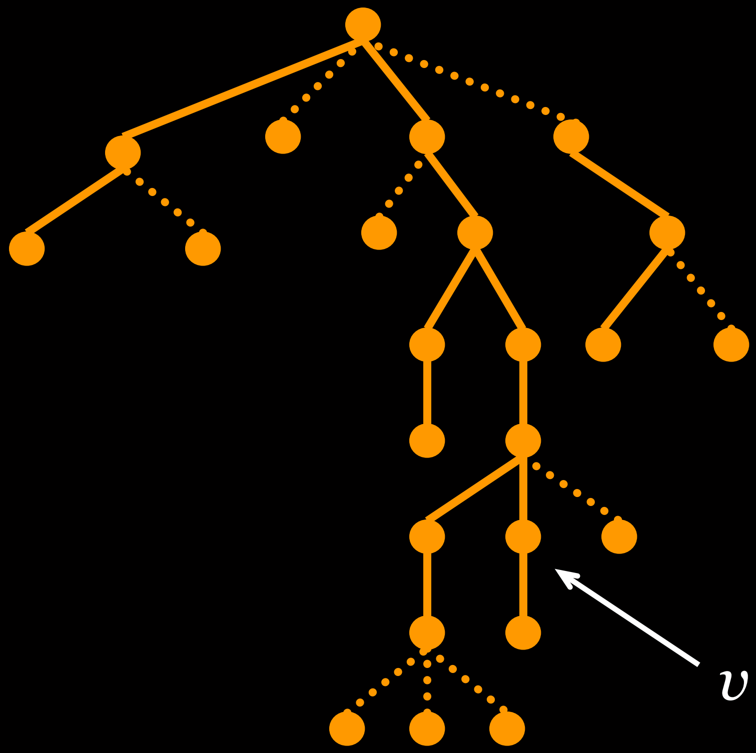
Dynamic Trees

- Example: expose(v)
  - Take all edges in the path to the root, ...



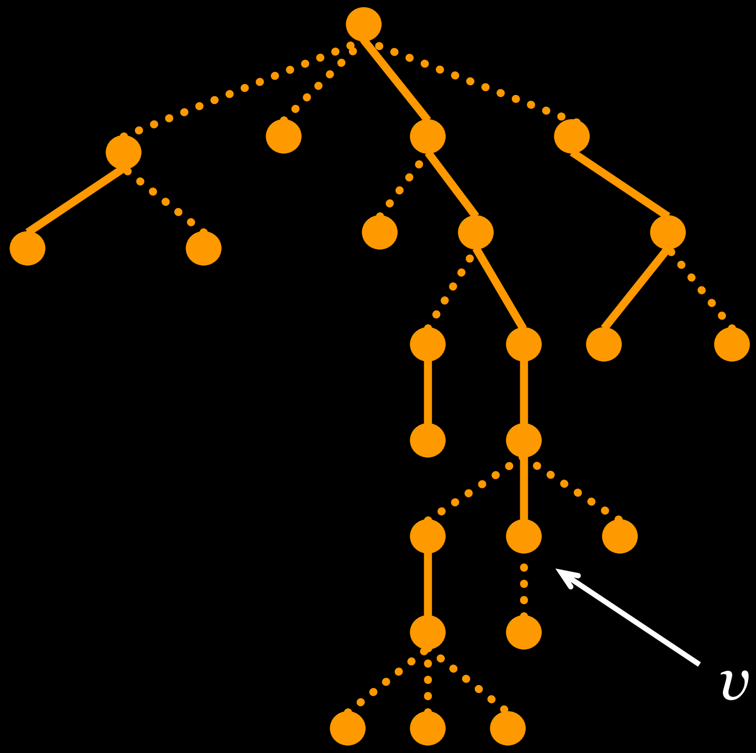
Dynamic Trees

- Example: expose(v)
  - ..., make them solid, ...



Dynamic Trees

- Example: expose(v)
  - …make sure there is no other solid edge incident into the path.
    - Uses splice operation.

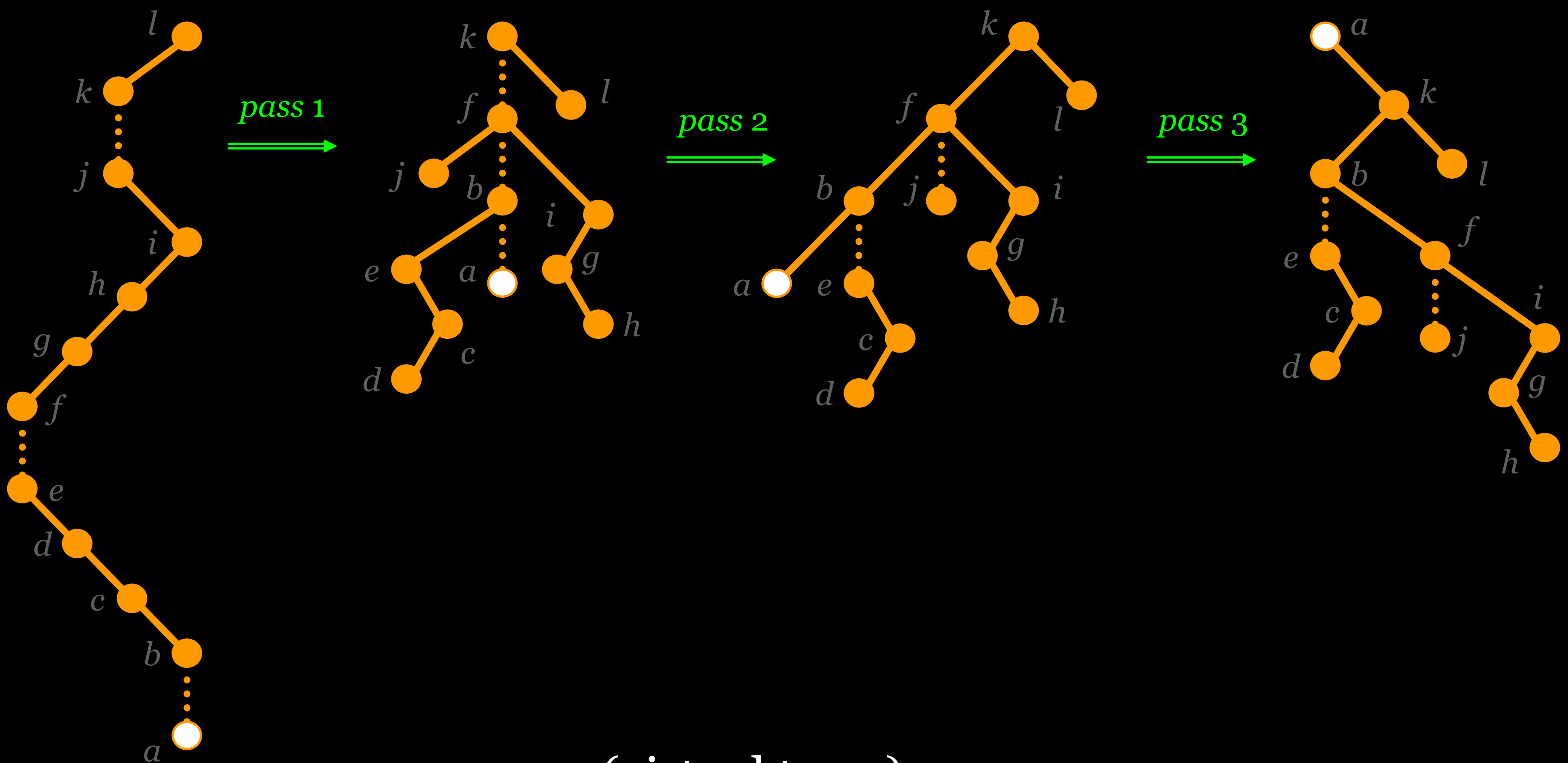


Exposing a Vertex

- expose(x): makes the path from x to the root solid.
- Implemented in three steps:
  1. Splay within each solid tree in the path from x to root.
  2. Splice each dashed edge from x to the root.
     - splice makes a dashed become the left solid child;
     - If there is an original left solid child, it becomes dashed.
  3. Splay on x, which will become the root.

Exposing a Vertex: An Example

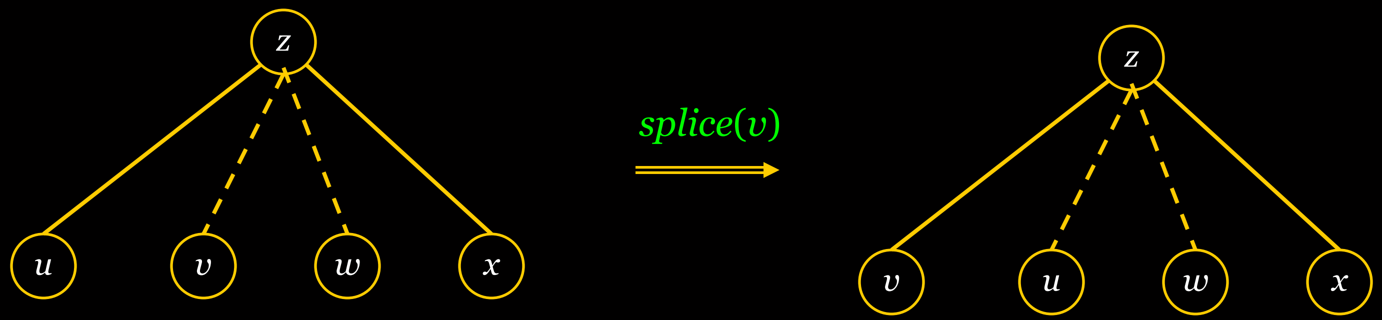
- expose(a)



(virtual trees)

Dynamic Trees: Splice

- Additional restructuring primitive: *splice*.



  - Will only occur when z is the root of a tree.

- Updates:
  - $\Delta cost'(v) = \Delta cost(v) - \Delta cost(z)$
  - $\Delta cost'(u) = \Delta cost(u) + \Delta cost(z)$
  - $\Delta min'(z) = \max\{0, \Delta min(v) - \Delta cost'(v), \Delta min(x) - \Delta cost(x)\}$

Exposing a Vertex: Running Time

- Running time of expose(x):
  - proportional to initial depth of x;
  - x is rotated all the way to the root;
  - we just need to count the number of rotations;
    - will actually find amortized number of rotations: $O(\log n)$.
  - proof uses the Access Lemma.
    - $s(x)$, $r(x)$ and potential are defined as before;
    - In particular, $s(x)$ is the size of the whole subtree rooted at x.
      - Includes both solid and dashed edges.

Exposing a Vertex: Running Time (Proof)

- k: number of dashed edges from x to the root t.
- Amortized costs of each pass:
  1. Splay within each solid tree:
     - x_i: vertex splayed on the i-th solid tree.
     - amortized cost of i-th splay: $6\,(r'(x_i) - r(x_i)) + 1$.
     - $r(x_{i+1}) \geq r'(x_i)$, so the sum over all steps telescopes;
     - Amortized cost first of pass: $6(r'(x_k) - r(x_1)) + k \leq 6 \log n + k$.
  2. Splice dashed edges:
     - no rotations, no potential changes: amortized cost is zero.
  3. Splay on x:
     - amortized cost is at most $6 \log n + 1$.
     - x ends up in root, so exactly k rotations happen;
     - each rotation costs one credit, but is charged two;
     - they pay for the extra k rotations in the first pass.
- Amortized number of rotations = $O(\log n)$.

Implementing Dynamic Tree Operations

- findcost(v):
  - expose v, return $cost(v)$.
- findroot(v):
  - expose v;
  - find w, the rightmost vertex in the solid subtree containing v;
  - splay at w and return w.
- findmin(v):
  - expose v;
  - use $\Delta cost$ and Δmin to walk down from v to w, the last minimum-cost node in the solid subtree;
  - splay at w and return w.

Implementing Dynamic Tree Operations

- addcost(v, x):
  - expose v;
  - add x to $\Delta cost(v)$;
- link(v,w):
  - expose v and w (they are in different trees);
  - set $p(v)=w$ (that is, make v a middle child of w).
- cut(v):
  - expose v;
  - add $\Delta cost(v)$ to $\Delta cost(right(v))$;
  - make $p(right(v))$=**null** and $right(v)$=**null**.

Extensions and Variants

- Simple extensions:
  - Associate values with edges:
    - just interpret cost(v) as cost(v,p(v)).
  - other path queries (such as length):
    - change values stored in each node and update operations.
  - free (unrooted) trees.
    - implement evert operation, which changes the root.
- Not-so-simple extension:
  - subtree-related operations:
    - requires that vertices have bounded degree;
    - Approach for arbitrary trees: "ternarize" them:
      - [Goldberg, Grigoriadis and Tarjan, 1991]

Alternative Implementation

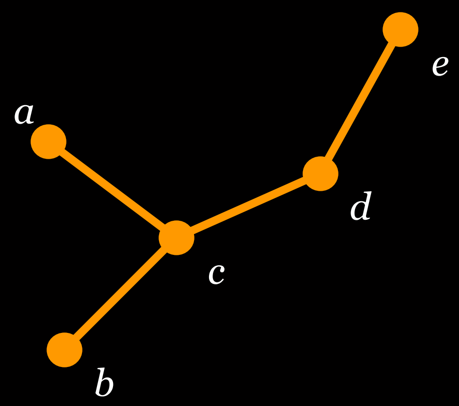
- Total time per operation depends on the data structure used to represent paths:
  - Splay trees: O(log n) amortized [ST85].
  - Balanced search tree: O($\log^2 n$) amortized [ST83].
  - Locally biased search tree: O(log n) amortized [ST83].
  - Globally biased search trees: O(log n) worst-case [ST83].
- Biased search trees:
  - Support leaves with different "weights".
  - Some solid leaves are "heavier" because they also represent subtrees dangling from it from dashed edges.
  - Much more complicated than splay trees.

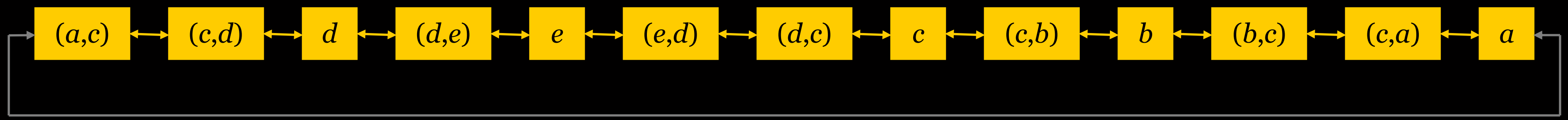
Other Data Structures

- Some applications require tree-related information:
  - minimum vertex in a tree;
  - add value to all elements in the tree;
  - link and cut as usual.
- ET-Trees can do that:
  - Henzinger and King (1995);
  - Tarjan (1997).

ET-Trees

- Each tree represented by its *Euler tour*.
  - Edge $\{v,w\}$:
    - appears as arcs (v,w) and (w,v)
  - Vertex v:
    - appears once as a self-loop (v,v):
    - used as an “anchor” for new links.
    - stores vertex-related information.
  - Representation is not circular: tour broken at arbitrary place.





ET-Trees

- Consider link(v,w):
  - Create elements representing arcs (v,w) and (w,v):

    $(v,w) \quad (w,v)$

  - Split and concatenate tours appropriately:
    - Original tours:

      $L_v \leftrightarrow v \leftrightarrow L'_v \qquad L_w \leftrightarrow w \leftrightarrow L'_w$

    - Final tour:

      $L_v \leftrightarrow v \leftrightarrow (v,w) \leftrightarrow L'_w \leftrightarrow L_w \leftrightarrow w \leftrightarrow (w,v) \leftrightarrow L'_v$

- The cut operation is similar.

ET-Trees

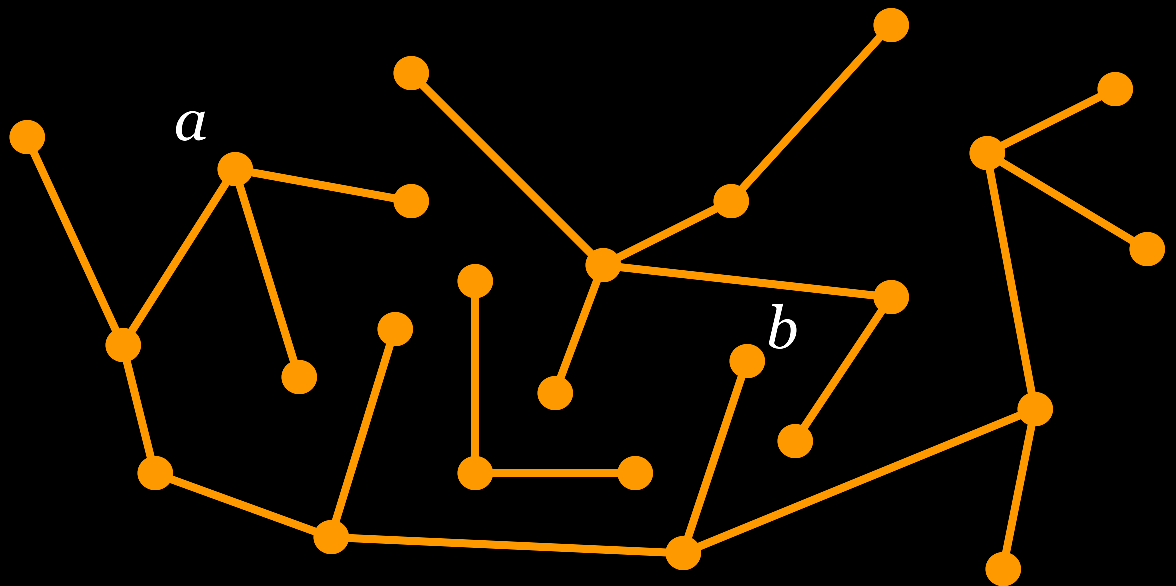
- Tours as doubly-linked lists:
  - Natural representation.
  - link/cut: O(1) time.
  - addcost/findmin: $O(n)$ time.
- Tours as balanced binary search trees:
  - link/cut: O(log n) time (binary tree join and split).
  - addcost/findmin: O(log n) time:
    - values stored in difference form.

Contractions

- ST-Trees [ST83, ST85]:
  - first data structure to handle paths within trees efficiently.
  - It is clearly path-oriented:
    - relevant paths explicitly exposed and dealt with.
- Other approaches are based on contractions:
  - Original tree is progressively contracted until a structure representing only the relevant path (or tree) is left.

Contractions

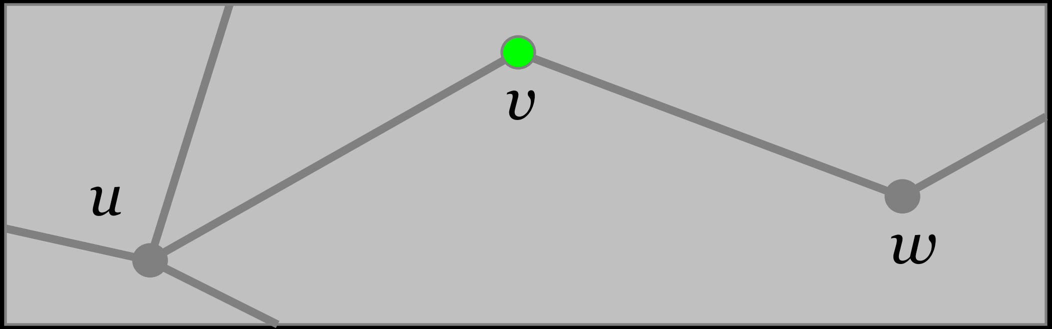
- Assume we are interested in the path from a to b:



  - Using only local information, how can we get closer to the solution?

Contractions

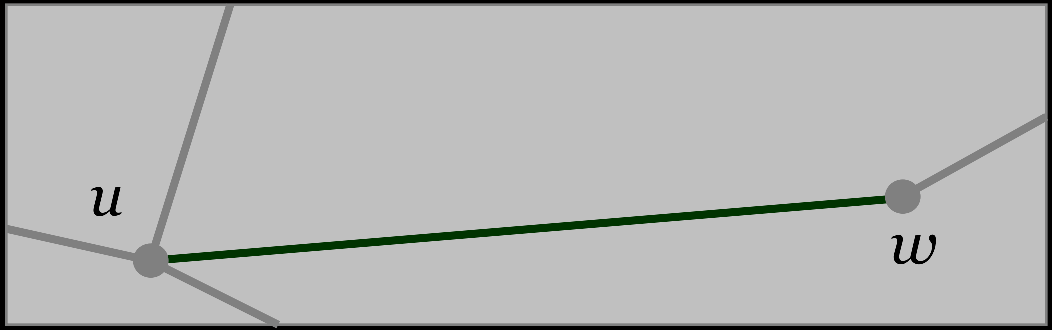
- Consider any vertex v with degree 2 in the tree



- Possibilities if v is neither a nor b:
  - a and b on same "side": v is not in $a \cdots b$.
  - If a and b on different sides: v belongs to path $a \cdots b$.

Contractions

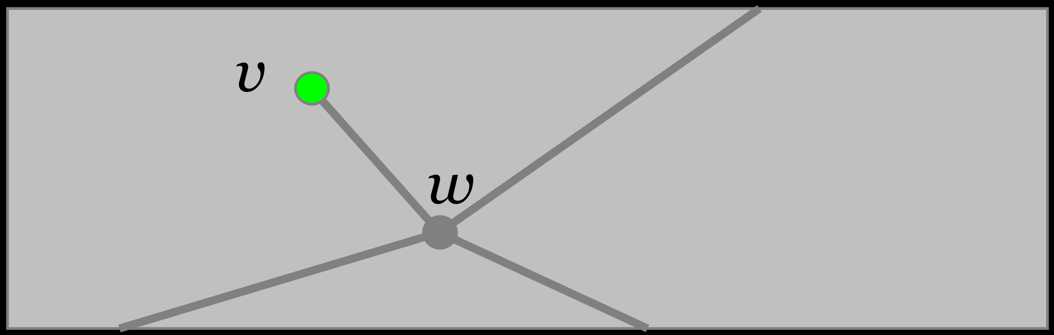
- Consider any vertex v with degree 2 in the tree



- Possibilities if v is neither a nor b:
  - a and b on same "side": v is not in $a \cdot\cdot b$.
  - If a and b on different sides: v belongs to path $a \cdot\cdot b$.
- We can replace (u,v) and (v,w) with a new edge (u,w):
  - This is a compress operation.

Contractions

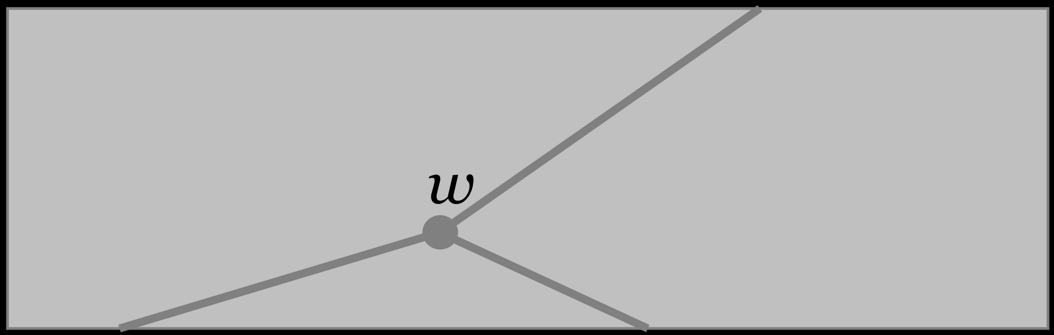
- Consider any vertex v with degree 1 in the tree:



  - If v is neither a nor b, it is clearly not in $a \cdots b$.

Contractions

- Consider any vertex v with degree 1 in the tree:



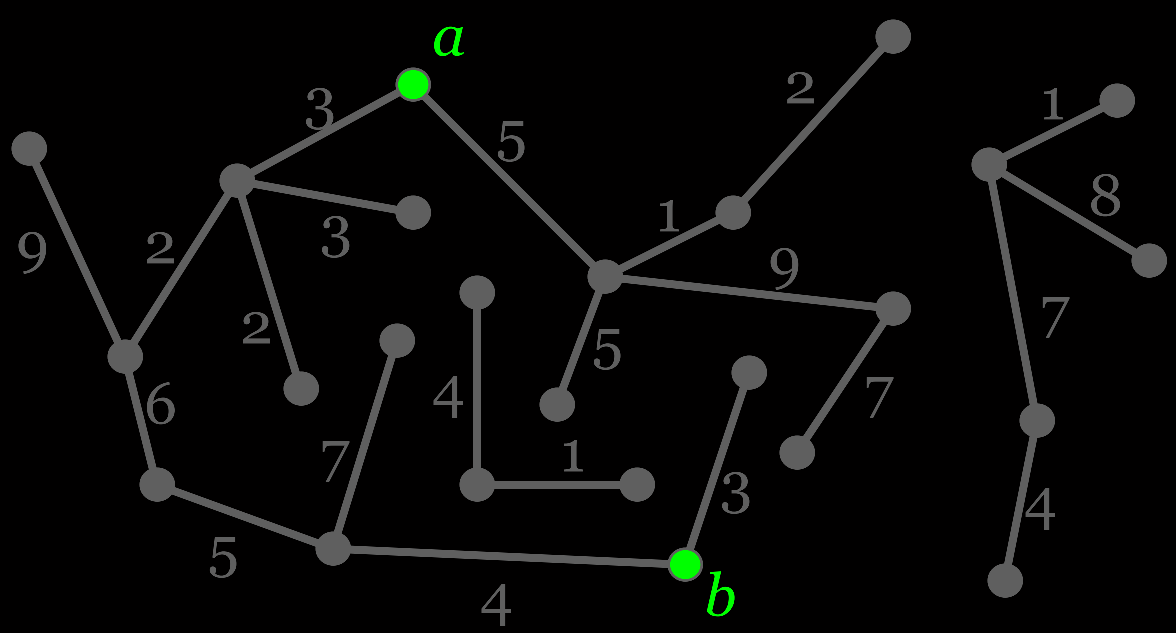
  - If v is neither a nor b, it is clearly not in $a \cdots b$.
  - We can simply eliminate (v,w), reducing the problem size.
    - This is a rake operation.

Contractions

- A contraction-based algorithm:
  - Work in rounds;
  - In each round, perform some rakes and/or compresses:
    - this will create a new, smaller tree;
    - moves within a round are usually "independent".
  - Eventually, we will be down to a single element (vertex/edge) that represents a path (or the tree).

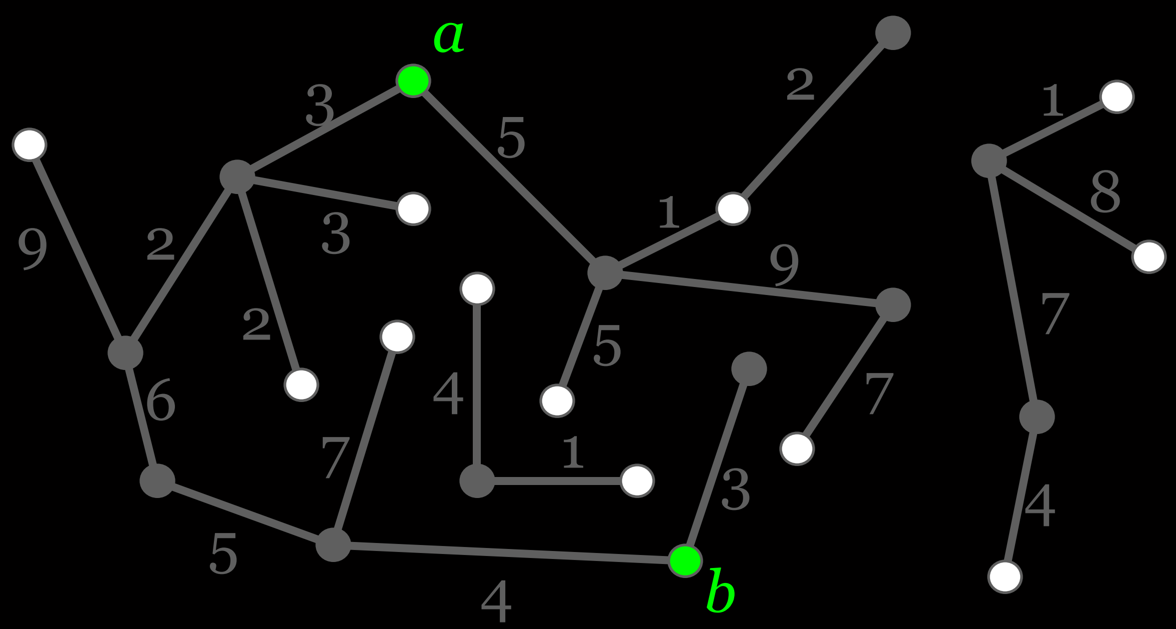
Path Queries

- Computing the minimum cost from a to b:



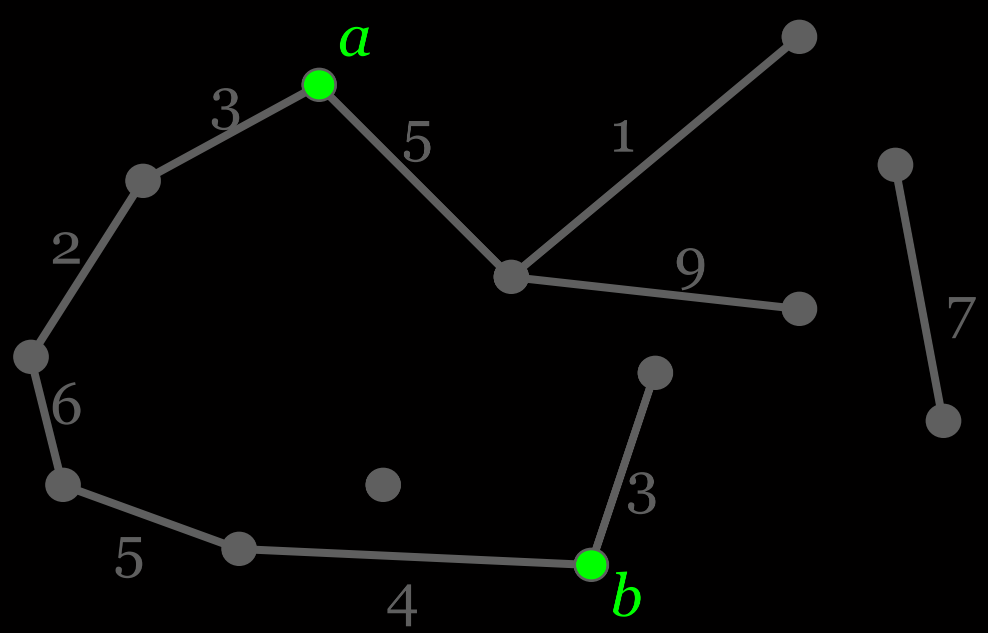
Path Queries

- Computing the minimum cost from *a* to *b*:



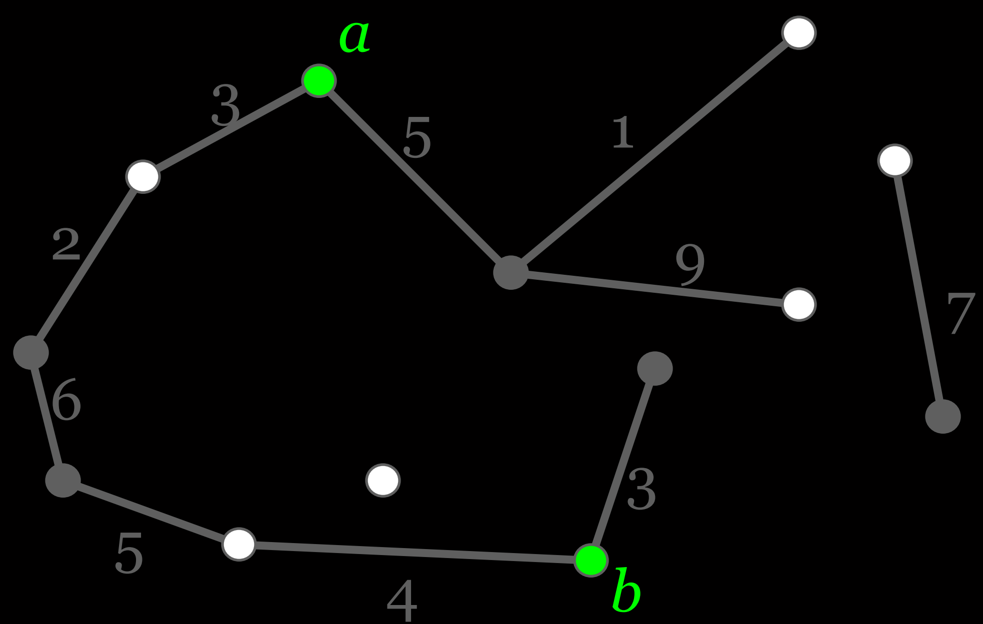
Path Queries

- Computing the minimum cost from a to b:



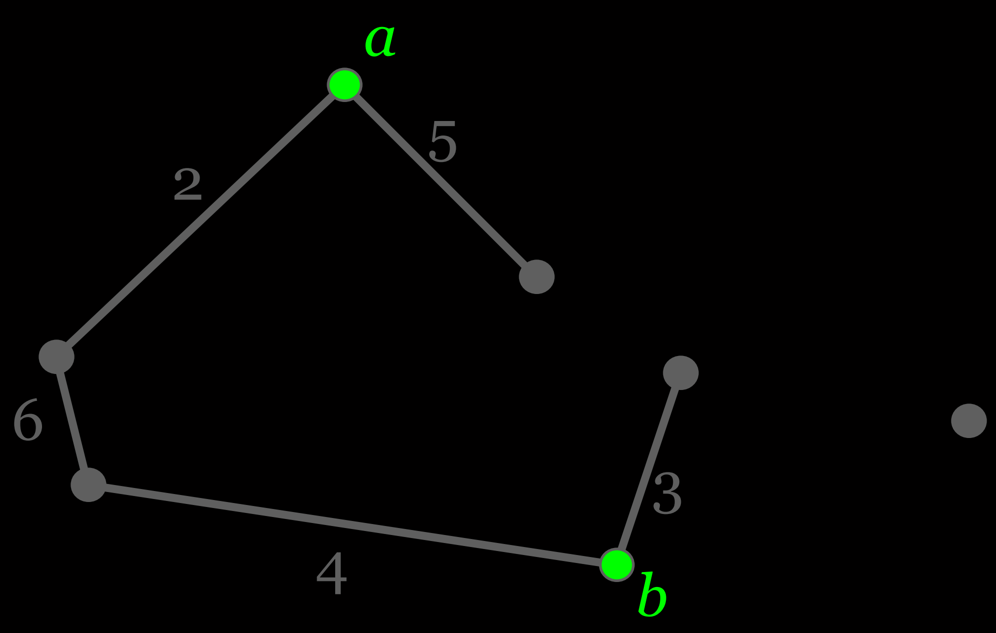
Path Queries

- Computing the minimum cost from *a* to *b*:



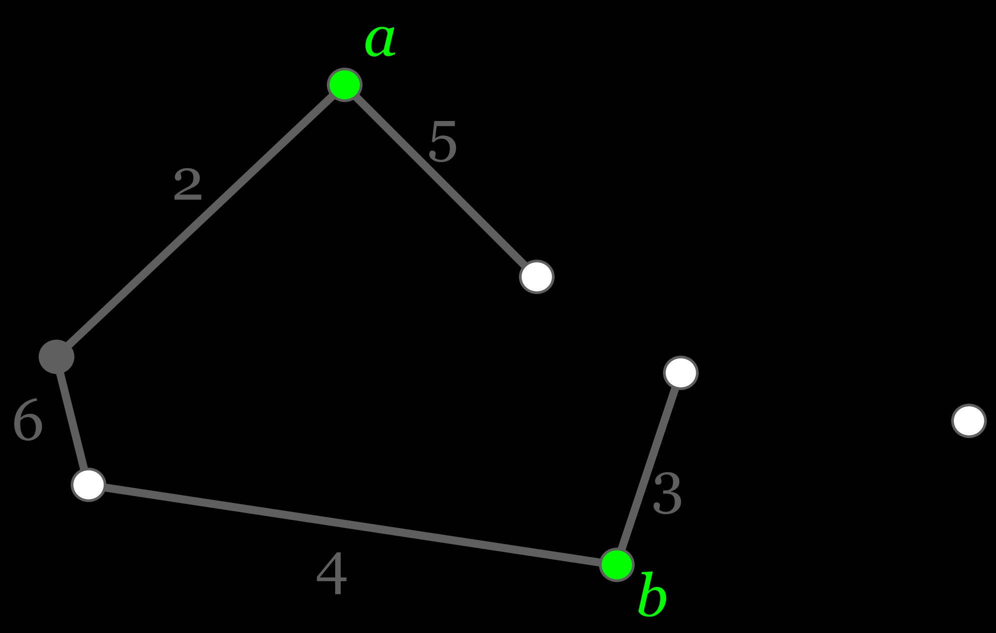
Path Queries

- Computing the minimum cost from a to b:



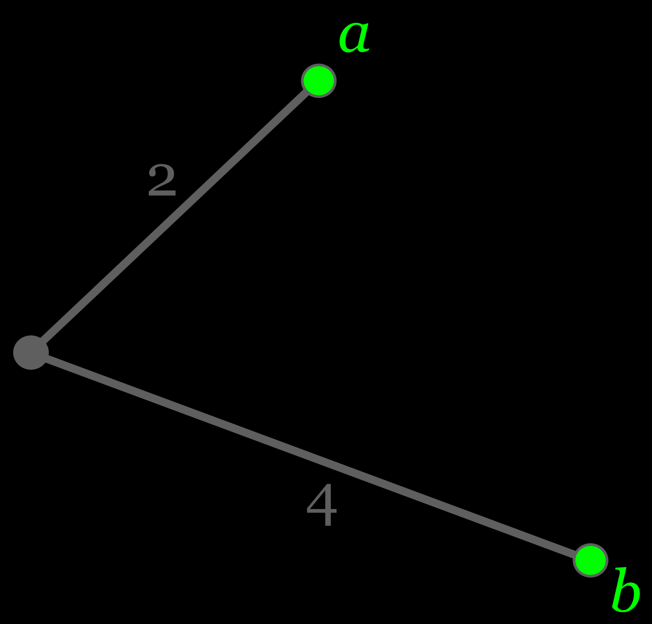
Path Queries

- Computing the minimum cost from a to b:



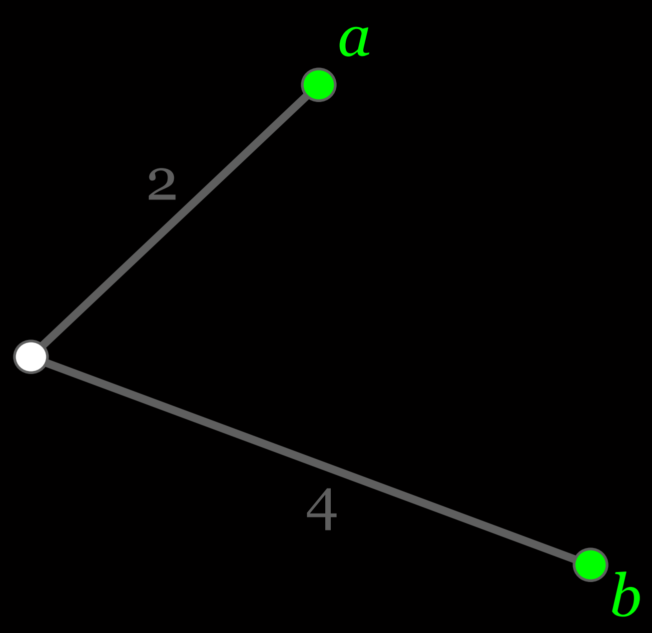
Path Queries

- Computing the minimum cost from *a* to *b*:



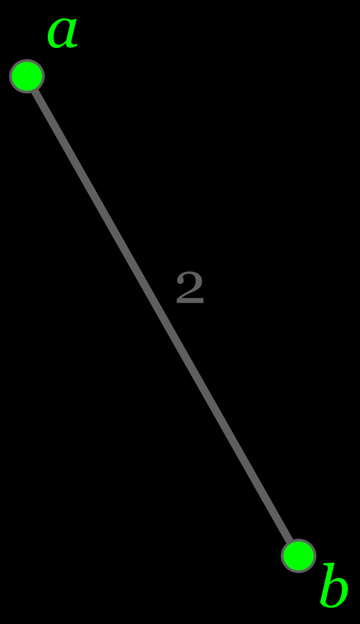
Path Queries

- Computing the minimum cost from *a* to *b*:



Path Queries

- Computing the minimum cost from a to b:



Contractions

- Suppose a definition of independence guarantees that a fraction $1/k$ of all possible rakes and compresses will be executed in a round.
  - All degree-1 vertices are rake candidates.
  - All degree-2 vertices are compress candidates.
  - *Fact*: at least half the vertices in any tree have degree 1 or 2.
  - *Result*: a fraction $1/2k$ of all vertices will be removed.
  - Total number of rounds is $\lceil \log_{(2k)/(2k-1)} n \rceil = O(\log n)$.

Contractions

- rake and compress proposed by Miller and Reif [1985].
  - Original context: parallel algorithms.
  - Perform several operations on trees in O(log n) time.

The Update Problem

- Coming up with a definition of independence that results in a contraction with O(log n) levels.
  - But that is not the problem we need to solve.
- Essentially, we want to repair an existing contraction after a tree operation (link/cut).
- So we are interested in the update problem:
  - Given a contraction C of a forest F, find another contraction C' of a forest F' that differs from F in one single edge (inserted or deleted).
  - Fast: O(log n) time.

Our Problem

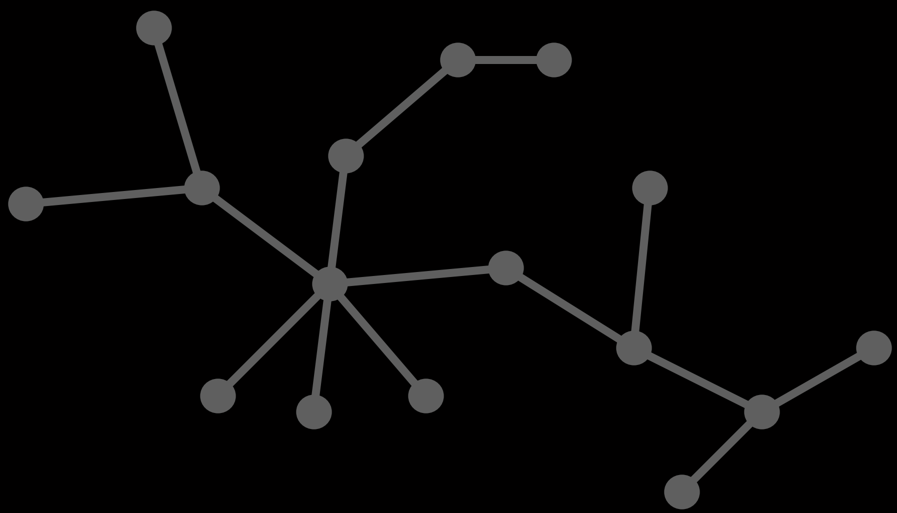
- Several data structures deal with this problem.
  - [Frederickson, 85 and 97]: Topology Trees;
  - [Alstrup et al., 97 and 03]: Top Trees;
  - [Acar et al. 03]: RC-Trees.

Top Trees

- Proposed by Alstrup et al. [1997,2003]
- Handle unrooted (free) trees with arbitrary degrees.
- Key ideas:
  - Associate information with the edges directly.
  - Pair edges up:
    - compress: combines two edges linked by a degree-two vertex;
    - rake: combines leaf with an edge with which it shares an endpoint.
    - All pairs (clusters) must be are disjoint.
  - expose: determines which two vertices are relevant to the query (they will not be raked or compressed).

Top Trees

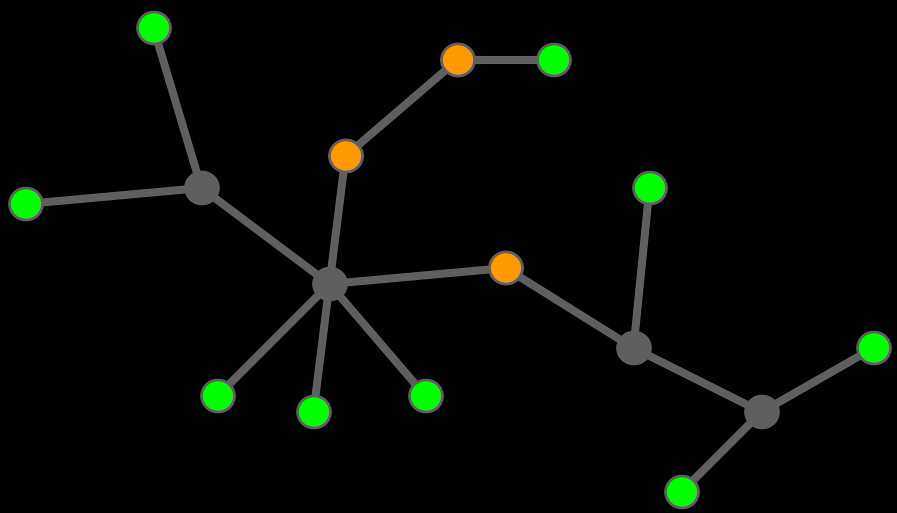
- Consider some free tree.



(level zero: original tree)

Top Trees

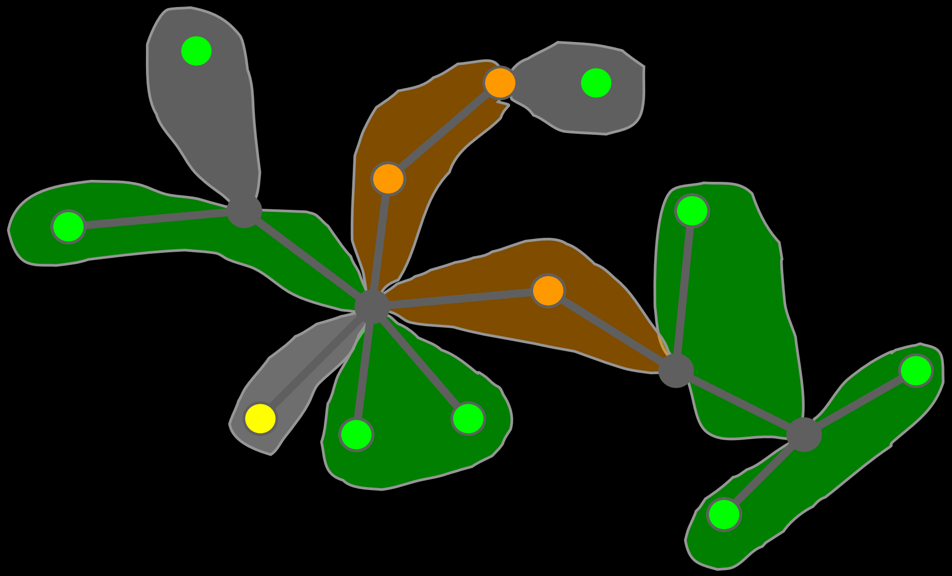
- All degree-1 and degree-2 vertices are candidates for a move (rake or compress).



(level zero: original tree)

Top Trees

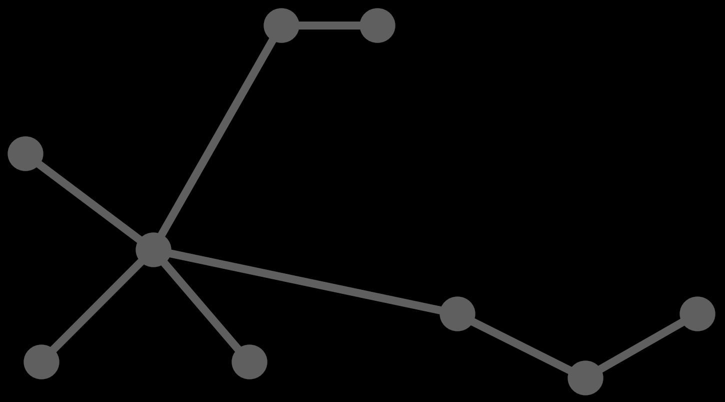
- When two edges are matched, they create new clusters, which are edge-disjoint.



(level zero: original tree)

Top Trees

- Clusters are new edges in the level above:
  - New rakes and compresses can be performed as before.



(level one)

Top Trees

- The top tree itself represents the hierarchy of clusters:
  - original edge: leaf of the top tree (level zero).
  - two edges/clusters are grouped by rake or compress:
    - Resulting cluster is their parent in the level above.
  - edge/cluster unmatched: parent will have only one child.
- What about values?

Top Trees

- Alstrup et al. see top tree as an API.
- The top tree engine handles structural operations:
  - User has limited access to it.
- Engine calls user-defined functions to handle values properly:
  - join(A,B,C): called when A and B are paired (by rake or compress) to create cluster C.
  - split(A,B,C): called when a rake or compress is undone (and C is split into A and B).
  - create(C, e): called when base cluster C is created to represent edge e.
  - destroy(C): called when base cluster C is deleted.

Top Trees

- Example (path operations: findmin/addcost)
  - Associate two values with each cluster:
    - mincost(C): minimum cost in the path represented by C.
    - extra(C): cost that must be added to all subpaths of C.
  - create(C, e): (*called when base cluster C is created*)
    - mincost(C) = cost of edge e.
    - extra(C) = 0
  - destroy(C): (*called when base cluster C is deleted*).
    - Do nothing.

Top Trees

- Example (path operations: findmin/addvalue)
  - join(A,B,C): (*called when A and B are combined into C*)
    - compress: mincost(C) = min{mincost(A), mincost(B)}
    - rake: mincost(C) = mincost(B) (assume A is the leaf)
    - Both cases: *extra*(C) = 0
  - split(A,B,C): (*called when C is split into A and B*)
    - compress: for each child $X \in \{A,B\}$:
      - mincost(X) = mincost(X) + extra(C)
      - extra(X) = extra(X) + extra(C)
    - rake: same as above, but only for the edge/cluster that was not raked.

Top Trees

- Example (path operations: findmin/addvalue)
  - To find the minimum cost in path $a \cdots b$:
    - R = expose(a, b);
    - return mincost(R).
  - To add a cost x to all edges in path $a \cdots b$:
    - R = expose(a, b);
    - mincost(R) = mincost(R) + x;
    - extra(R) = extra(R) + x.

Top Trees

- Can handle operations such as:
  - tree costs (just a different way of handling rakes);
  - path lengths;
  - tree diameters.
- Can handle non-local information using the select operation:
  - allows user to perform binary search on top tree.
  - an example: tree center.
- Top trees are implemented on top of topology trees, which they generalize.

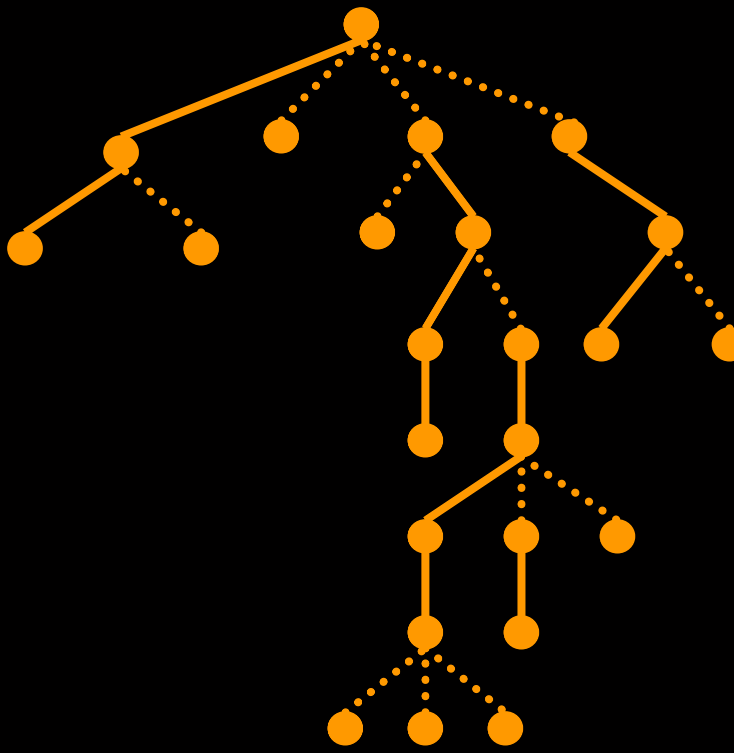
Topology Trees

- Proposed by Frederickson [1985, 1997].
- Work on rooted trees of bounded degree.
  - Assume each vertex has at most two children.
    - Values (and clusters) are associated with vertices.
  - Perform a maximal set of independent moves in each round.
  - Handle updates in O(log n) worst-case time.

RC-Trees

- Proposed by Acar et al. [2003].
- Can be seen as a variant of topology trees.
  - Information stored on vertices.
  - Trees of bounded degree.
- Main differences:
  - Not necessarily rooted.
  - Alternate rake and compress rounds.
  - Not maximal in compress rounds (randomization).
  - Updates in $O(\log n)$ expected time.

Contractions



- Topology, Top, and Trace trees:
  - contraction-based.
- ST-Trees: path-based.
  - But there is a (rough) mapping:
    - dashed $\leftrightarrow$ rake
      - “this is a path that goes nowhere”
    - solid $\leftrightarrow$ compress
      - “both part of a single path”
  - ST-Trees can be used to implement topology trees [AHdLT03].

Chronology

- ST-Trees:
  - Sleator and Tarjan (1983): with balanced and biased search trees;
  - Sleator and Tarjan (1985): splay trees.
- Topology Trees:
  - Frederickson (1985, 1987).
- ET-trees:
  - Hensinger and King (1995);
  - Tarjan (1997).
- Top Trees:
  - Alstrup, de Lichtenberg, and Thorup (1997);
  - Alstrup, Holm, de Lichtenberg, and Thorup (2003).
- RC-Trees:
  - Acar, Blelloch, Harper, and Woo (2003).