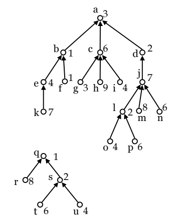
Dynamic Trees

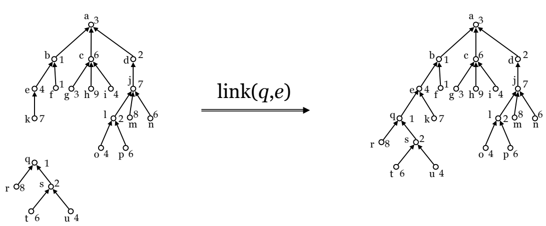
- Goal: maintain a forest of rooted trees with costs on vertices.
  - Each tree has a root, every edge directed towards the root.
- Operations allowed:
  - link(v,w): creates an edge between v (a root) and w.
  - cut(v,w): deletes edge (v,w).
  - findcost(v): returns the cost of vertex v.
  - findroot(v): 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

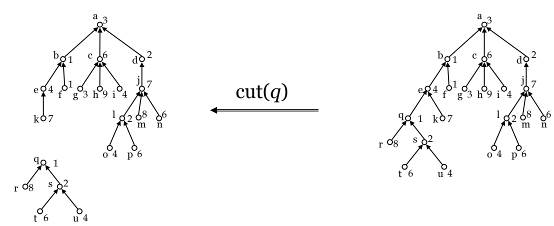
- An example (two trees):



Dynamic Trees

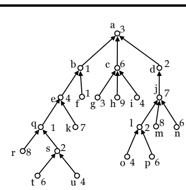


Dynamic Trees

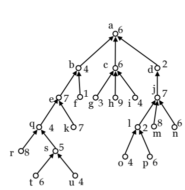


Dynamic Trees

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



- addcost(s,3)

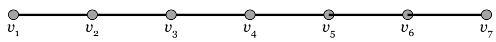


Obvious Implementation

- 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.

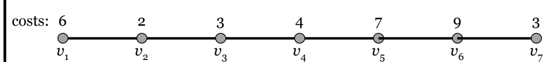
Simple Paths

- 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.



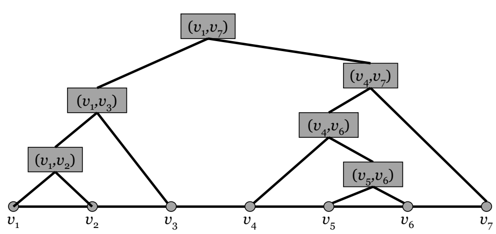
Simple Paths as Lists

- 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?



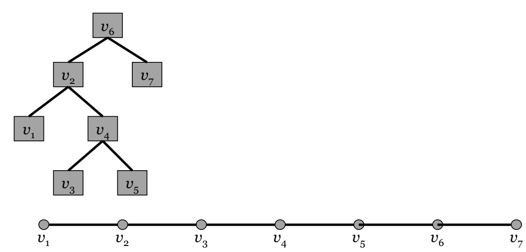
Simple Paths as Binary Trees

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



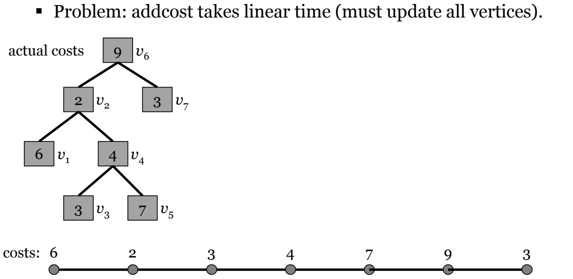
Simple Paths as Binary Trees

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



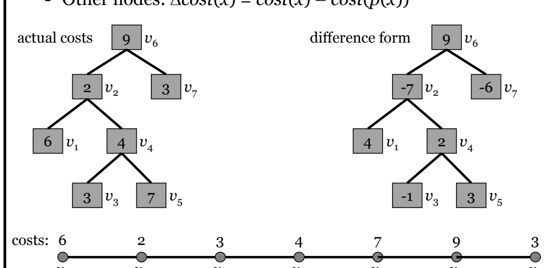
Simple Paths: Maintaining Costs

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



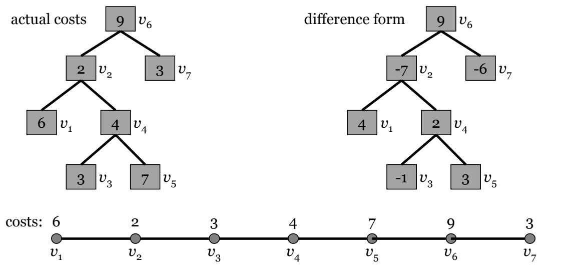
Simple Paths: Maintaining Costs

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



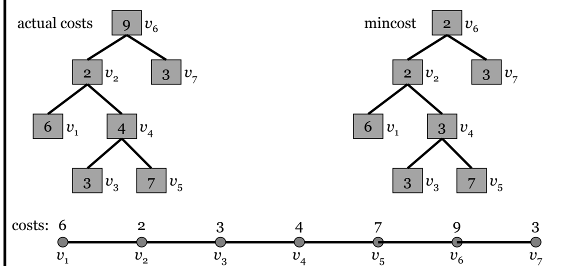
Simple Paths: Maintaining Costs

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



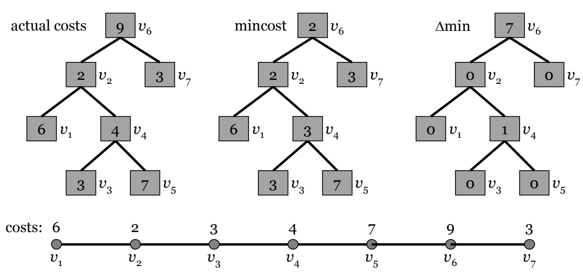
Simple Paths: Finding Minima

- 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.



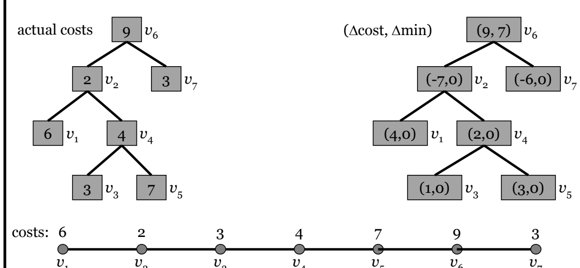
Simple Paths: Finding Minima

- Store Δmin(x) = cost(x)–mincost(x) instead.
  - findmin still runs in O(log n) time, addcost now constant.



Simple Paths: Data Fields

- Final version:
  - Stores Δmin(x) and Δcost(x) for every vertex

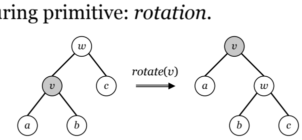


Simple Paths: Structural Changes

- 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*.



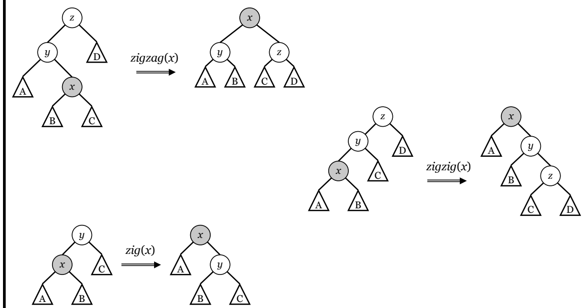
- Fields are updated as follows (for left and right rotations):
  - $\Delta cost'(v) = \Delta cost(v) + \Delta cost(w)$
  - $\Delta cost'(w) = -\Delta cost(v)$
  - $\Delta cost'(b) = \Delta cost(v) + \Delta cost(b)$
  - $\Delta min'(w) = \max\{0, \Delta min(b) - \Delta cost'(b), \Delta min(c) - \Delta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

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.

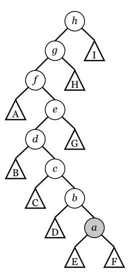
Splaying

- Three restructuring operations:

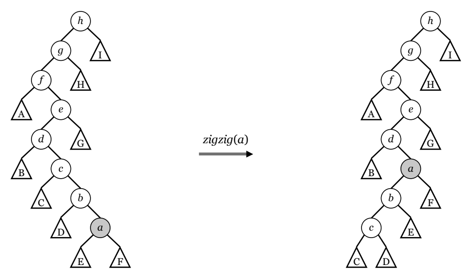


(only happens if y is root)

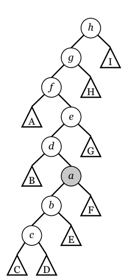
An Example of Splaying



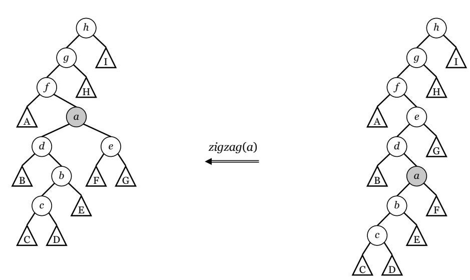
An Example of Splaying



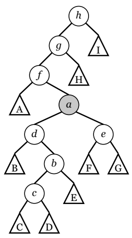
An Example of Splaying



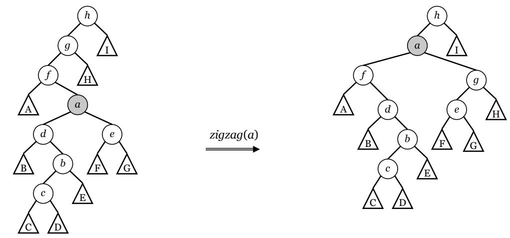
An Example of Splaying



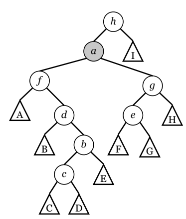
An Example of Splaying



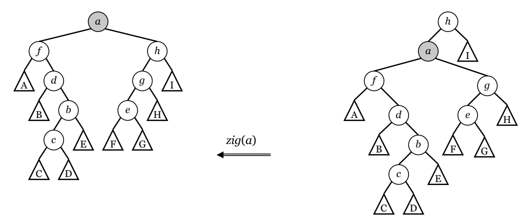
An Example of Splaying



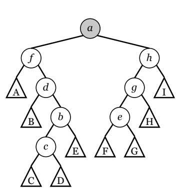
An Example of Splaying



An Example of Splaying

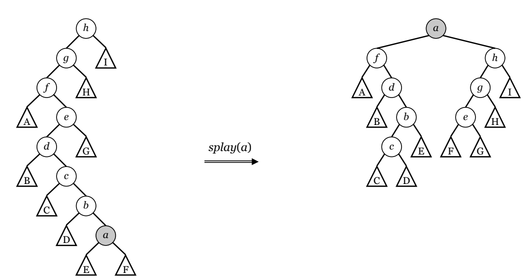


An Example of Splaying



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*

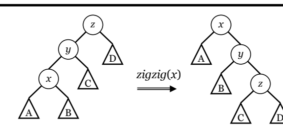
$$6(r(t)-r(x)) + 1 = O(\log(s(t)/s(x))).$$

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

$$\Sigma_{i=1..k}\, a_i \le \Sigma_{i=1..k-1}[6(r_i(x)-r_{i-1}(x))] + [6(r_k(x)-r_{k-1}(x)) + 1]$$
$$= 6r_k(x) - 6r_0(x) + 1$$

Proof of Access Lemma: Splaying Step

- Zig-zig:



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

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

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

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

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

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

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

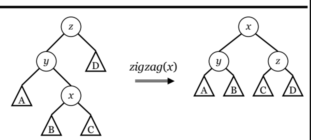
$(r(x) - r'(x)) + (r'(z) - r'(x)) \le -1$ rearranging

$\log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \le -1$ 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: $a \le 4\,(r'(x) - r(x))$

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

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

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

$2 + 2r'(y) + 2r'(z) - 4r(x) \le 4\,(r'(x) - r(x))$, since $r(y) \ge r(x)$

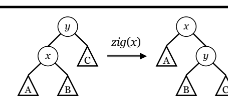
$(r'(y) - r'(x)) + (r'(z) - r'(x)) \le -1$, rearranging

$\log(s'(y)/s'(x)) + \log(s'(z)/s'(x)) \le -1$ 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 $r(y) \ge r'(y)$

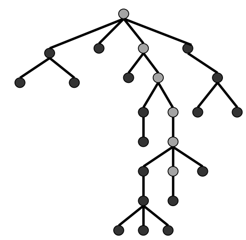
TRUE because $r'(x) \ge r(x)$.

Splaying

- To sum up:
  - No rotation: $a = 1$
  - Zig: $a \le 6\,(r'(x) - r(x)) + 1$
  - Zig-zig: $a \le 6\,(r'(x) - r(x))$
  - Zig-zag: $a \le 4\,(r'(x) - r(x))$
  - 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.

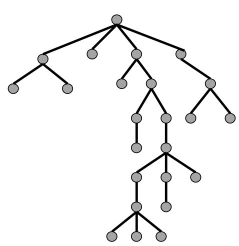
Dynamic Trees

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



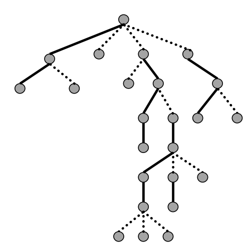
Dynamic Trees

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



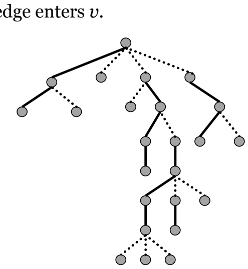
Dynamic Trees

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



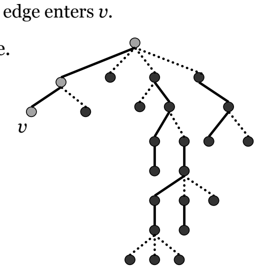
Dynamic Trees

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



Dynamic Trees

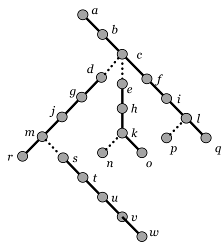
- A vertex v is exposed if:
  - There is a solid path from v to the root;
  - No solid edge enters v.
- It is unique.



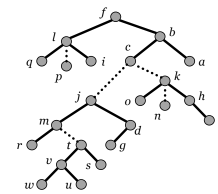
Dynamic Trees

- 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



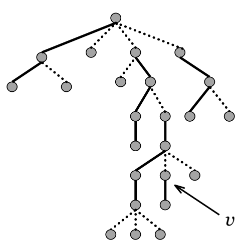
actual tree



virtual tree

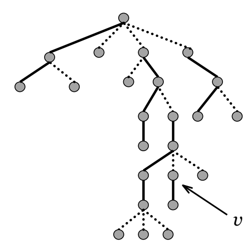
Dynamic Trees

- Example: expose(v)



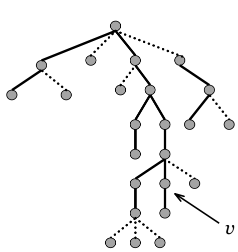
Dynamic Trees

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



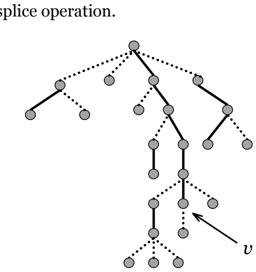
Dynamic Trees

- 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.

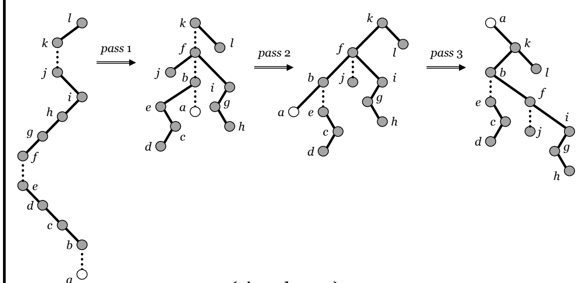


Exposing a Vertex

- 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

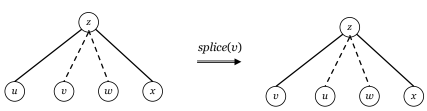
- expose(a)



(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.