COMP3251
Lecture 12: Kruskal's Algorithm
(Chapter 5.1)

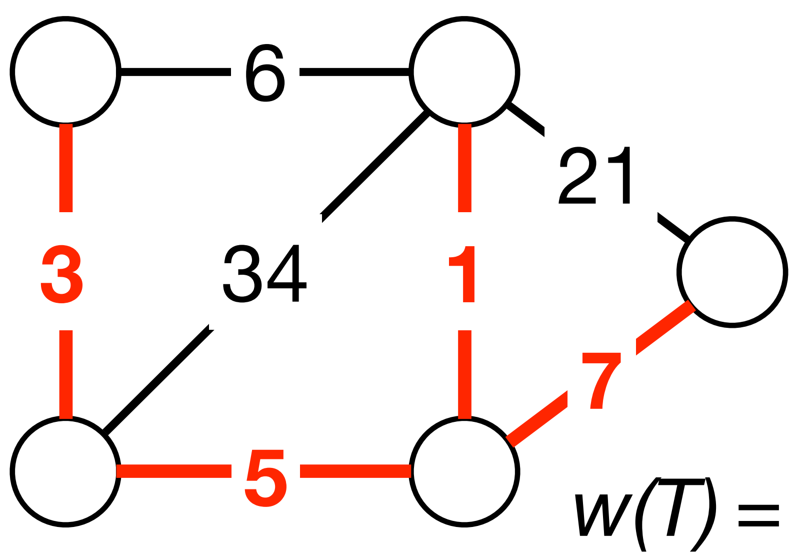
Minimum Spanning Tree

Definition: Given a connected undirected graph $G = (V, E)$ in which every edge $e \in E$ is associated with a positive weight $w(e)$, a **minimum spanning tree (MST)** is a subset of edges $T \subseteq E$ s.t.

(i) T forms a spanning tree; and

(ii) the sum of edge weights of T is minimized.

Example:



$w(T) = 16$

Two Greedy Algorithms for MST

Prim's algorithm (last lecture)

- Start with some root node *s* and grow a tree *T* outward.
- At each step, add the minimum weight outgoing edge.
- This algorithm is almost the same as the Dijkstra's algorithm, except that we add the outgoing edge with the minimum weight, not the one with minimum *T*-distance.

Kruskal's algorithm (this lecture)

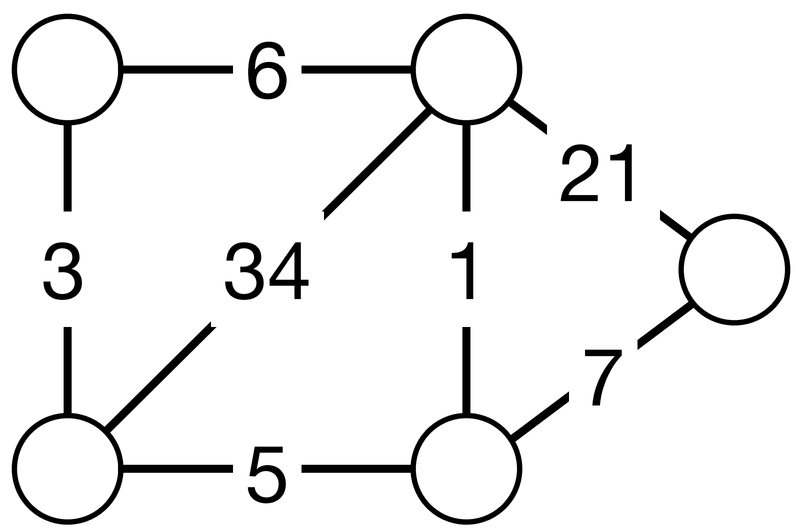
- Start with *T* being the empty forest.
- Consider edges in ascending order of cost; insert edge *e* in *T* unless doing so would create a cycle.

Kruskal's Algorithm (Chapter 5.1.3)

1) Start from an empty forest T.
2) **for** all edges e in ascending order of weights **:**
3) insert edge e in T unless doing so would create a cycle.

Kruskal's Algorithm (Chapter 5.1.3)

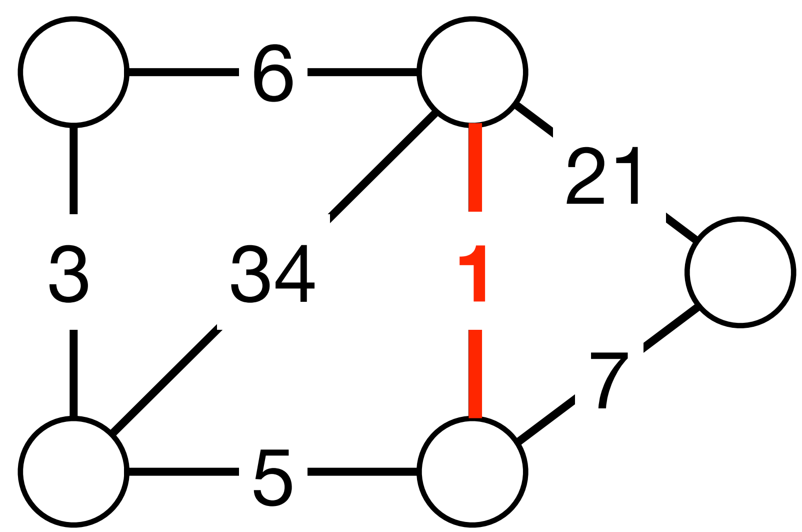
1) Start from an empty forest *T*.
2) **for** all edges *e* in ascending order of weights **:**
3) insert edge *e* in *T* unless doing so would create a cycle.



Initially, *T* contains no edges.

Kruskal's Algorithm (Chapter 5.1.3)

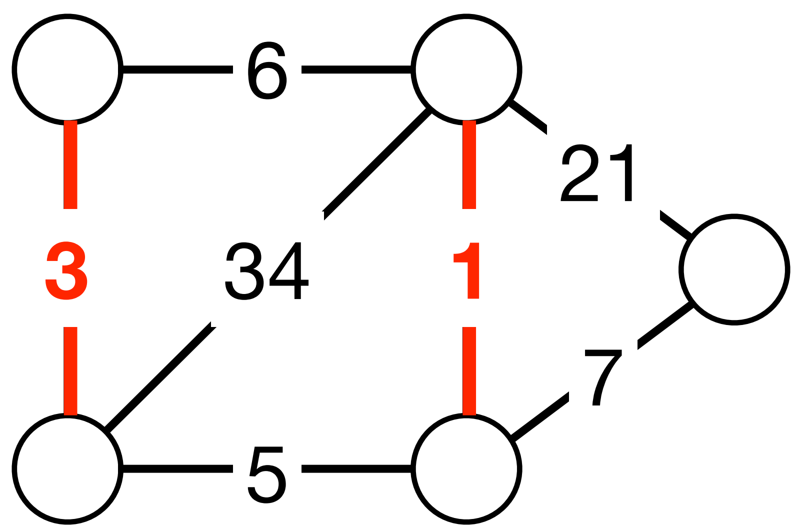
1) Start from an empty forest *T*.
2) **for** all edges *e* in ascending order of weights **:**
3) insert edge *e* in *T* unless doing so would create a cycle.



At step 1, the minimum weight edge
we could add has weight *1*.

Kruskal's Algorithm (Chapter 5.1.3)

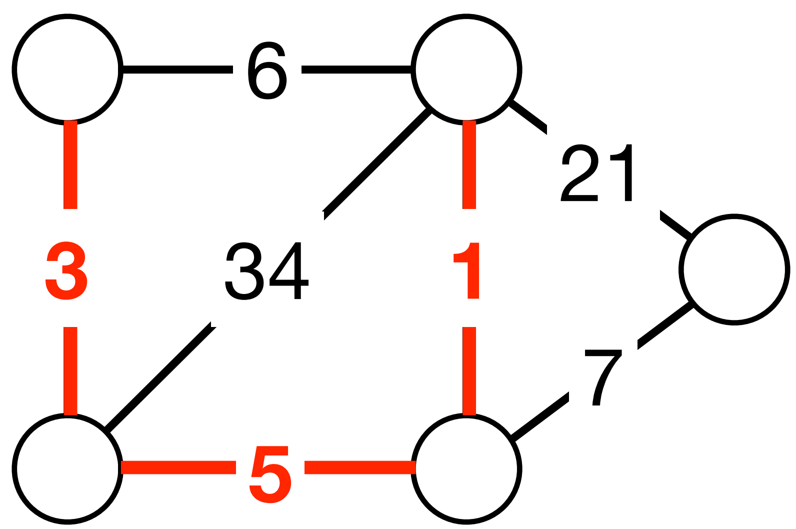
1) Start from an empty forest *T*.
2) **for** all edges *e* in ascending order of weights **:**
3) insert edge *e* in *T* unless doing so would create a cycle.



At step 2, the minimum weight edge
we could add has weight *3*.

Kruskal's Algorithm (Chapter 5.1.3)

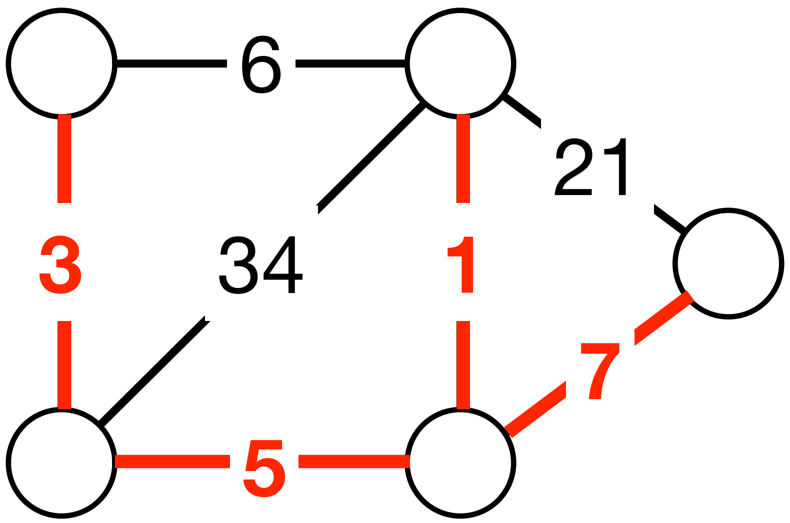
1) Start from an empty forest *T*.
2) **for** all edges *e* in ascending order of weights **:**
3) insert edge *e* in *T* unless doing so would create a cycle.



At step 3, the minimum weight edge
we could add has weight *5*.

Kruskal's Algorithm (Chapter 5.1.3)

1) Start from an empty forest *T*.
2) **for** all edges *e* in ascending order of weights **:**
3) insert edge *e* in *T* unless doing so would create a cycle.

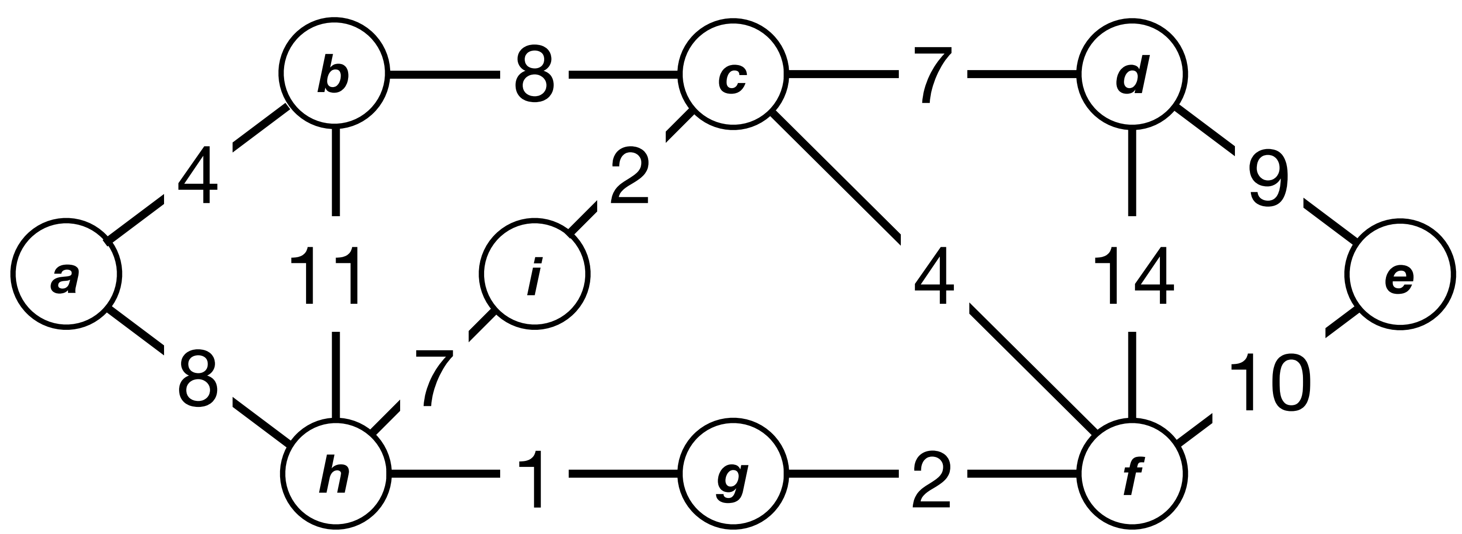


At step 4, the edge with weight *6* creates a cycle, so we add the edge with weight *7* instead.

Correctness of Kruskal's Algorithm

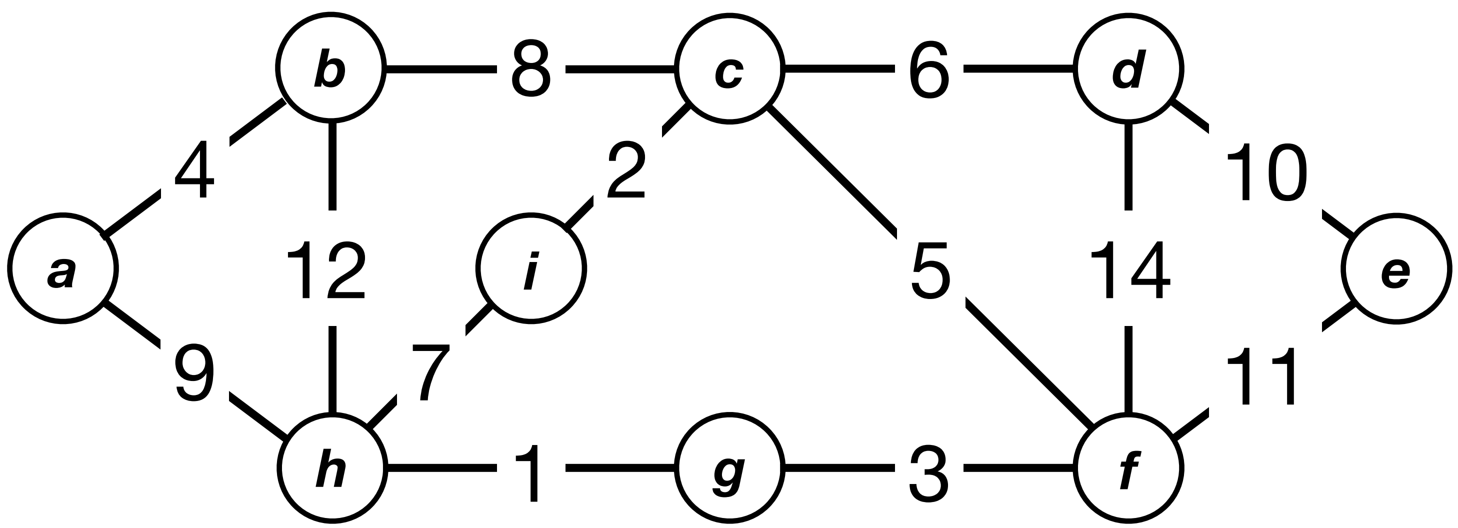
Proof by picture:

- Recall that if *e* is the minimum weight outgoing edge of some subset of vertices *S*, then the MST must contain *e*.
- For every edge *e* we add in the sample run, we will explain the subset of vertices *S* for which *e* is the minimum weight outgoing edge of *S*, certifying *e* must be in the MST.



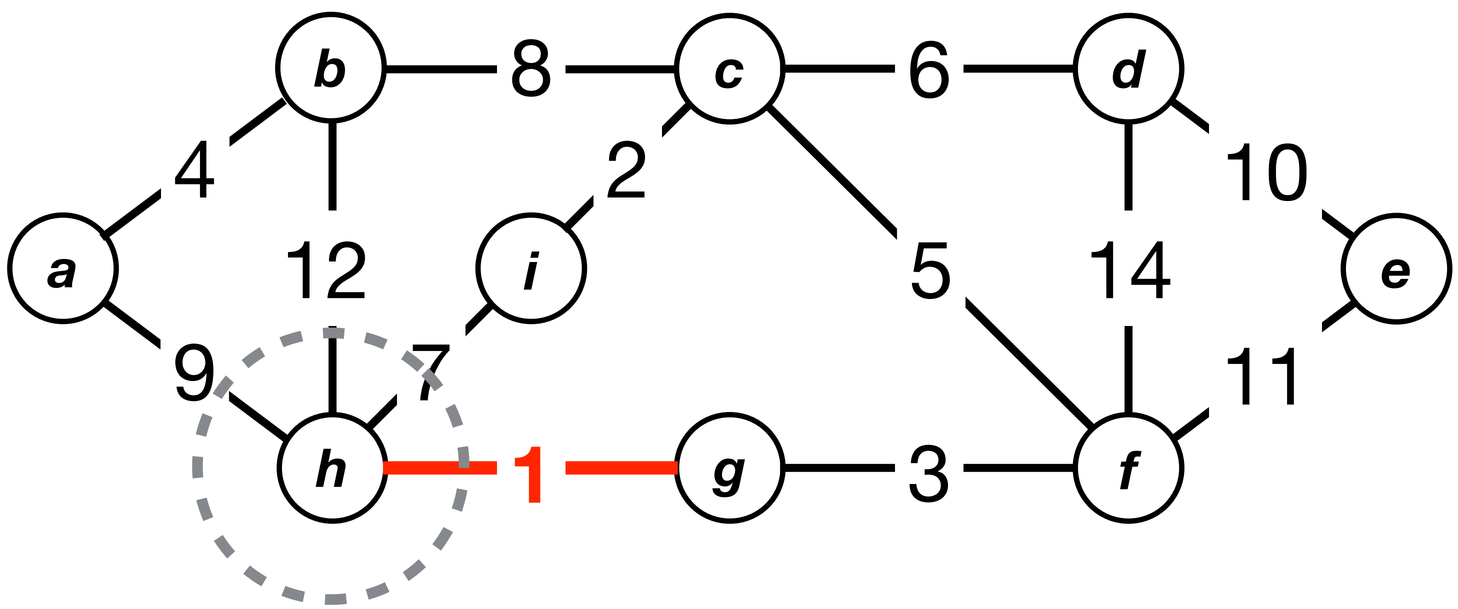
Correctness of Kruskal's Algorithm

(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

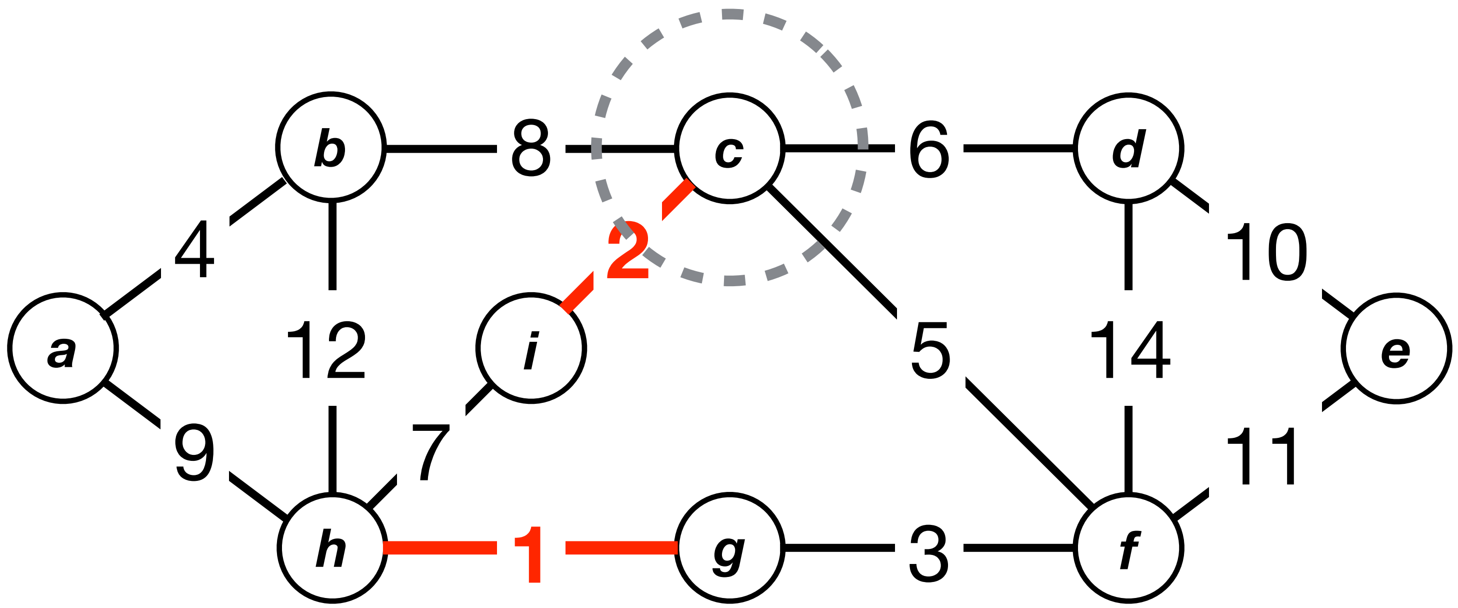
(h,g)	**1**
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(h,g) is the minimum weight edge going out from $S = \{ h \}$.

Correctness of Kruskal's Algorithm

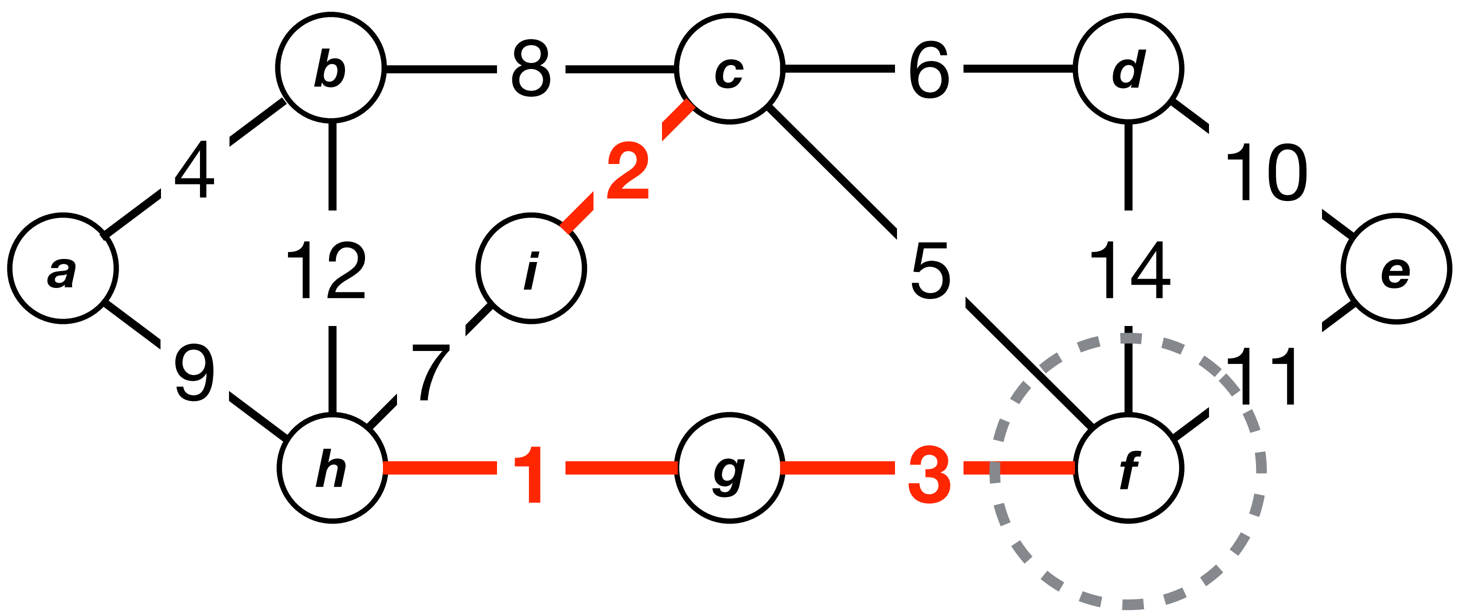
(h,g)	**1**
(i,c)	**2**
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(i,c) is the minimum weight edge going out from $S = \{ c \}$.

Correctness of Kruskal's Algorithm

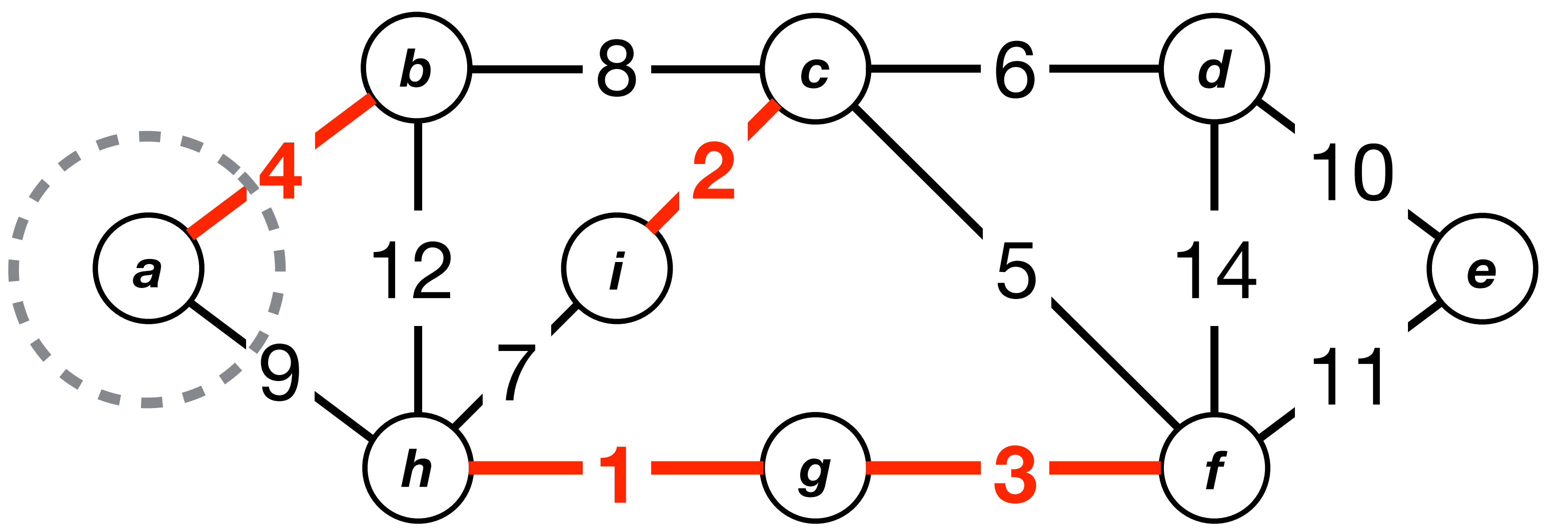
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(g,f) is the minimum weight edge going out from $S = \{ f \}$.

Correctness of Kruskal’s Algorithm

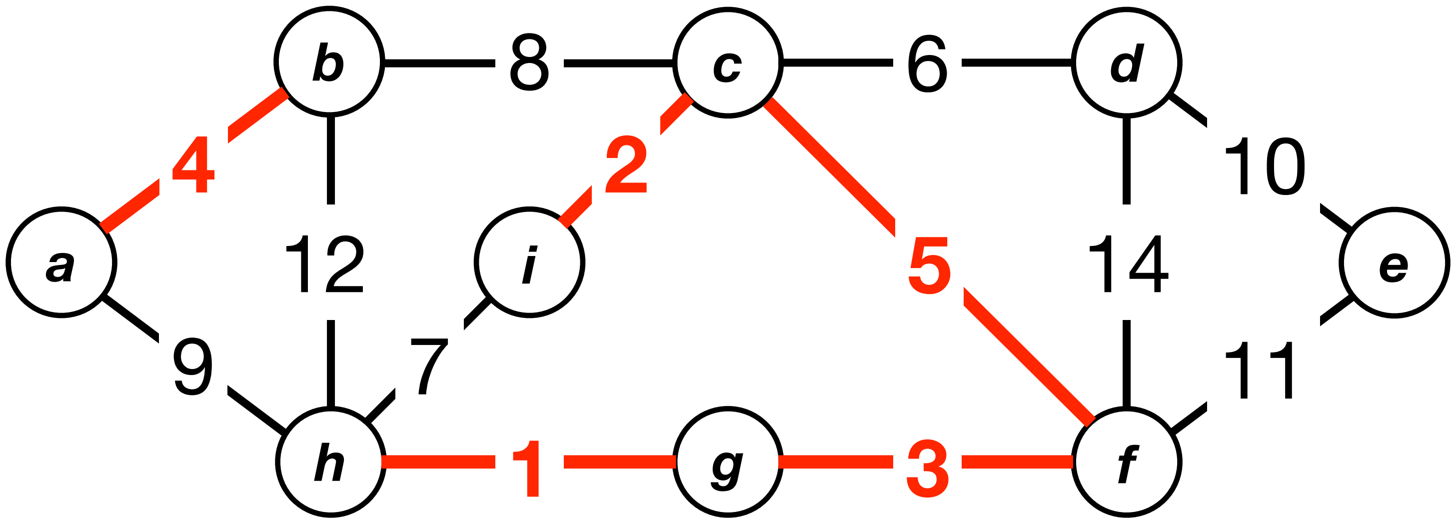
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(a,b) is the minimum weight edge going out from $S = \{ a \}$.

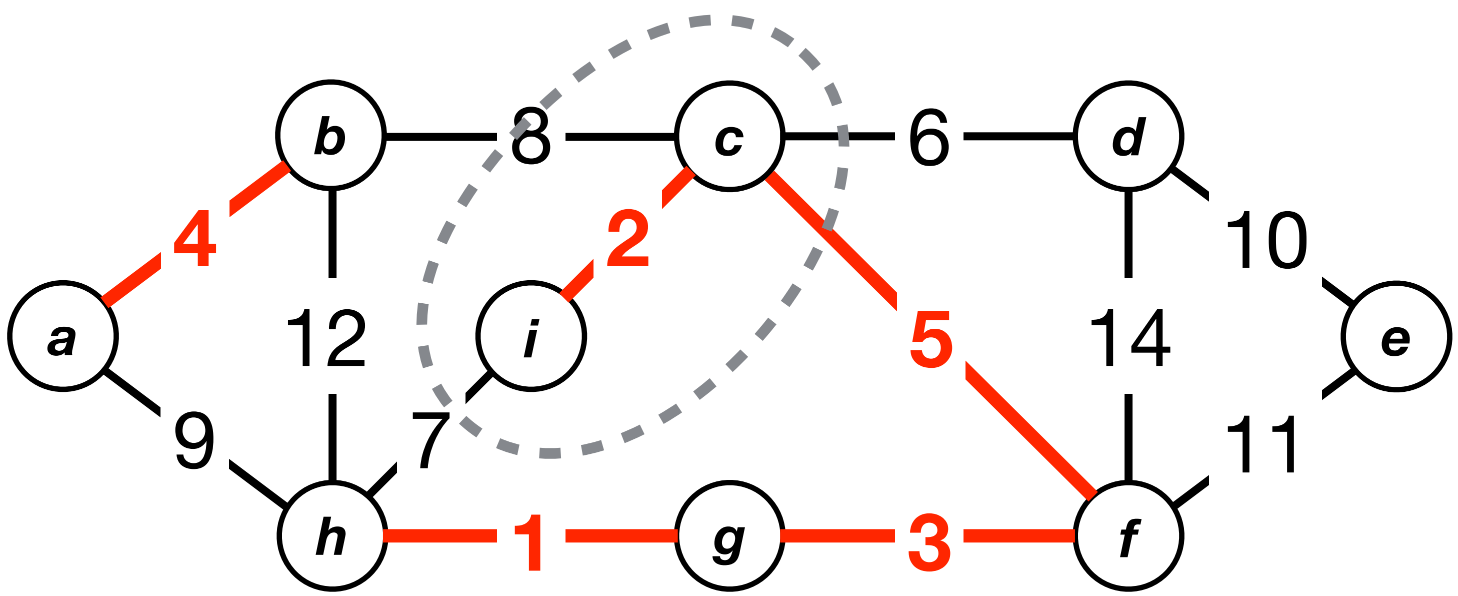
Correctness of Kruskal's Algorithm

(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

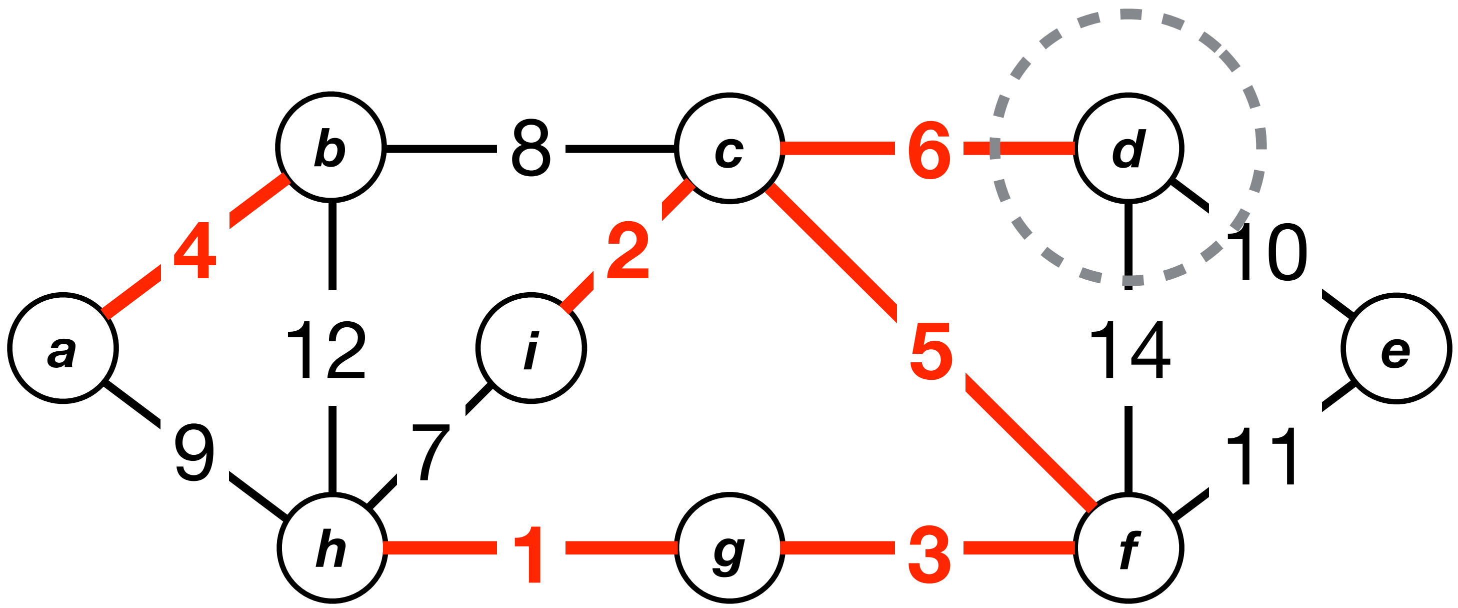
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(c,f) is the minimum weight edge going out from $S = \{ c, i \}$.

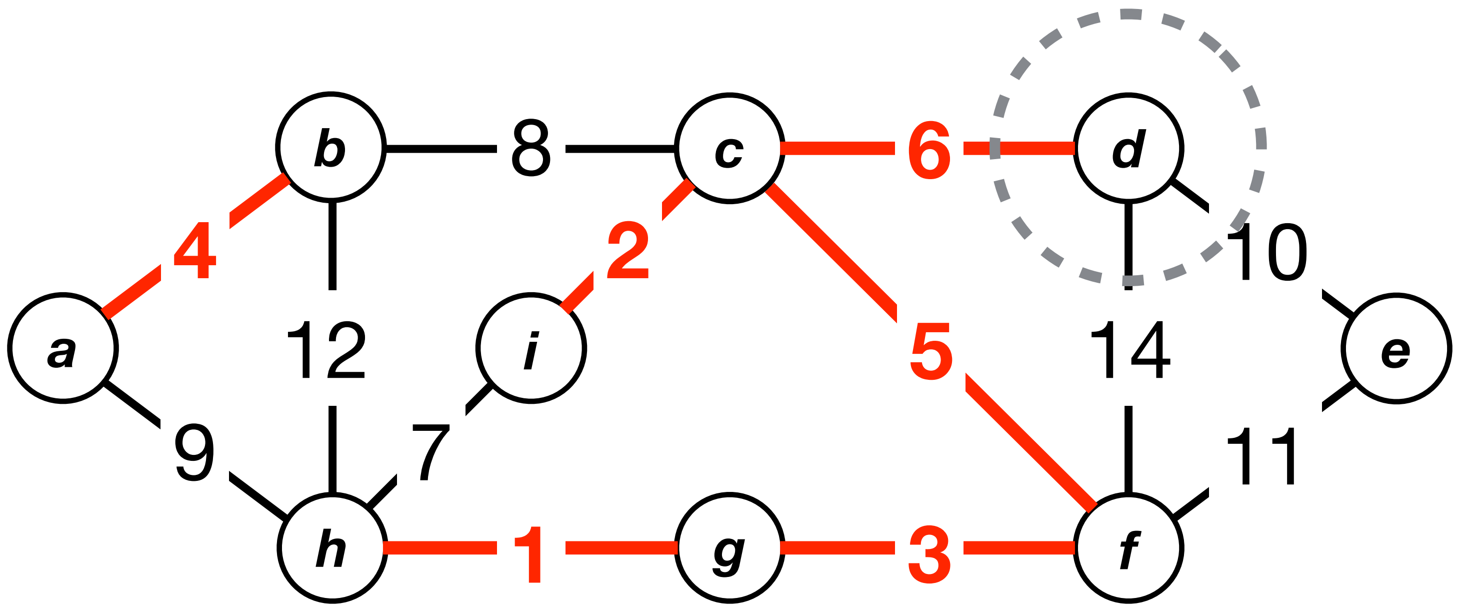
Correctness of Kruskal's Algorithm

(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

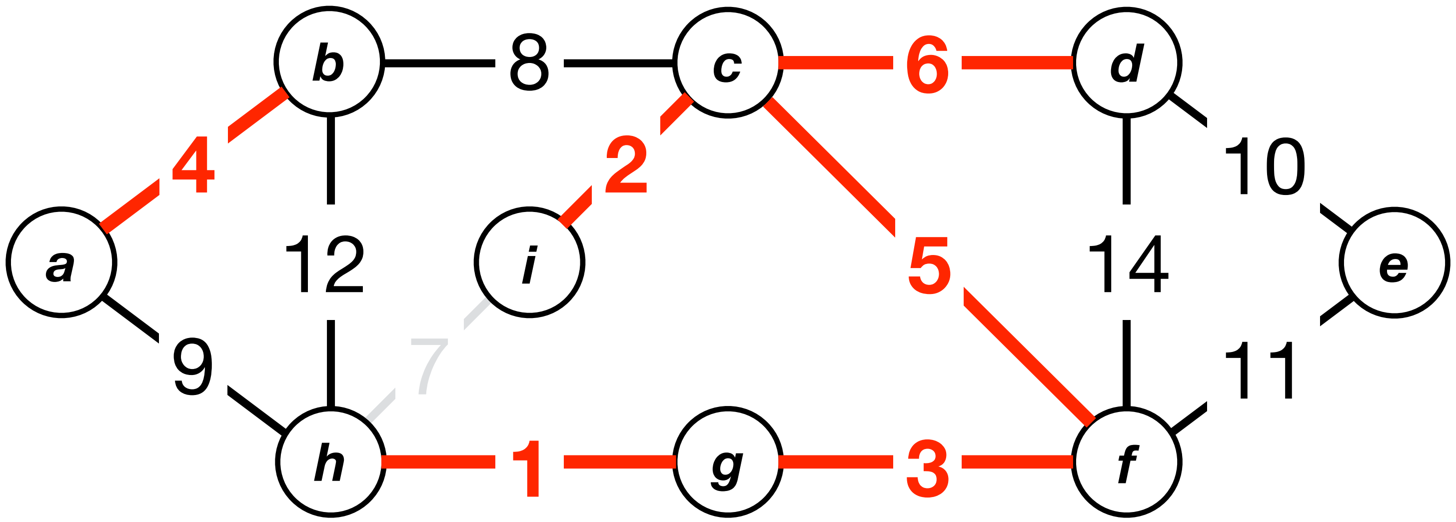
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(c,d) is the minimum weight edge going out from $S = \{ d \}$.

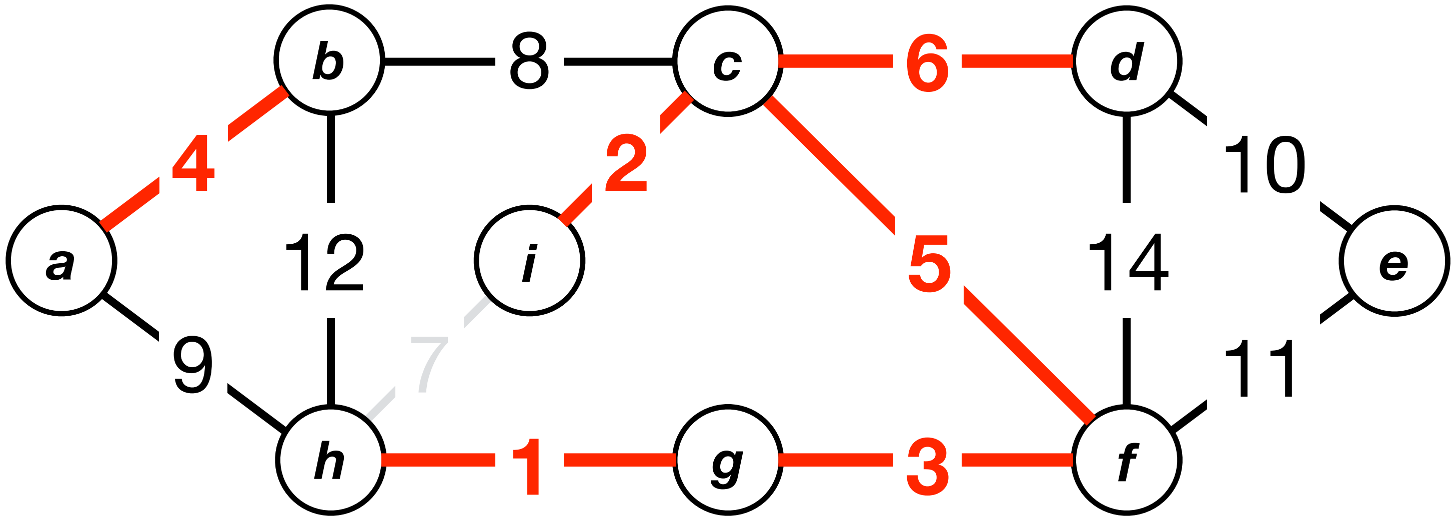
Correctness of Kruskal's Algorithm

(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

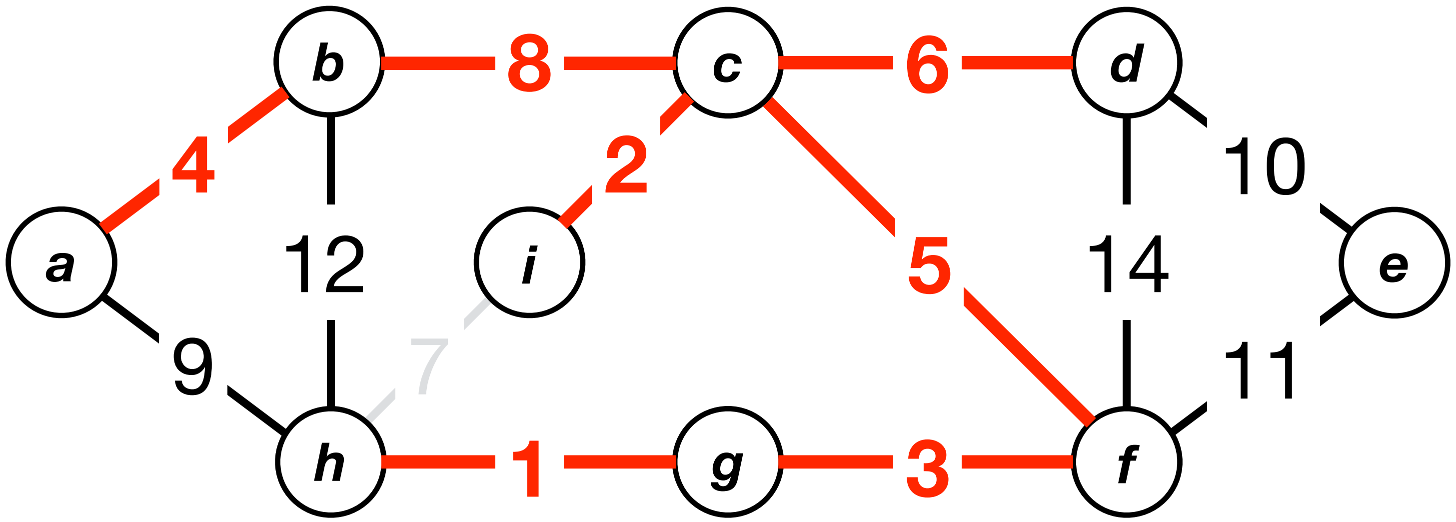
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(h,i) cannot be added to the solution because doing so would create a cycle.

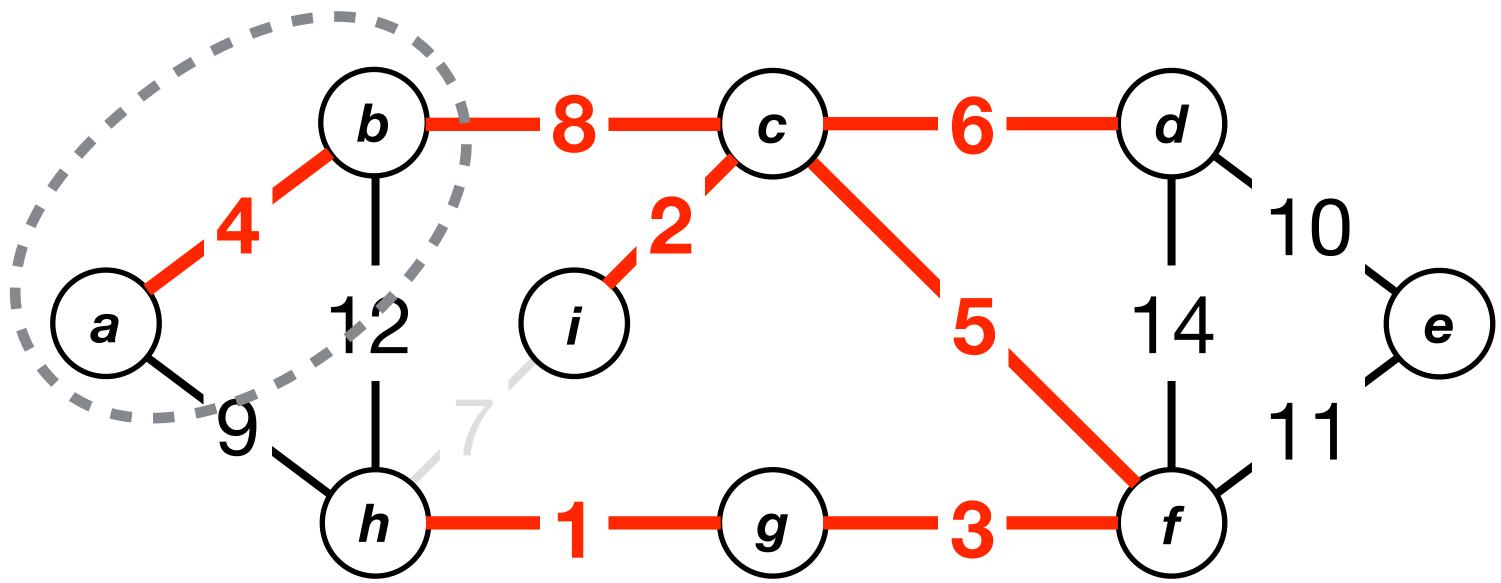
Correctness of Kruskal's Algorithm

(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	**8**
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

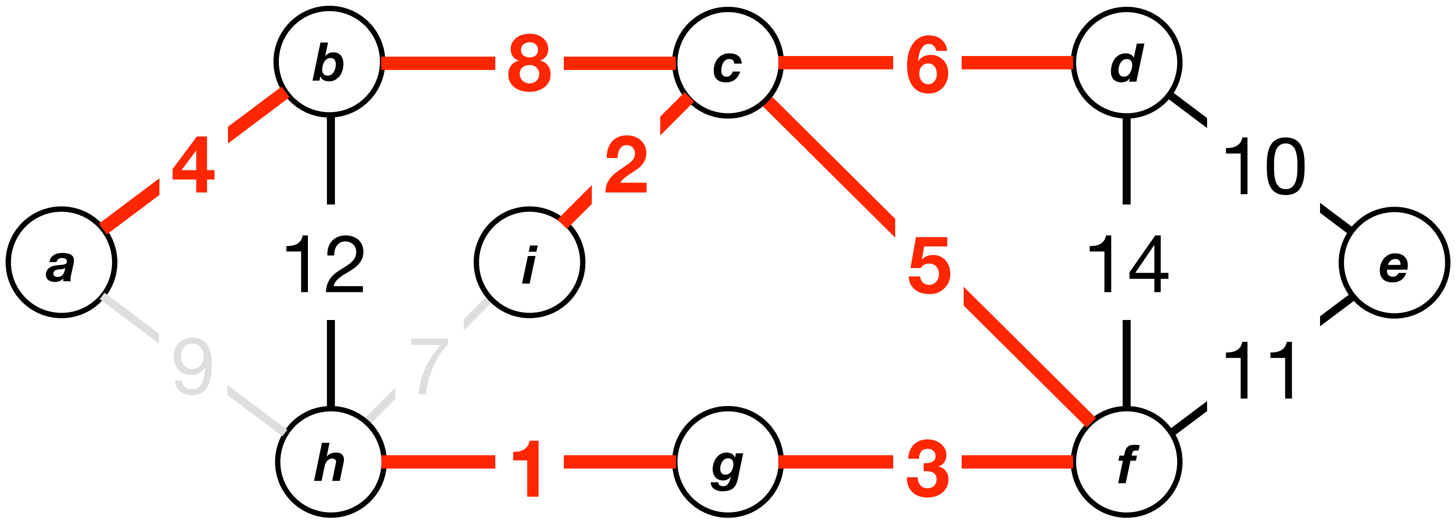
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	**8**
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(b,c) is the minimum weight edge going out from $S = \{ a, b \}$.

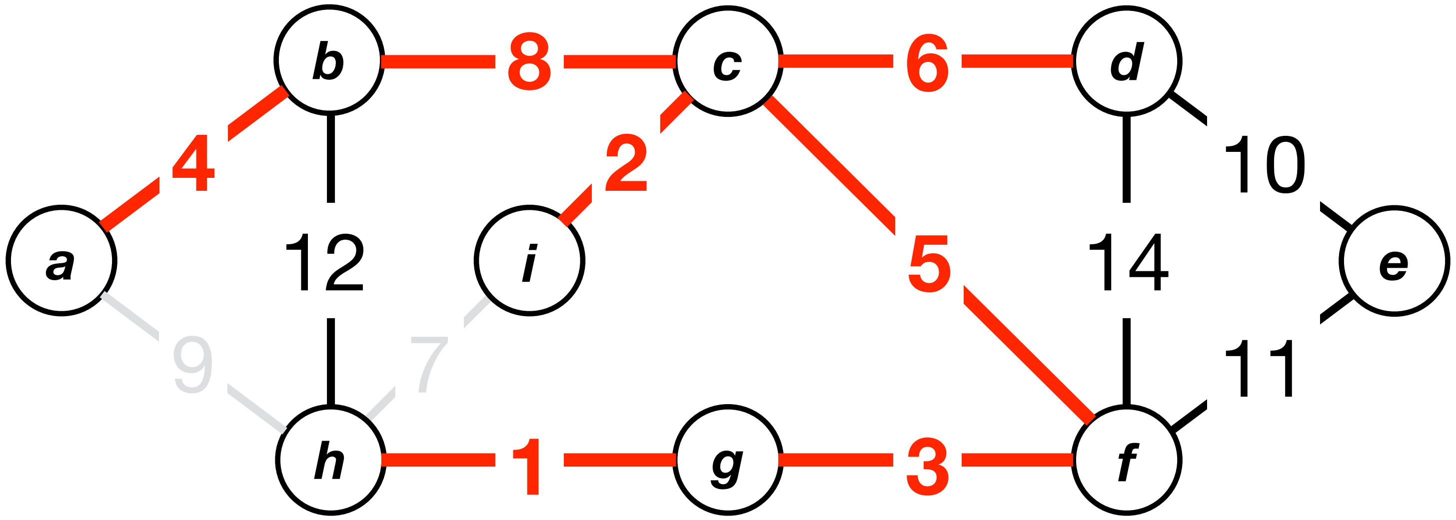
Correctness of Kruskal's Algorithm

(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

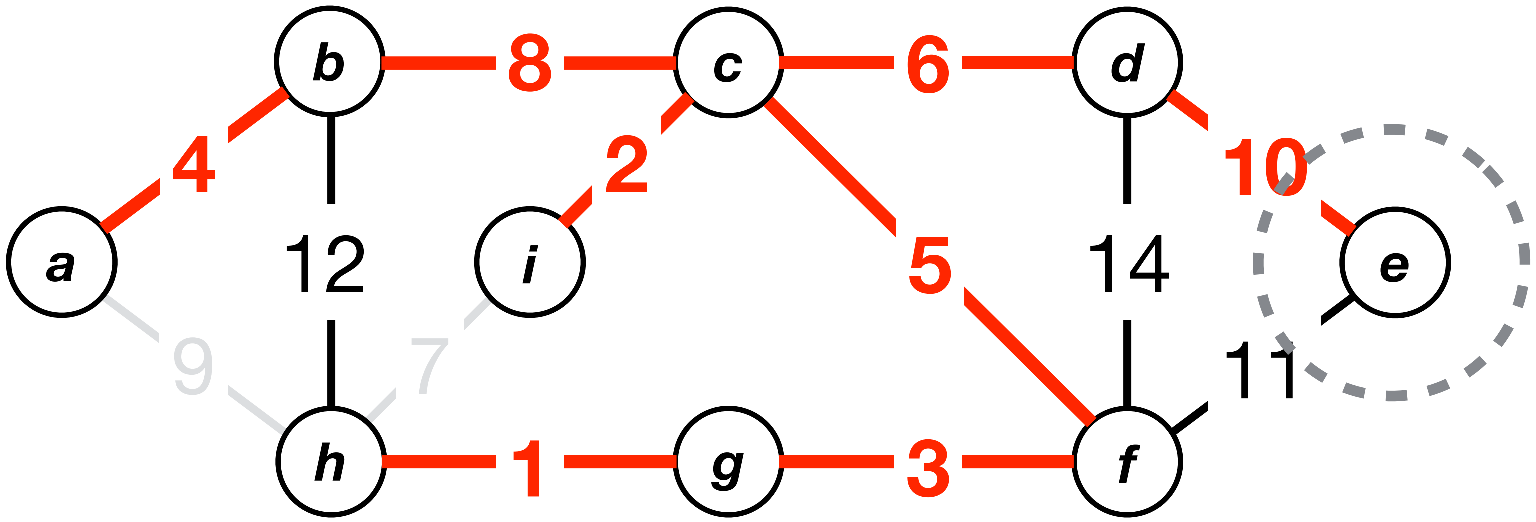
(h,g)	**1**
(i,c)	**2**
(g,f)	**3**
(a,b)	**4**
(c,f)	**5**
(c,d)	**6**
(h,i)	7
(b,c)	**8**
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(a,h) cannot be added to the solution because doing so would create a cycle.

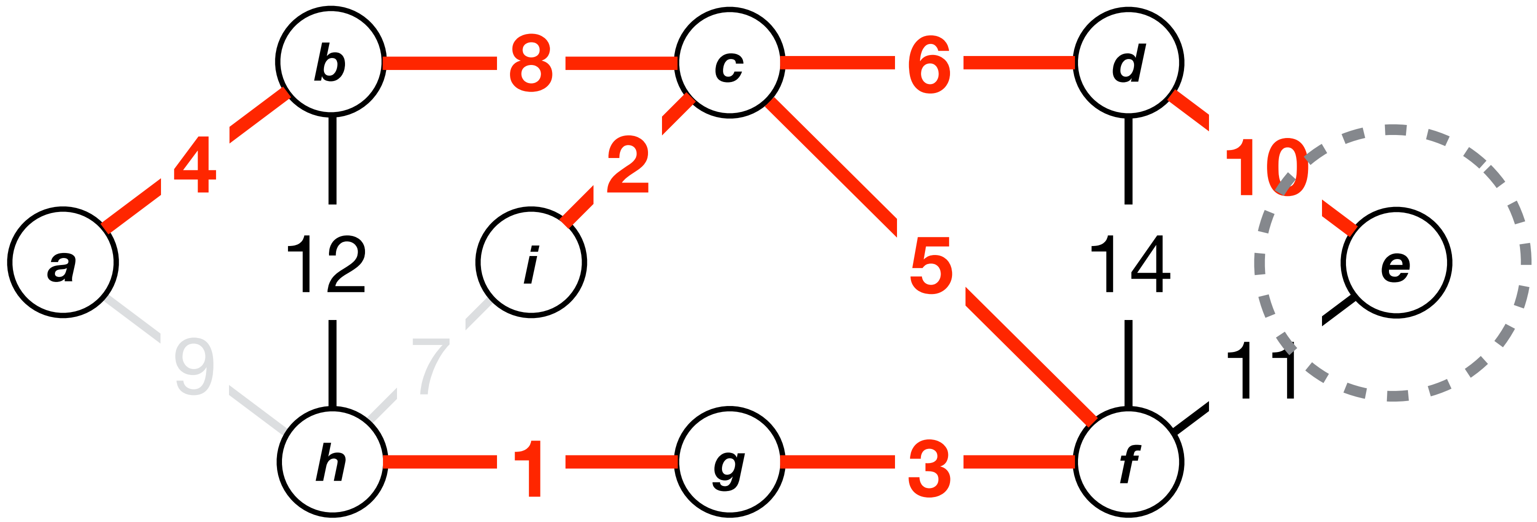
Correctness of Kruskal's Algorithm

(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

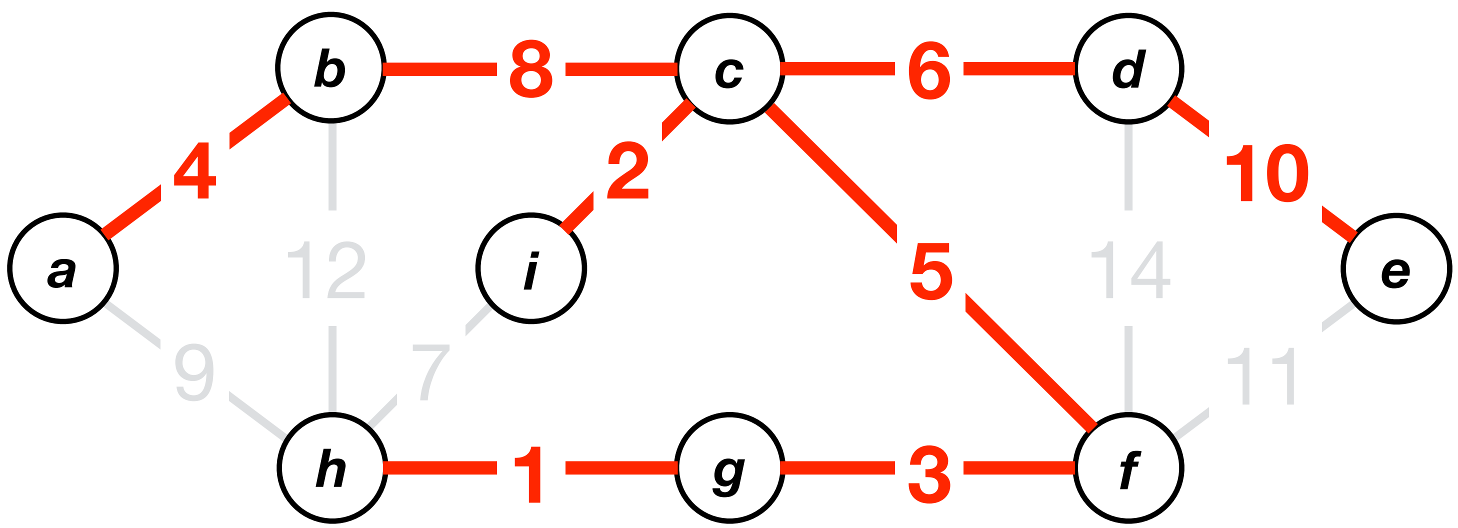
(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



(d,e) is the minimum weight edge going out from $S = \{ e \}$.

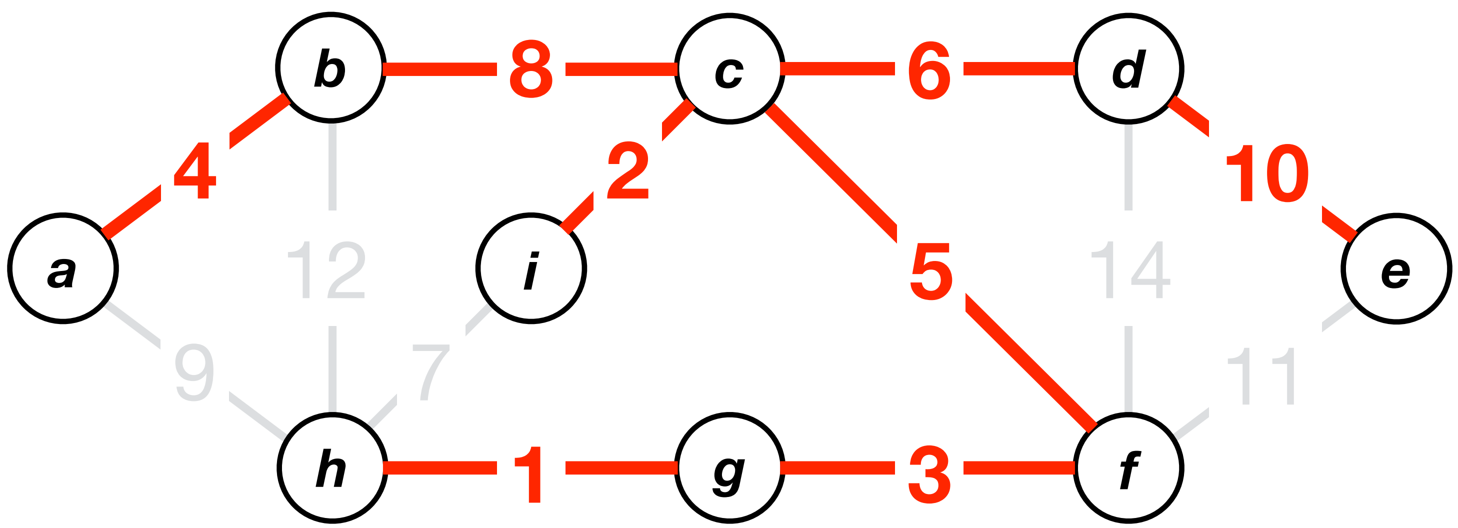
Correctness of Kruskal's Algorithm

(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



Correctness of Kruskal's Algorithm

(h,g)	1
(i,c)	2
(g,f)	3
(a,b)	4
(c,f)	5
(c,d)	6
(h,i)	7
(b,c)	8
(a,h)	9
(d,e)	10
(e,f)	11
(b,h)	12
(d,f)	14



We already have |V|-1 edges. So adding any of the remaining edges would create a cycle.

Correctness of Kruskal's Algorithm

Fact 1. The algorithm adds an edge only if it is in the MST.

Key question: Why we can always find a subset of vertices S such that e is the minimum weight outgoing edge from S?

- Throughout the algorithm, we have a set of subtrees.
- We add an edge *e* only if it does not create any cycle.
- So the two endpoints of *e* cannot be in the same subtree. That is, *e* connects two different subtrees, say T_1 and T_2
- So *e* is an outgoing edges of T_1. Further, the algorithm has not processed any outgoing edges of T_1 when we add *e*.
- Choosing $S = T_1$ suffices because by our choice of *e*, it must have the minimum edge weight among them.

Correctness of Kruskal's Algorithm

Fact 2. Each edge in the MST will be added by the algorithm.

Proof: Consider an edge *e* in the MST.

- Since the algorithm checks all edges before it stops, it must have checked edge *e*.
- Since the only edges added by the algorithm are those in the MST, *e* cannot create a cycle.
- So the algorithm would have added edge *e* to the solution.

Implementing Kruskal's Algorithm
(A Data Structure for Disjoint Sets)

Implementing Kruskal's Algorithm

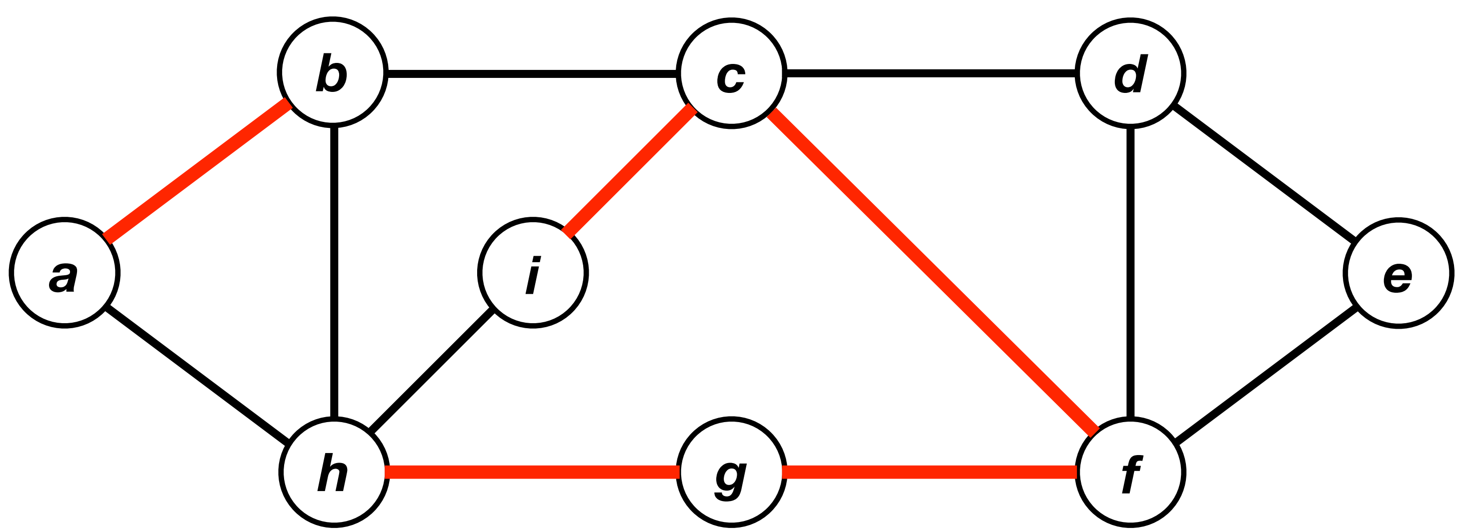
1) Start from an empty forest T.
2) **for** all edges e in ascending order of weights **:**
3) insert edge e in T unless doing so would create a cycle.

- Note that $log\ |E| = O(log\ |V|)$ because $|E| = O(|V|^2)$.
- Sorting the edges in ascending order takes $O(|E|\ log\ |E|) = O(|E|\ log\ |V|)$ time.
- The for loop has $|E|$ iterations.
- **Key questions:**
  - How to implement an iteration of the for loop?
  - How to determine if adding an edge creates a cycle?

Implementing Kruskal's Algorithm

Observation: During the execution of the algorithm, the set of edges added to the solution (red edges) forms a set of disjoint sub-trees of the MST.

- To determine if adding an edge creates a cycle is the same as to determine if its end points are in the same sub-tree.

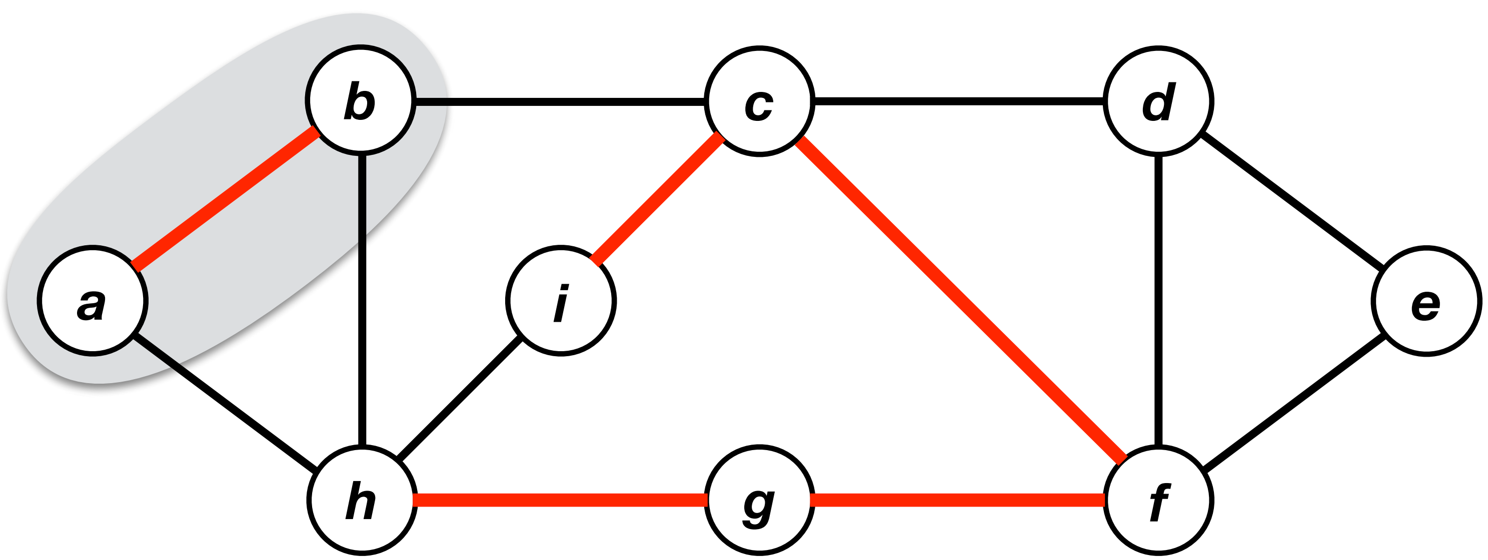


In this step, there are 4 disjoint sub-trees.

Implementing Kruskal's Algorithm

Observation: During the execution of the algorithm, the set of edges added to the solution (red edges) forms a set of disjoint sub-trees of the MST.

- To determine if adding an edge creates a cycle is the same as to determine if its end points are in the same sub-tree.

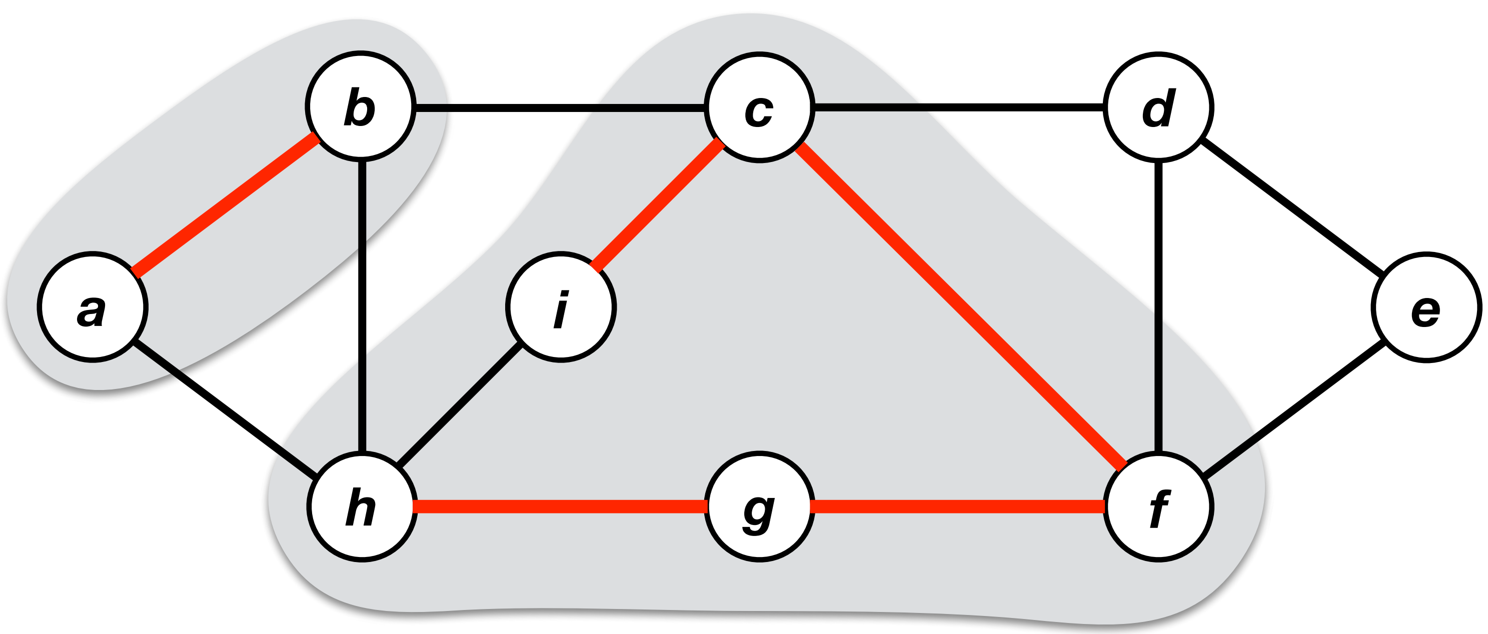


In this step, there are 4 disjoint sub-trees.

Implementing Kruskal's Algorithm

Observation: During the execution of the algorithm, the set of edges added to the solution (red edges) forms a set of disjoint sub-trees of the MST.

- To determine if adding an edge creates a cycle is the same as to determine if its end points are in the same sub-tree.

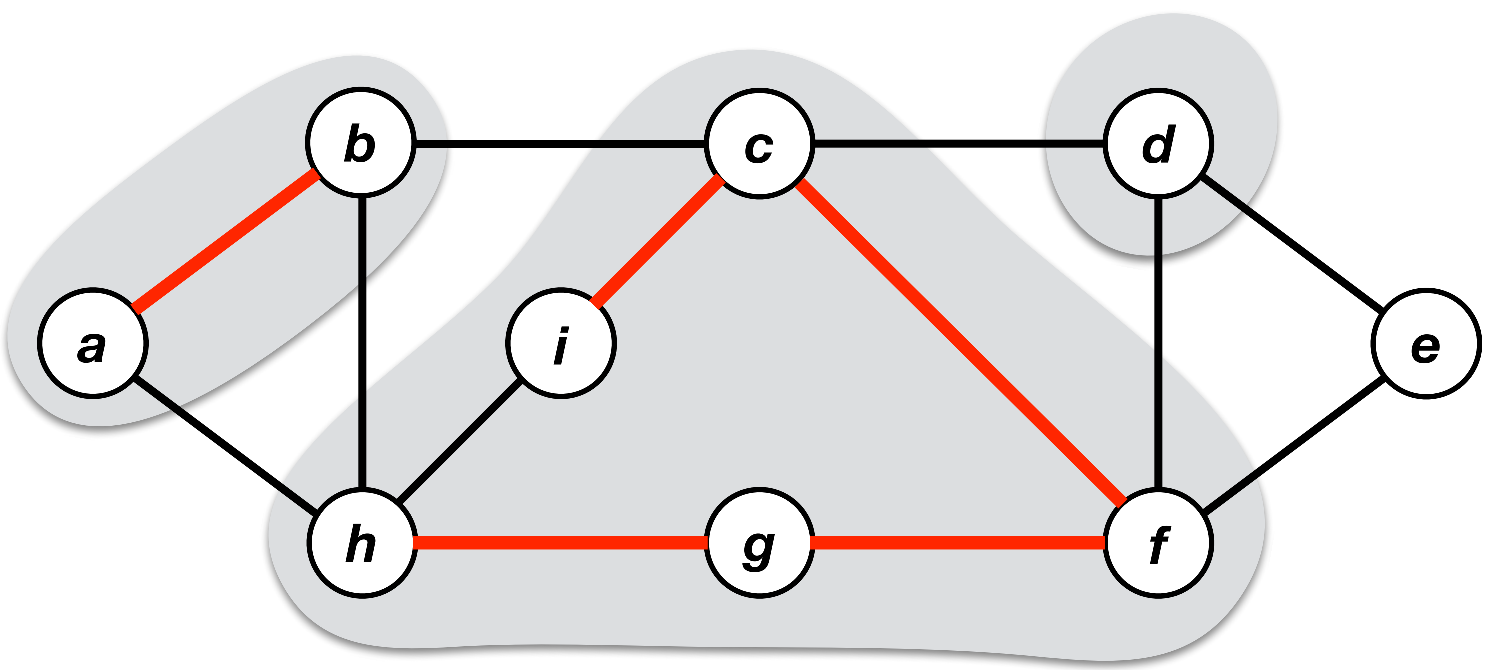


In this step, there are 4 disjoint sub-trees.

Implementing Kruskal's Algorithm

Observation: During the execution of the algorithm, the set of edges added to the solution (red edges) forms a set of disjoint sub-trees of the MST.

- To determine if adding an edge creates a cycle is the same as to determine if its end points are in the same sub-tree.

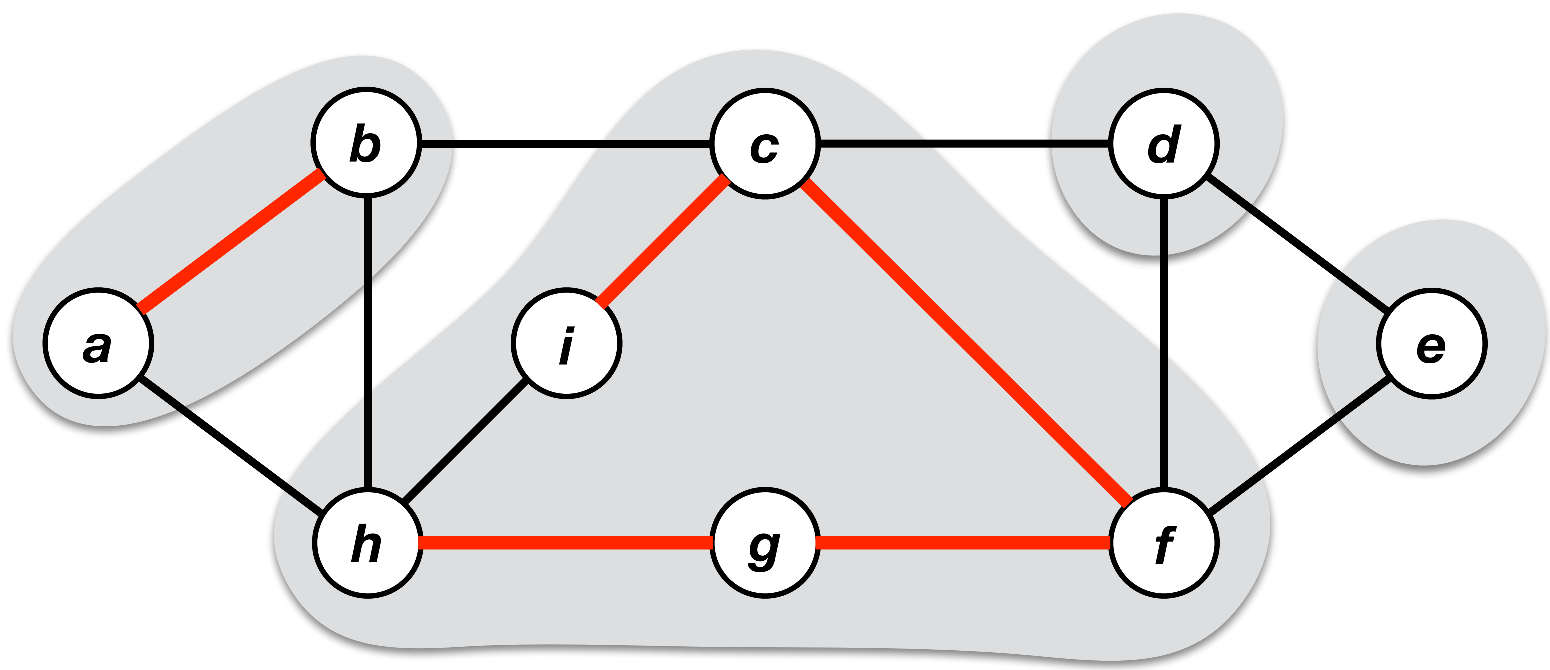


In this step, there are 4 disjoint sub-trees.

Implementing Kruskal's Algorithm

Observation: During the execution of the algorithm, the set of edges added to the solution (red edges) forms a set of disjoint sub-trees of the MST.

- To determine if adding an edge creates a cycle is the same as to determine if its end points are in the same sub-tree.

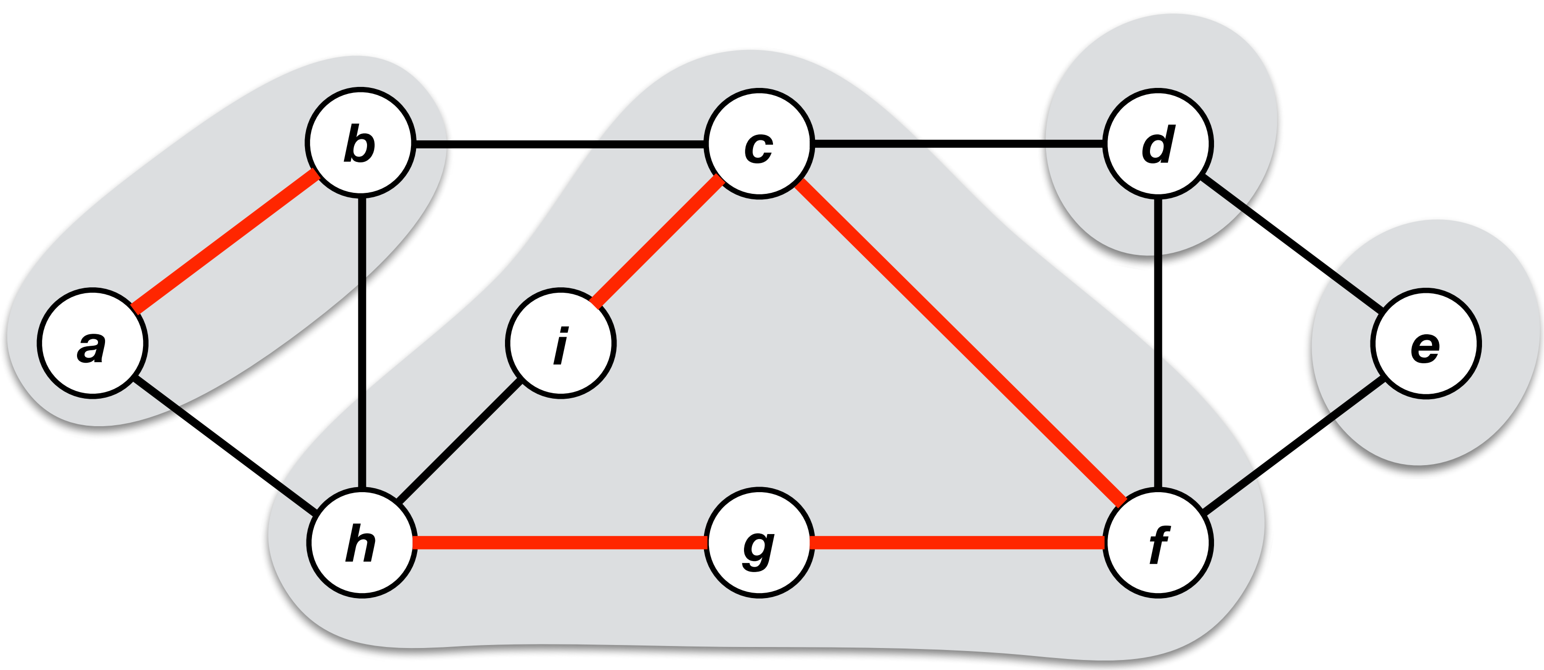


In this step, there are 4 disjoint sub-trees.

Implementing Kruskal's Algorithm

Idea: Design a data structure that remembers the current set of disjoint sub-trees such that we can efficiently

1) find the subtree that a vertex, say, *c*, belongs to, through a procedure **find-set**(*c*); and
2) merge two subtrees through **union**(**find-set**(*c*), **find-set**(*d*)).



Implementing Kruskal's Algorithm

1) **initialize** $T = \{\ \}$.
2) **for** all vertices v **: initialize** a sub-tree for v via **make-set**(v).
3) **for** all edges (u, v) in ascending order of weights **:**
4)   **if find-set**(u) $\neq$ **find-set**(v) :
5)   add (u, v) to T;
6)   **union**(**find-set**(u), **find-set**(v)).

Running Time:

- # of **make-set**: $|V|$; # of **find-set**: $2|E|$; # of **union**: $|V|-1$.
- Suppose the data structure implements these subroutine in $O(\log |V|)$ time, then the total running time is $O(|E| \log |V|)$.

A Data Structure of Disjoint Sets

We need to maintain a collection of sets from *n* elements (vertices). Our data structure must support the followings:

- Given any two elements *x*, *y*, we need to determine whether **find-set**(*x*) = **find-set**(*y*), i.e., to determine whether *x* and *y* belongs to the same set.
- Given any two sets in the current collection, we need to replace these two sets by its **union**.

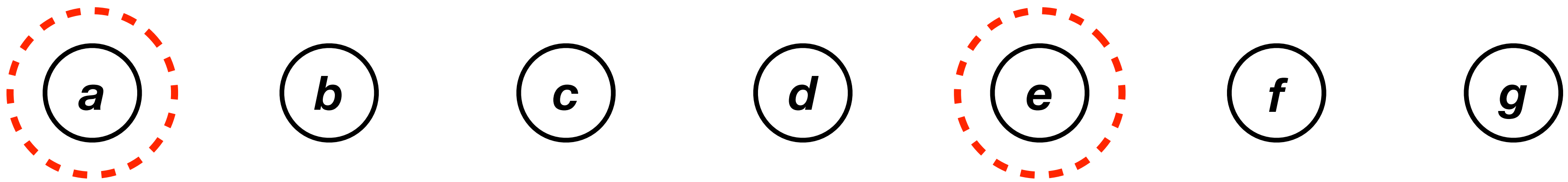
High-Level Approach

Idea: Maintain a tree for the vertices in each set and name each set after the root vertex.

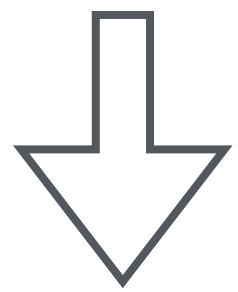
- For **find-set**(x), we just trace back to the root.
- To **union** two sets, we append the root of one set to be a child of the root of the other set.

Sample Run

{a} {b} {c} {d} {e} {f} {g}

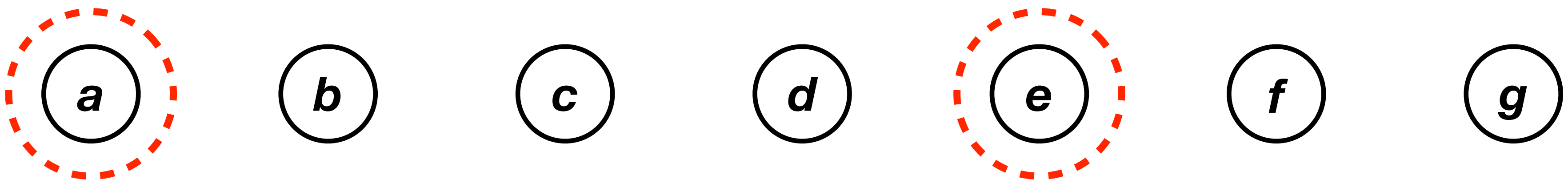


union(**find**(*a*), **find**(*e*))

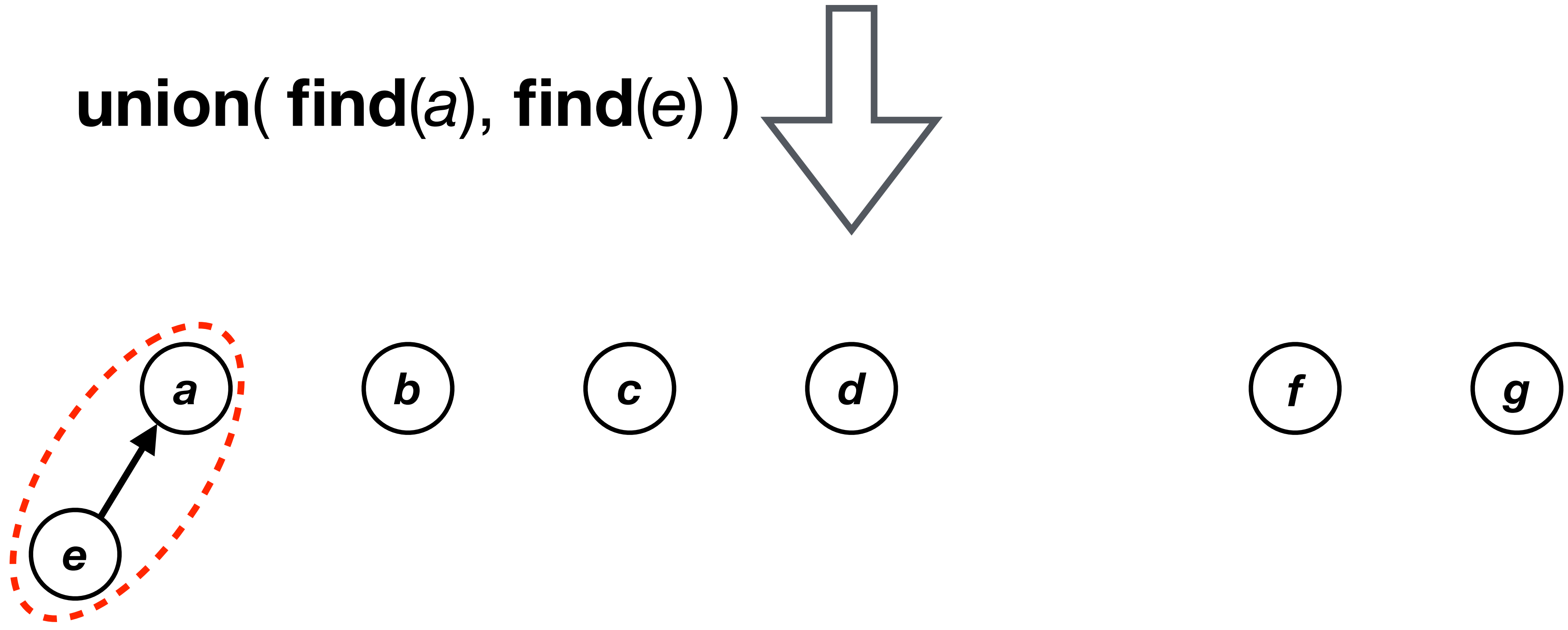


Sample Run

{a} {b} {c} {d} {e} {f} {g}

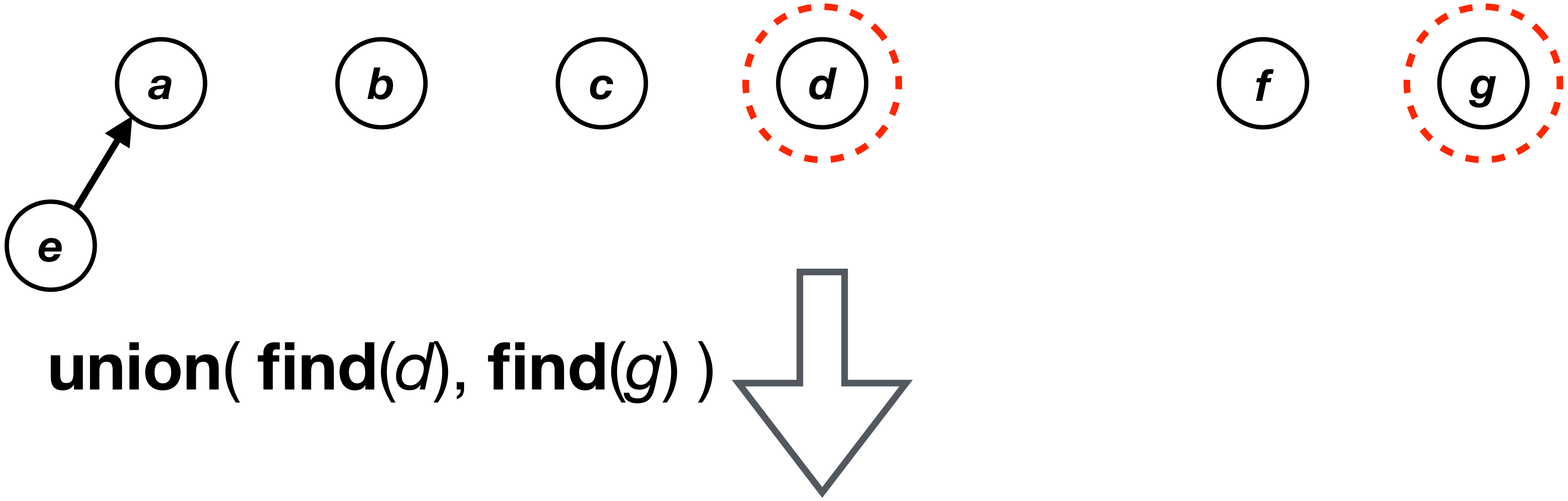


union(find(*a***), find(***e***))**



Sample Run

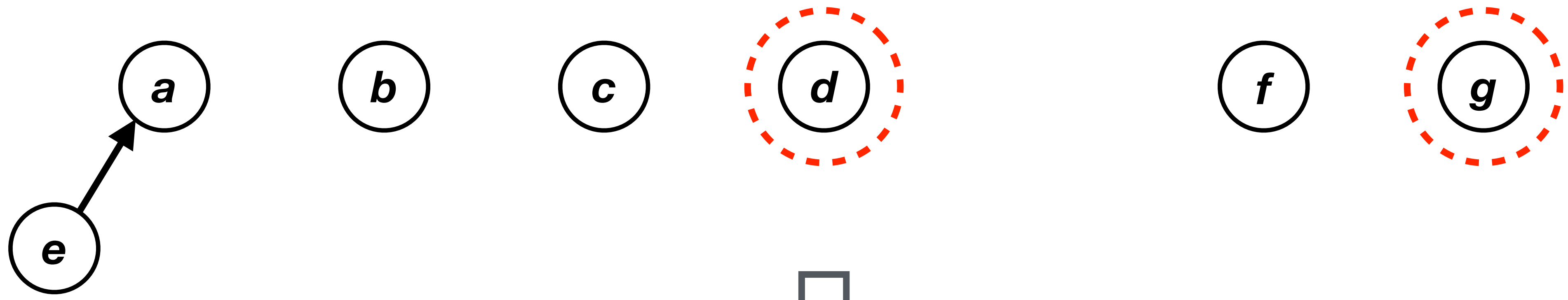
{a, e} {b} {c} {d} {f} {g}



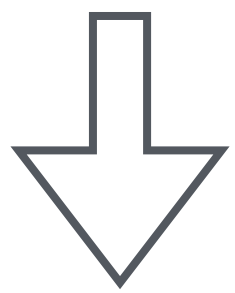
union(**find**(*d*), **find**(*g*))

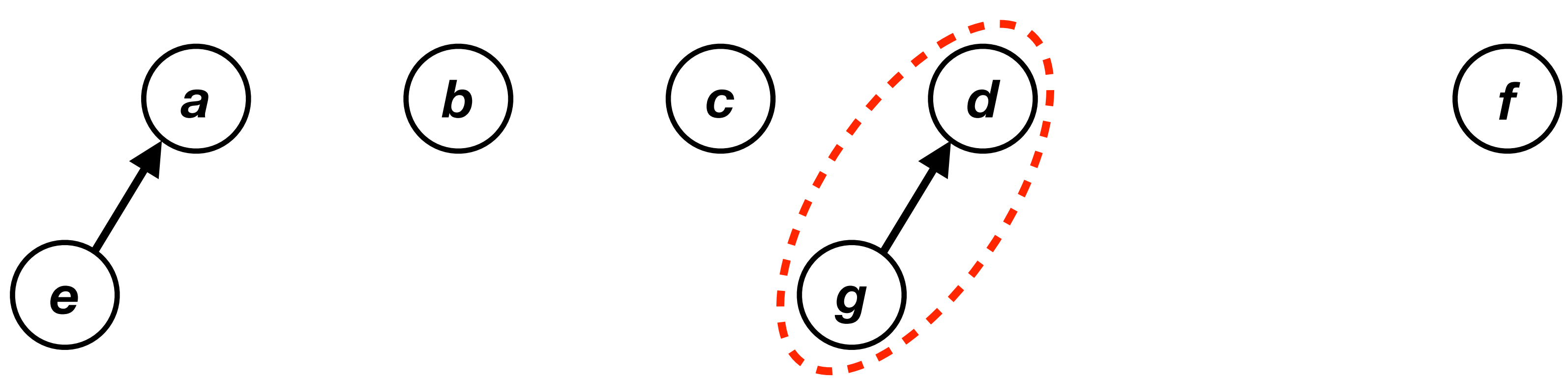
Sample Run

{a, e} {b} {c} {d} {f} {g}



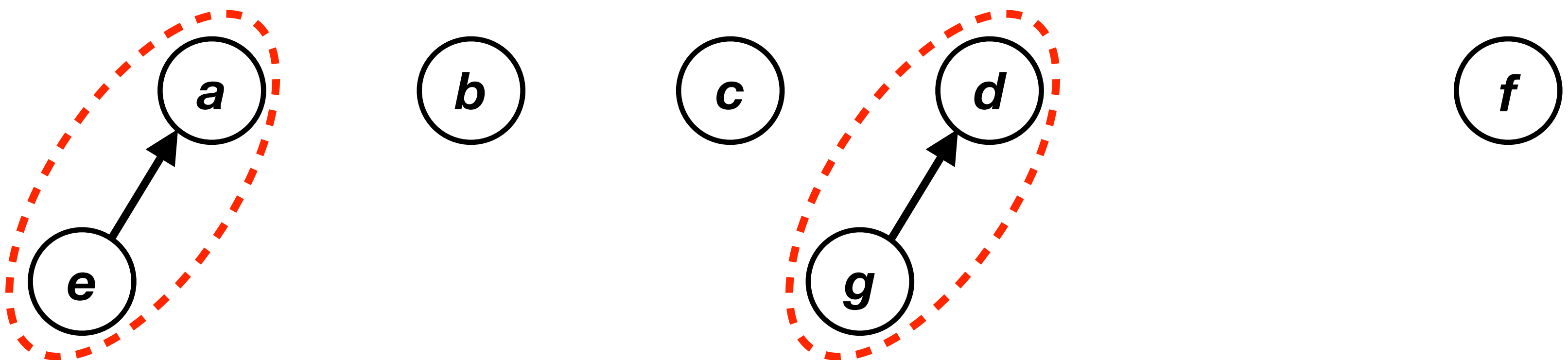
union(**find**(*d*), **find**(*g*))



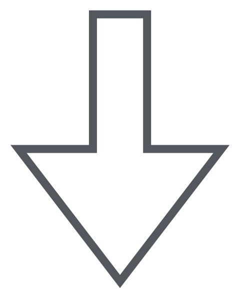


Sample Run

{a, e} {b} {c} {d, g} {f}

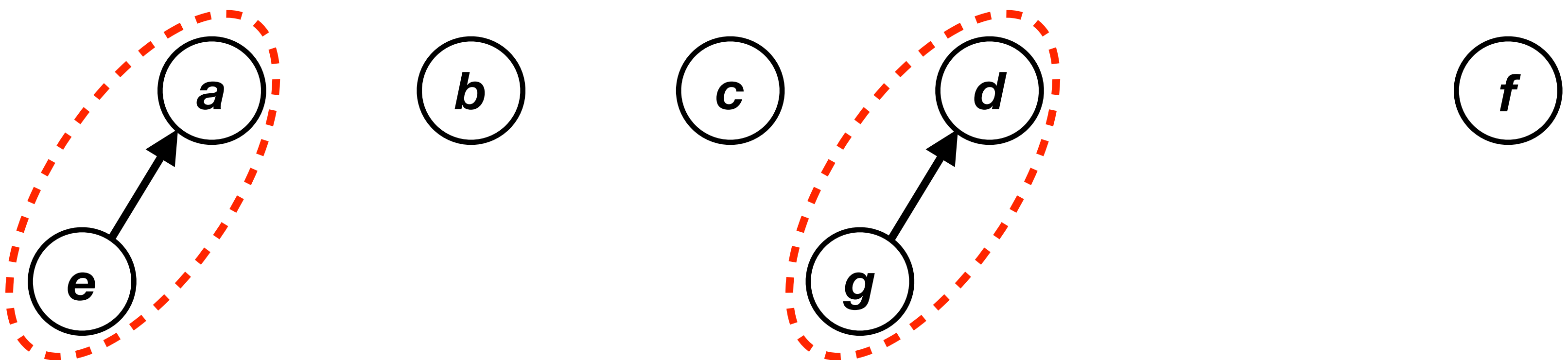


union(**find**(*e*), **find**(*g*))

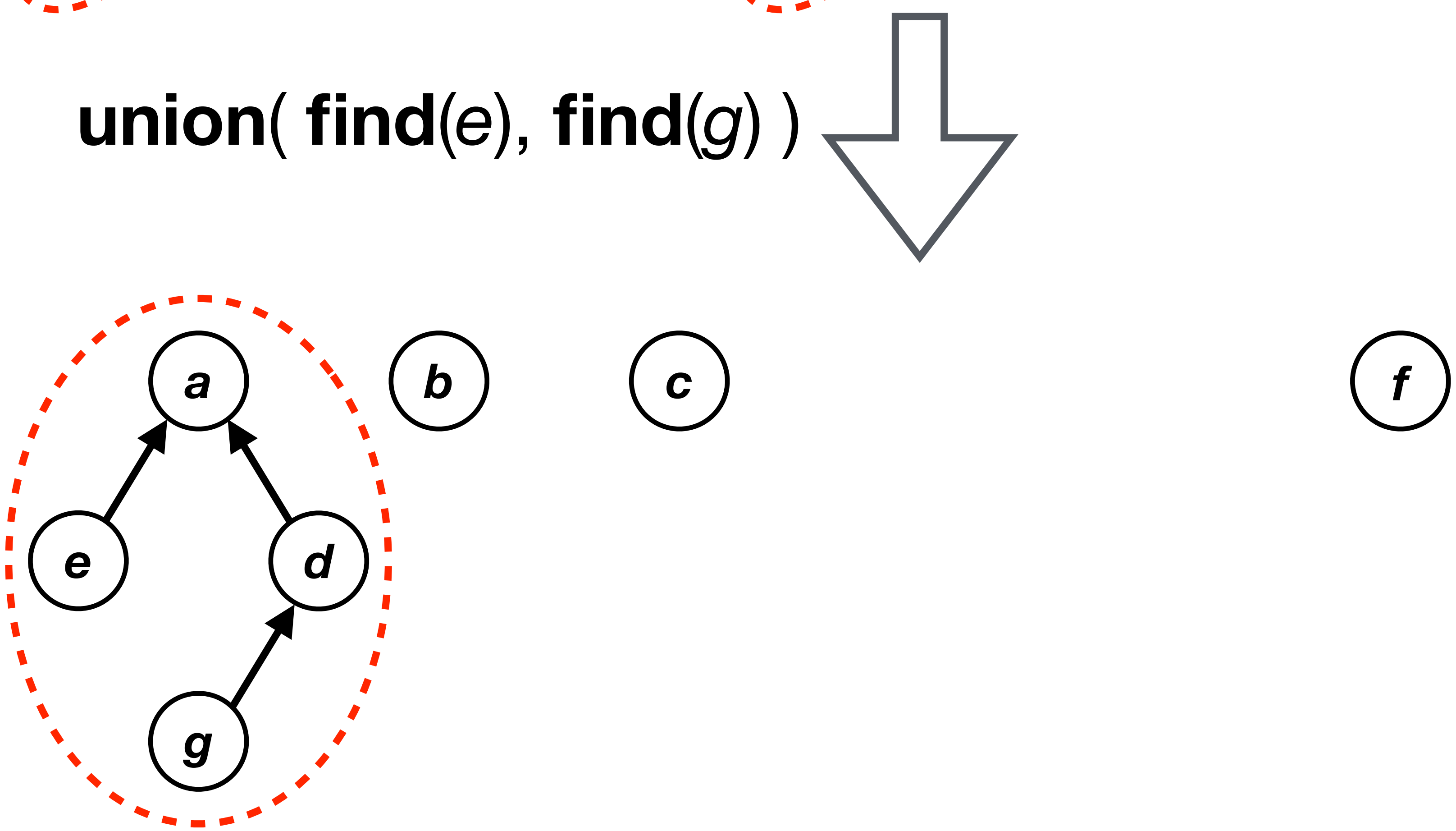


Sample Run

{a, e} {b} {c} {d, g} {f}

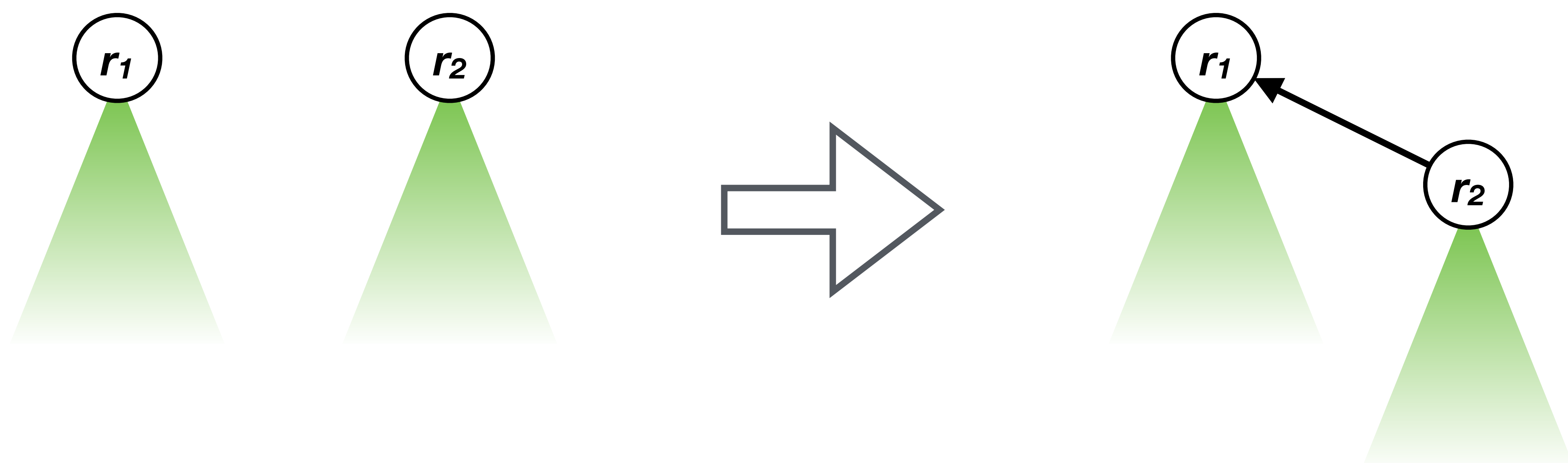


union(find(*e*), find(*g*))



Summary

- To execute **find-set**(x), we traverse the parent pointers from x up to the root, and the root is used as the name of the set.
- So, **find-set**(x) = **find-set**(y) if and only if the root returned by **find-set**(x) is equal to that returned by **find-set**(y).
- To execute **union**(r_1, r_2), we make a root to be the child of the other root.

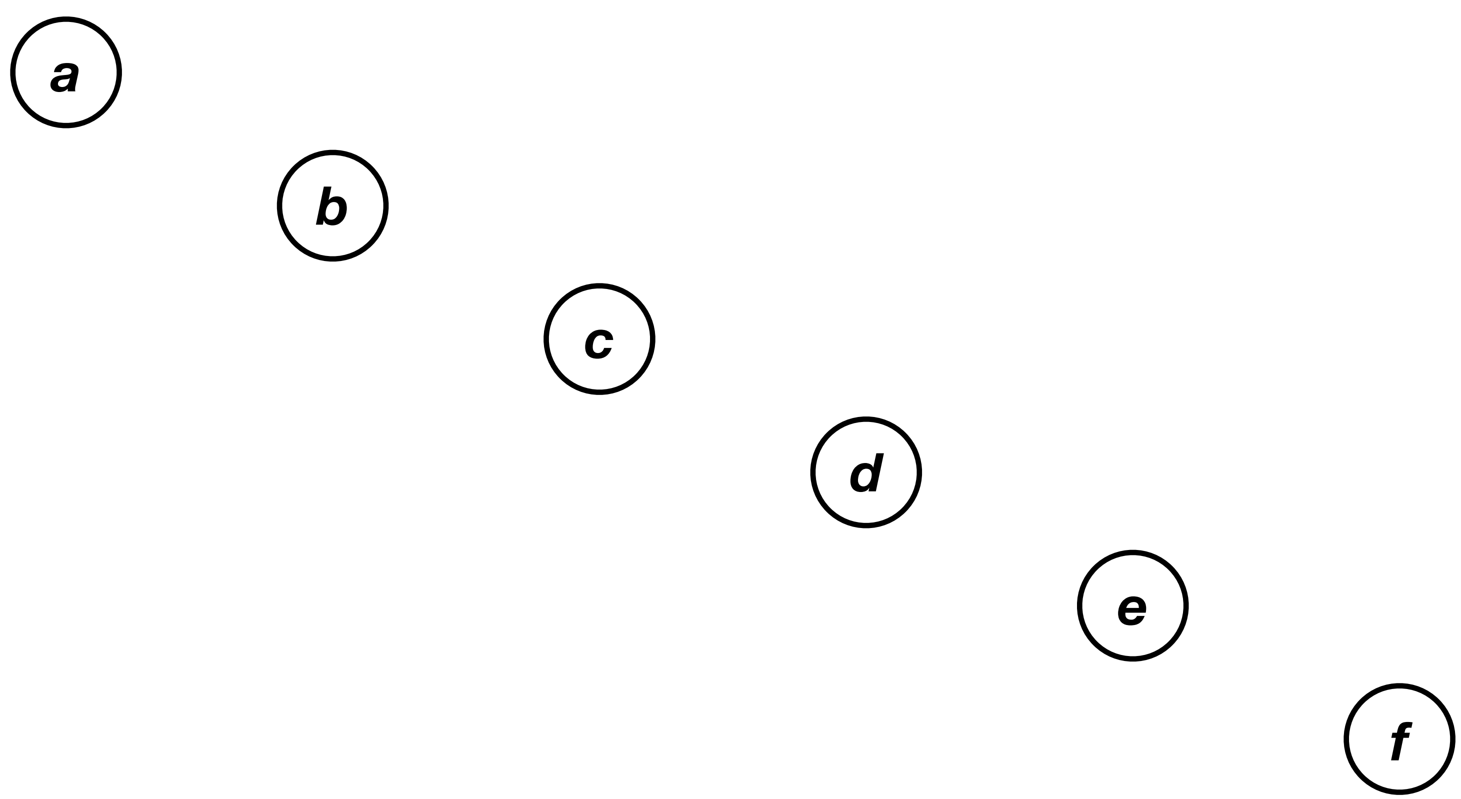


Question: Which vertex shall be the new root?

Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x):
  the height of the tree containing *x*, which can be O(n).

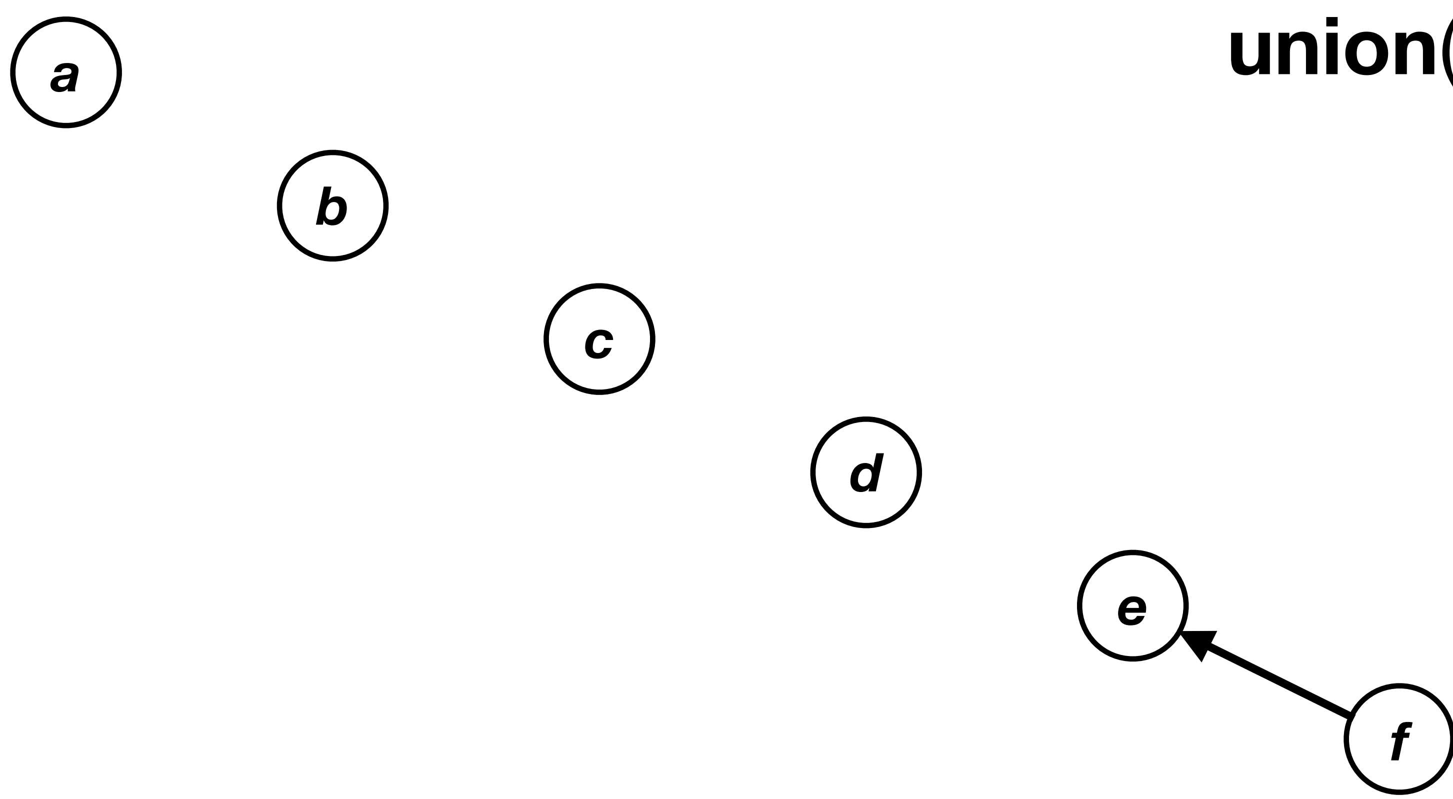


Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x):
  the height of the tree containing *x*, which can be O(n).

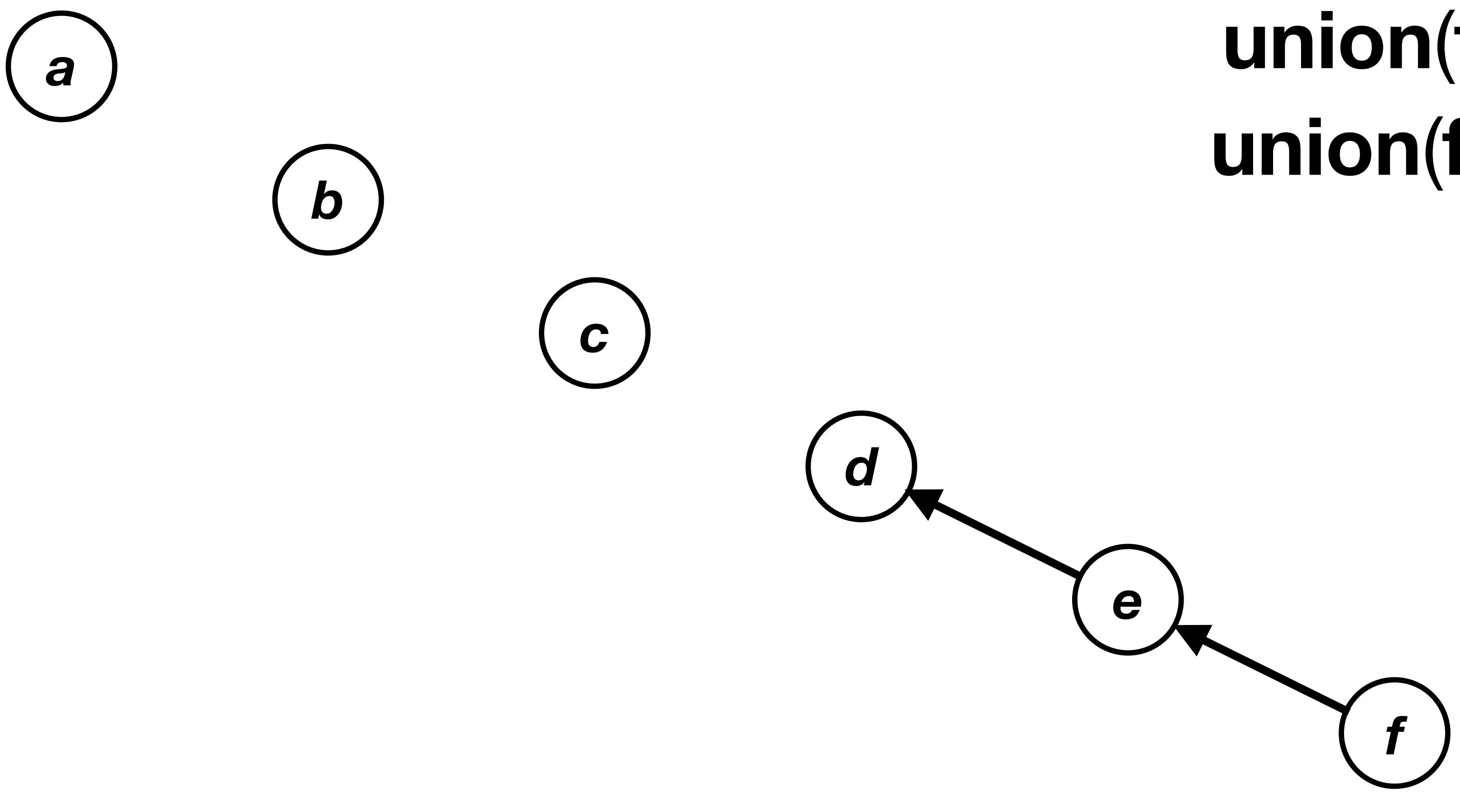
union(**find-set**(e), **find-set**(f))



Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x): the height of the tree containing *x*, which can be O(n).

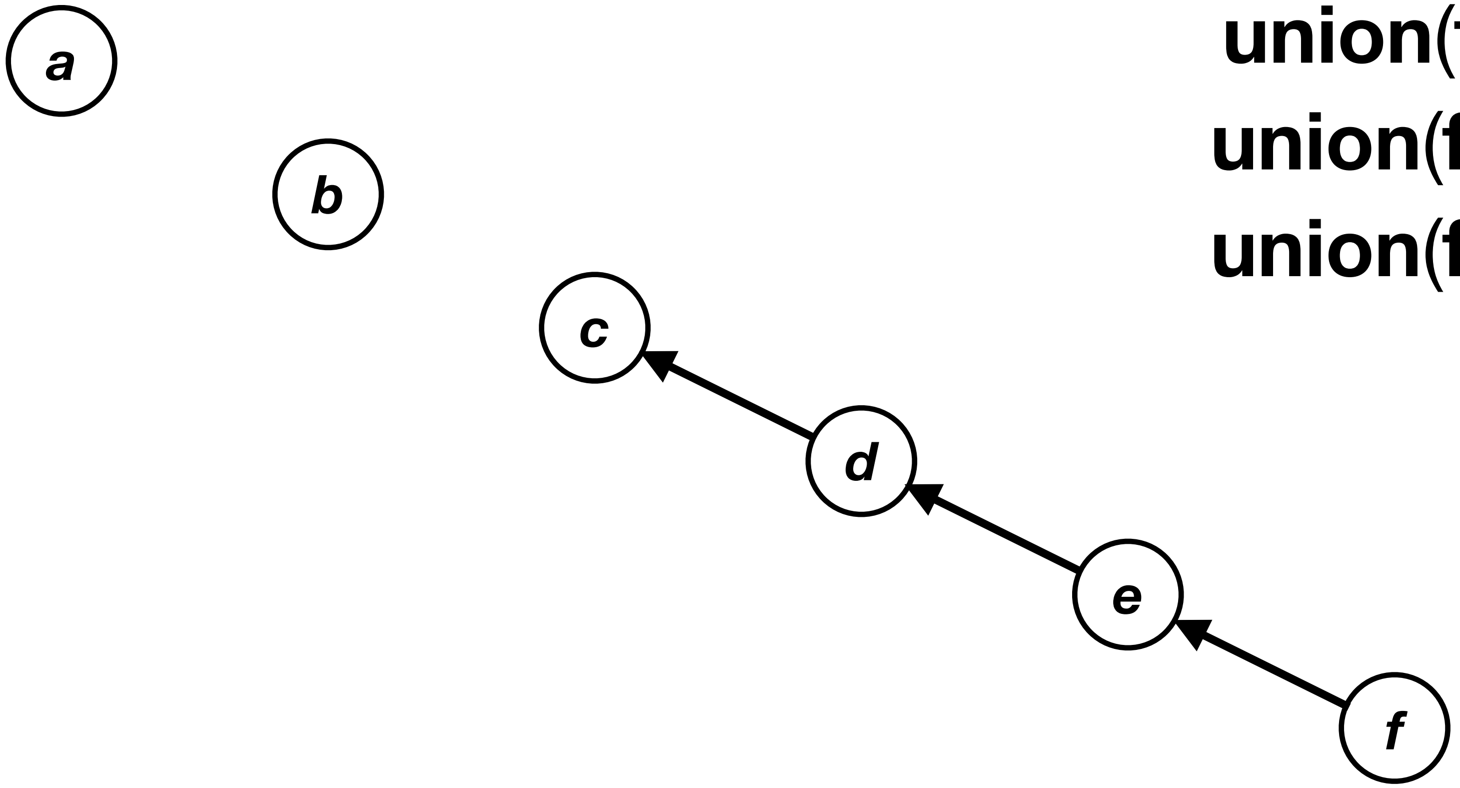


union(**find-set**(e), **find-set**(f))
union(**find-set**(d), **find-set**(e))

Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x): the height of the tree containing *x*, which can be O(n).

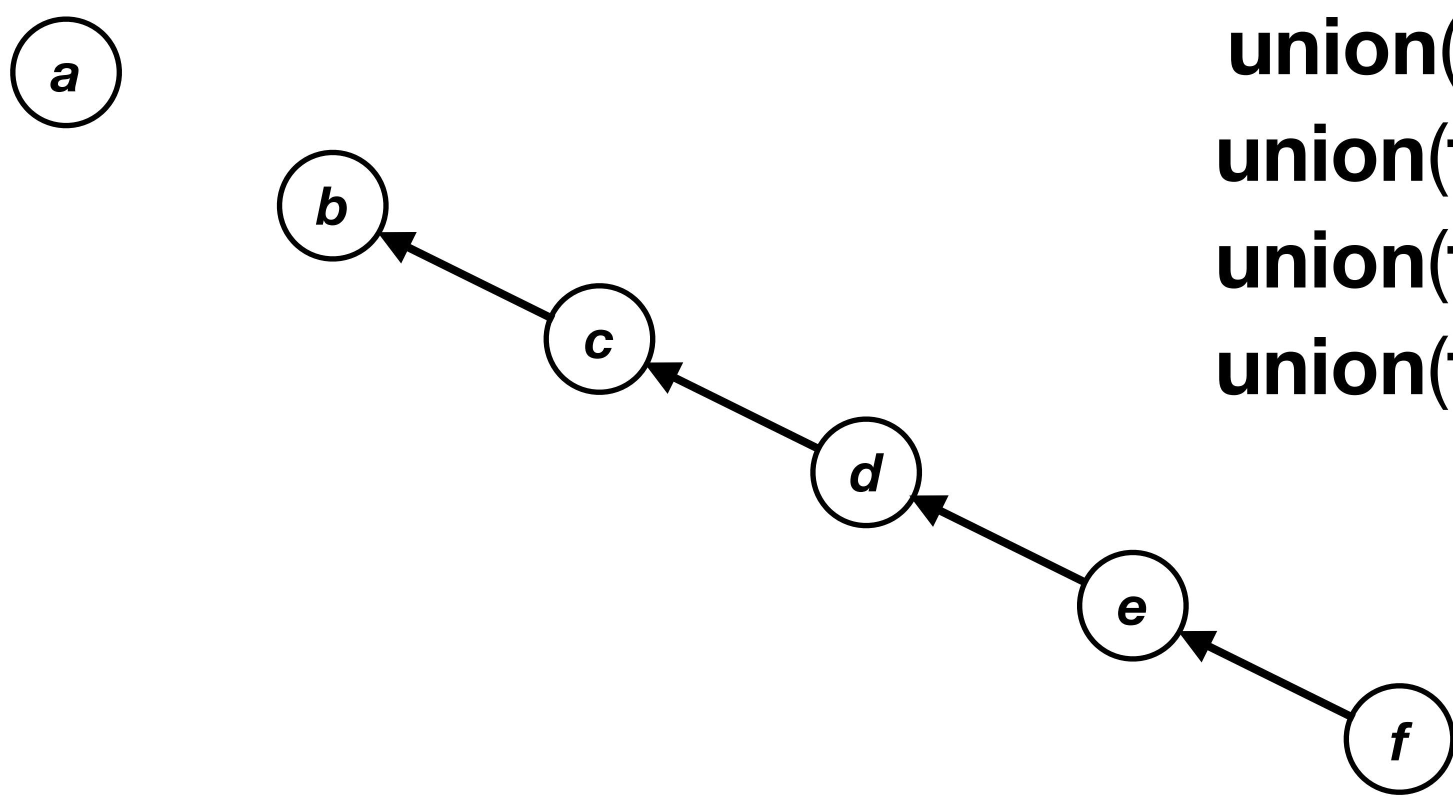


union(**find-set**(e), **find-set**(f))
union(**find-set**(d), **find-set**(e))
union(**find-set**(c), **find-set**(d))

Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x):
  the height of the tree containing *x*, which can be O(n).

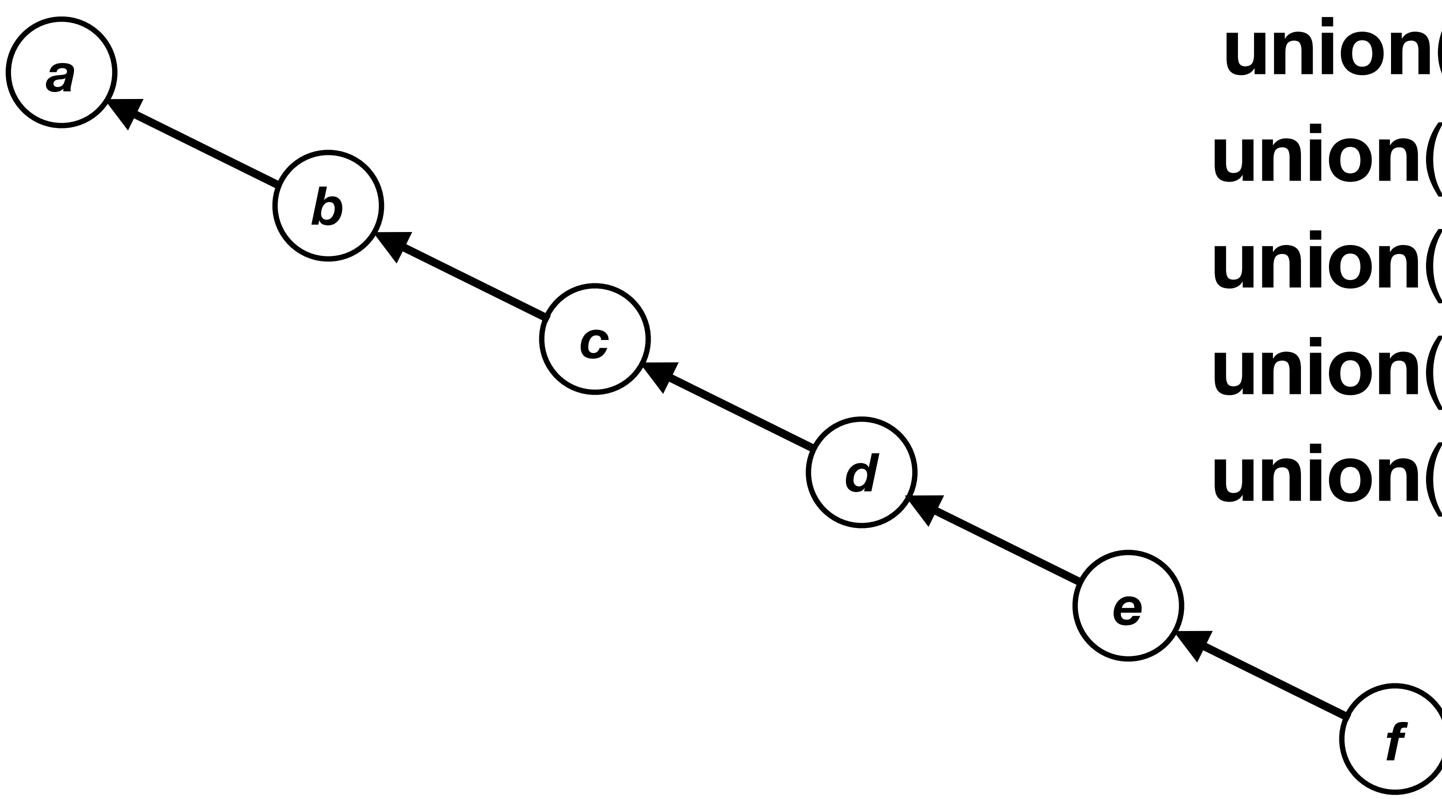


union(**find-set**(e), **find-set**(f))
union(**find-set**(d), **find-set**(e))
union(**find-set**(c), **find-set**(d))
union(**find-set**(b), **find-set**(c))

Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x):
  the height of the tree containing *x*, which can be O(n).

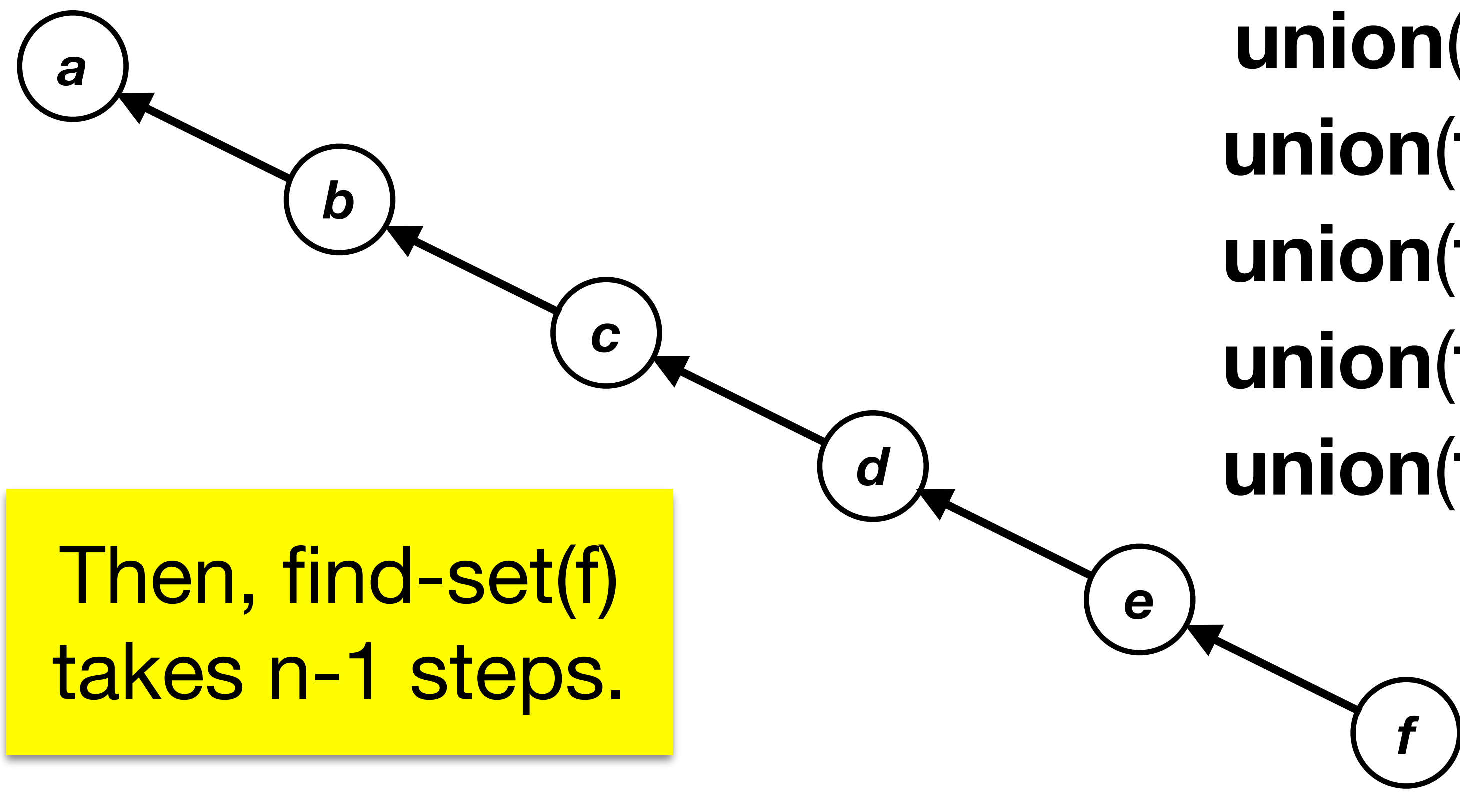


union(**find-set**(e), **find-set**(f))
union(**find-set**(d), **find-set**(e))
union(**find-set**(c), **find-set**(d))
union(**find-set**(b), **find-set**(c))
union(**find-set**(a), **find-set**(b))

Running Time Analysis

To answer the question, we need to first understand how the running time of **find-set** and **union** depends on the structure of the trees that we maintain.

- Running time for a **union** operation: *O(1)*.
- Running time for a **find-set**(x):
  the height of the tree containing *x*, which can be O(n).

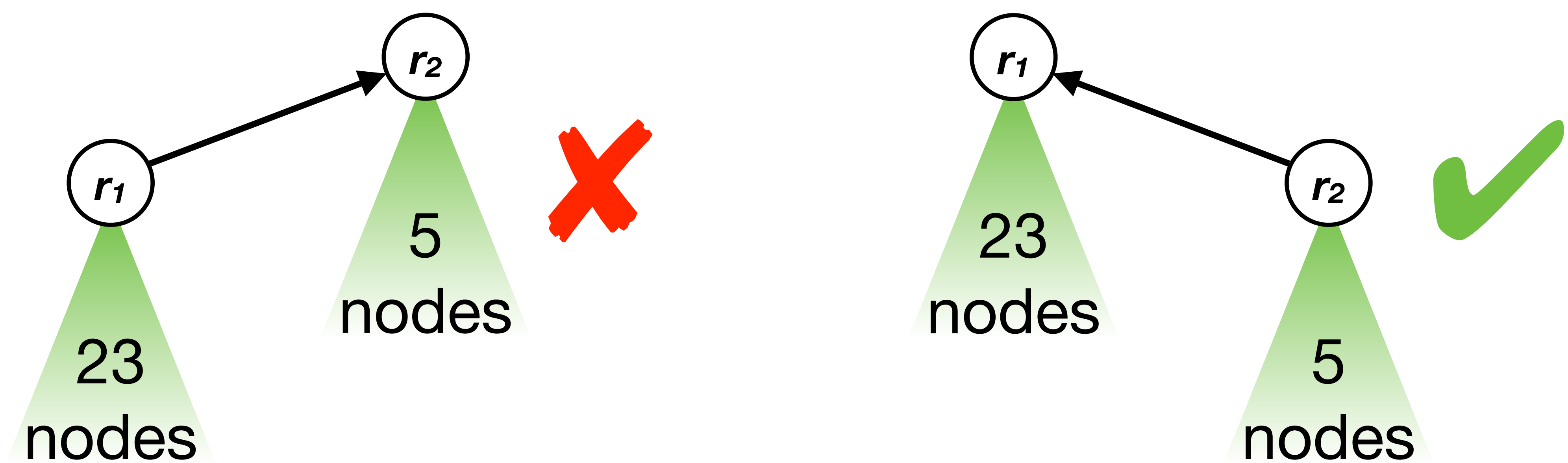


union(**find-set**(e), **find-set**(f))
union(**find-set**(d), **find-set**(e))
union(**find-set**(c), **find-set**(d))
union(**find-set**(b), **find-set**(c))
union(**find-set**(a), **find-set**(b))

Then, find-set(f) takes n-1 steps.

Can we guarantee small height?

- If we could implement **union** such that the trees have small height, then **find-set** would have small running time.
- **How?** One way to do it is the union-by-size heuristic:
  - To **union**(s_1, s_2), we make the tree with smaller size the child of the one with larger size.
  - Break ties arbitrarily.
  - **Example:**

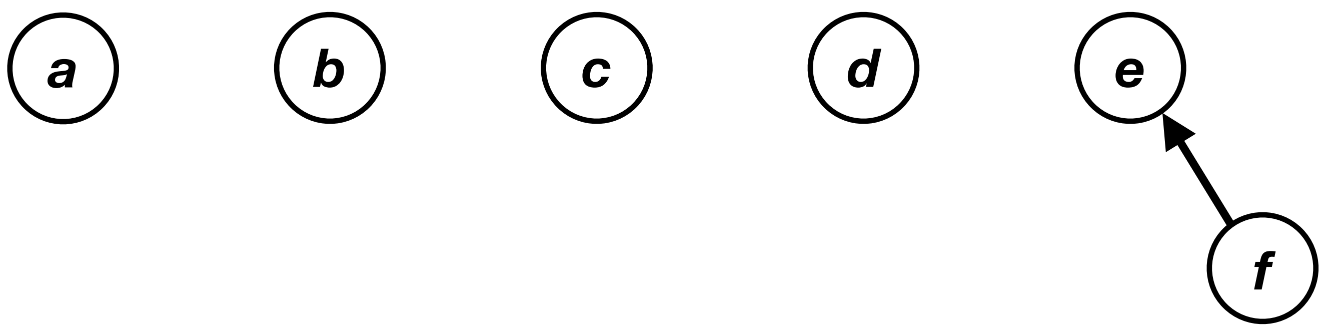


Revisiting the *O(n)*-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))

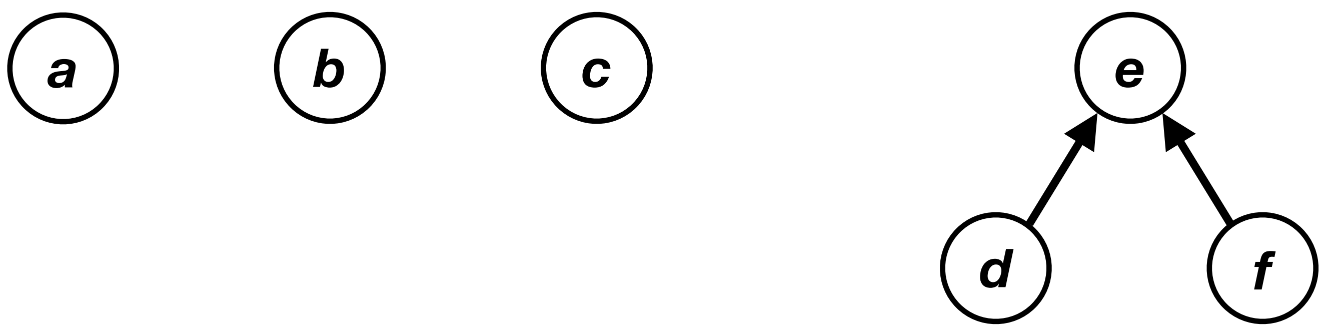
Revisiting the O(n)-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))



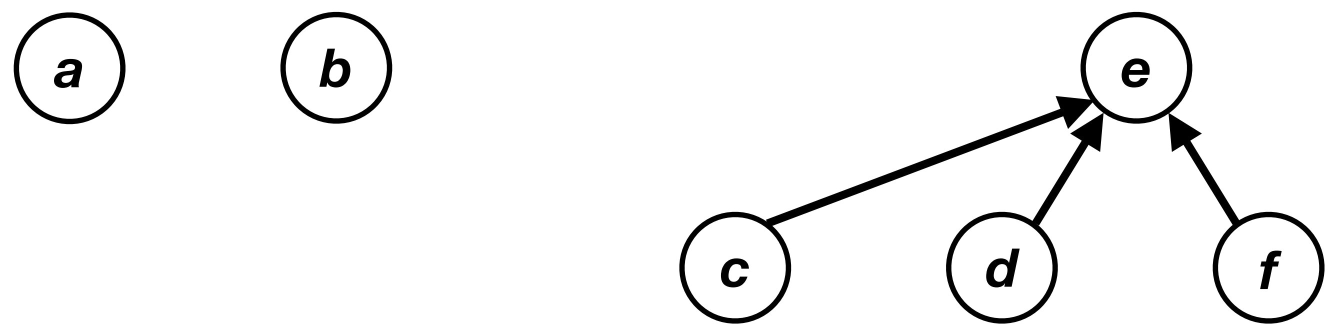
Revisiting the O(n)-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))



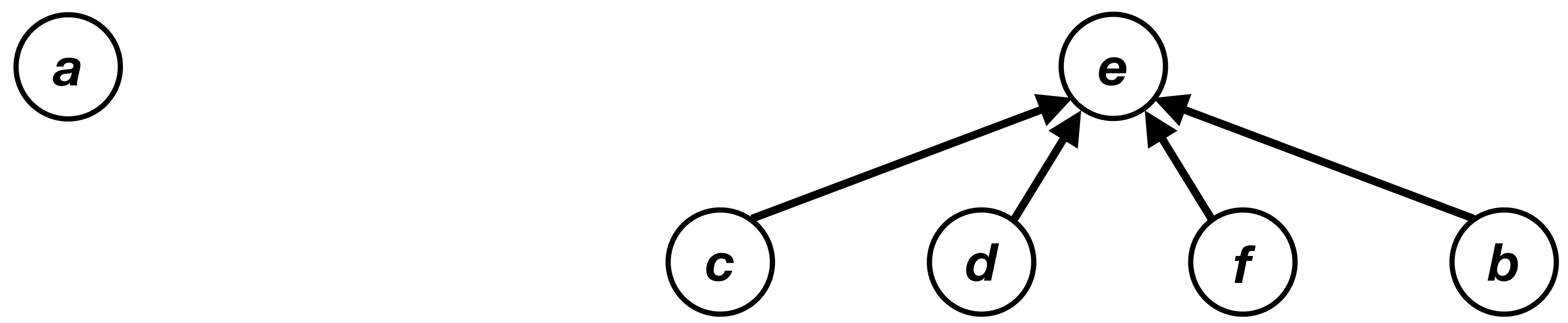
Revisiting the O(n)-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))



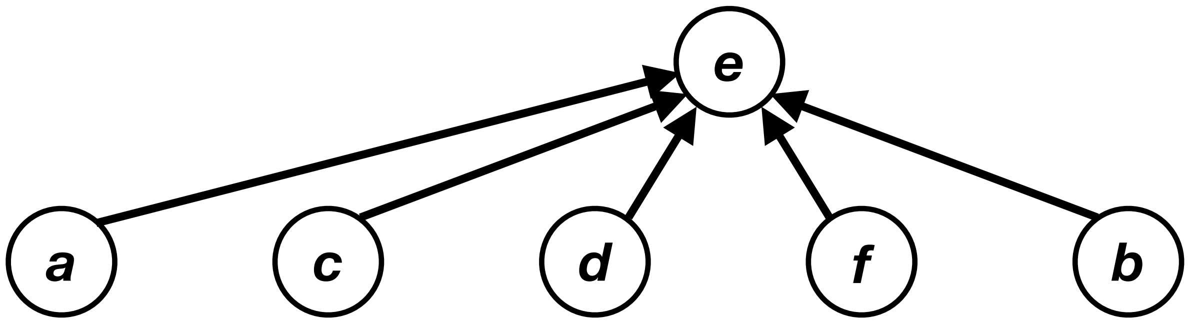
Revisiting the O(n)-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))



Revisiting the O(n)-Time Example

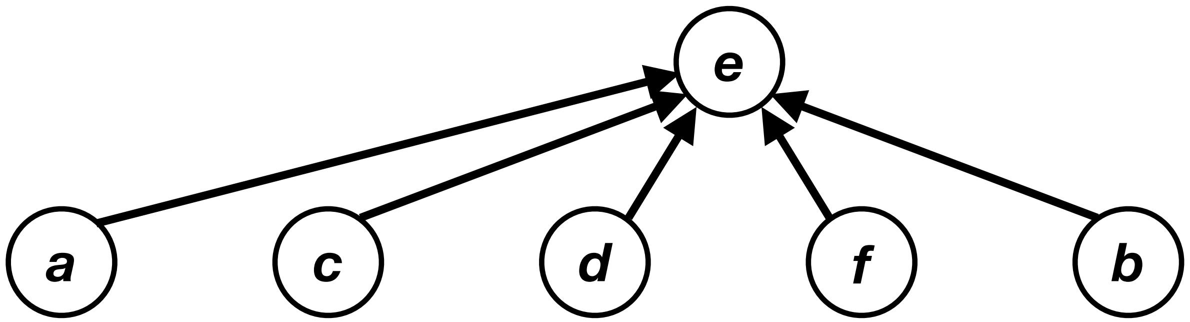
union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)), **union**(**find-set**(a), **find-set**(b))



Revisiting the O(n)-Time Example

union(**find-set**(e), **find-set**(f)), **union**(**find-set**(d), **find-set**(e)), **union**(**find-set**(c), **find-set**(d)), **union**(**find-set**(b), **find-set**(c)),

union(**find-set**(a), **find-set**(b))



The union-by-size heuristic gives a tree with height 1!

Analyzing the Union-By-Size Heuristic

Claim. If we use union-by-size, then any tree with height h must have at least 2^h nodes.

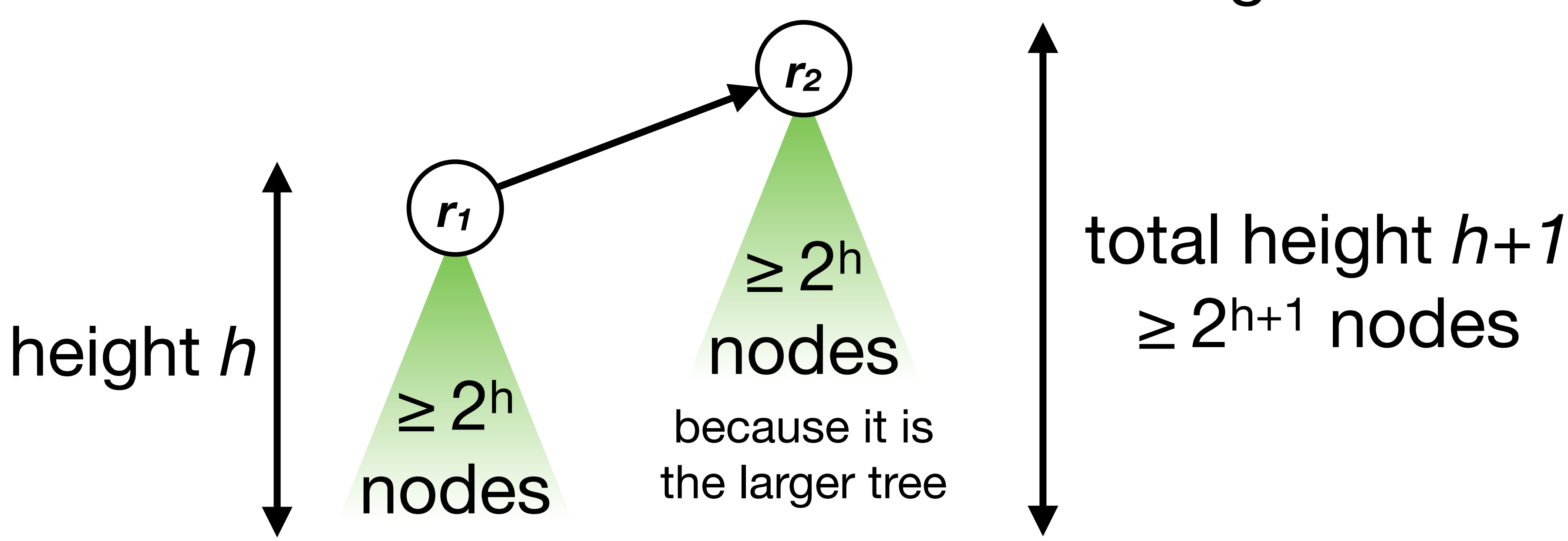
Proof (by induction).

Base case: h = 0 is true as any tree has at least 1 node.

Inductive hypothesis: The claim is true for trees with height h.

Inductive step: We will show it is true for trees with height $h+1$.

- **Key observation:** We get a tree with height $h+1$ only when we union two trees and one of them has hight h.



Analyzing the Union-By-Size Heuristic

Claim. If we use union-by-size, then any tree with height h must have at least 2^h nodes.

Corollary. If we use union-by-size, then any tree must have height $h \leq \log n$.

Proof.

- Since there are only n nodes, the size of any tree is $\leq n$.
- Together with the above claim, we have $n \geq$ tree size $\geq 2^h$.
- Equivalently, we have $\log n \geq \log(\text{tree size}) \geq h$.

Summary

- If we implement the Disjoint Sets data structure using the union-by-size heuristic, then
  - **find-set** runs in time *O(log n)*;
  - **union** runs in time *O(1)*.
- Substitute these bounds in our analysis of the running time, we conclude that Kruskal's algorithm runs in *O(m log n)* time.