COS 423 Lecture 12
Disjoint Sets and Compressed Trees

Three problems

Maintenance of disjoint sets under union

Finding nearest common ancestors in a rooted tree

Finding maxima on tree paths

Disjoint set union

Devise a data structure for an intermixed sequence of the following kinds of operations:

make-set(x) (x in no set): create a set $\{x\}$, with *root* x.

find(x): (x in a set): return the root of the set containing x.

link(x, y) ($x \neq y$): combine the sets whose roots are x and y into a single set; choose x or y as the root of the new set.

Each element is in at most one set (sets are *disjoint*).

The root of a set serves to identify it, can store information about the set (size, name, etc.)

Applications

Global greedy MST algorithm

FORTRAN compilers: COMMON and EQUIVALENCE statements

Incremental connected components

Percolation

Additional operations

unite(x, y) (*find*(x) $\neq$ *find*(y)): *link*(*find*(x), *find*(y))

contingent-unite(x, y):

    if *find*(x) = *find*(y) **then return** false

        else {*link*(*find*(x), *find*(y)); **return** true}

make-set, *contingent-unite* suffice to implement global greedy MST algorithm

Variant: named sets

make-set(x, g): create a set $\{x\}$, named g, with root x

find-name(x): return the name of the set containing x

unite(x, y, g) (*find*(x) $\neq$ *find*(y)): combine the sets containing x and y; name the new set g

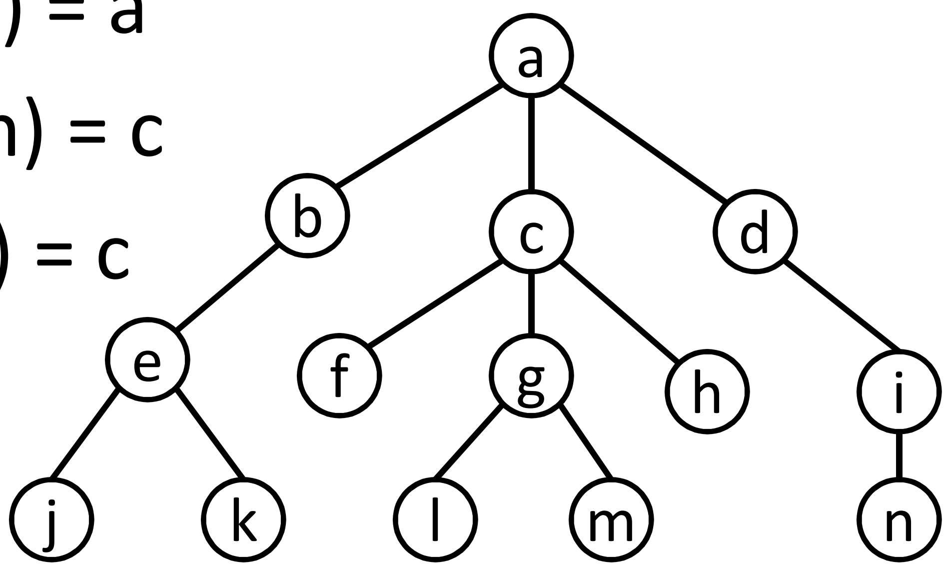
Nearest common ancestors

Given a rooted tree T and a set Q of pairs of vertices (x, y), find the *nearest common ancestor* $nca(x, y)$ of each pair.

$nca(\text{e},\text{h}) = \text{a}$

$nca(\text{f}, \text{m}) = \text{c}$

$nca(\text{c}, \text{l}) = \text{c}$



Maxima on tree paths

Given a tree T with edge weights and a set Q of vertex pairs (x, y), find the maximum weight of an edge on $T(x, y)$ for each pair.

Disjoint set implementation

Represent each set by a rooted tree, whose nodes are the elements of the set, with the set root the tree root, and each node x having a pointer to its parent $a(x)$. Store information about set (such as name) in root.

The shape of the tree is *arbitrary*.

n = #links, m = #finds, assume $n = O(m)$

Set operations

make-set(x): make x the root of a new one-node tree: $a(x) \leftarrow$ null

find(x): follow parent pointers from x to the tree root: **if** $a(x)$ = null **then return** x

else return *find*($a(x)$)

link(x, y): make y the parent of x (or x the parent of y): $a(x) \leftarrow y$ (*or* $a(y) \leftarrow x$)

A bad sequence of links can create a tree that is a path of n nodes, on which each *find* can take $\Omega(n)$ time, totaling $\Omega(mn)$ time for *m finds*

Goal: reduce the amortized time per *find*: reduce node depths

Improve links: linking by *size* or by *rank*

Improve finds: *compress* the trees

Linking by size: maintain the number of nodes in each tree (store in root). Link root of smaller tree to larger. Break a tie arbitrarily.

make-set(x): $\{a(x) \leftarrow x;\ s(x) \leftarrow 1\}$

link(x, y):

if $s(x) < s(y)$ **then** $\{a(x) \leftarrow y;\ s(y) \leftarrow s(y) + s(x)\}$

else $\{a(y) \leftarrow x;\ s(x) \leftarrow s(x) + s(y)\}$

Linking by rank: Maintain an integer *rank* for each root, initially 0. Link root of smaller rank to root of larger rank. If tie, increase rank of new root by 1.

$\mathit{make\text{-}set}(x)$: $\{a(x) \leftarrow x;\ r(x) \leftarrow 0\}$

$\mathit{link}(x, y)$: $\{\textbf{if } r(x) = r(y) \textbf{ then } r(y) \leftarrow r(y) + 1;$

$\quad\quad\quad\quad \textbf{if } r(x) < r(y) \textbf{ then } a(x) \leftarrow y \textbf{ else } a(y) \leftarrow x\}$

$r(x) = h(x)$, the height of x

Linking by size and linking by rank have similar efficiency. Linking by rank needs fewer bits (lglg$n$ for rank vs. lg$n$ for size) and less time: use linking by rank

$$\text{For any } x,\ r(a(x)) > r(x)$$

Proof: Immediate.

$$\#\text{nodes of rank} \geq k \leq n/2^k$$

Proof: Only roots increase in rank. Production of one root of rank $k + 1$ consumes two roots of rank k.

$$\rightarrow r(x) \leq \lg n,\ find(x) \text{ takes } O(\lg n) \text{ time}$$

Compression

$$\mathit{compress}(x)\ (a(a(x)) \neq \text{null}){:}\ a(x) \leftarrow a(a(x))$$

Reduces depth of x, reducing find time for x and increasing no find time; preserves sets

Compression preserves $r(a(x)) > r(x)$

With compression, $r(x) \geq h(x)$, but not necessarily equal

Collapsing (fast find)

After a link that makes x a child of y, compress each child of x (make the grandchildren of y children of y). Each tree is *flat*: each node is a root or a child of a root → *find* takes O(1) time. To implement: for each tree, maintain a circular linked list of its nodes; during a link, catenate lists

Collapsing: total time is O(n^2 + m): each node changes parent ≤n times

Collapsing with union by rank: total time is O($n$lg$n$ + m): each node changes parent ≤lg$n$ times

But collapsing uses one extra pointer per node, dominated by path compression (next) no matter how links are done

Collapsing is too eager: better to do compressions only on *find* paths

Path compression

During each find, make the root the parent of each node on the find path, by doing compression top-down along the find path

find(x): **if** $a(x)$ = null **then return** x

else {**if** $a(a(x)) \neq$ null **then** $a(x) \leftarrow find(a(x))$;

return $a(x)$}

Alternative implementations

Since parent of root is null, can use parent field of root to store rank (or size), but violates type, bad programming practice

Another use of parent of root: set equal to root instead of null. Saves one test in *find* loop, but does an unneeded assignment

make-set(x): $\{a(x) \leftarrow x;\ r(x) \leftarrow 0\}$

find(x): $\{$**if** $a(a(x)) \neq a(x)$ **then** $a(x) \leftarrow a(a(x))$;

return $a(x)\}$

Collapsing vs. path compression

For any sequence of operations and any linking rule, path compression changes no more parents than collapsing: path compression dominates

Proof: Compare three scenarios: (i) no compression, (ii) collapsing, (iii) path compression. If in (iii) w becomes a parent of v, then in (i) w becomes a proper ancestor of v, and in (ii) w becomes a parent of v.

Nearest common ancestors

Depth-first traversal using named sets

Do a depth-first traversal of the tree *T*. For each vertex *x* visited in preorder, maintain a set named *x*, containing *x* and all descendants of *x* so far visited in postorder. If (*x*, *y*) is a query pair with *x* visited second in preorder, *nca*(*x*, *y*) is the name of the set containing *y* when *x* is visited in preorder.

Implementation

$C(x)$ = children of x, $Q(x)$ = query pairs (x, y),

t = root of T

traverse(t) **where** *traverse*(x) =

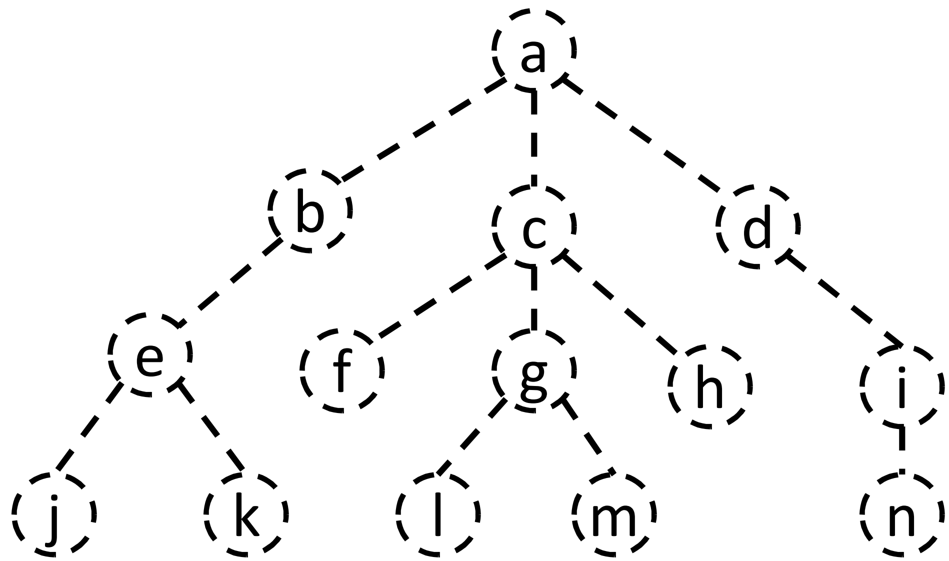
{*make-set*(x, x);

for $(x, y) \in Q(x)$ **do**

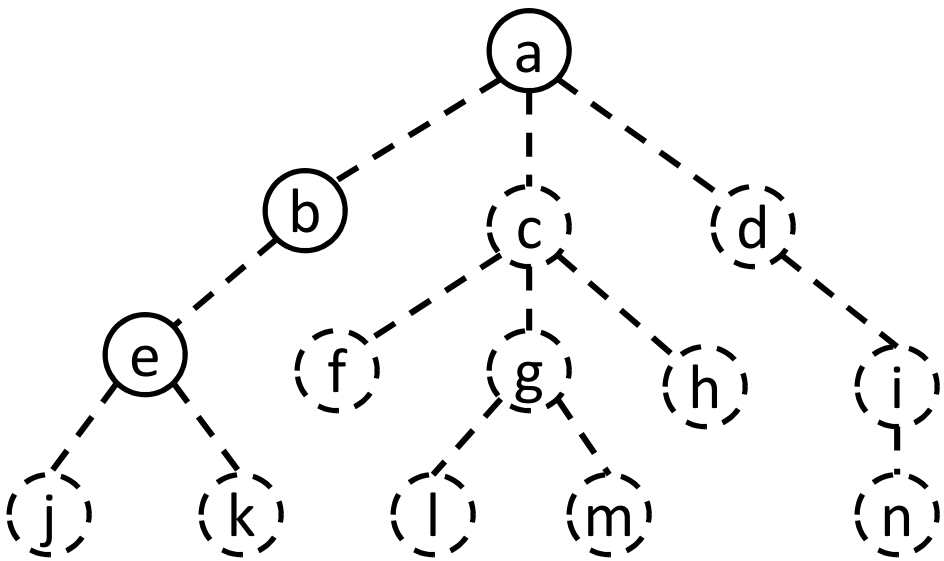
if y in a set **then** *nca*$(x, y) \leftarrow$ *find-name*(y)

for $y \in C(x)$ **do** {*traverse*(y); *unite*(y, x, x)}}

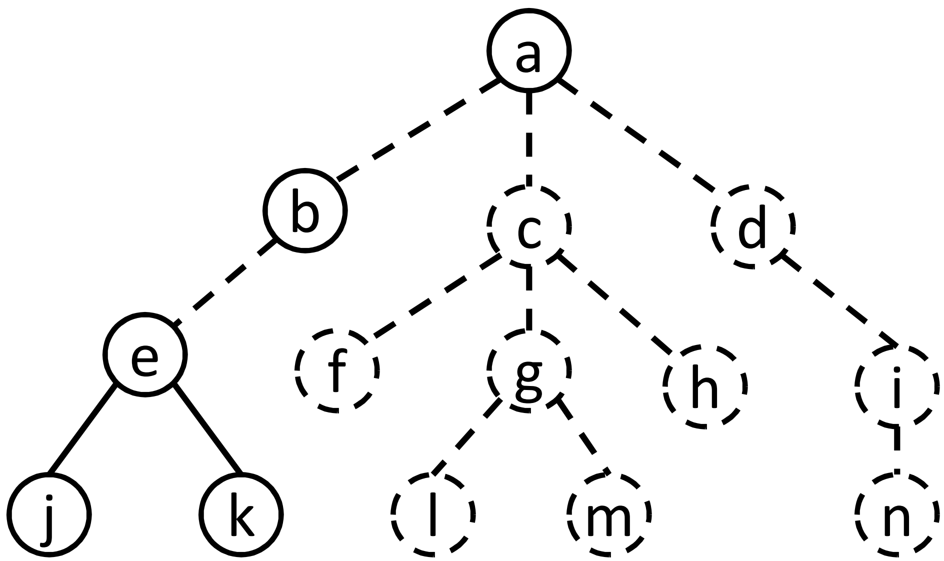
$Q = \{(e, h), (f, m), (c, l)\}$



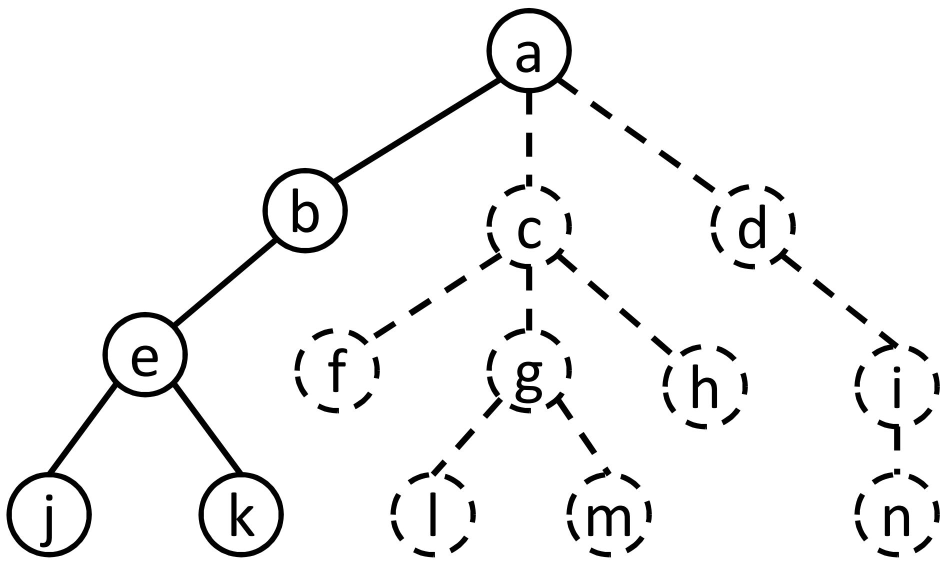
$Q = \{(e, h), (f, m), (c, l)\}$



$$Q = \{(e, h), (f, m), (c, l)\}$$

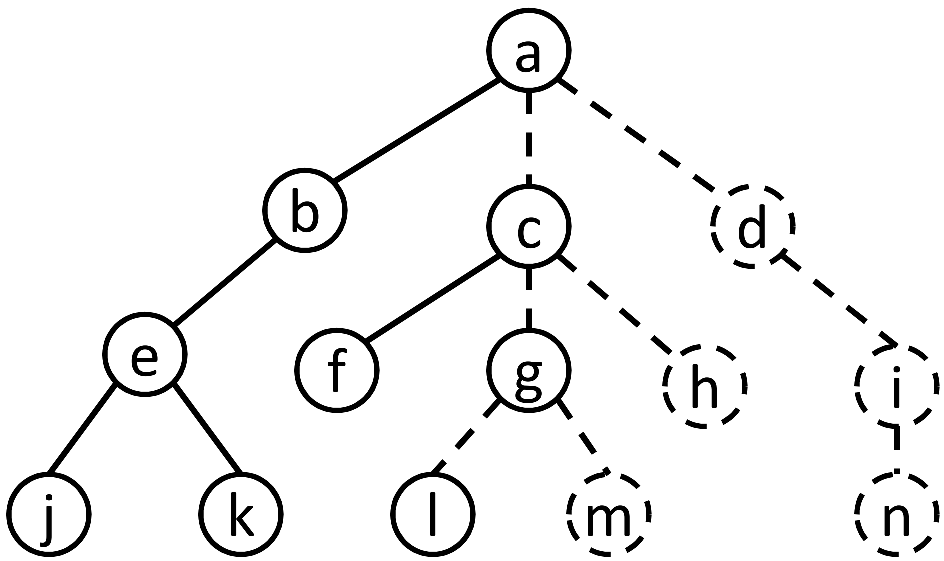


$Q = \{(e, h), (f, m), (c, l)\}$



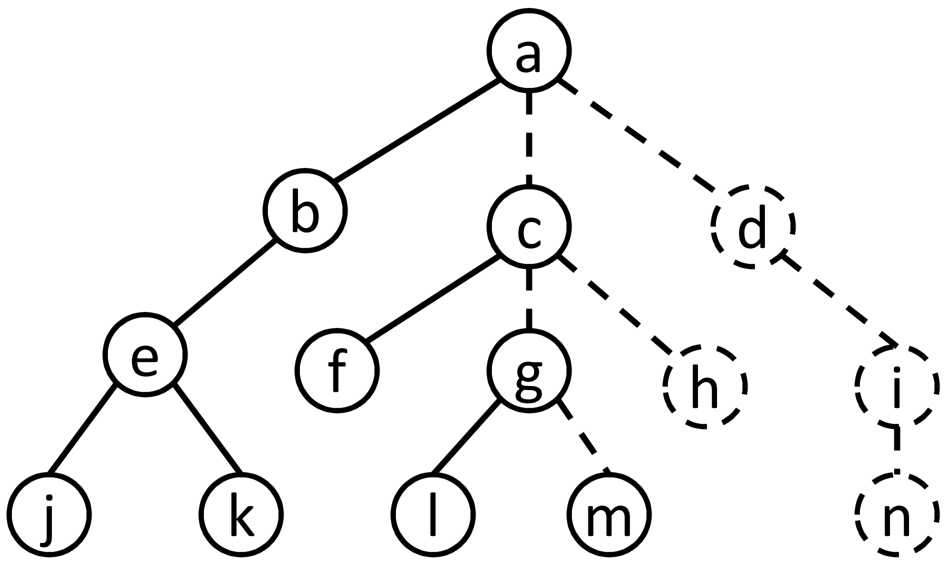
Q = {(e, h), (f, m), (c, l)}

nca(c, l) = *find-name*(c) = c



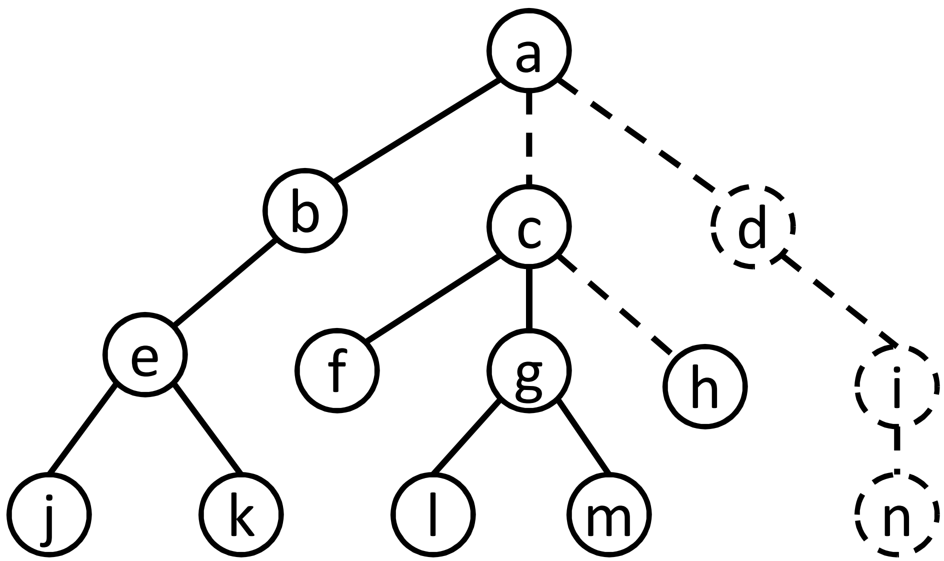
$Q = \{(\mathrm{e}, \mathrm{h}), (\mathrm{f}, \mathrm{m}), (\mathrm{c}, \mathrm{l})\}$

$\mathit{nca}(\mathrm{f}, \mathrm{m}) = \mathit{find\text{-}name}(\mathrm{f}) = \mathrm{c}$



$Q = \{(\text{e}, \text{h}), (\text{f}, \text{m}), (\text{c}, \text{l})\}$

$nca(\text{e}, \text{h}) = \textit{find-name}(\text{e}) = \text{a}$



Correctness of *nca* algorithm

Let (x, y) be a query pair, z = *nca*(x, y). Suppose x is visited in preorder after y. All ancestors of y that are proper descendants of z have been visited in postorder by the time x is visited in preorder, so they are all in the same set as z. In particular, x is in the same set as z. When x is visited in preorder, z has not yet been visited in postorder, so *find-name*(y) = z.

Maxima on tree paths

Let T be a tree with edge weights. Build the corresponding Borůvka tree B. Associate the weight of each edge $(v, p(v))$ in B with child node v: $c(v) = c(v, p(v))$. Build a compressed copy of B during a depth-first traversal. Find path maxima using compression steps that update node weights:

c-compress(v) $(a(a(v)) \neq a(v))$:

$\{c(v) \leftarrow \max\{c(v), c(a(v))\}; a(v) \leftarrow a(a(v))\}$

Let (x, y) be a query pair and $z = nca(x, y)$

Then $B(x, y) = B(x, z)$ & $B(z, y)$ where "&" is catenation of paths $\rightarrow$ max on $B(x, y) =$ max{max on $B(x, z)$, max on $B(z, y)$}

max on B(x, y) is unaffected by $compress(v)$ unless $z = a(v)$ before the compression

If path maxima are found in proper order, can use path compression to help find them

Path compression with weight updates

c-find(x):

if $a(x)$ = null **then return** x

else {**if** $a(a(x)) \neq$ null **then**

{$c(x) \leftarrow \max\{c(x), c(a(x))$; $a(x) \leftarrow$ *c-find*($a(x)$)};

return $a(x)$}

Path max algorithm is similar to *nca* algorithm, but does naïve linking and one or two finds per query, computing *path-max*(x, y) during postorder visit to *nca*(x, y)

$C(x)$ = children of x in B

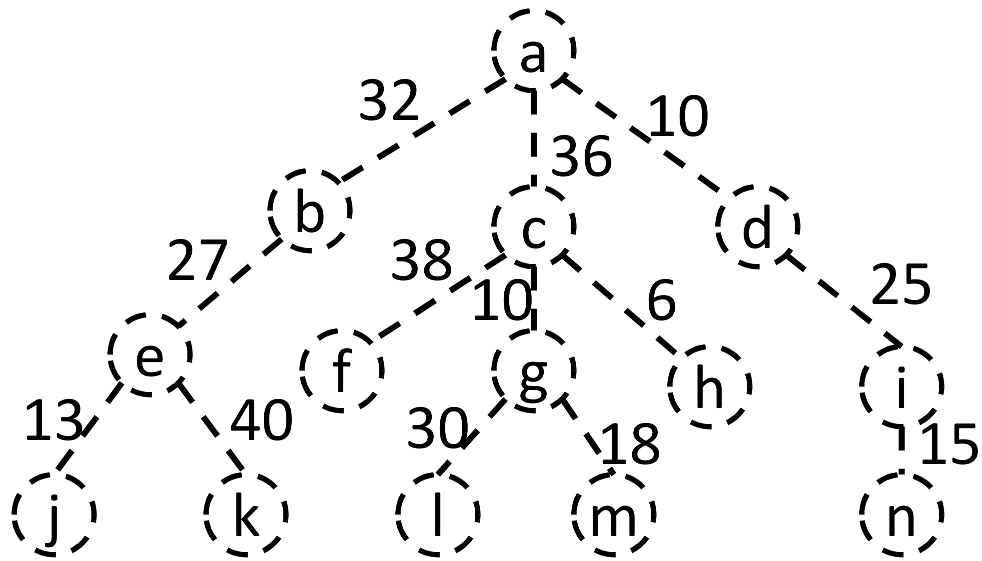
$Q(x)$ = query pairs (x, y)

t = root of T

$S(z)$ = query pairs (x, y) such that $z = nca(x, y)$, computed by the algorithm

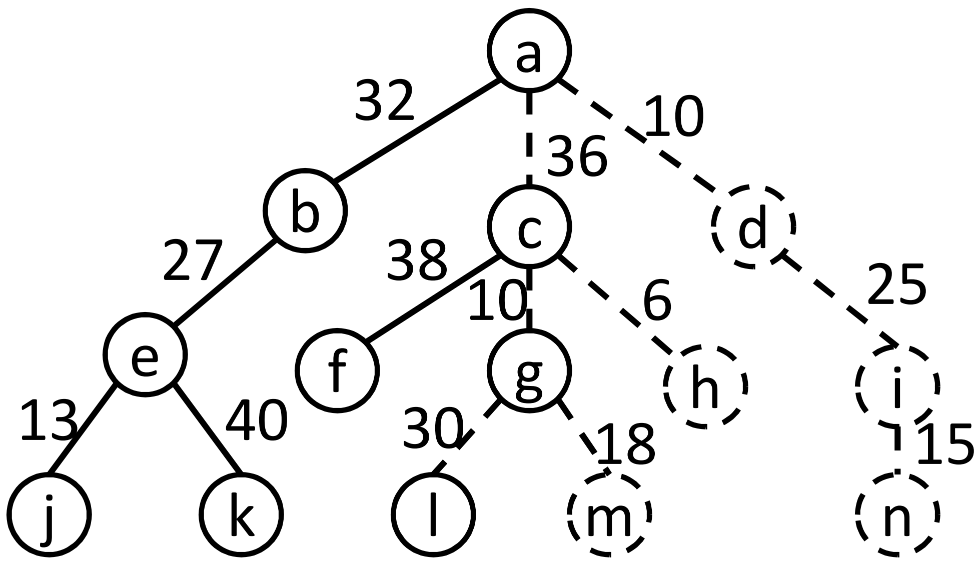
```
traverse(t) where traverse(x) =
  {make-set(x); S(x) ← { };
   for (x, y) ∈ Q(x) do
     if y in a set then add (x, y) to S(c-find(y));
   for y ∈ C(x) do {traverse(y); a(y) ← x}
   for (v, w) in S(x) do
     {z ← c-find(v);
      if w = x then path-max(v, w) ← c(v)
        else path-max(v, w) ← max{c(v), c(w)}}
```

$Q = \{(e, h), (f, m), (c, l)\}$



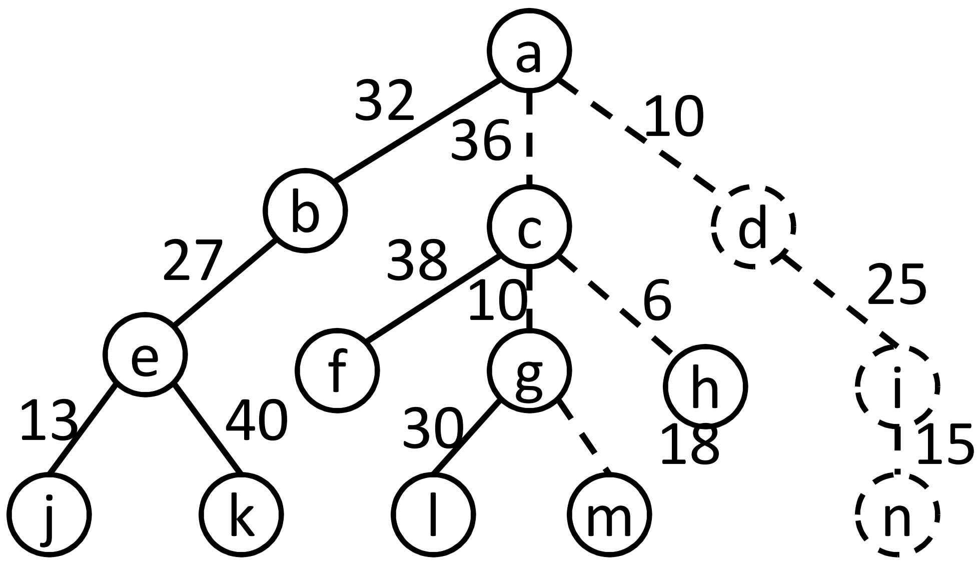
Q = {(e, h), (f, m), (c, l)}

S(c) = {(c, l)}



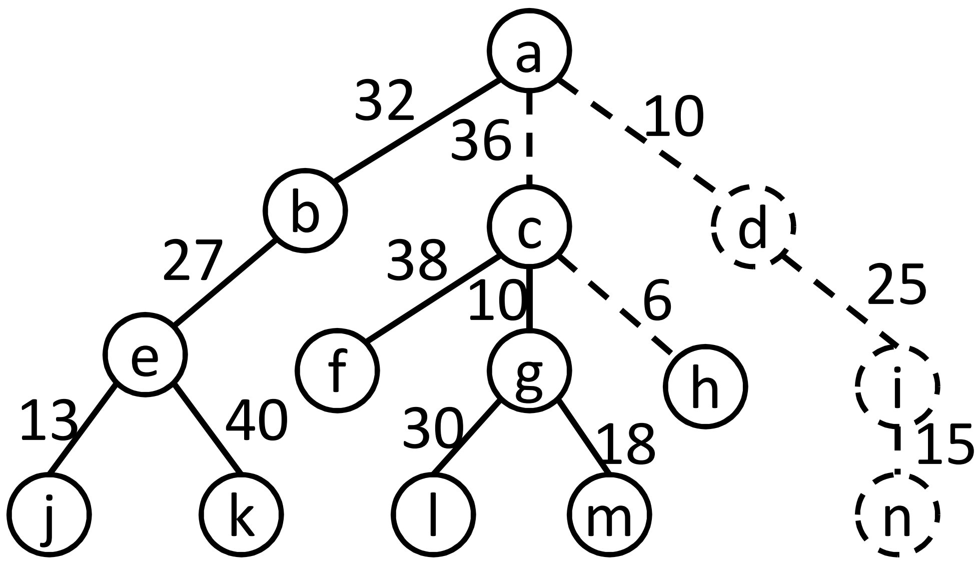
$Q = \{(e, h), (f, m), (c, l)\}$

$S(c) = \{(c, l), (f, m)\}$



$Q = \{(e, h), (f, m), (c, l)\}$

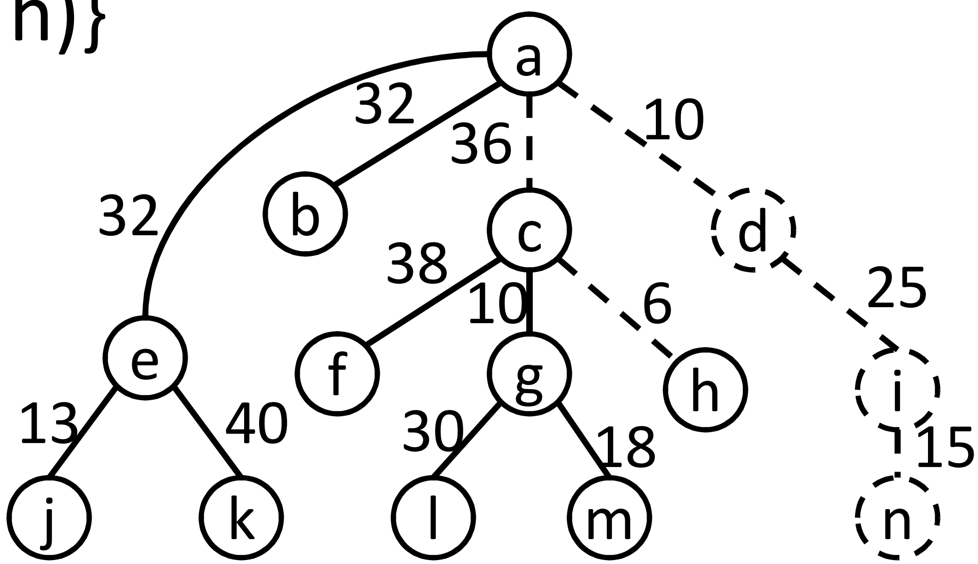
$S(c) = \{(c, l), (f, m)\}$



Q = {(e, h), (f, m), (c, l)}

S(c) = {(c, l), (f, m)}

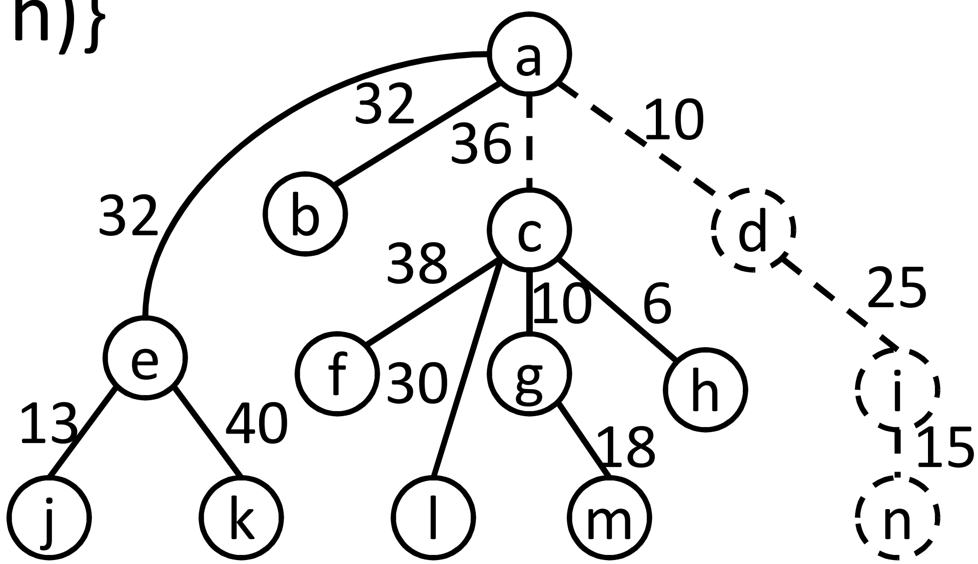
S(a) = {(e, h)}



$Q = \{(\text{e}, \text{h}), (\text{f}, \text{m}), (\text{c}, \text{l})\}$

$S(\text{c}) = \{(\text{c}, \text{l}), (\text{f}, \text{m})\}$

$S(\text{a}) = \{(\text{e}, \text{h})\}$

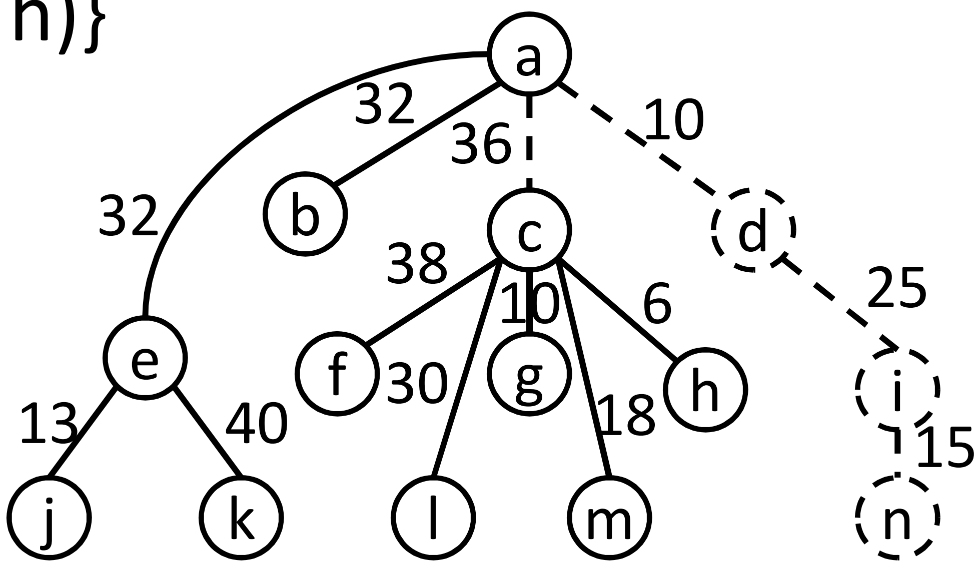


path-max(c, l) =30

Q = {(e, h), (f, m), (c, l)}

S(c) = {(c, l), (f, m)}

S(a) = {(e, h)}

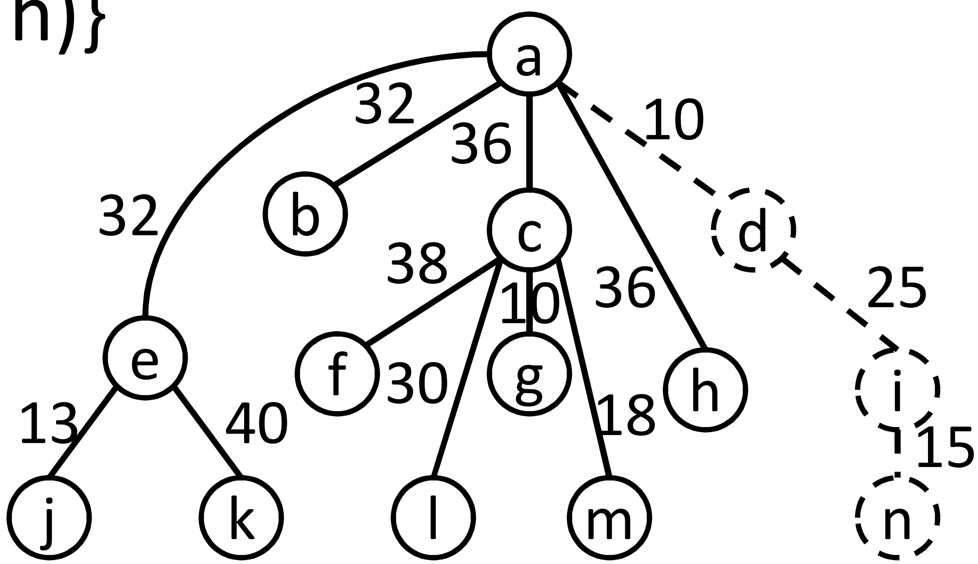


path-max(c, l) =30, path-max(f, m) = 38

Q = {(e, h), (f, m), (c, l)}

S(c) = {(c, l), (f, m)}

S(a) = {(e, h)}



path-max(c, l) =30, path-max(f, m) = 38,

path-max(e, h) = 36

Correctness: If x is a vertex visited in preorder but not yet in postorder, then x is the root of a set containing all its descendants that have been visited in postorder. When x is visited in postorder, it is the root of a set containing all its descendants. Let (x, y) be a query pair with z = nca(x, y) and y visited first in preorder. Then z = *c-find*(y) when x is visited in preorder, because the *nca* algorithm is correct. Thus (x, y) is added to $S(z)$. If $y \neq z$, after *c-find*(y) $a(y) = z$ and $c(y)$ is the maximum weight of an edge on $B(y, z)$. When z is visited in postorder, after *c-find*(x), $a(x) = z$ and $c(x)$ is the maximum weight of an edge on $B(z, x)$.

Balls in the air

How efficient is path compression, with or without linking by rank?

How many comparisons needed to find path maxima?