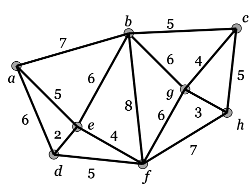
Dynamic Trees

- Motivation (online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Online Minimum Spanning Trees

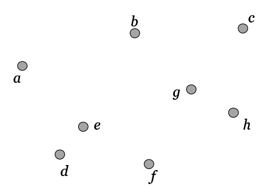
- The *online minimum spanning trees* problem:
  - Input: a sequence of edges (with costs), one at a time.
  - Goal: keep the minimum spanning forest of the graph.
- An algorithm:
  - For each new edge (v,w):
    - If v and w belong to different components, insert the edge.
    - If v and w are in the same component:
      - Insert (v,w) into the solution; and
      - Remove the most expensive edge in the cycle created.

Online Minimum Spanning Trees



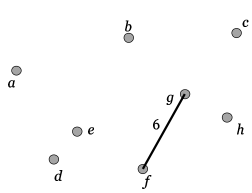
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



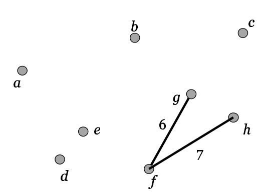
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



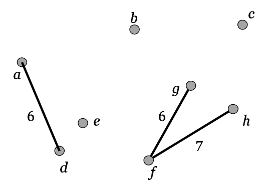
edge	cost
→ (f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



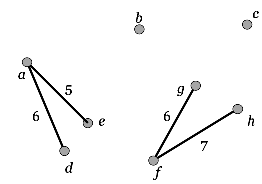
edge	cost
(f,g)	6
→ (f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



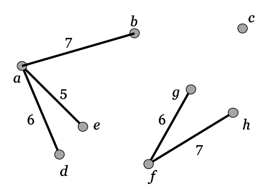
edge	cost
(f,g)	6
(f,h)	7
→ (a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



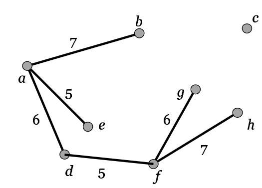
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
→ (a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



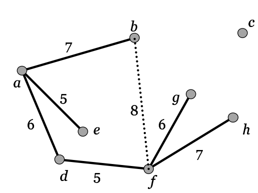
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
→ (a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



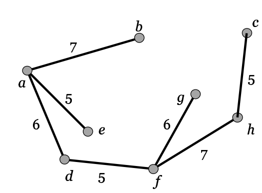
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
→ (d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



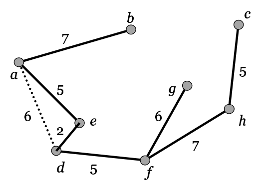
edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
→ (b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
→ (c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

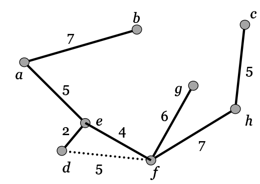
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
→ (d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

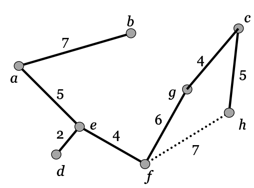
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
→ (e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

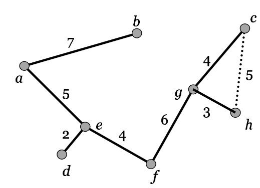
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
→ (c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

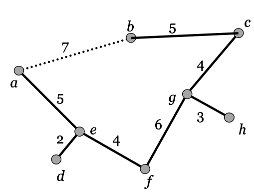
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
→ (g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

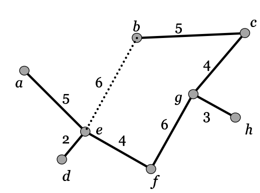
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
→ (b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

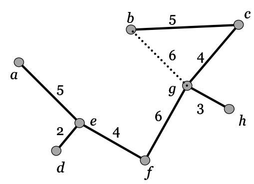
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
→ (b,e)	6
(b,g)	6

Dynamic Trees

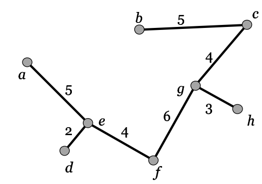
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
→ (b,g)	6

Dynamic Trees

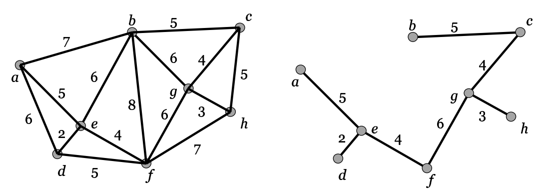
Online Minimum Spanning Trees



edge	cost
(f,g)	6
(f,h)	7
(a,d)	6
(a,e)	5
(a,b)	7
(d,f)	5
(b,f)	8
(c,h)	5
(d,e)	2
(e,f)	4
(c,g)	4
(g,h)	3
(b,c)	5
(b,e)	6
(b,g)	6

Dynamic Trees

Online Minimum Spanning Trees



- How fast is the algorithm?
  - How fast can we find the most expensive edge in a cycle?
    - O(log *n*), with the right data structure.
  - Total running time: O(*m* log *n*) (*m* edges, *n* vertices)

Dynamic Trees

Dynamic Trees

- Motivation (online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Dynamic Trees

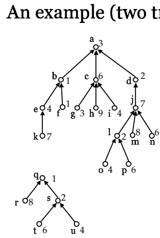
Dynamic Trees - Problem Definition

- 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*): deletes edge (*v*, *p*(*v*)) (where *p*(*v*) is its parent).
  - 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 in the path from *v* to the root.
- A possible extension:
  - evert(*w*): makes *w* the root of its tree

Dynamic Trees

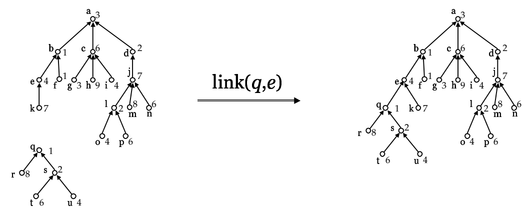
Dynamic Trees

- An example (two trees):

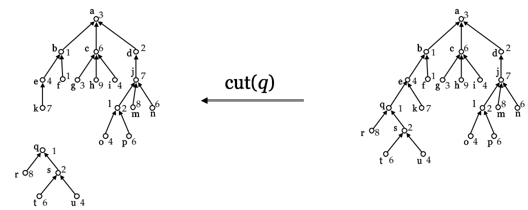


Dynamic Trees

Dynamic Trees

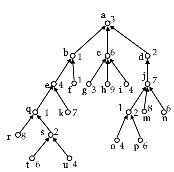


Dynamic Trees



Dynamic Trees

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

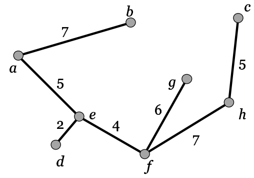


Applications

- Used as a building block of several graph algorithms:
  - online minimum spanning trees
  - dynamic graphs
  - directed minimum spanning trees
  - network flows (e.g., maximum flow)
  - ...

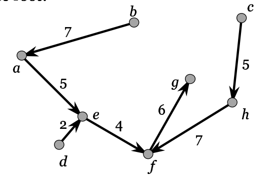
Dynamic Trees and Online MSTs

- How can dynamic trees help us in the online MST problem?
  - We must answer the following (equivalent) questions:
    - Should we insert (c,g), with cost 4, into the following tree?
    - Is (c,g) cheaper than some other edge in the cycle it creates?
    - What is the most expensive edge in the path between c and g?



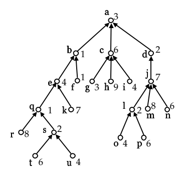
Dynamic Trees and Online MST

- How can dynamic trees help us in the online MST problem?
  - We must answer the following (equivalent) questions:
    - Should we insert (c,g), with cost 4, into the following tree?
    - Is (c,g) cheaper than some other edge in the cycle it creates?
    - What is the most expensive edge in the path between c and g?
    - Imagine the tree is rooted at g: now, what is the most expensive edge in the path from c to the root?



Obvious Implementation of Dynamic Trees

- Each node represents a vertex.
- Each node x points to its parent $p(x)$:
  - cut, link, findcost: constant time.
  - findroot, findmin: time proportional to path length.
- Acceptable if paths are small, but O(n) in the worst case.
- We can get O(log n) for all operations.

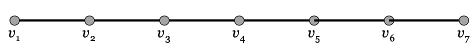


Dynamic Trees

- Motivation (online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

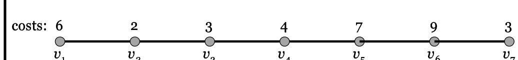
Dynamic Paths

- We start with a simpler problem:
  - Maintain set of paths subject to the following operation:
    - split: removes an edge, cutting a path in two;
    - concatenate: links endpoints of two paths, creating a new path.
  - Operations allowed:
    - findcost(v): returns the cost of vertex v;
    - findmin(v): returns minimum-cost vertex in the path containing v.



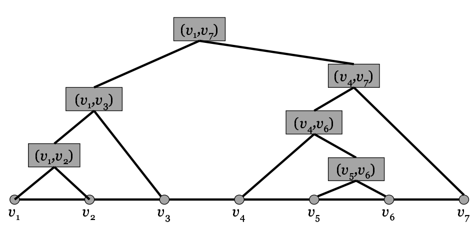
Simple Paths as Lists

- Natural representation: doubly-linked list:
  - Path characterized by two endpoints.
    - findcost: constant time.
    - concatenate: constant time.
    - split: constant time.
    - findmin: linear time (not good).
- Can we do it O(log n) time?



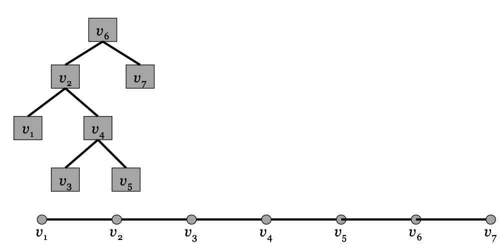
Simple Paths as Binary Trees

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



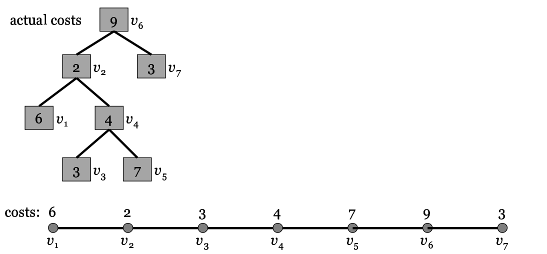
Simple Paths as Binary Trees

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



Simple Paths: Maintaining Costs

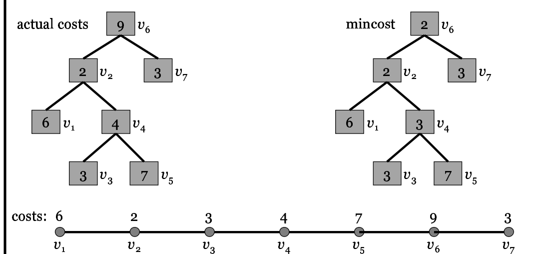
- We store cost(x) directly on each vertex.
  - Problem: findmin still takes linear time (must visit all vertices).



Dynamic Trees

Simple Paths: Finding Minima

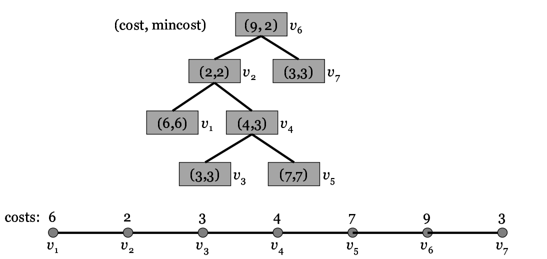
- Also store mincost(x), minimum cost in subpath with root x.
  - findmin(x) now runs in O(log n) time.



Dynamic Trees

Simple Paths: Data Fields

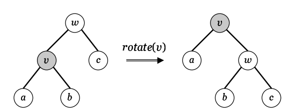
- Final version:
  - Stores mincost(x) and cost(x) for every vertex x.



Dynamic Trees

Simple Paths: Structural Changes

- Concatenating and splitting paths:
  - Join or split the corresponding binary trees;
  - Time proportional to tree height.
  - For balanced trees (AVL, red-black, etc.), this is O(log n):
    - Rotations must be supported in constant time.
    - We must be able to update mincost, but that's easy:



$mincost'(w)$ = min {$cost(w)$, $mincost(b)$, $mincost(c)$}
$mincost'(v)$ = min {$cost(v)$, $mincost(a)$, $mincost'(w)$}

Dynamic Trees

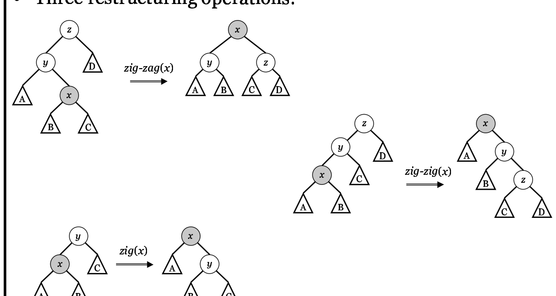
Splaying

- 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:
    - Moves v to the root;
    - Roughly halves the depth of other nodes in the access path.
  - Primitive operation: rotation.
- All operations (insert, delete, join, split) use splaying.

Dynamic Trees

Splaying

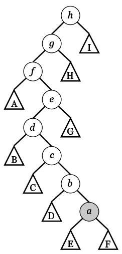
- Three restructuring operations:



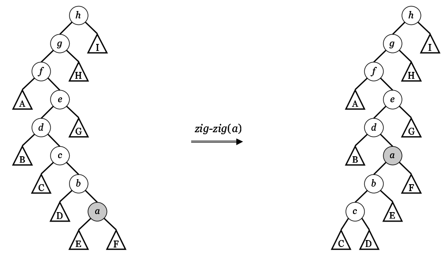
(only happens if y is the original root)

Dynamic Trees

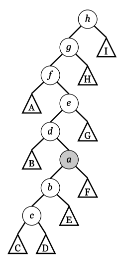
An Example of Splaying



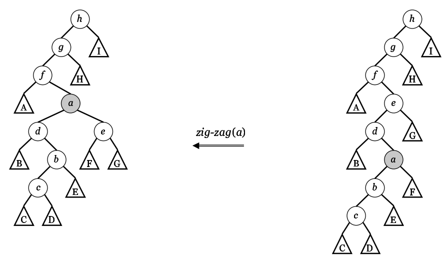
An Example of Splaying



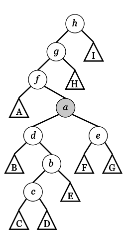
An Example of Splaying



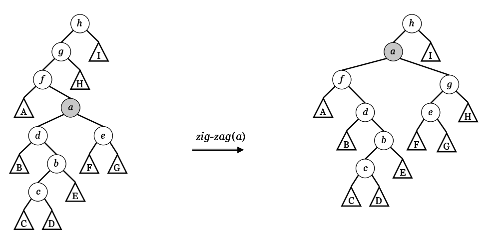
An Example of Splaying



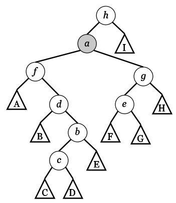
An Example of Splaying



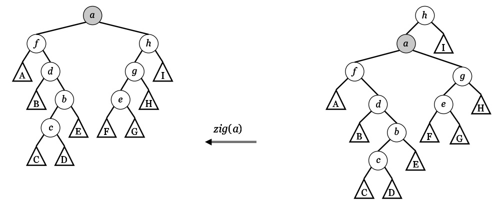
An Example of Splaying



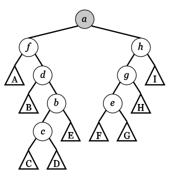
An Example of Splaying



An Example of Splaying

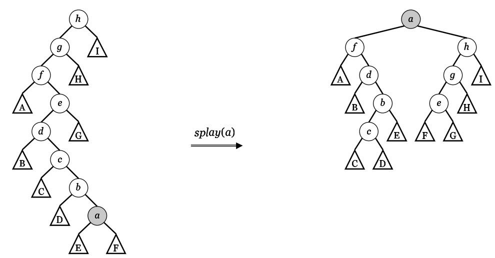


An Example of Splaying



An Example of Splaying

- Final result:



Amortized Analysis

- Bounds the running time of a sequence of operations.
- Potential function Φ maps configurations to real numbers.
- 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 $n \log n$, 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*

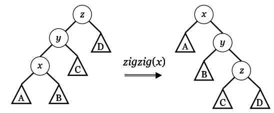
  $$6(r(t)-r(x)) + 1 = O(\log(s(t)/s(x))).$$

- 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:

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

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 $r'(x) \ge r'(y)$

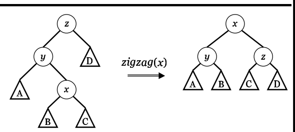
$(r(x) - r'(x)) + (r'(z) - r'(x)) \le -1$ rearranging

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

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

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 $r(y) \ge r(x)$

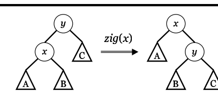
$(r'(y) - r'(x)) + (r'(z) - r'(x)) \le -1$, rearranging

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

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

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

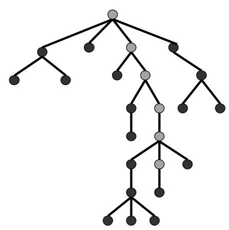
- Summing up:
  - No rotation: $a = 1$
  - Zig: $a \le 6\,(r'(x) - r(x)) + 1$
  - Zig-zig: $a \le 6\,(r'(x) - r(x))$
  - Zig-zag: $a \le 4\,(r'(x) - r(x))$
  - Total amortized time at most $6\,(r(t) - r(x)) + 1 = O(\log n)$
- Since accesses bring the relevant element to the root, other operations (insert,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

Dynamic Trees

- Motivation (online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

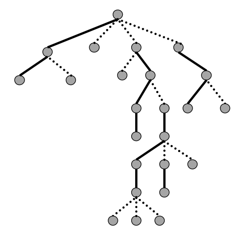
Dynamic Trees

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



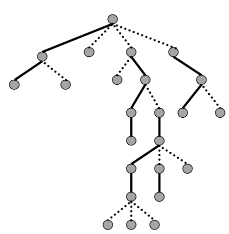
Dynamic Trees

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



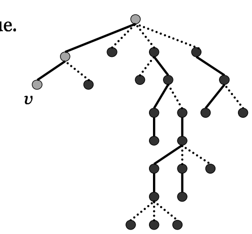
Dynamic Trees

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



Dynamic Trees

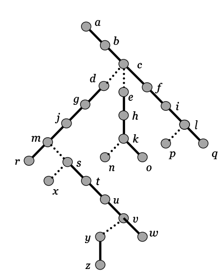
- A vertex v is exposed if:
  - There is a solid path from v to the root;
  - No solid edge enters v.
- It is unique.



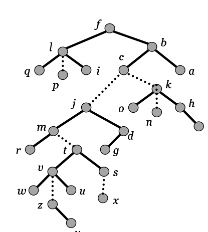
Dynamic Trees

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

Virtual Tree: An Example



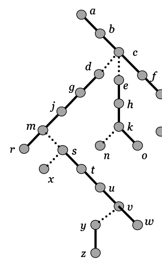
the actual tree



a virtual tree

Dynamic Trees

- Example: expose(y)

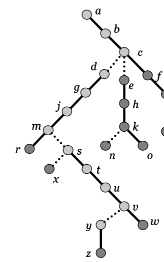


(actual tree)

Dynamic Trees

Dynamic Trees

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

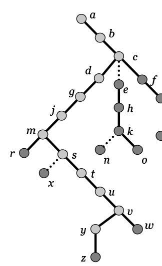


(actual tree)

Dynamic Trees

Dynamic Trees

- Example: expose(y)
  - …, make them solid, …

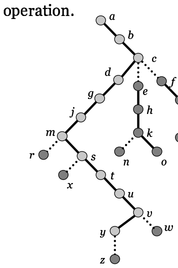


(actual tree)

Dynamic Trees

Dynamic Trees

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



(actual tree)

Dynamic Trees

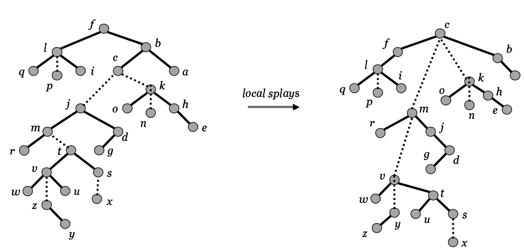
Exposing a Vertex

- expose(y): makes the path from y 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 replaces left solid child with dashed child;
  3. Splay on x, which will become the root.

Dynamic Trees

Exposing a Vertex: An Example

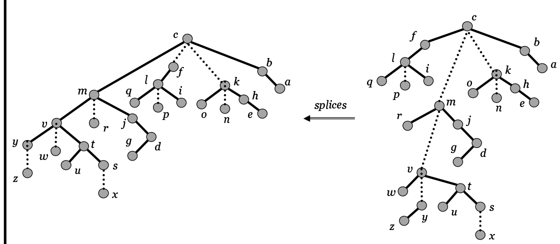
- expose(y): (1) splay within each solid tree;
  - Does not change the partition into solid paths.



Dynamic Trees

Exposing a Vertex: An Example

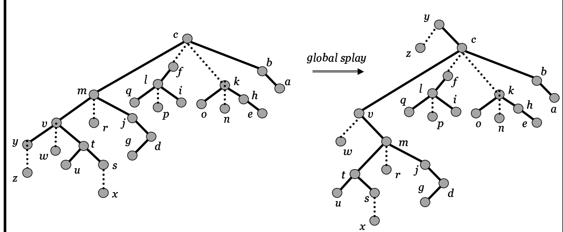
- expose(y): (2) splice on all vertices from y to the root.
  - Original exposed path: ($q\ l\ i\ f\ c\ b\ a$)
  - New exposed path: ($y\ v\ u\ t\ s\ m\ j\ g\ d\ c\ b\ a$)



Dynamic Trees

Exposing a Vertex: An Example

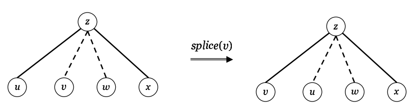
- expose(y): (3) splay on y.
  - Does not change the exposed path.



Dynamic Trees

Dynamic Trees: Splice

- Additional restructuring primitive: *splice*.
  - Dashed child becomes solid, replaces left child.



  - Update: $mincost'(z) = \min\{cost(z), mincost(v), mincost(x)\}$

Dynamic Trees

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 use 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.

Dynamic Trees

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: $6\,(r'(x_i) - r(x_i)) + 1$ (Access Lemma)
     - $r(x_{i+1}) \ge r'(x_i)$, so the sum over all steps telescopes;
     - amortized cost first of pass: $6(r'(x_k) - r(x_1)) + k \le 6 \log n + k$.
  2. Splice dashed edges:
     - no rotations, no changes in potential: 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 = $O(\log n)$.

Dynamic Trees

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 *mincost* to walk down from v to w, the last minimum-cost node in the solid subtree containing v;
  - splay at w and return w.

Dynamic Trees

Implementing Dynamic Tree Operations

- 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;
  - make $p(right(v))$=**null** and $right(v)$=**null**;
  - set $mincost(v) = \min\{cost(v), mincost(left(v))\}$.

Alternative Implementations

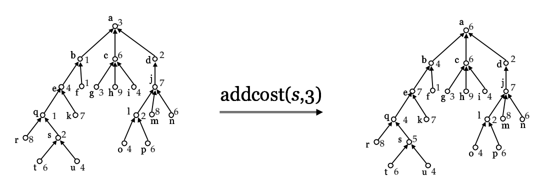
- Total time per operation depends on path representation:
  - Splay trees: O(log n) amortized [Sleator and Tarjan, 85].
  - Balanced search tree: O($\log^2 n$) amortized [ST83].
  - Locally biased search tree: O(log n) amortized [ST83].
  - Globally biased search trees: O(log n) worst-case [ST83].
- Biased search trees:
  - Support leaves with different "weights".
  - Some solid leaves are "heavier" because they also represent dashed subtrees.
  - Much more complicated than splay trees.

Dynamic Trees

- Motivation (online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

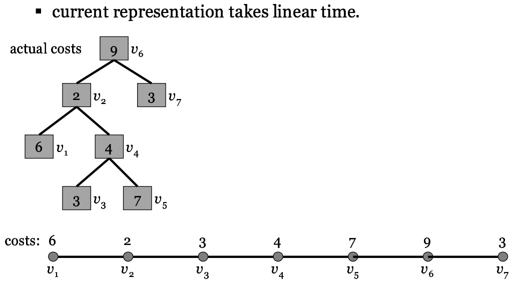
Extension: Adding Costs

- addcost(v,x): adds x to the cost of all vertices in the path from v to the root.



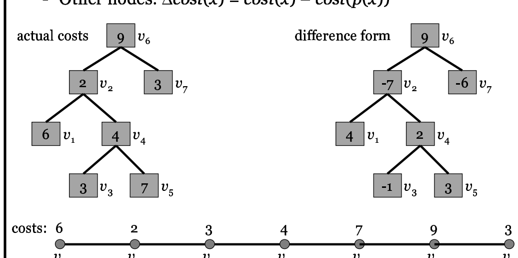
Adding Costs to Dynamic Paths

- Corresponding operation on dynamic paths:
  - addcost(v,x): adds x to the cost of vertices in path containing v;
  - current representation takes linear time.



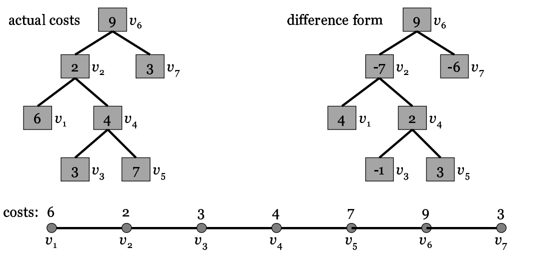
Adding Costs to Dynamic Paths

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



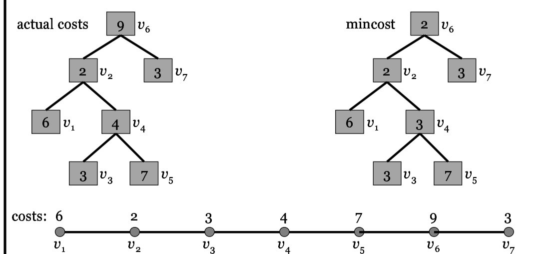
Adding Costs to Dynamic Paths

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



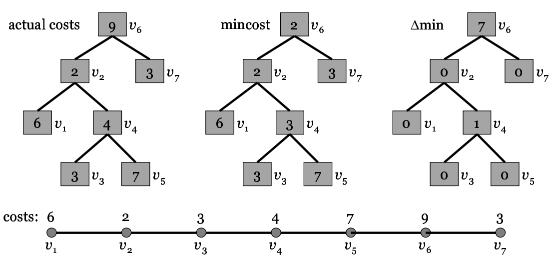
Adding Costs to Dynamic Paths

- Still have to implement findmin:
  - Cannot store mincost(x) explicitly (addcost would be linear).



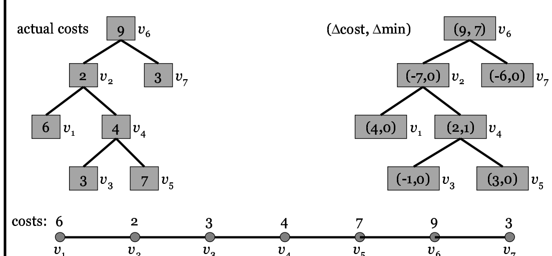
Adding Costs to Dynamic Paths

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



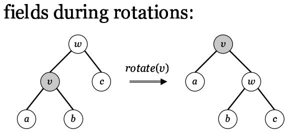
Adding Costs to Dynamic Paths

- Final version:
  - Store Δmin(x) and Δcost(x) on each node.



Adding Costs to Dynamic Paths: Updating Fields

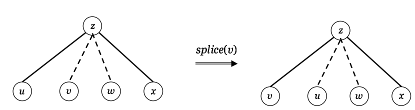
- Updating fields during rotations:



  - $\Delta cost'(v) = \Delta cost(v) + \Delta cost(w)$
  - $\Delta cost'(w) = -\Delta cost(v)$
  - $\Delta cost'(b) = \Delta cost(v) + \Delta cost(b)$
  - $\Delta min'(w) = \max\{0, \Delta min(b) - \Delta cost'(b), \Delta min(c) - \Delta cost(c)\}$
  - $\Delta min'(v) = \max\{0, \Delta min(a) - \Delta cost(a), \Delta min'(w) - \Delta cost'(w)\}$

Adding Costs: Updating Fields

- Updating fields during *splice*:



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

- Recall that w is always the root of a solid tree.

Adding Costs: Operations

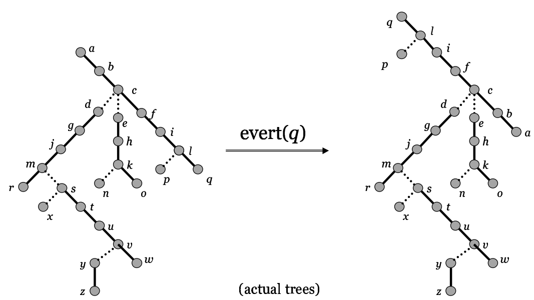
- findcost(v):
  - expose v, return $\Delta cost(v)$.
- findroot(v):
  - expose v;
  - find w, the rightmost vertex in the solid subtree containing v;
  - splay at w and return w.
- findmin(v):
  - expose v;
  - use $\Delta cost$ and Δmin to walk down from v to w, the last minimum-cost node in the solid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

Adding Costs: Operations

- addcost(v, x):
  - expose v;
  - add x to $\Delta cost(v)$, subtract x from $\Delta cost(left(v))$
- 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**.
  - set $\Delta min(v) = \max\{0, \Delta min(left(v)) - \Delta cost(left(v))\}$

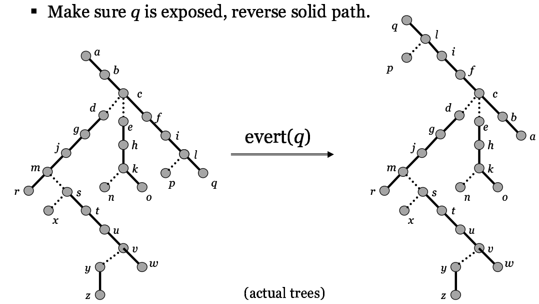
Another Extension: Change Root

- evert(q): makes q the root of its tree



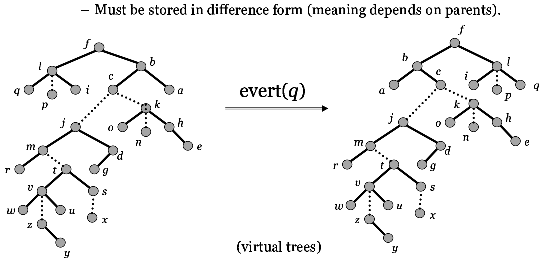
Another Extension: Change Root

- evert(q): makes q the root of its tree
  - Make sure q is exposed, reverse solid path.



Another Extension: Change Root

- evert(q): makes q the root of its tree
  - In the virtual tree: reverse left-right pointers:
    - This can be done implicitly with a *reverse* bit.
      - Must be stored in difference form (meaning depends on parents).



Other Extensions

- Associate values with edges:
  - just interpret cost(v) as cost(v,$p(v)$).
- Other path queries (such as length):
  - modify values stored in each node appropriately.
- Free (unrooted) trees: use evert to change root.
- Subtree-related operations:
  - Can be implemented, but parent must have access to middle children in constant time:
    - Tree must have bounded degree.
  - Approach for arbitrary trees: "ternarize" them:
    - [Goldberg, Grigoriadis and Tarjan, 1991]