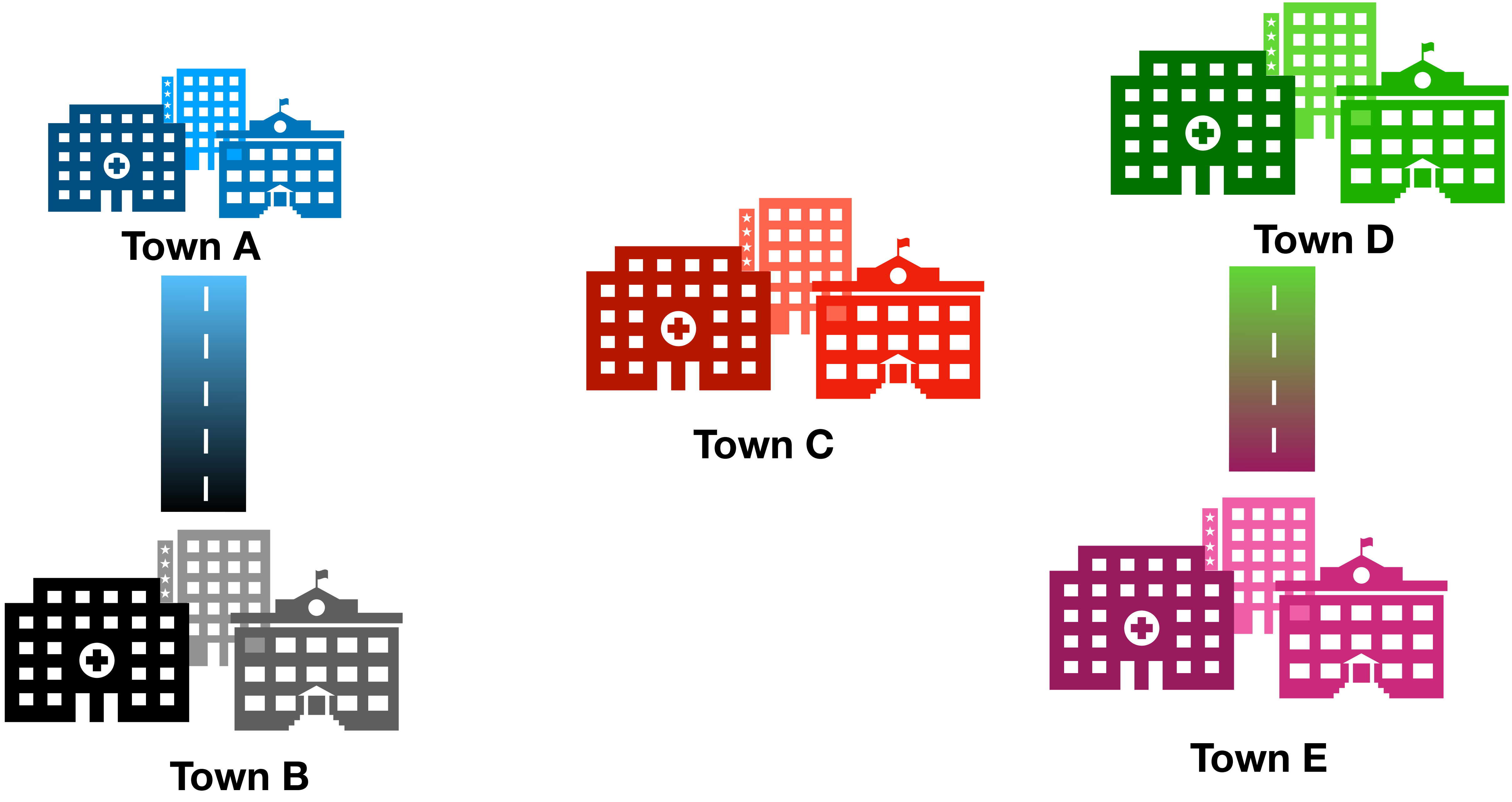
CS 61BL Lab 21

Ryan Purpura

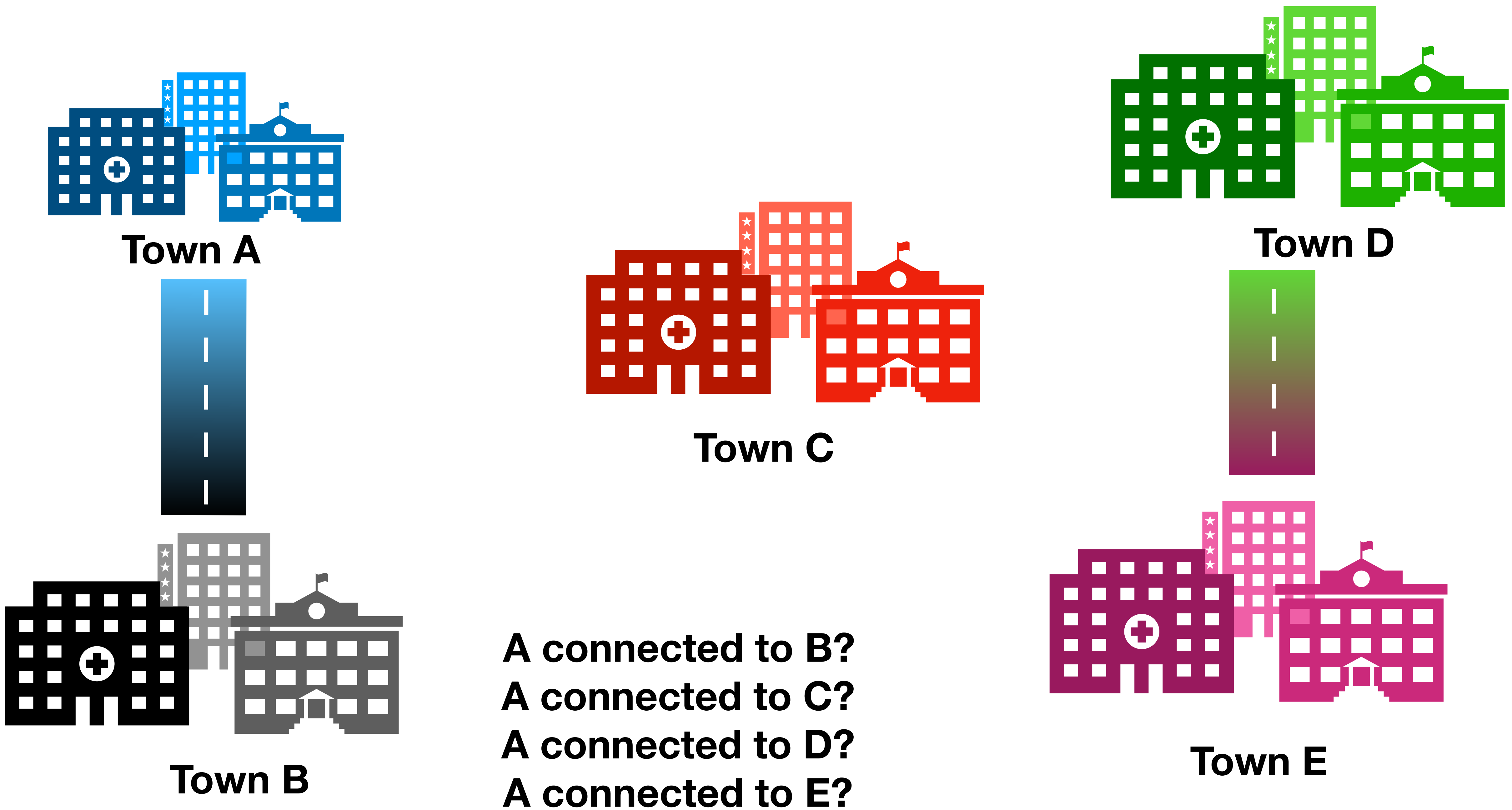
Announcements

- It is the final week!
- Final on Thursday!

Disjoint Sets

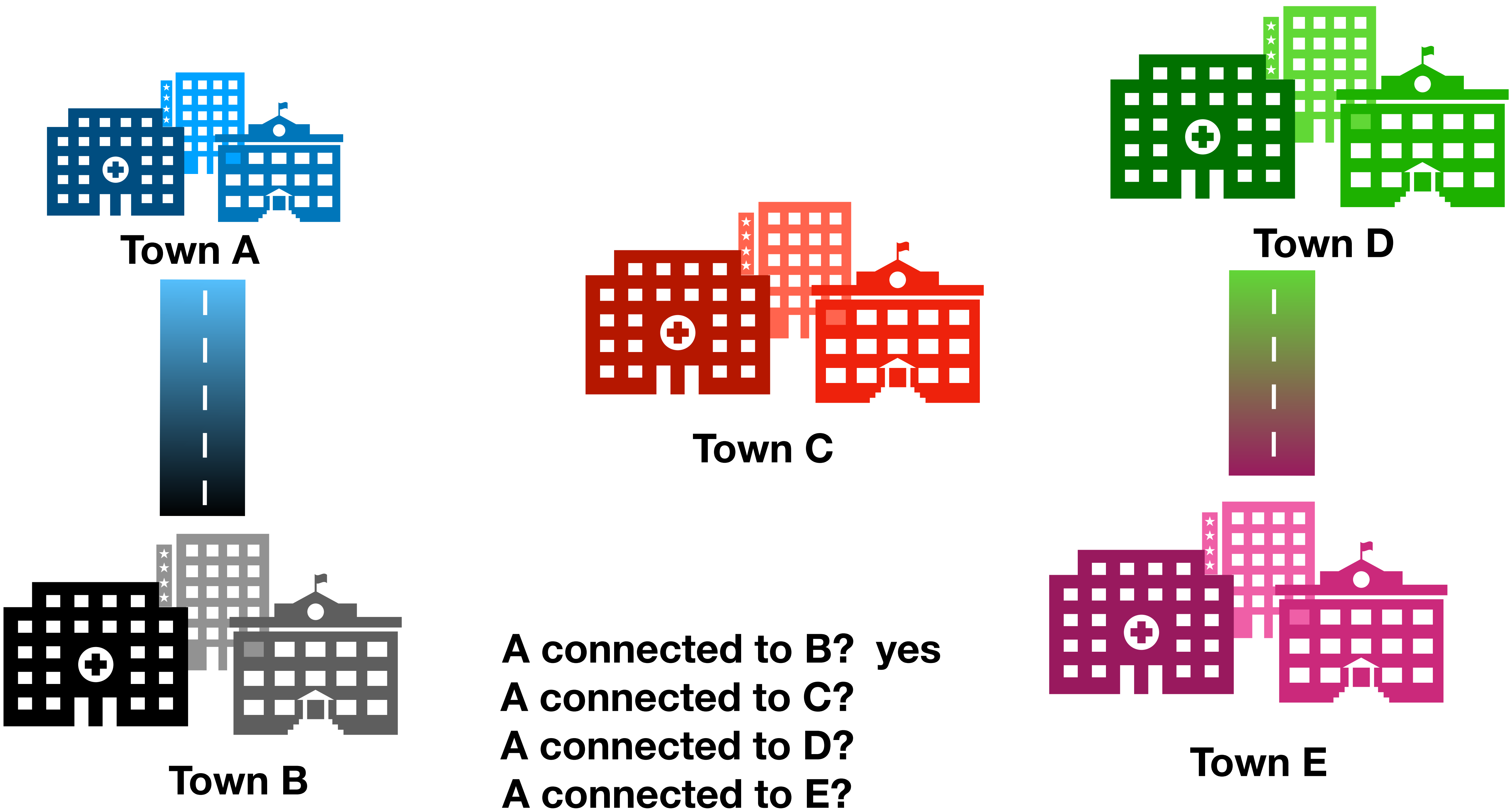


Disjoint Sets



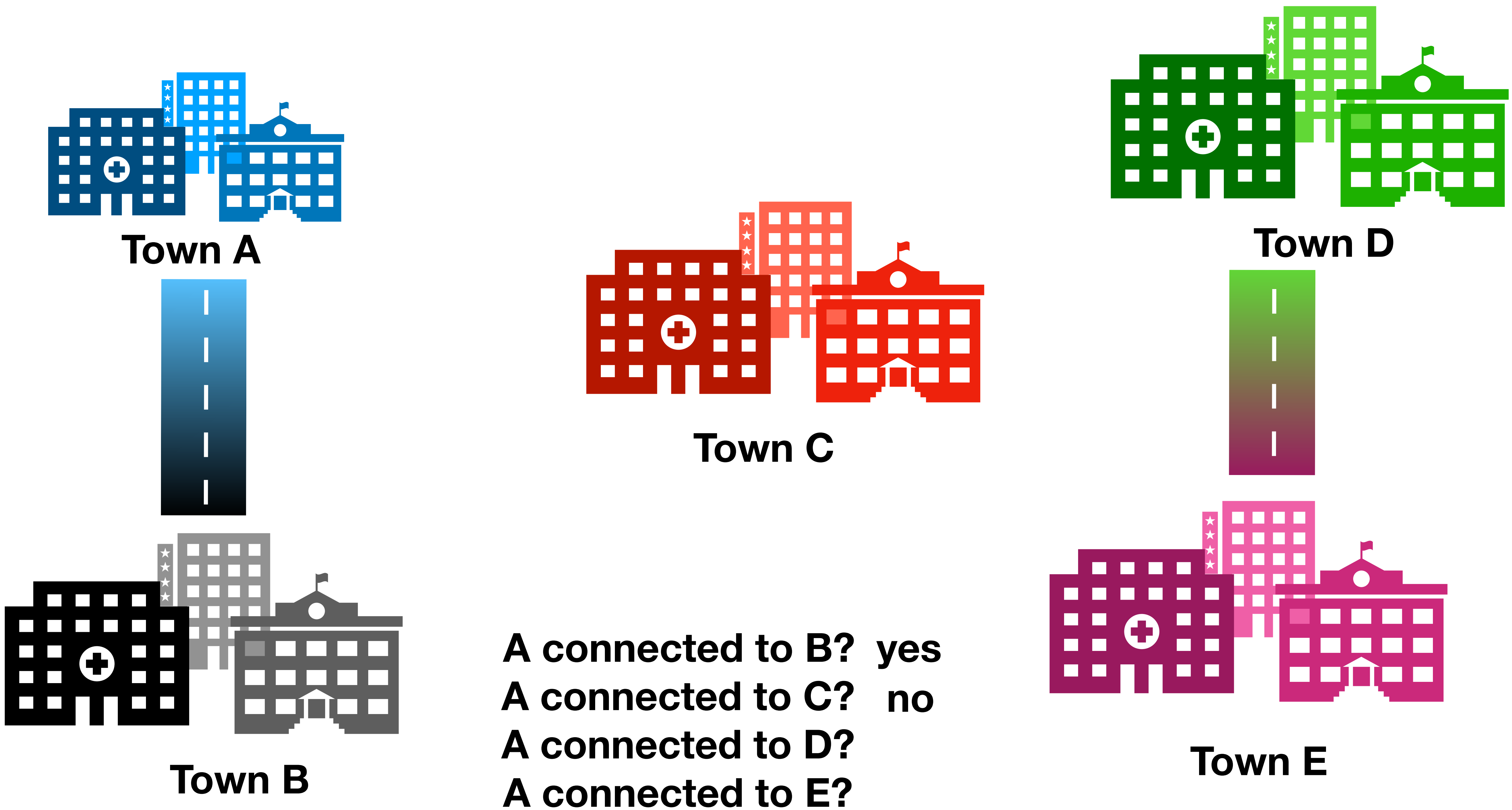
A connected to B?
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets



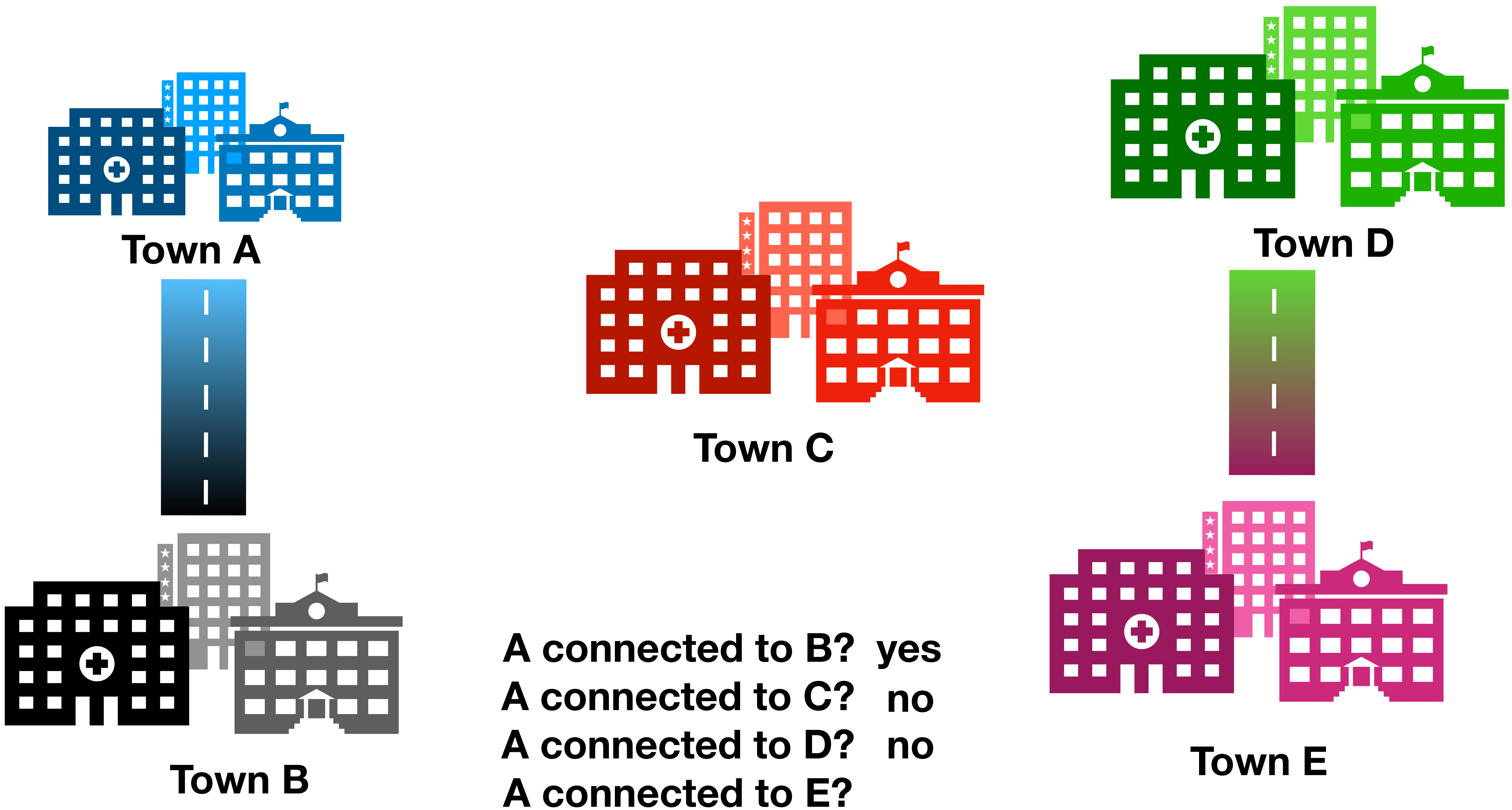
A connected to B? yes
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets



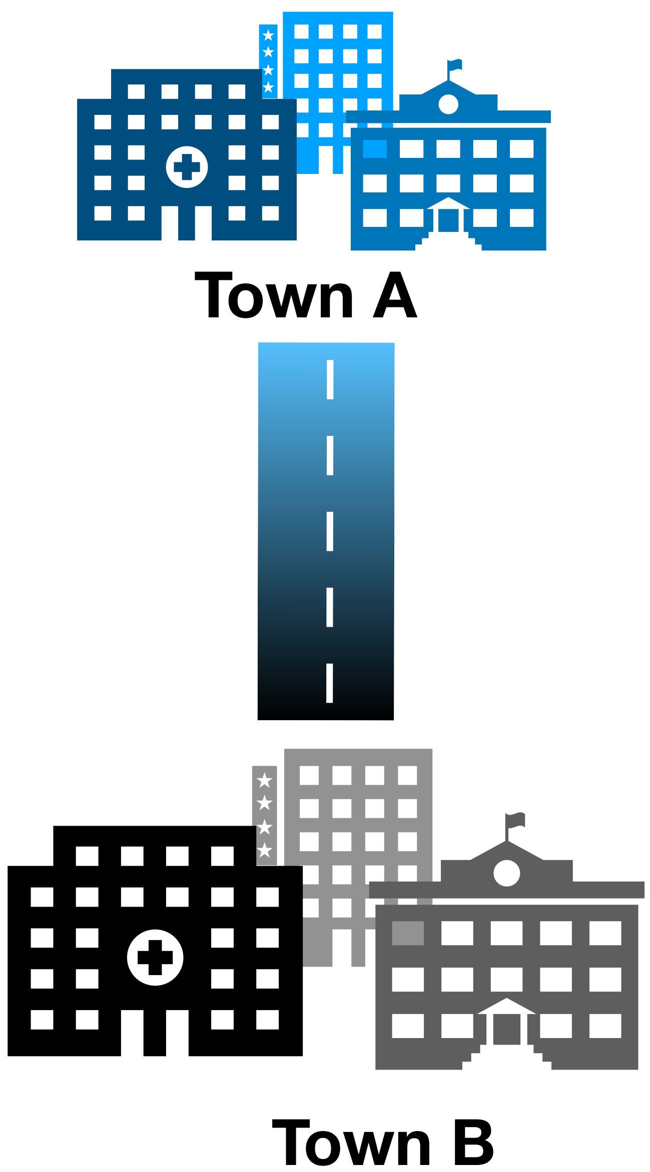
A connected to B? yes
A connected to C? no
A connected to D?
A connected to E?

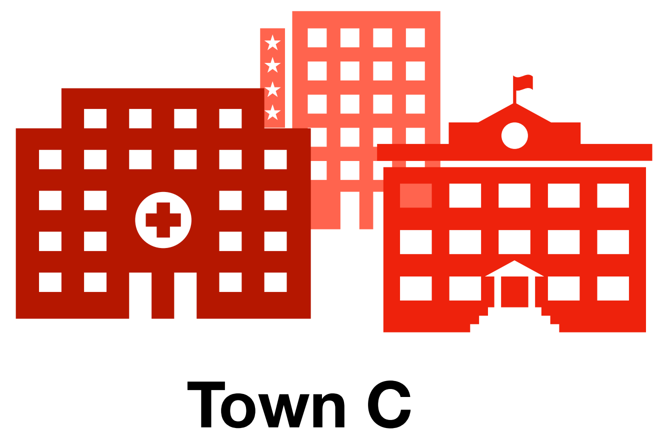
Disjoint Sets

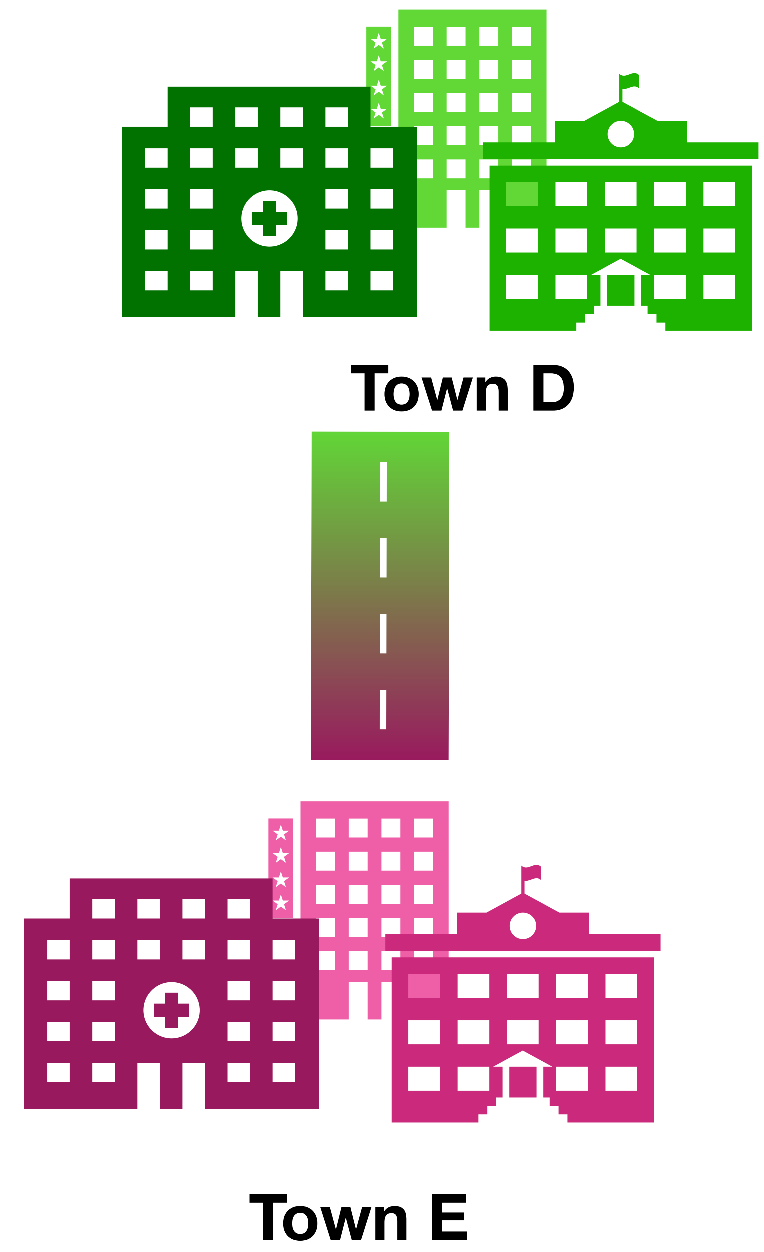


A connected to B? yes
A connected to C? no
A connected to D? no
A connected to E?

Disjoint Sets

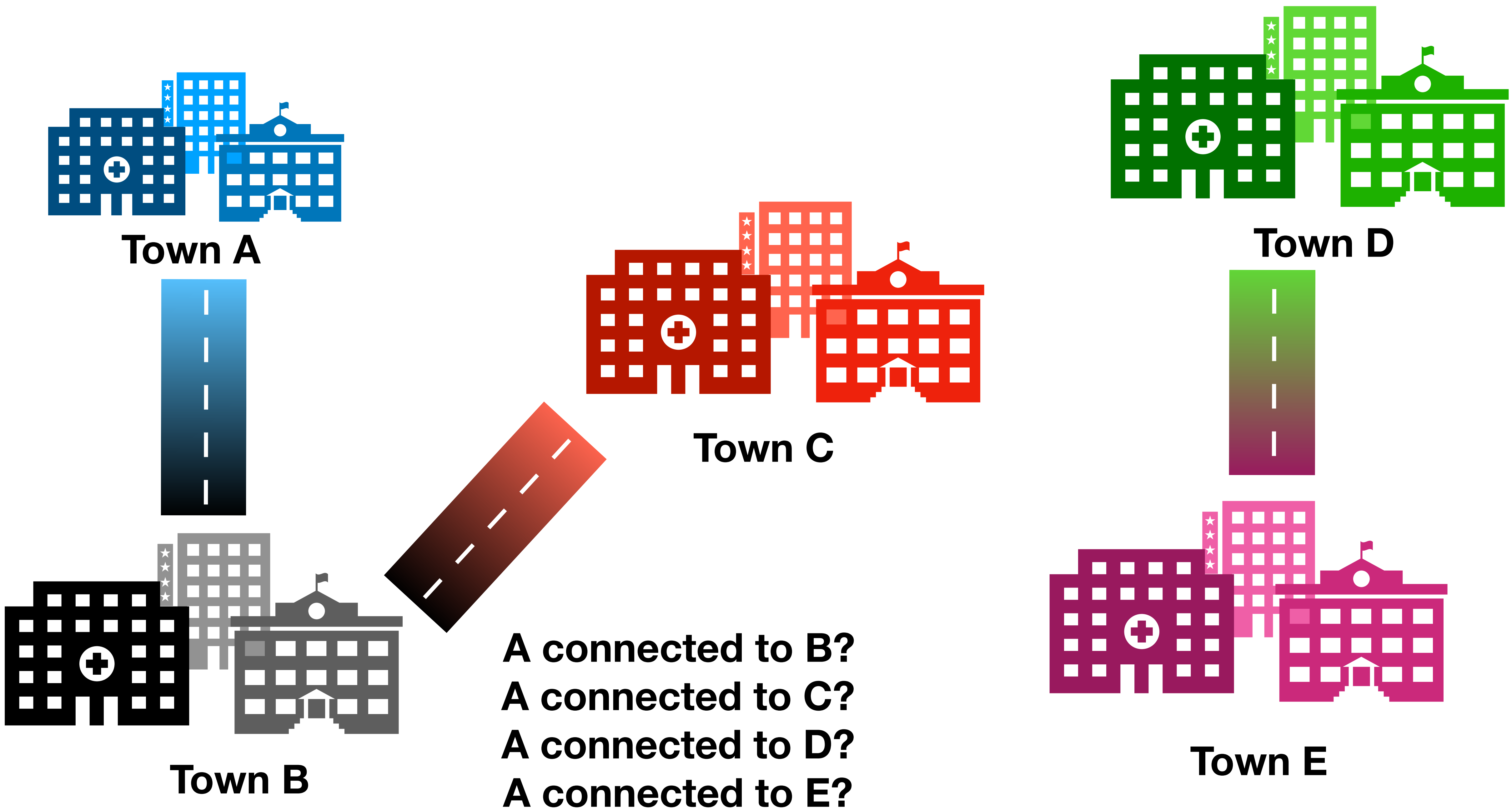






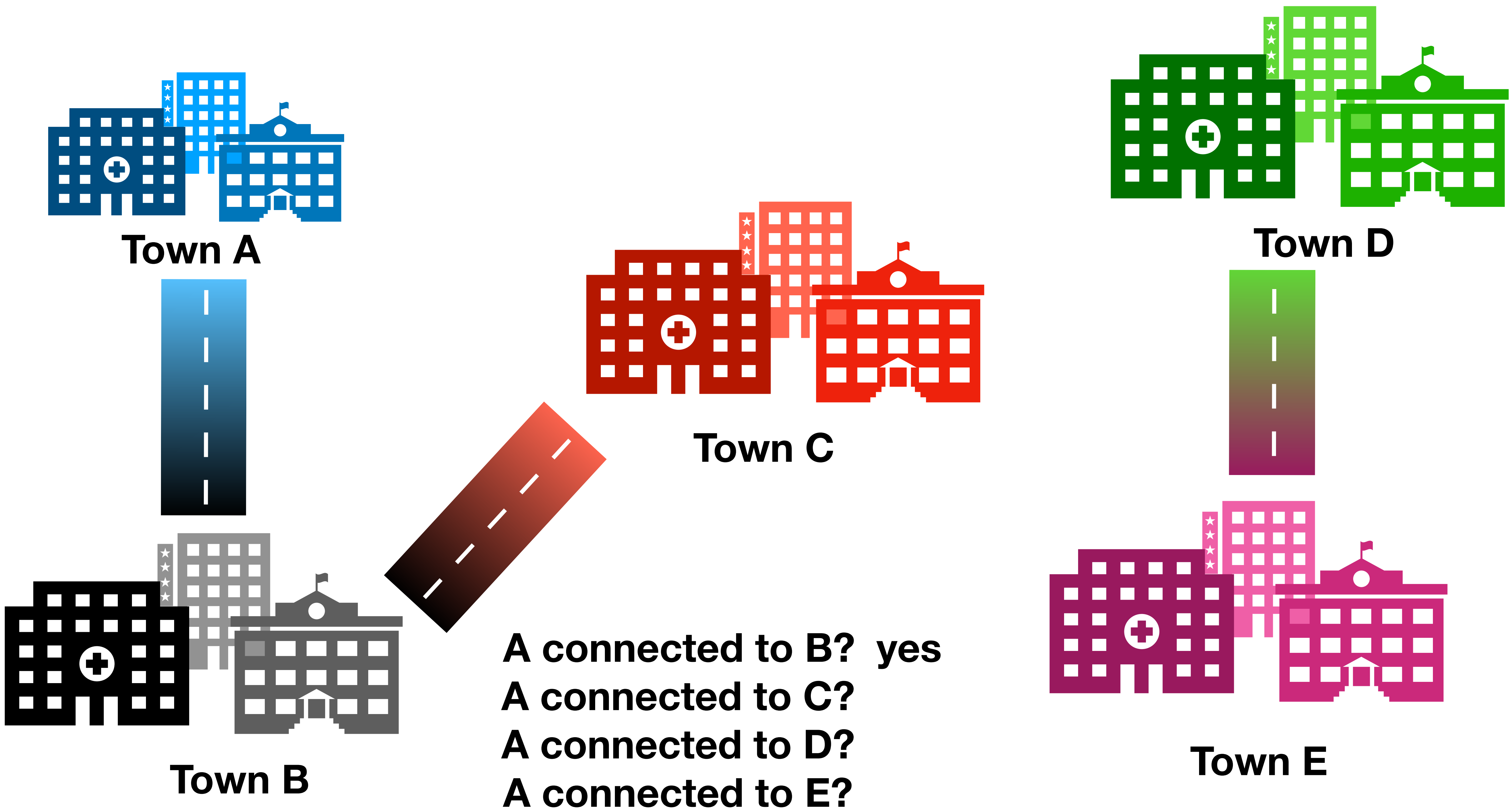
A connected to B? yes
A connected to C? no
A connected to D? no
A connected to E? no

Disjoint Sets



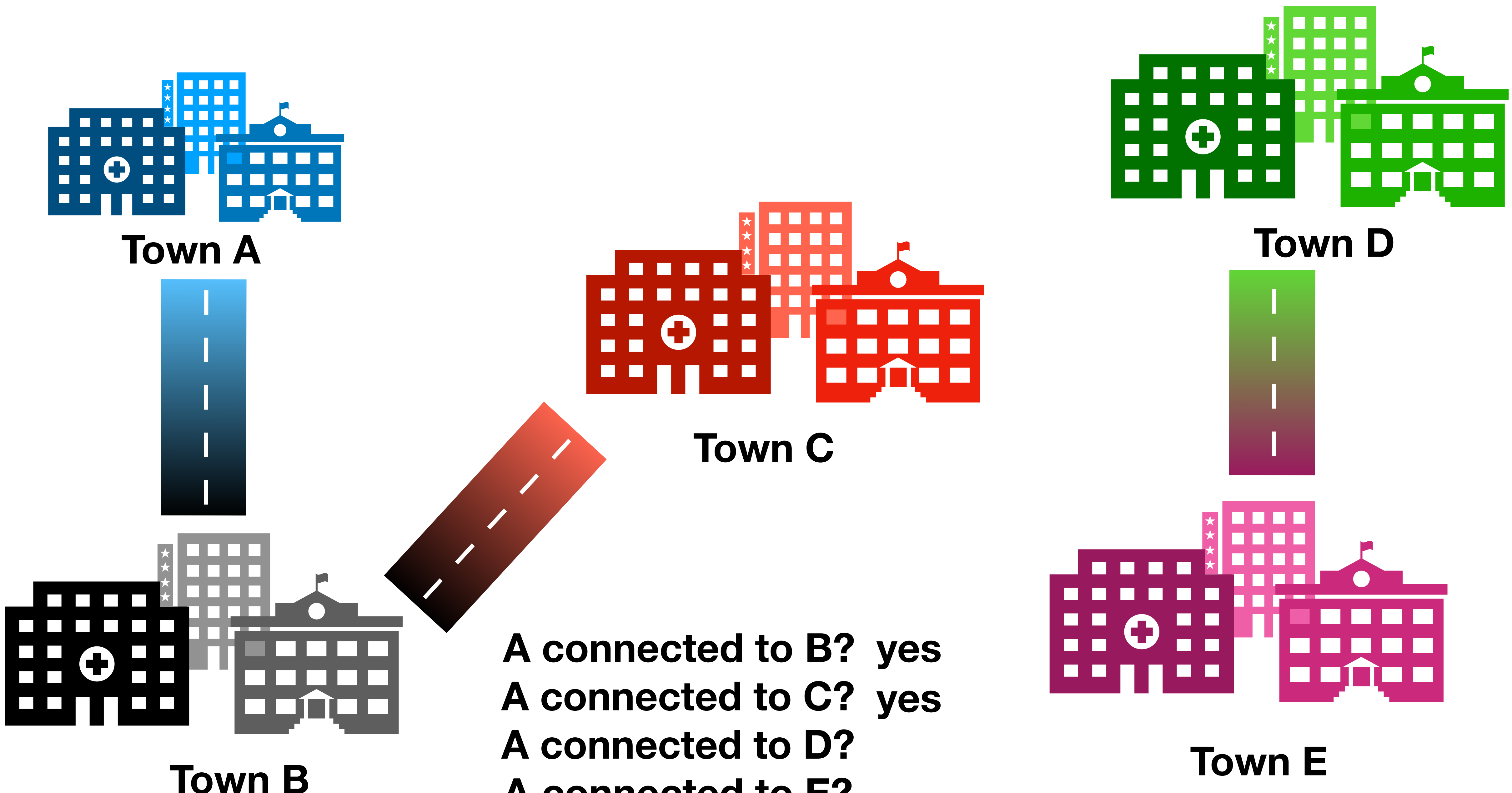
A connected to B?
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets



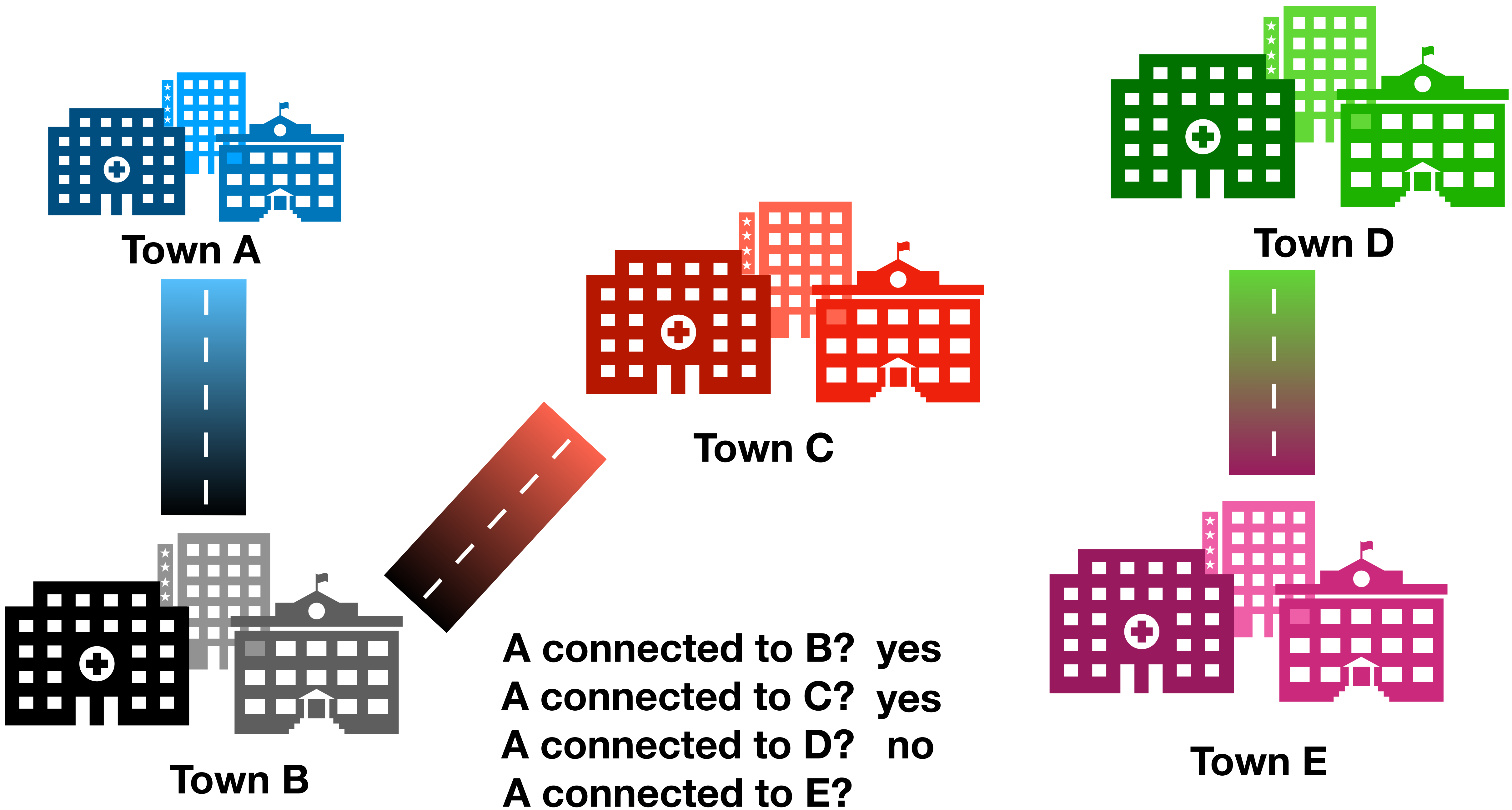
A connected to B? yes
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets

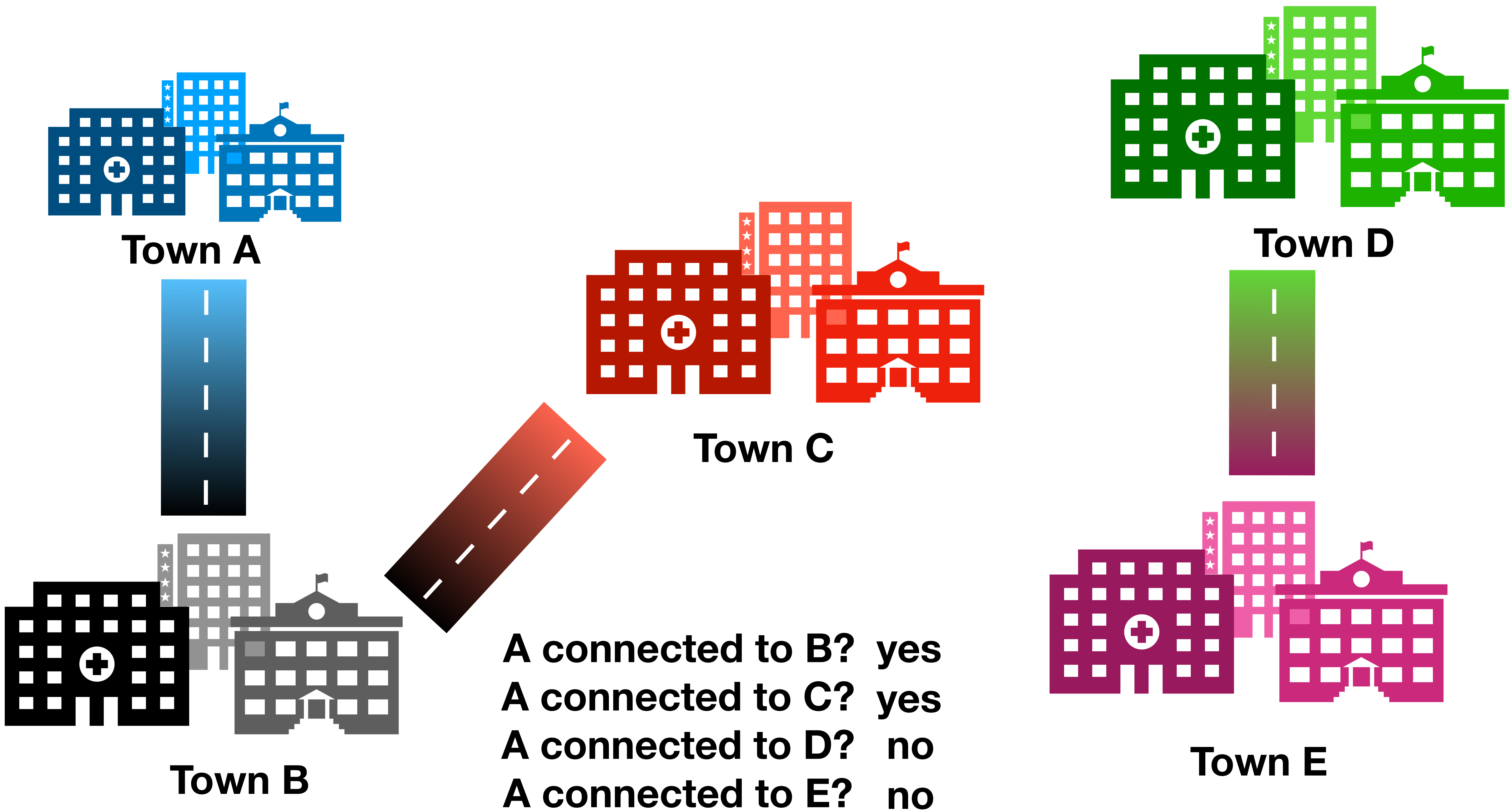


A connected to B? yes
A connected to C? yes
A connected to D?
A connected to E?

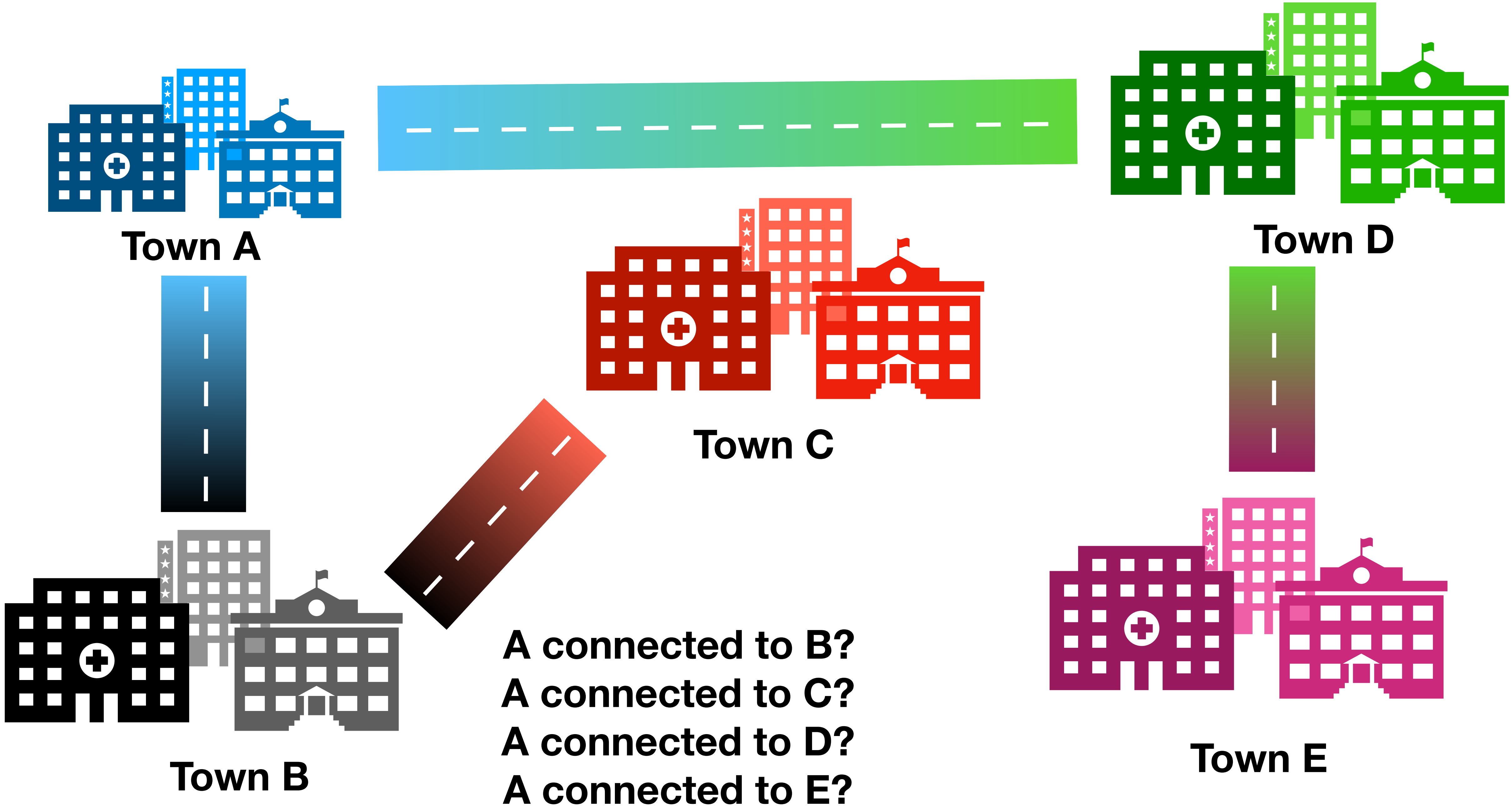
Disjoint Sets



Disjoint Sets

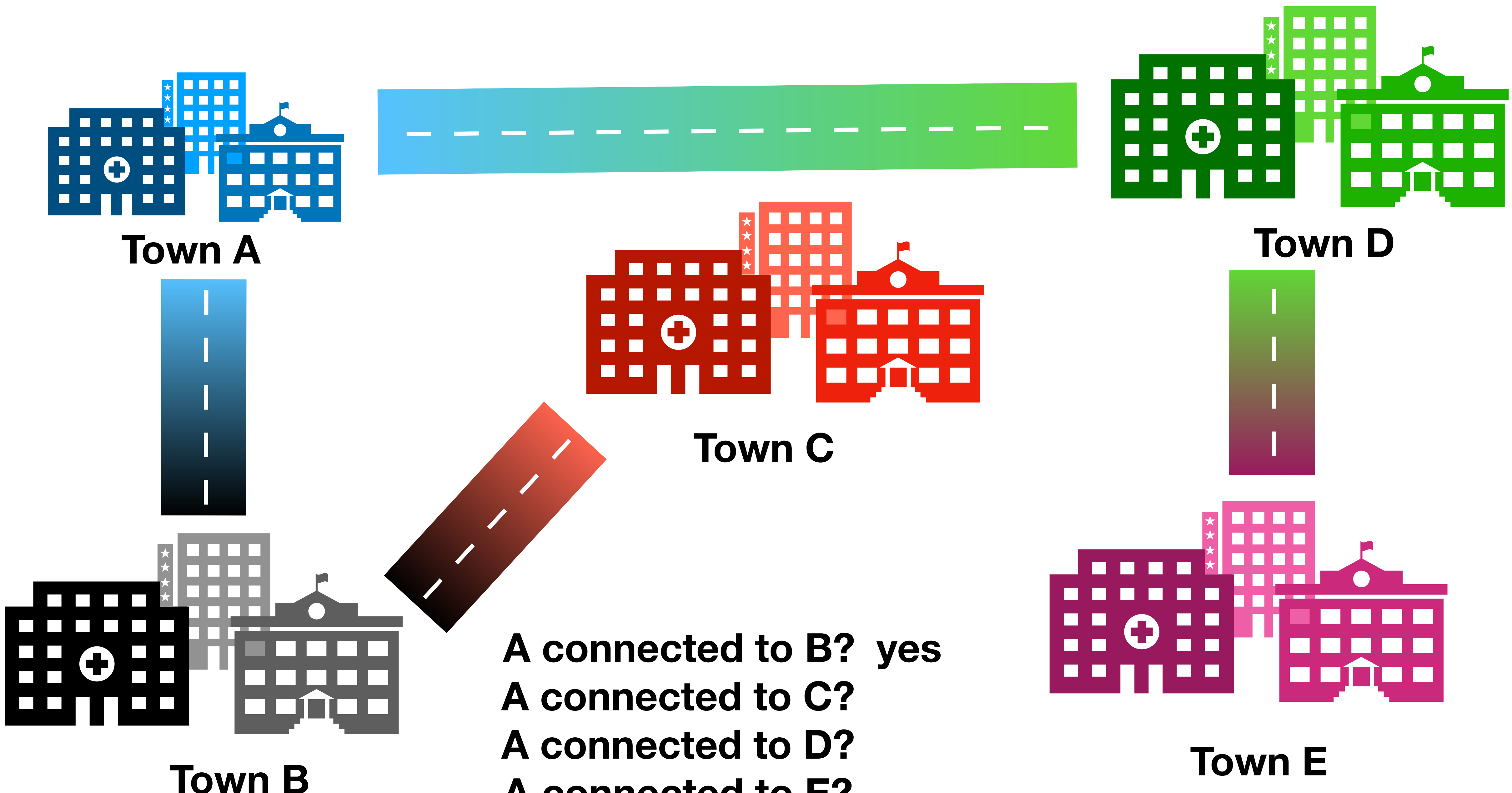


Disjoint Sets



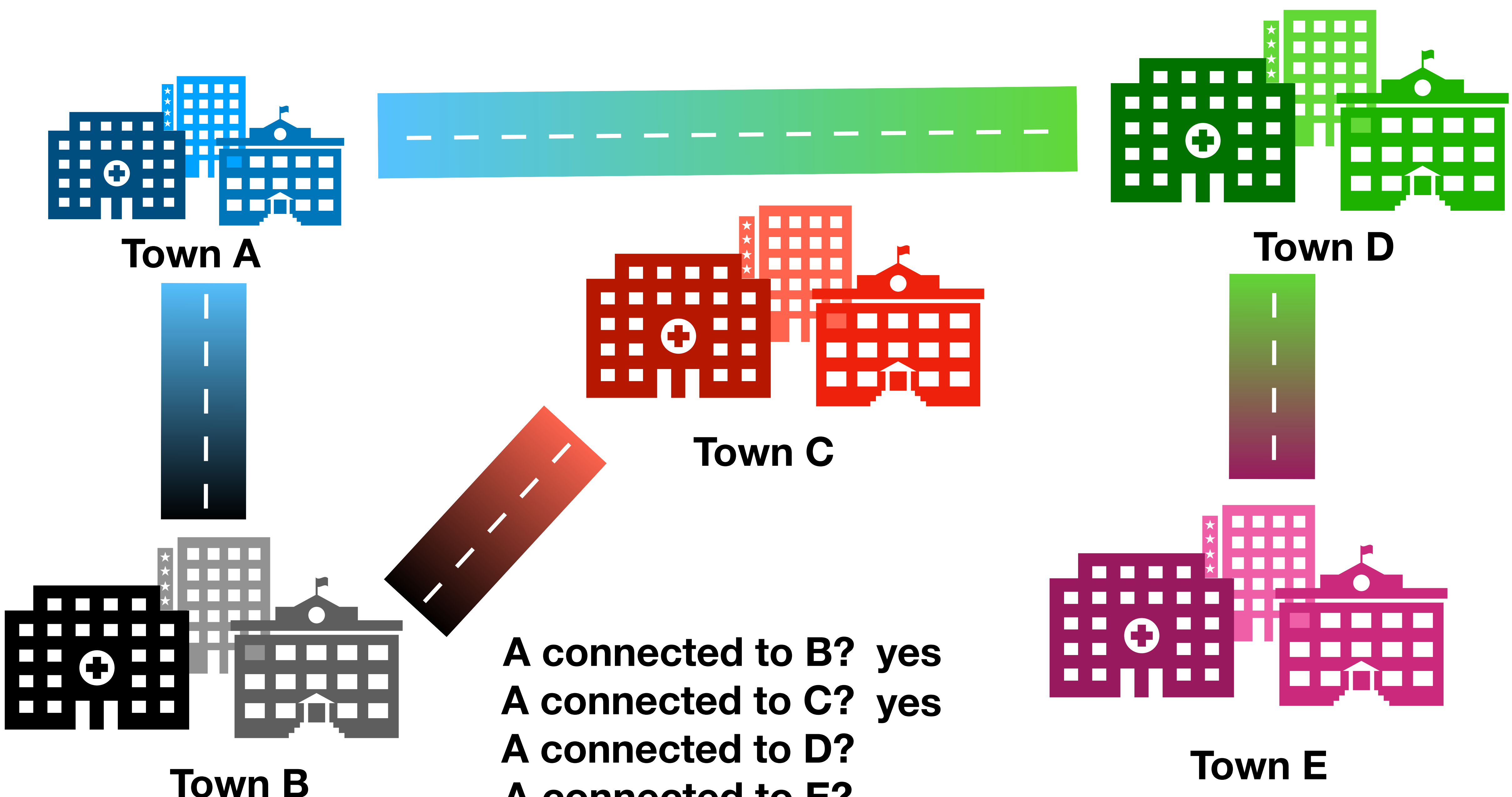
A connected to B?
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets



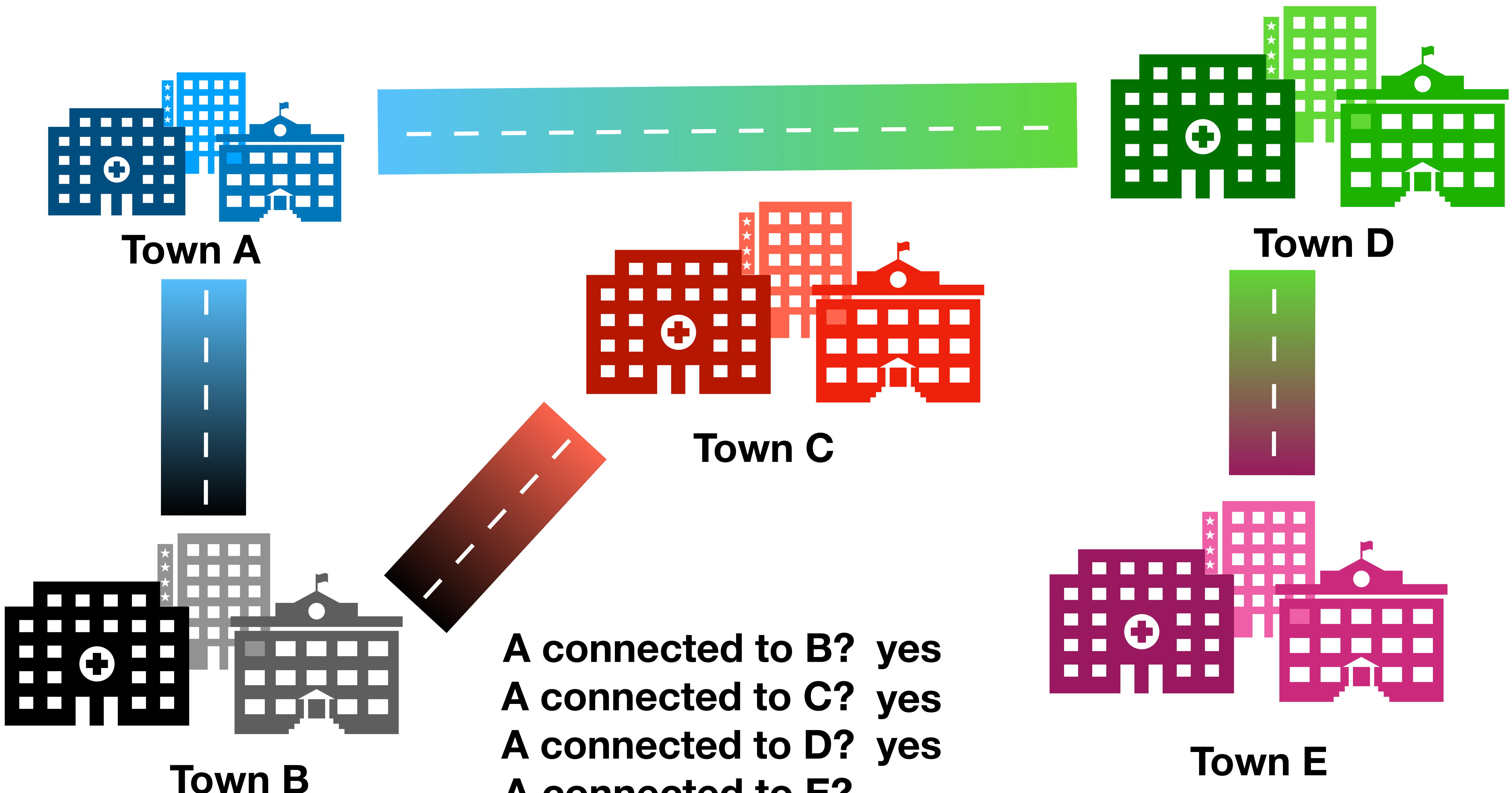
A connected to B? yes
A connected to C?
A connected to D?
A connected to E?

Disjoint Sets



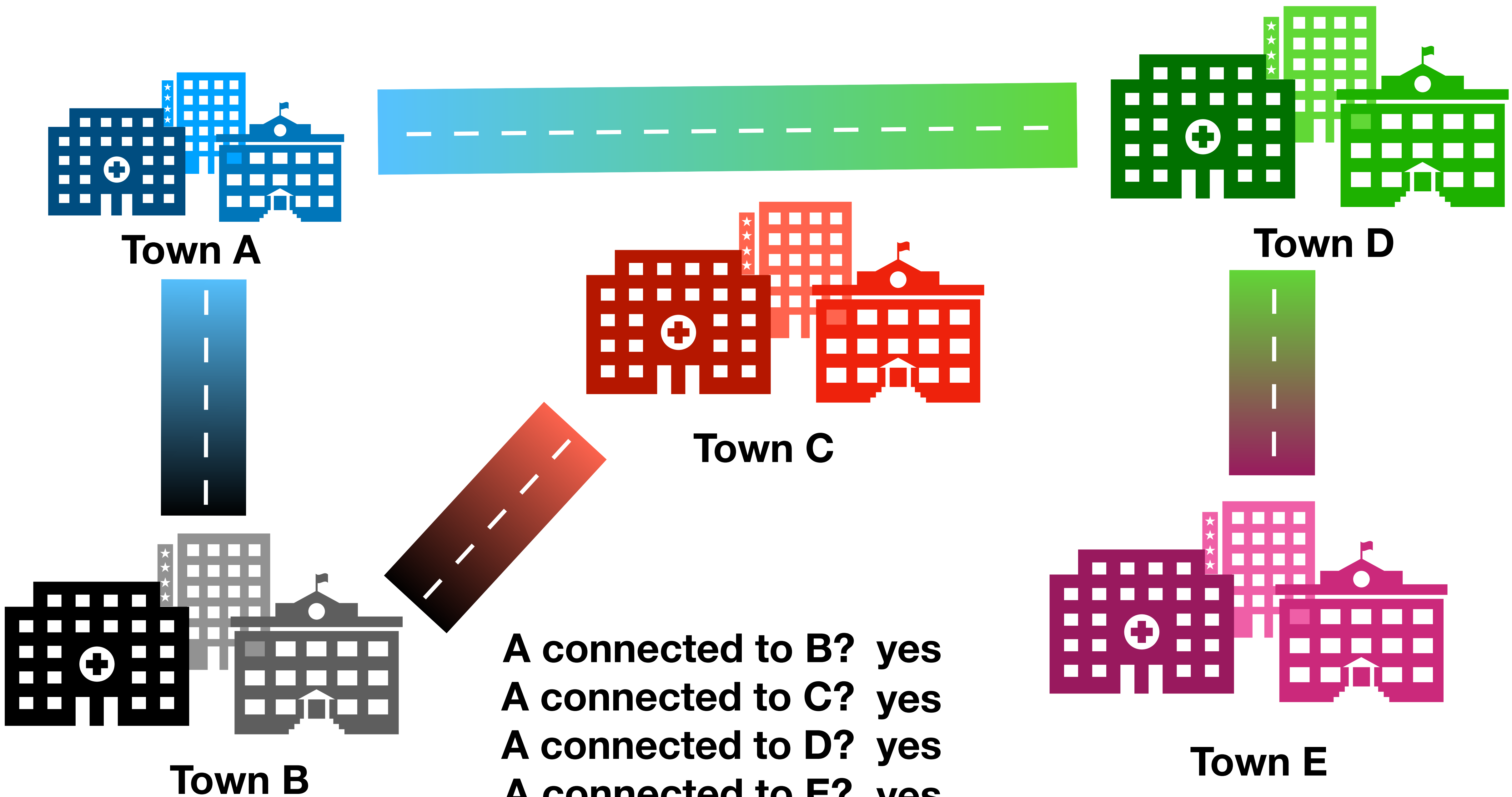
A connected to B? yes
A connected to C? yes
A connected to D?
A connected to E?

Disjoint Sets



A connected to B? yes
A connected to C? yes
A connected to D? yes
A connected to E?

Disjoint Sets



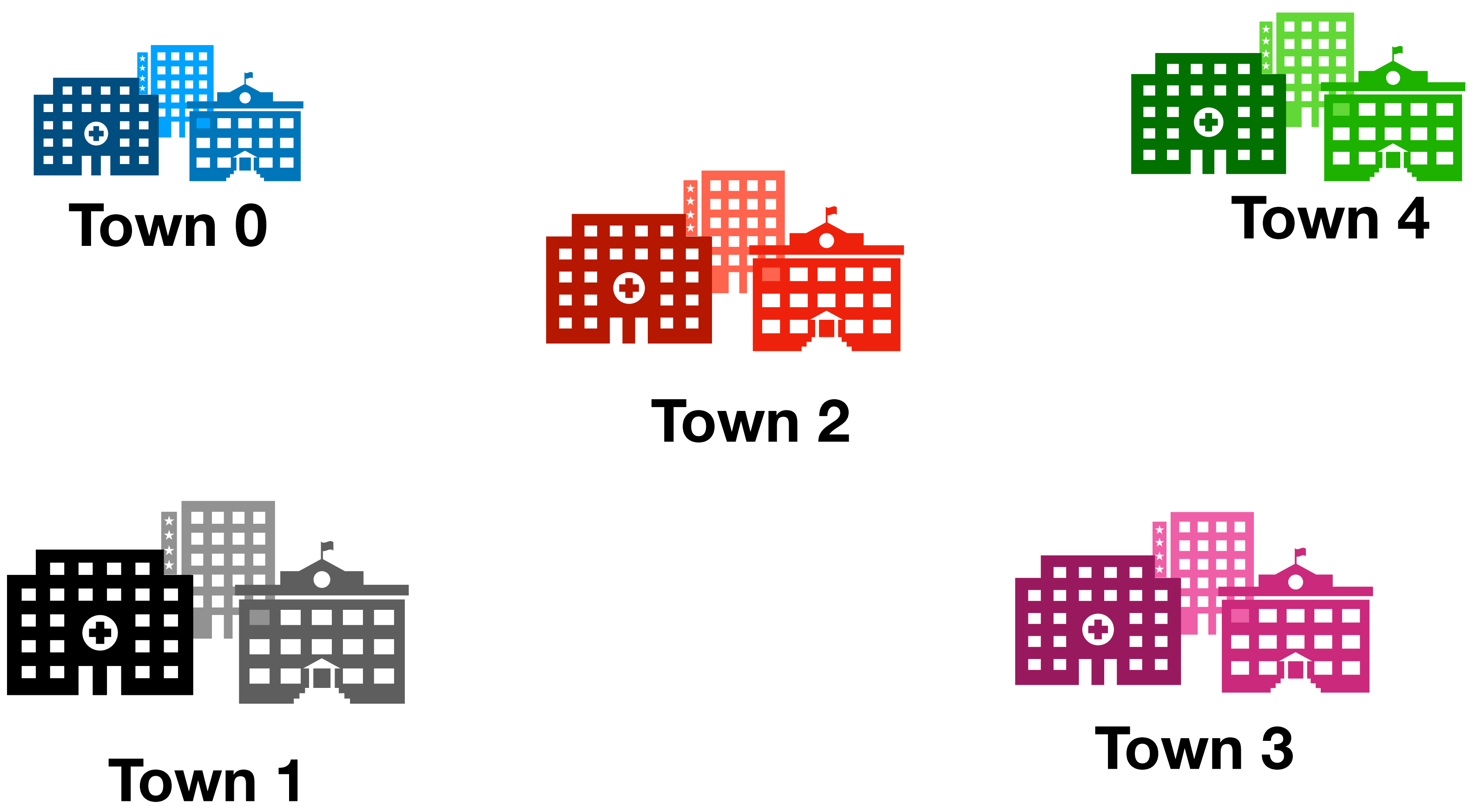
A connected to B? yes
A connected to C? yes
A connected to D? yes
A connected to E? yes

Disjoint Sets

- Given a collection of elements:
  - Join two elements ("union")
  - See which set an element belongs to ("find")
- How to do both efficiently?

Idea 1: Prioritize finds ("Quick find")

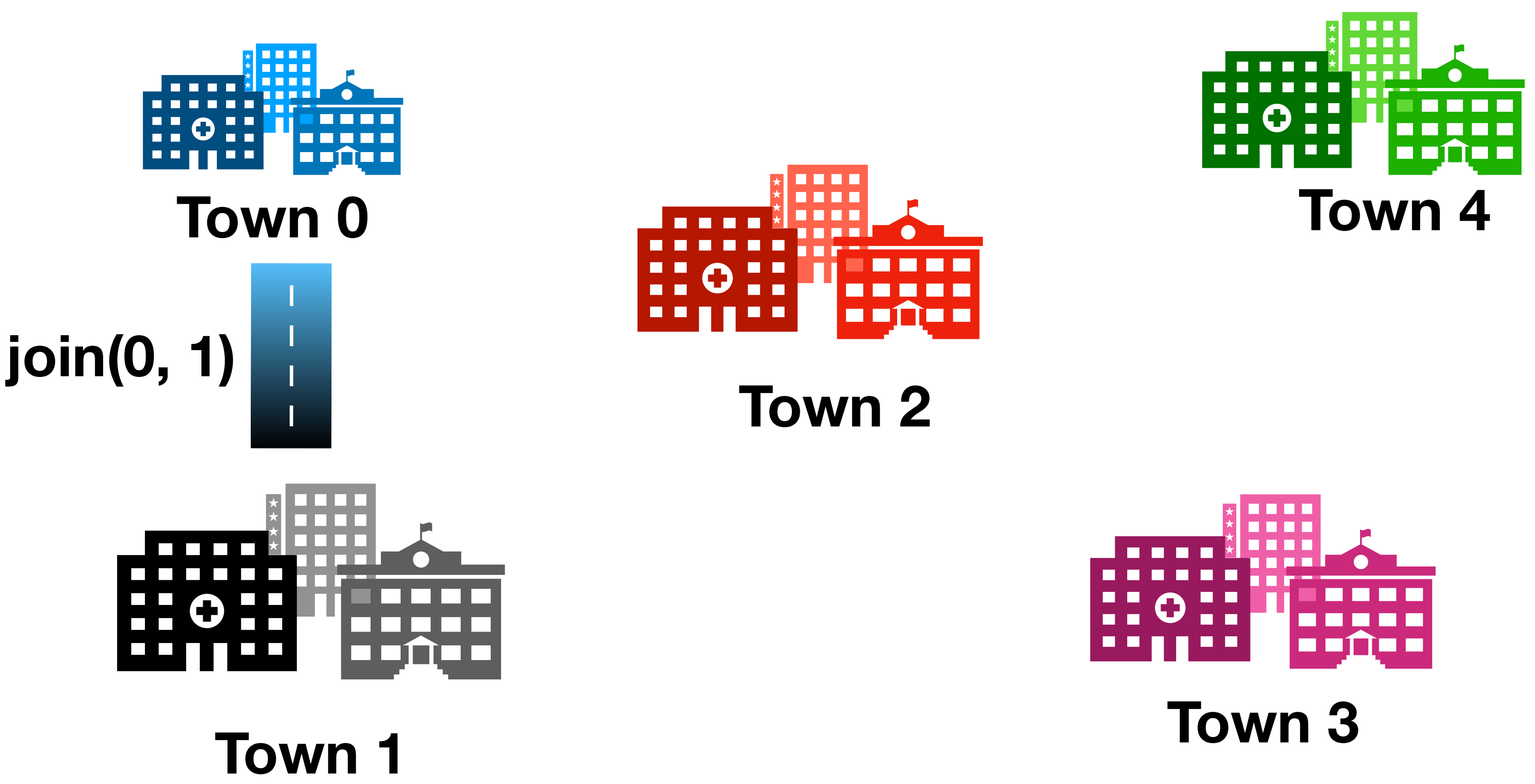
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	**1**	**2**	**3**	**4**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

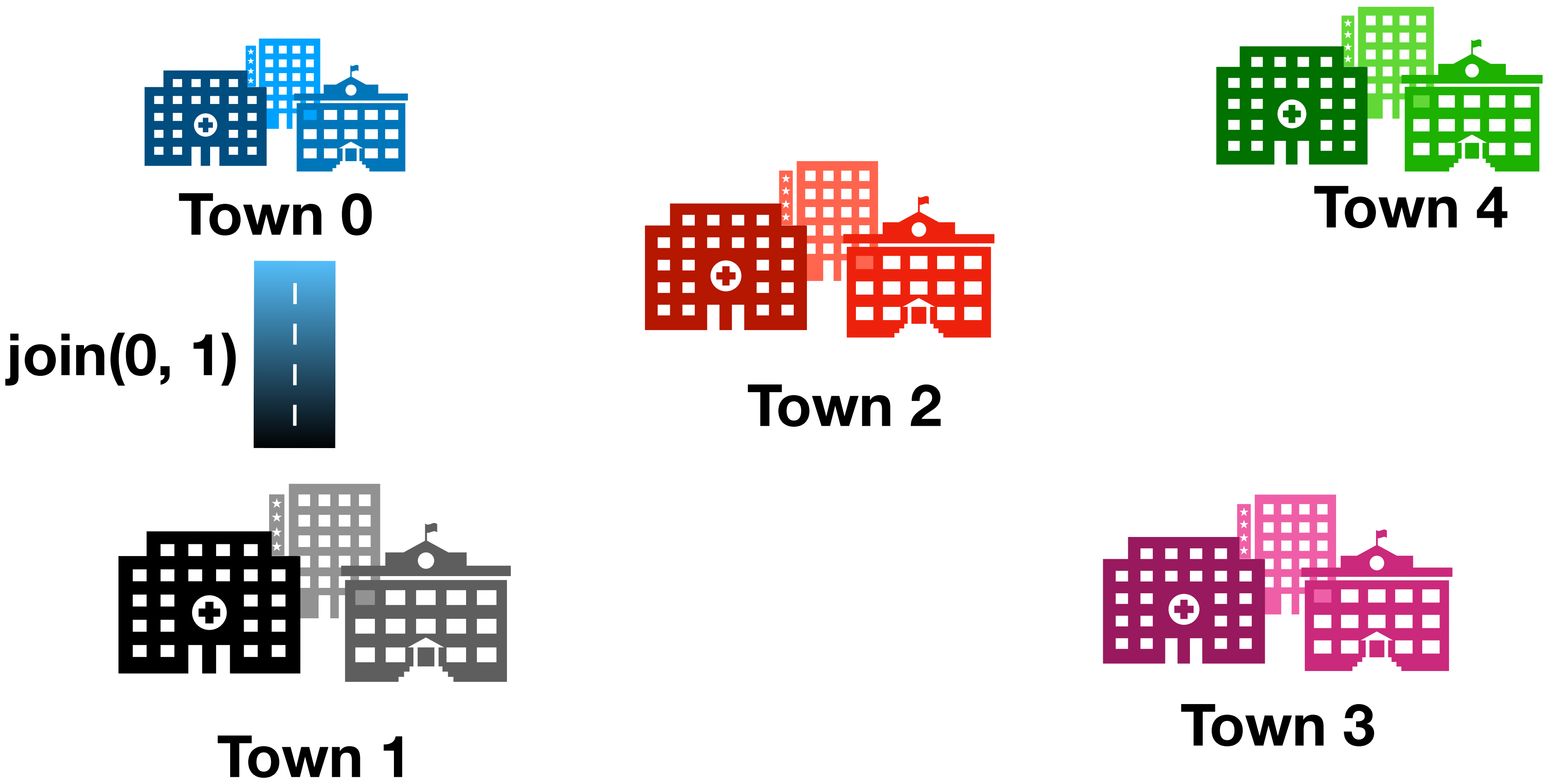
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	**1**	**2**	**3**	**4**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

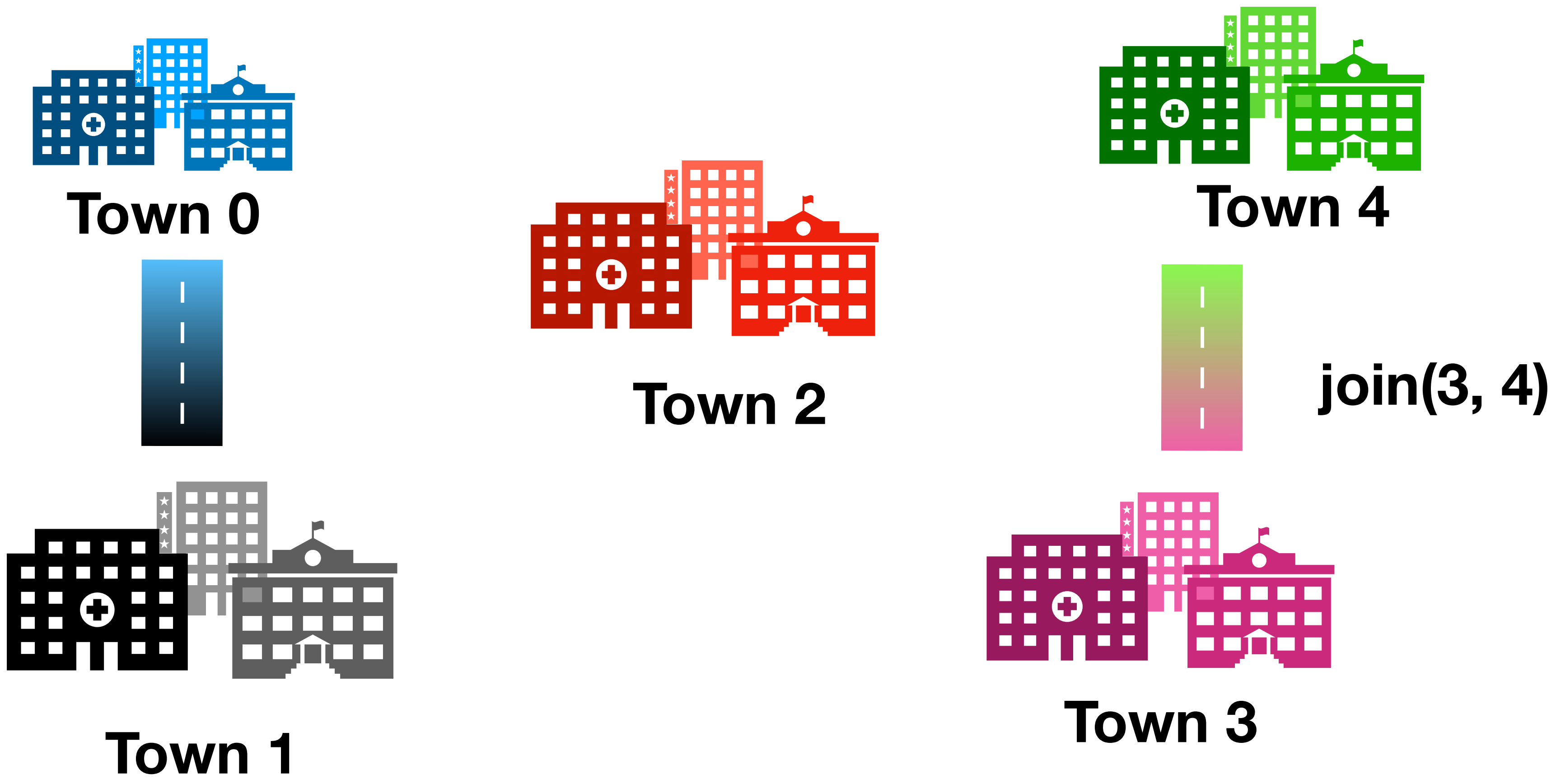
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	**0**	**2**	**3**	**4**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

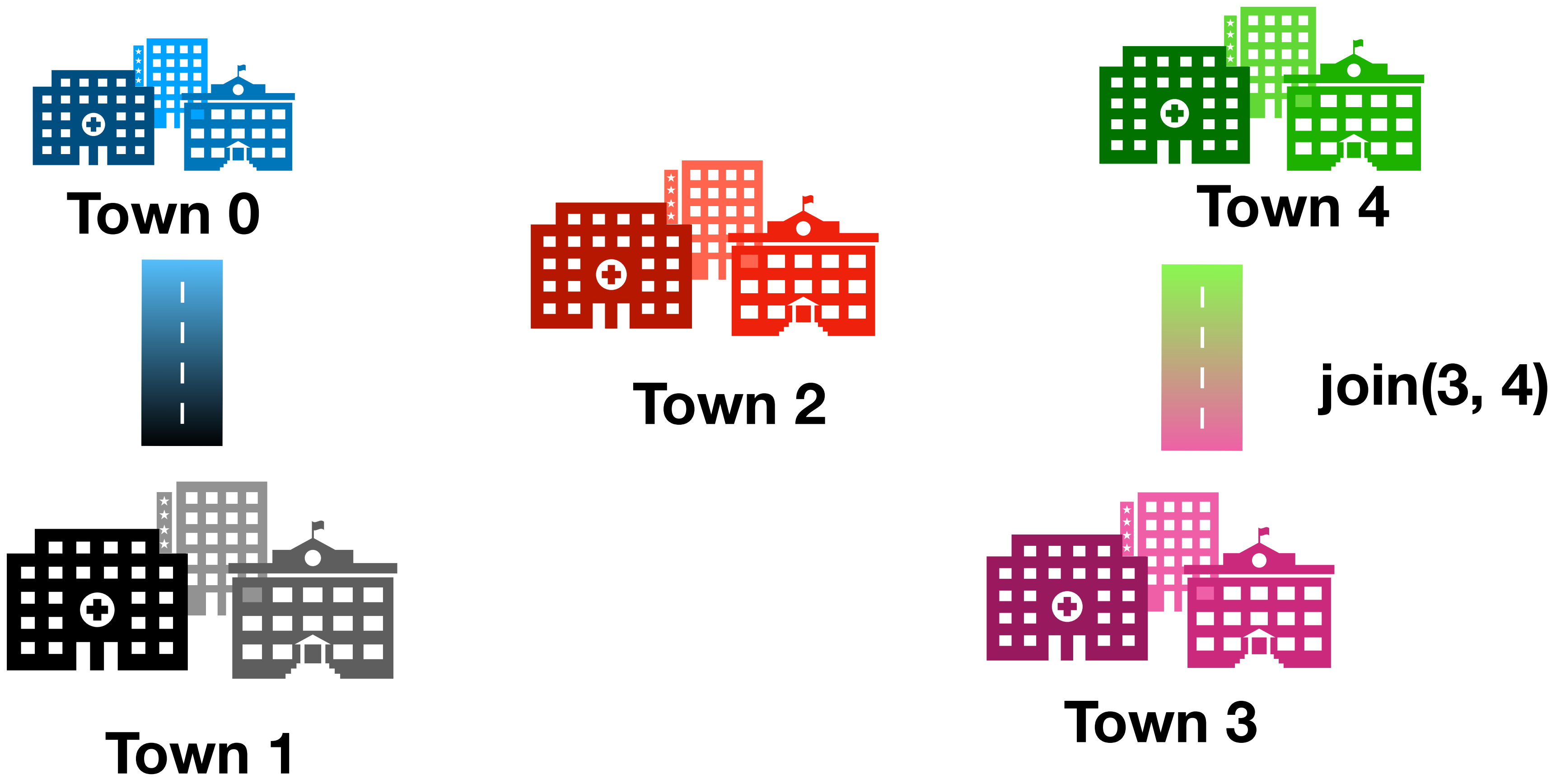
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	**0**	**2**	**3**	**4**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

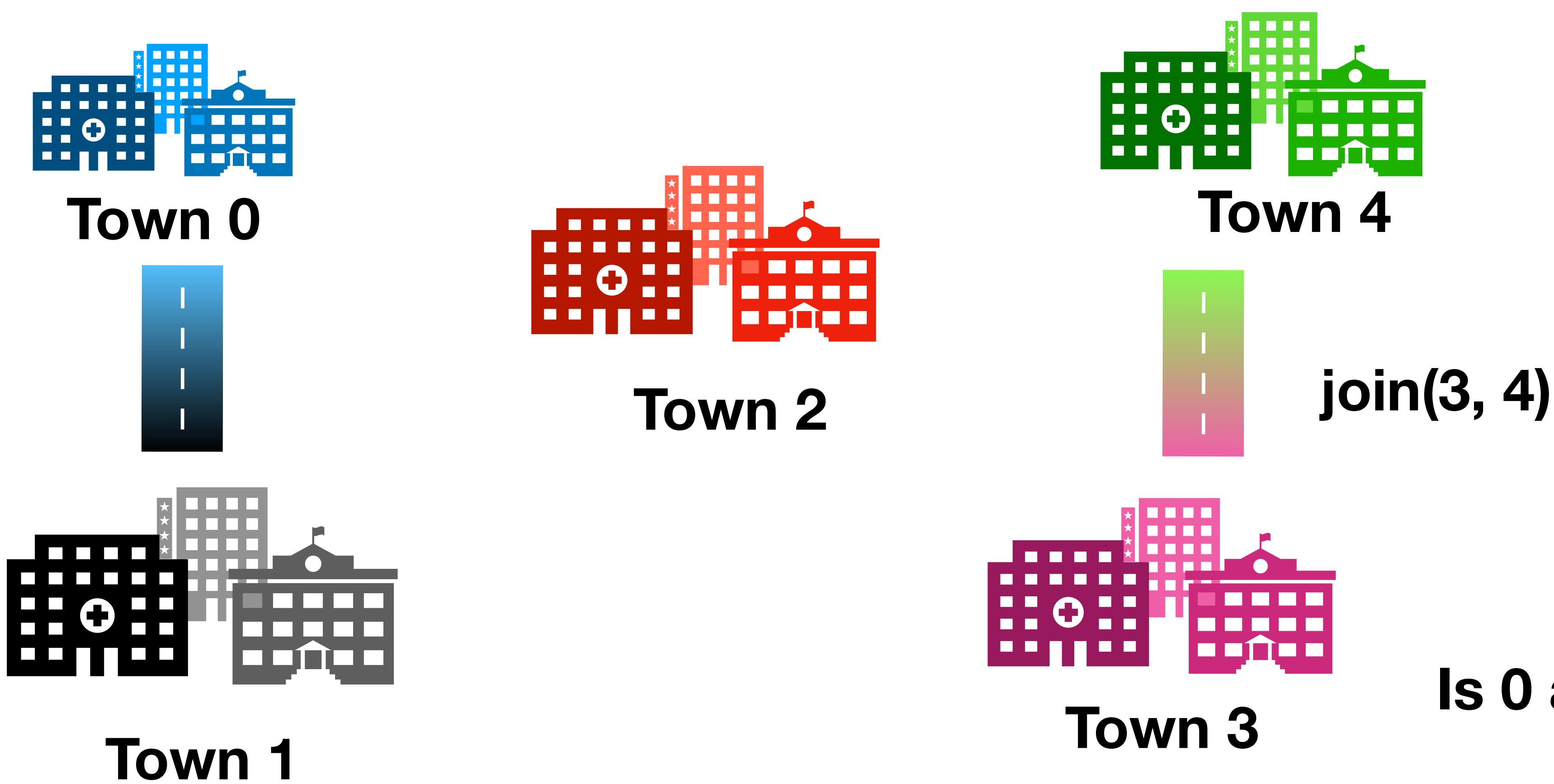
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	0	2	3	3
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.

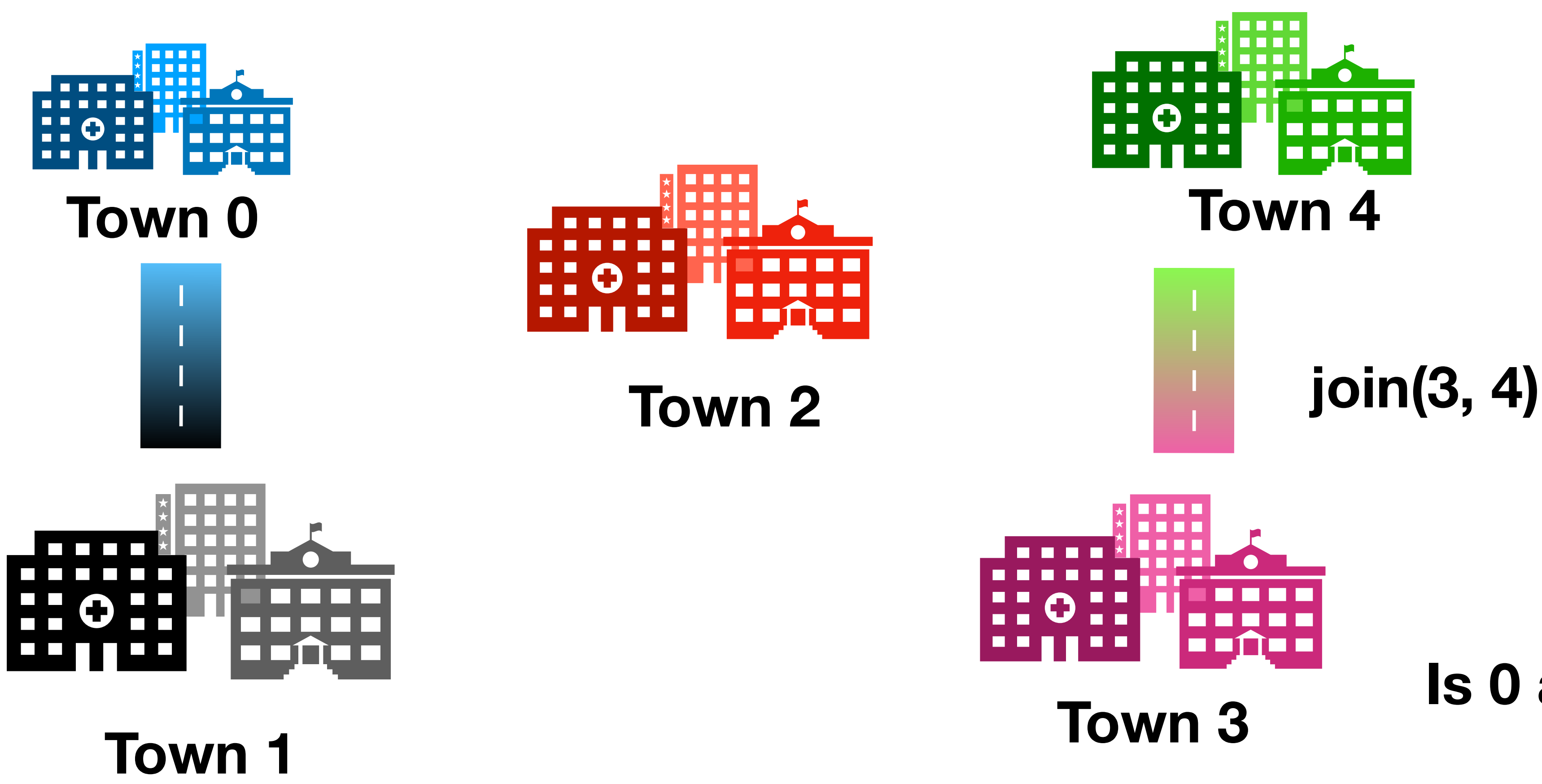


Is 0 and 1 connected?

0	**0**	**2**	**3**	**3**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



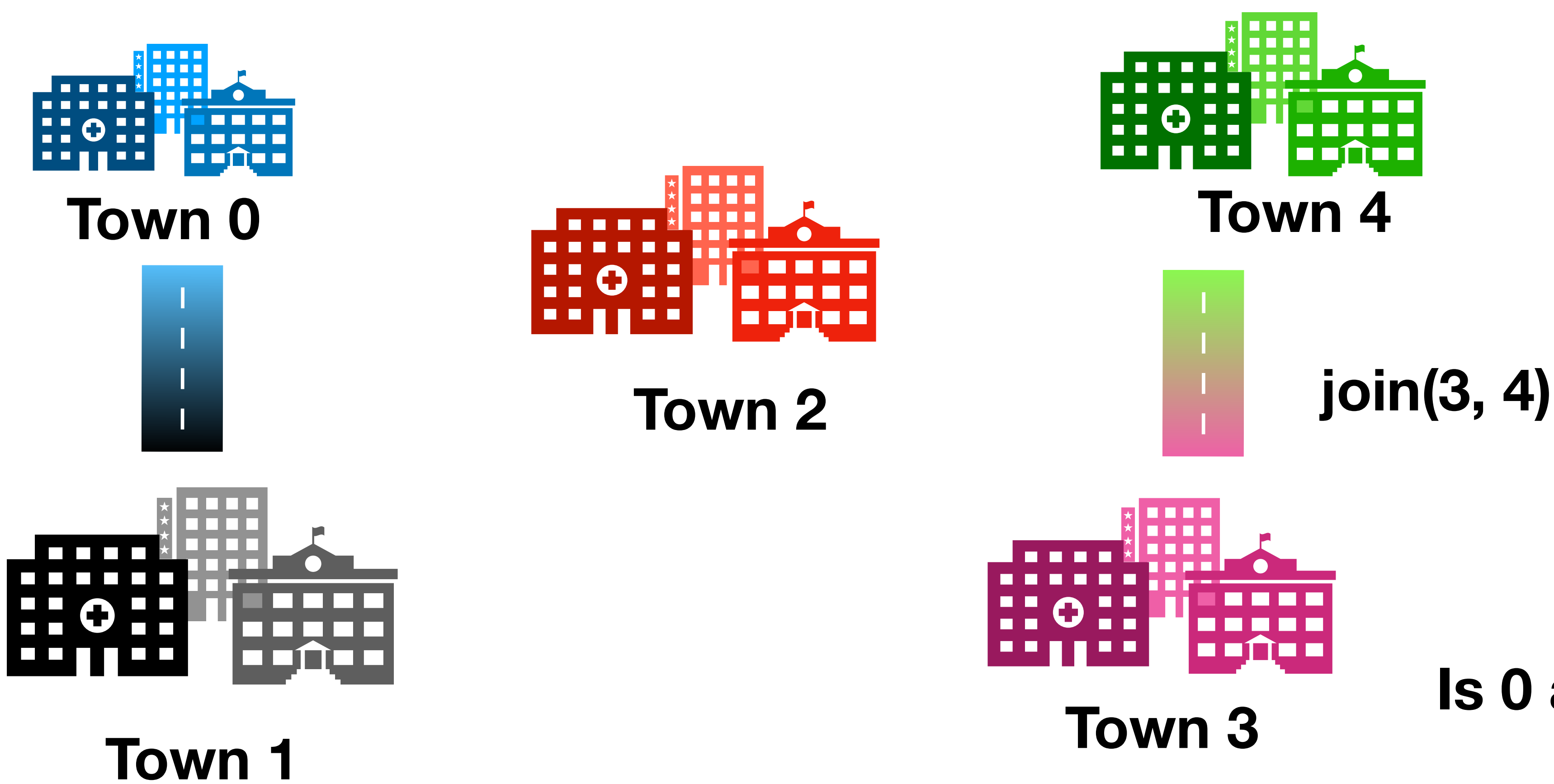
Is 0 and 1 connected?

Yes. 0 == 0.

0	**0**	**2**	**3**	**3**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



Is 0 and 1 connected?

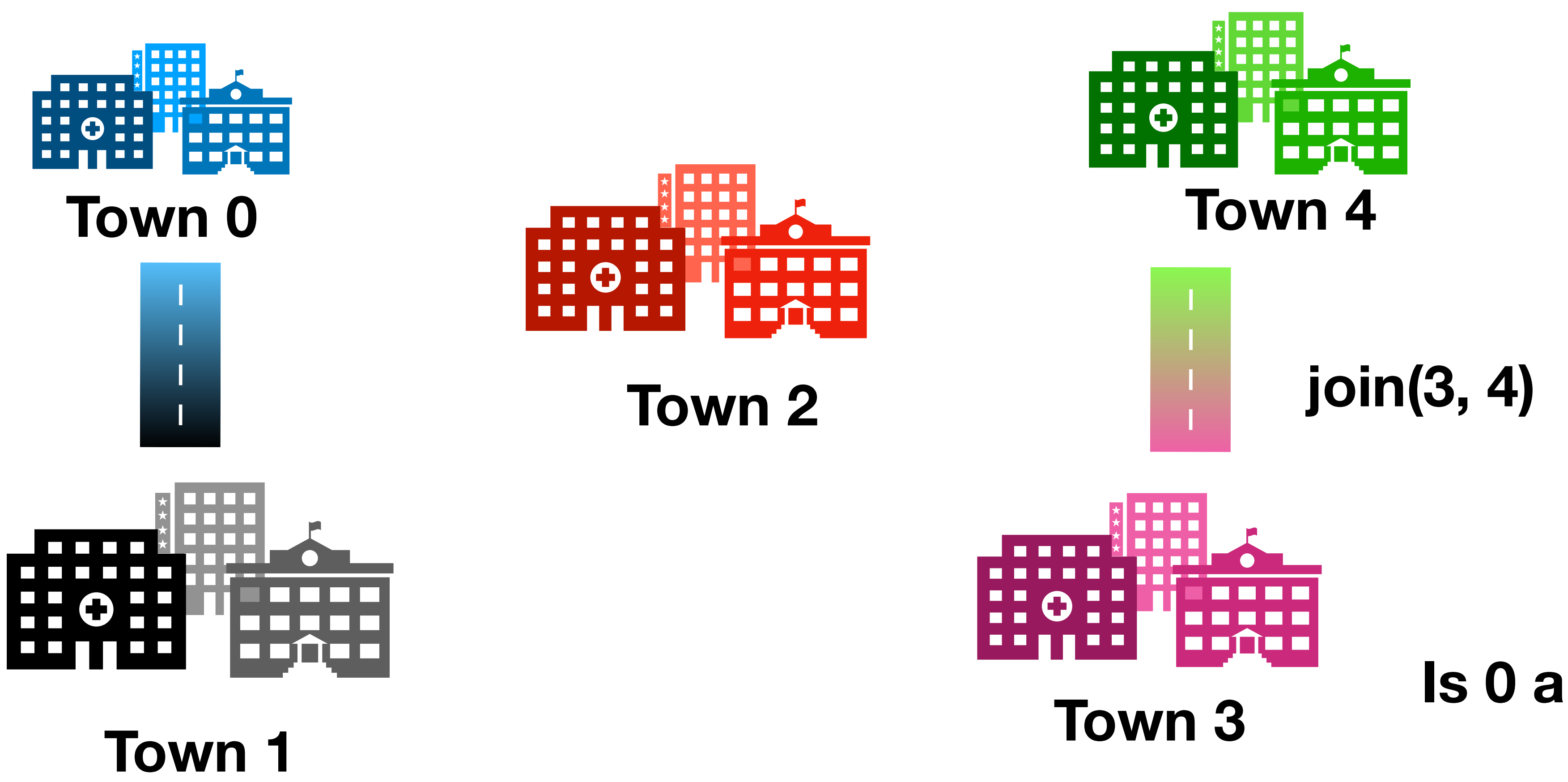
Yes. 0 == 0.

Is 0 and 4 connected?

0	0	2	3	3
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



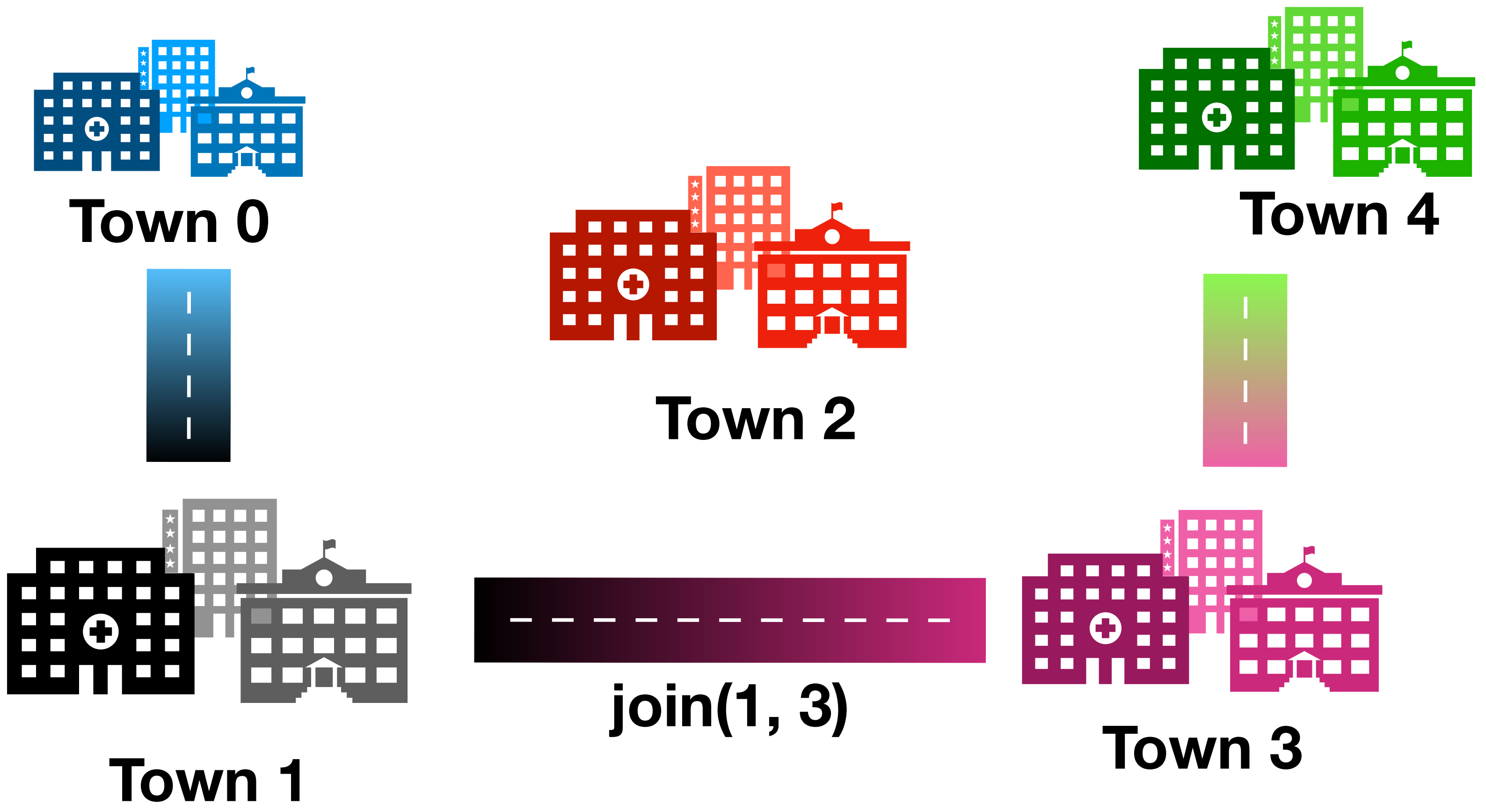
Is 0 and 1 connected?
Yes. 0 == 0.

Is 0 and 4 connected?
No. 0 != 3

0	**0**	**2**	**3**	**3**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

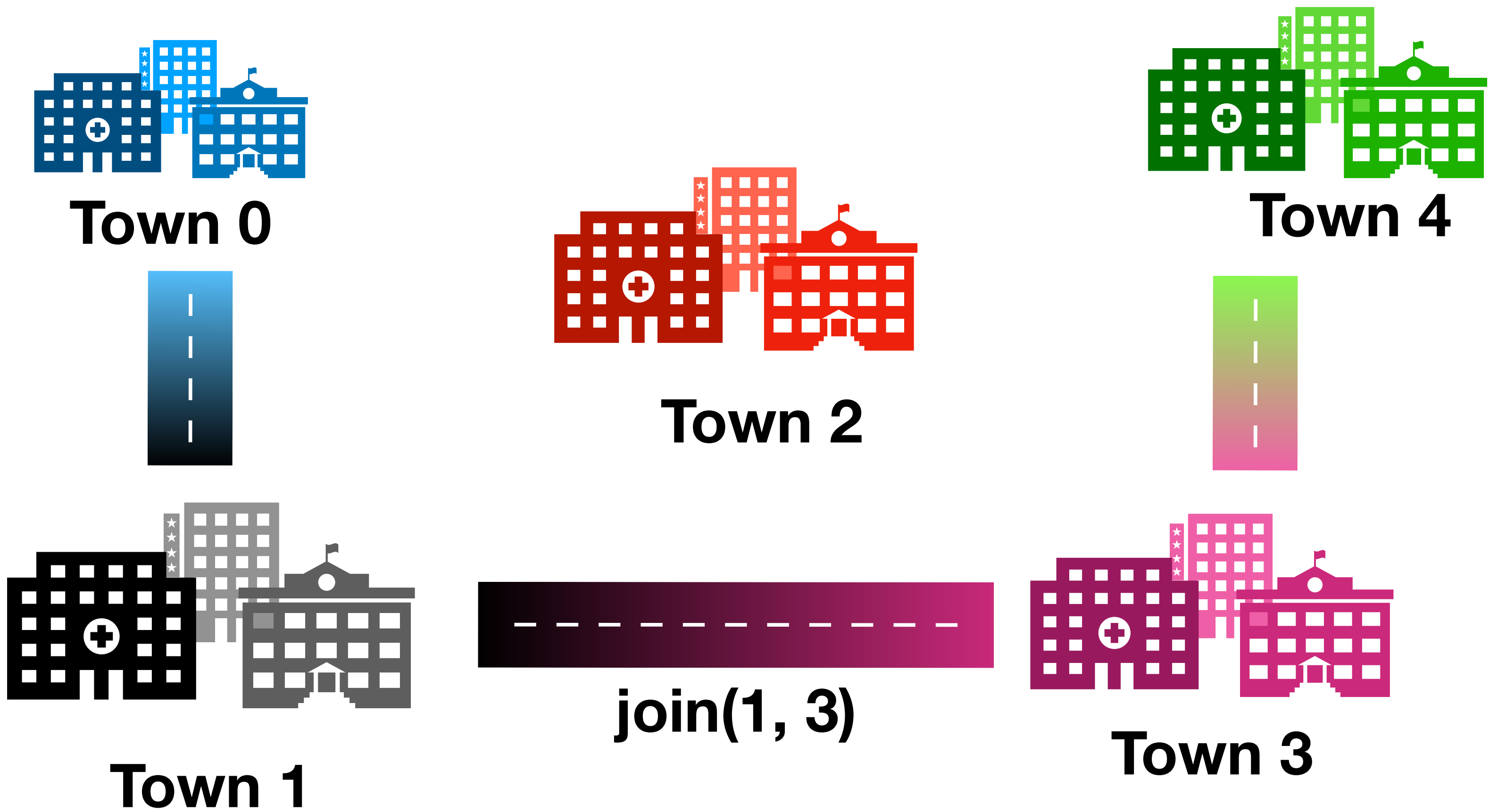
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	**0**	**2**	**3**	**3**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

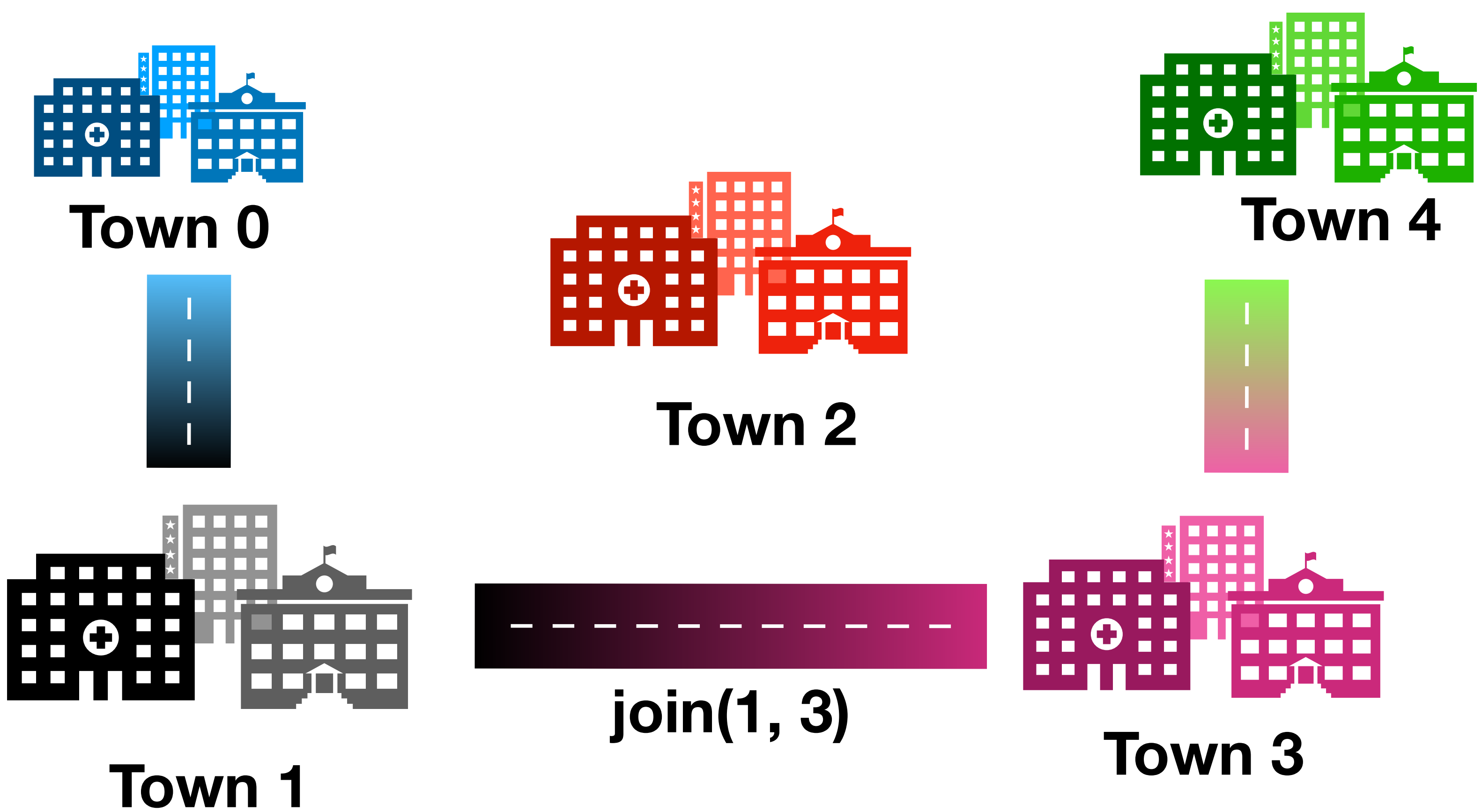
Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



0	0	2	0	0
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.

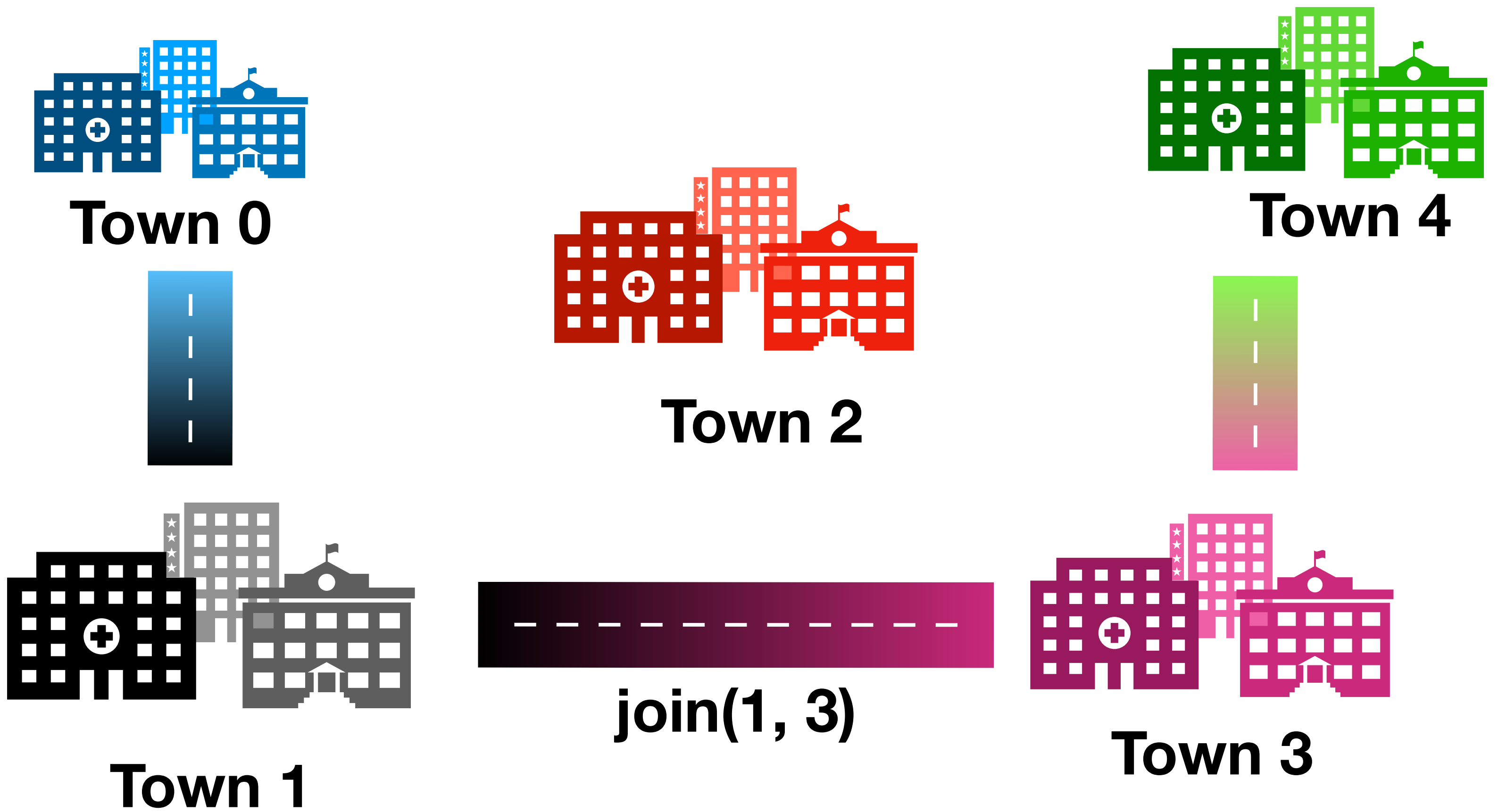


Is 0 and 1 connected?

0	1	2	3	4
0	**0**	**2**	**0**	**0**

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



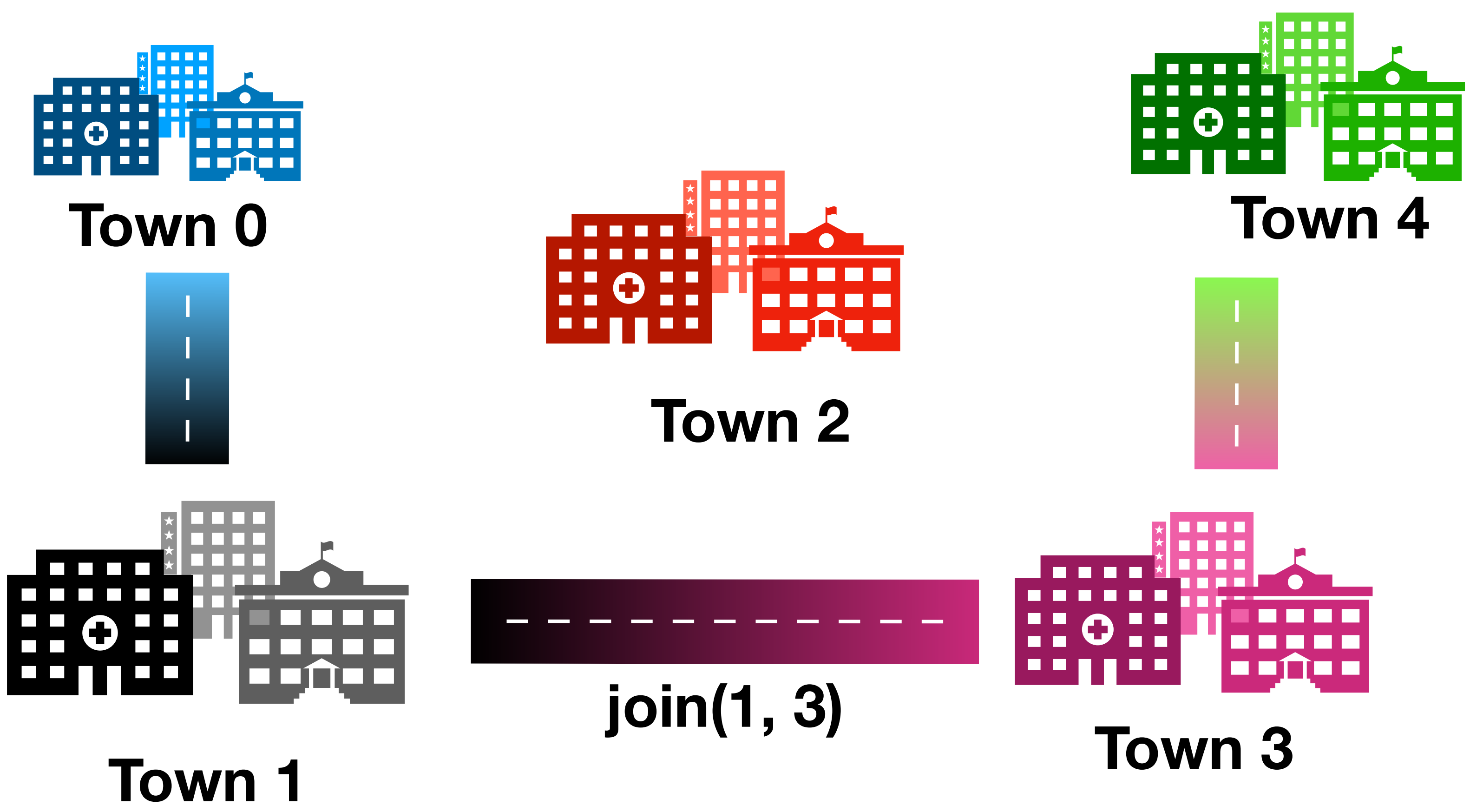
Is 0 and 1 connected?

Yes. 0 == 0.

0	**0**	**2**	**0**	**0**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



Is 0 and 1 connected?

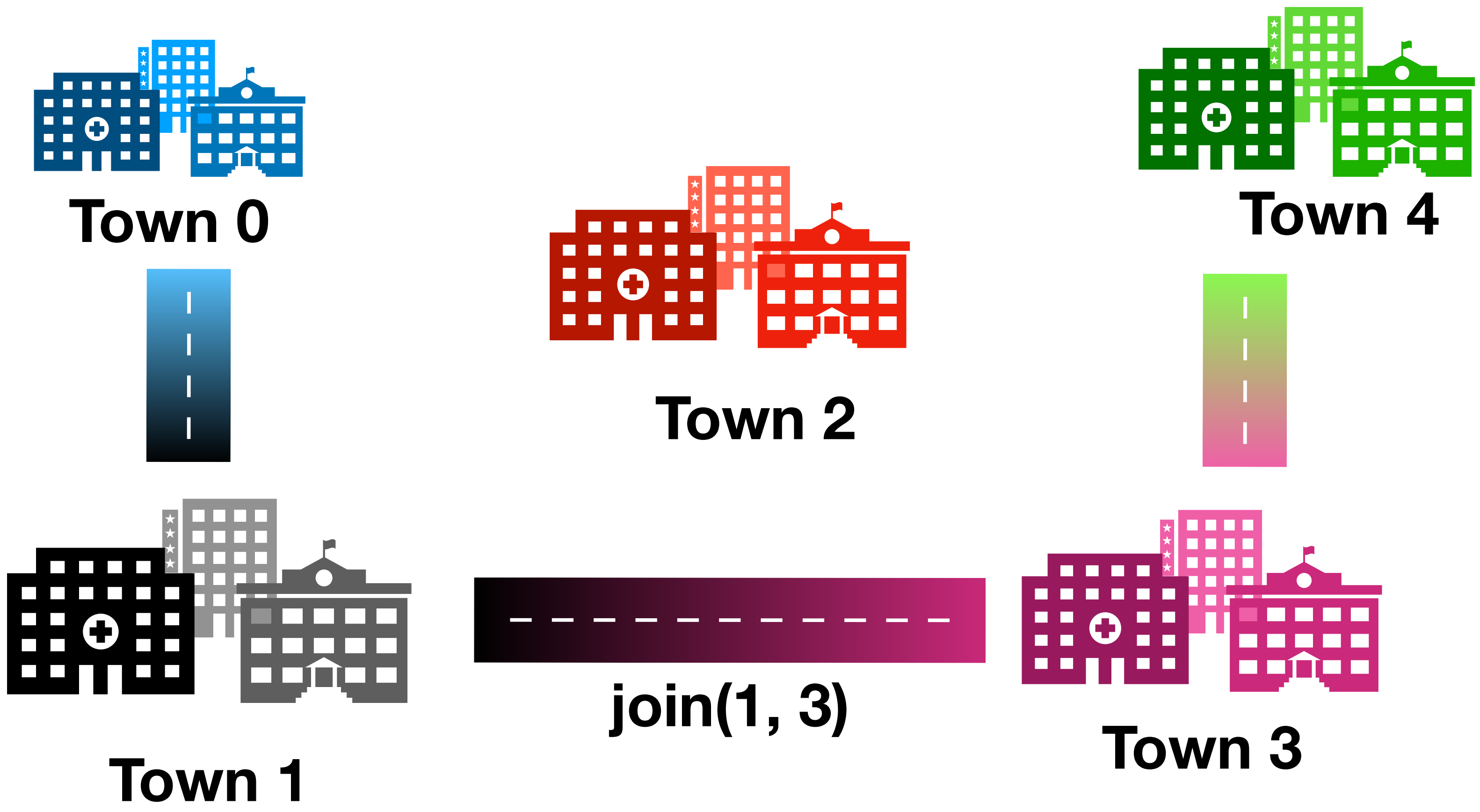
Yes. 0 == 0.

Is 0 and 4 connected?

0	**0**	**2**	**0**	**0**
0	1	2	3	4

Idea 1: Prioritize finds ("Quick find")

Assign each connected set a number. When unioning two sets, set all elements in those two sets to the same group number.



Is 0 and 1 connected?

Yes. 0 == 0.

Is 0 and 4 connected?

Yes. 0 == 0.

0	1	2	3	4
0	**0**	**2**	**0**	**0**

QuickFind Pseudocode

```
class QuickFindDS:
  int[] id

  QuickFindDS(N):
    id = new int[N]
    for i in 0...N:
      id[i] = i

  find(a):
    return id[a]

  union(a, b):
    let aId = id[a]
    let bId = id[b]
    for i in 0...N:
      if id[i] == a_id:
        id[i] = b_id
```

Runtime?

Runtime?

QuickFind Pseudocode

```
class QuickFindDS:
  int[] id

  QuickFindDS(N):
    id = new int[N]
    for i in 0...N:
      id[i] = i

  find(a):
    return id[a]

  union(a, b):
    let aId = id[a]
    let bId = id[b]
    for i in 0...N:
      if id[i] == a_id:
        id[i] = b_id
```

Runtime? $\Theta(1)$

Runtime?

QuickFind Pseudocode

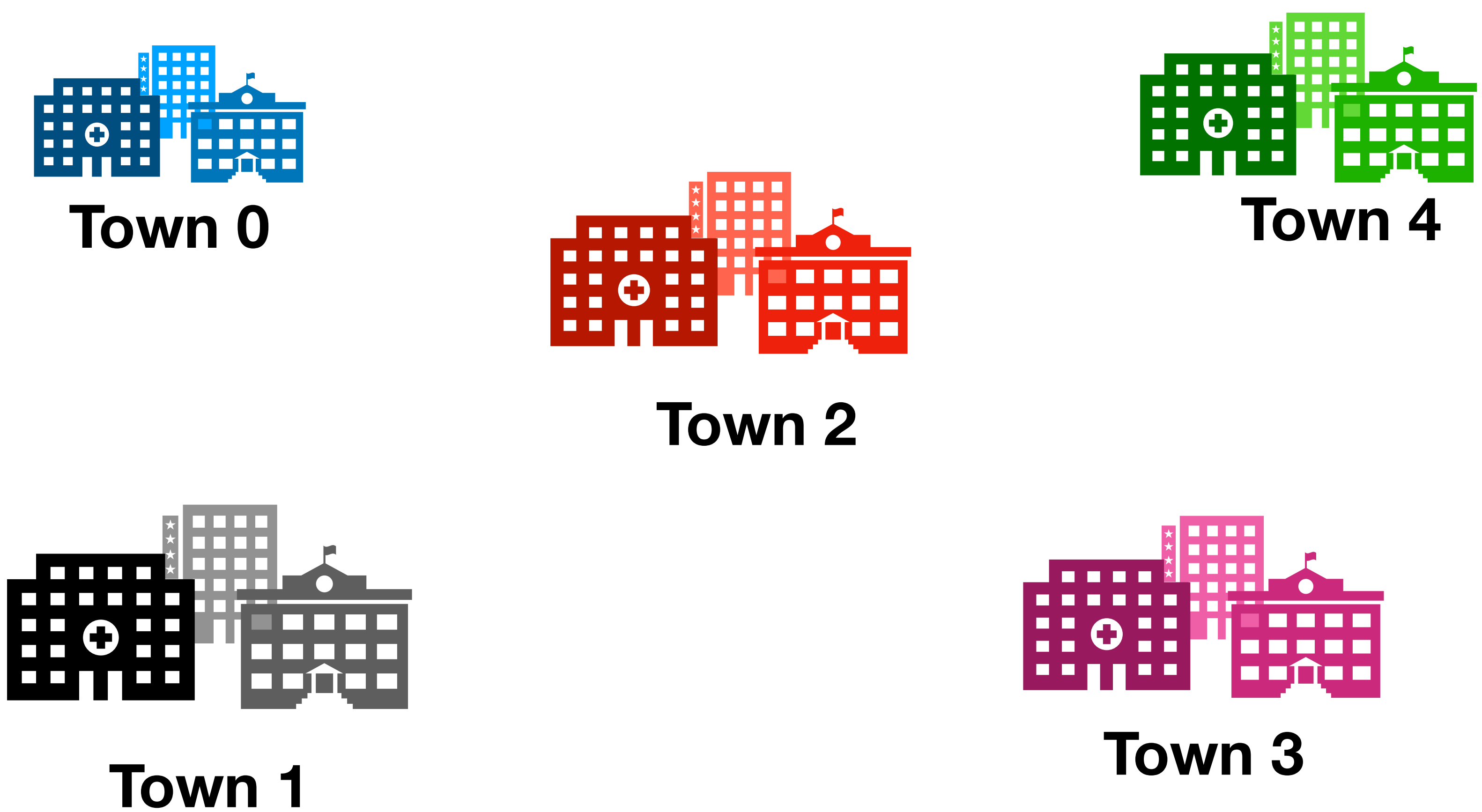
```
class QuickFindDS:
  int[] id

  QuickFindDS(N):
    id = new int[N]
    for i in 0...N:
      id[i] = i

  find(a):
    return id[a]

  union(a, b):
    let aId = id[a]
    let bId = id[b]
    for i in 0...N:
      if id[i] == a_id:
        id[i] = b_id
```

Runtime? $\Theta(1)$

Runtime? $\Theta(N)$

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).

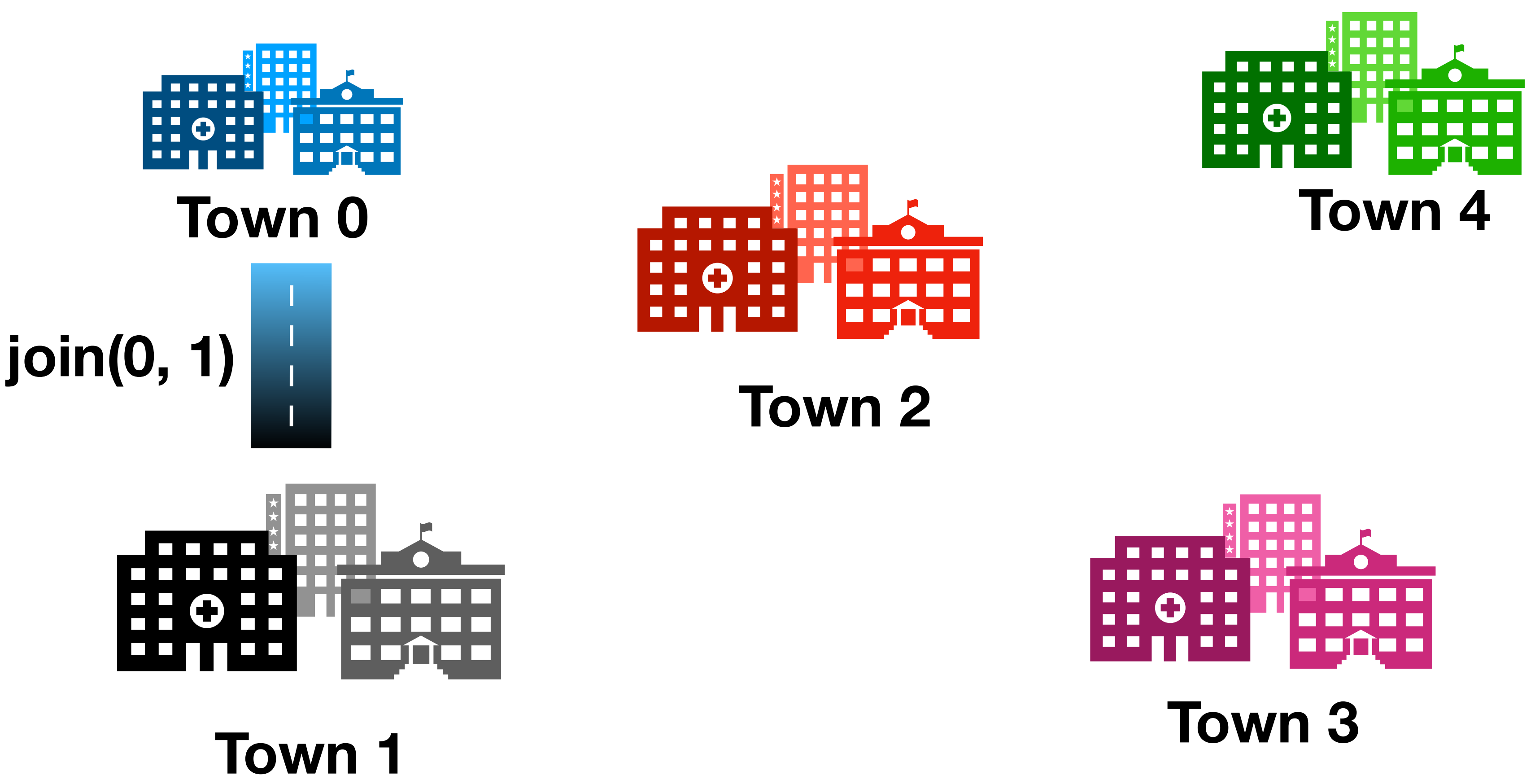


-1 represents no parent.

-1	-1	-1	-1	-1
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

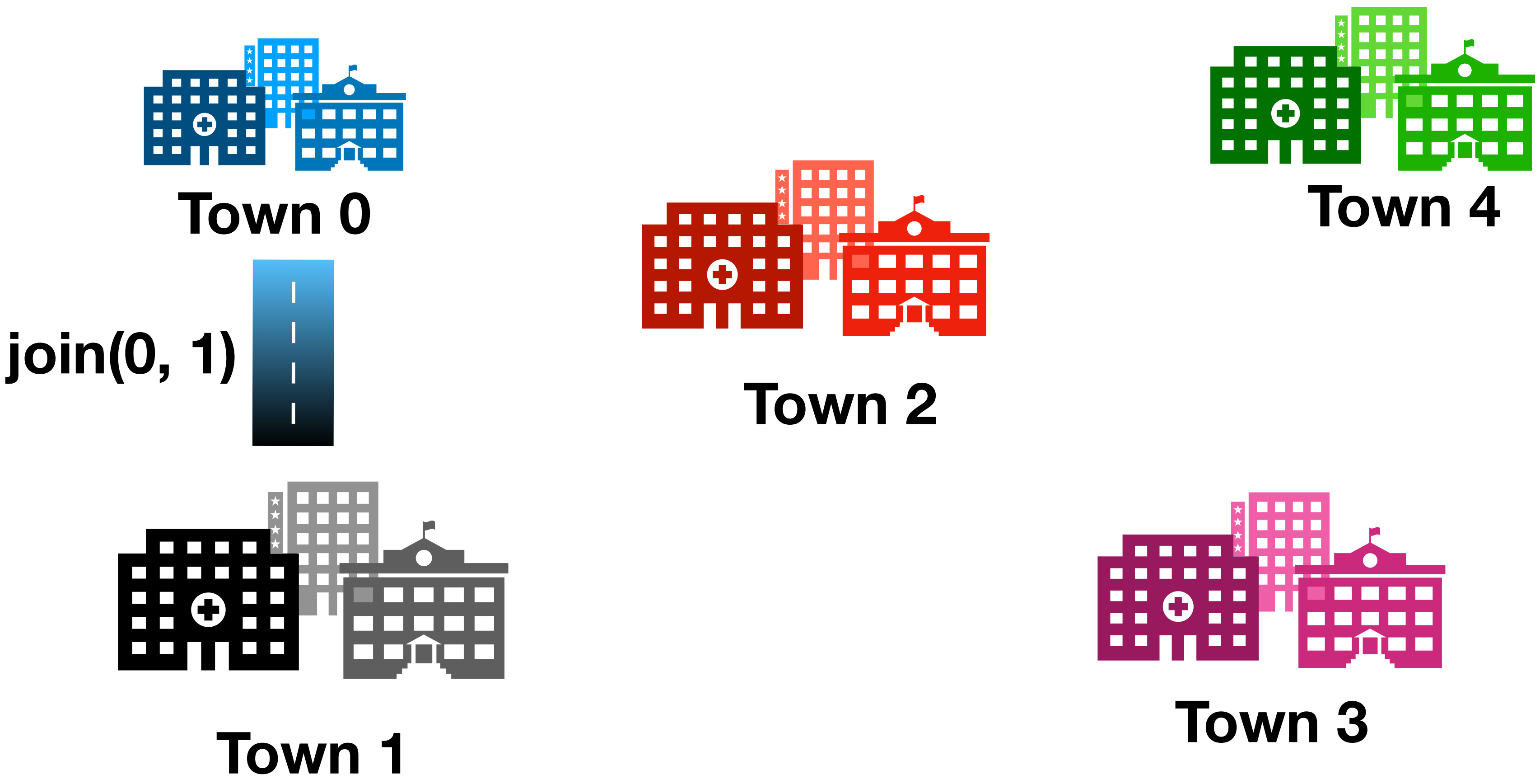
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	-1	-1	-1	-1
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

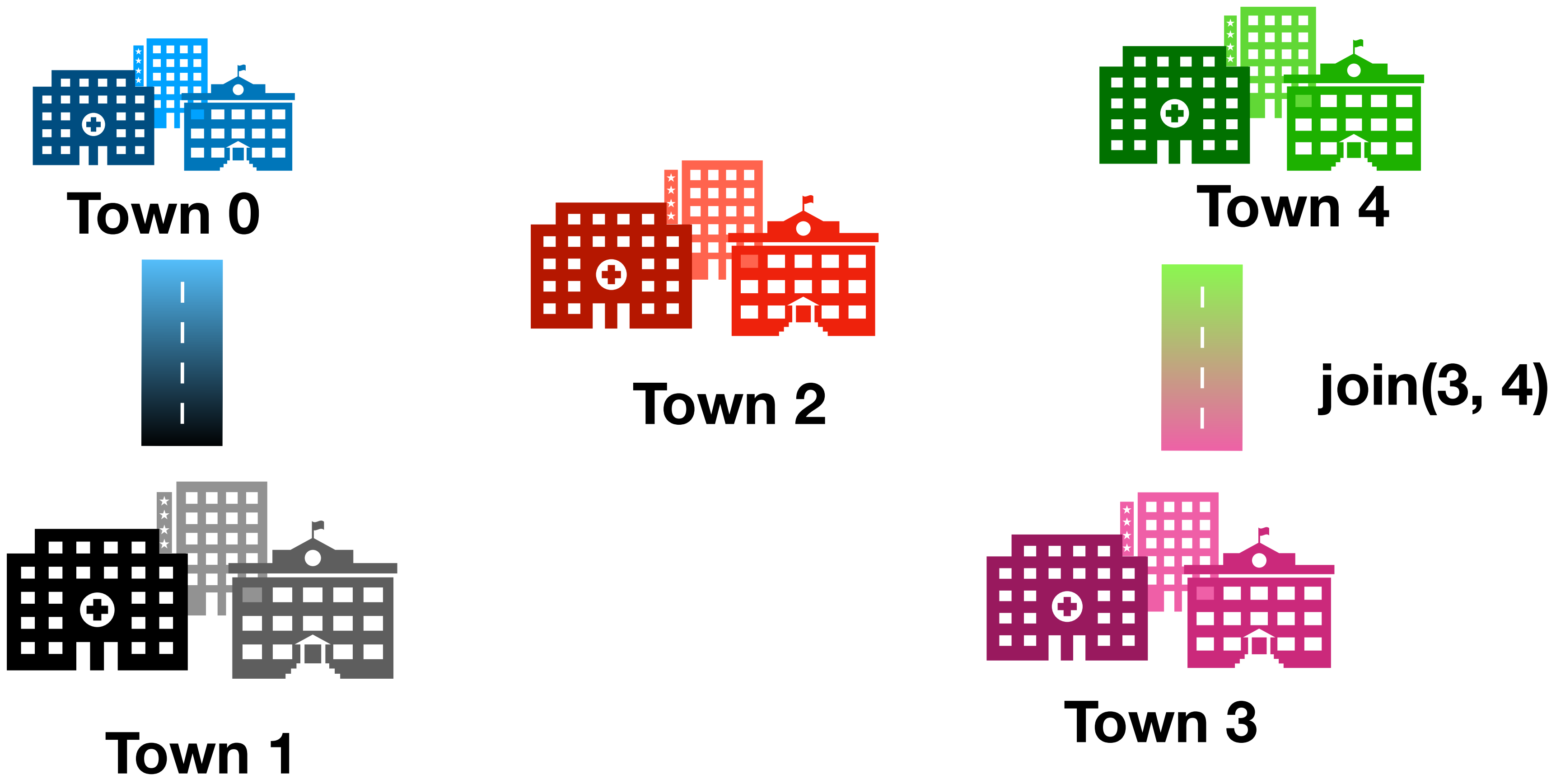
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



0	1	2	3	4
-1	**0**	**-1**	**-1**	**-1**

Idea 2: Prioritize unions ("Quick union")

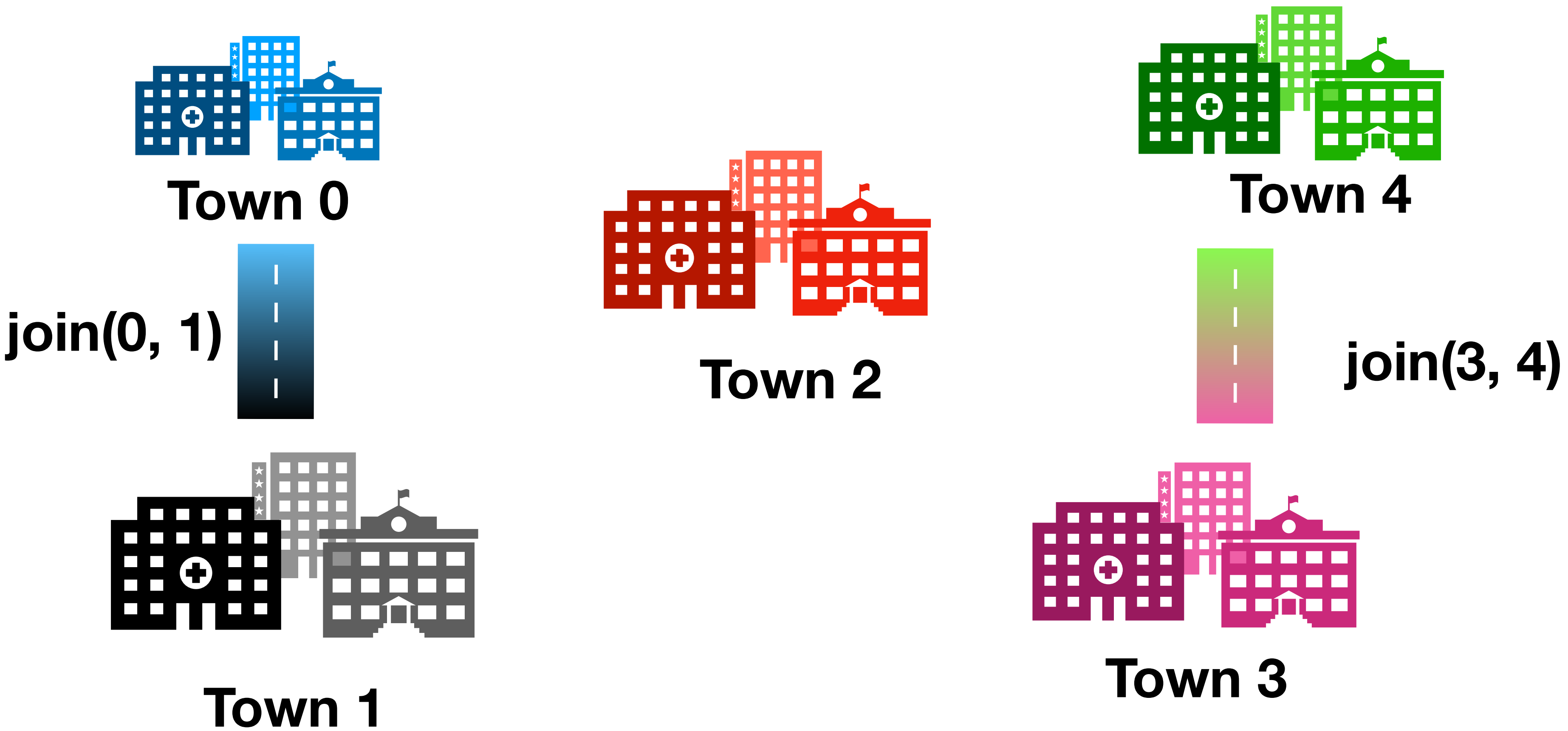
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



0	1	2	3	4
-1	0	-1	-1	-1

Idea 2: Prioritize unions ("Quick union")

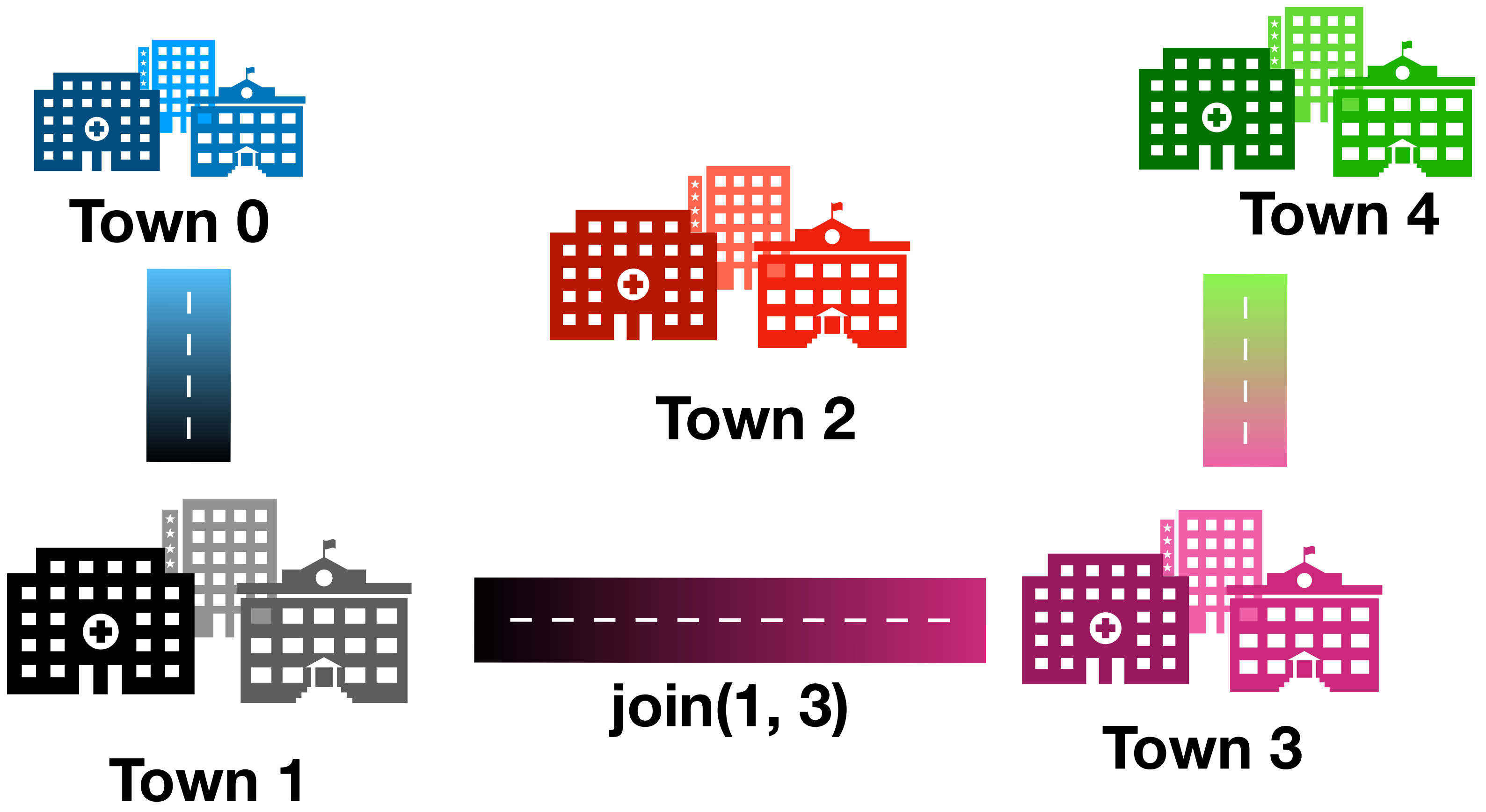
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	**0**	**-1**	**-1**	**3**
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

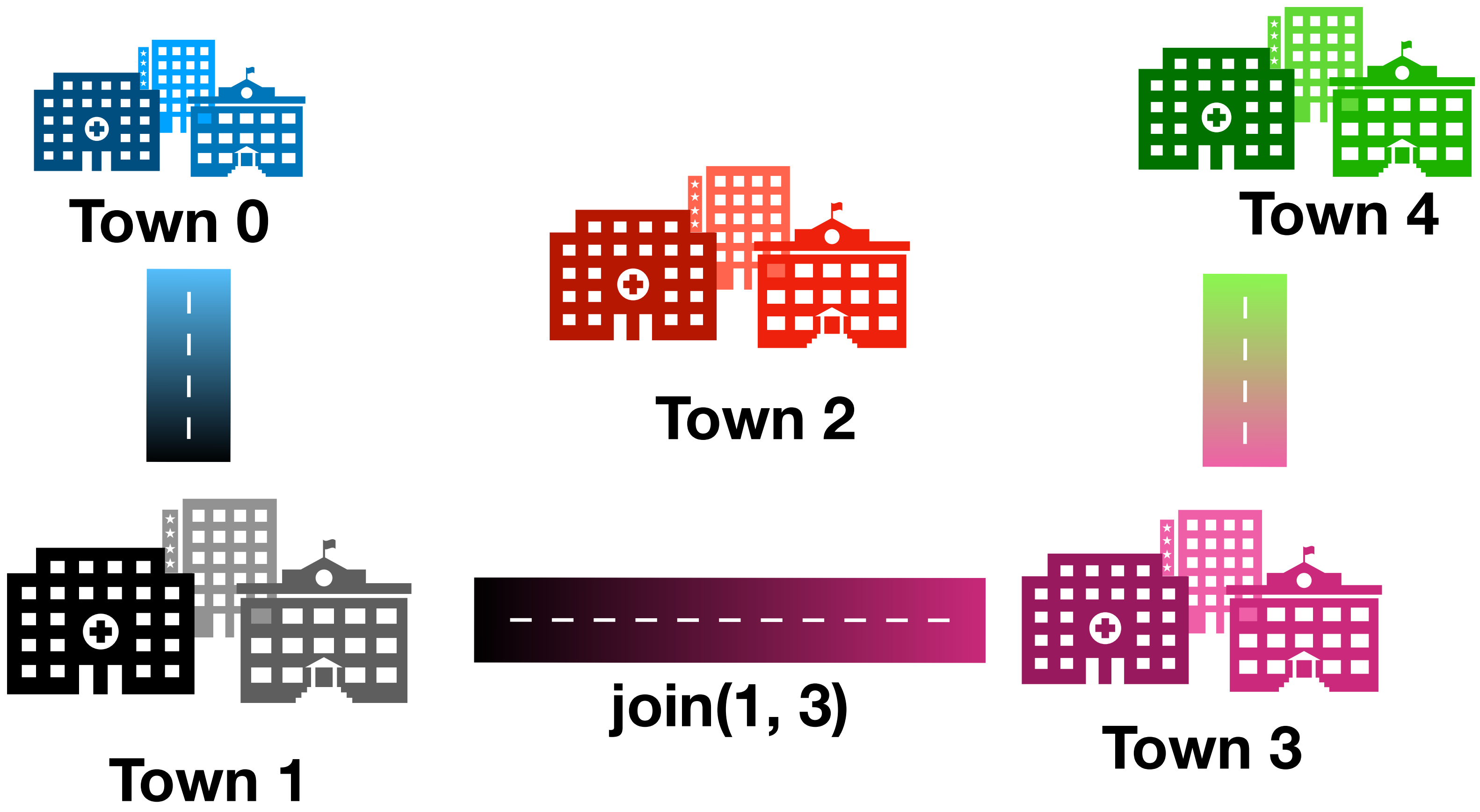
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	**0**	**-1**	**-1**	**3**
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

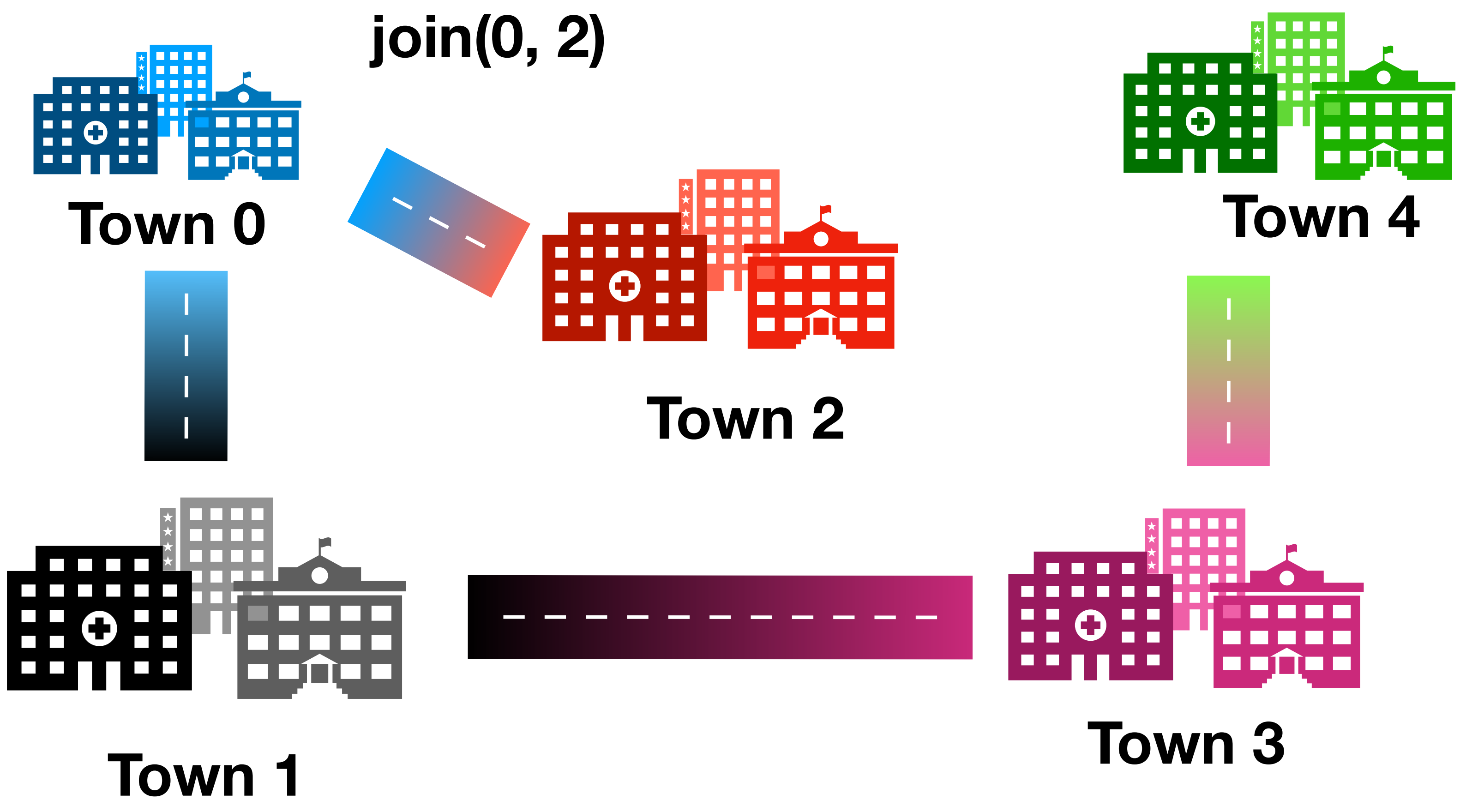
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	0	-1	0	3
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

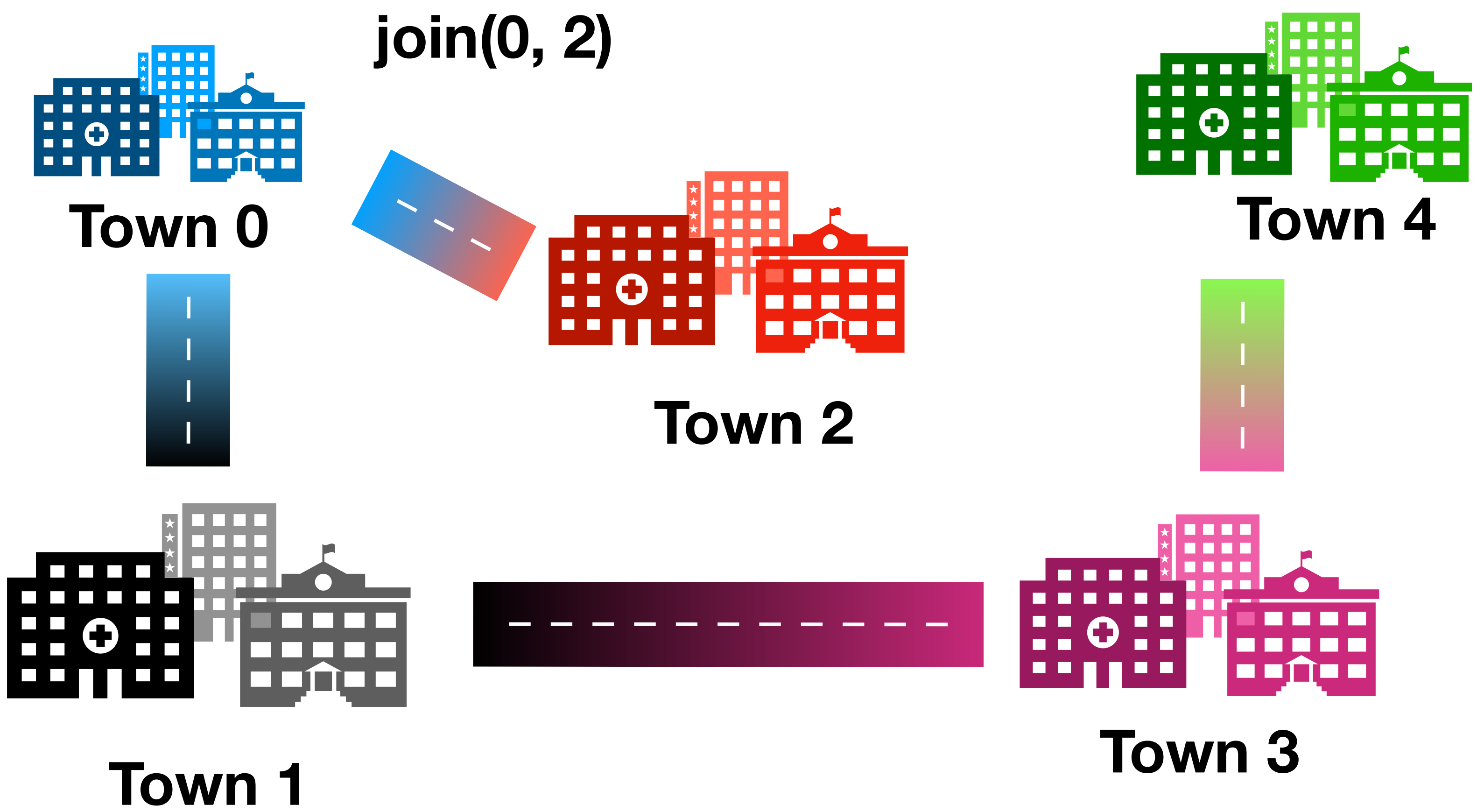
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	**0**	**-1**	**1**	**3**
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

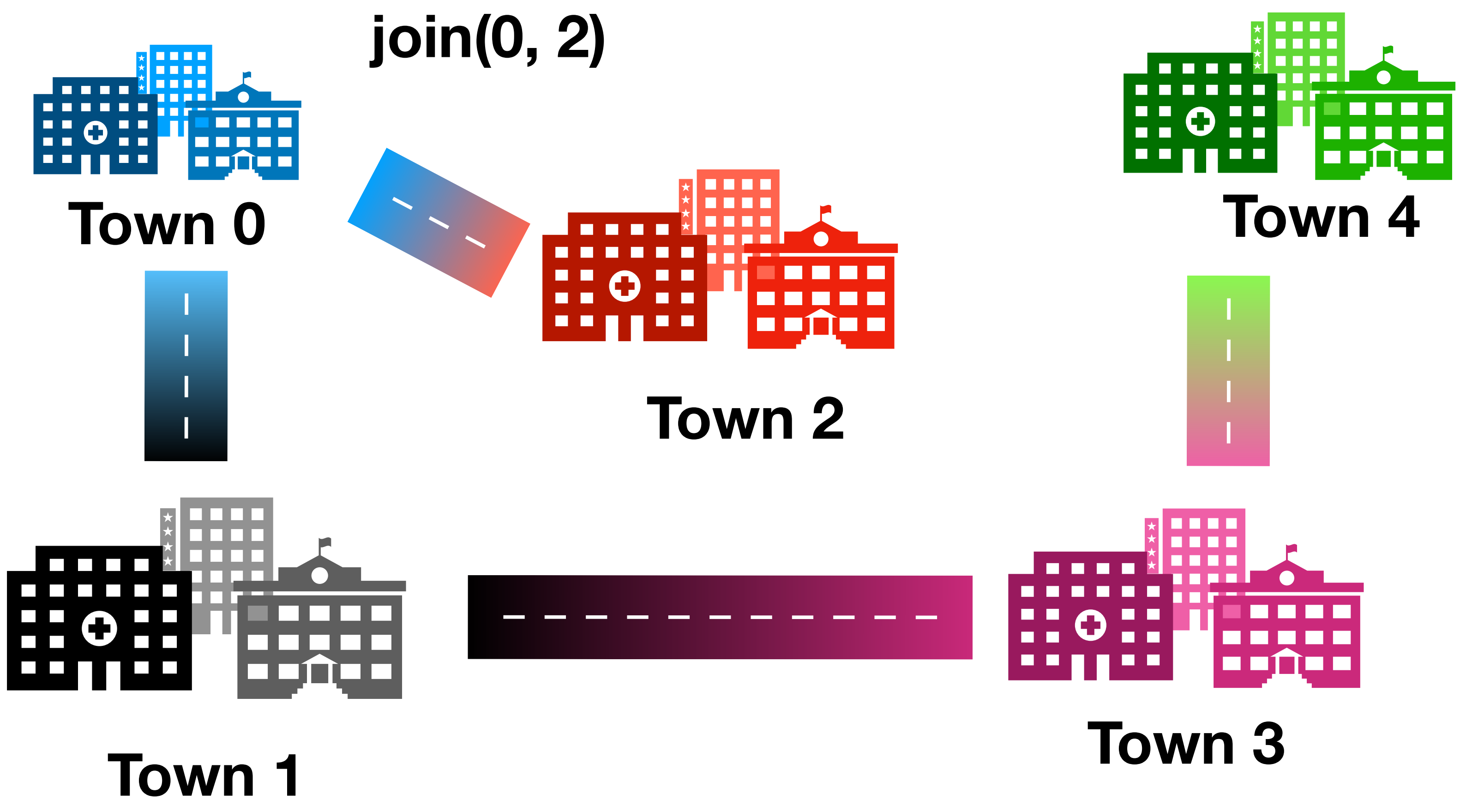
We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



-1	**0**	**0**	**1**	**3**
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).

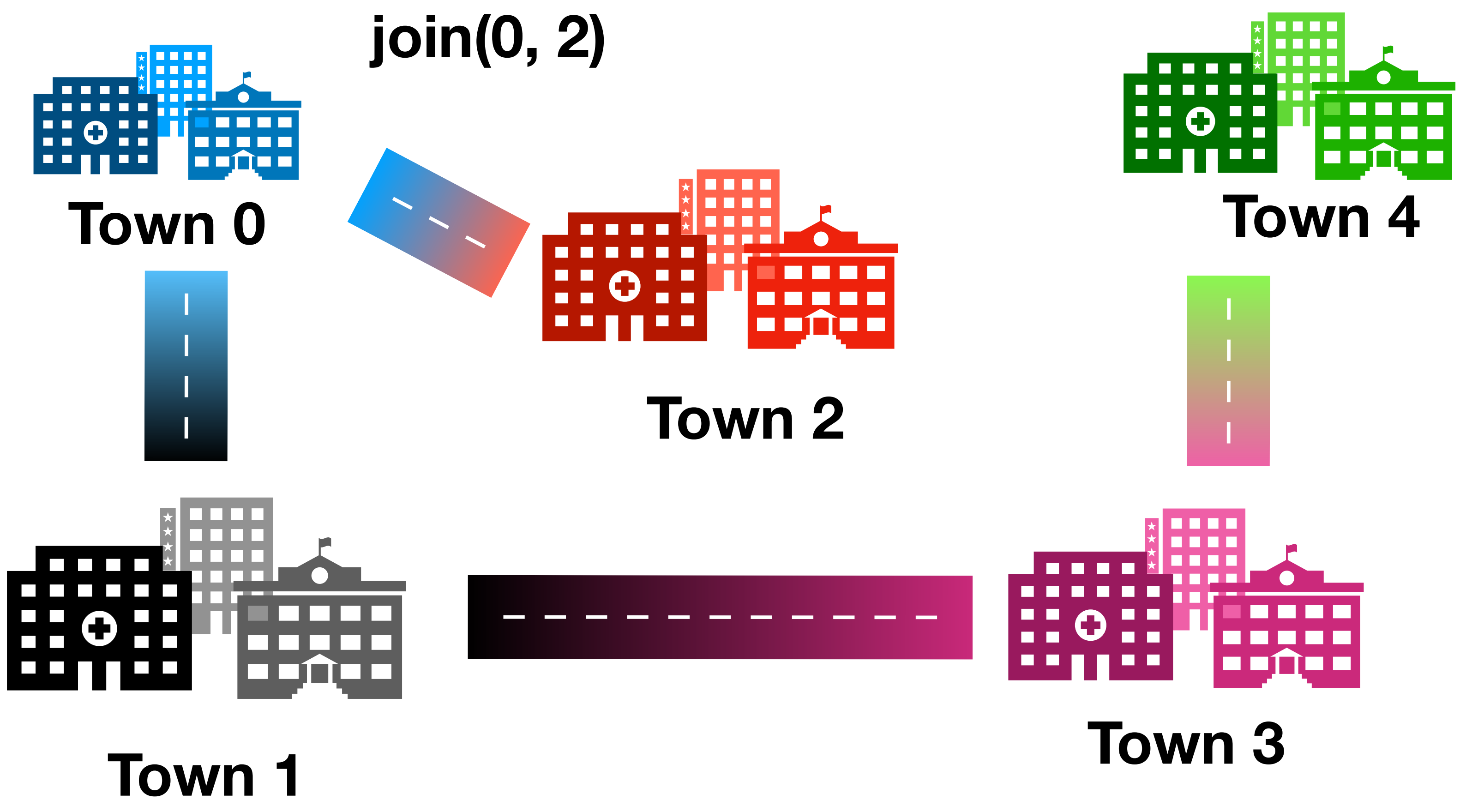


Is 2 and 4 connected?

-1	0	0	1	3
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



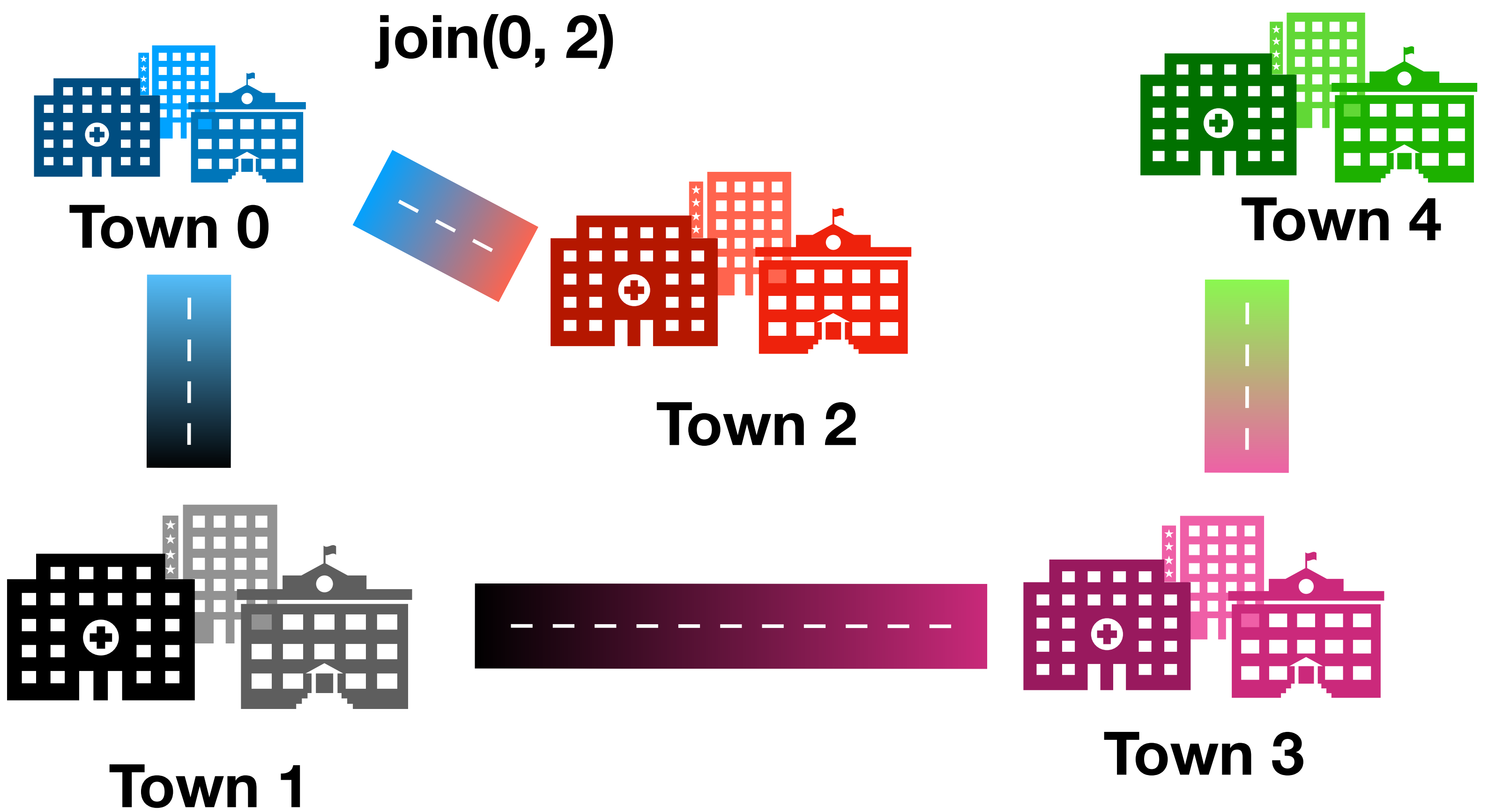
Is 2 and 4 connected?

See if find() returns the same thing.

-1	0	0	1	3
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



Is 2 and 4 connected?

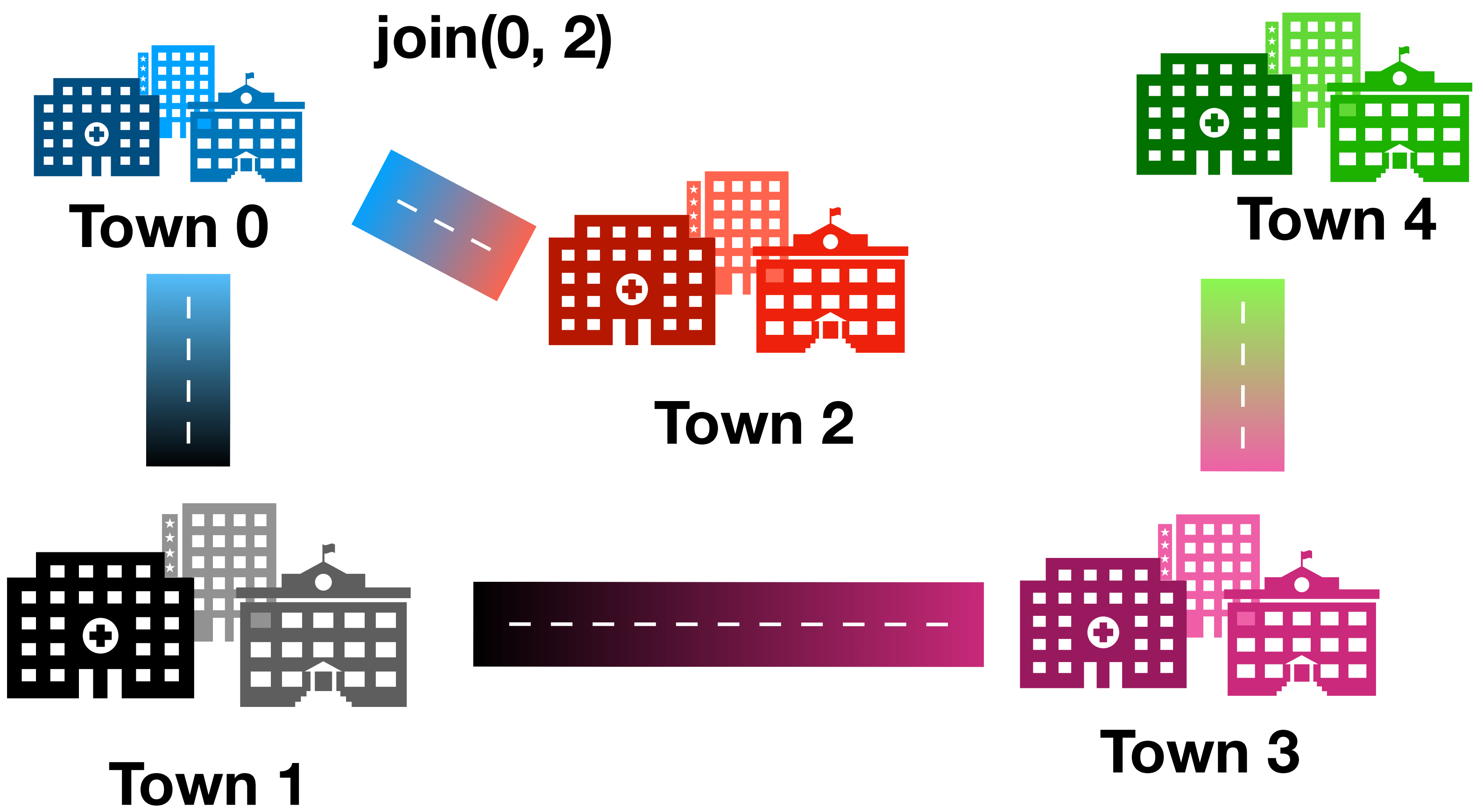
See if find() returns the same thing.

2 -> 0

0	1	2	3	4
-1	0	0	1	3

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



Is 2 and 4 connected?

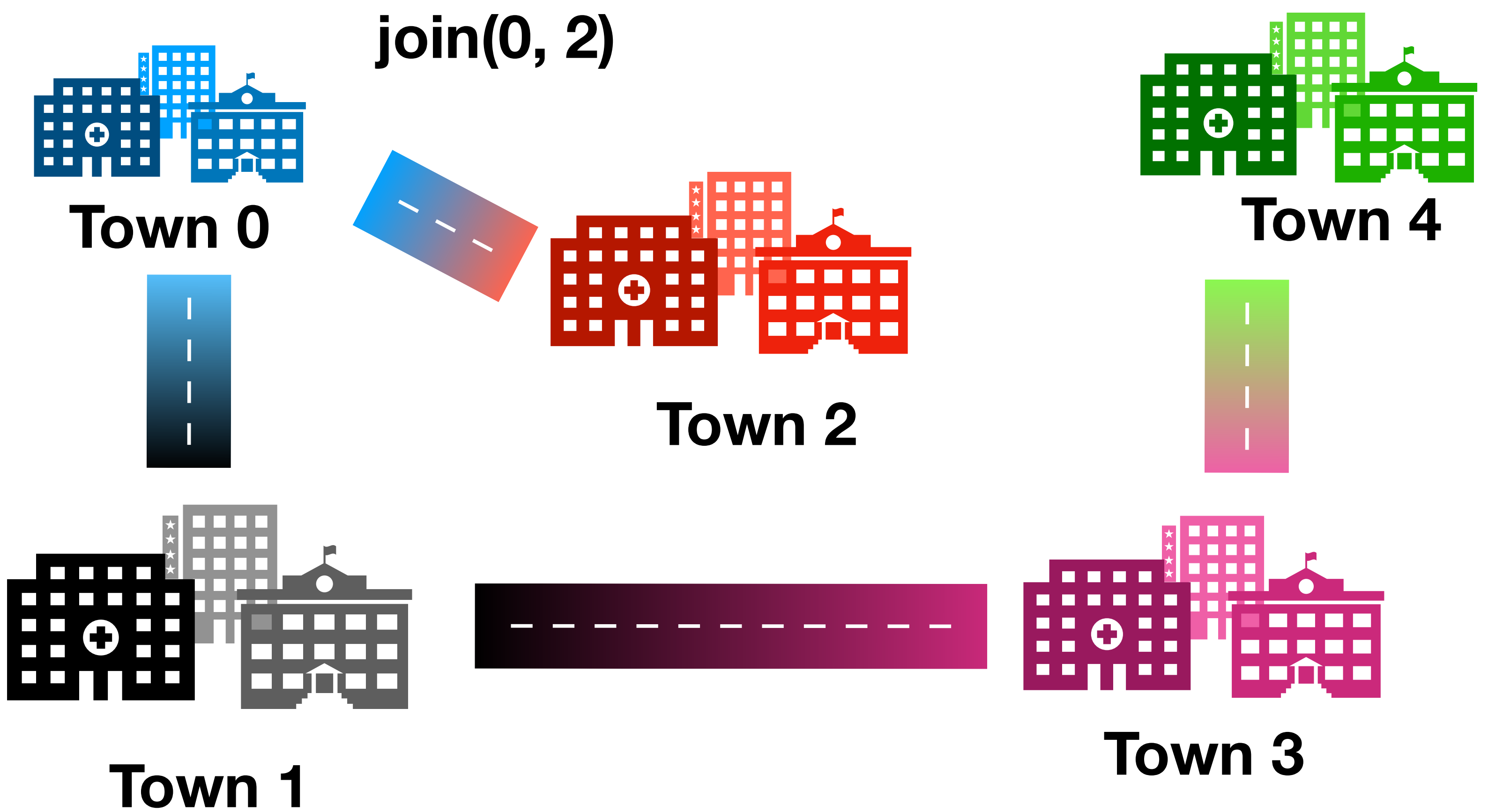
See if find() returns the same thing.

2 -> 0
4 -> 3 -> 1 -> 0

-1	**0**	**0**	**1**	**3**
0	1	2	3	4

Idea 2: Prioritize unions ("Quick union")

We will represent the sets using a tree structure.
When you join a and b, set the root of a to be the parent of the root of b.
The set # is the oldest ancestor (i.e. the root).



Is 2 and 4 connected?

See if find() returns the same thing.

2 -> 0
4 -> 3 -> 1 -> 0

Yes.

-1	0	0	1	3
0	1	2	3	4

QuickUnion Pseudocode

```
class QuickUnionDS:
  int[] parent

  QuickUnionDS(N):
    parent = new int[N]
    for i in 0...N:
      id[i] = -1

  find(a):
    let curr = a
    while parent[curr] != -1:
      curr = parent[curr]
    return curr

  union(a, b):
    parent[find(b)] = find(a)
```

Problem: we're forming a tree structure and it's possible the tree is spindly!

Runtime?

Runtime?

QuickUnion Pseudocode

```
class QuickUnionDS:
  int[] parent

  QuickUnionDS(N):
    parent = new int[N]
    for i in 0...N:
      id[i] = -1

  find(a):
    let curr = a
    while parent[curr] != -1:
      curr = parent[curr]
    return curr

  union(a, b):
    parent[find(b)] = find(a)
```

Problem: we're forming a tree structure and it's possible the tree is spindly!

Runtime? $O(N)$

Runtime?

QuickUnion Pseudocode

```
class QuickUnionDS:
  int[] parent

  QuickUnionDS(N):
    parent = new int[N]
    for i in 0...N:
      id[i] = -1

  find(a):
    let curr = a
    while parent[curr] != -1:
      curr = parent[curr]
    return curr

  union(a, b):
    parent[find(b)] = find(a)
```

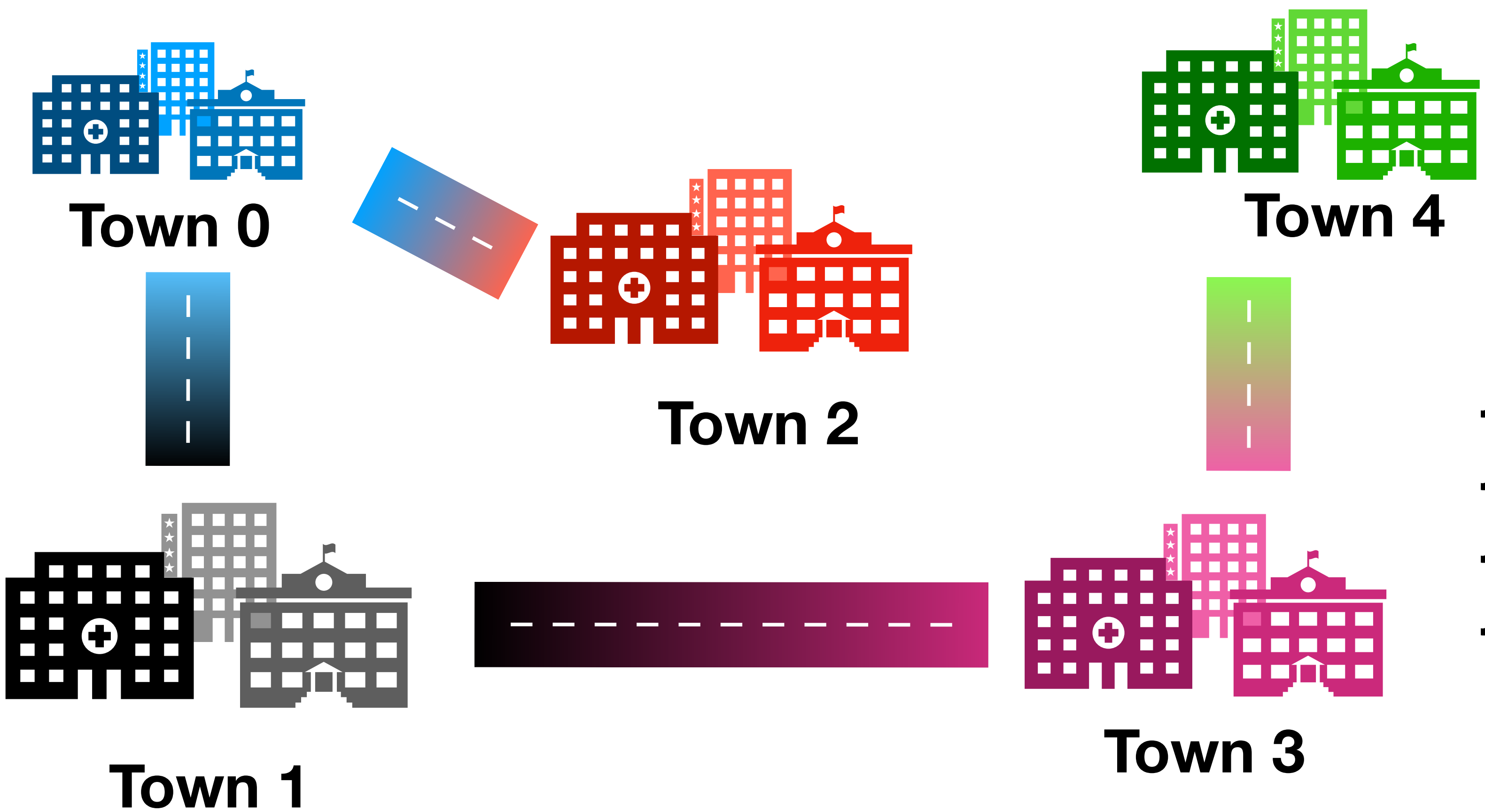
Problem: we're forming a tree structure and it's possible the tree is spindly!

Runtime? $O(N)$

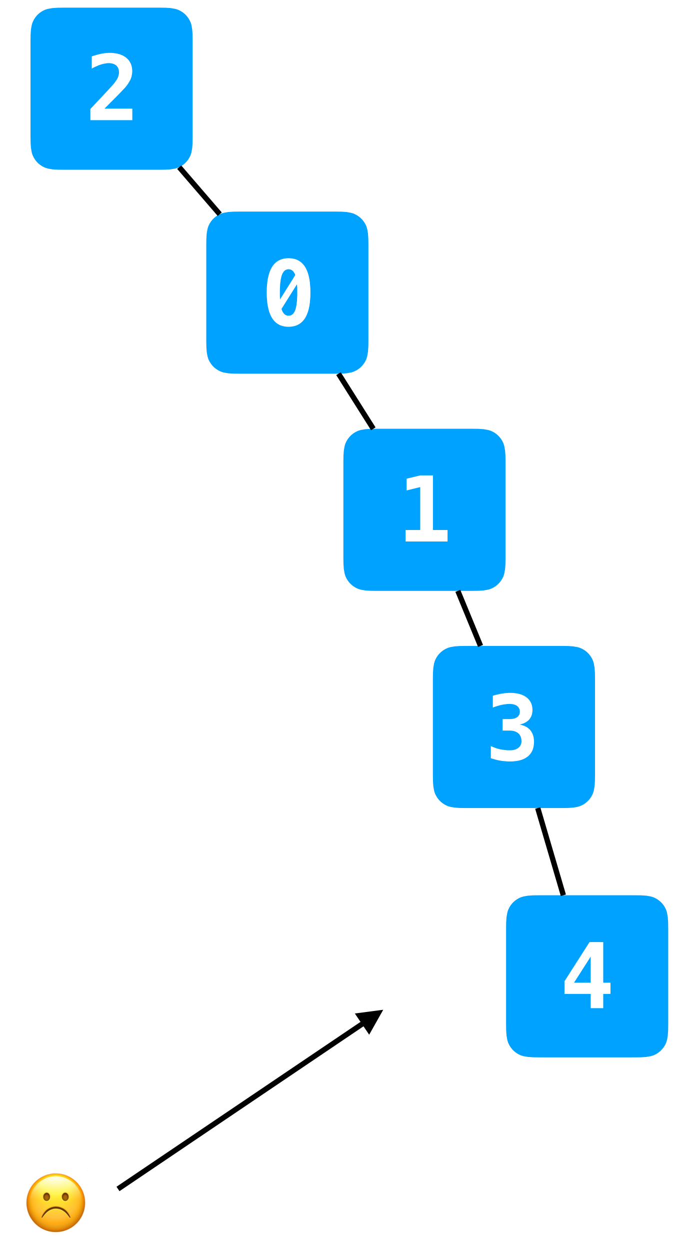
Runtime? $O(N)$

Problem with QuickUnion

- Easy to get spindly tree depending on insertion order.



join(3, 4)
join(1, 3)
join(0, 1)
join(2, 0)



Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

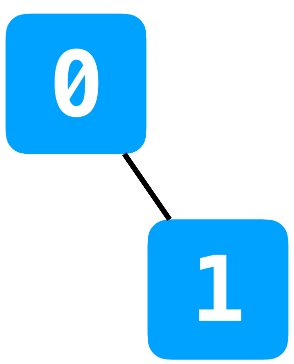
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

join(0, 1)?

Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

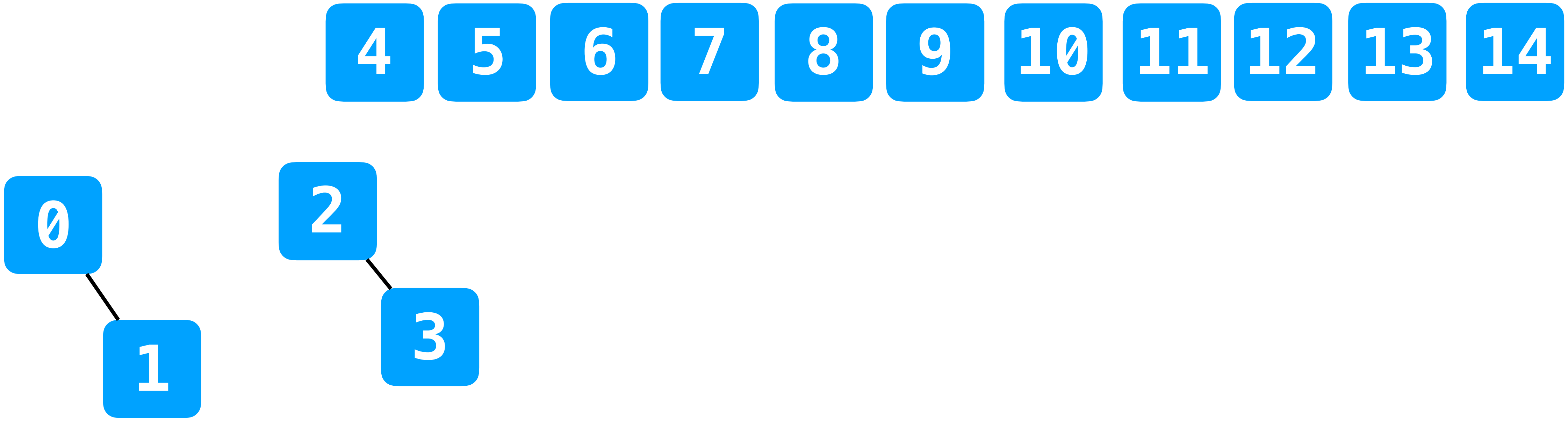


join(2, 3)?

Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

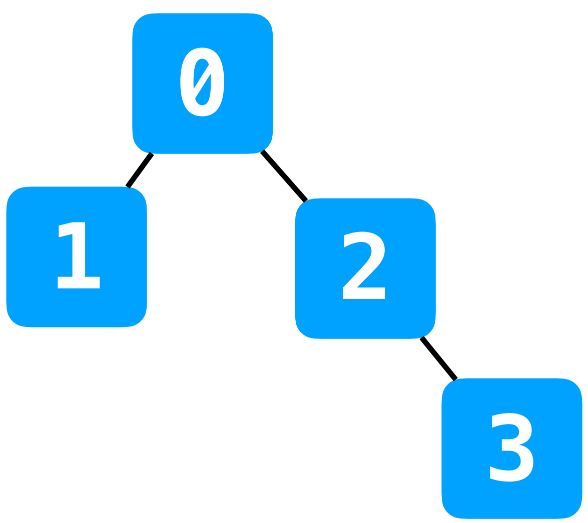


join(1, 3)?

Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

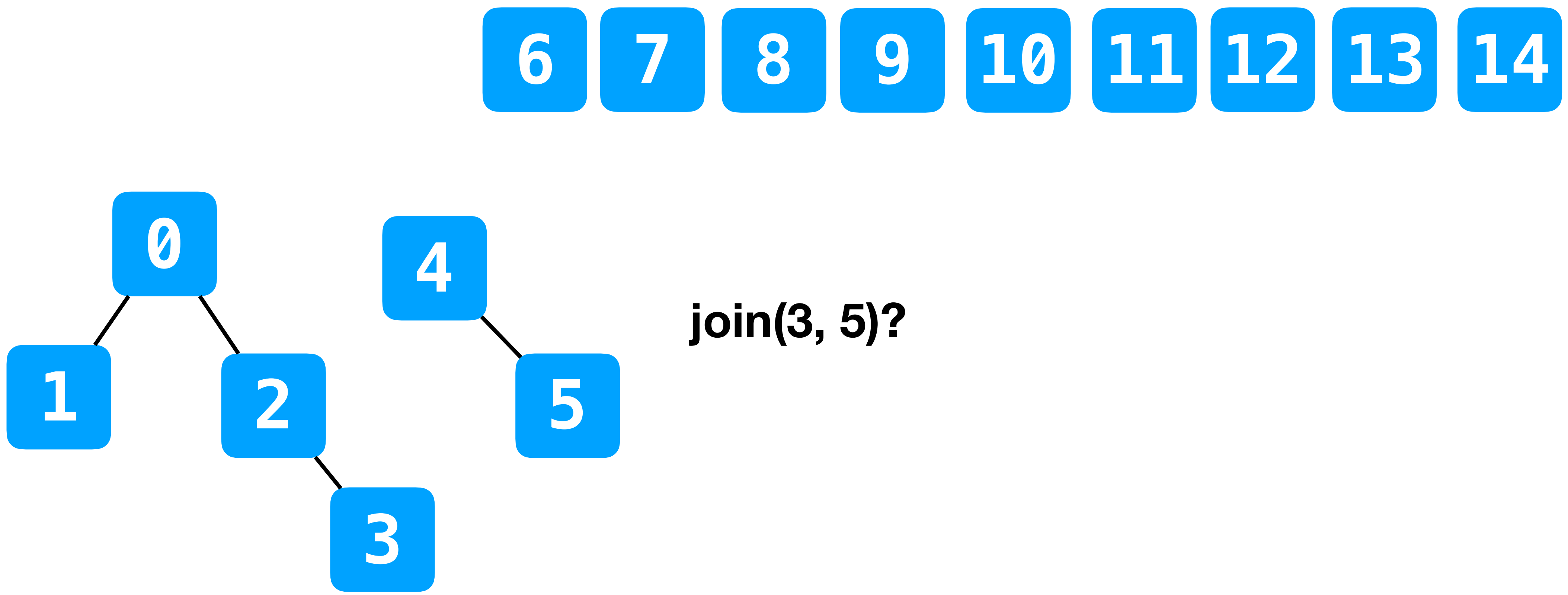


join(4, 5)?

Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

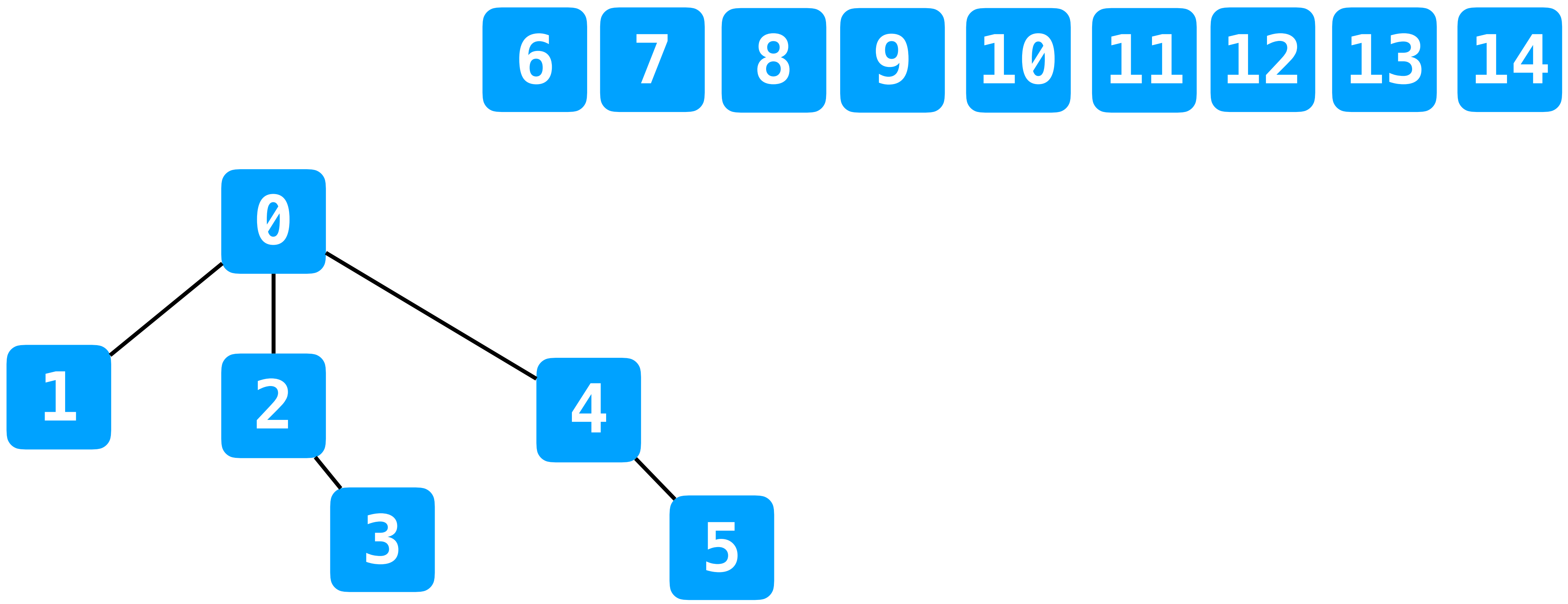


join(3, 5)?

Weighted QuickUnion

Let's add a new rule:

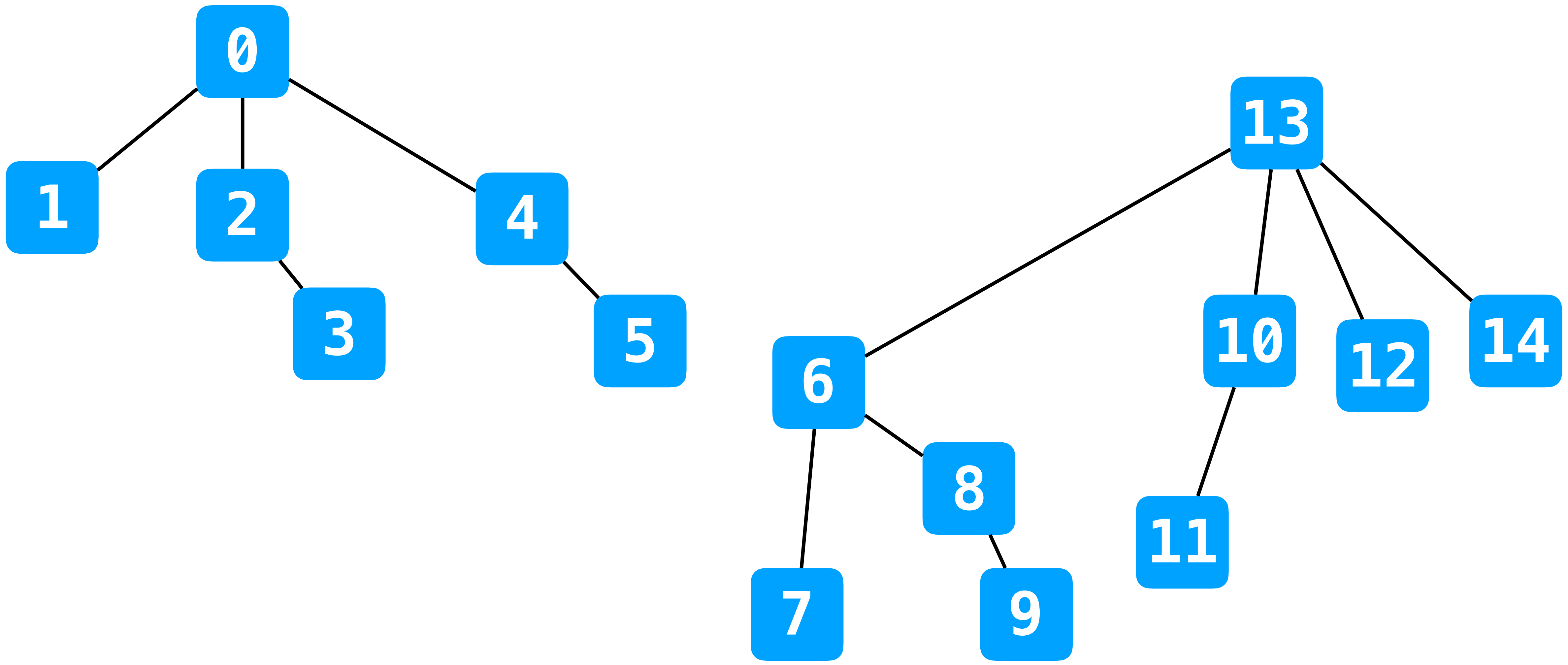
1. **Always join the *smaller tree* to the *larger tree***



Weighted QuickUnion

Let's add a new rule:

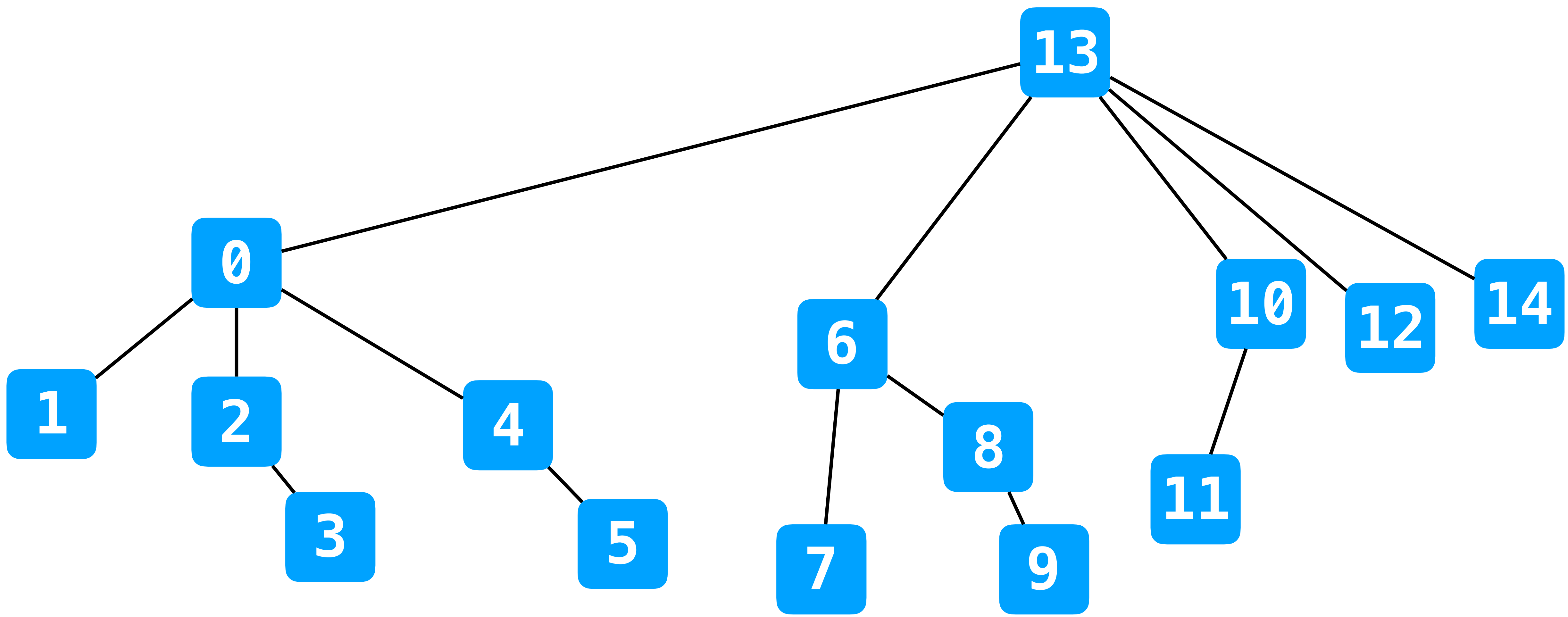
1. **Always join the *smaller tree* to the *larger tree***



Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

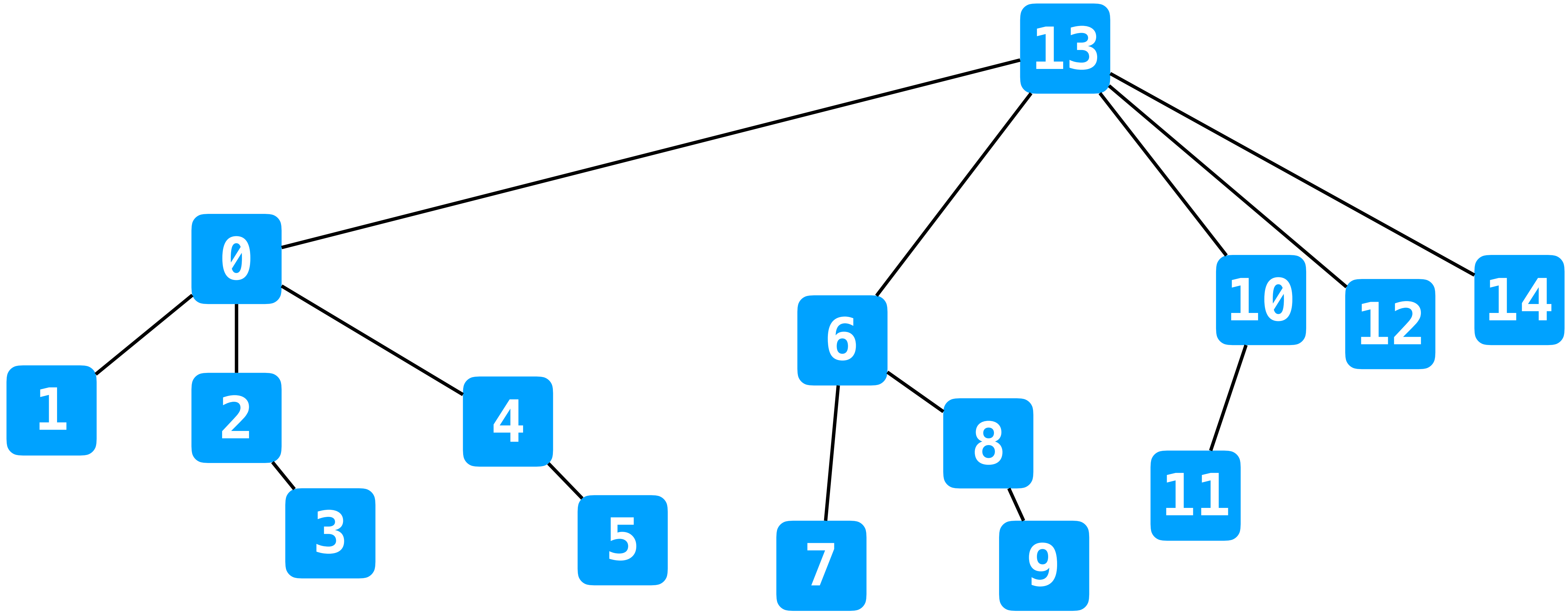


Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

What constitutes "larger"?
Height seems most intuitive but is more annoying to calculate.
Use # elements instead -- same asymptotic performance.



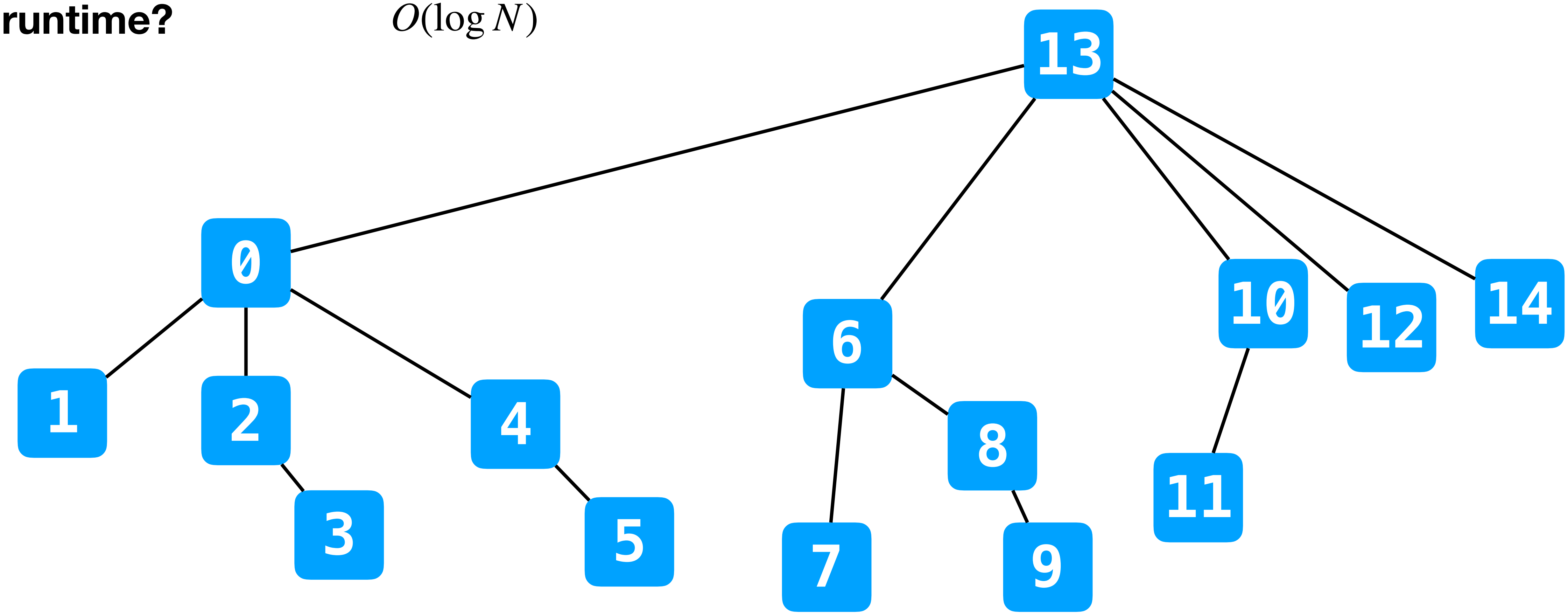
Weighted QuickUnion

Let's add a new rule:

1. **Always join the *smaller tree* to the *larger tree***

connect runtime? $O(\log N)$

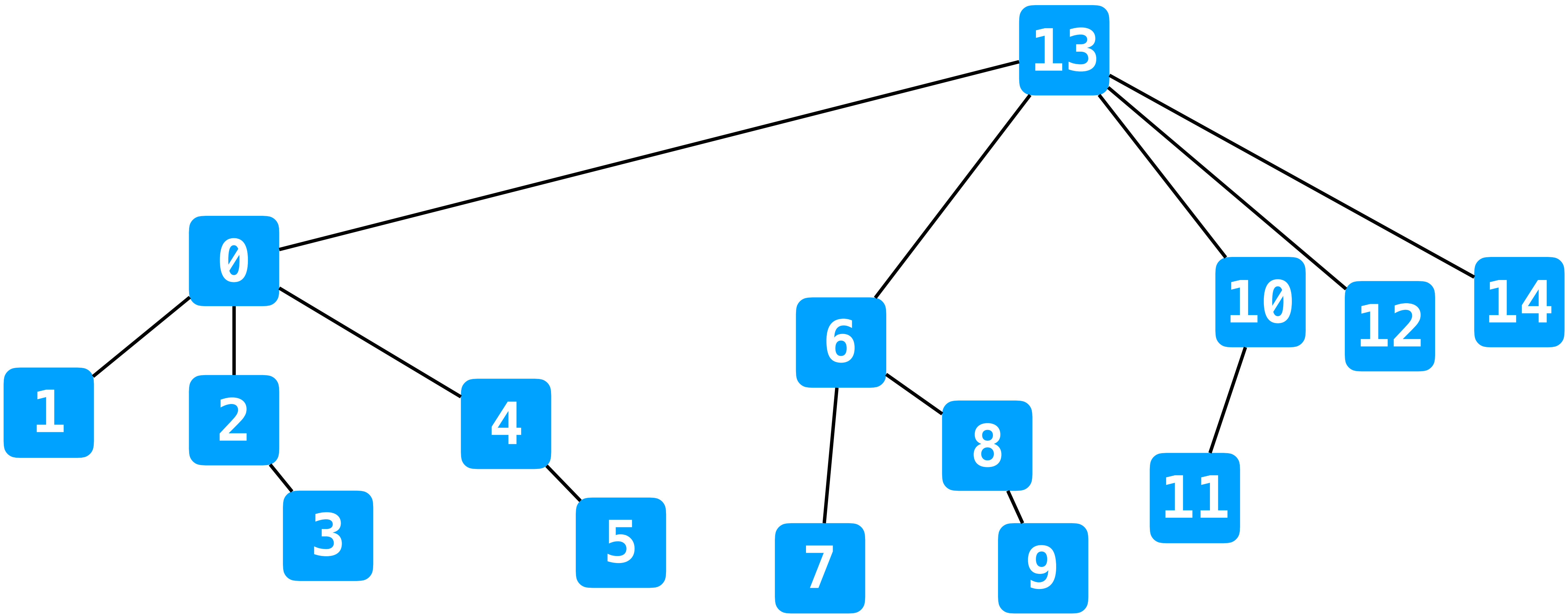
find runtime? $O(\log N)$



Path Compression

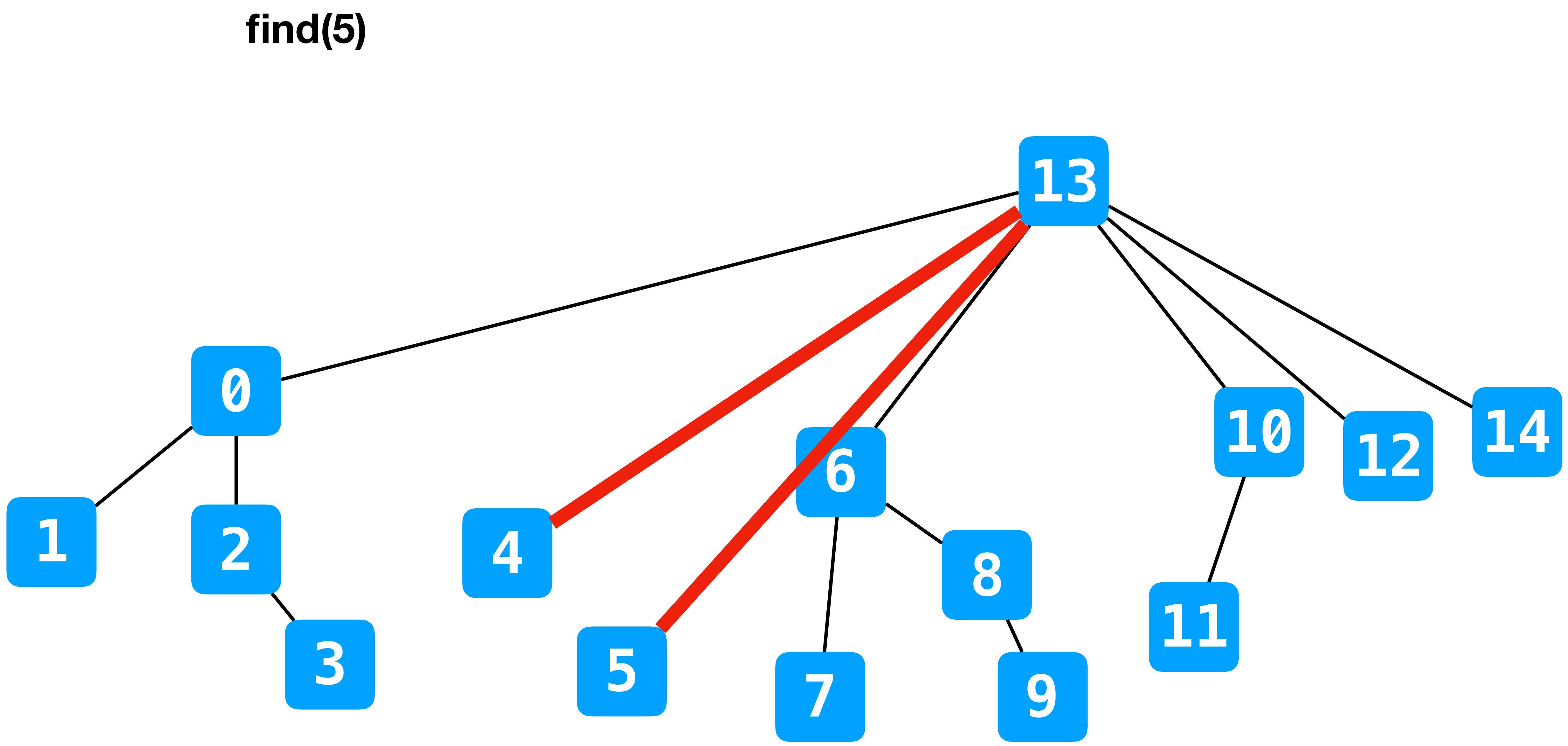
Whenever you run find, tie nodes along the way back to the root.

find(5)



Path Compression

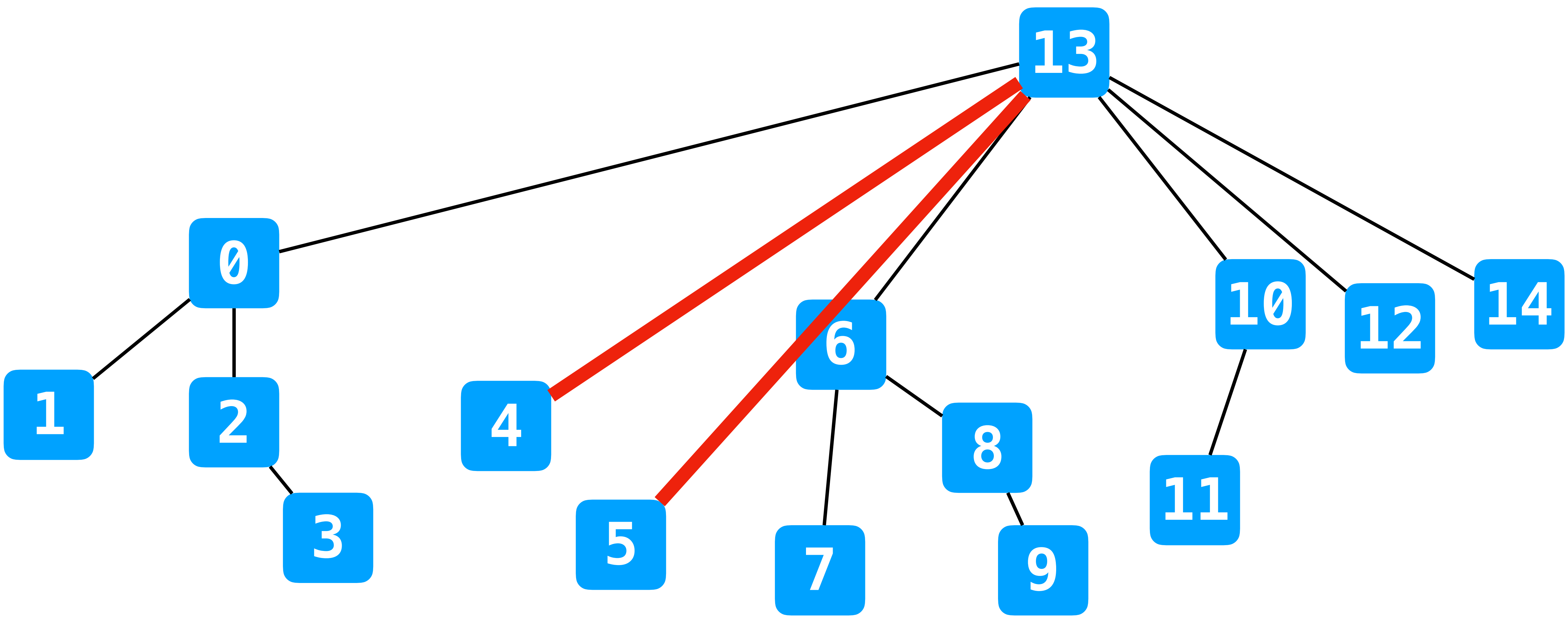
Whenever you run find, tie nodes along the way back to the root.



Path Compression

Whenever you run find, tie nodes along the way back to the root.

find(9)

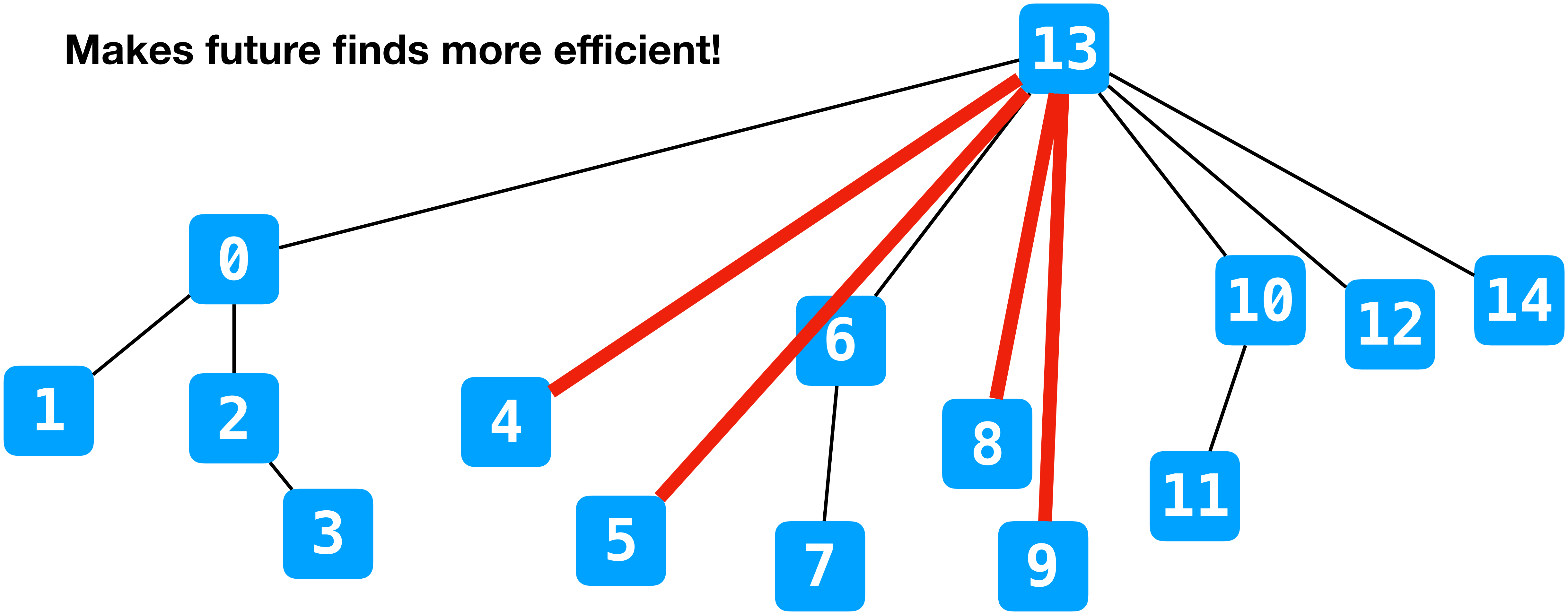


Path Compression

Whenever you run find, tie nodes along the way back to the root.

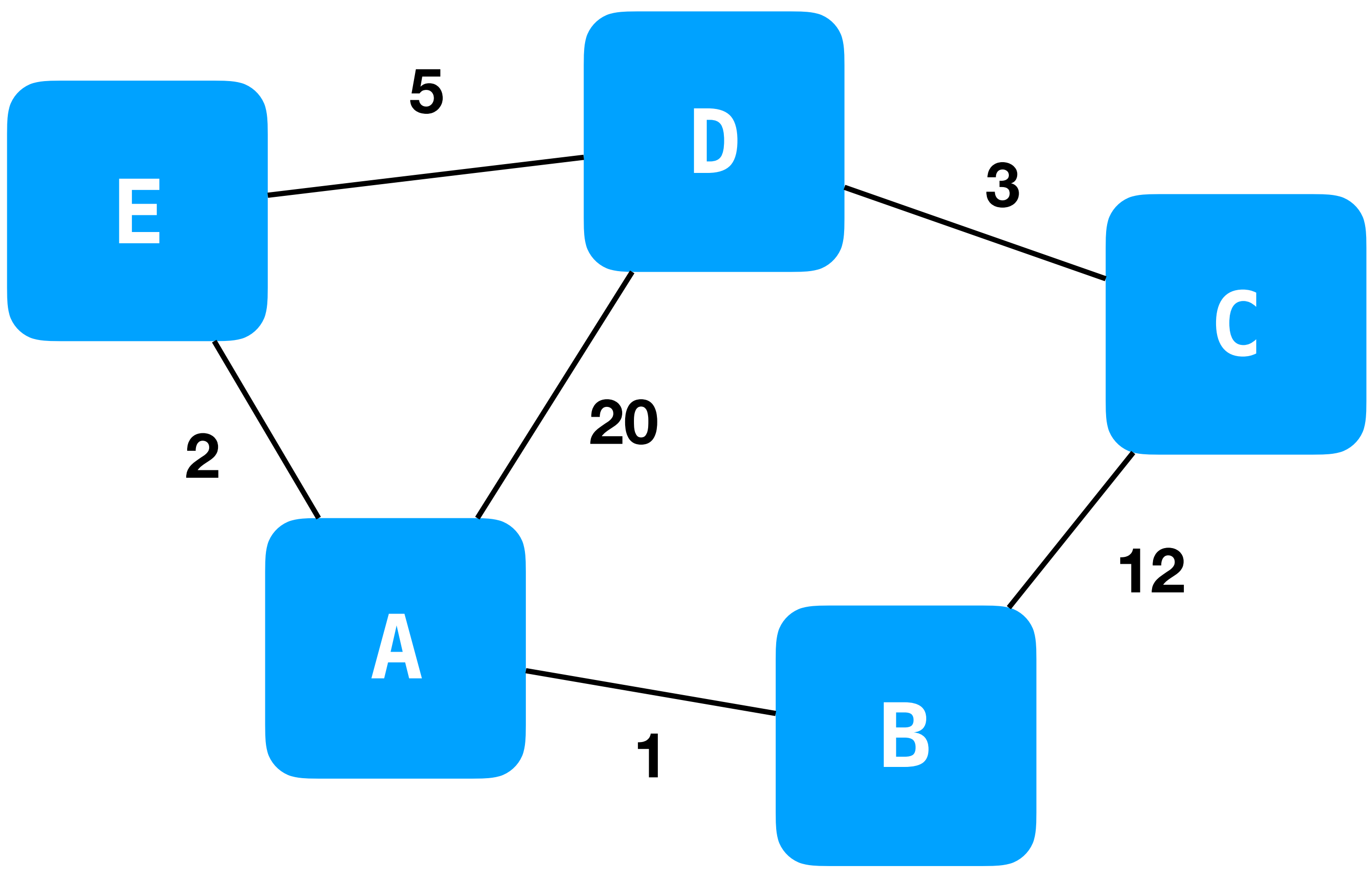
find(9)

Makes future finds more efficient!



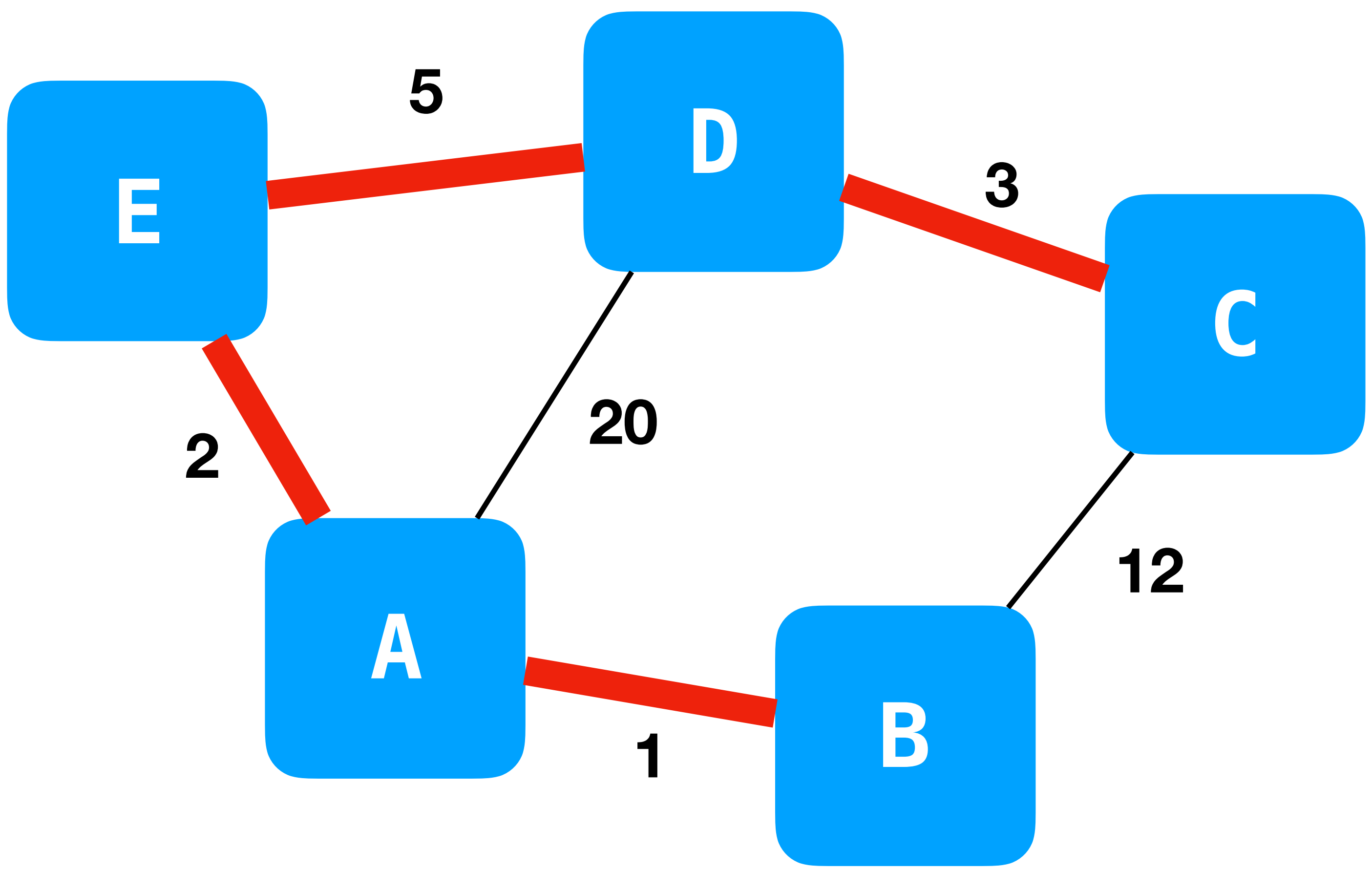
Minimum Spanning Tree

- The MST of a graph is a tree consisting of all of the vertices of the graph but whose total edge weights are minimized.



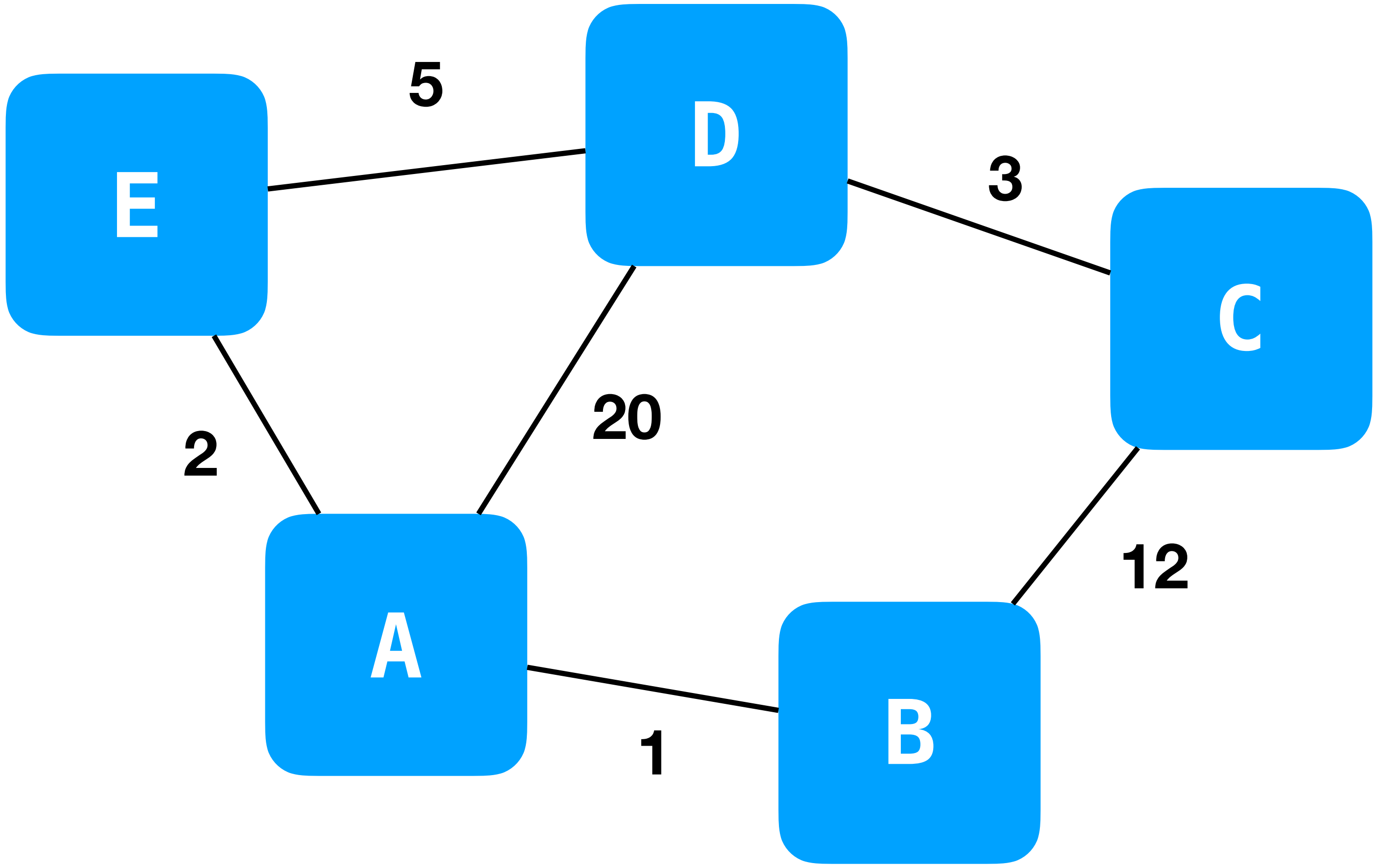
Minimum Spanning Tree

- The MST of a graph is a tree consisting of all of the vertices of the graph but whose total edge weights are minimized.



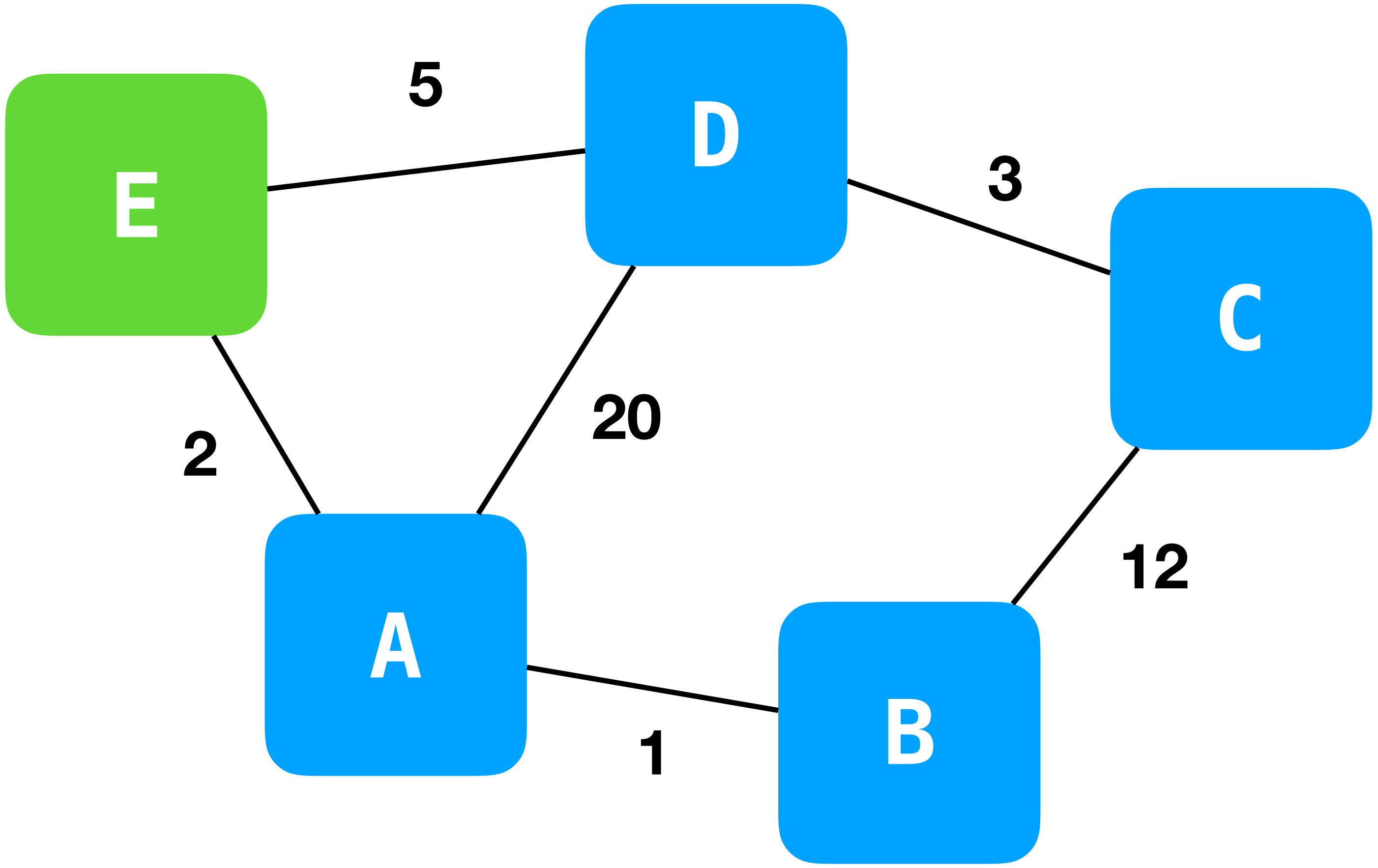
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



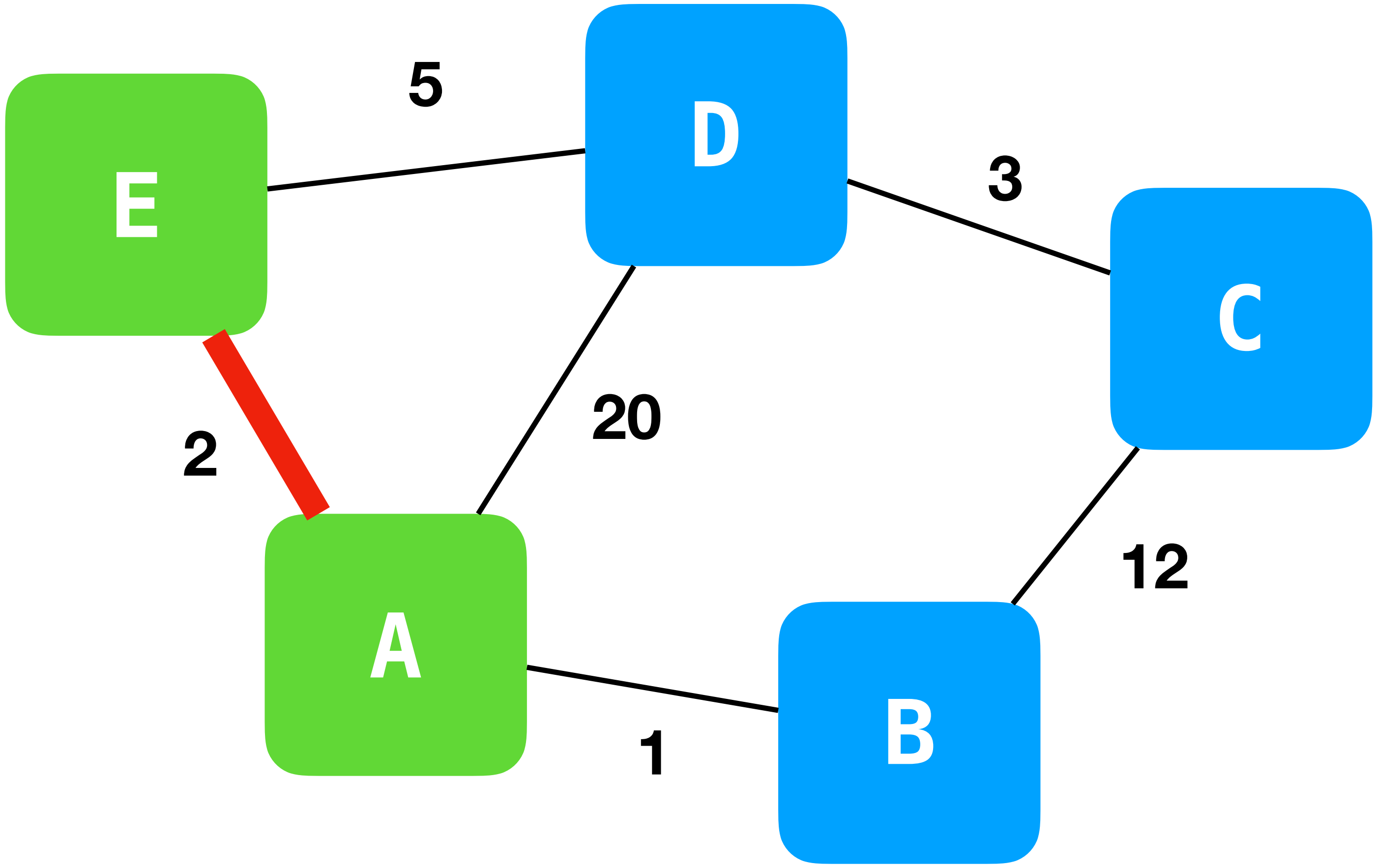
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



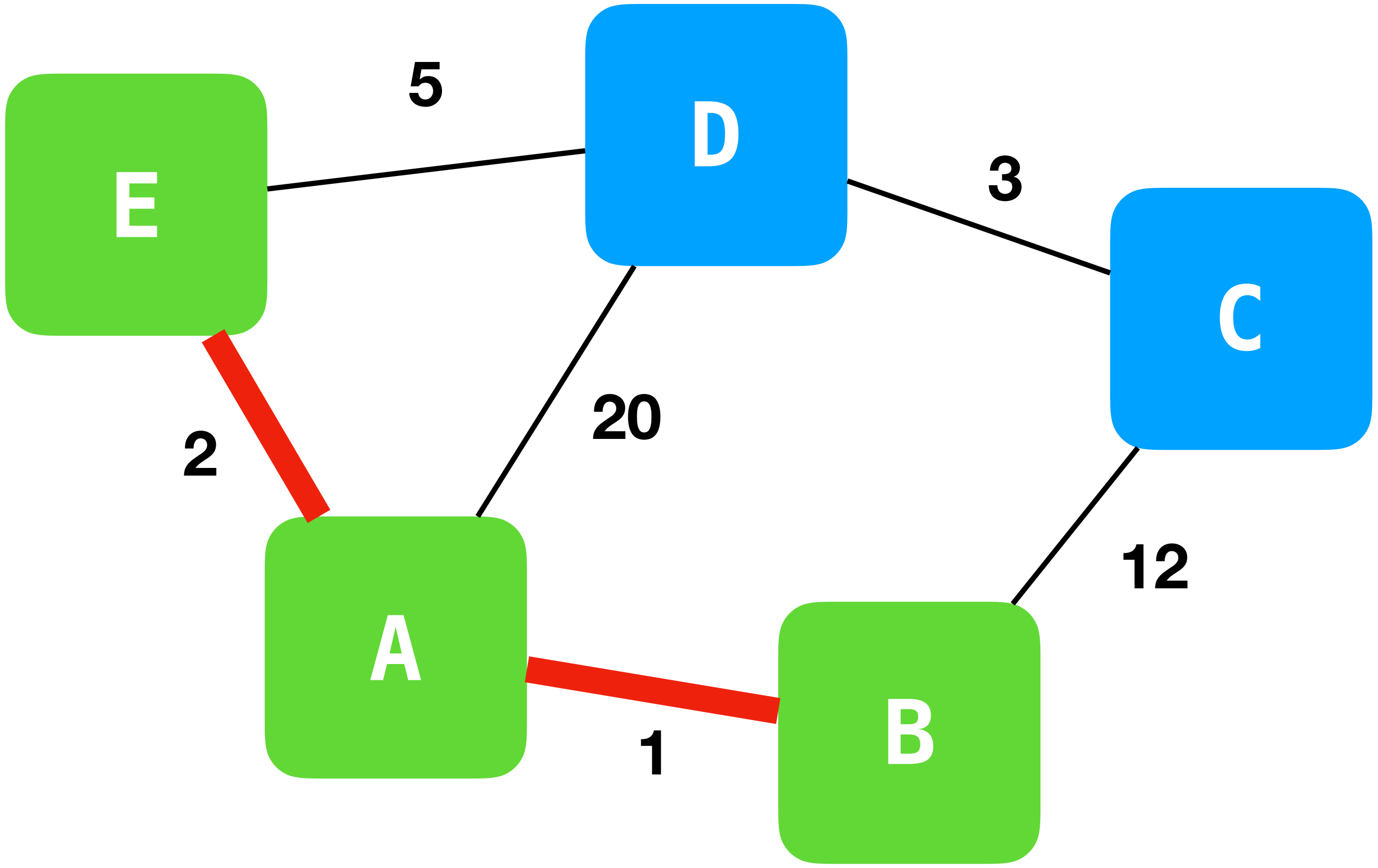
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



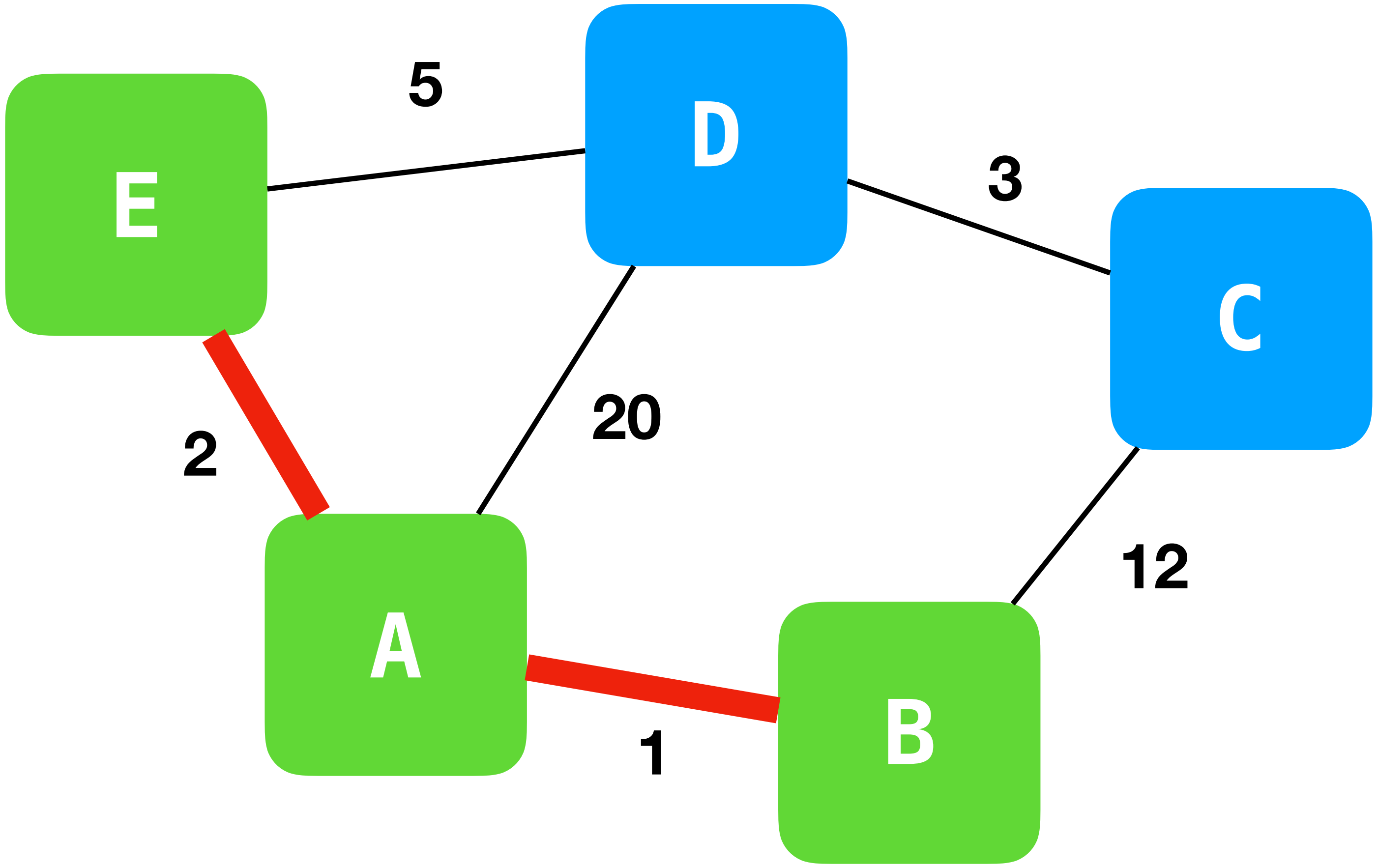
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



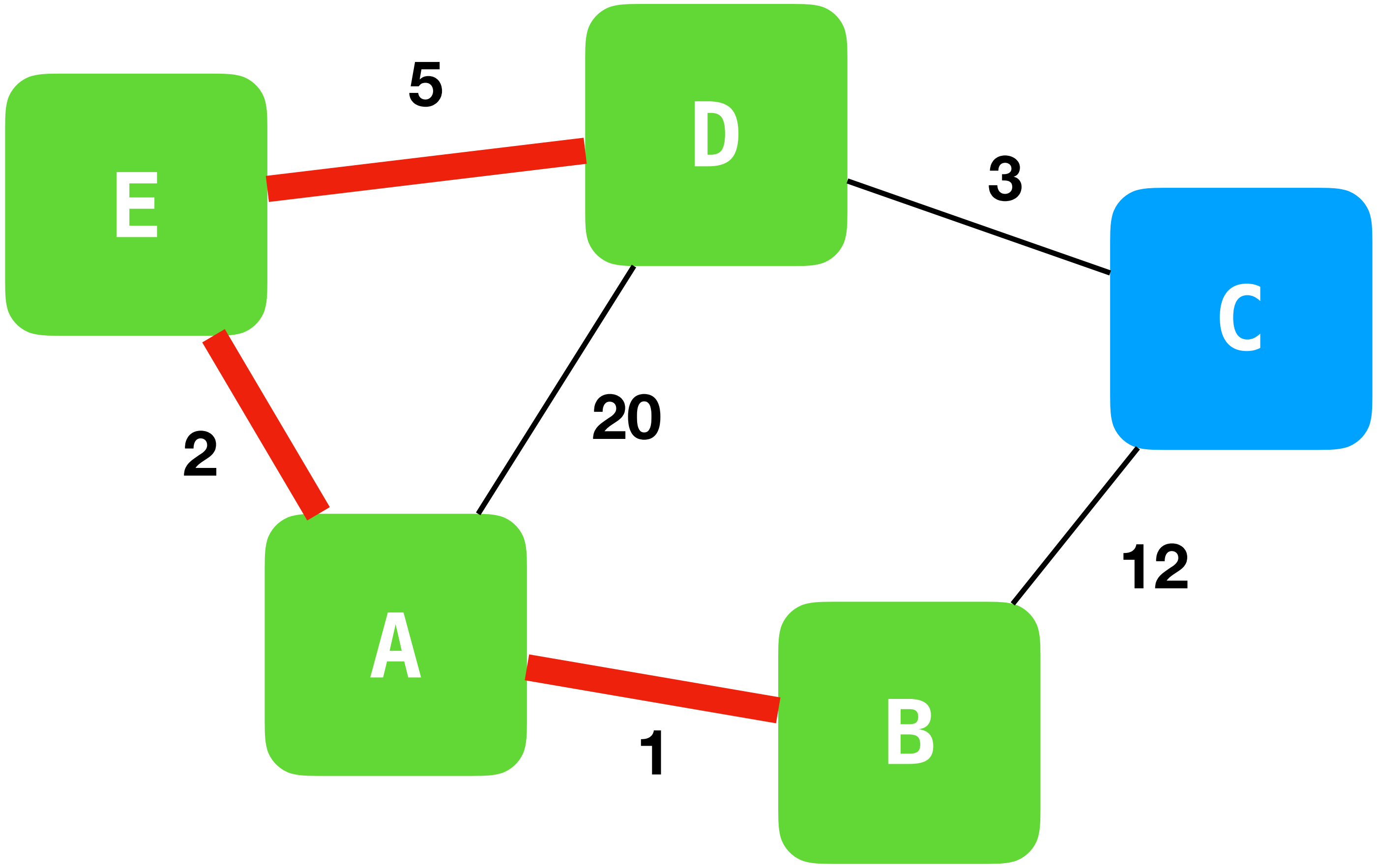
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



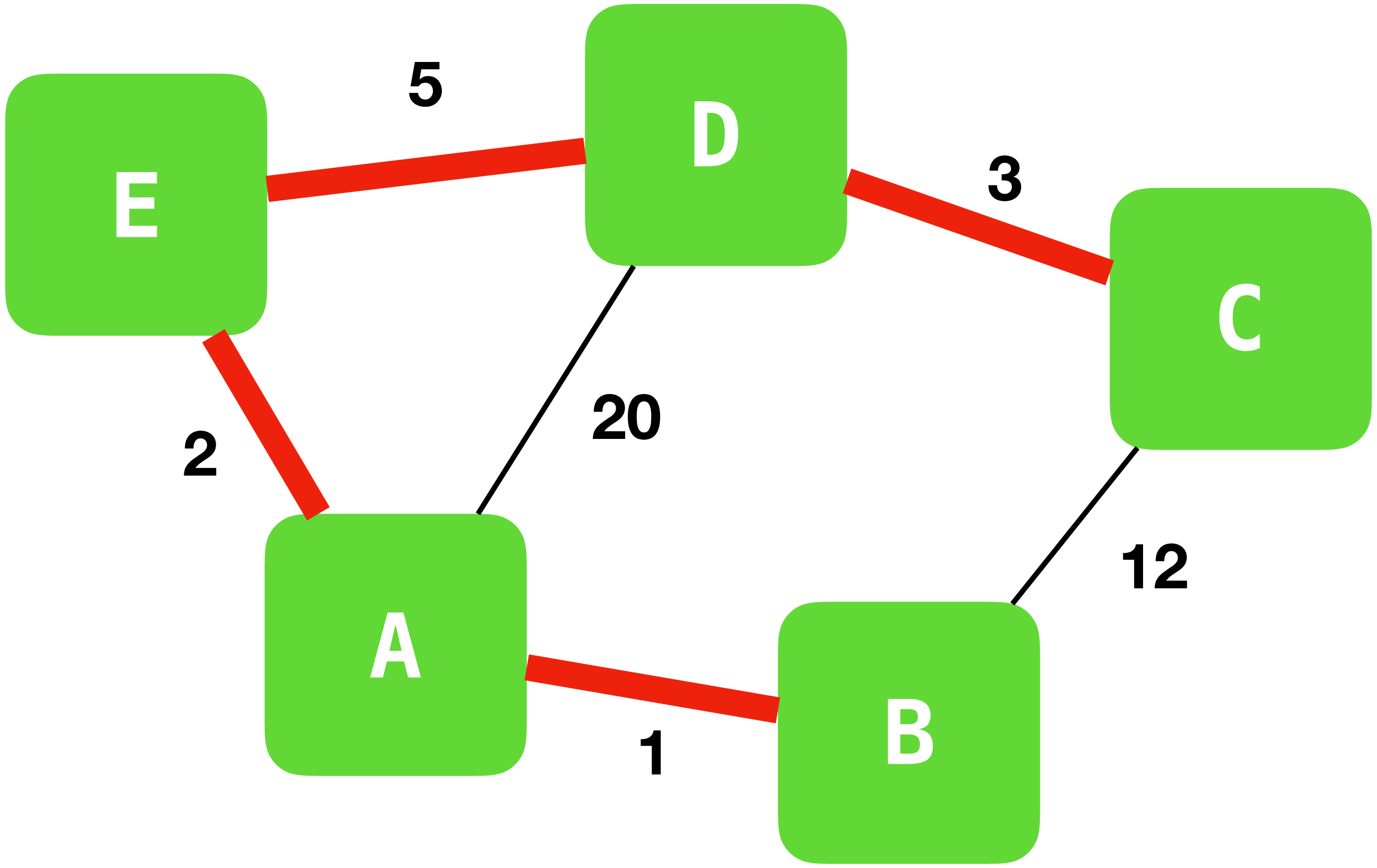
Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



Prim's Algorithm

- Start at an arbitrary node, add to MST.
- Find the closest node to the in-progress MST and add it to the MST.
- Repeat until V - 1 edges.



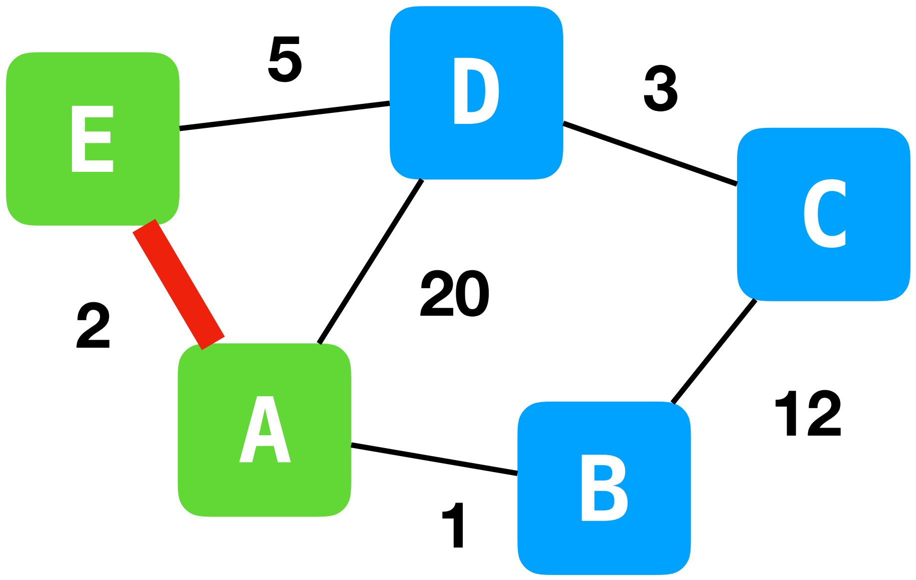
Prim's Algorithm Runtime

- We'll use a priority queue of vertices to determine which node is closest to the in-progress MST.
- Cost of inserting/updating a priority queue? $\log V$.
- How many insertions/updates are we doing? However many edges we have. E.
- In total, worst-case runtime is $\Theta(E \log V)$.

Start at an arbitrary node, add to MST.

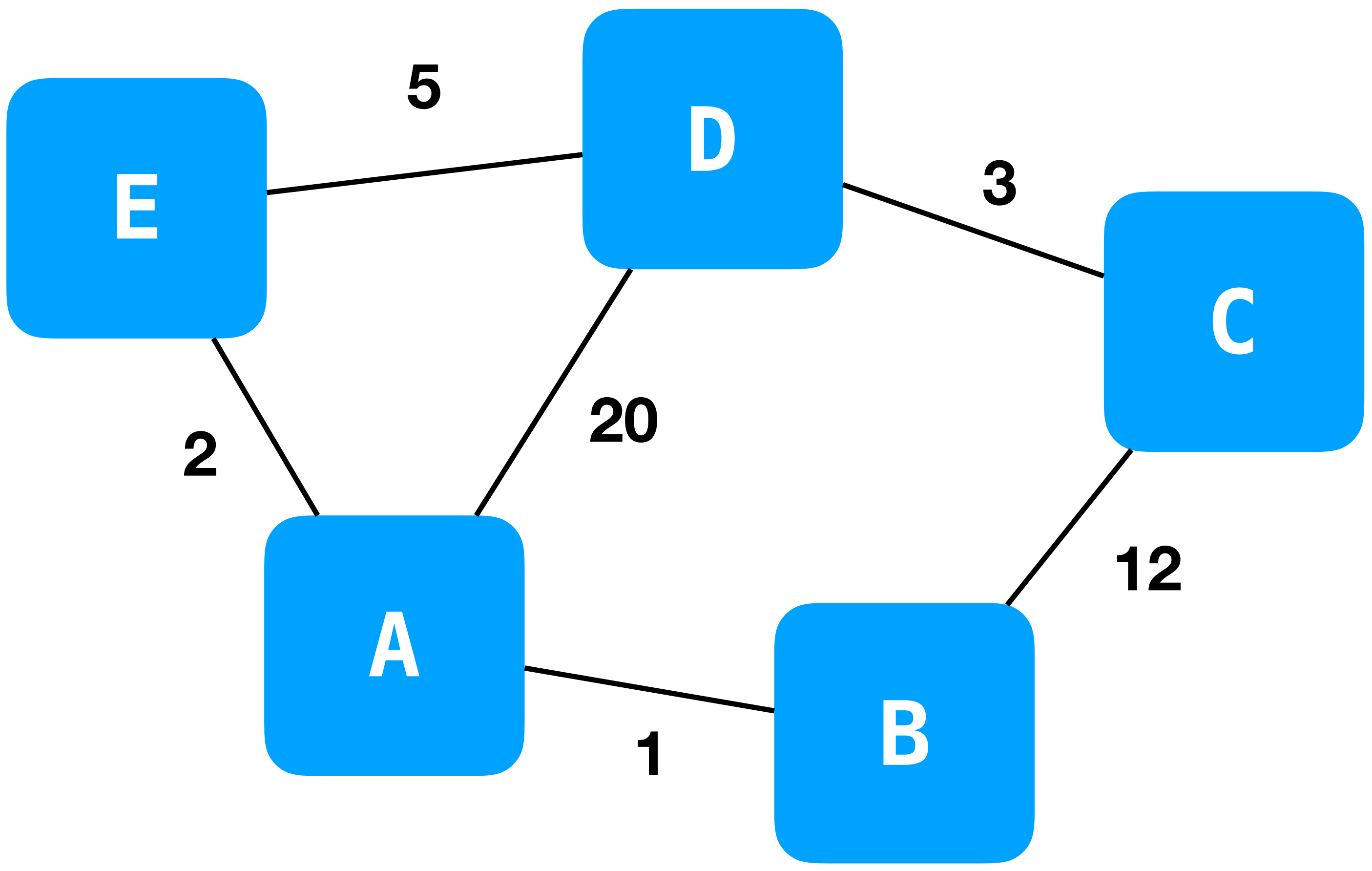
Find the closest node to the in-progress MST and add it to the MST.

Repeat until V - 1 edges.



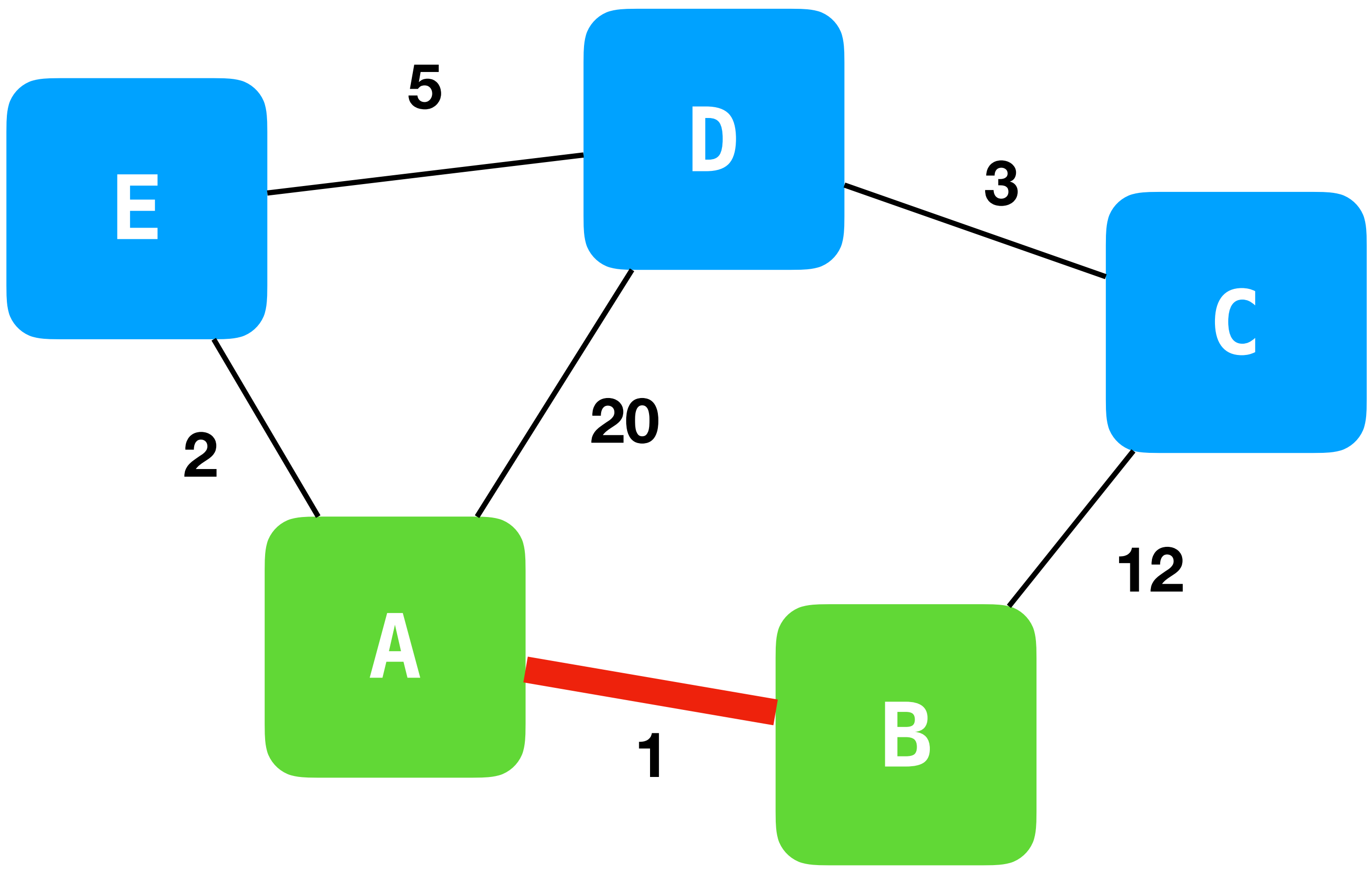
Kruskal's Algorithm

- Consider edges in order of weight, smallest first.
- Add edge to MST unless creates a cycle.
- Repeat until V - 1 edges.



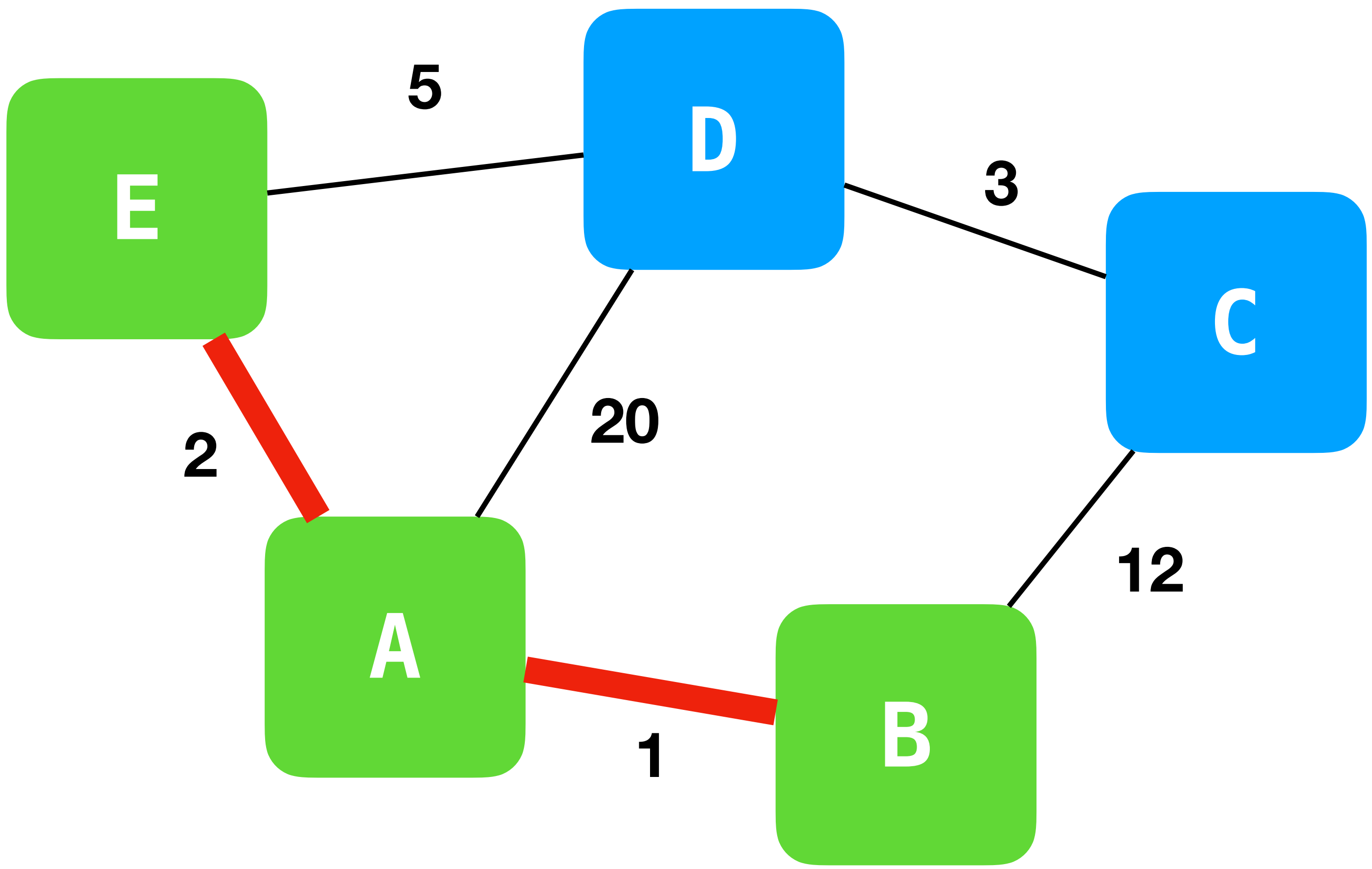
Kruskal's Algorithm

- Consider edges in order of weight, smallest first.
- Add edge to MST unless creates a cycle.
- Repeat until V - 1 edges.



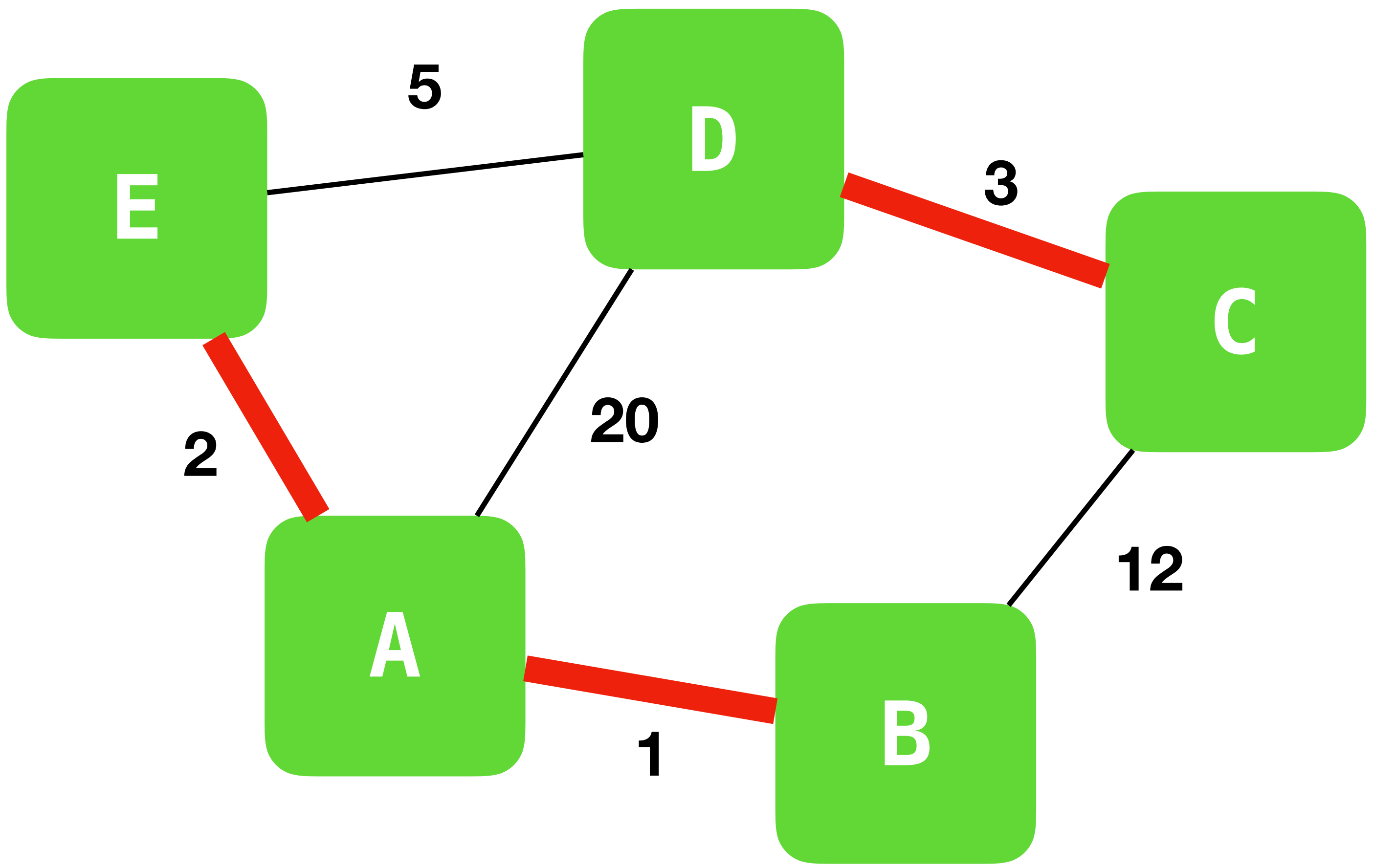
Kruskal's Algorithm

- Consider edges in order of weight, smallest first.
- Add edge to MST unless creates a cycle.
- Repeat until V - 1 edges.



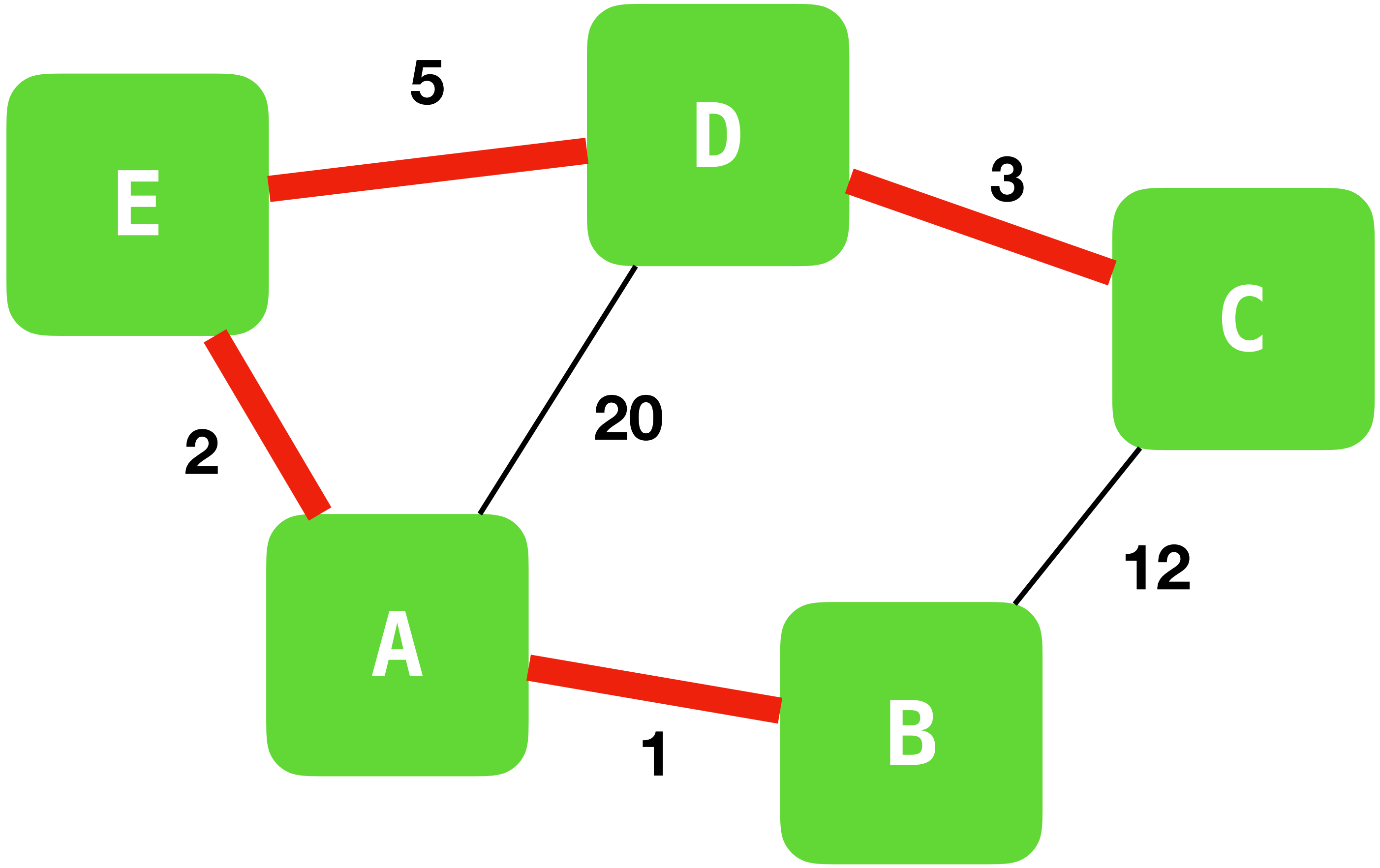
Kruskal's Algorithm

- Consider edges in order of weight, smallest first.
- Add edge to MST unless creates a cycle.
- Repeat until V - 1 edges.



Kruskal's Algorithm

- Consider edges in order of weight, smallest first.
- Add edge to MST unless creates a cycle.
- Repeat until V - 1 edges.



Kruskal's Algorithm Runtime

- Cost of sorting edges? $E \log E$
- Cost of cycle detection? We need to check if the two nodes joined by the edge are already connected by the in-progress MST.
  - We can use a weighted quick union to see if the two nodes are connected already! Cost: $E \log V$ since we do ~V unions and at worst ~E finds (if we detect a lot of cycles)
- $E > V$ assuming graph is connected so worst-case runtime is $E \log E$.

Consider edges in order of weight, smallest first.

Add edge to MST unless creates a cycle.

Repeat until V - 1 edges.