

Kruskal's Algorithm

Minimum Spanning Tree

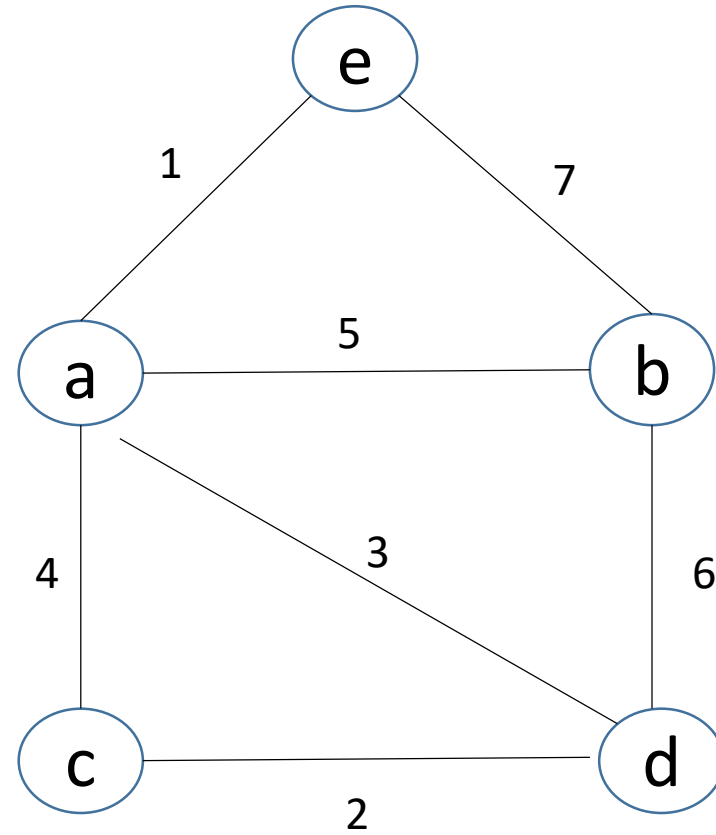
MST Review

- **Input:** Undirected graph $G = (V; E)$, edge costs c_e
- **Output:** Min-cost spanning tree (no cycles, connected)
- **Assumptions:** G is connected, distinct edge costs.
- **Cut Property:** If e is the cheapest edge crossing some cut $(A; B)$, then e belongs to the MST.

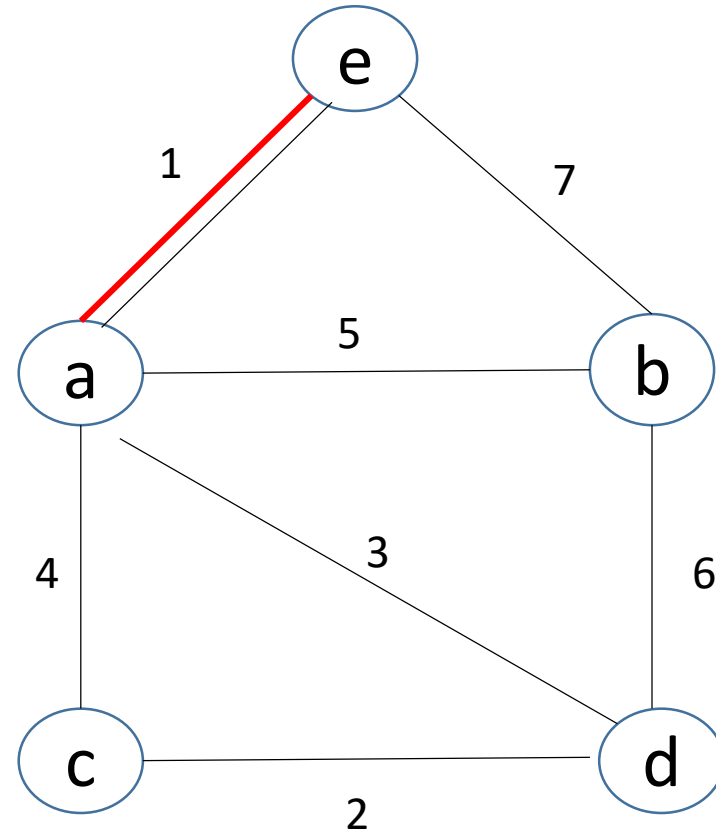
Kruskal's MST Algorithm

- Sort edges in order of increasing cost
[Rename edges 1; 2; : : : ;m so that $c_1 < c_2 < : : : < c_m$]
- $T = \emptyset$
- For $i = 1$ to m
 - If $T \cup \{i\}$ has no cycles
 - Add i to T
- Return T

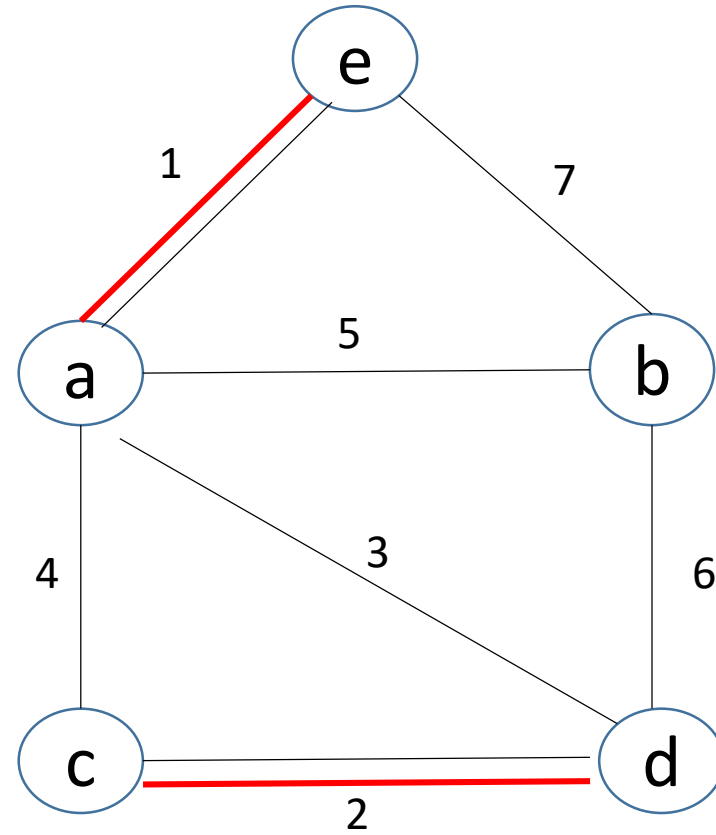
Kruskal's Algorithm Dry Run



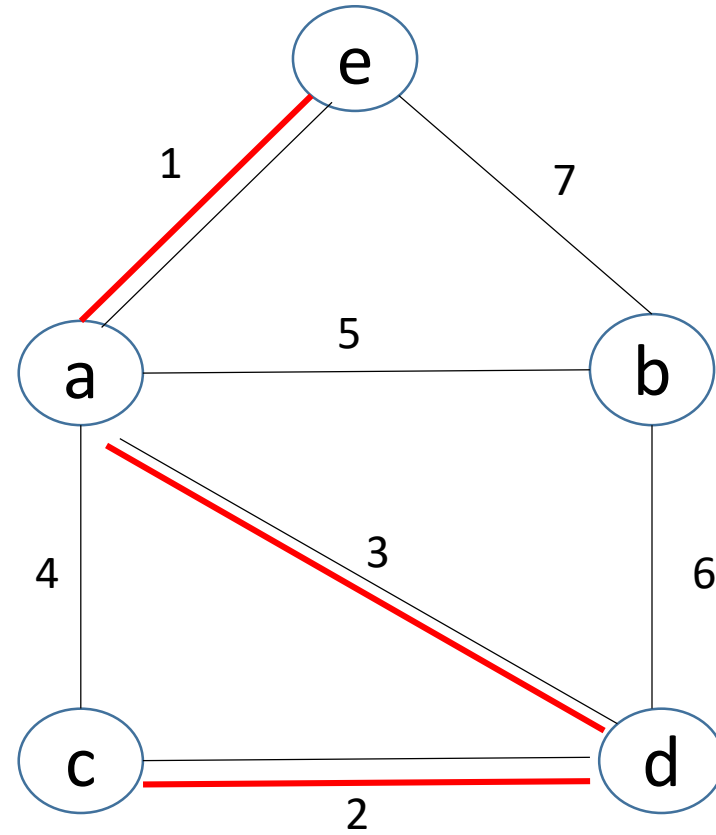
Kruskal's Algorithm Dry Run



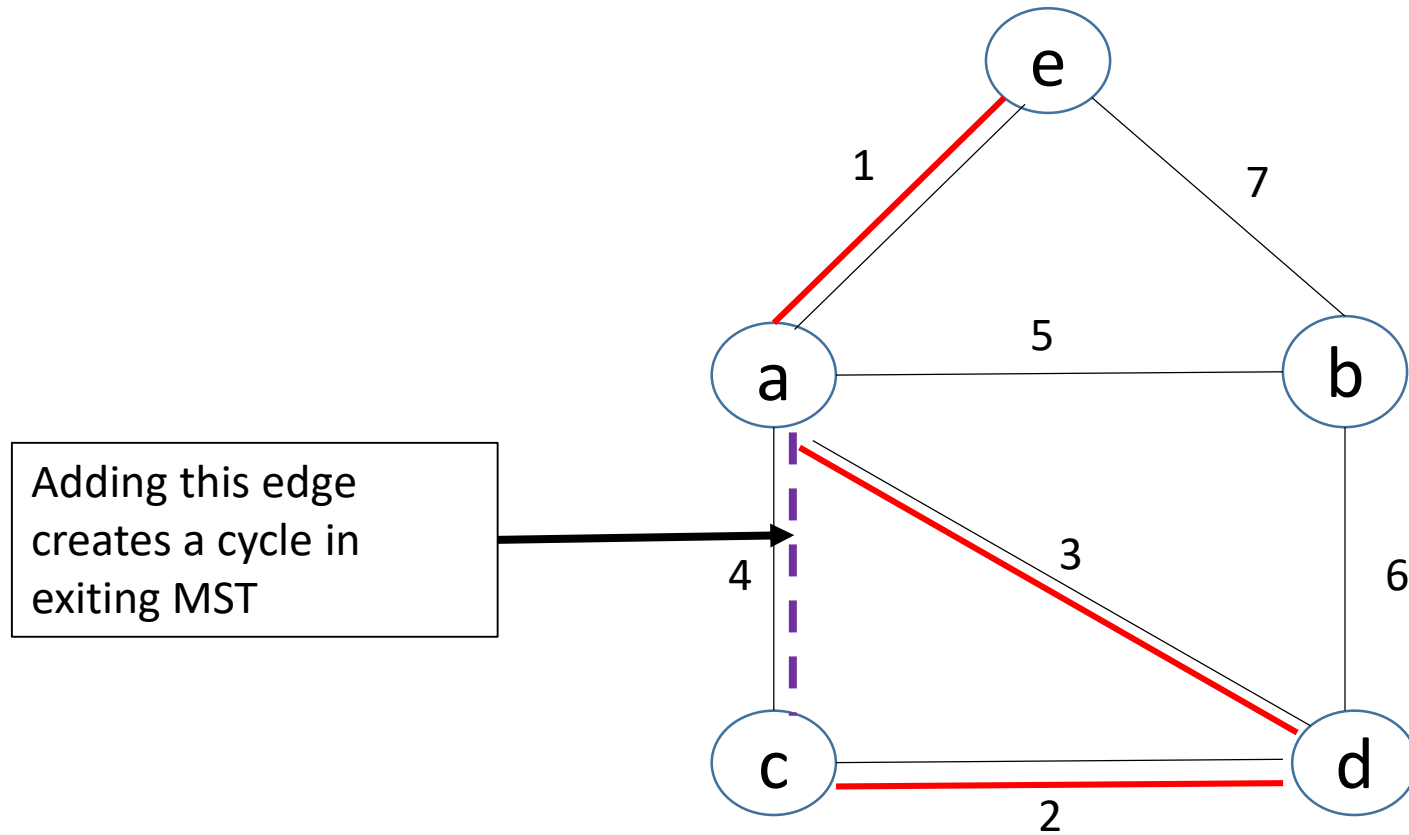
Kruskal's Algorithm Dry Run



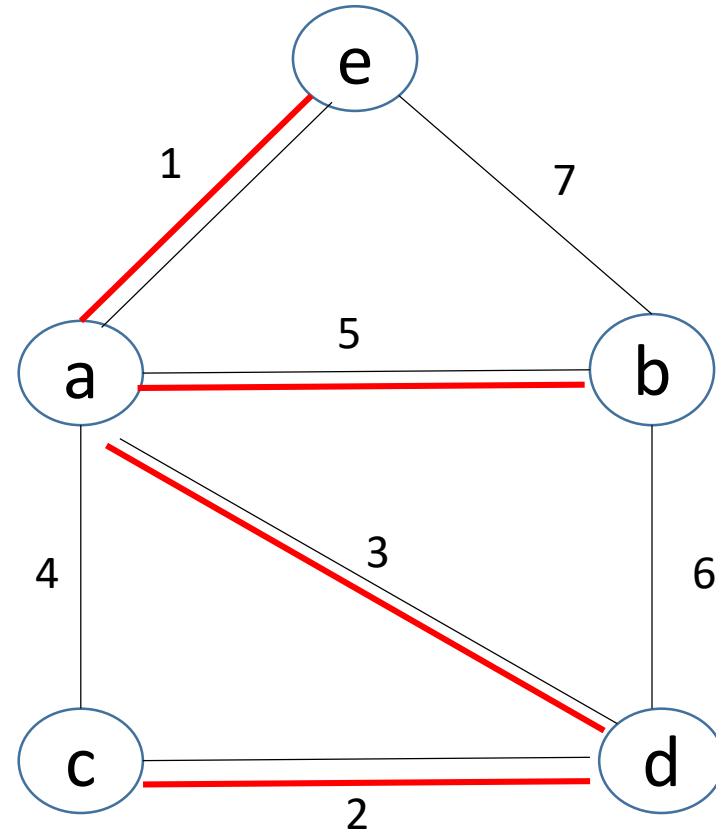
Kruskal's Algorithm Dry Run



Kruskal's Algorithm Dry Run



Kruskal's Algorithm Dry Run



How to check Cycles in MST?



Kruskal's MST Algorithm

- Sort edges in order of increasing cost. ($O(m \log n)$, recall $m = O(n^2)$ assuming nonparallel edges)
- $T = \emptyset$
 - For $i = 1$ to m ($O(m)$ iterations) $\longrightarrow O(m)$
 - If $T \cup \{i\}$ has no cycles ($O(n)$ time to check for cycle [Use BFS or DFS in the graph (V, T) which contains $\leq n - 1$ edges]) $\longrightarrow O(n)$
 - Add i to T
- Return T

Running time of straightforward implementation: ($m = \#$ of edges, $n = \#$ of vertices) $O(m \log n) + O(mn) = O(mn)$

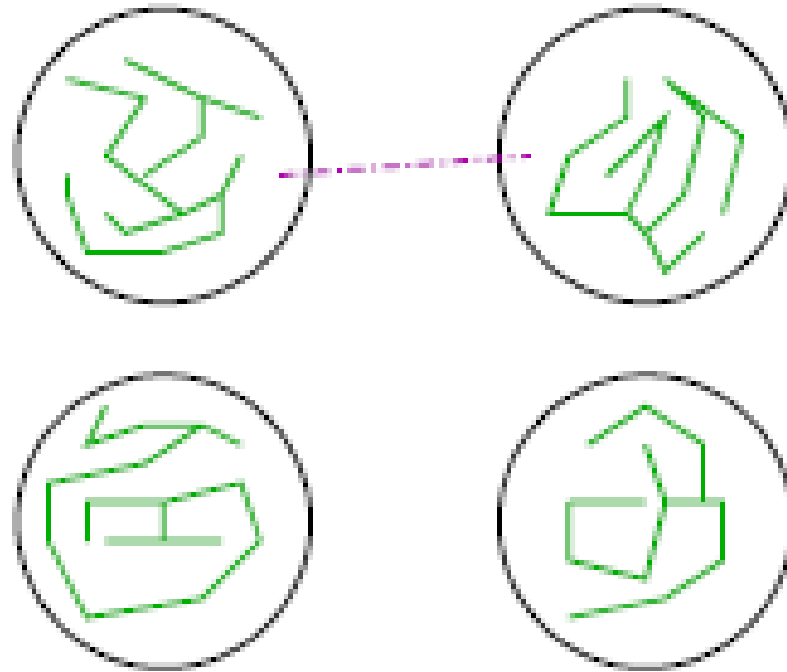
Plan: Data structure for $O(1)$ -time cycle checks $\Rightarrow O(m \log n)$ time.

Kruskal's Algorithm

- Can we improve the time to check cycles?
- Can we use some data structure to speed up Kruskal's algorithm?
- What kind of data structure can check efficiently if adding an edge creates a cycle

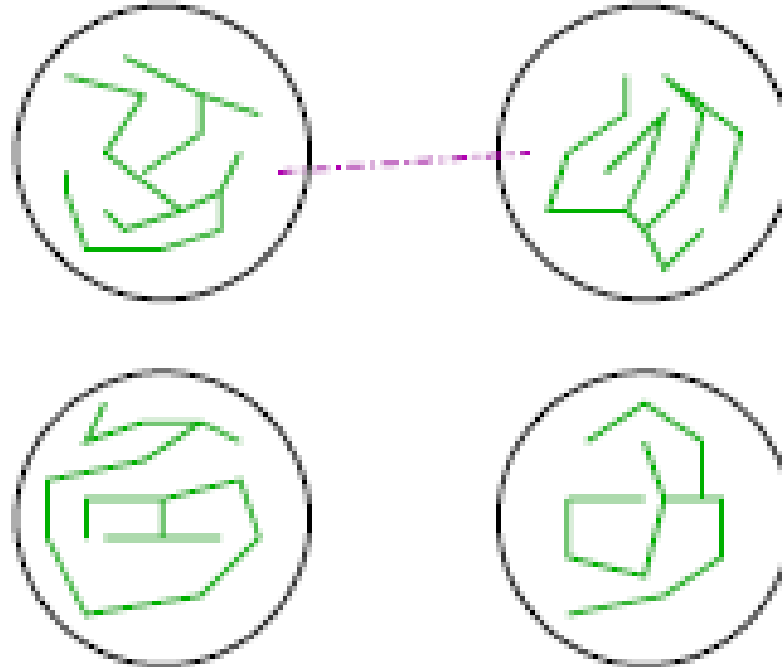
How to Check Cycle ?

- In Kruskal's algorithm, the partial MST is in form of different connected components



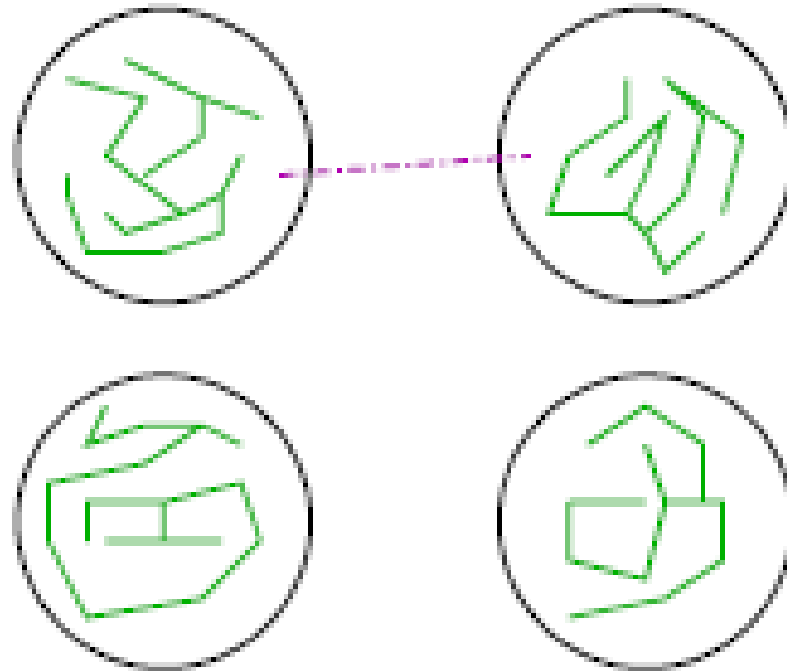
How to Check Cycle ?

- If the edge is added between vertices of different components then it will **not create a cycle**.
- If the edge is added between vertices of same component then it **will create a cycle**.



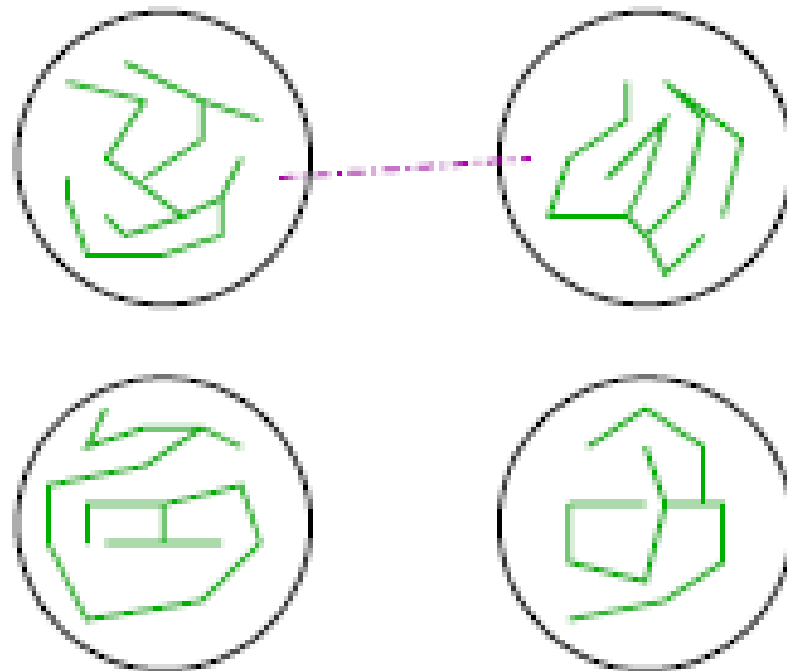
How to Check Cycle ?

- So we need a data structure that can keep track of different components created in the MST (created so far)



How to Check Cycle ?

- Suppose you are given a data structure that can keep track of different components of MST in Kruskal's algorithm
- What operations must be supported by that data structure ?

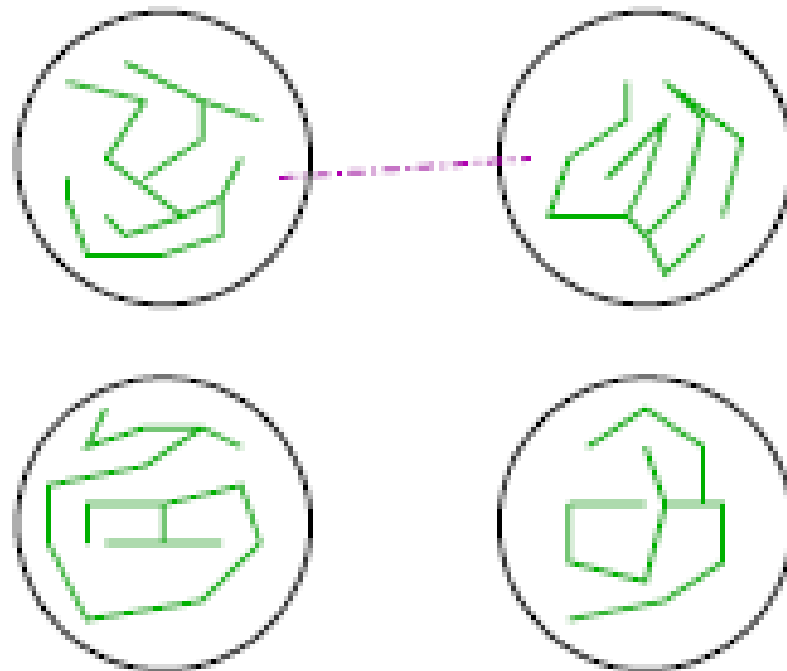


Operation of Data Structure

- Two Operations

1. Given a vertex check its component // needed for cycle check
2. What is second operation?

After cycle check, if an edge is selected to be added in MST, then should we make any update?

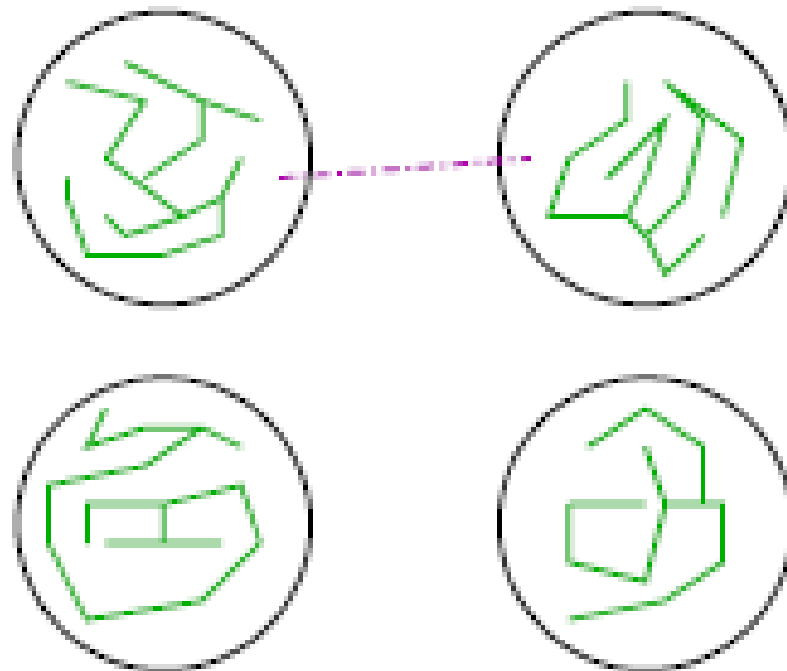


Operation of Data Structure

- Two Operations

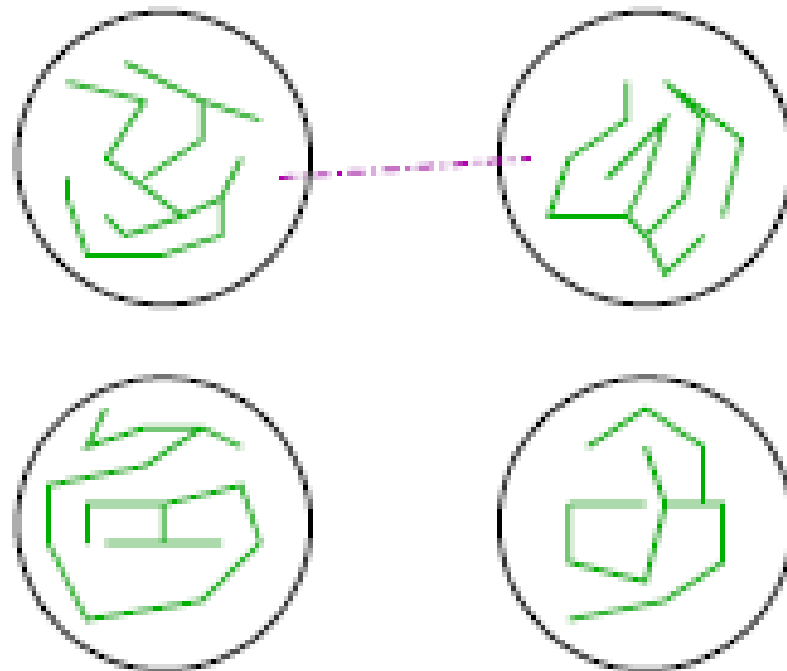
1. Given a vertex check its component
2. What is second operation?

When an edge is added, we should merge the two components since now they become one connected component due to the new edge



Operation of Data Structure

- Two Operations
 1. Given a vertex check its component
 2. Merge two components into one

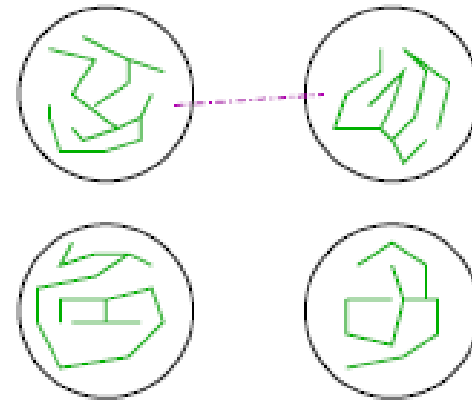
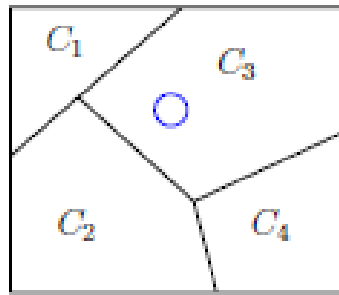


The Union-Find Data Structure

union-find data structure: Maintain partition of a set of objects.

FIND(X): Return name of group that X belongs to.

UNION(C_i, C_j): Fuse groups C_i, C_j into a single one.



Why useful for Kruskal's algorithm: Objects = vertices

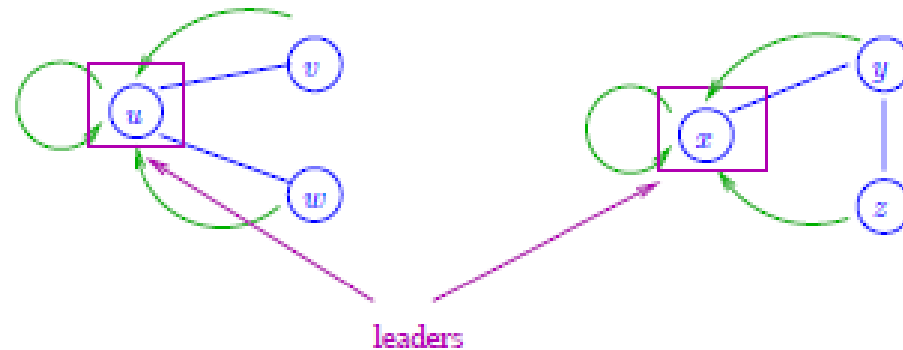
- Groups = Connected components w.r.t. chosen edges T .
- Adding new edge (u, v) to $T \iff$ Fusing connected components of u, v .

Union Find Basics

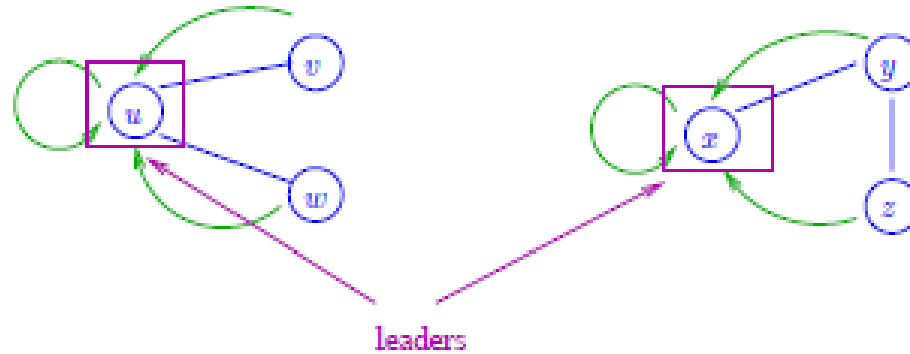
Motivation: $O(1)$ -time cycle checks in Kruskal's algorithm.

Idea #1: - Maintain one linked structure per connected component of (V, T) .

- Each component has an arbitrary leader vertex.



Union Find Basics



Invariant: Each vertex points to the leader of its component
[“name” of a component inherited from leader vertex]

Key point: Given edge (u, v) , can check if u & v already in same component in $O(1)$ time. [if and only if leader pointers of u, v match, i.e., $\text{FIND}(u) = \text{FIND}(v)$] $\Rightarrow O(1)$ -time cycle checks!

Maintaining the Invariant

Note: When new edge $(u; v)$ added to T , connected components of u & v merge.

Question: How many leader pointer updates are needed to restore the invariant in the worst case?

- a) (1)
- b) ($\log n$)
- c) (n) (e.g., when merging two components with $n/2$ vertices each)
- d) (m)

Maintaining the Invariant

Note: When new edge $(u; v)$ added to T , connected components of u & v merge.

Question: How many leader pointer updates are needed to restore the invariant in the worst case?

- a) (1)
- b) ($\log n$)
- c) (n) (e.g., when merging two components with $n/2$ vertices each)
- d) (m)

Maintaining the Invariant

Idea #2: When two components merge, have smaller one inherit the leader of the larger one. [Easy to maintain a size field in each component to facilitate this]

Question: How many leader pointer updates are now required to restore the invariant in the worst case?

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$ (for same reason as before, i.e., when merging two components with $n/2$ vertices each)
- d) (m)

Maintaining the Invariant

Idea #2: When two components merge, have smaller one inherit the leader of the larger one. [Easy to maintain a size field in each component to facilitate this]

Question: How many leader pointer updates are now required to restore the invariant in the worst case?

- a) (1)
- b) ($\log n$)
- c) (n) (for same reason as before, i.e., when merging two components with $n/2$ vertices each)
- d) (m)

Updating Leader Pointers

But: How many times does a single vertex v have its leader pointer updated over the course of Kruskal's algorithm?

- a) (1)
- b) $(\log n)$
- c) (n)
- d) (m)

Updating Leader Pointers

But: How many times does a single vertex v have its leader pointer updated over the course of Kruskal's algorithm?

a) (1)

b) ($\log n$)

c) (n)

d) (m)

Reason: Every time v 's leader pointer gets updated, population of its component at least doubles \Rightarrow Can only happen $\leq \log_2 n$ times.

Kruskal's MST Algorithm

- Sort edges in order of increasing cost
[Rename edges $1; 2; \dots; m$ so that $c_1 < c_2 < \dots < c_m$]
- $T = \emptyset$
- For $i = 1$ to m
 - If $\text{Find}(u) \neq \text{Find}(v)$ // edge i is between vertices u and v
 - Add i to T
 - Union ($\text{Find}(u)$, $\text{Find}(v)$)
- Return T

Running Time of Fast Implementation

Scorecard:

$O(m \log n)$ time for sorting

$O(m)$ times for cycle checks [$O(1)$ per iteration]

$O(n \log n)$ time overall for leader pointer updates

$O(m \log n)$ total (Matching Prim's algorithm)

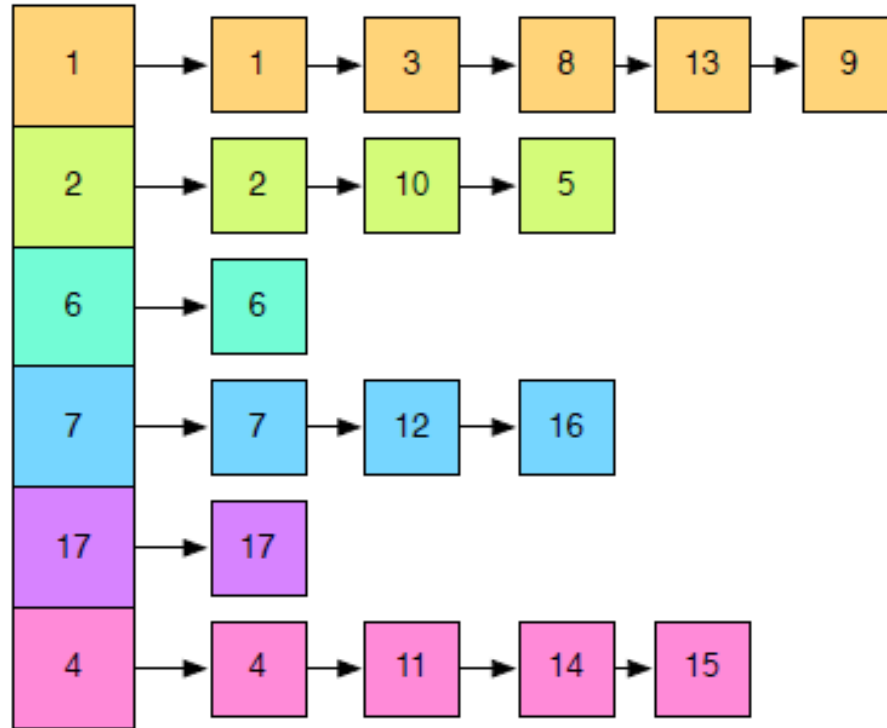
Union-Find Abstract Data Type

The Union-Find abstract data type supports the following operations:

- **UF.create(S)**: create the data structure containing $|S|$ sets, each containing one item from S .
- **UF.Find(i)**: return the “name” of the set containing item i .
- **UF.union(a,b)**: merge the sets with names a and b into a single set.

A Union Find Data Structure

UF Items:



UF Sizes:

1	5
2	3
6	1
7	3
17	1
4	4

UF Sets Array:

1	2	1	4	2	6	7	1	1	2	4	7	1	4	4	7	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Implementing the union & find operations

- **make union find(S)** Create data structures on previous slide.
Takes time proportional to the size of S (number of vertices).
- **find(i)** Return UF.sets[i].
Takes a constant amount of time.
- **union(x,y)** Use the “size” array to decide which set is smaller.
Assume x is smaller.
Walk down elements i in set x (UF linked Items), setting
sets[i] = y.
Set size[y] = size[y] + size[x].
Takes time proportional to size[x].

Kruskal Algorithm Dry Run Using Union Find

UF Items

a	→
b	→
c	→
d	→
e	→
f	→
g	→
h	→
i	→

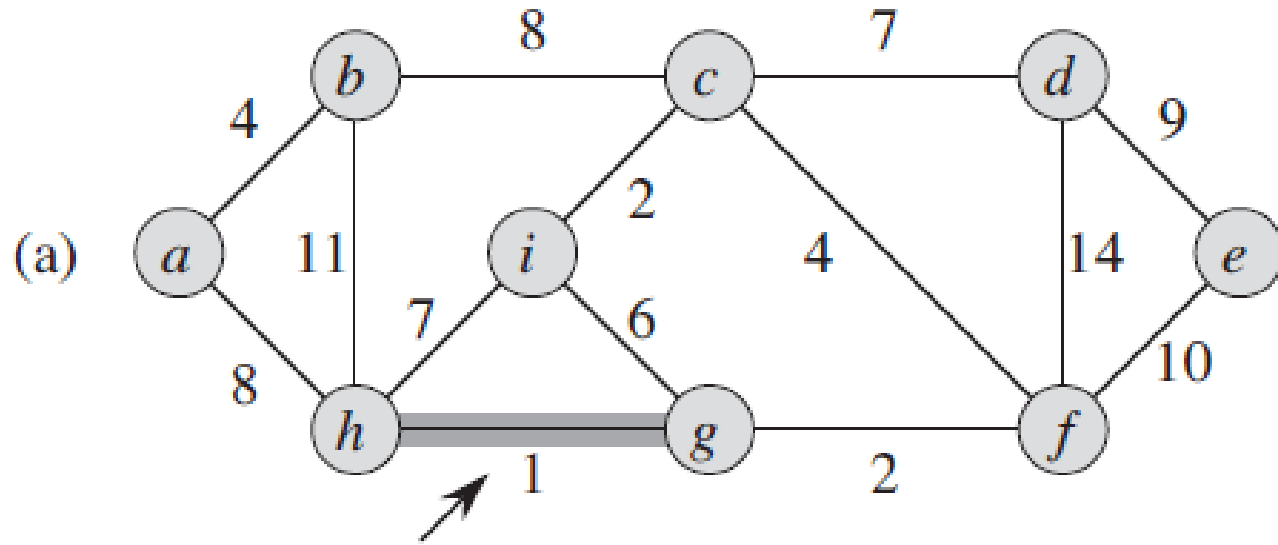
UF Sizes

a	1
b	1
c	1
d	1
e	1
f	1
g	1
h	1
i	1

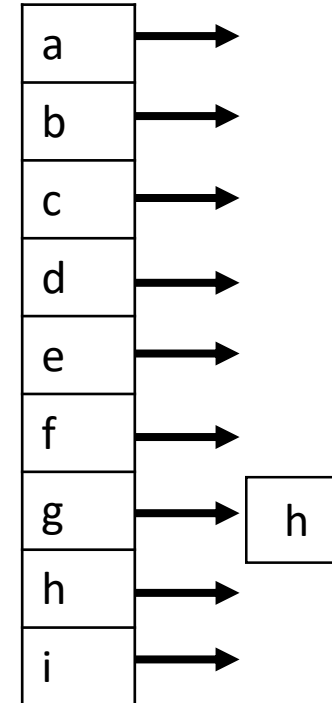
UF Sets Array

a	b	c	d	e	f	g	h	i
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



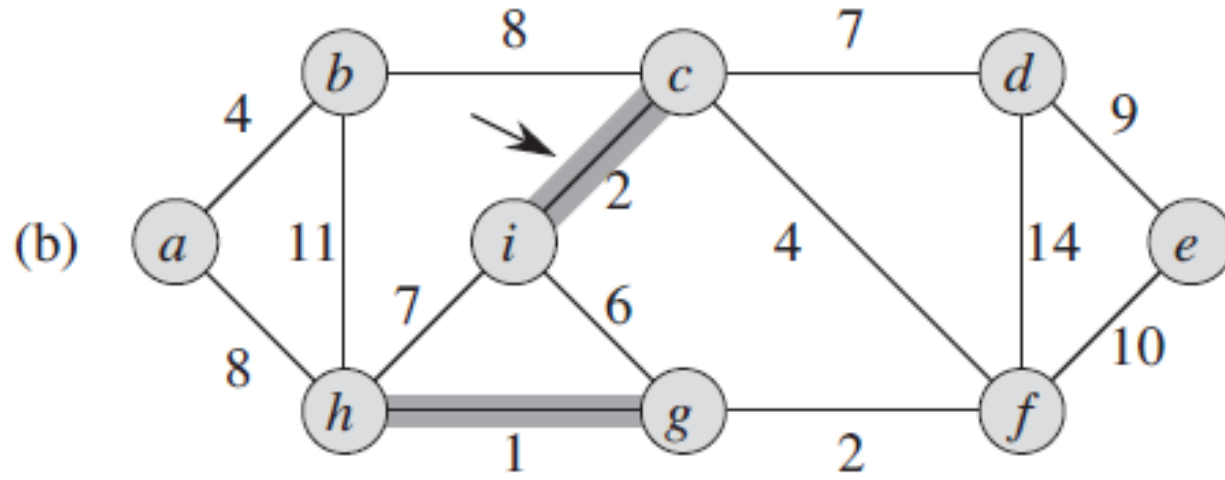
UF Sizes

a	1
b	1
c	1
d	1
e	1
f	1
g	2
h	0
i	1

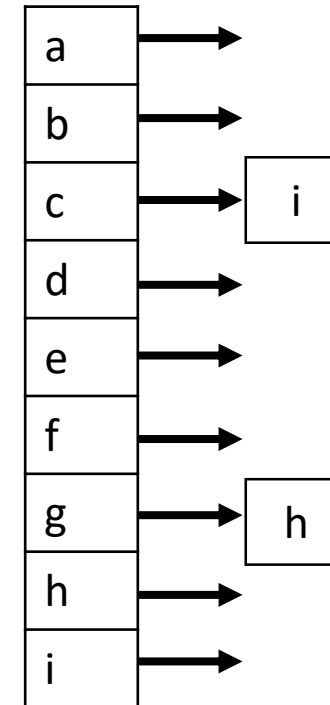
UF Sets Array

a	b	c	d	e	f	g	g	i
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



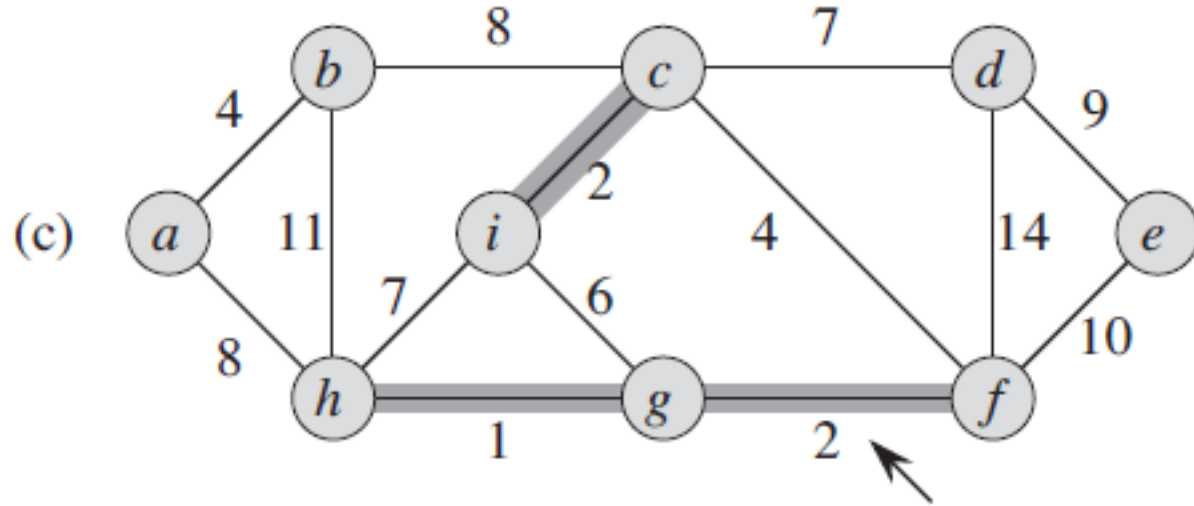
UF Sizes

a	1
b	1
c	2
d	1
e	1
f	1
g	2
h	0
i	0

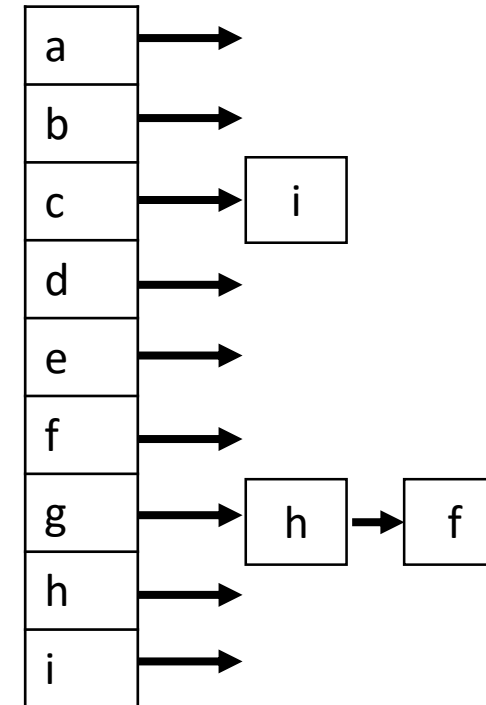
UF Sets Array

a	b	c	d	e	f	g	g	c
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



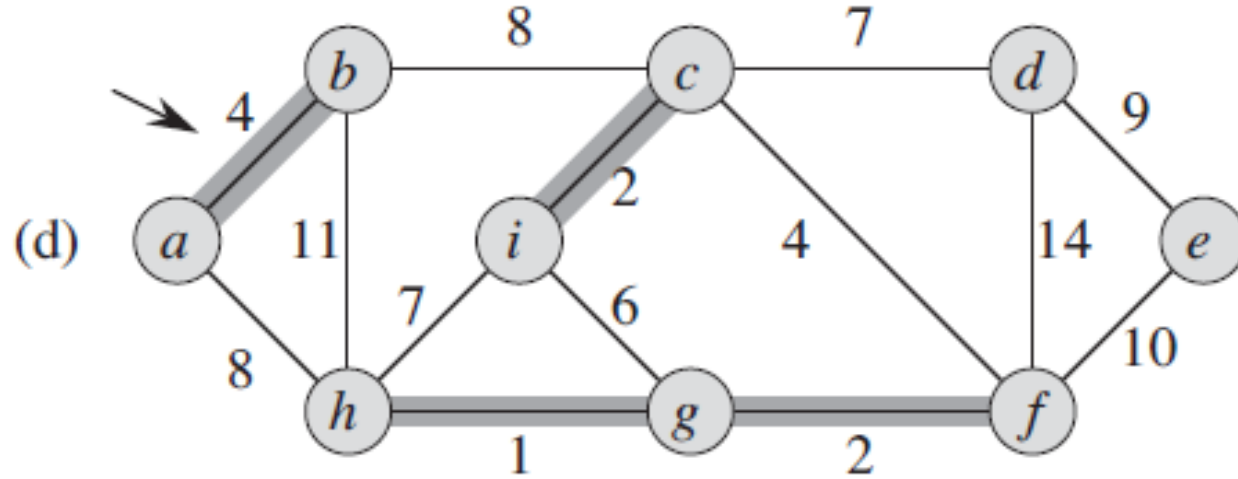
UF Sizes

a	1
b	1
c	2
d	1
e	1
f	0
g	3
h	0
i	0

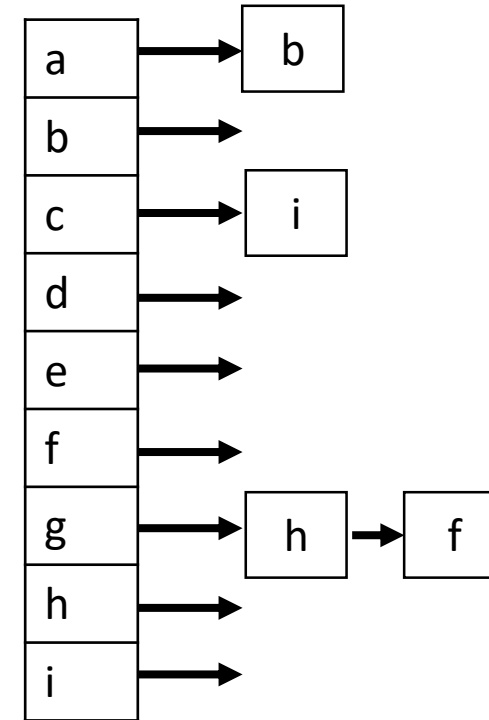
UF Sets Array

a	b	c	d	e	g	g	g	c
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



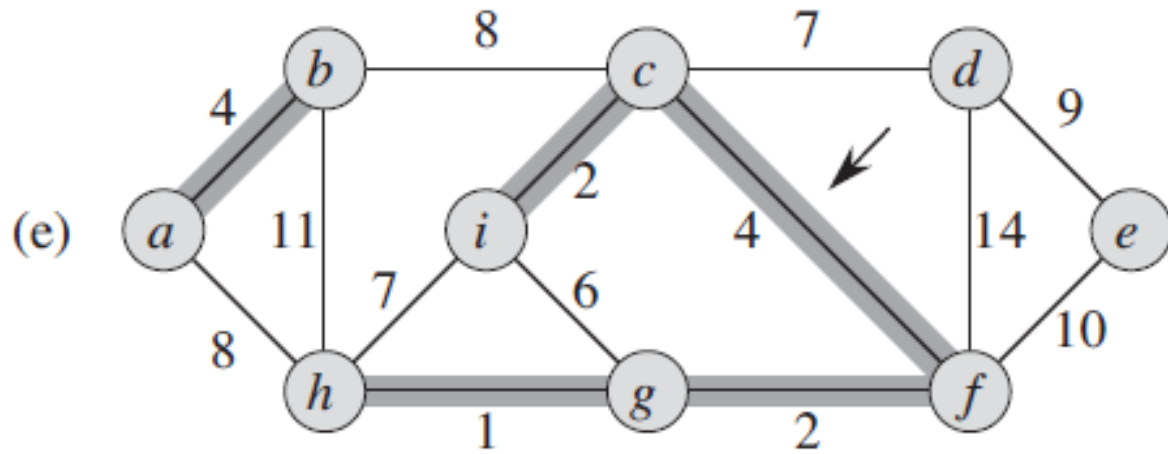
UF Sizes

a	2
b	0
c	2
d	1
e	1
f	0
g	3
h	0
i	0

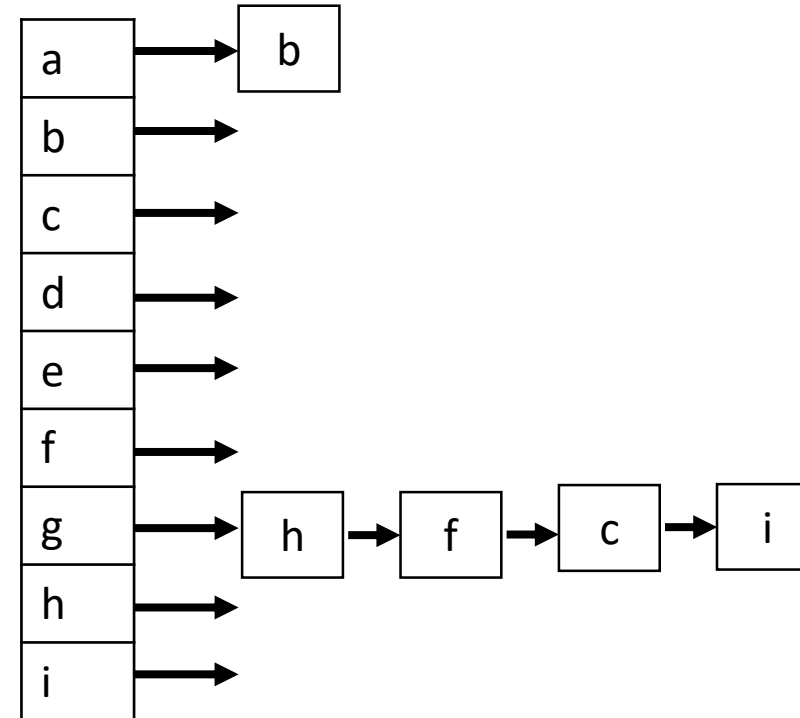
UF Sets Array

a	a	c	d	e	g	g	g	c
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



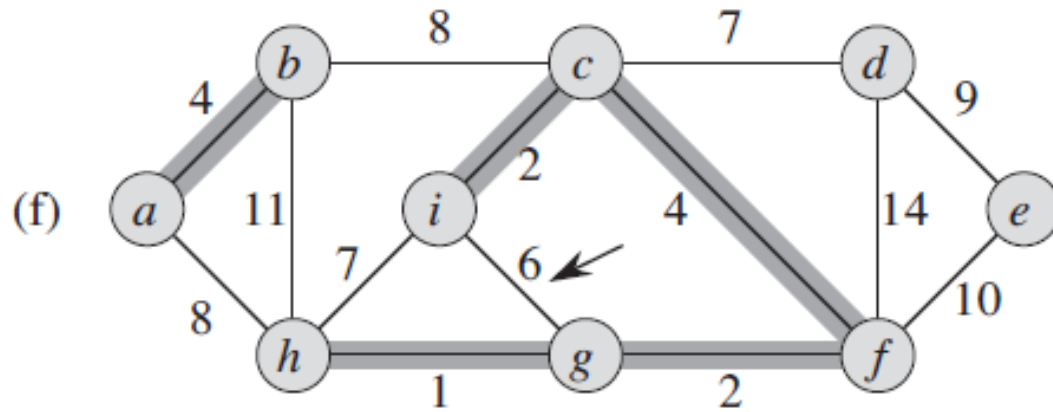
UF Sizes

a	2
b	0
c	0
d	1
e	1
f	0
g	5
h	0
i	0

UF Sets Array

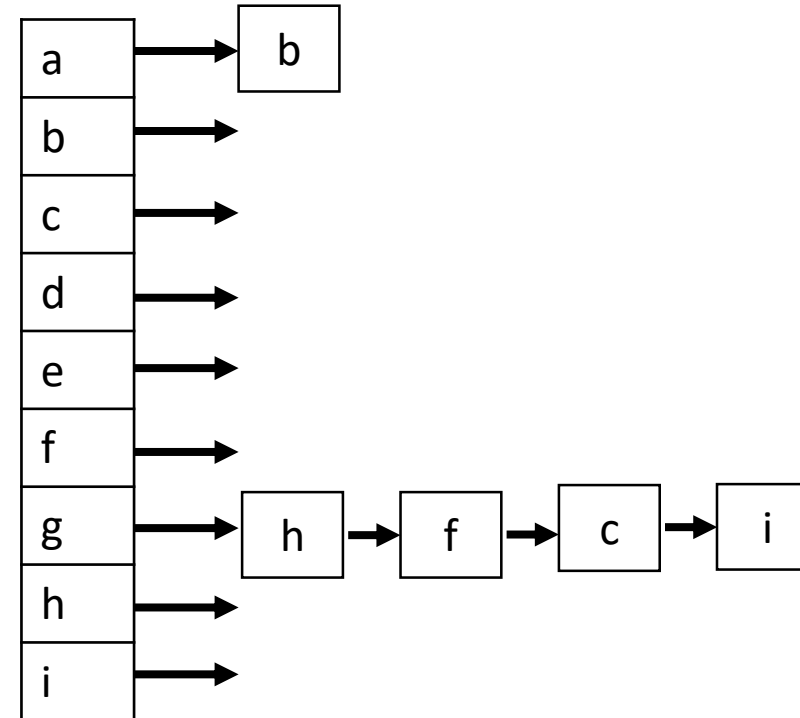
a	a	g	d	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



Edge (i, g) creates cycle so it will not be added in MST

UF Items



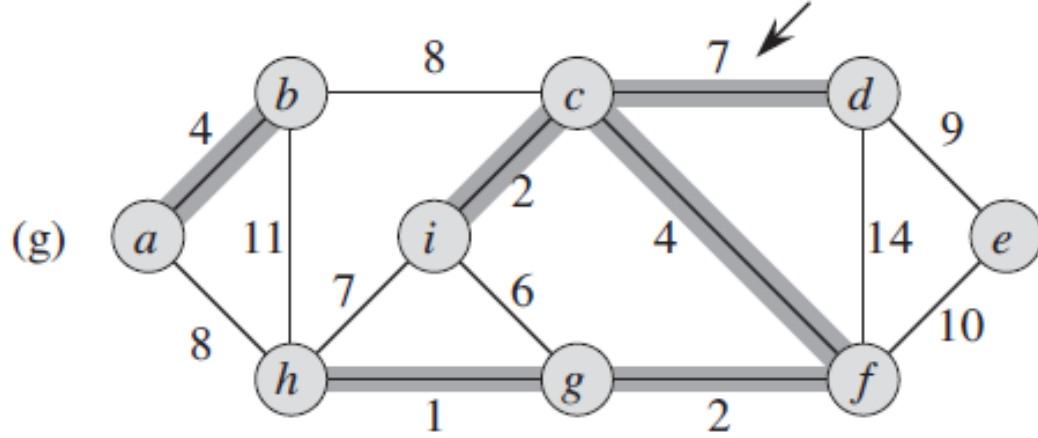
UF Sizes

a	2
b	0
c	0
d	1
e	1
f	0
g	5
h	0
i	0

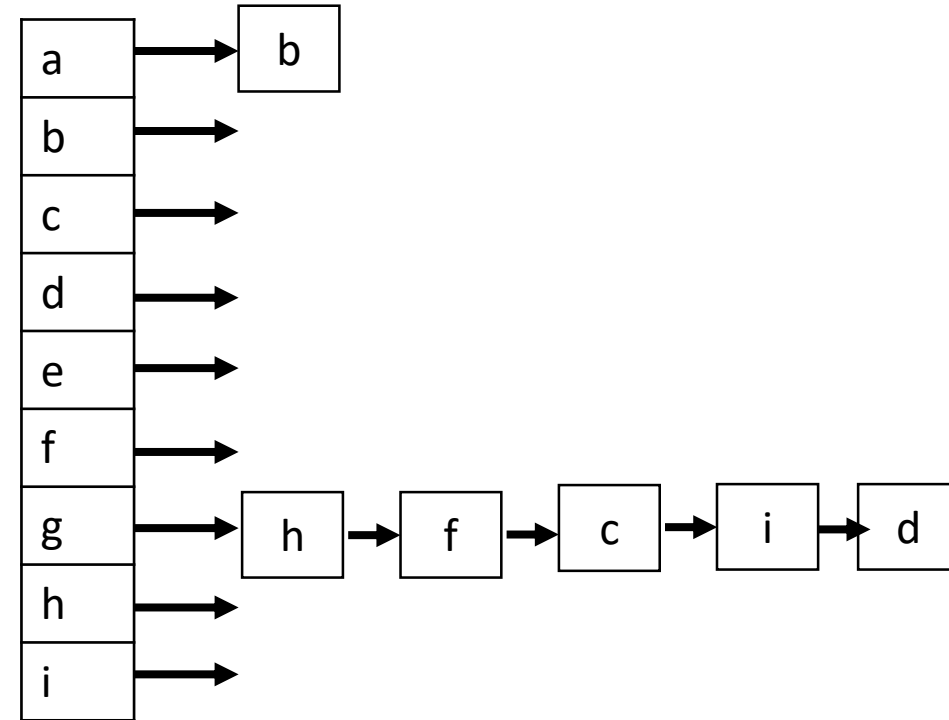
UF Sets Array

a	a	g	d	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



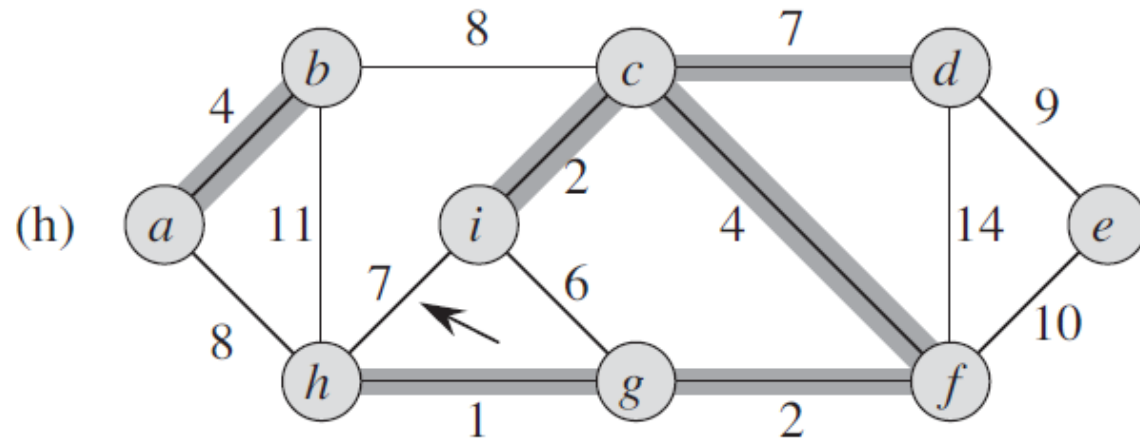
UF Sizes

a	2
b	0
c	0
d	0
e	1
f	0
g	6
h	0
i	0

UF Sets Array

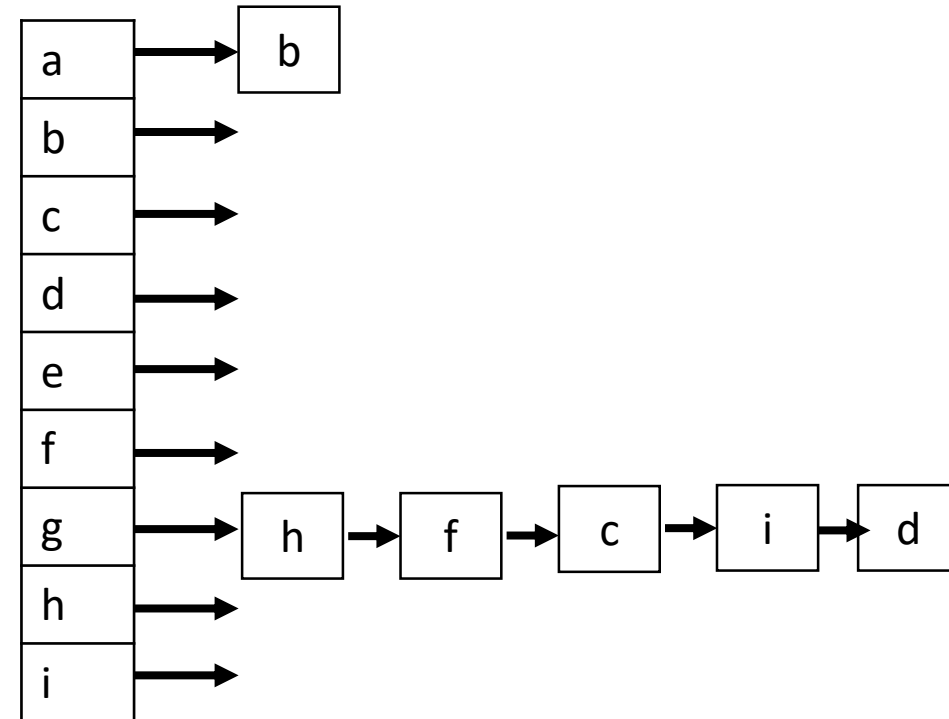
a	a	g	g	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



Edge (i, h) creates cycle so it will not be added in MST

UF Items



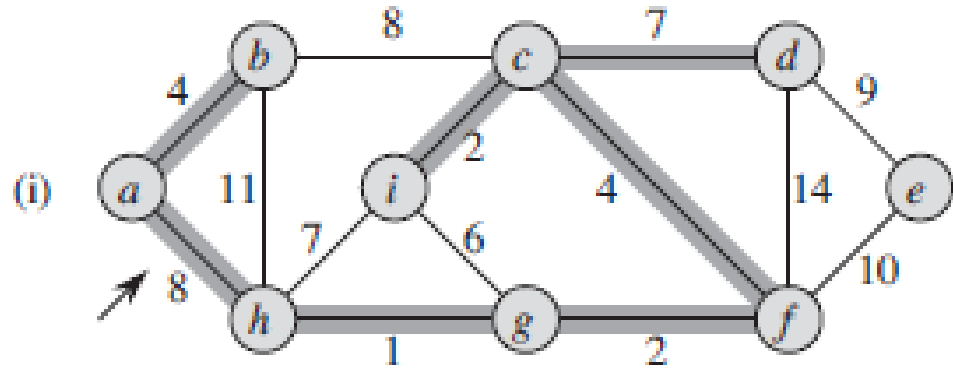
UF Sizes

a	2
b	0
c	0
d	0
e	1
f	0
g	6
h	0
i	0

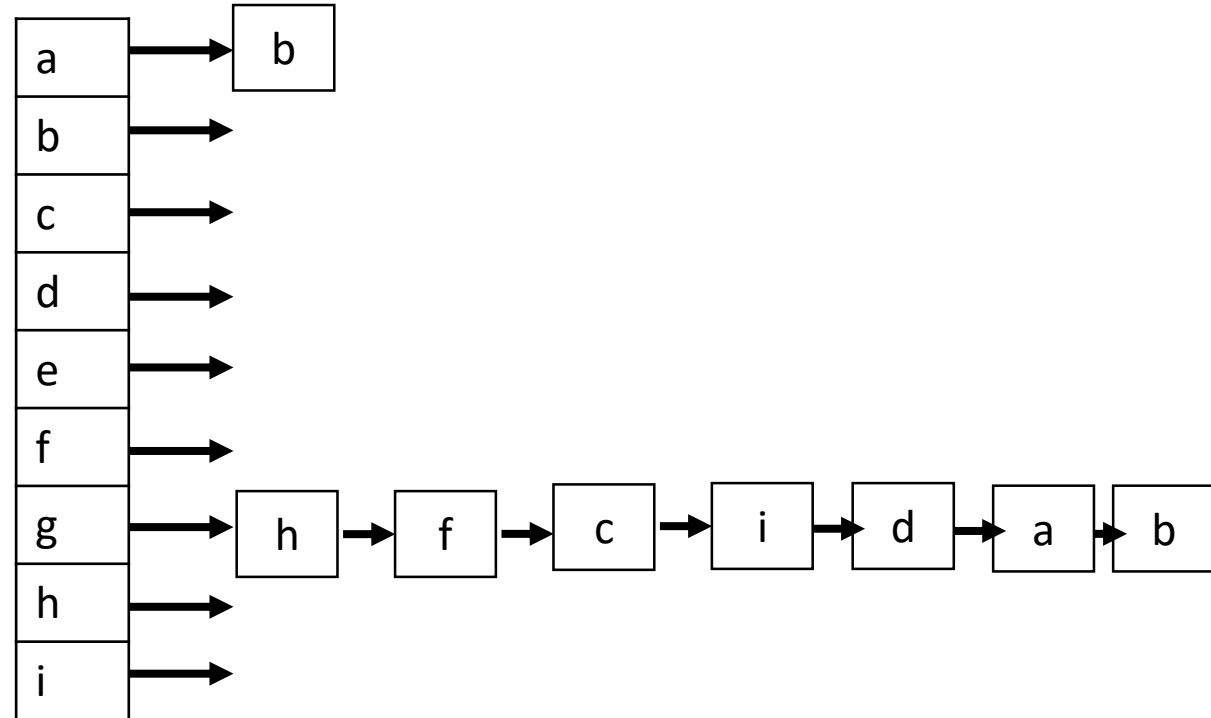
UF Sets Array

a	a	g	g	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



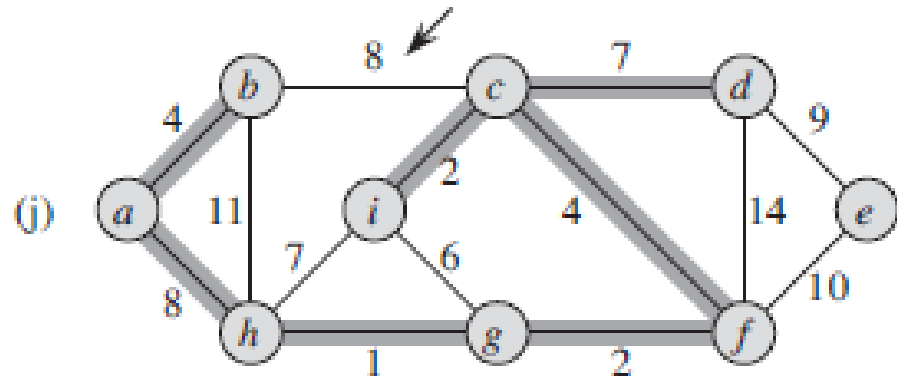
UF Sizes

a	0
b	0
c	0
d	0
e	1
f	0
g	8
h	0
i	0

UF Sets Array

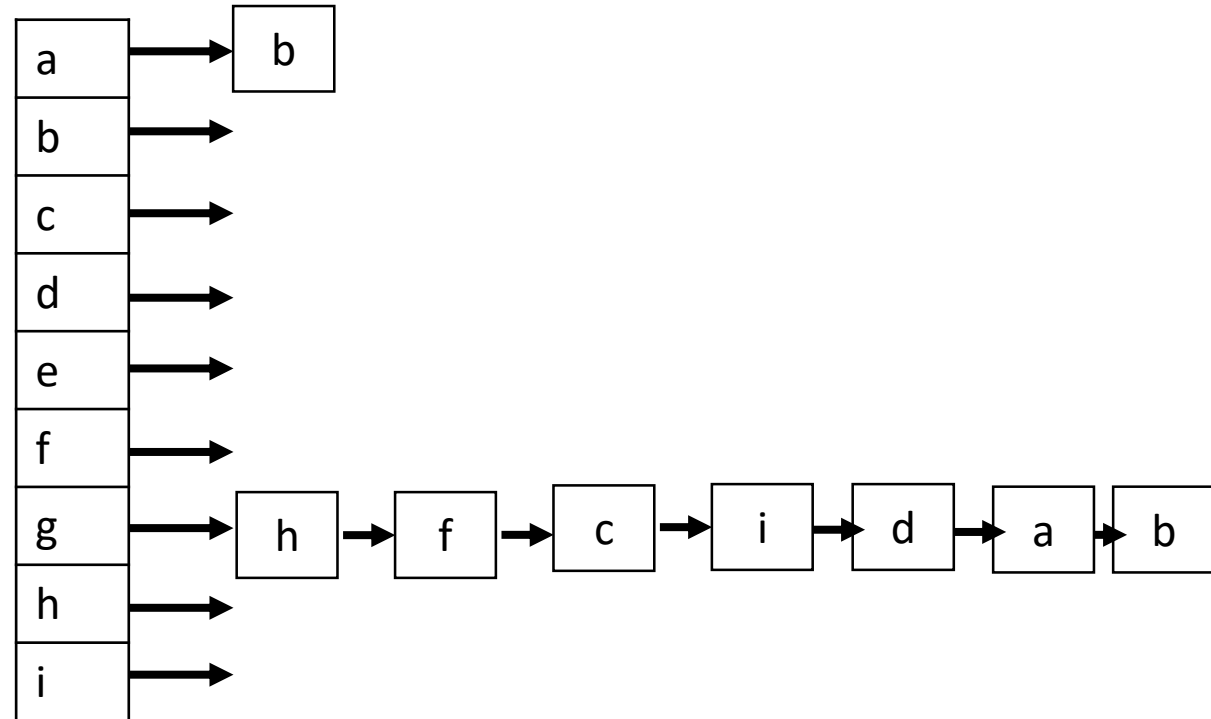
g	g	g	g	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



Edge (b, c) creates cycle so it will not be added in MST

UF Items



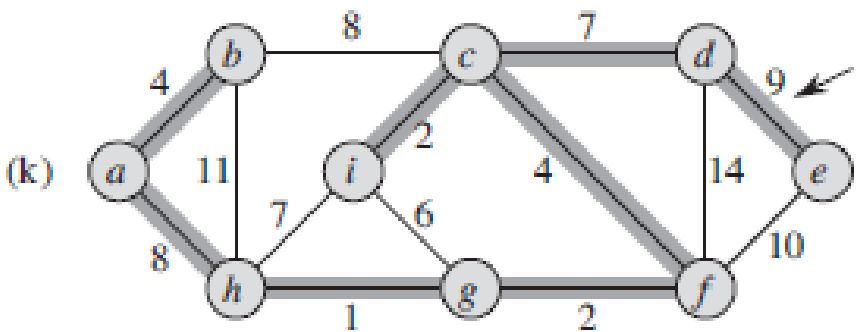
UF Sizes

a	0
b	0
c	0
d	0
e	1
f	0
g	8
h	0
i	0

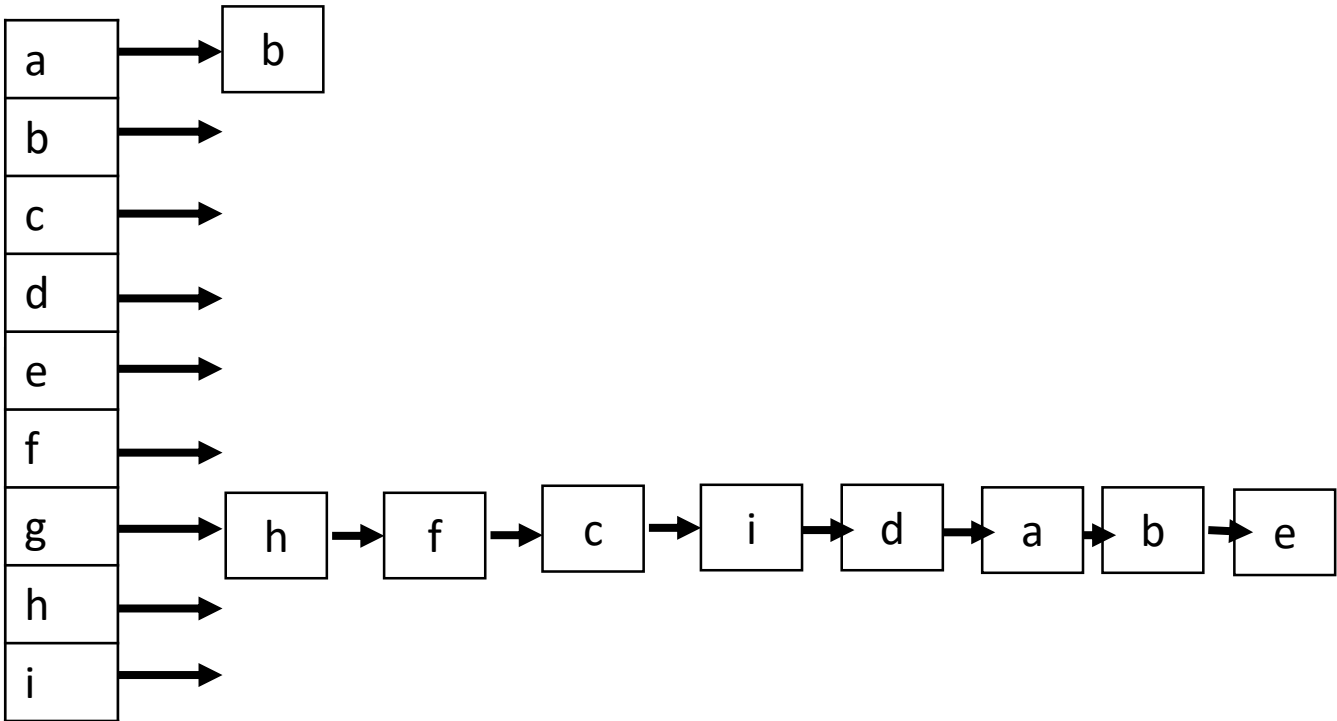
UF Sets Array

g	g	g	g	e	g	g	g	g
a	b	c	d	e	f	g	h	i

Kruskal Algorithm Dry Run Using Union Find



UF Items



UF Sizes

a	0
b	0
c	0
d	0
e	0
f	0
g	9
h	0
i	0

UF Sets Array

g	g	g	g	g	g	g	g	g
a	b	c	d	e	f	g	h	i

Proof of Correctness Part 1

Theorem: Kruskal's algorithm is correct.

Proof: Let T^* = output of Kruskal's algorithm on input graph G .

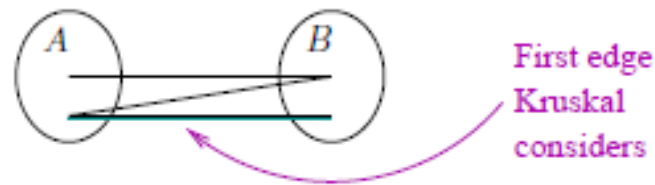
(1) Clearly T^* has no cycles.

(2) T^* is connected. Why?

(2a) By Empty Cut Lemma, only need to show that T^* crosses every cut.

(2b) Fix a cut (A, B) . Since G connected at least one of its edges crosses (A, B) .

Key point: Kruskal will include first edge crossing (A, B) that it sees [by Lonely Cut Corollary, cannot create a cycle]



Proof of Correctness Part 2

(3) Every edge of T^* satisfied by the Cut Property. (Implies T^* is the MST)

Reason for (3): Consider iteration where edge (u, v) added to current set T . Since $T \cup \{(u, v)\}$ has no cycle, T has no $u - v$ path.

$\Rightarrow \exists$ empty cut (A, B) separating u and v . (As in proof of Empty Cut Lemma)

\Rightarrow By (2b), no edges crossing (A, B) were previously considered by Kruskal's algorithm.

$\Rightarrow (u, v)$ is the first (+ hence the cheapest!) edge crossing (A, B) .

$\Rightarrow (u, v)$ justified by the Cut Property. QED

