CS 170
DISCUSSION 6

MINIMUM SPANNING TREES

Raymond Chan
UC Berkeley Fall 17

PRIM'S ALGORITHM

- Grow tree similar to Dijkstra's algorithm.
- Take lightest edge that connects existing tree to unseen vertex.
- O((|V| + |E|) log |V|) with binary heap
- O(|E| + |V| log |V|) with Fibonacci heap
- Demo

```
generic_MST(G):
  start S = v:
    find lightest edge (x, y)
      in crossing (S, V - S)
    S = S union {y}
```

```
prim (G, s):
  c[v] = infinity
c[s] = 0
prev[s] = s
PriorityQueue.add(G.V, c)
while PQ not empty:
  u = PQ.DeleteMin()
  add (u, prev(u)) to T
  for edge (u, v):
    if w(u, v) < c(v):
      c(v) = w(u, v)
      prev(v) = u
      PQ.DecreaseKey(v, c[v])
```

KRUSKAL'S ALGORITHM

- Choose lightest edge that does not form a cycle.
- O(|E| log |E| + |E| log |V|) = O(|E| log |E|) = O(|E| log |V|)
- Demo
- How do we actually achieve the above runtime?

KRUSKAL'S ALGORITHM

- At any point, we have partial solutions: sets of vertices that are connected based on the lightest edge at some previous iteration.
- In the next step, we can join these sets together (vertex by itself is a set).
- Make use of disjoint set such that joining two elements from different sets joins all elements for the vertices' respective sets.

DISJOINT SETS

- Union-Find Implementation
  - Tree with directed edges pointing towards root.
- Union-By-Rank (O(log n) time per operation)
  - Point smaller rank root to larger rank root.
- Path Compression (O(log n) amortized time)
  - On union, point all descendants of smaller rank root (inclusive) to larger rank root.

UNION-FIND

- Each disjoint set is a directed tree with edges point to root.
- Consider a vertex's rank to be reverse height of vertex in the tree. (Root rank is height, direct children: height - 1, leaf: 0)
- Finding a vertex returns the root of tree that vertex is in.
- Union of two vertices join at root. Directed edge from smaller rank root to larger rank root.
  - If ranks are equal, increment new root by 1

UNION-FIND

```
kruskal (G, v):
  for all vertices v: makeset(v)
  X = {}
  sort edges by weight
  for edge (u, v) in sorted edges E:
    if find(u) ≠ find(v):
      add edge (u, v) to X
      union(u, v)

makeset(v):
  π(v) = v      # parent of v
  rank(v) = 0


find(v):
  while v ≠ π(v): v = π(v)
  return v
```

```
union(x, y):
  r_x = find(x)
  r_y = find(y)
  if r_x = r_y: return
  if rank(r_x) > rank(r_y):
    π(r_y) = r_x
  else:
    π(r_x) = r_y
  if rank(r_x) > rank(r_y):
    rank(r_x) = rank(r_y) + 1
```

UNION-FIND

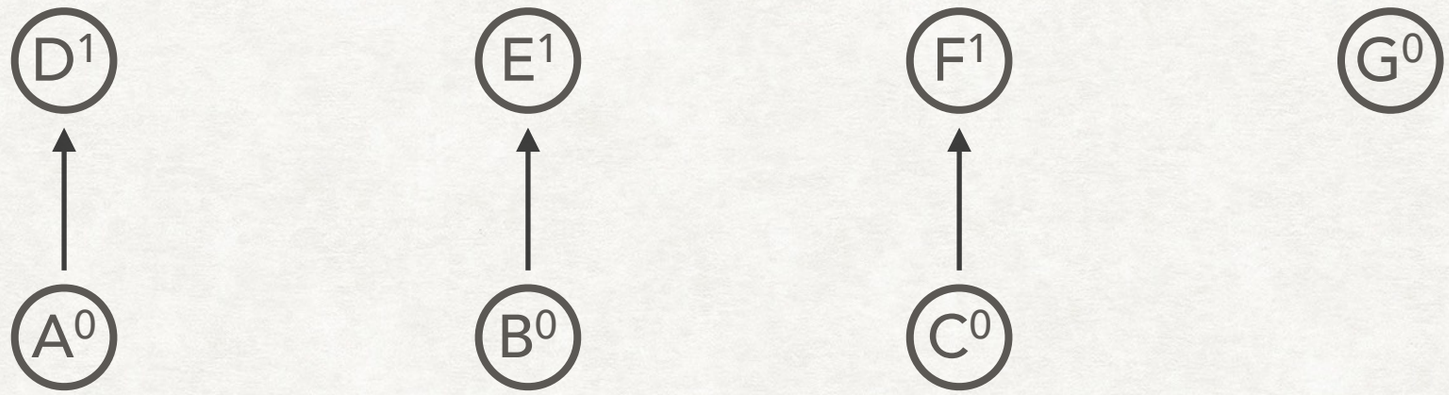
makeset(*A*), makeset(*B*), …, makeset(*G*)

UNION-FIND

union(*A*, *D*), union(*B*, *E*), union(*C*, *F*)

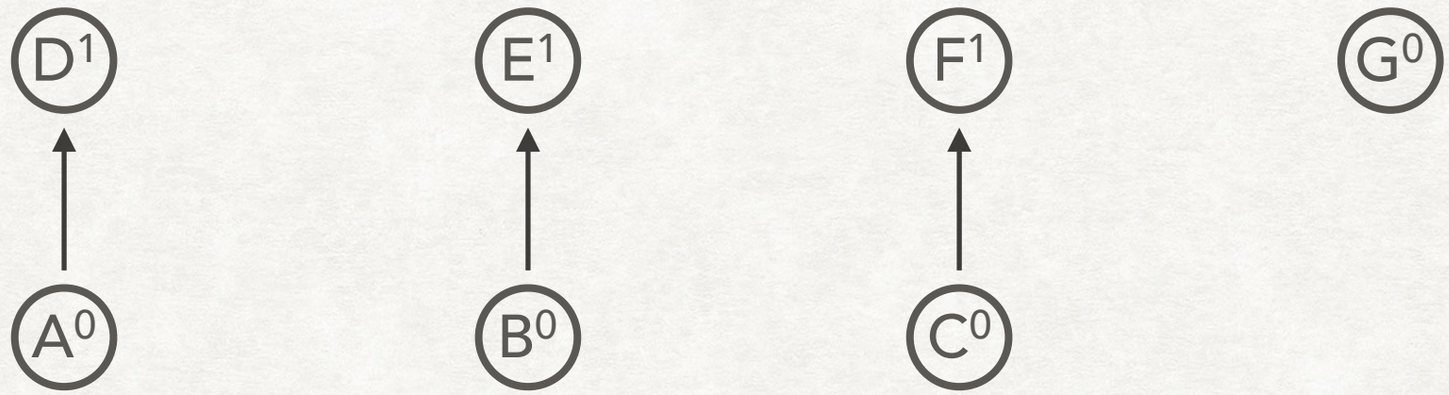
UNION-FIND

union(*A*, *D*), union(*B*, *E*), union(*C*, *F*)



UNION-FIND

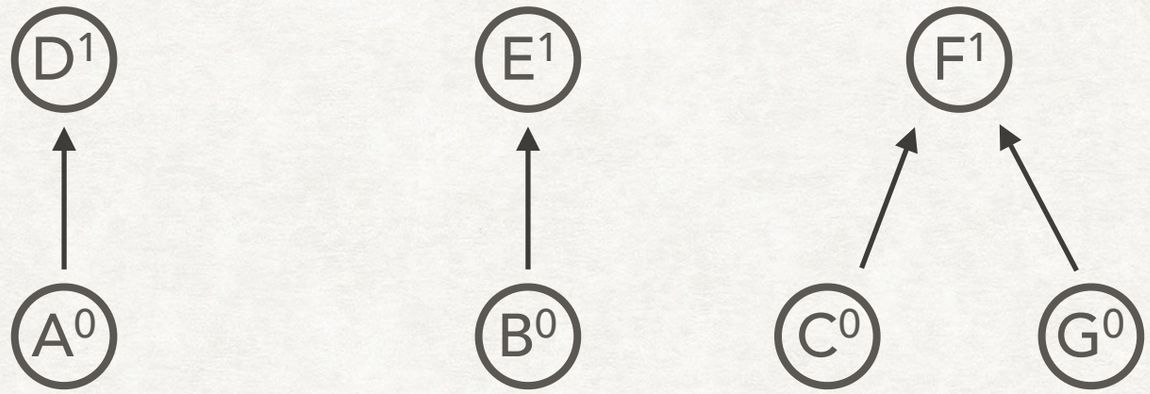
union(*C*, *G*)



UNION-FIND

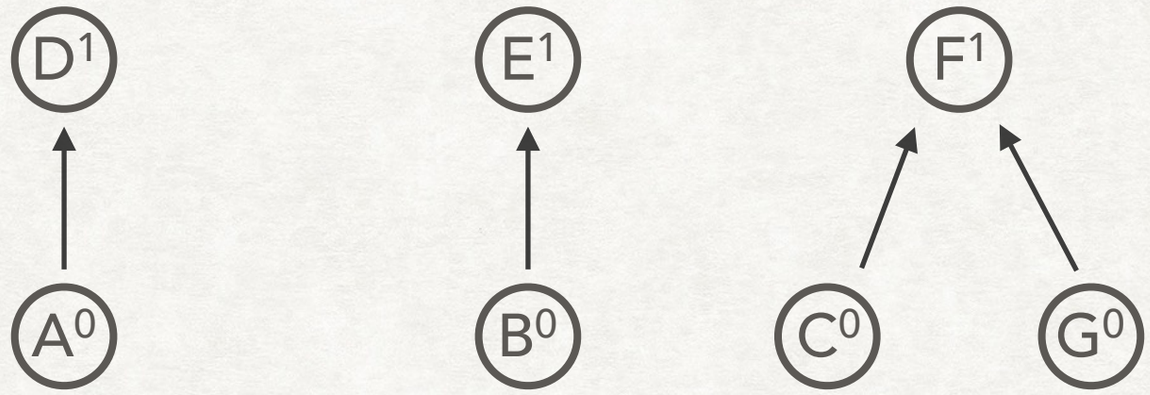
union(*C*, *G*)

- G's root is G.
- C's root is F.
- Connect G to F as F has higher rank.



UNION-FIND

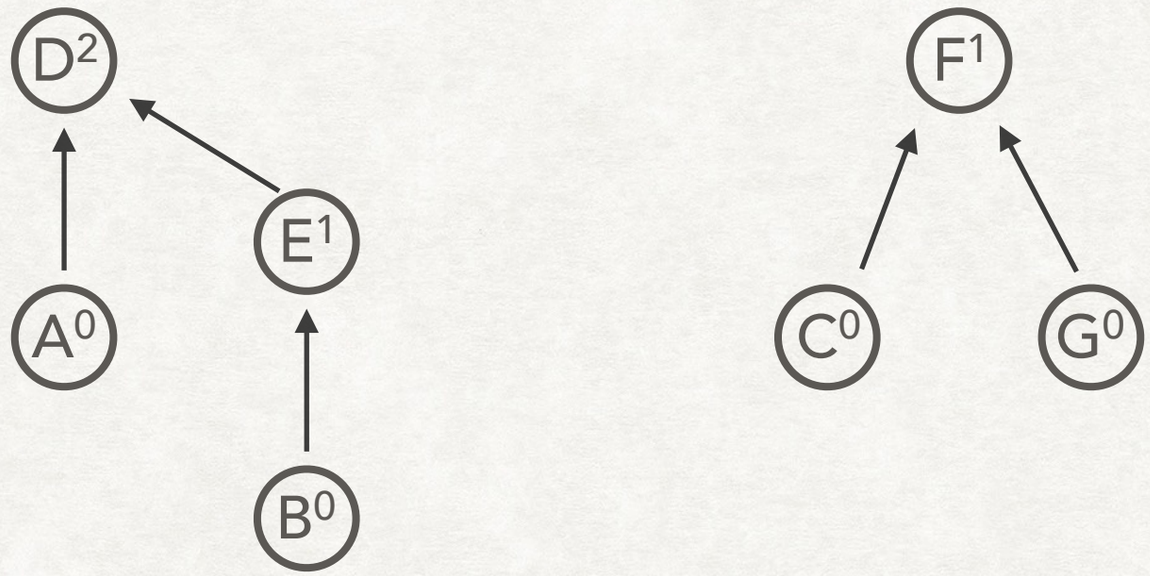
union(*E*, *A*)



UNION-FIND

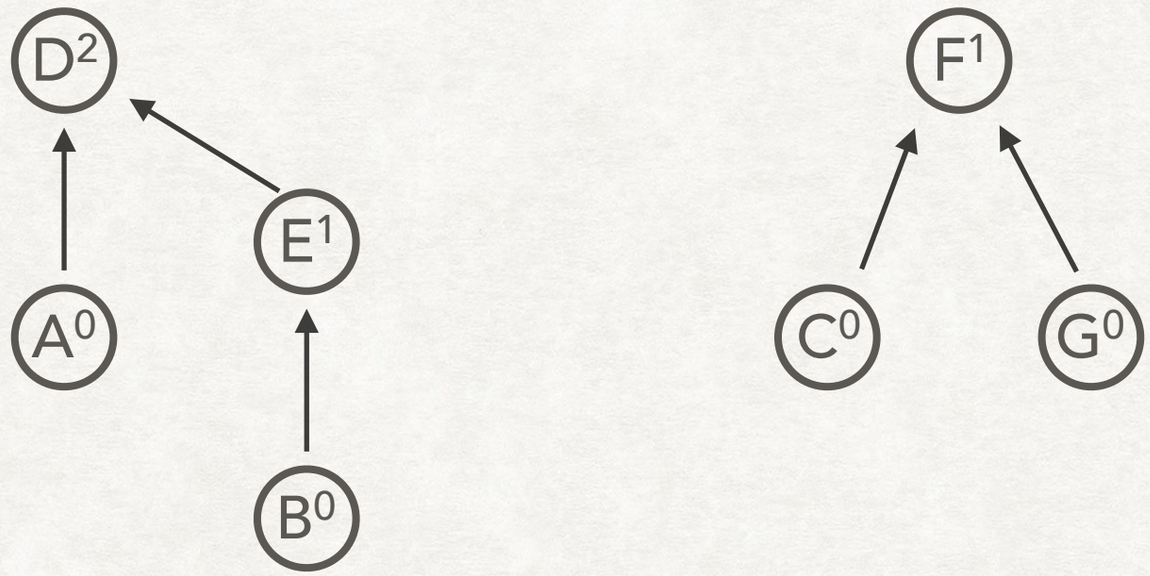
union(*E*, *A*)

- *E*'s root is E.
- A's root is D.
- D and E has same rank. Increment rank of new root (D).



UNION-FIND

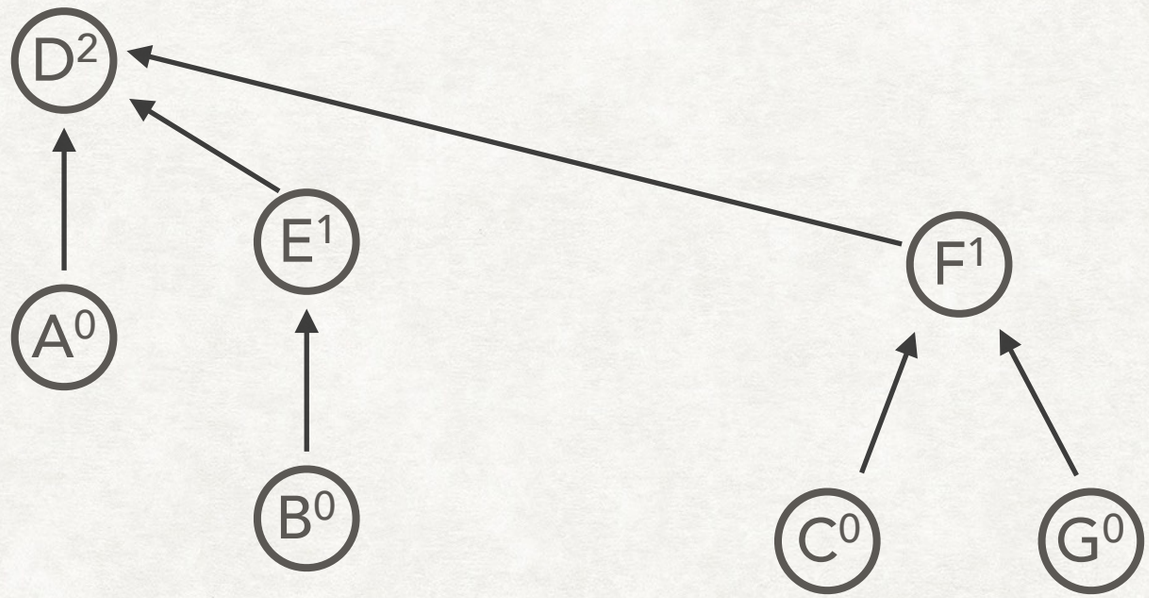
union(*B*, *G*)



UNION-FIND

union(*B*, *G*)

- *B*'s root is now D.
- G's root is F.
- Connect F to D as D has higher rank.

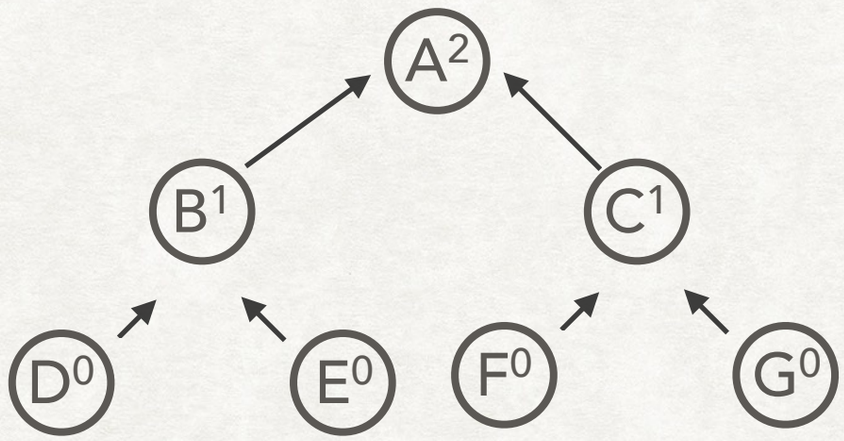


UNION-FIND

- Property 1: A node's rank is smaller than the node's parent's rank.
  - Rank of node is height of subtree rooted at that node.
  - Strictly increasing rank as we travel toward root node.

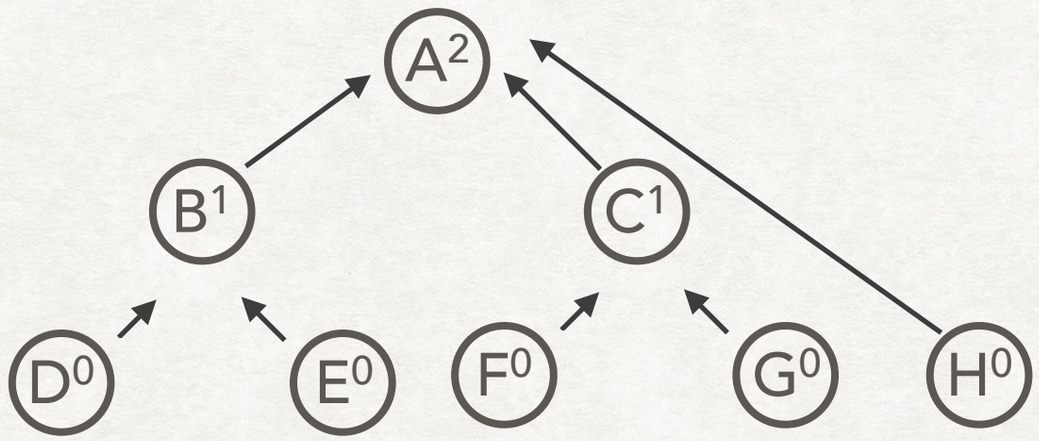
UNION-FIND

- Property 2: Any root node of rank k has at least 2^k nodes in its tree.
  - Can only increase rank when merging two roots with same rank.
  - Rank k root required merging **two** trees with roots of rank k - 1.
  - Root of rank k - 1 needed **two** tree roots of k - 2.



UNION-FIND

- Property 2: Any root node of rank k has at least 2^k nodes in its tree.
  - Can only increase rank when merging two roots with same rank.
  - Rank k root required merging **two** trees with roots of rank k - 1.
  - Root of rank k - 1 needed **two** tree roots of k - 2.



UNION-FIND

- Property 2: Any root node of rank k has at least 2^k nodes in its tree.

  (A^0) k = 0, at least $2^0 = 1$ node in tree

UNION-FIND

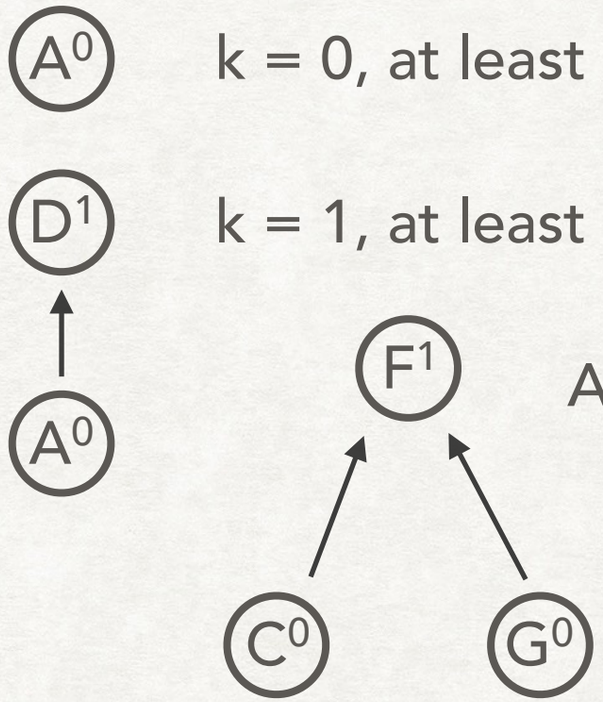
- Property 2: Any root node of rank k has at least 2^k nodes in its tree.

$k = 0$, at least $2^0 = 1$ node in tree

$k = 1$, at least $2^1 = 2$ nodes in tree

UNION-FIND

- Property 2: Any root node of rank k has at least 2^k nodes in its tree.



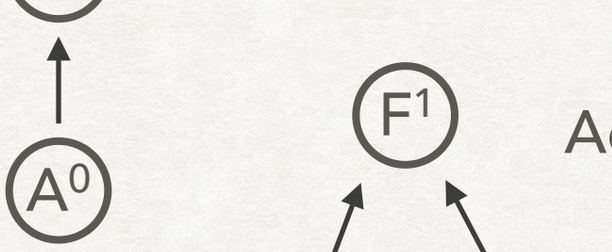
Adding lower rank doesn't change lower bound.

UNION-FIND

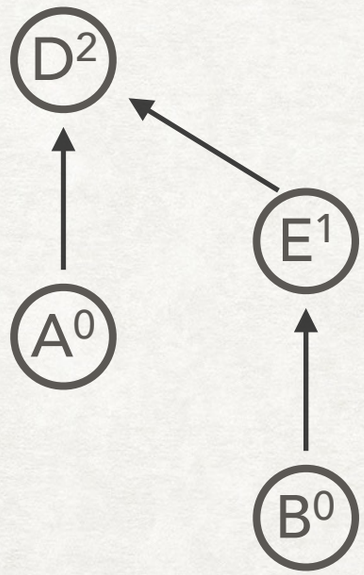
- Property 2: Any root node of rank k has at least 2^k nodes in its tree.

A^0 k = 0, at least $2^0 = 1$ node in tree

D^1 k = 1, at least $2^1 = 2$ nodes in tree



Adding lower rank doesn't change lower bound.



k = 2, at least $2^2 = 4$ nodes in tree. Can't have 3 nodes as it would contradict the fact that E has rank 1

UNION-FIND

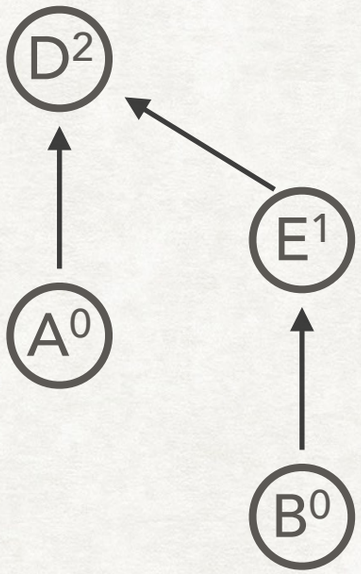
- Property 3: If there are n elements overall in a set, there can be at most n / 2^k nodes of rank k.
  - Root has at least 2^k descendants (including self).
  - Internal nodes also have at least 2^k descendants as they were roots in the past, where k is rank of internal node.
- Different rank-k nodes cannot have common descendants.
  - Merging at root.
  - Any element has at most one ancestor of rank k.

UNION-FIND

- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - We have some sort to tree structure that's balanced.
  - Maximum rank is log n.
  - Tree have maximum possible height of log n.
  - Running time of find and union upper bounded by log n.

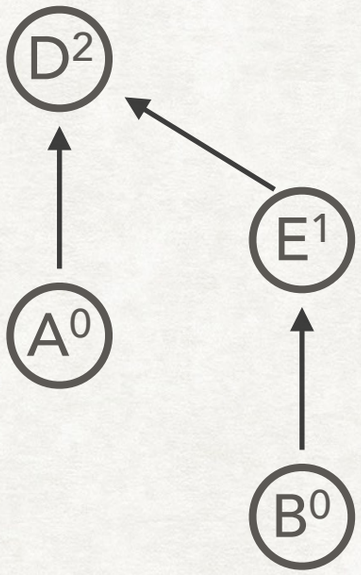
UNION-FIND

- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - Consider n = 4.



UNION-FIND

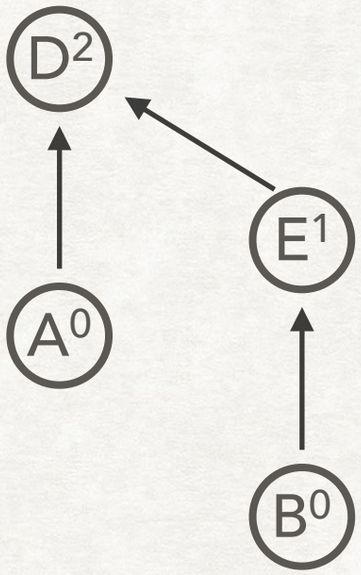
- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - Consider n = 4.



Can't have higher rank than 2 without more elements.

UNION-FIND

- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - Consider n = 4.

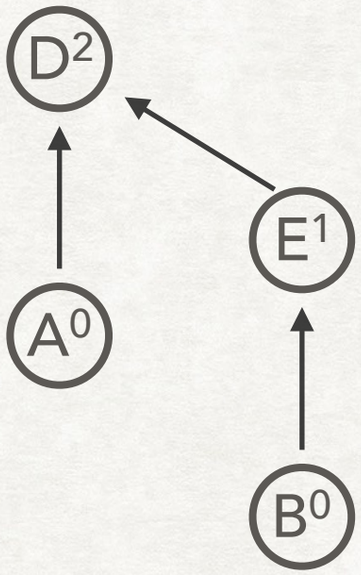


Can't have higher rank than 2 without more elements.

At most $4 / 2^2 = 1$ element of rank 2

UNION-FIND

- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - Consider n = 4.



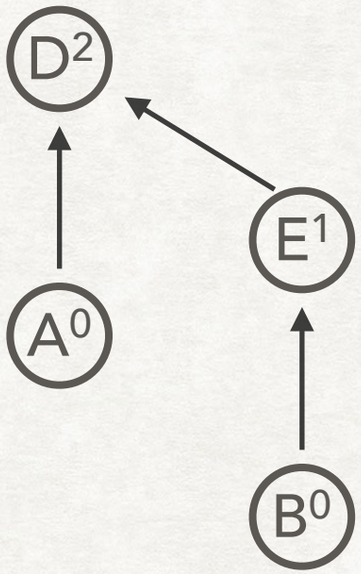
Can't have higher rank than 2 without more elements.

At most $4 / 2^2 = 1$ element of rank 2

At most $4 / 2^1 = 2$ elements of rank 1

UNION-FIND

- Property 3: If there are n elements overall, there can be at most $n / 2^k$ nodes of rank k.
  - Consider n = 4.



Can't have higher rank than 2 without more elements.

At most $4 / 2^2 = 1$ element of rank 2

At most $4 / 2^1 = 2$ elements of rank 1

At most $4 / 2^0 = 4$ elements of rank 0

UNION-FIND

```
kruskal (G, v):
  for all vertices v: makeset(v)
  X = {}
  sort edges by weight
  for edge (u, v) in sorted edges E:
    if find(u) ≠ find(v):
      add edge (u, v) to X
      union(u, v)

makeset(v):
  π(v) = v      # parent of v
  rank(v) = 0


find(v):
  while v ≠ π(v): v = π(v)
  return v
```

```
union(x, y):
  r_x = find(x)
  r_y = find(y)
  if r_x = r_y: return
  if rank(r_x) > rank(r_y):
    π(r_y) = r_x
  else:
    π(r_x) = r_y
    if rank(r_x) > rank(r_y):
      rank(r_x) = rank(r_y) + 1
```

Sorting is O(|E| log(|E|))

|E| iterations.

find and union take O(log(|V|))

Kruskal takes O(|E| log(|E|) + |E| log(|V|))

PATH COMPRESSION

- Let's keep trees short,
- On find, have all nodes all path to root point to root.
- Increases cost of find.
- But when considering sequences of find and union operations, amortized (average) cost becomes bit more than O(1).

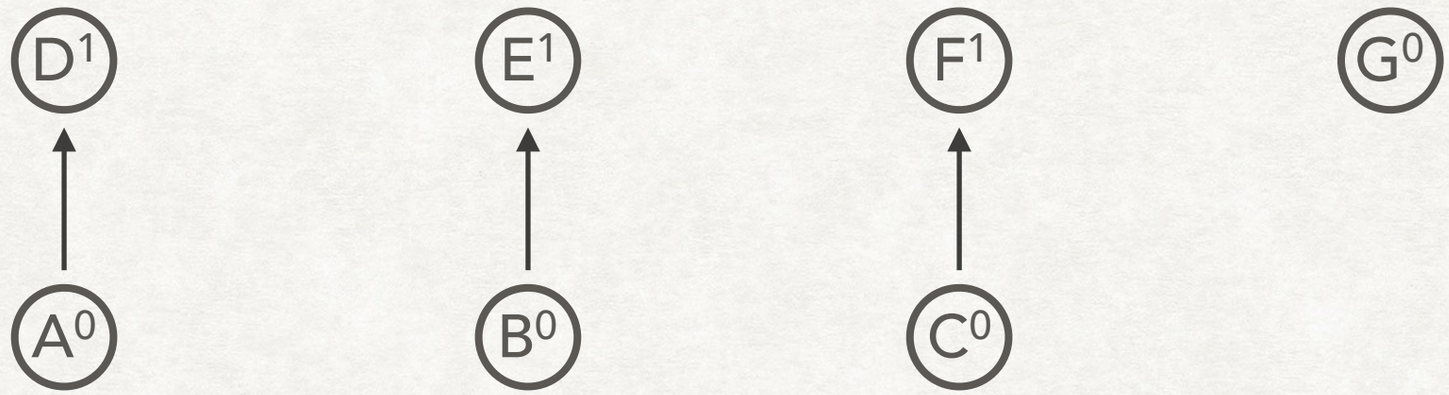
```
find(v):
  while v ≠ π(v): π(v) = find(π(v))
  return π(v)
```

PATH COMPRESSION

makeset(*A*), makeset(*B*), …, makeset(*G*)

PATH COMPRESSION

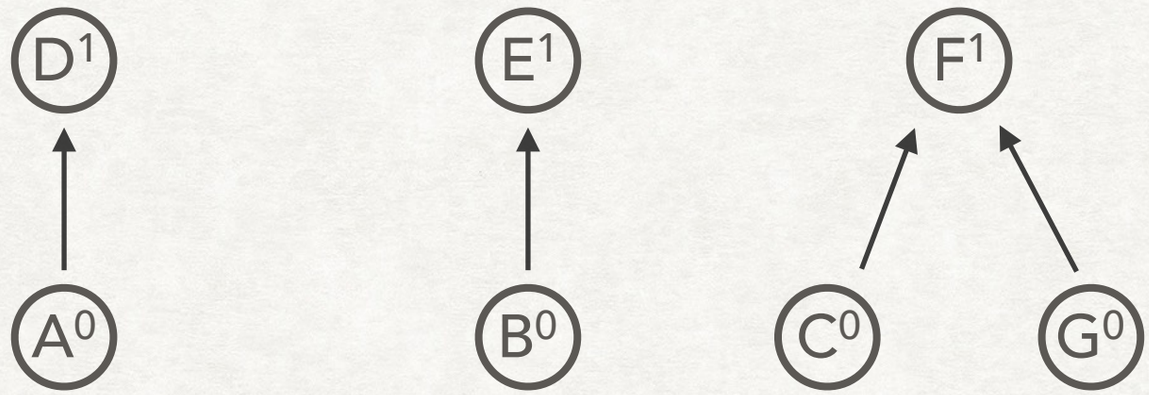
union(*A*, *D*), union(*B*, *E*), union(*C*, *F*)



PATH COMPRESSION

union(*C*, *G*)

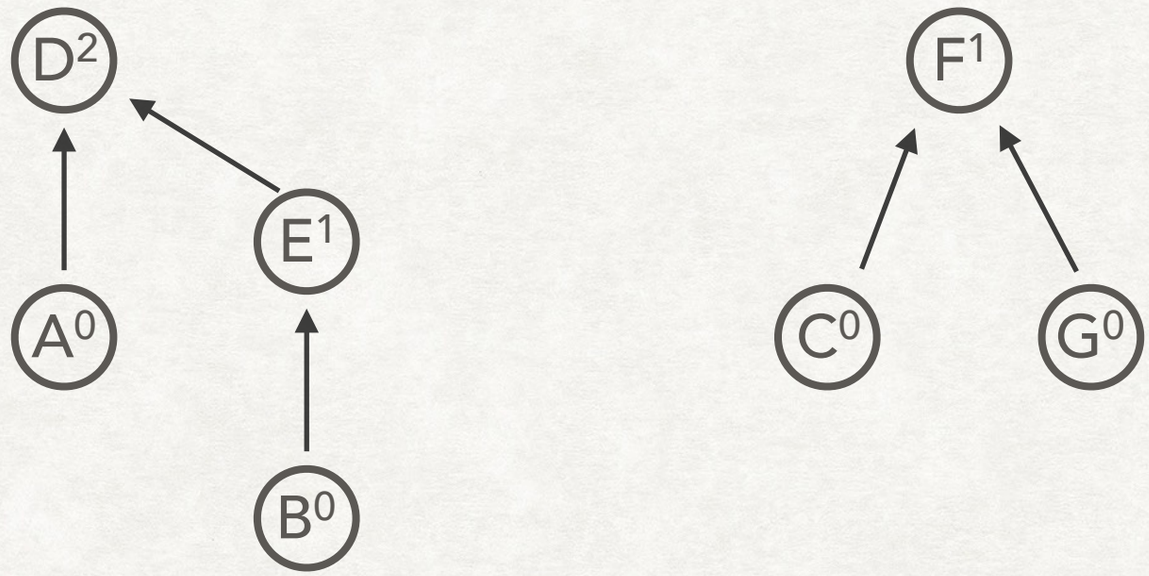
- G's root is G.
- C's root is F.
- Connect G to F as F has higher rank.



PATH COMPRESSION

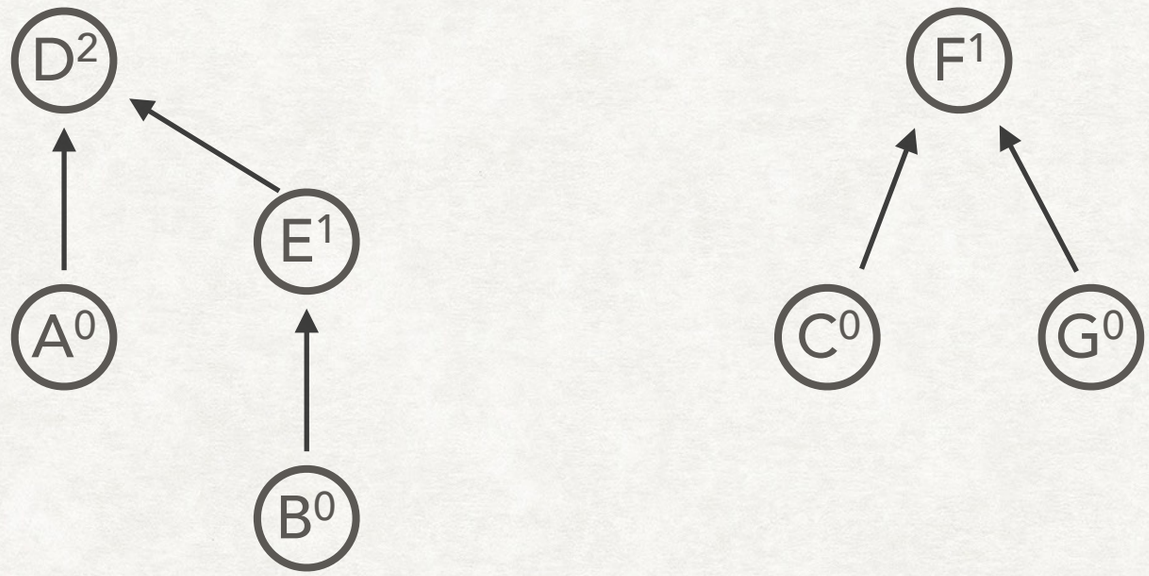
union(*B*, *A*)

- *B*'s root is E.
- A's root is D.
- D and E has same rank. Increment rank of new root (D).



PATH COMPRESSION

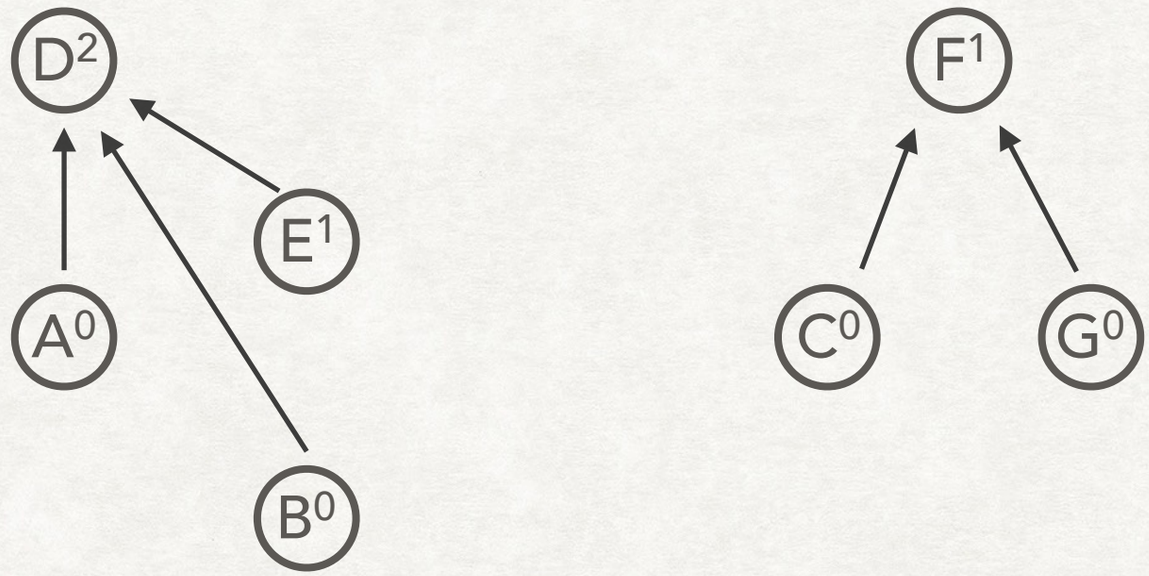
find(*B*)



PATH COMPRESSION

find(*B*)

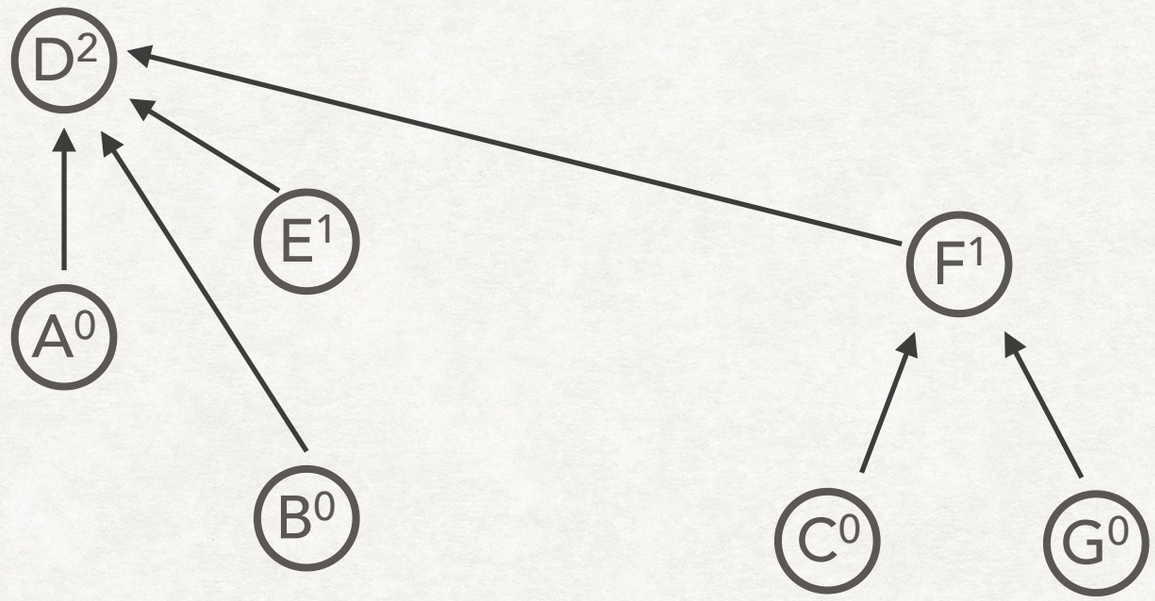
- *B*'s root is D.
- B is in path to root. Connect it to D. Rank of B doesn't change.



PATH COMPRESSION

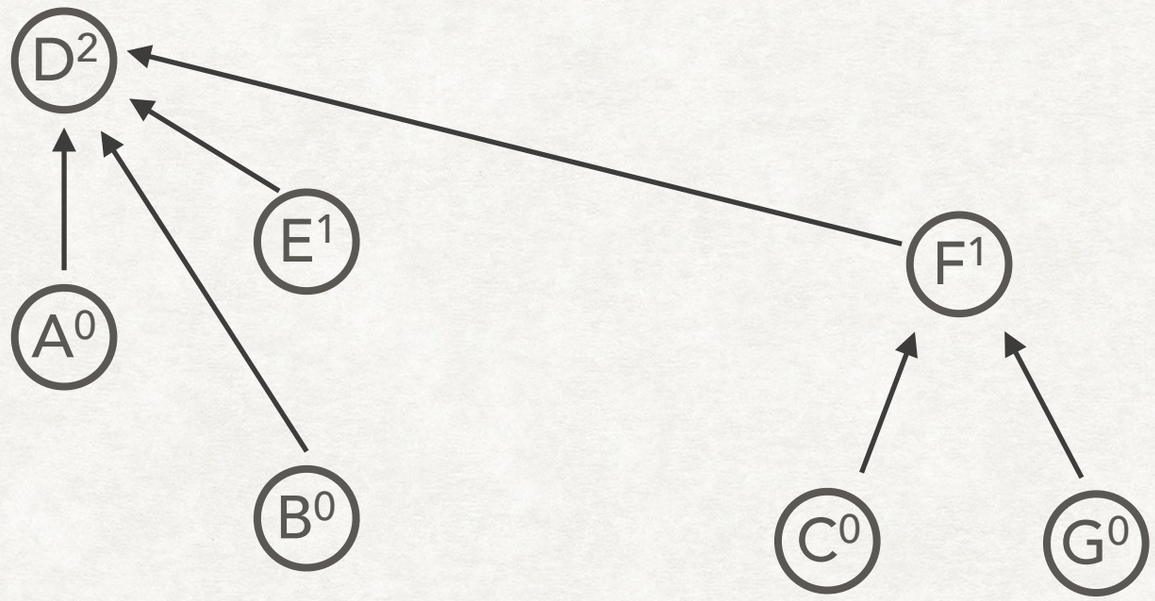
union(*B*, *G*)

- *B*'s root is now D.
- G's root is F.
- Connect F to D as D has higher rank.
- G on path to F.



PATH COMPRESSION

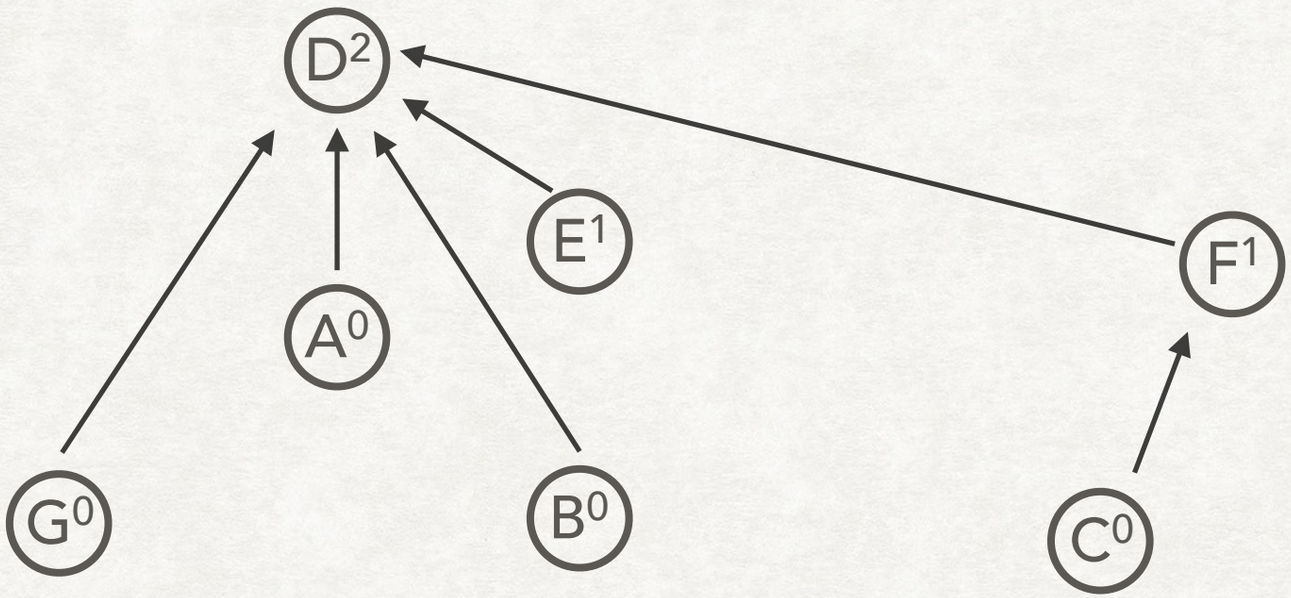
find(*G*)



PATH COMPRESSION

find(*G*)

- *G*'s root is now D.
- G on path to D. Connect G to D. C is not on path.



PATH COMPRESSION

- Don’t worry too much about path compression proof, but…
- Know ***log*(n)***.
  - The number of successively applying log operations on *n* to get it down to 1.
- *$\log^*(1000) = 4$. log log log log 1000 ≤ 1*

```
find(v):
  while v ≠ π(v): π(v) = find(π(v))
  return π(v)
```

PATH COMPRESSION

- Also understand idea of amortized cost.
- Single operation in worst case may take longer due to some overhead.
- But applying a sequence of operations allows us to distribute that overhead to many number of operations.
- Overhead cost applied only from time to time.
- Total cost averages out over each operation in sequence.

PATH COMPRESSION

- Example: resizing array.
- Only need to resizing when array is full.
- Resize array to be 2x as before.
- Overhead is copying: O(n)
- But until array is full, it’s constant appending and indexing.
- Overhead costs is amortized or averaged over the many appending and indexing operations.