CSE100: Design and Analysis of Algorithms
Lecture 24 – Greedy Algorithms (wrap up) and Minimum Spanning Trees

Apr 21st 2022

Huffman coding, Minimum Spanning Trees!

Last time

- Greedy algorithms
  - Make a series of choices.
    - Choose this activity, then that one, ..
    - Never backtrack.
  - Show that, at each step, your choice does not rule out success.
    - At every step, there exists an optimal solution consistent with the choices we've made so far.
  - At the end of the day:
    - you've built only one solution,
    - never having ruled out success,
    - **so your solution must be correct.**

One more example

Huffman coding

- everyday english sentence
- 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000 01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101

- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

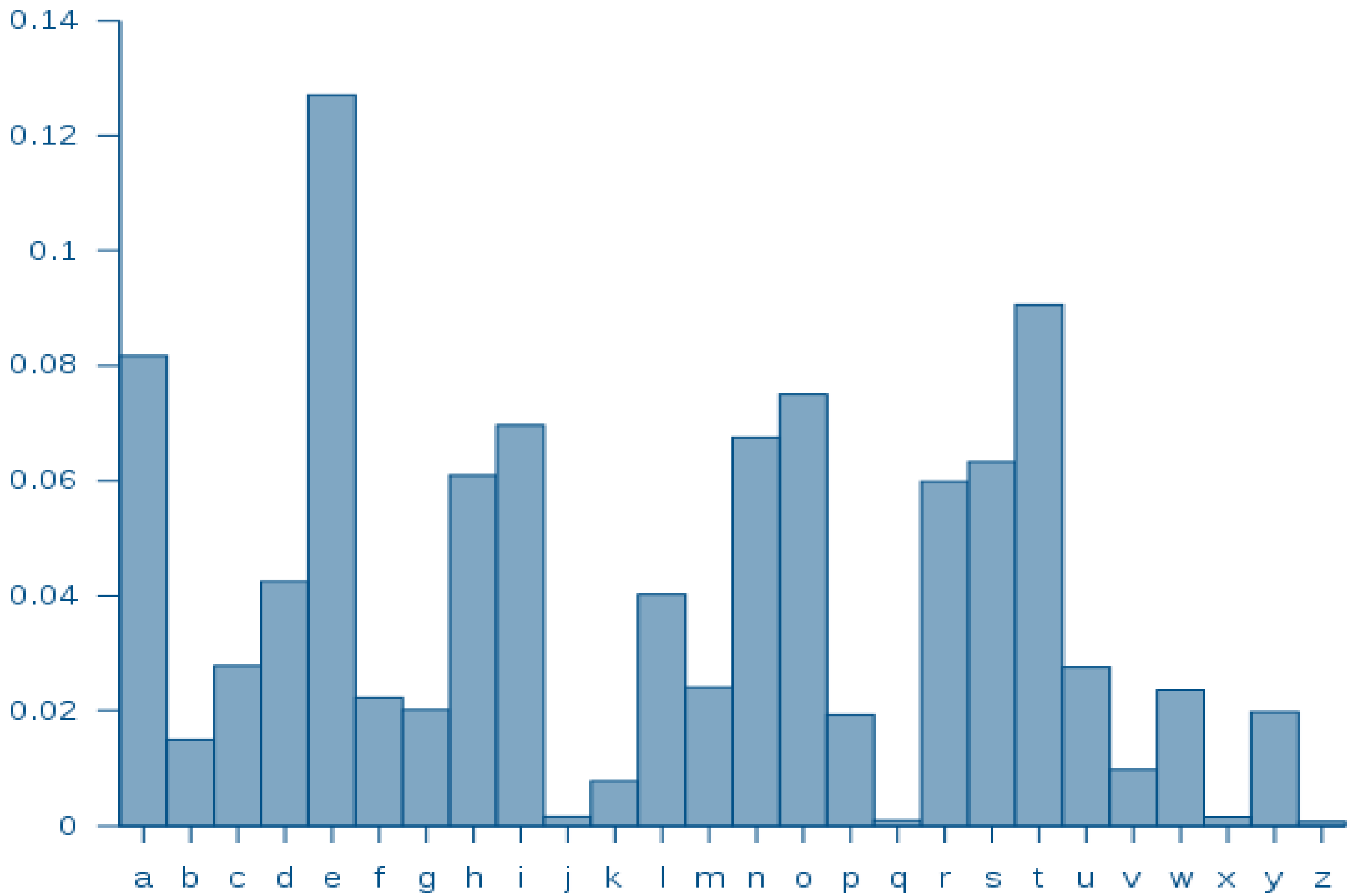
One more example
Huffman coding

ASCII is pretty wasteful for English sentences. If **e** shows up so often, we should have a more parsimonious way of representing it!

- **e**v**e**ryday **e**nglish s**e**nt**e**nc**e**
- **01100101** 01110110 **01100101** 01110010 01111001 01100100 01100001 01111001 00100000 **01100101** 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 **01100101** 01101110 01110100 **01100101** 01101110 01100011 **01100101**

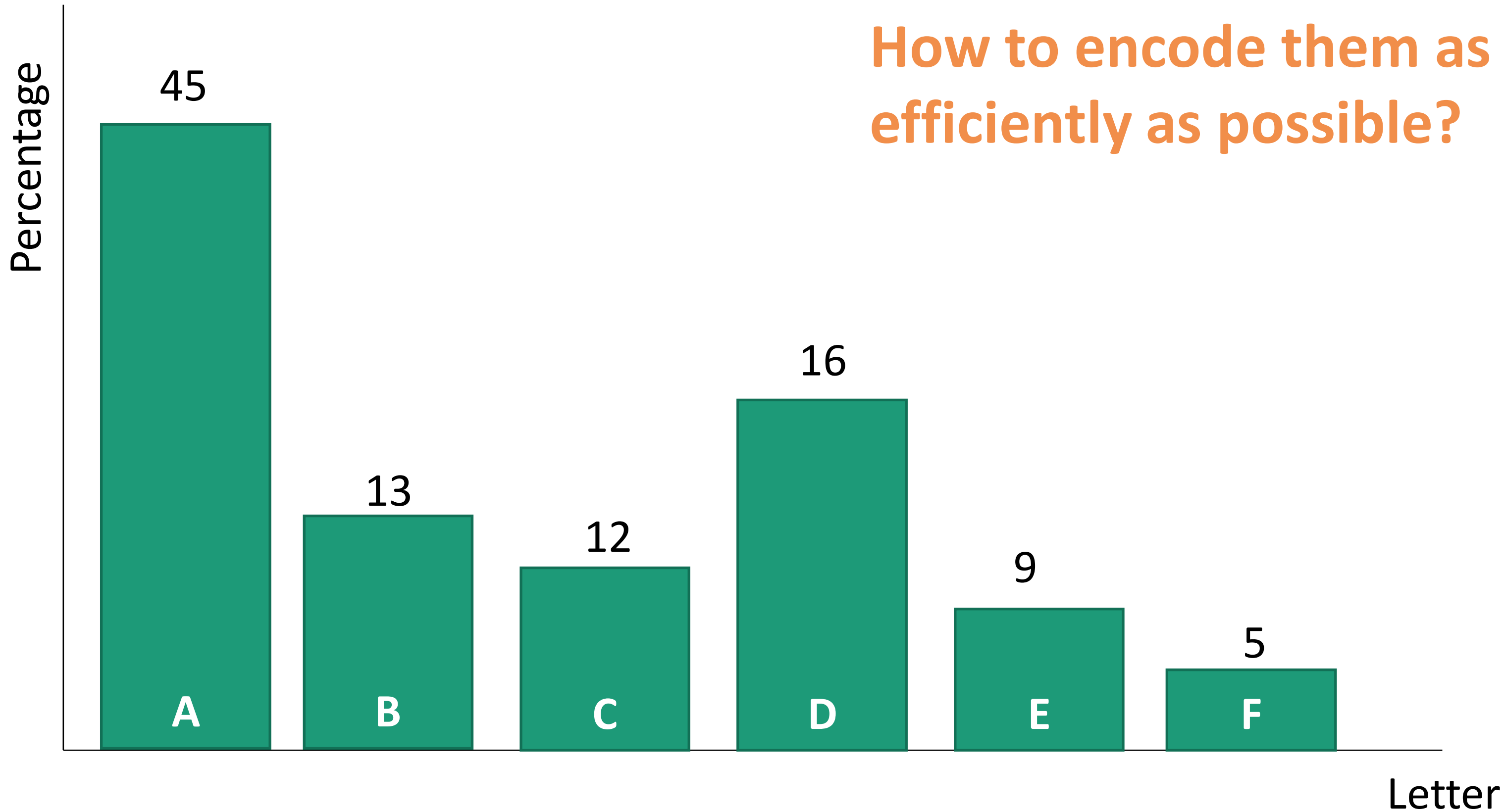
- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Suppose we have some distribution on characters



Suppose we have some distribution on characters

For simplicity, let's go with this made-up example



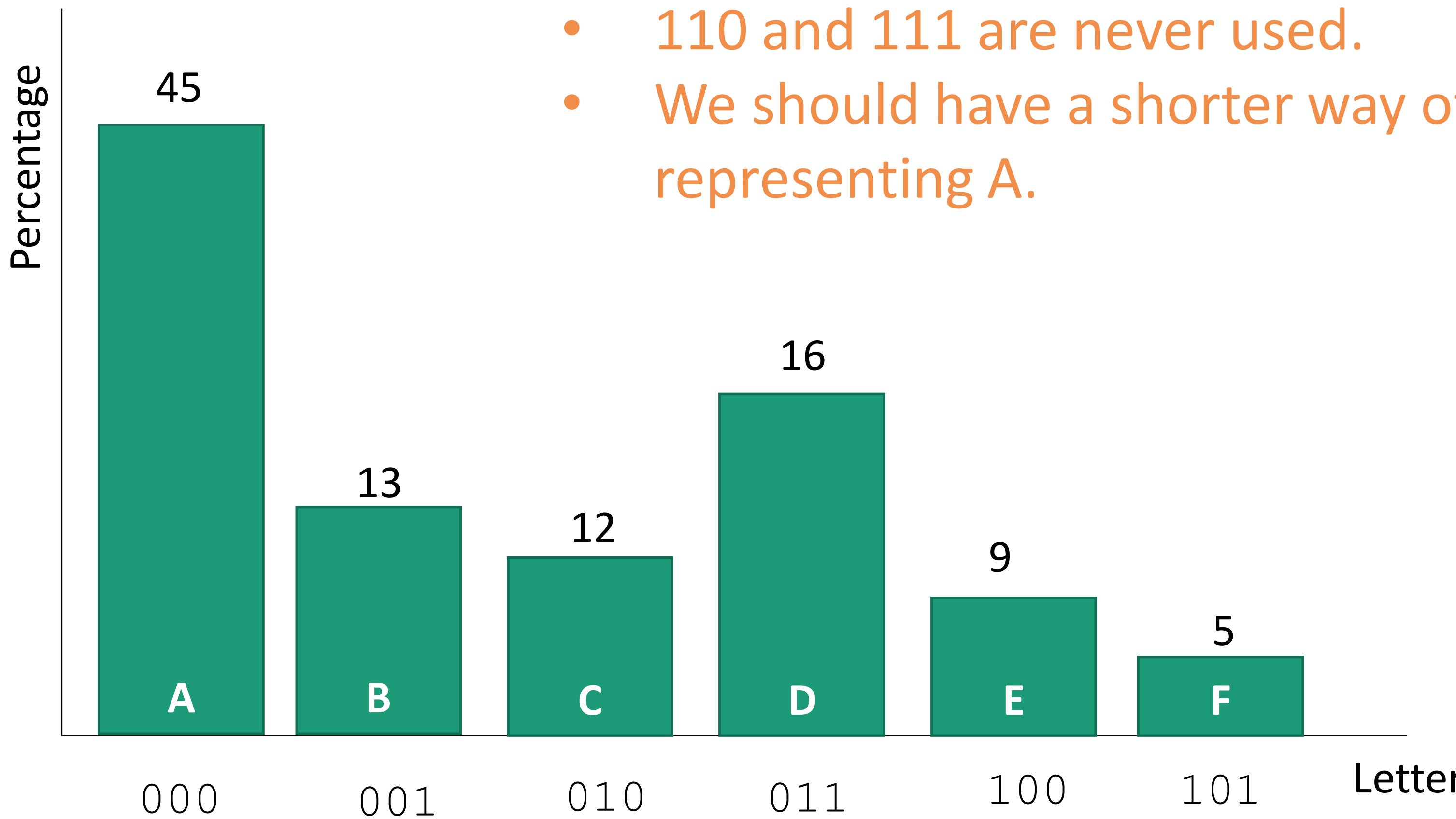
How to encode them as efficiently as possible?

Try 0
(like ASCII)

- Every letter is assigned a **binary string** of three bits.

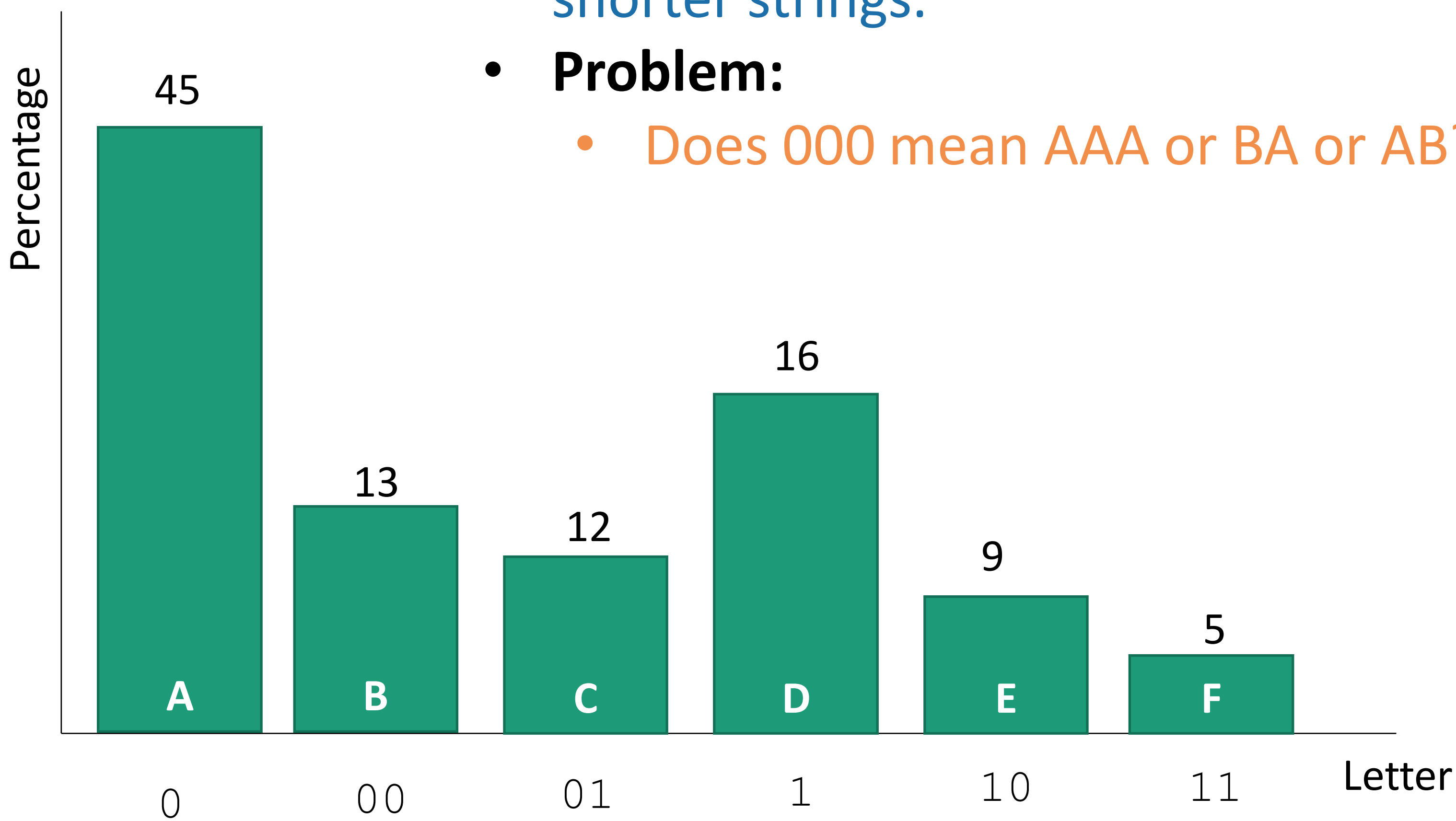
Wasteful!

- 110 and 111 are never used.
- We should have a shorter way of representing A.



Try 1

- Every letter is assigned a **binary string** of one or two bits.
- The more frequent letters get the shorter strings.
- **Problem:**
  - Does 000 mean AAA or BA or AB?

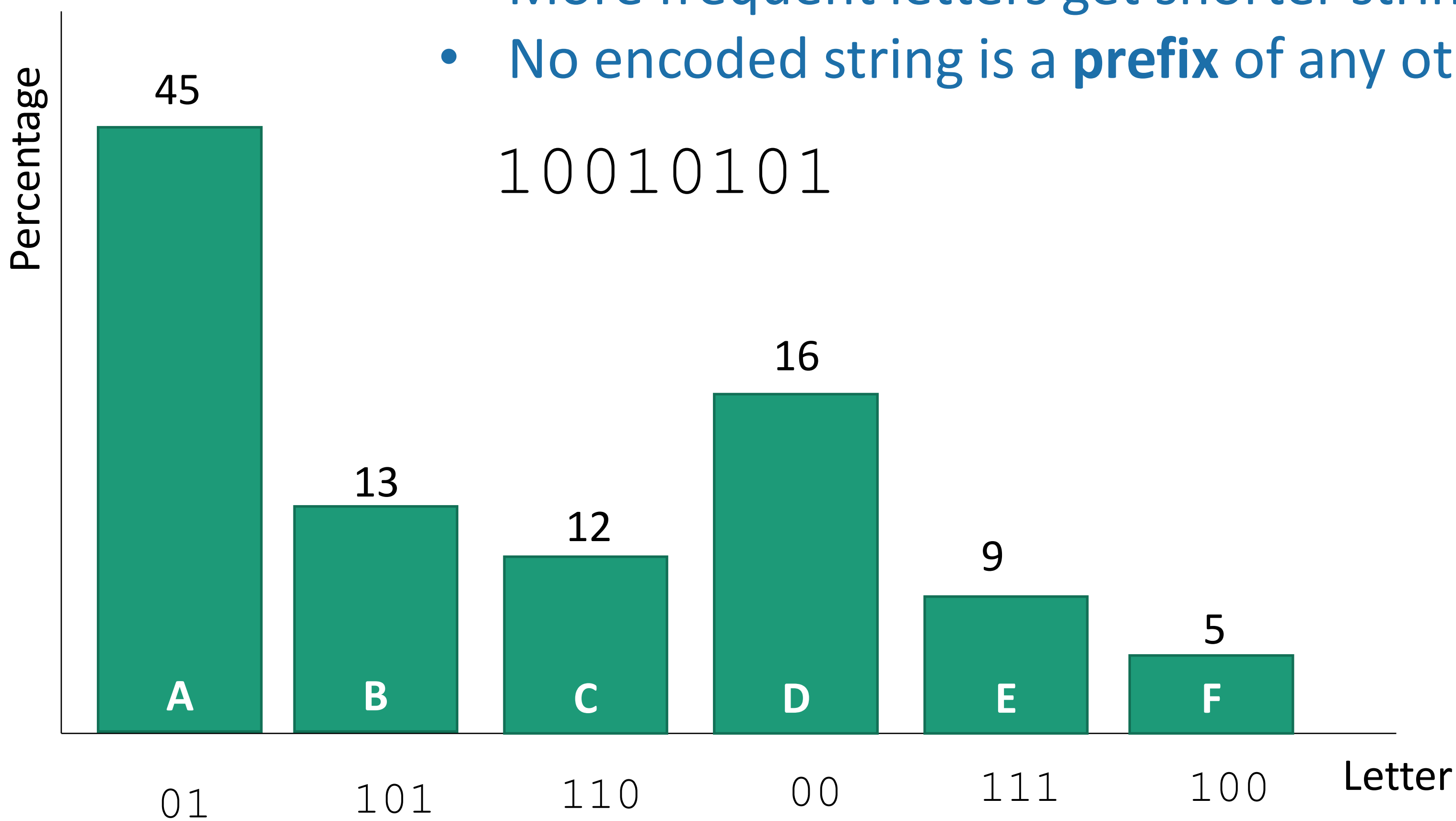


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

Try 2: prefix-free coding

- Every letter is assigned a **binary string.**
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

10010101

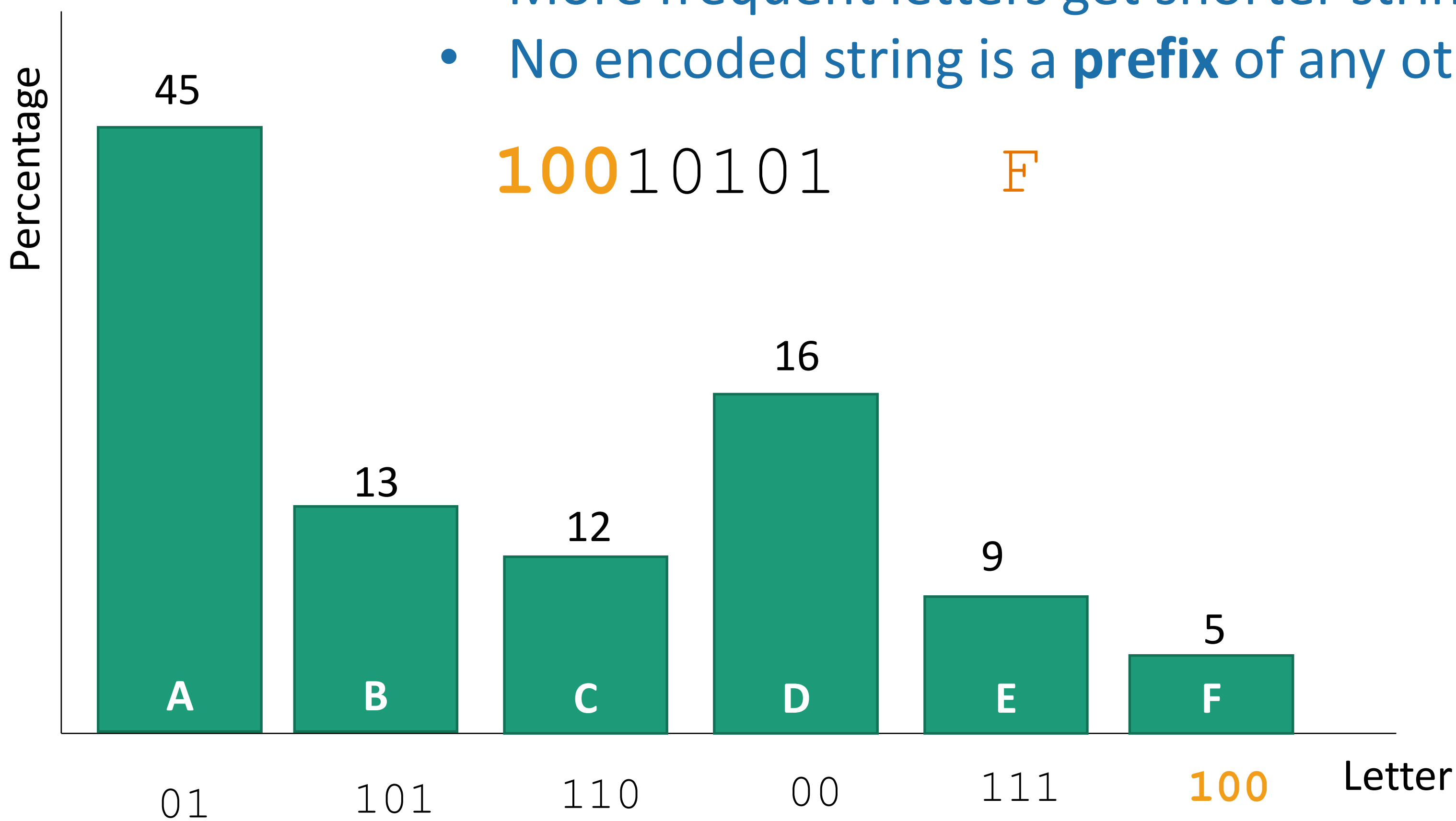


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

Try 2: prefix-free coding

- Every letter is assigned a **binary string.**
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

10010101 F

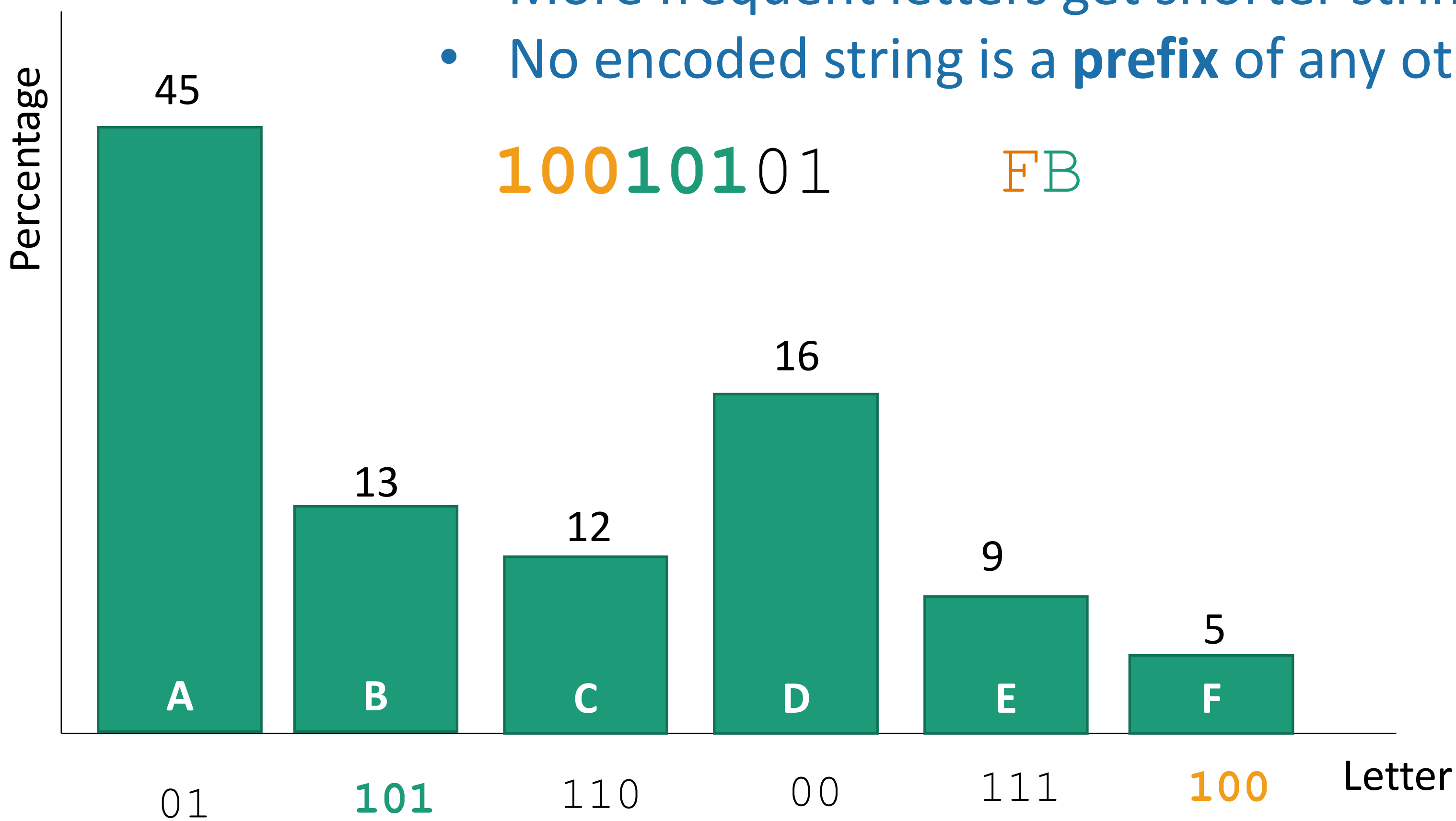


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

Try 2: prefix-free coding

- Every letter is assigned a **binary string.**
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

10010101 FB



Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

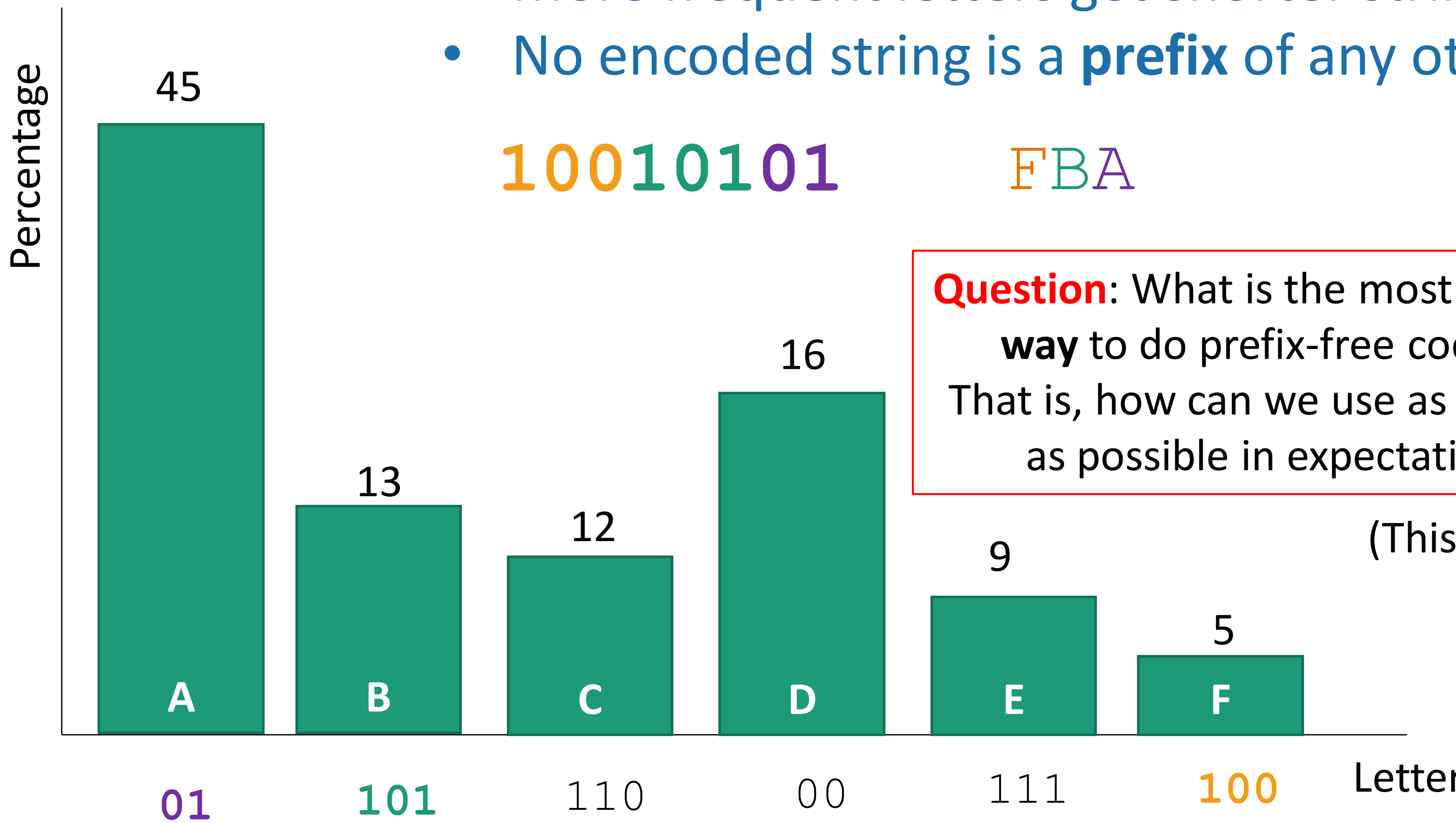
Try 2: prefix-free coding

- Every letter is assigned a **binary string.**
- More frequent letters get shorter strings.
- No encoded string is a **prefix** of any other.

10010101 FBA

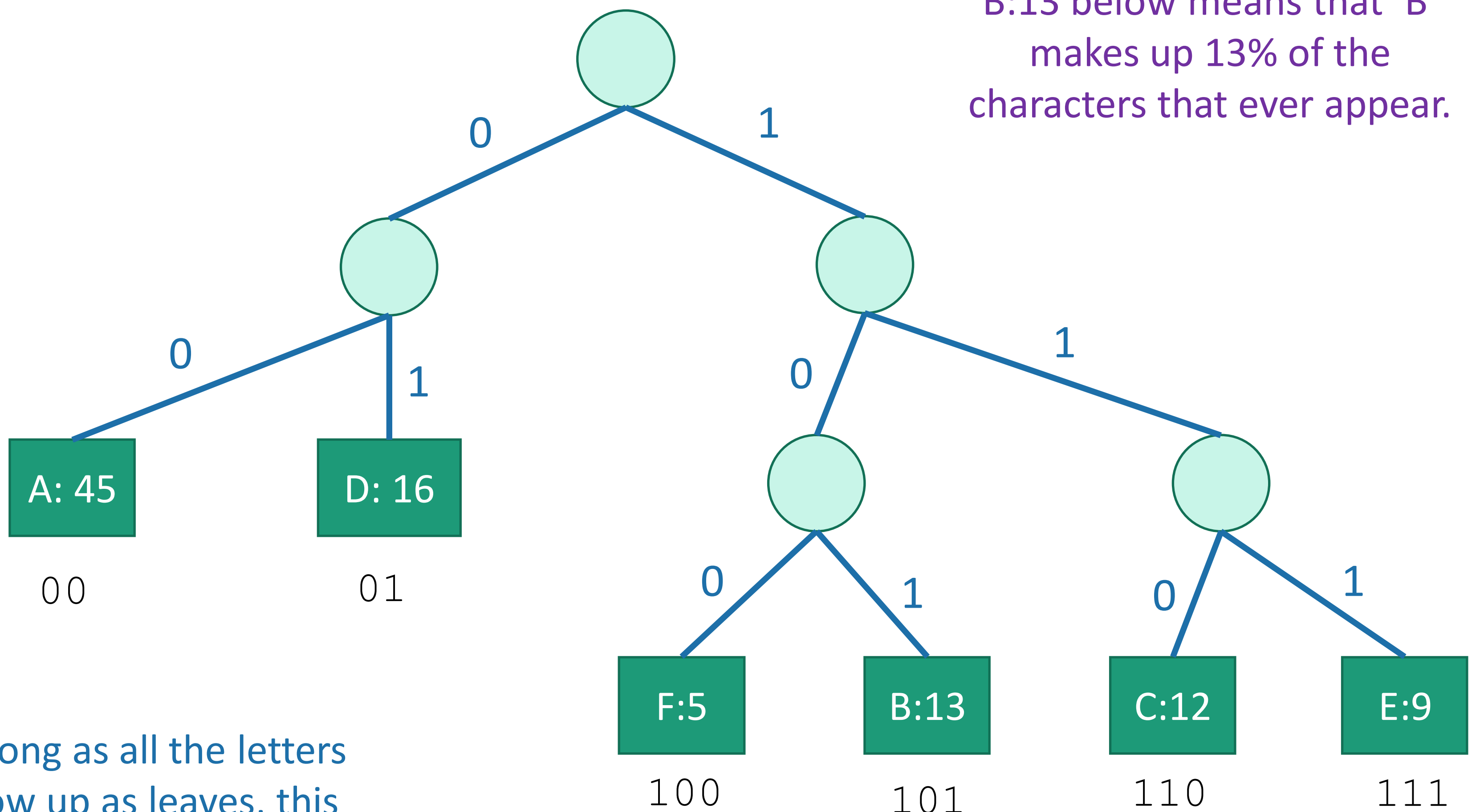
Question: What is the most **efficient way** to do prefix-free coding? That is, how can we use as few bits as possible in expectation?

(This is not it).



A prefix-free code is a tree

B:13 below means that 'B' makes up 13% of the characters that ever appear.



As long as all the letters show up as leaves, this code is **prefix-free**.

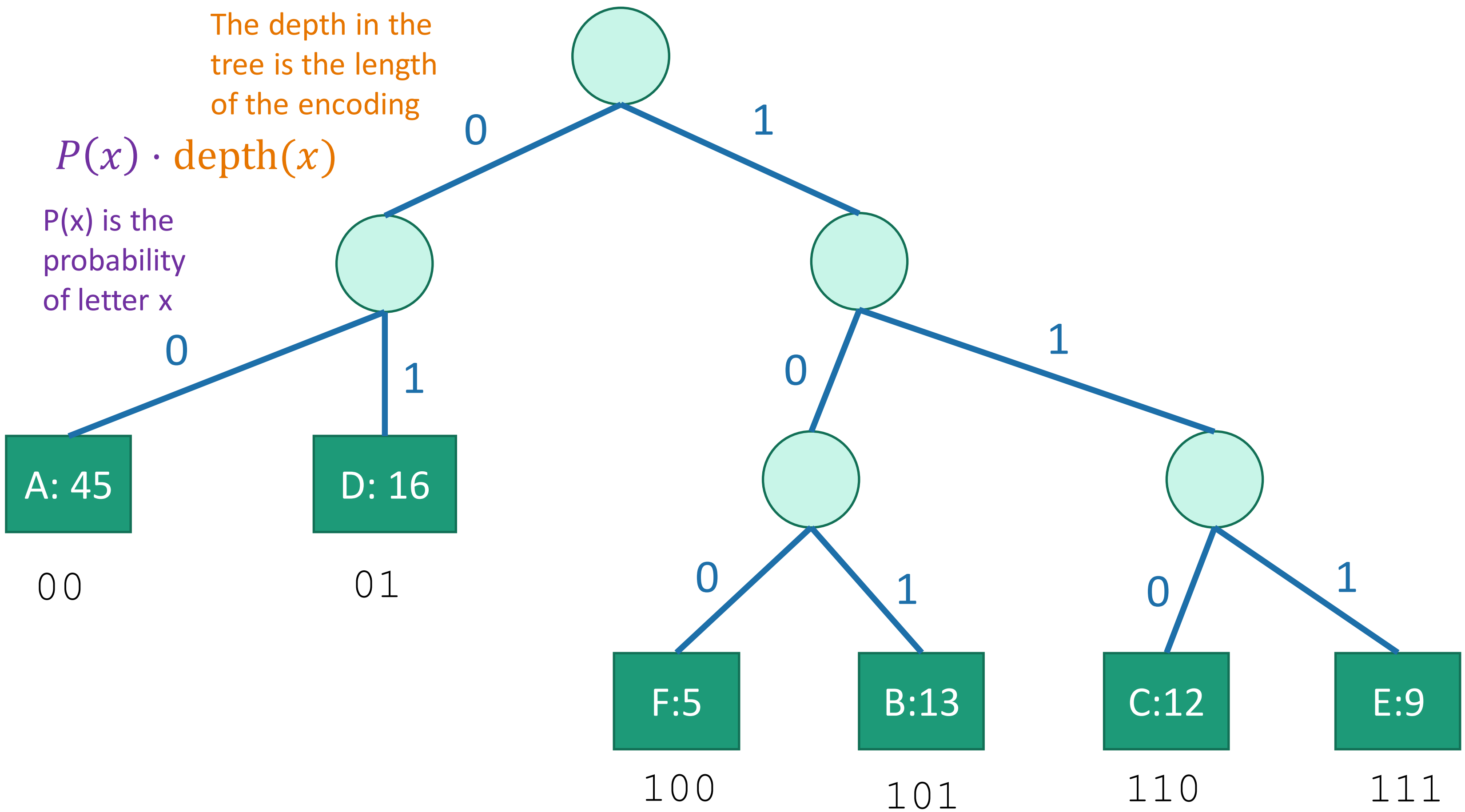
How good is a tree?

- Imagine choosing a letter at random from the language.
  - Not uniform, but according to our histogram!
- The **cost of a tree** is the expected length of the encoding of that letter.

Cost = $\sum_{leaves\ x} P(x) \cdot \mathrm{depth}(x)$

The depth in the tree is the length of the encoding

P(x) is the probability of letter x



Expected cost of encoding a letter with this tree:

$$\mathbf{2(0.45 + 0.16) + 3(\ 0.05 + 0.13 + 0.12 + 0.09) = 2.39}$$

Question

- Given a distribution P on letters, find the lowest-cost tree, where

$$\text{cost}(\text{tree}) = \sum_{\text{leaves } x} P(x) \cdot \text{depth}(x)$$

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

Greedy algorithm

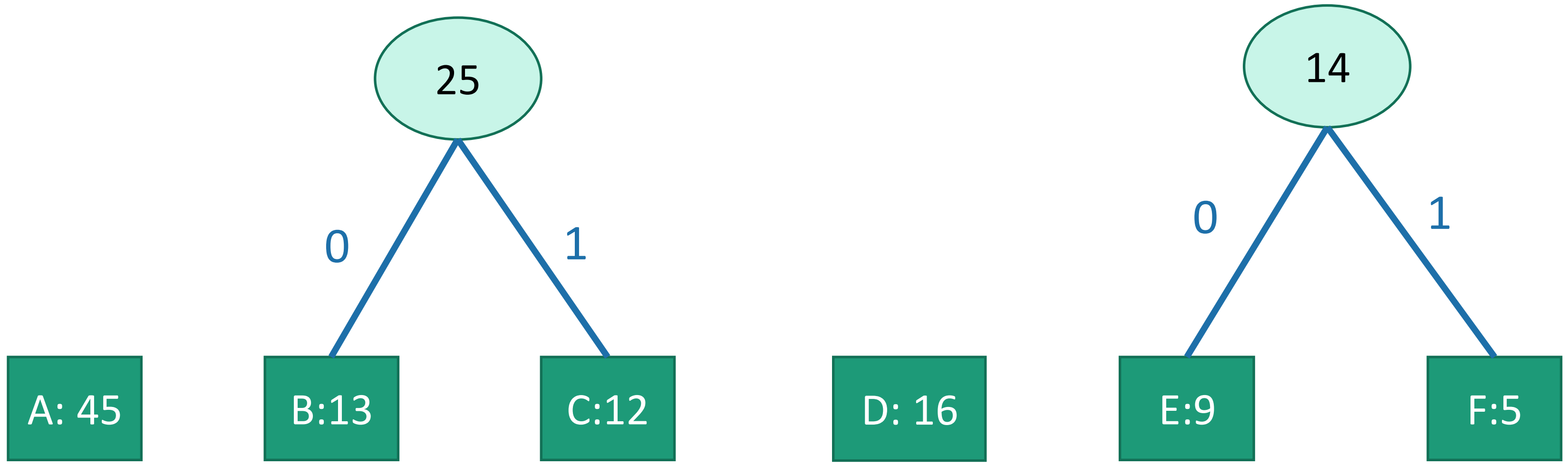
- Greedily build sub-trees from the bottom up.
- Greedy goal: less frequent letters should be further down the tree.

Solution

greedily build subtrees, starting with the infrequent letters

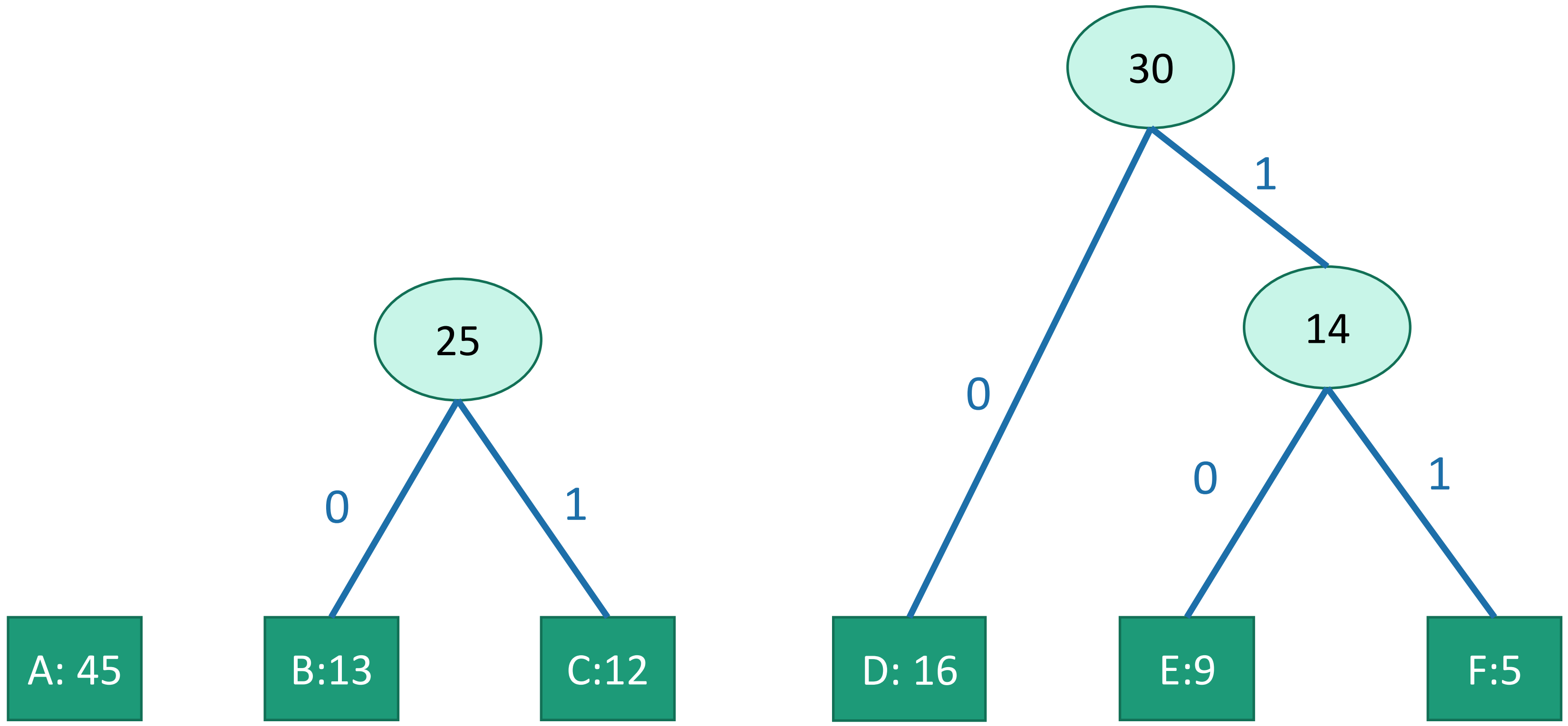
Solution

greedily build subtrees, starting with the infrequent letters



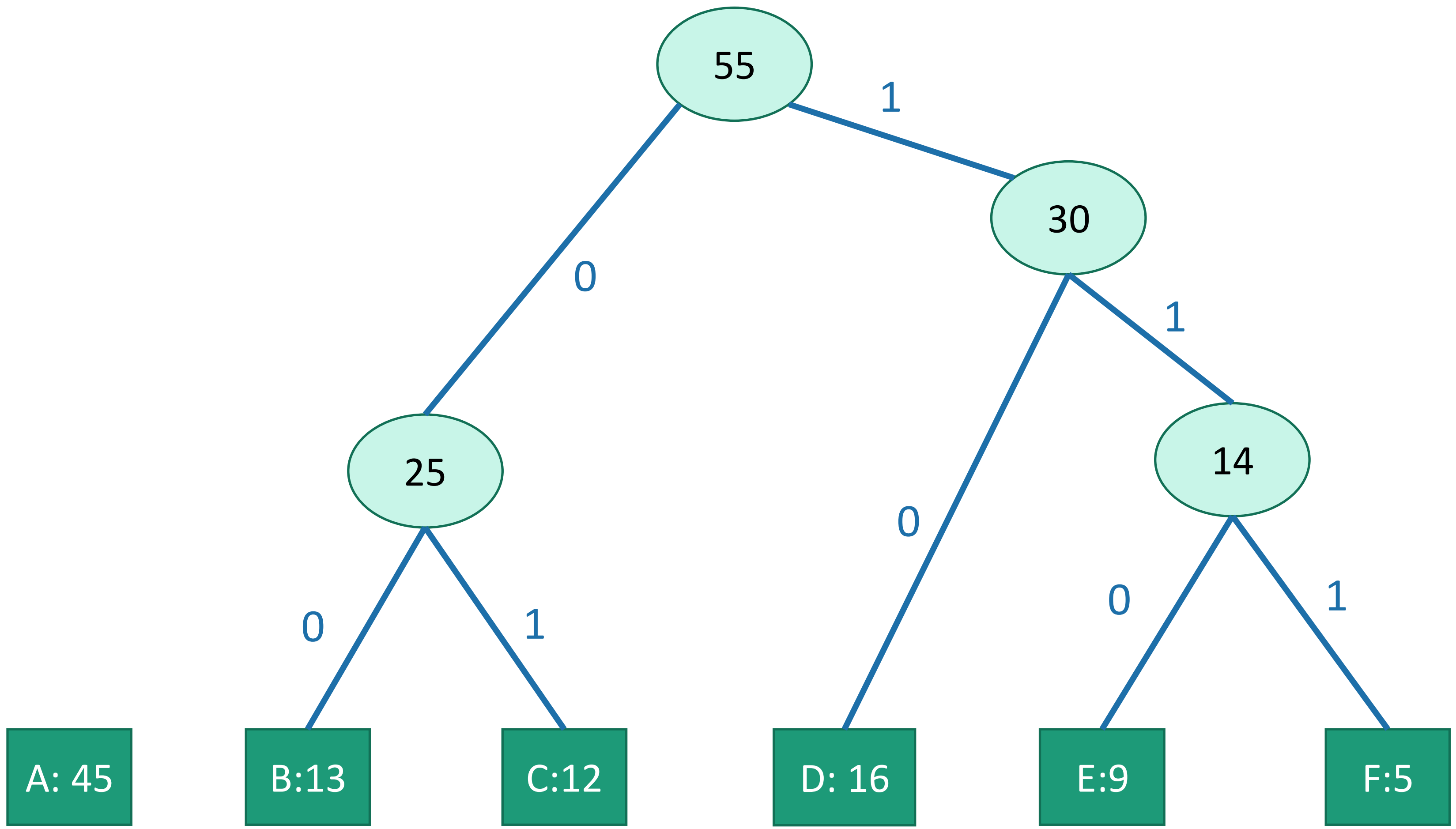
Solution

greedily build subtrees, starting with the infrequent letters



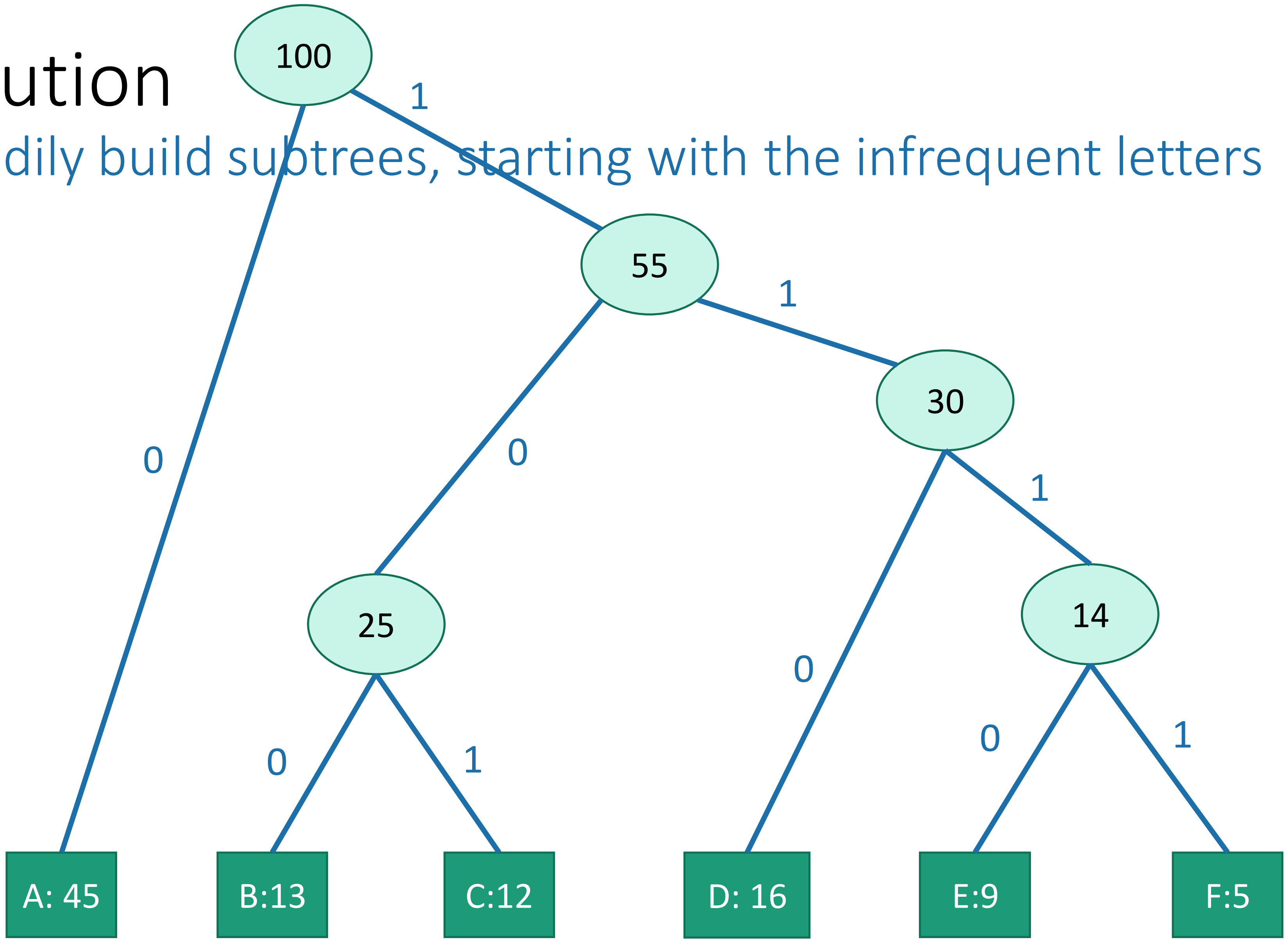
Solution

greedily build subtrees, starting with the infrequent letters



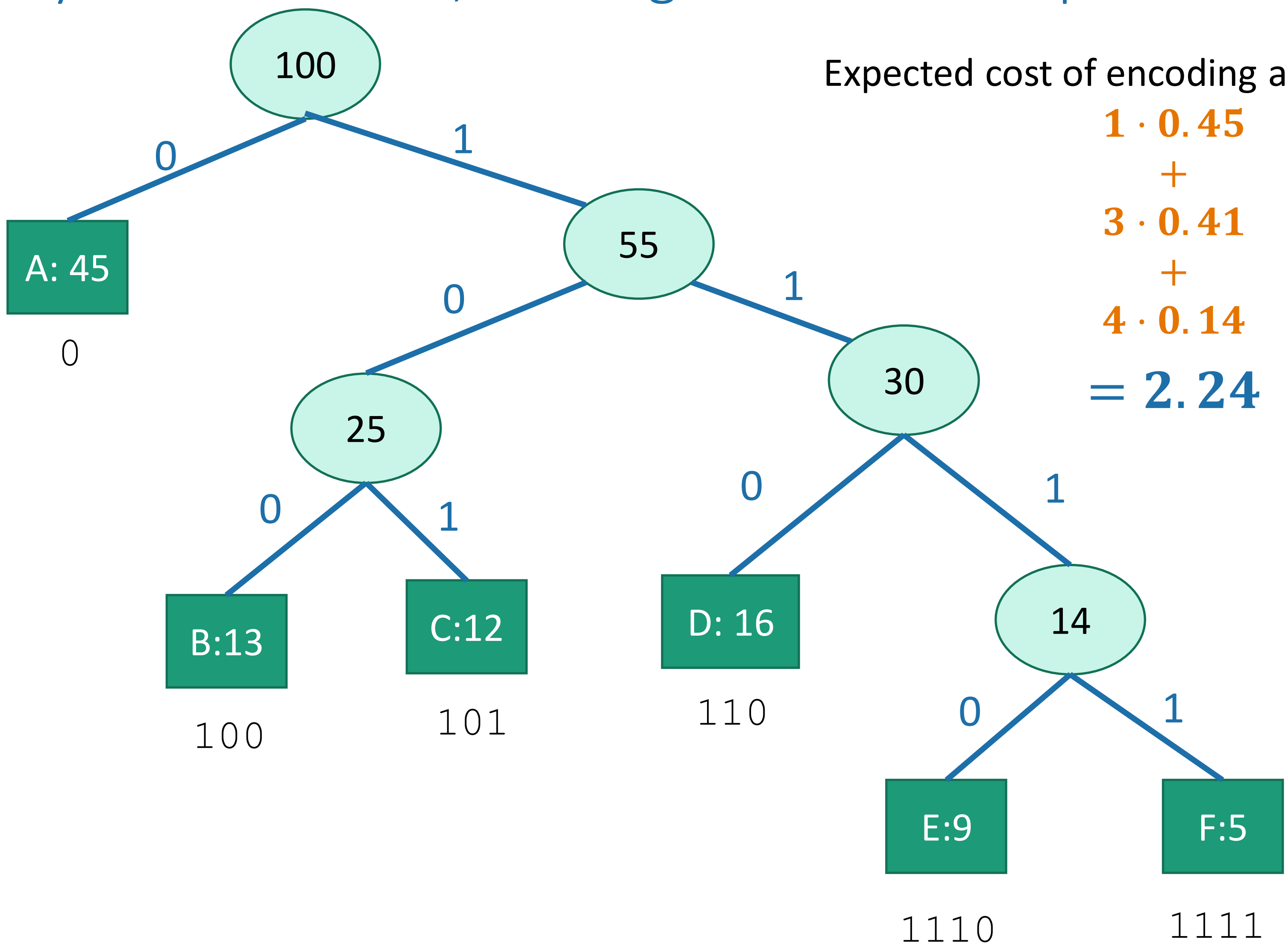
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters

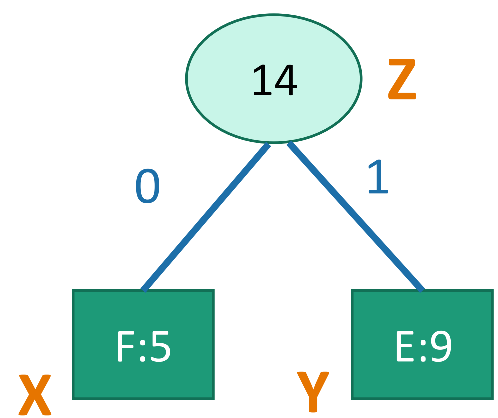


Expected cost of encoding a letter:

$$1 \cdot 0.45 + 3 \cdot 0.41 + 4 \cdot 0.14 = 2.24$$

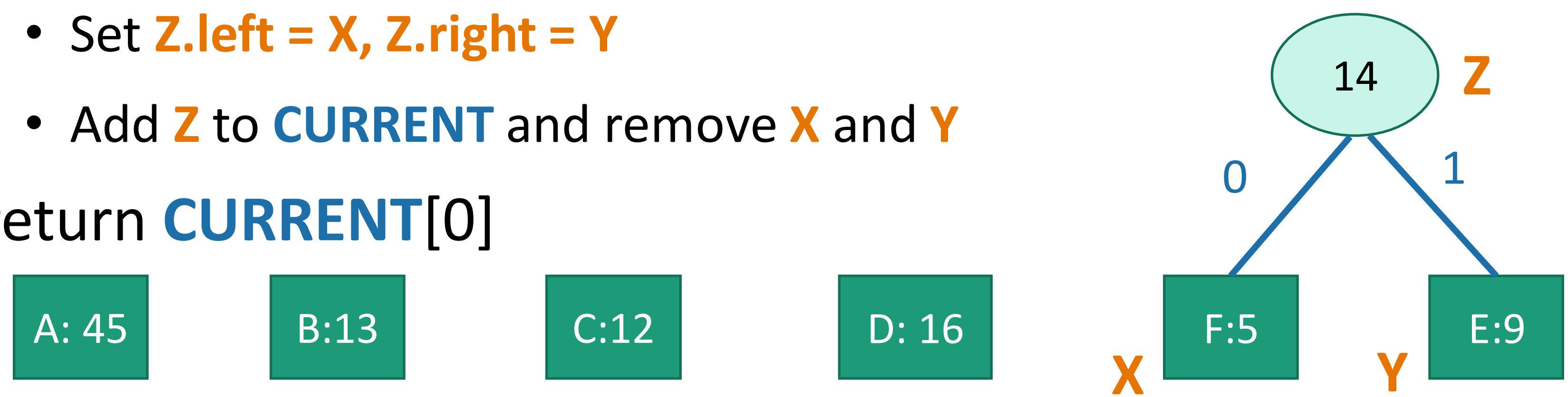
What exactly was the algorithm?

- Create a node like D: 16 for each letter/frequency
  - The key is the frequency (16 in this case)
- Let **CURRENT** be the list of all these nodes.
- **while** len(**CURRENT**) > 1:
  - X and Y ← the nodes in **CURRENT** with the smallest keys.
  - Create a new node Z with **Z.key = X.key + Y.key**
  - Set **Z.left = X, Z.right = Y**
  - Add Z to **CURRENT** and remove X and Y
- return **CURRENT**[0]



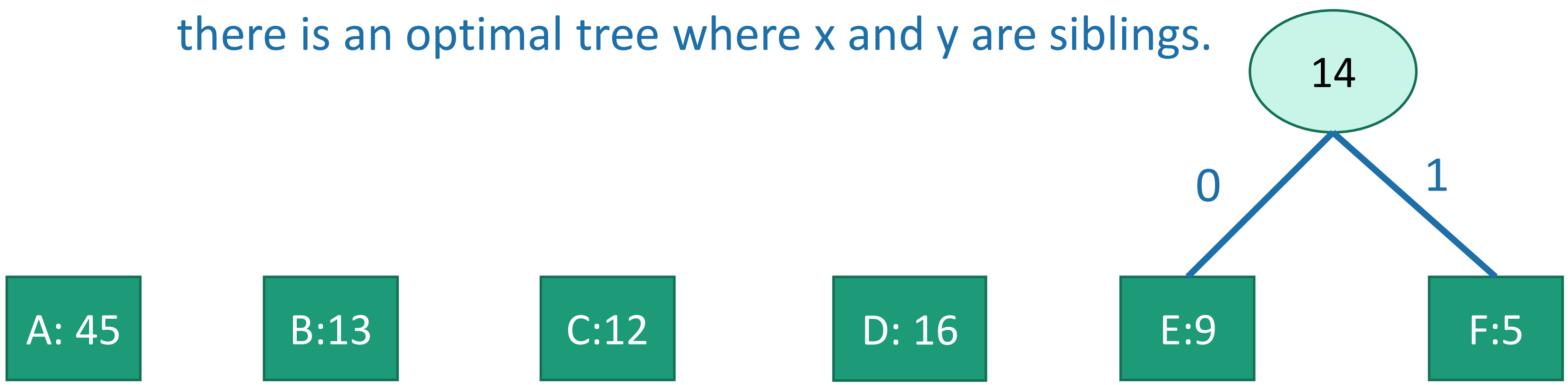
This is called Huffman Coding:

- Create a node like D: 16 for each letter/frequency
  - The key is the frequency (16 in this case)
- Let **CURRENT** be the list of all these nodes.
- **while** len(**CURRENT**) > 1:
  - **X** and **Y** ← the nodes in **CURRENT** with the smallest keys.
  - Create a new node **Z** with **Z.key = X.key + Y.key**
  - Set **Z.left = X, Z.right = Y**
  - Add **Z** to **CURRENT** and remove **X** and **Y**
- return **CURRENT**[0]



Does it work?

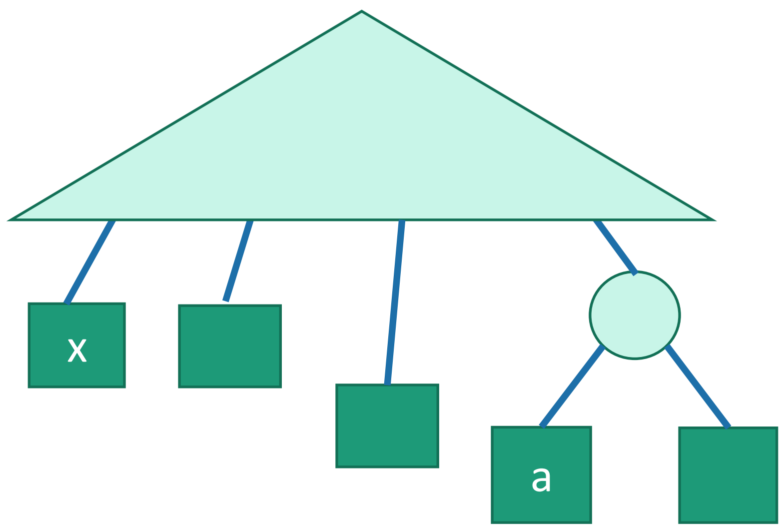
- Yes.
- We will ***sketch*** a proof here.
- Same strategy:
  - Show that at each step, the choices we are making **won't rule out** an optimal solution.
  - Lemma:
    - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.



Lemma
proof idea

If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.

- Say that an optimal tree looks like this:



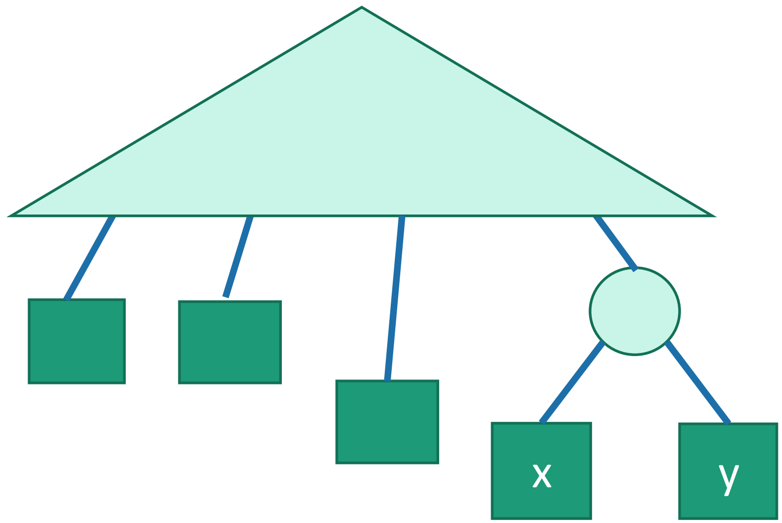
Lowest-level sibling nodes: at least one of them is neither x nor y

- What happens to the cost if we swap x for a?
  - the cost can't increase; a was more frequent than x, and we just made a's encoding shorter and x's longer.
- Repeat this logic until we get an optimal tree with x and y as siblings.
  - The cost never increased so this tree is still optimal.

Lemma
proof idea

If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.

- Say that an optimal tree looks like this:



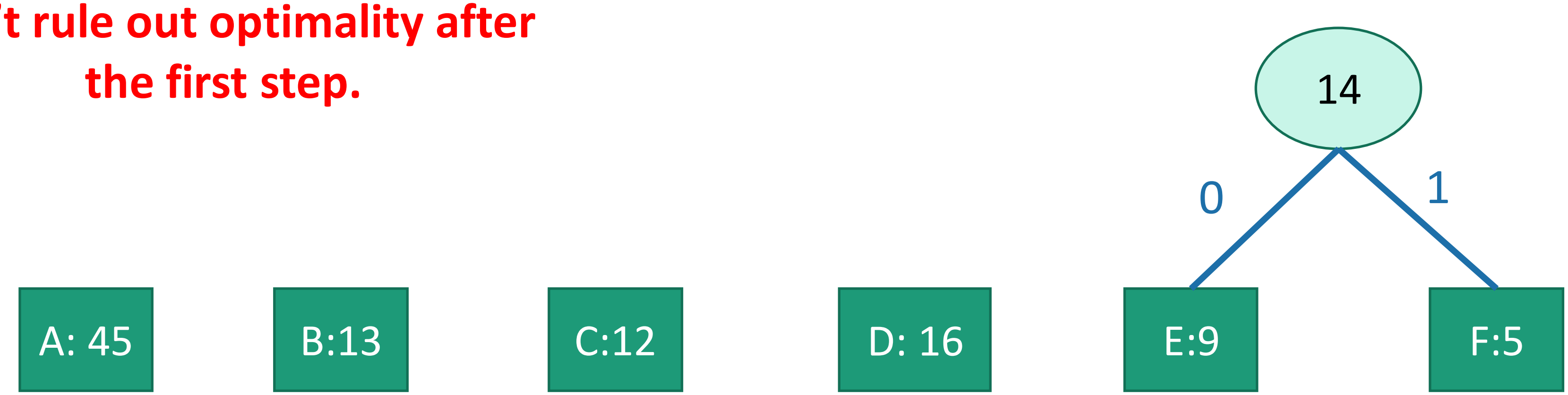
Lowest-level sibling nodes: at least one of them is neither x nor y

- What happens to the cost if we swap x for a?
  - the cost can't increase; a was more frequent than x, and we just made a's encoding shorter and x's longer.
- Repeat this logic until we get an optimal tree with x and y as siblings.
  - The cost never increased so this tree is still optimal.

Proof strategy
just like before

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

That's enough to show that we don't rule out optimality after the first step.



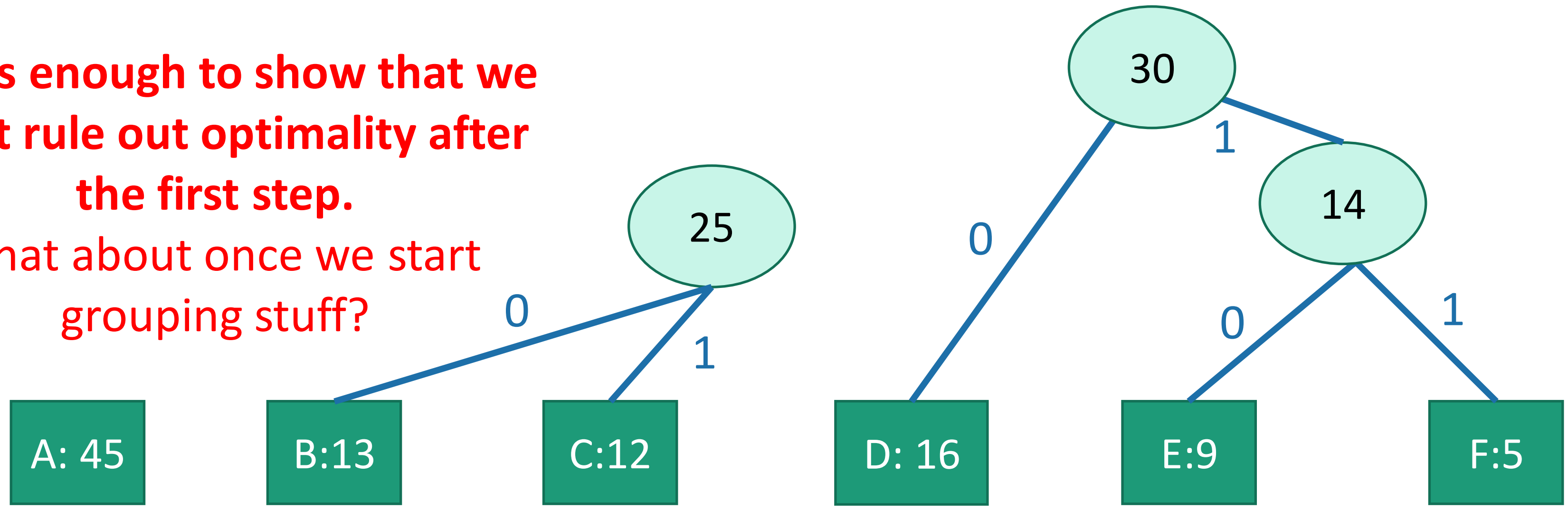
Proof strategy

just like before

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

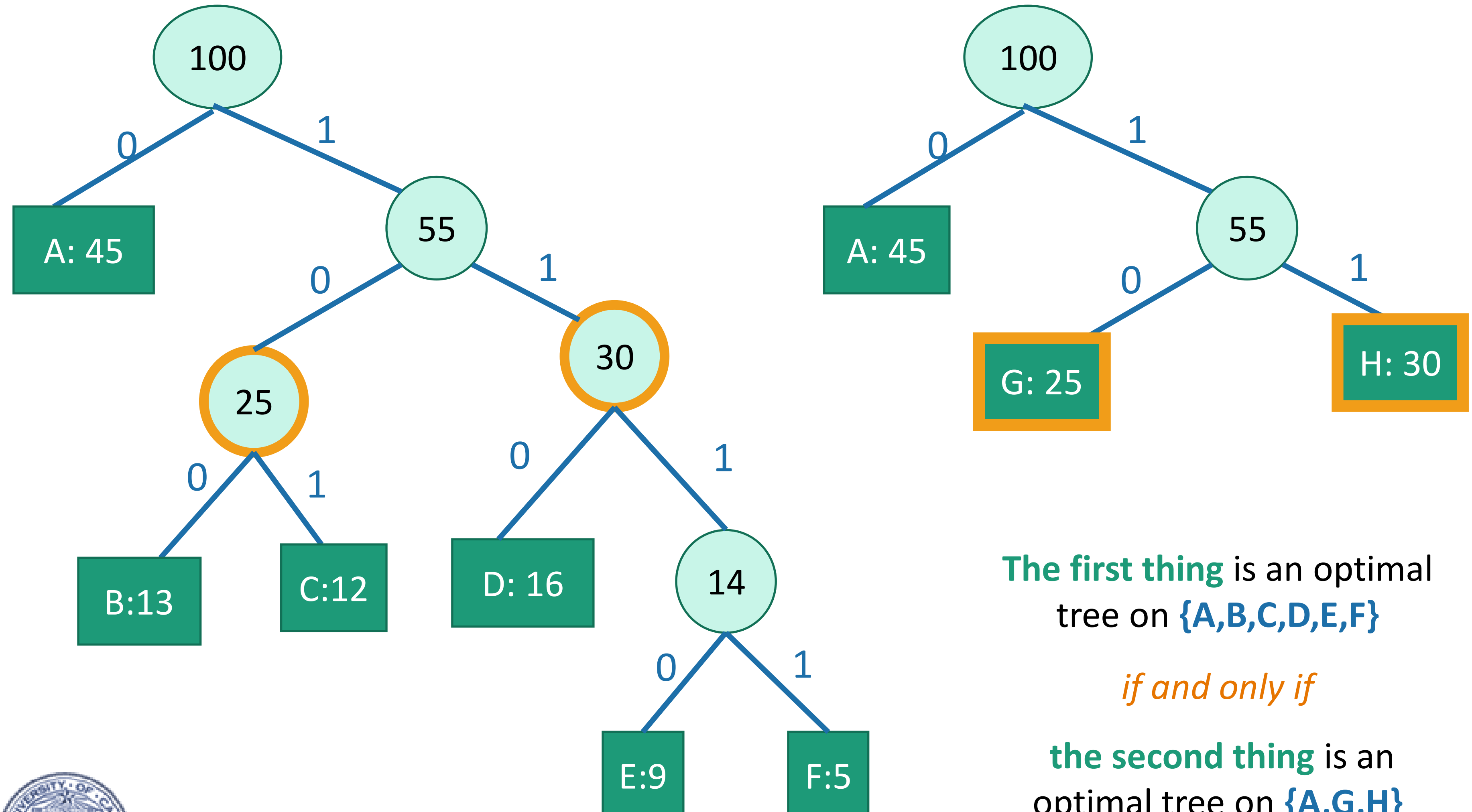
That's enough to show that we don't rule out optimality after the first step.

What about once we start grouping stuff?



Lemma 2

this distinction doesn't really matter



The first thing is an optimal tree on **{A,B,C,D,E,F}**

if and only if

the second thing is an optimal tree on **{A,G,H}**

Lemma 2
this distinction doesn't really matter

- For a proof:
  - See CLRS, Lemma 16.3
    - Rigorous although presented in a slightly different way
  - See the (optional) Lecture Notes
    - A bit sketchier, but presented in the same way as here
  - Prove it yourself!
    - This is the best!

Getting all the details isn't that important, but you should convince yourself that this is true.

Siggi the Studious Stork

Together

- Lemma 1:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.
- Lemma 2:
  - We may as well imagine that **CURRENT** contains only leaves.
- These imply:
  - At each step, our choice doesn't rule out an optimal tree.

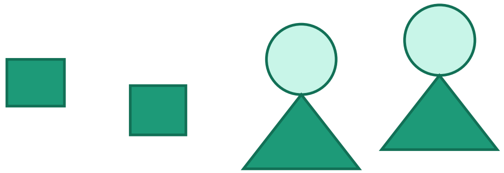
Write this out formally as a proof by induction! (See skipped slides for a starting point).

Siggi the Studious Stork

THIS SLIDE SKIPPED IN CLASS

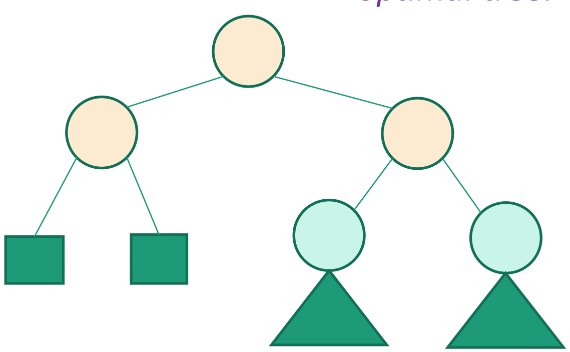
The whole argument

After the t'th step, we've got a bunch of current sub-trees:



- Inductive hypothesis:
  - after the t'th step,
    - there is an optimal tree containing the current subtrees as "leaves"
- Base case:
  - after the 0'th step,
    - there is an optimal tree containing all the characters.
- Inductive step:
  - **TO DO**
- Conclusion:
  - after the last step,
    - there is an optimal tree containing this whole tree as a subtree.
  - **aka,**
    - after the last step the tree we've constructed is optimal.

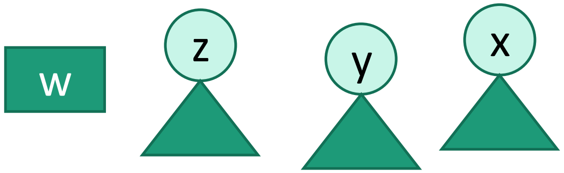
Inductive hyp. asserts that our subtrees can be assembled into an optimal tree:



THIS SLIDE SKIPPED IN CLASS

Inductive step

We've got a bunch of current sub-trees:



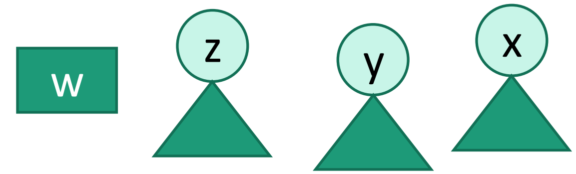
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."
- Want to show:
  - After t steps, there is an optimal tree containing all the current sub-trees as leaves.

THIS SLIDE SKIPPED IN CLASS

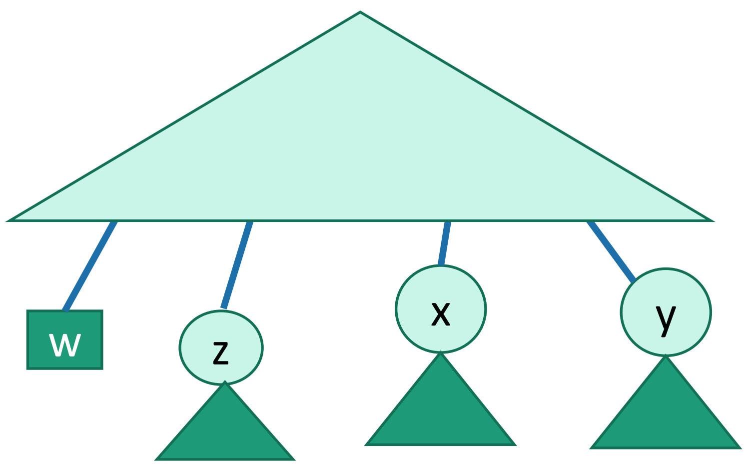
Inductive step

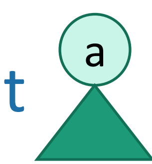
We've got a bunch of current sub-trees:



say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



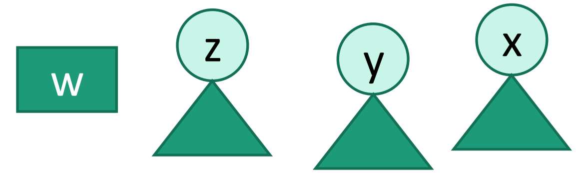
- By Lemma 2, may as well treat 

as 

THIS SLIDE SKIPPED IN CLASS

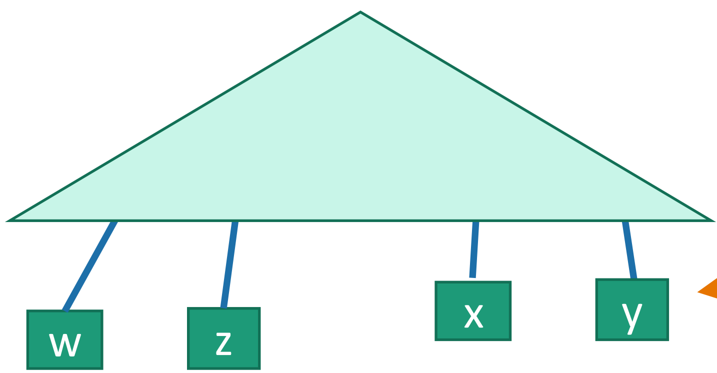
Inductive step

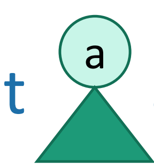
We've got a bunch of current sub-trees:



say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."

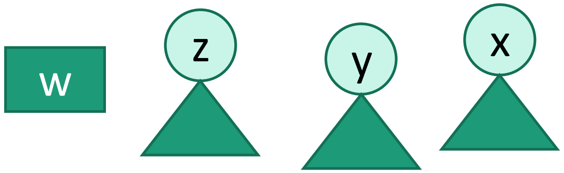


- By Lemma 2, may as well treat  as 
  - In particular, optimal trees on this new alphabet correspond to optimal trees on the original alphabet.

THIS SLIDE SKIPPED IN CLASS

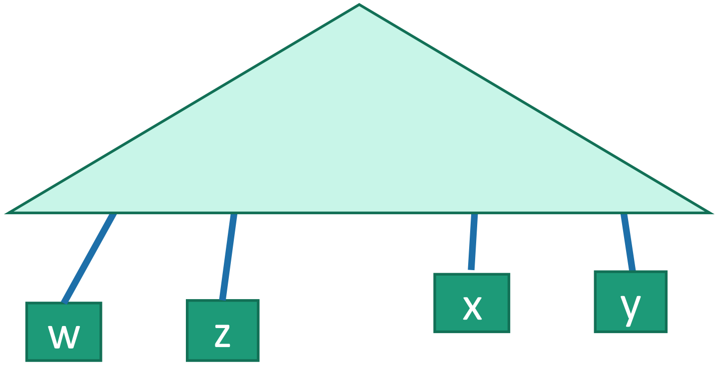
Inductive step

We've got a bunch of current sub-trees:

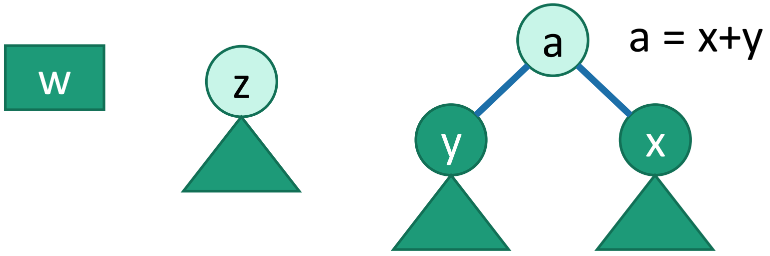


say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



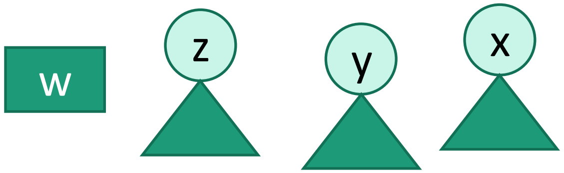
- Our algorithm would do this at level t:



THIS SLIDE SKIPPED IN CLASS

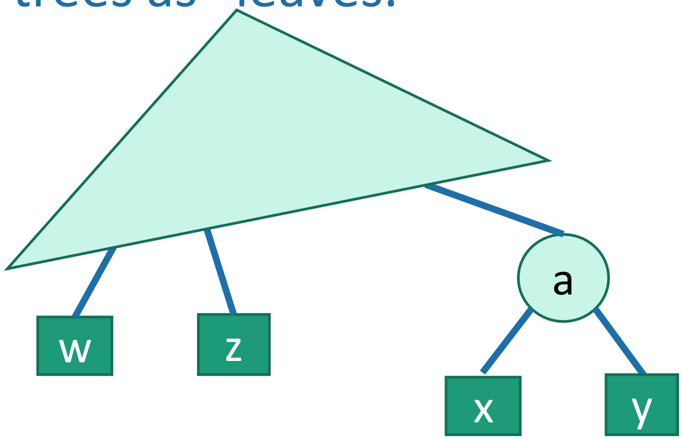
Inductive step

We've got a bunch of current sub-trees:



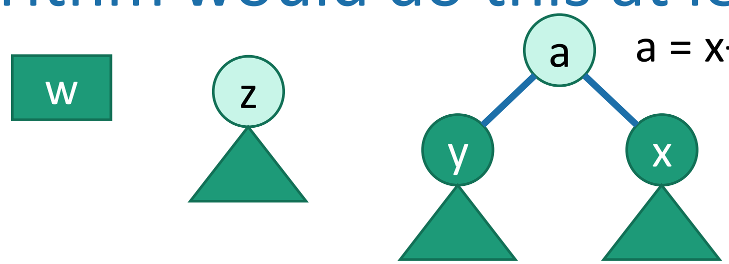
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



Lemma 1 implies that there's an optimal sub-tree that looks like this; aka, what our algorithm did okay.

- Our algorithm would do this at level t:

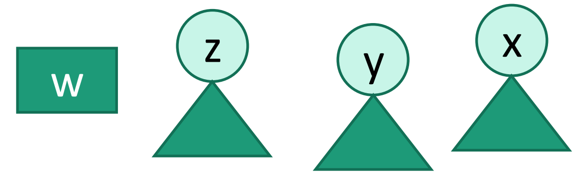


a = x+y

THIS SLIDE SKIPPED IN CLASS

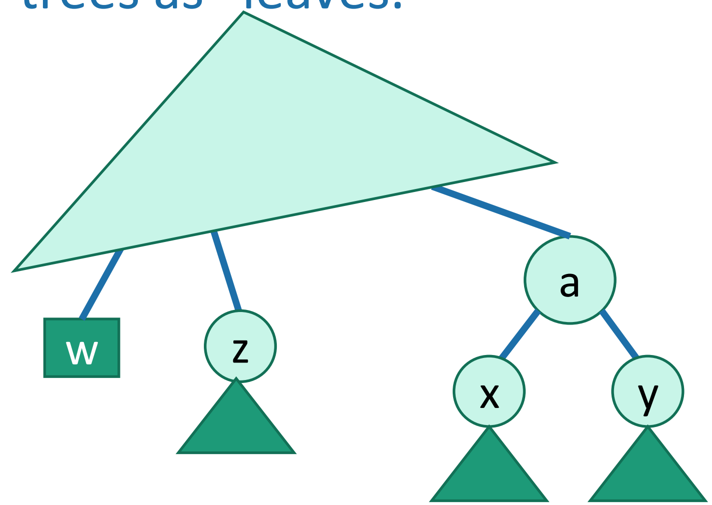
Inductive step

We've got a bunch of current sub-trees:



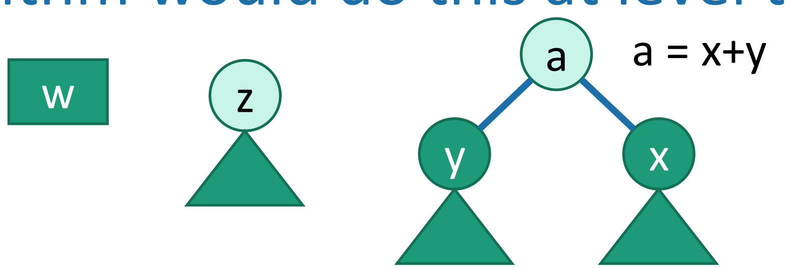
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



Lemma 2 again says that there's an optimal tree that looks like this

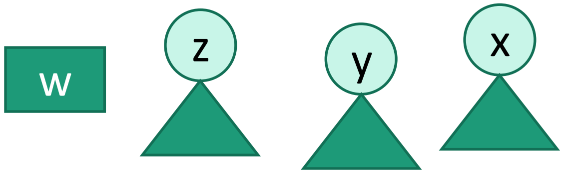
- Our algorithm would do this at level t:



THIS SLIDE SKIPPED IN CLASS

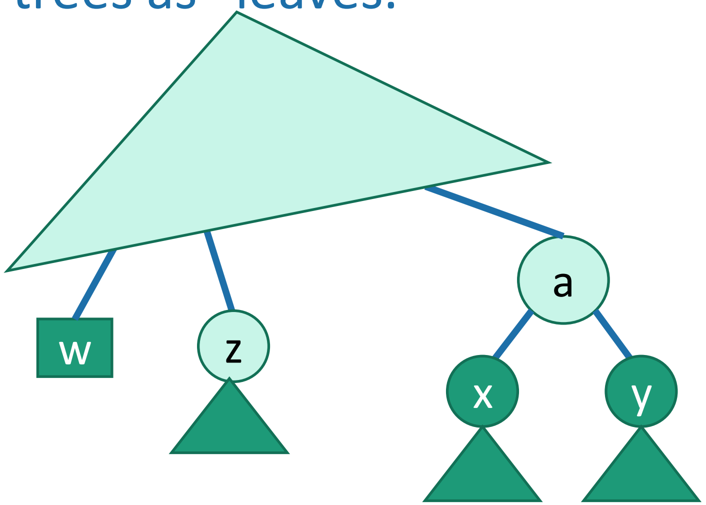
Inductive step

We've got a bunch of current sub-trees:



say that x and y are the two smallest.

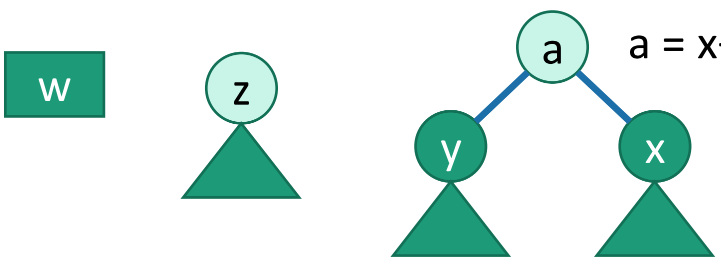
- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



Lemma 2 again says that there's an optimal tree that looks like this

aka, there is an optimal tree containing all the level-t sub-trees as "leaves".

- Our algorithm would do this at level t:



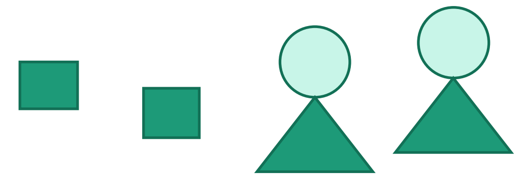
a = x+y

This is what we wanted to show for the inductive step.

THIS SLIDE SKIPPED IN CLASS

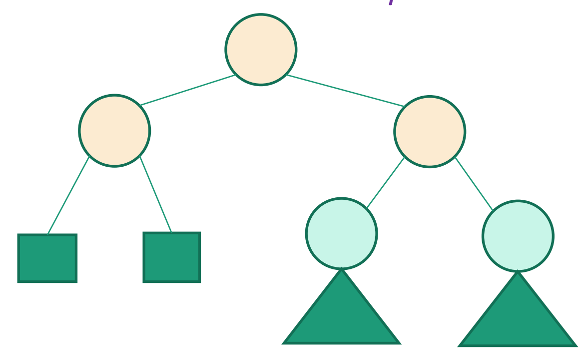
Inductive outline:

After the t'th step, we've got a bunch of current sub-trees:



- Inductive hypothesis:
  - after the t'th step,
    - there is an optimal tree containing the current subtrees as "leaves"
- Base case:
  - after the 0'th step,
    - there is an optimal tree containing all the vertices.
- Inductive step:
  - **TO DO**
- Conclusion:
  - after the last step,
    - there is an optimal tree containing this whole tree as a subtree.
  - **aka,**
    - after the last step the tree we've constructed is optimal.

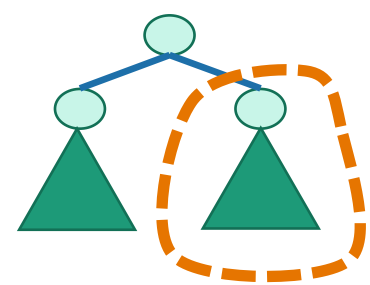
Inductive hyp. asserts that our subtrees can be assembled into an optimal tree:



*If all we care about is number of bits.

What have we learned?

- ASCII isn't an optimal way* to encode English, since the distribution on letters isn't uniform.
- Huffman Coding is an optimal way!
- To come up with an optimal scheme for any language efficiently, we can use a greedy algorithm.



- To come up with a greedy algorithm:
  - Identify **optimal substructure**
  - Find a way to make choices that **won't rule out an optimal solution.**
    - Create subtrees out of the smallest two current subtrees.

Recap I

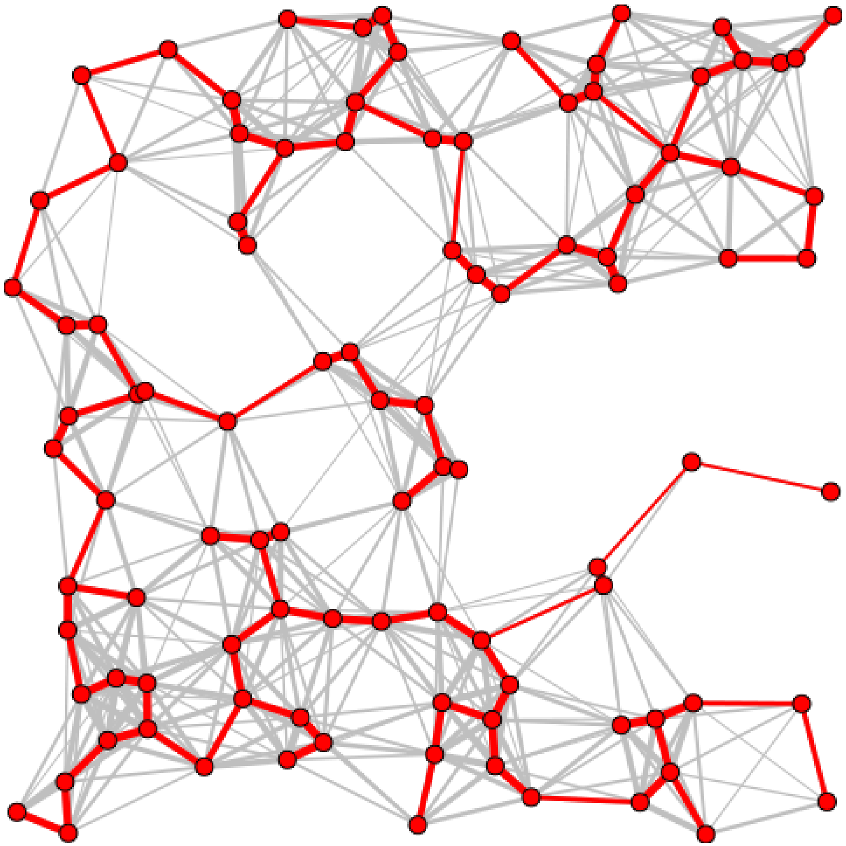
- Greedy algorithms!
- Three examples:
    - Activity Selection
    - Scheduling Jobs
    - Huffman Coding

Recap II

- Greedy algorithms!
- Often easy to write down
  - But may be hard to come up with and hard to justify
- The natural greedy algorithm may not always be correct.
- A problem is a good candidate for a greedy algorithm if:
  - it has optimal substructure
  - that optimal substructure is **REALLY NICE**
    - solutions depend on just one other sub-problem.

Next part

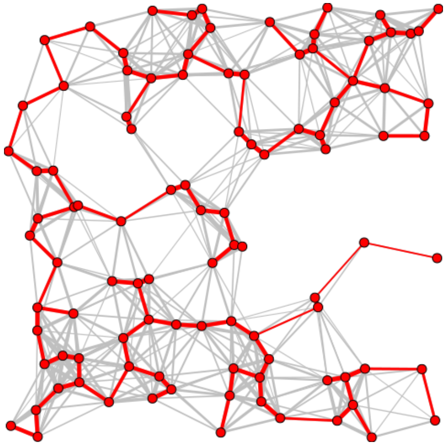
- Greedy algorithms for Minimum Spanning Tree!



Today (part 2)

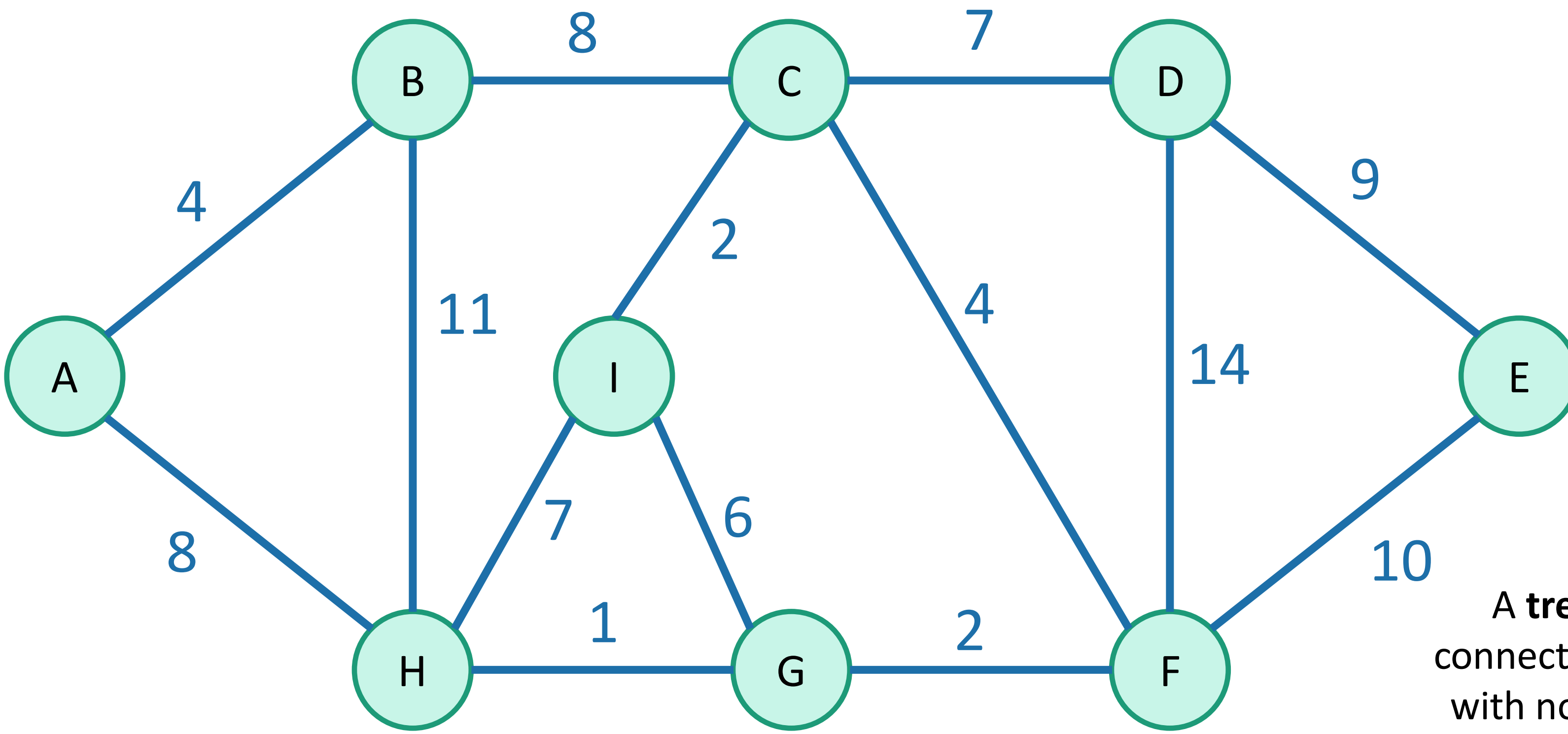
- Greedy algorithms for Minimum Spanning Tree

- Agenda:
    1. What is a Minimum Spanning Tree?
    2. Short break to introduce some graph theory tools
    3. Prim's algorithm
    4. Kruskal's algorithm



Minimum Spanning Tree

Say we have an undirected weighted graph



A **tree** is a connected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

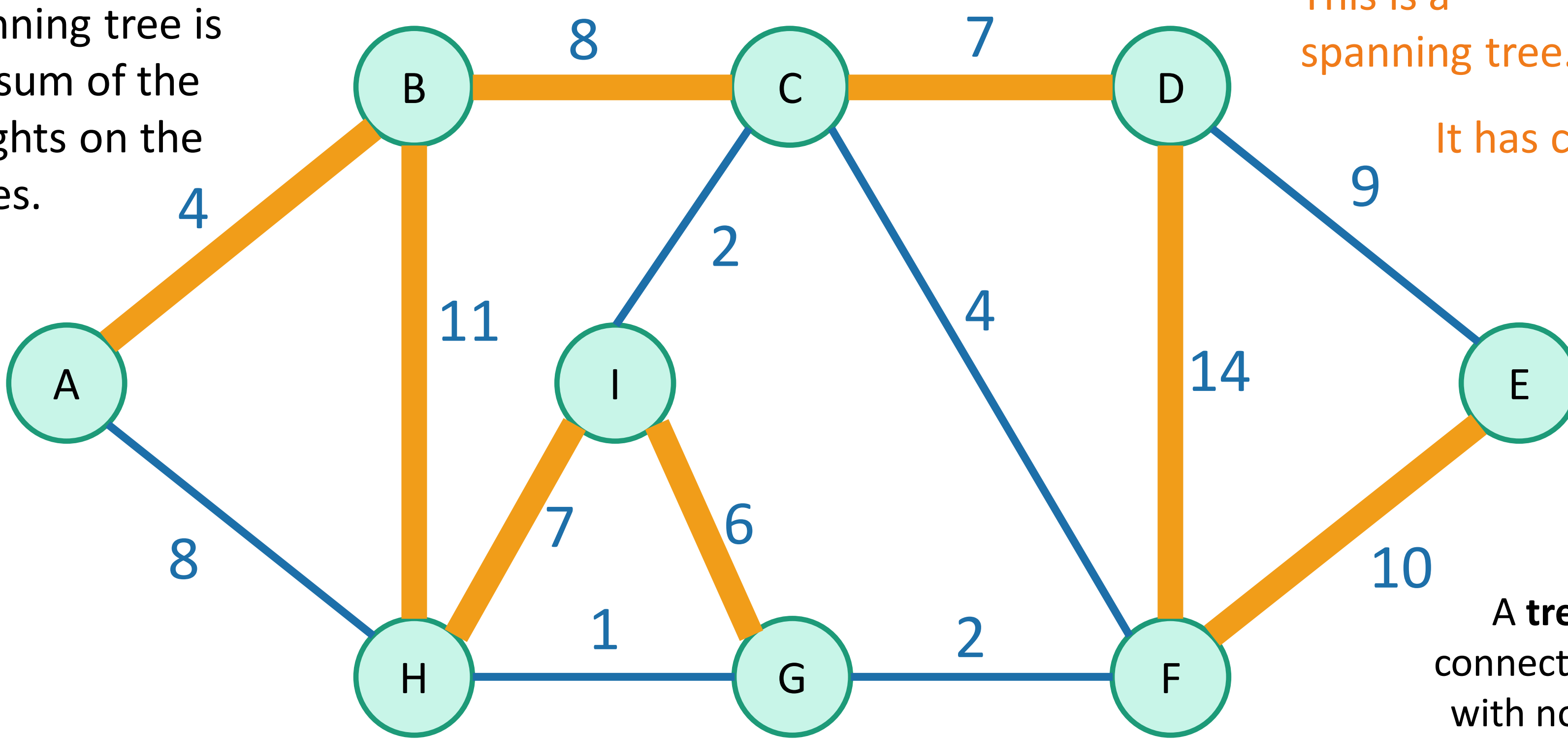
Minimum Spanning Tree

Say we have an undirected weighted graph

The **cost** of a spanning tree is the sum of the weights on the edges.

This is a spanning tree.

It has cost 67

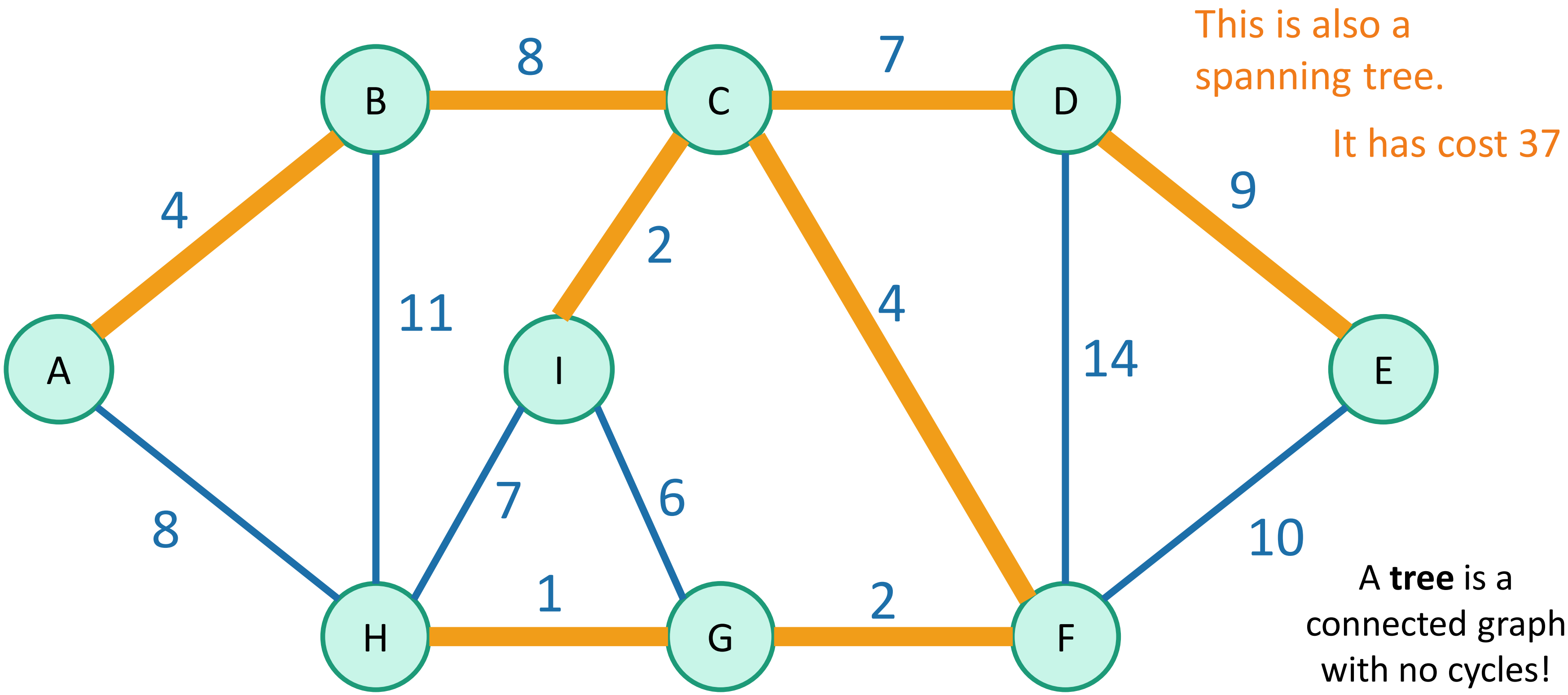


A **tree** is a connected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph

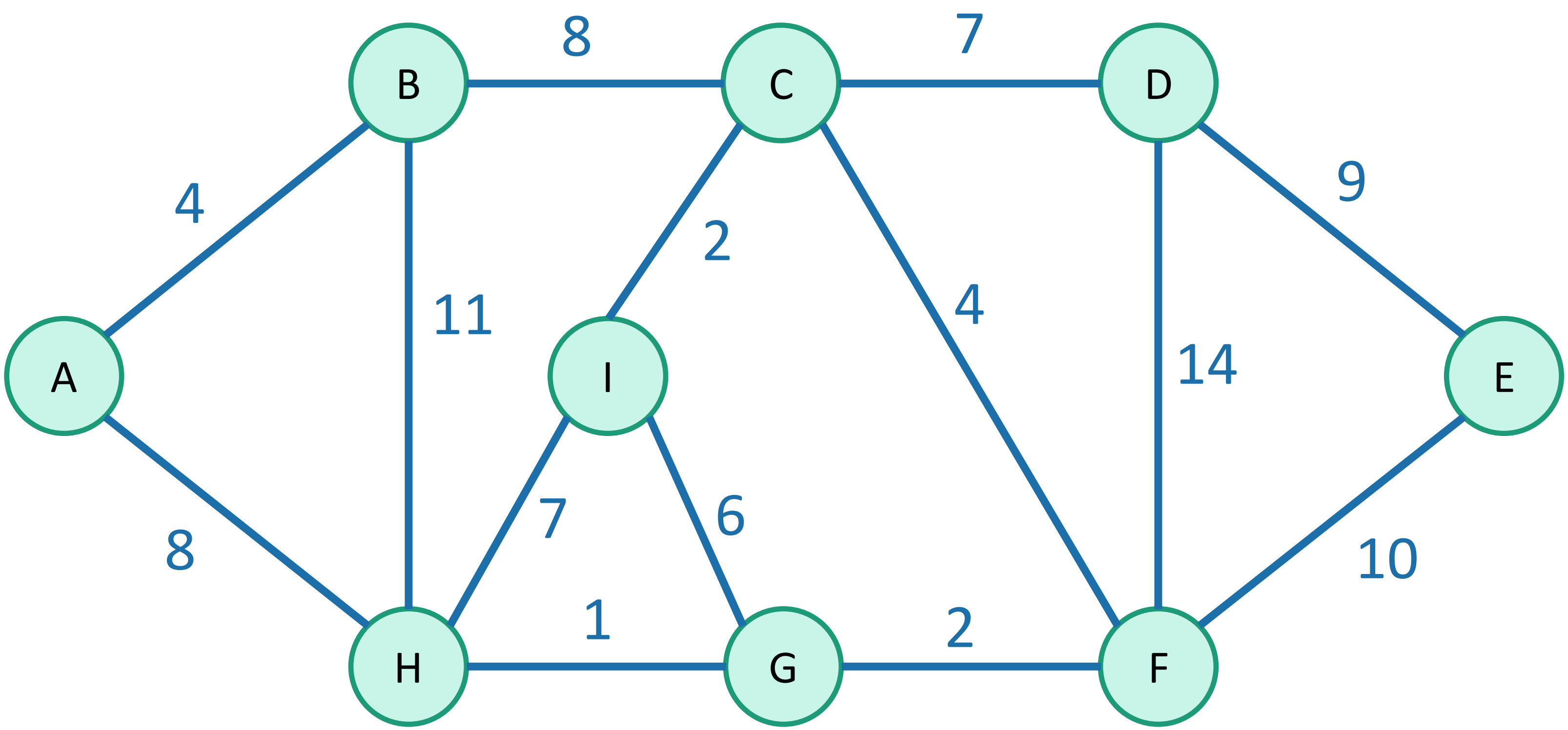


A **tree** is a connected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph



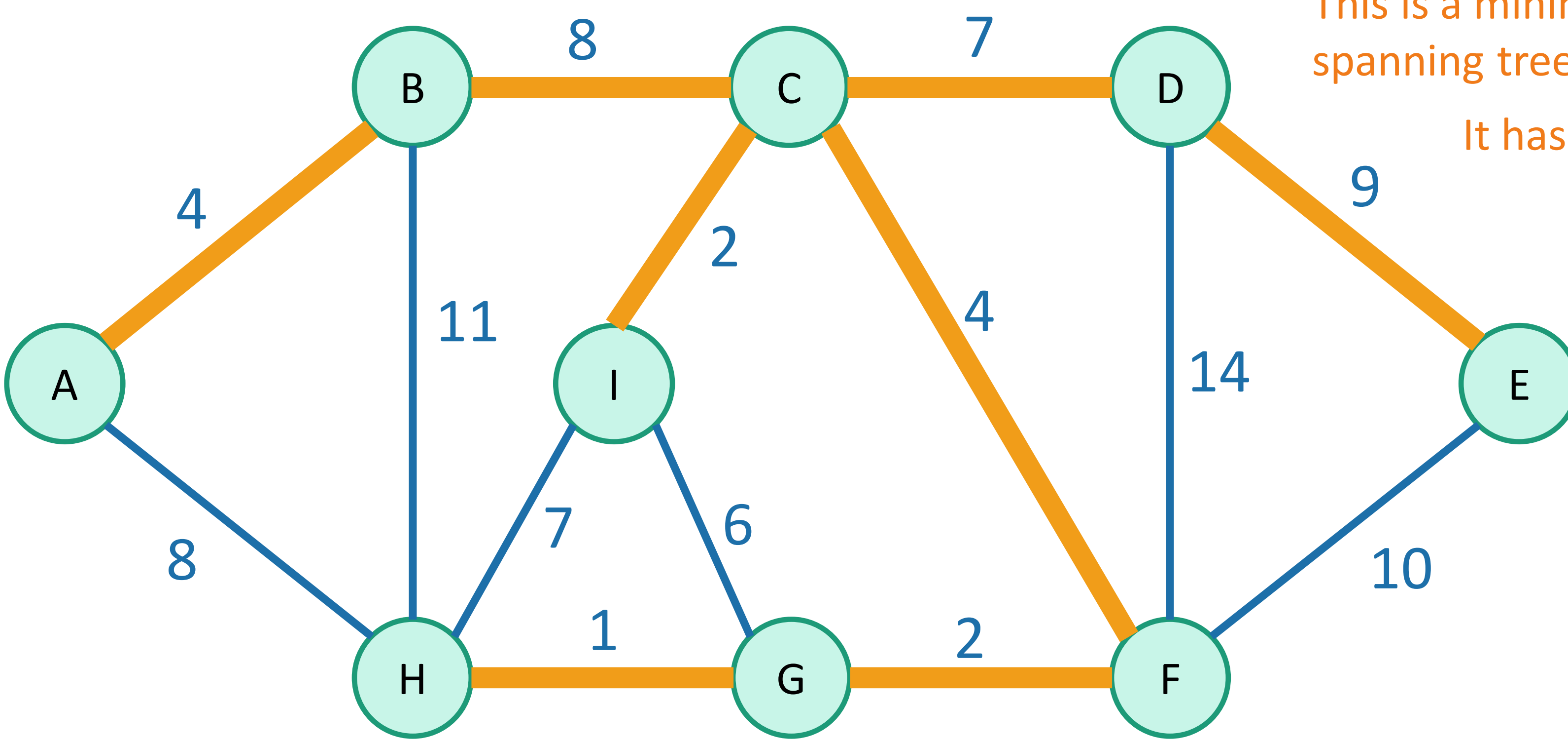
minimum

of minimal cost

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph



This is a minimum spanning tree.

It has cost 37

minimum

of minimal cost

A **spanning tree** is a **tree** that connects all of the vertices.

Why MSTs?

- Network design
  - Connecting cities with roads/electricity/telephone/…
- Cluster analysis
  - eg, genetic distance
- Image processing
  - eg, image segmentation
- Useful primitive
  - for other graph algs

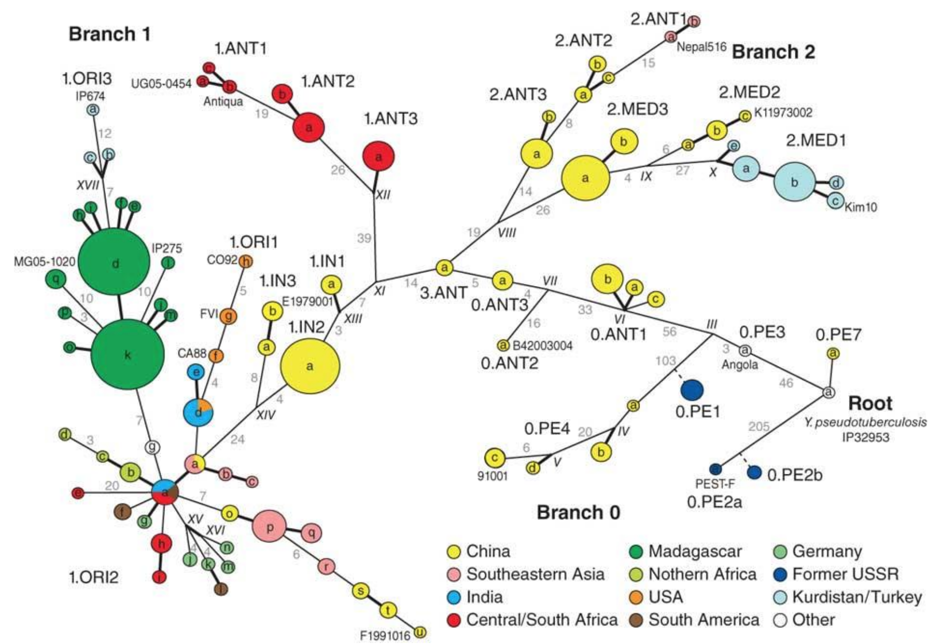
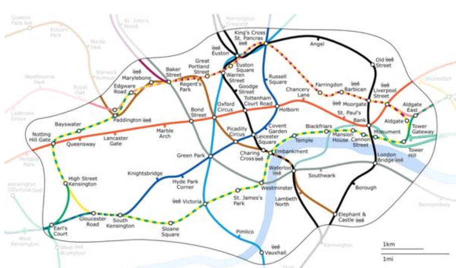


Figure 2: Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of *Y. pestis* colored by location.

Morelli et al. Nature Genetics 2010



How to find an MST?

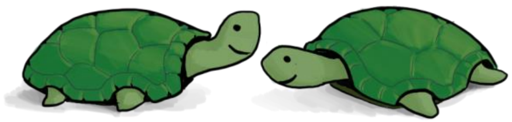
- Today we'll see two greedy algorithms.
- In order to prove that these greedy algorithms work, we'll need to show something like:

Suppose that our choices so far ***haven't ruled out success****.*

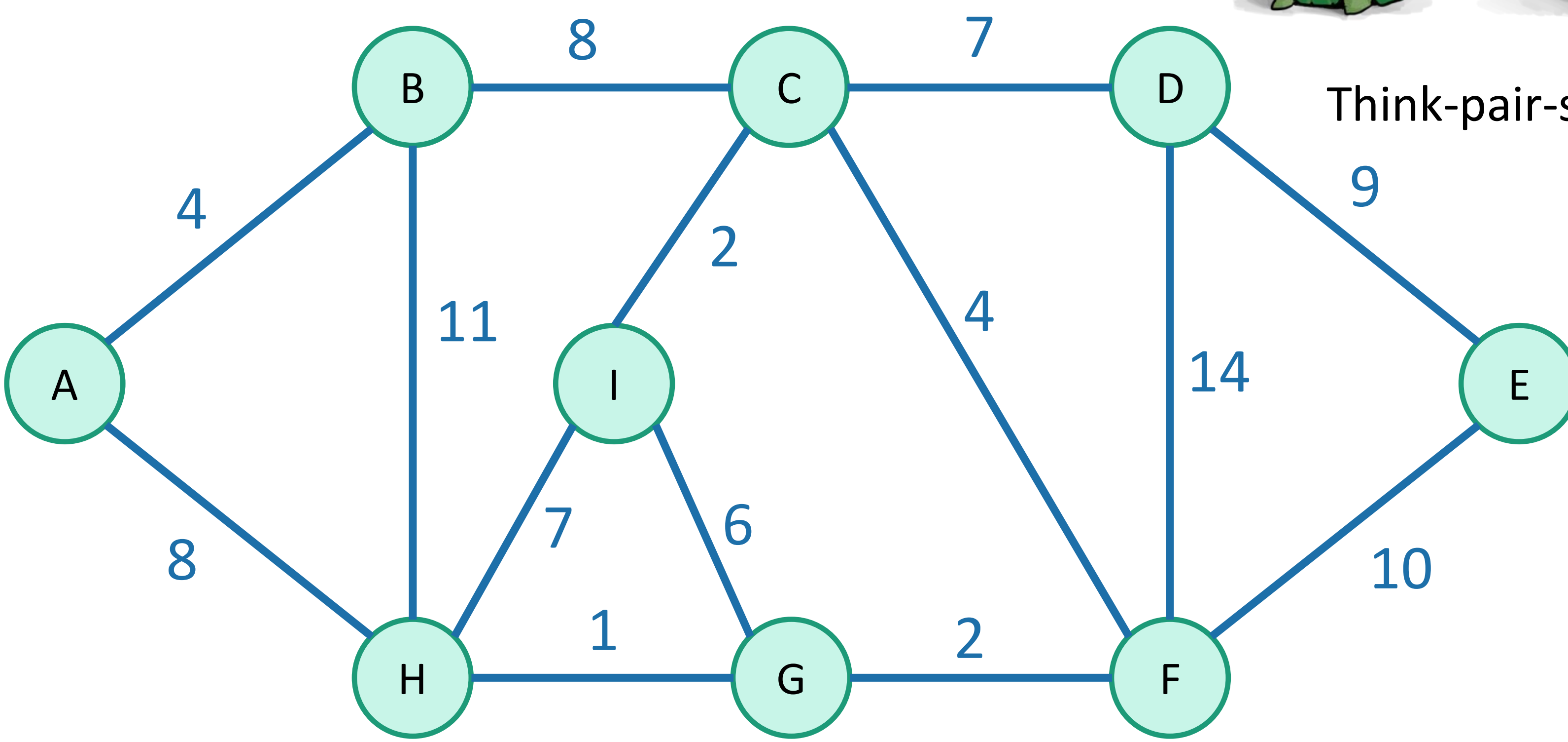
Then the next greedy choice that we make ***also won't rule out success****.*

- Here, **success** means finding an MST.

Let’s brainstorm some greedy algorithms!



Think-pair-share!

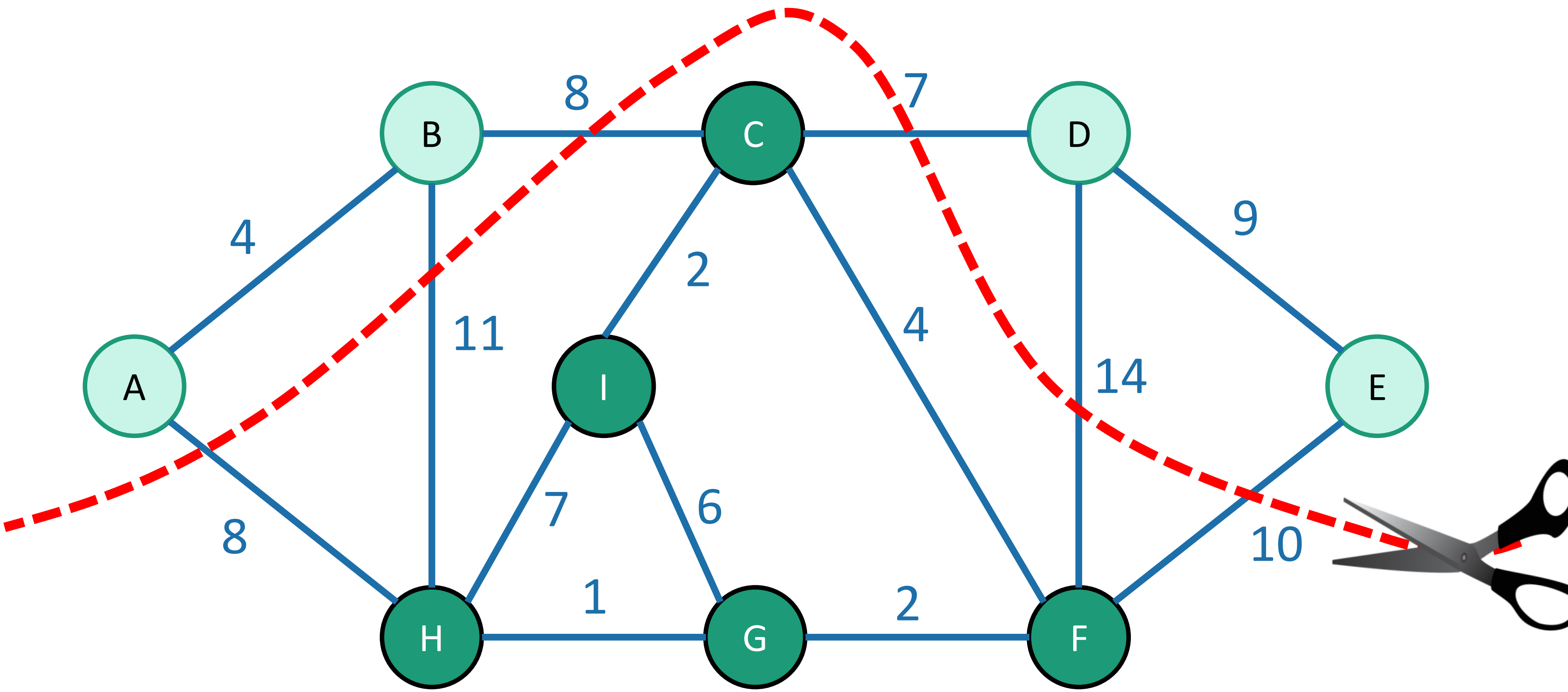


Brief aside

for a discussion of cuts in graphs!

Cuts in graphs

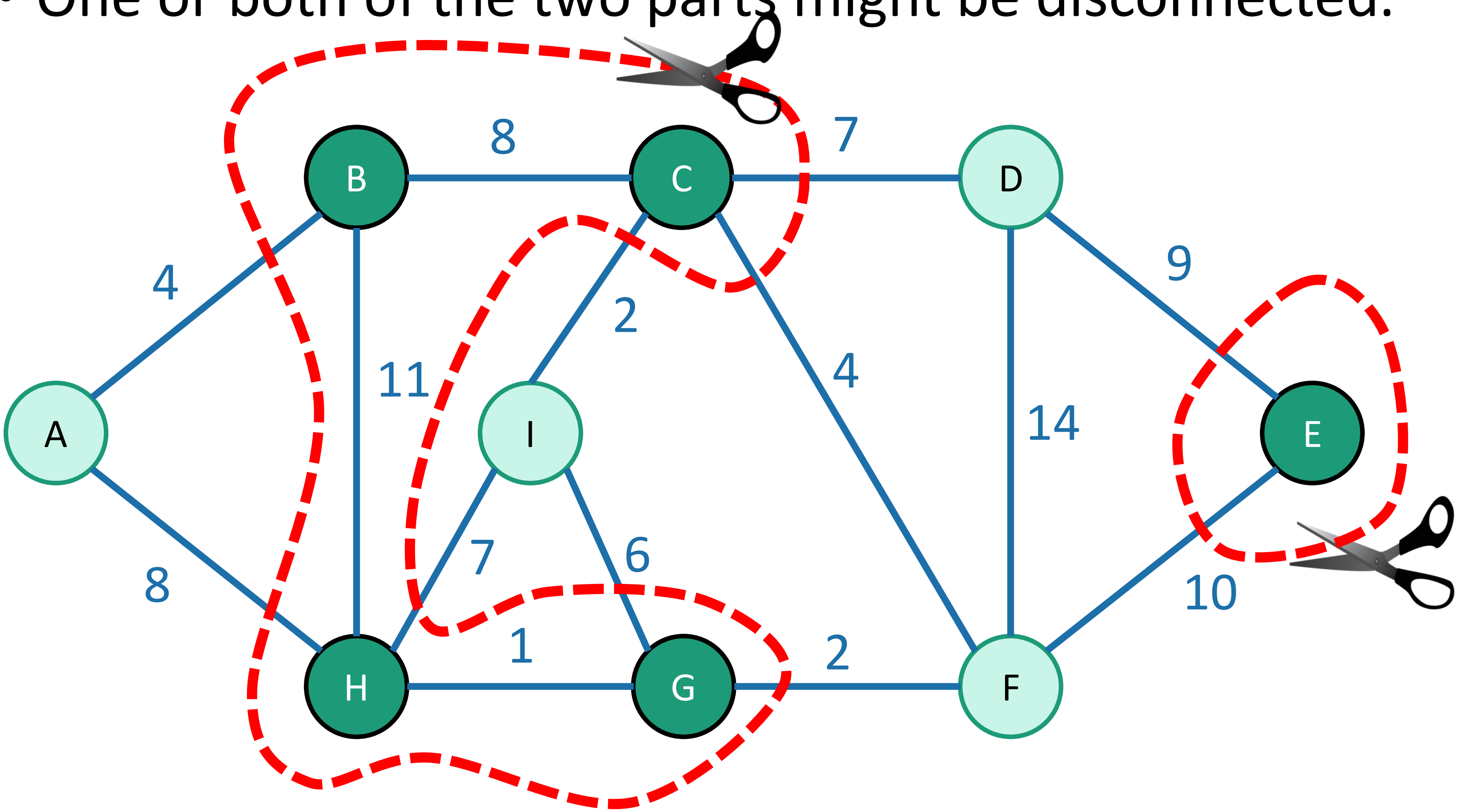
- A cut is a partition of the vertices into two parts:



This is the cut **"{A,B,D,E} and {C,I,H,G,F}"**

Cuts in graphs

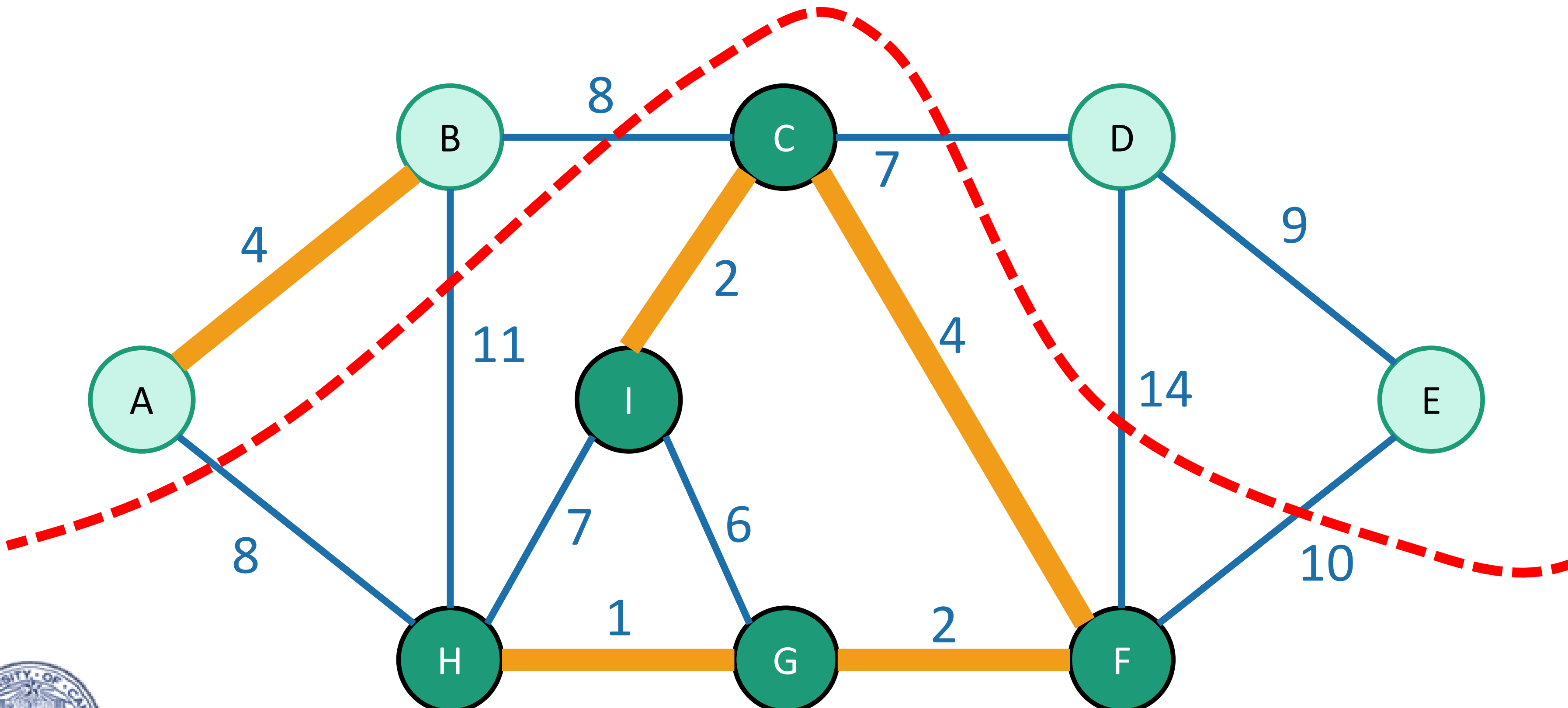
- One or both of the two parts might be disconnected.



This is the cut **"{B,C,E,G,H} and {A,D,I,F}"**

Let S be a set of edges in G

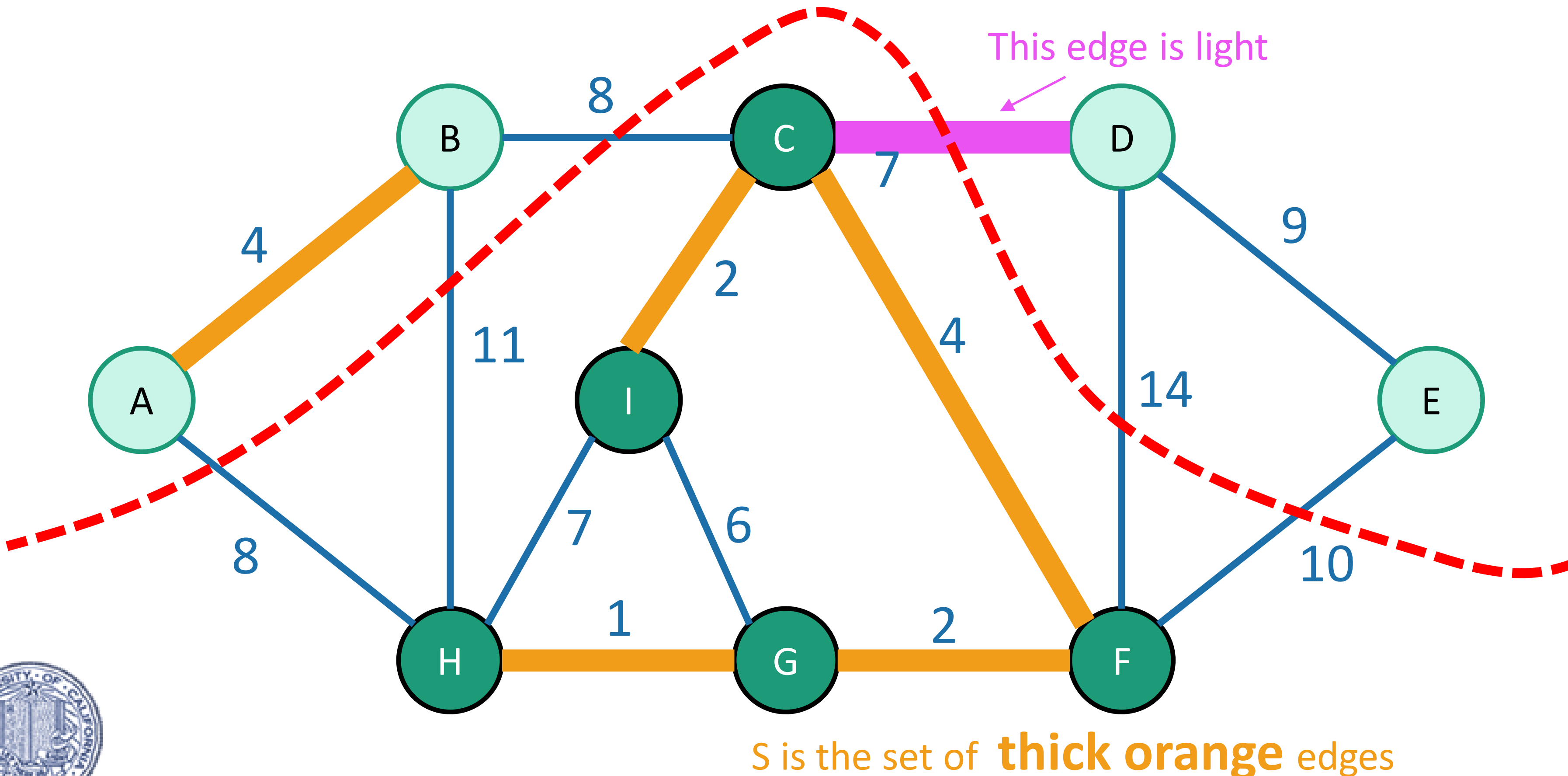
- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



S is the set of **thick orange** edges

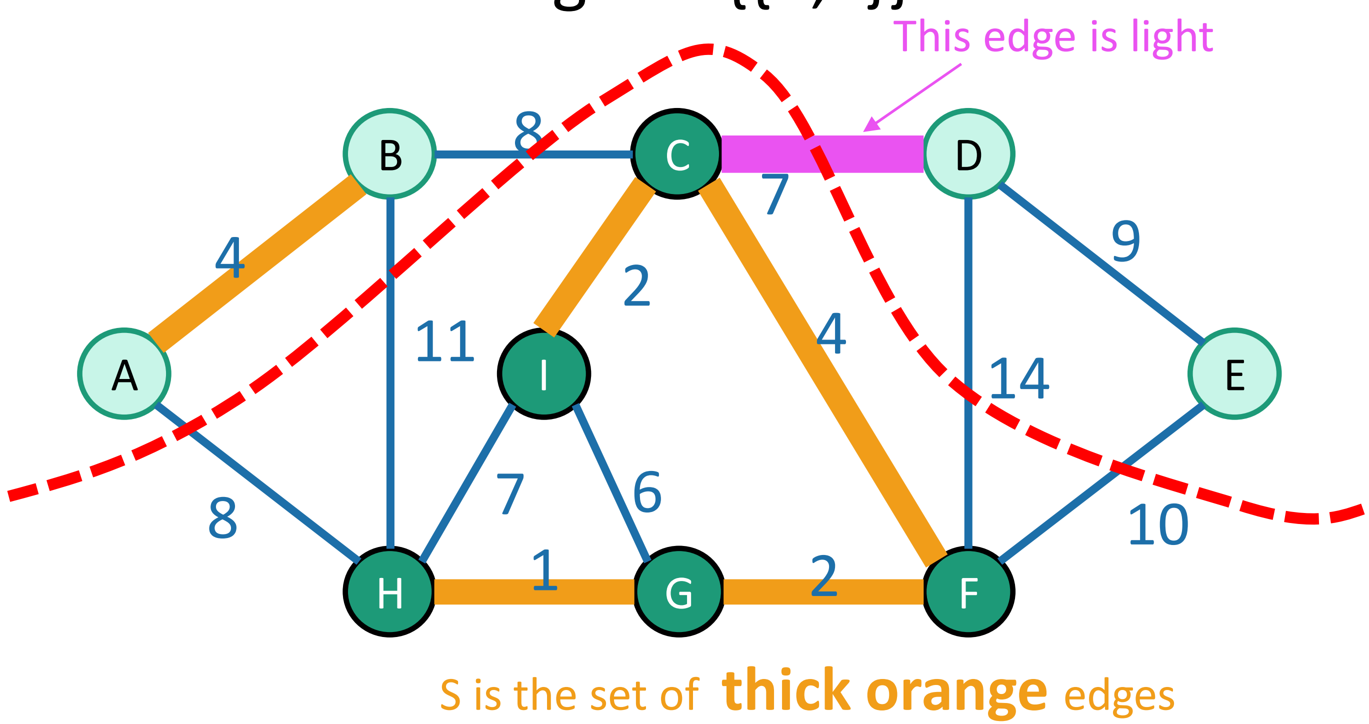
Let S be a set of edges in G

- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let {u,v} be a light edge.
- Then there is an MST containing S ∪ {{u,v}}

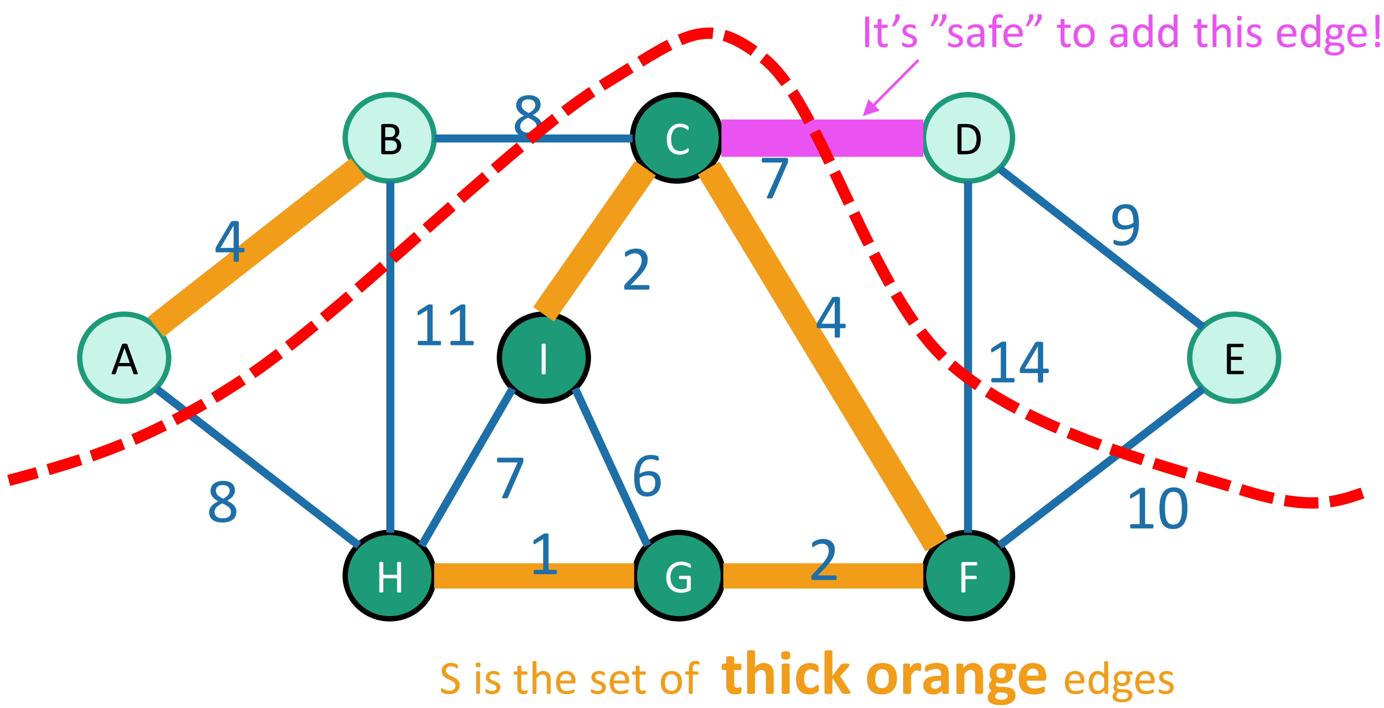


Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let {u,v} be a light edge.
- Then there is an MST containing S ∪ {{u,v}}

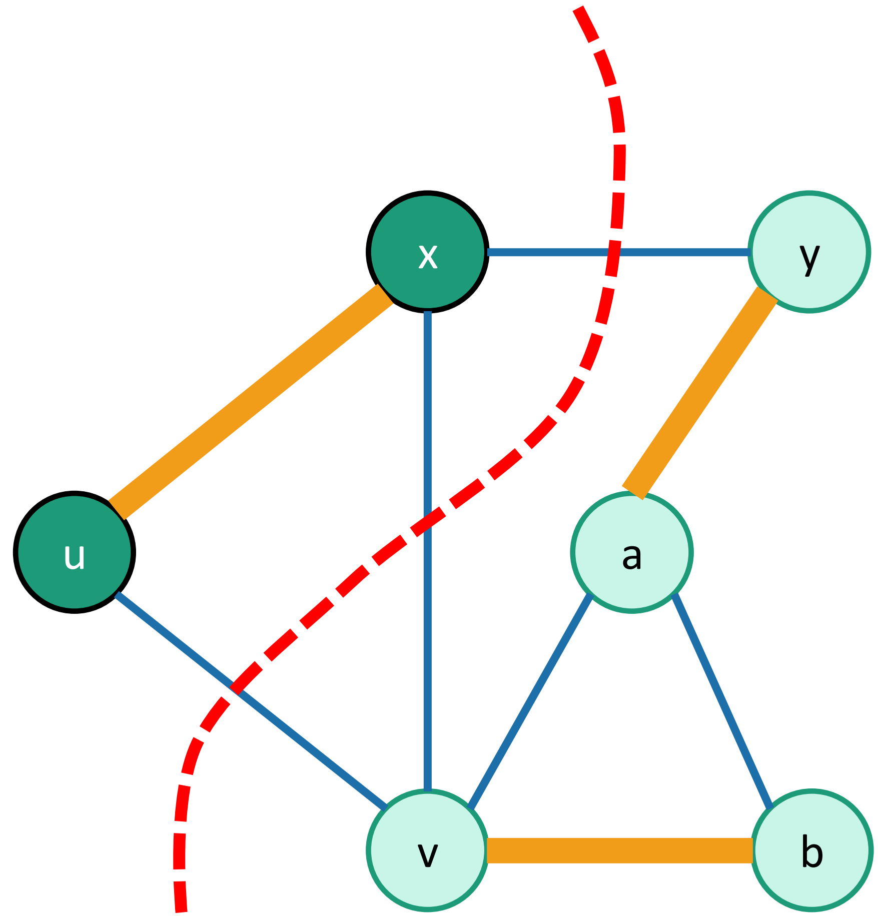
Aka:

If we haven't ruled out the possibility of success so far, then adding a light edge still won't rule it out.



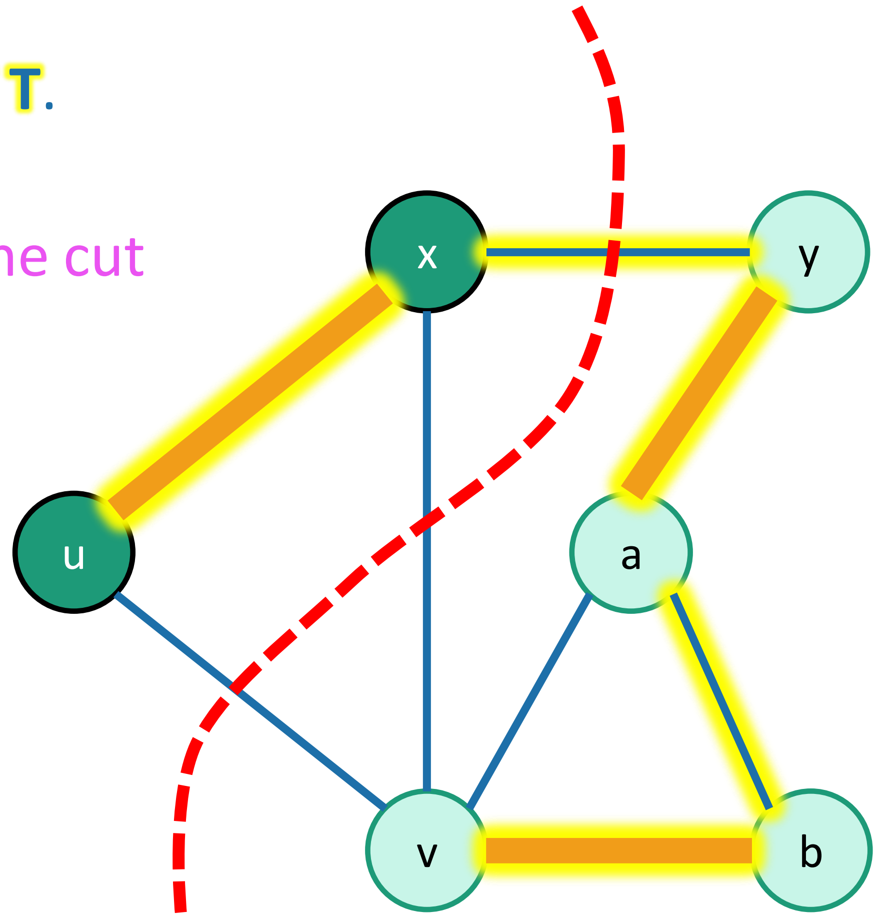
Proof of Lemma

- Assume that we have:
  - a **cut** that respects **S**



Proof of Lemma

- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut



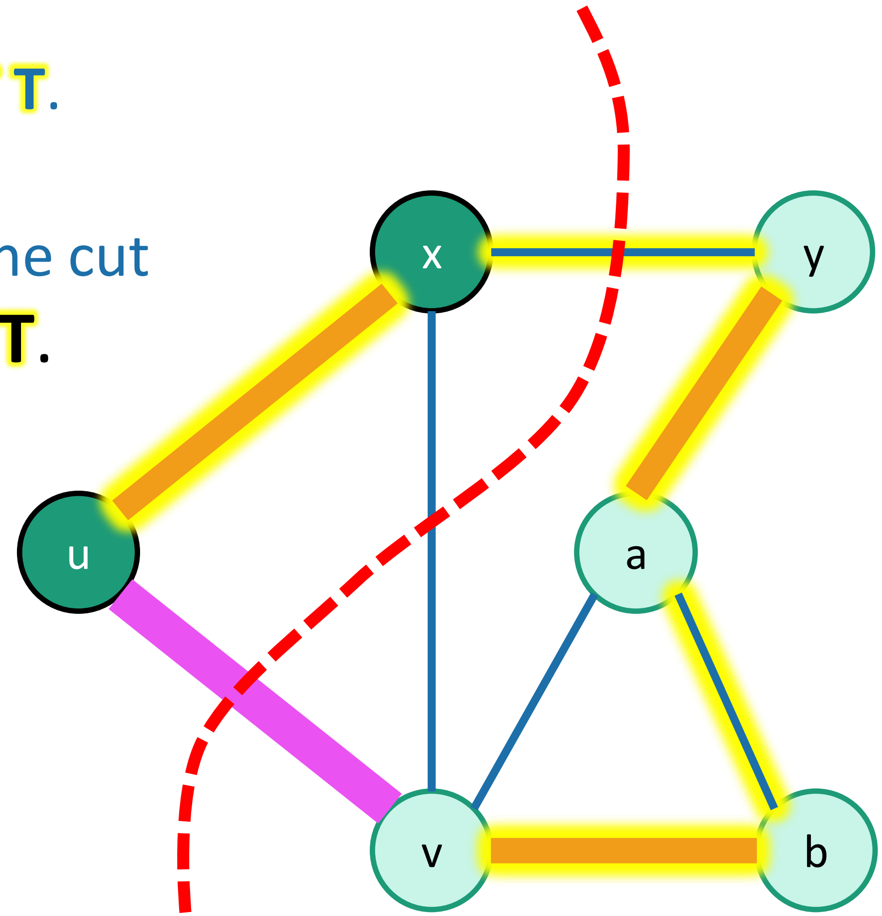
Claim: Adding any additional edge to a spanning tree will create a cycle.

Proof: Both endpoints are already in the tree and connected to each other.

Proof of Lemma

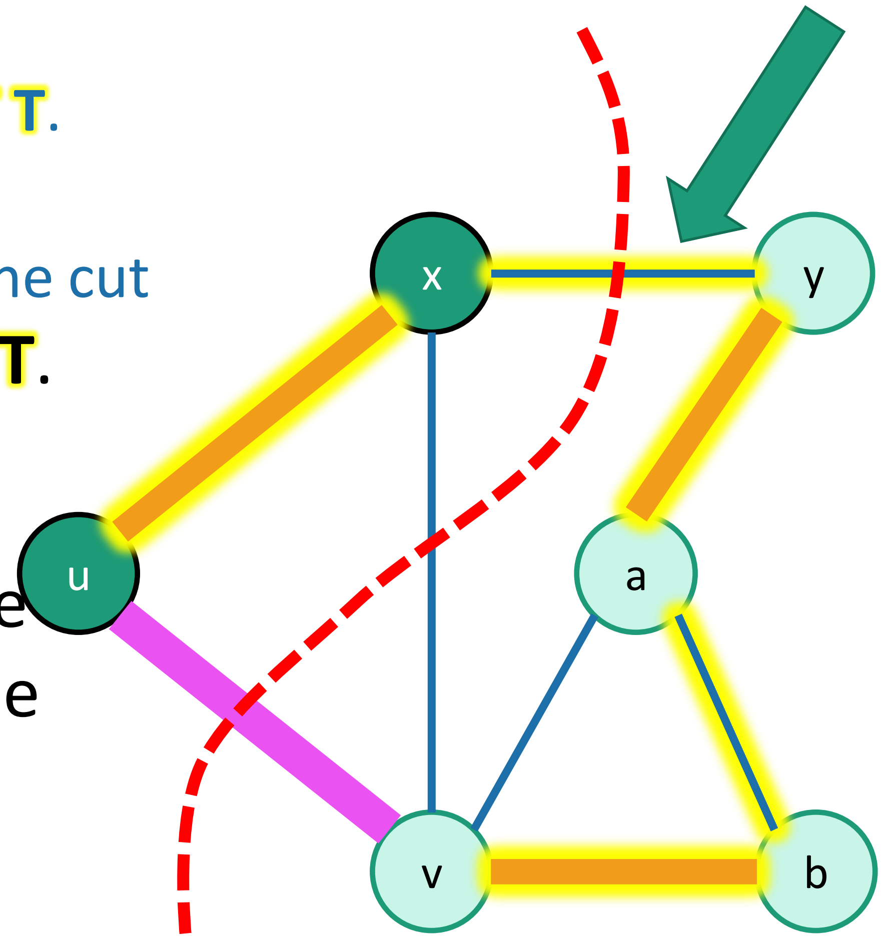
- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut
- But say **{u,v}** is not in **T**.
  - So adding **{u,v}** to **T** will make a cycle.

Otherwise we're done!



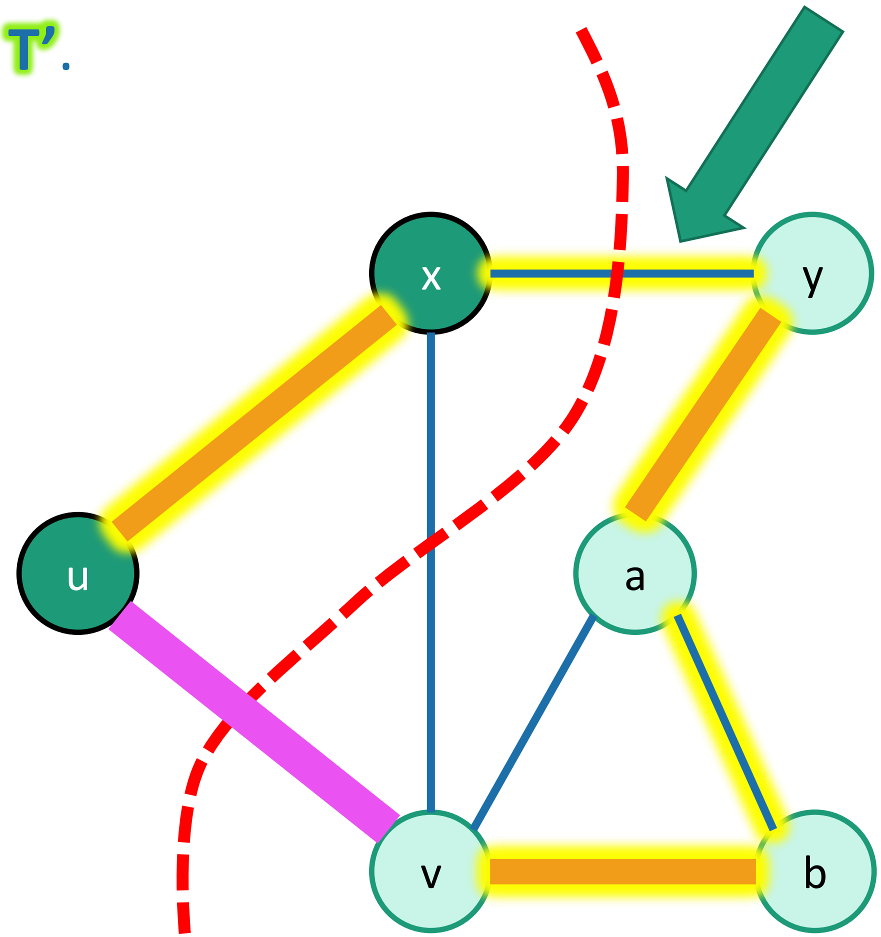
Proof of Lemma

- Assume that we have:
  - a **cut** that respects **S**
  - **S** is part of some **MST T**.
- Say that **{u,v}** is light.
  - lowest cost crossing the cut
- But say **{u,v}** is not in **T**.
  - So adding **{u,v}** to **T** will make a cycle.
- So there is at least one other edge in this cycle crossing the cut.
  - call it {x,y}



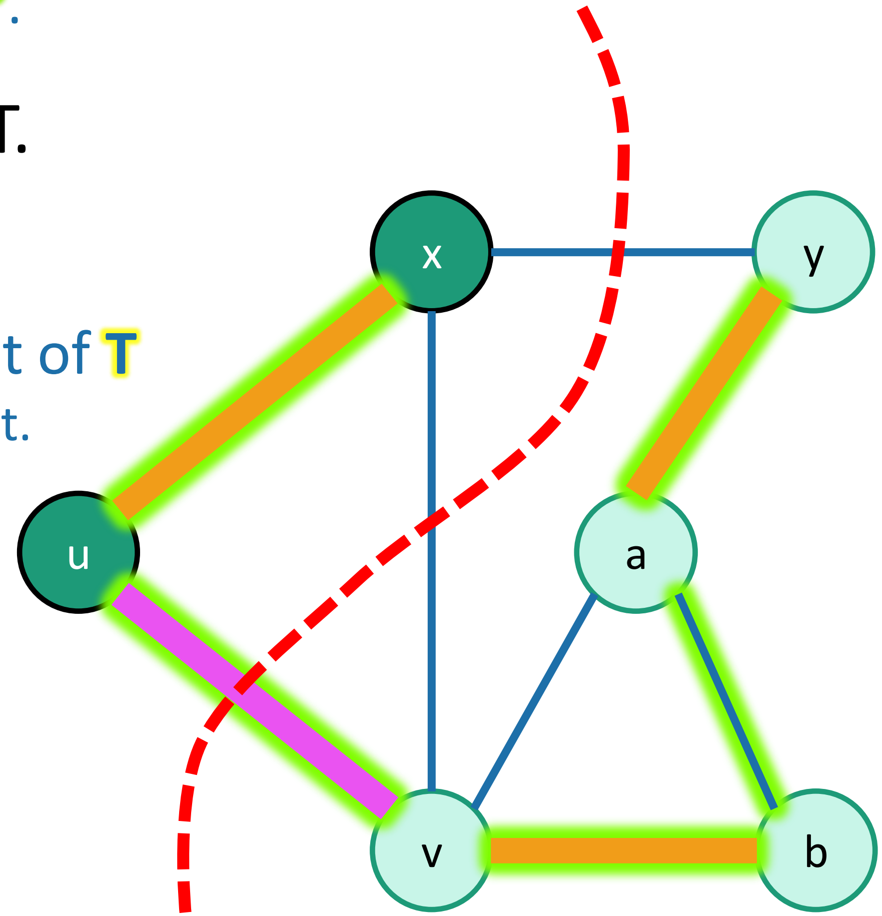
Proof of Lemma ctd.

- Consider swapping {u,v} for {x,y} in **T**.
  - Call the resulting tree **T'**.



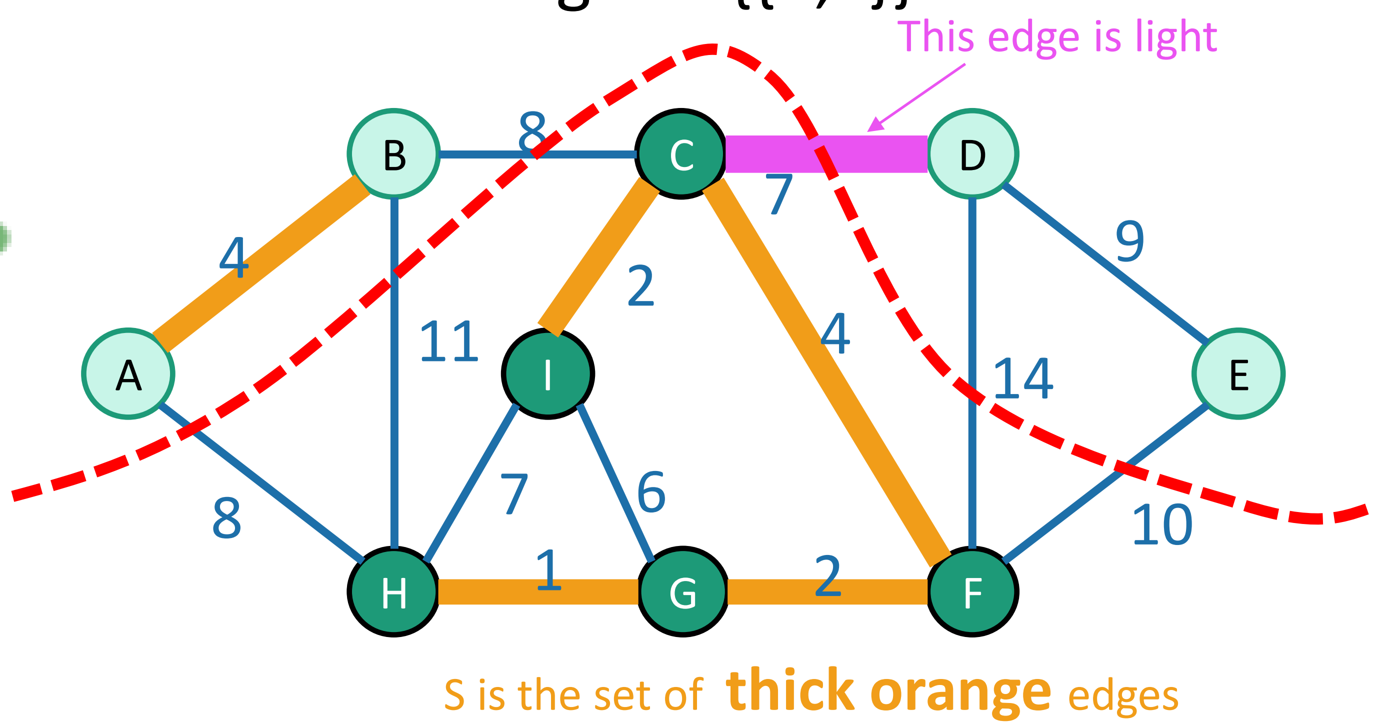
Proof of Lemma ctd.

- Consider swapping {u,v} for {x,y} in **T**.
  - Call the resulting tree **T'**.
- **Claim**: **T'** is still an MST.
  - It is still a tree:
    - we deleted {x,y}
  - It has cost at most that of **T**
    - because {u,v} was light.
  - **T** had minimal cost.
  - So **T'** does too.
- So **T'** is an MST containing S and {u,v}.
  - This is what we wanted.



Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let {u,v} be a light edge.
- Then there is an MST containing S ∪ {{u,v}}



End aside

Back to MSTs!

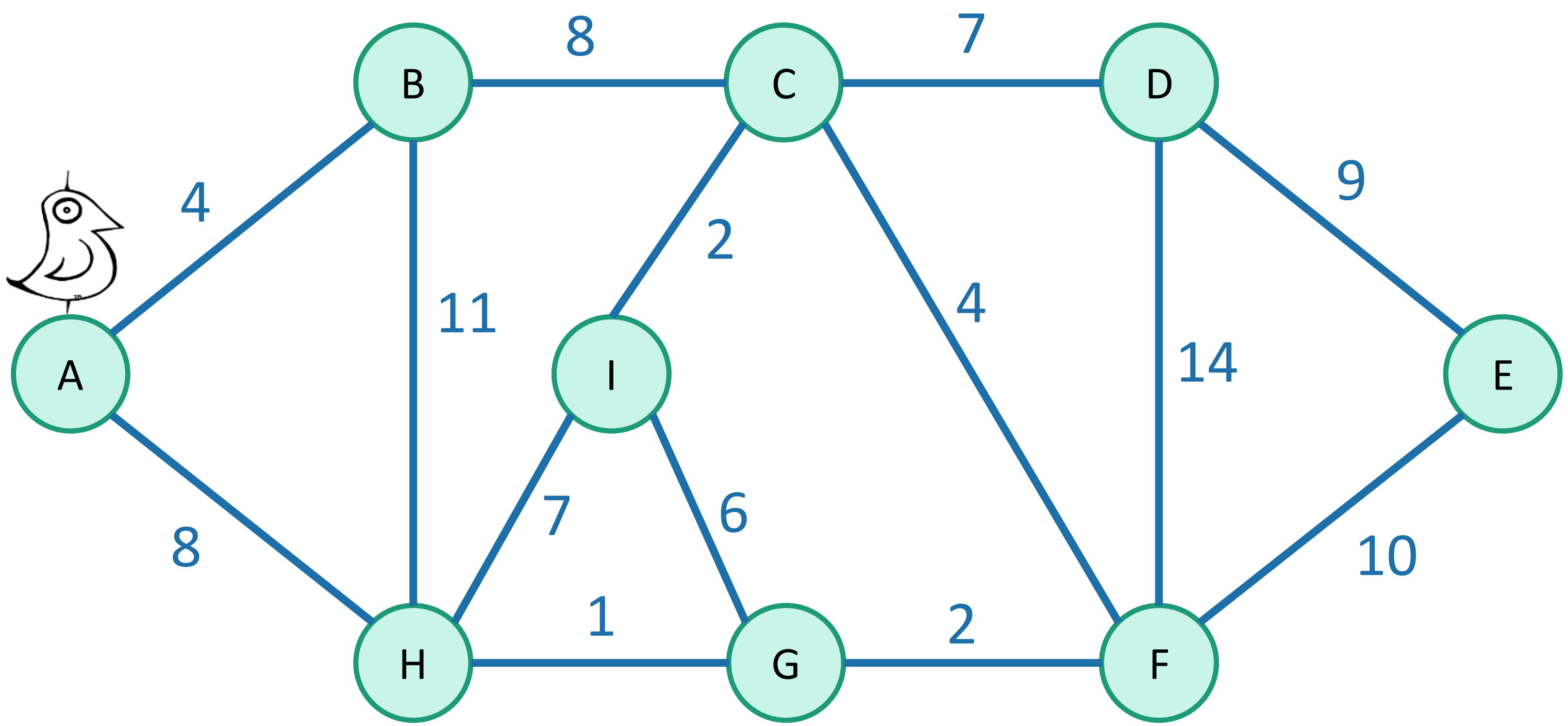
Back to MSTs

- How do we find one?
- Today we'll see **two greedy algorithms**.

- The strategy:
  - Make a **series of choices,** adding edges to the tree.
  - Show that each edge we add is **safe to add**:
    - we do not rule out the possibility of success
    - we will choose **light edges** crossing **cuts** and **use the Lemma**.
  - **Keep going** until we have an MST.

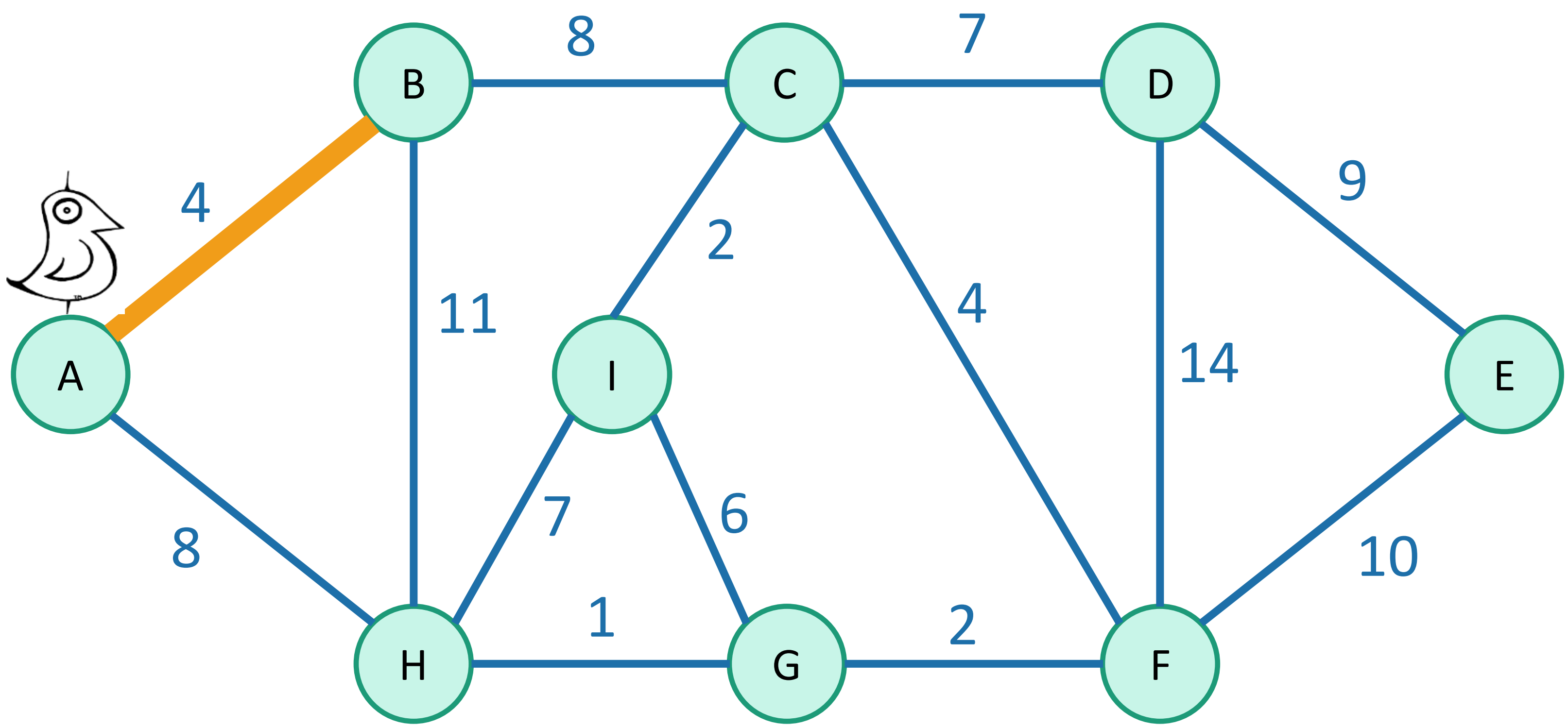
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



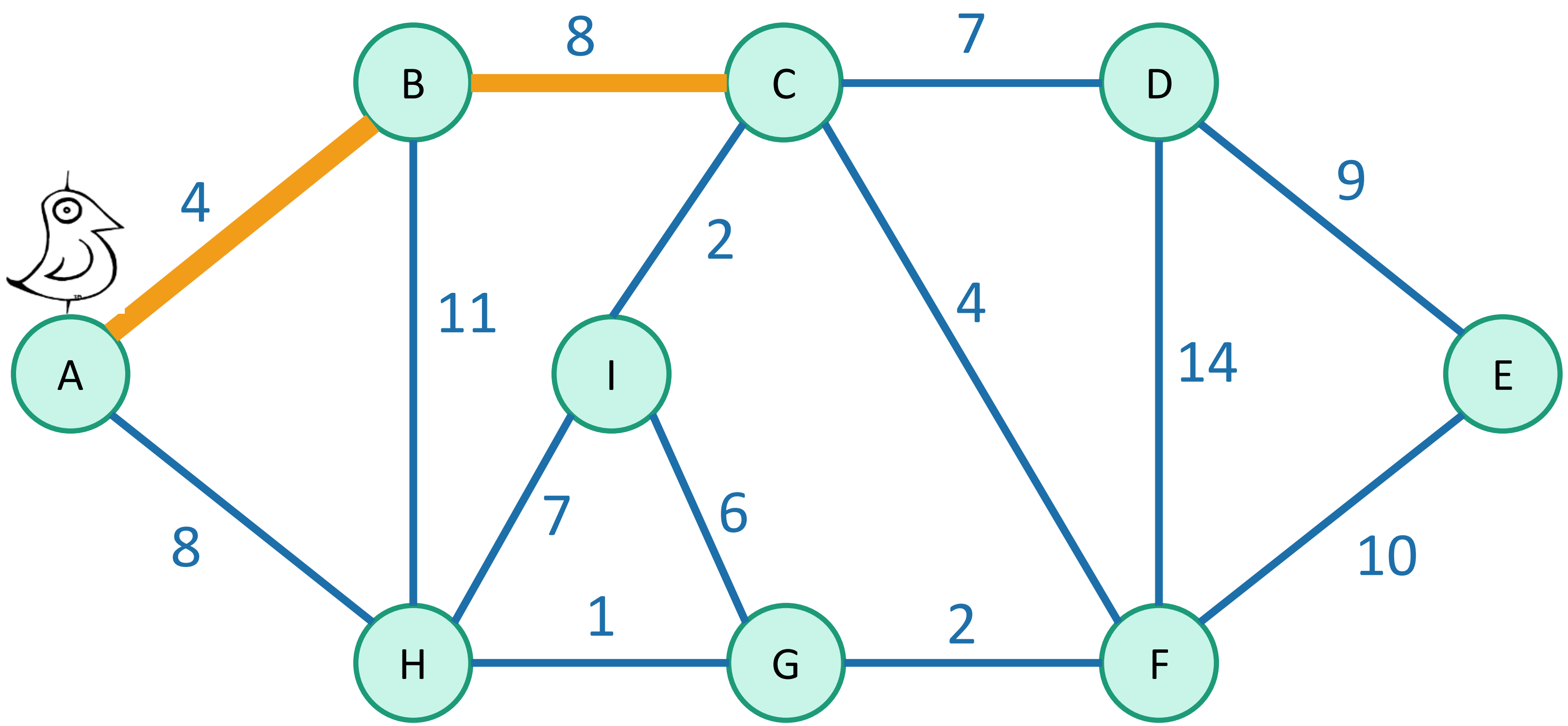
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



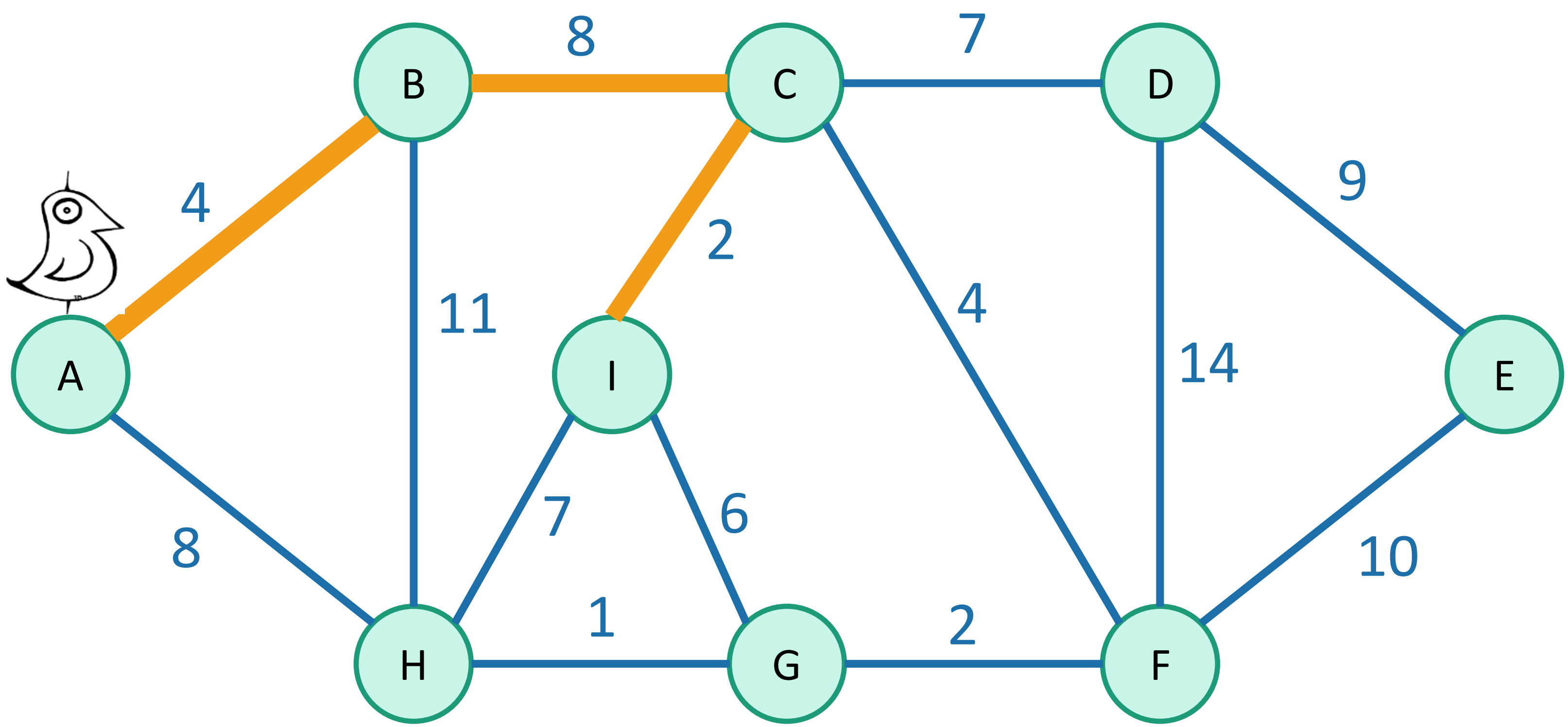
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



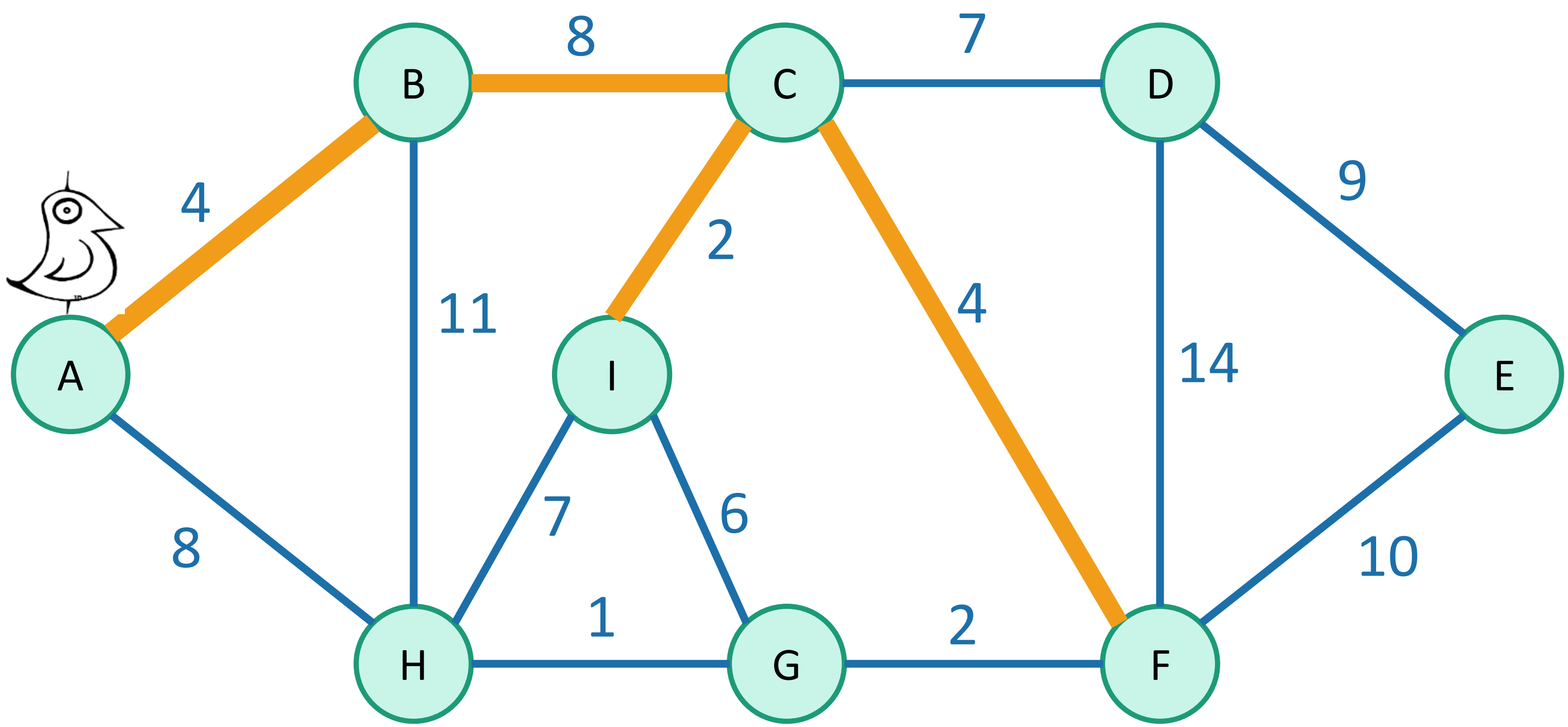
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



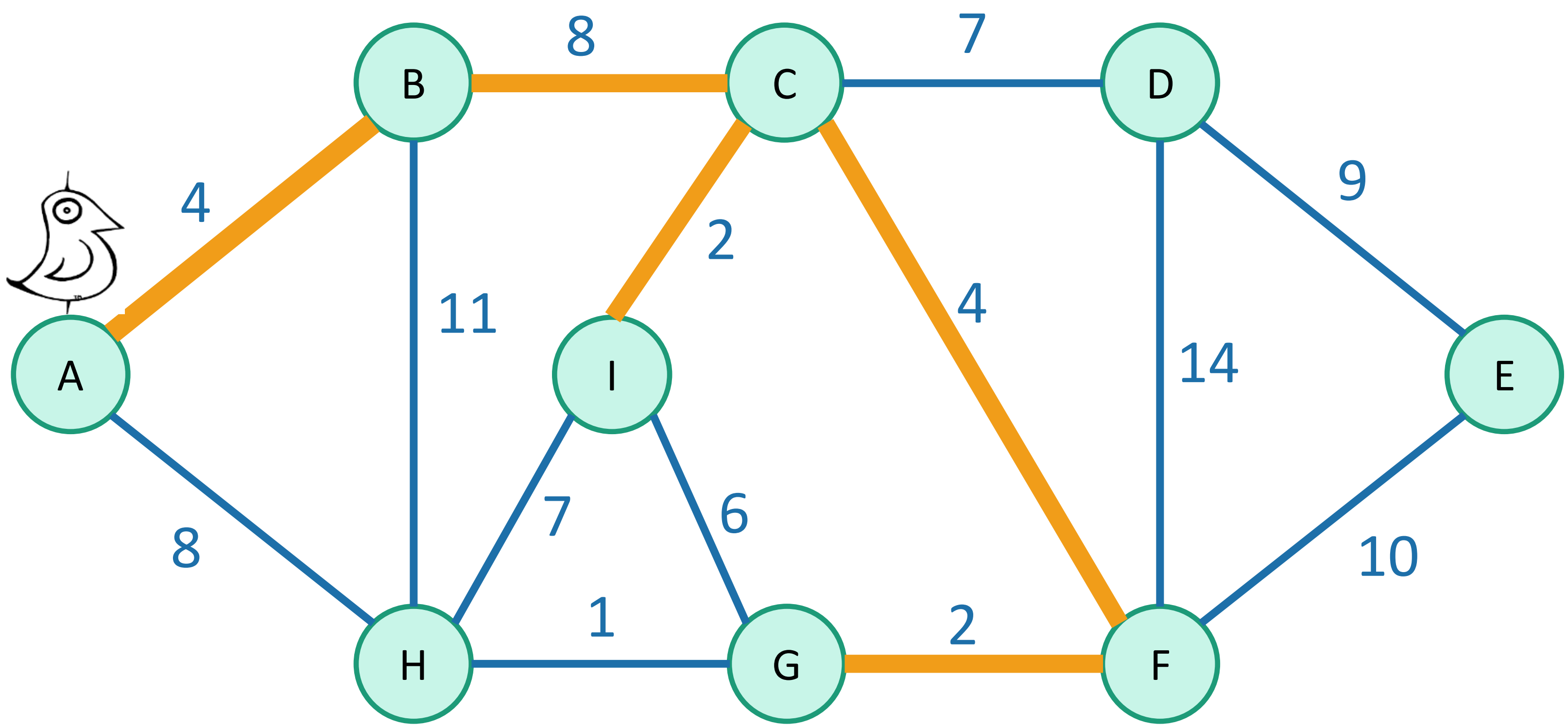
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



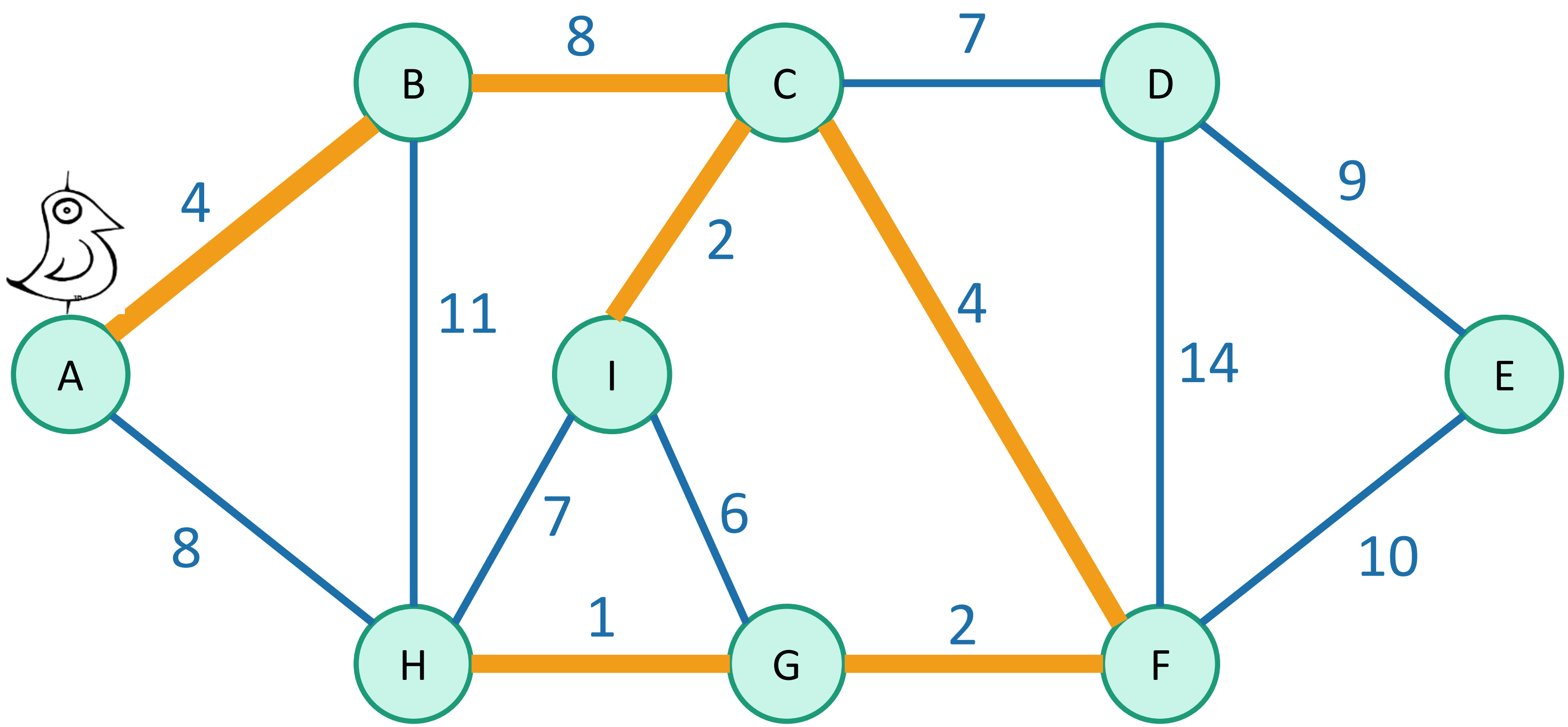
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



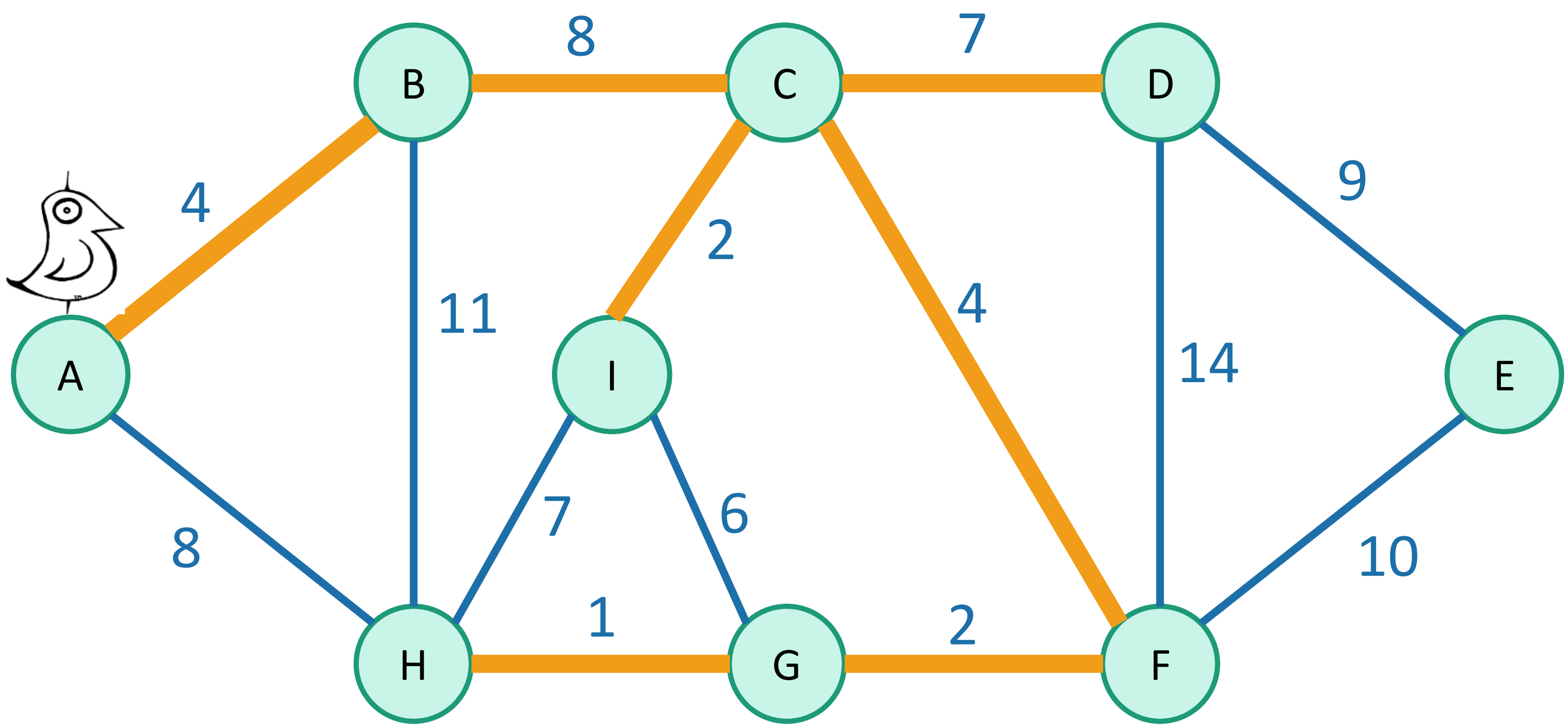
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



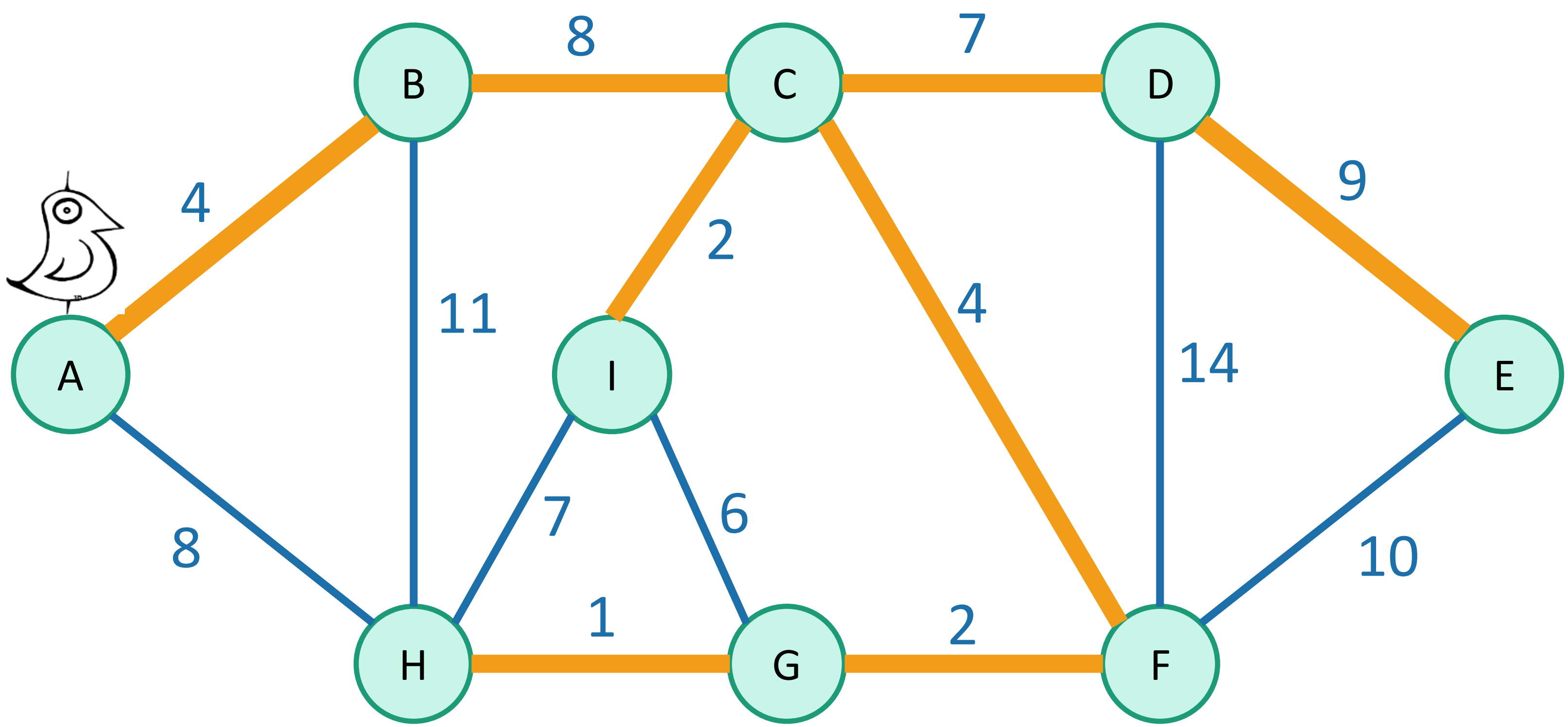
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



We've discovered

Prim's algorithm!

- slowPrim(G = (V,E), starting vertex s):
  - Let (s,u) be the lightest edge coming out of s.
  - MST = { (s,u) }
  - verticesVisited = { s, u }
  - **while** |verticesVisited| < |V|:
    - find the lightest edge {x,v} in E so that:
      - x is in verticesVisited
      - v is not in verticesVisited
    - add {x,v} to MST
    - add v to verticesVisited
  - **return** MST

n iterations of this while loop.

Time at most m to go through all the edges and find the lightest.

Naively, the running time is O(nm):

- For each of n-1 iterations of the while loop:
  - Go through all the edges.

Two questions

1. Does it work?
   - That is, does it actually return a MST?

2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…

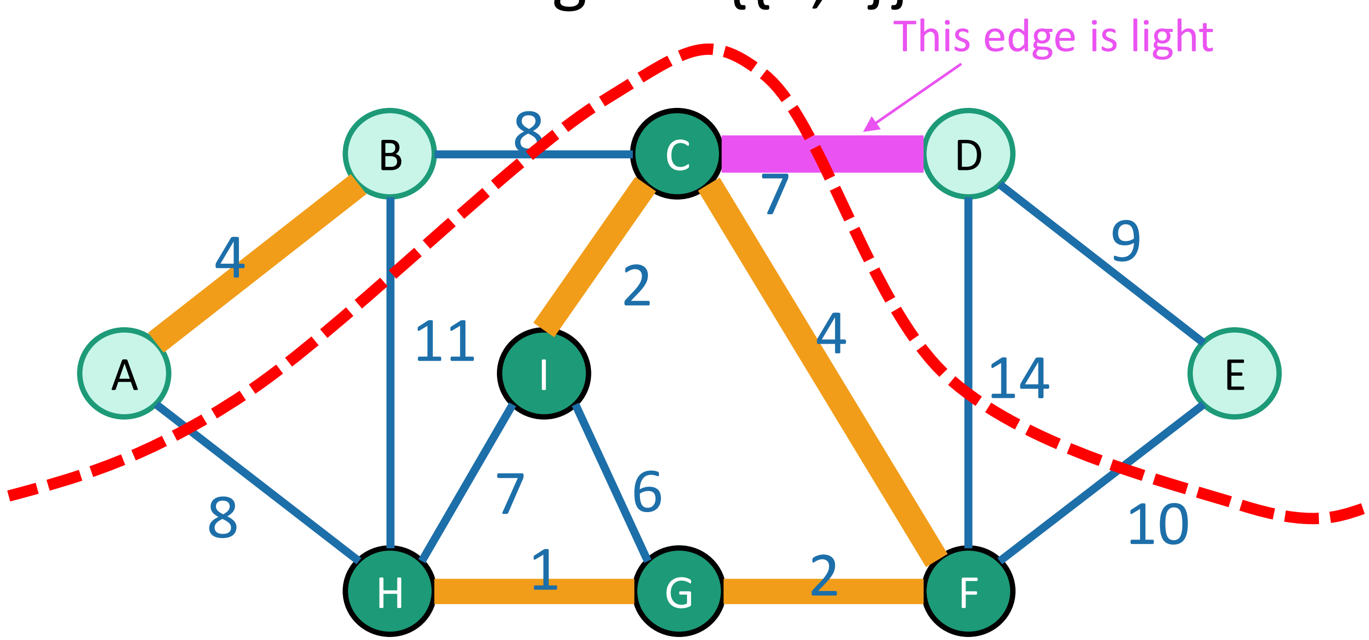
Does it work?

- We need to show that our greedy choices **don't rule out success.**
- That is, at every step:
  - If there exists an MST that contains all of the edges S we have added so far…
  - …then when we make our next choice {u,v}, there is still an MST containing S and {u,v}.

- Now it is time to use our lemma!

Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let {u,v} be a light edge.
- Then there is an MST containing S ∪ {{u,v}}

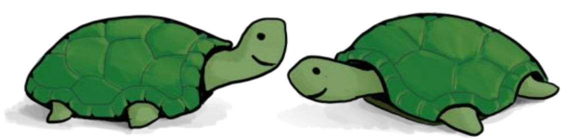


S is the set of **thick orange** edges

Partway through Prim

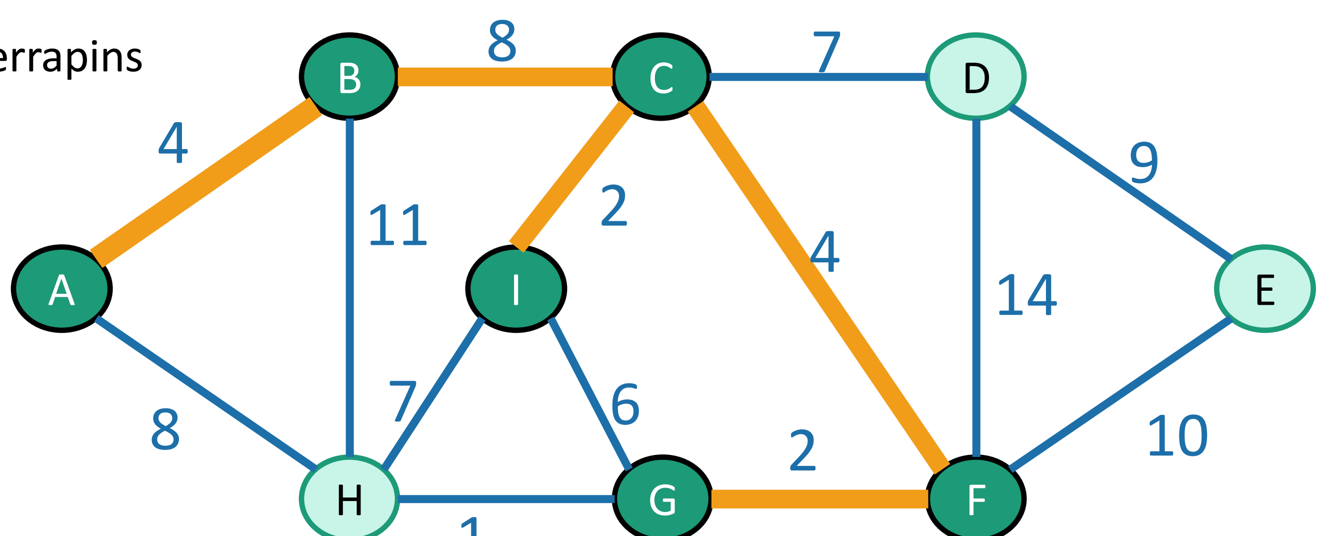
- Assume that our choices **S** so far don’t rule out success
  - There is an MST extending them

How can we use our lemma to show that our next choice also does not rule out success?



Think-Pair-Share Terrapins

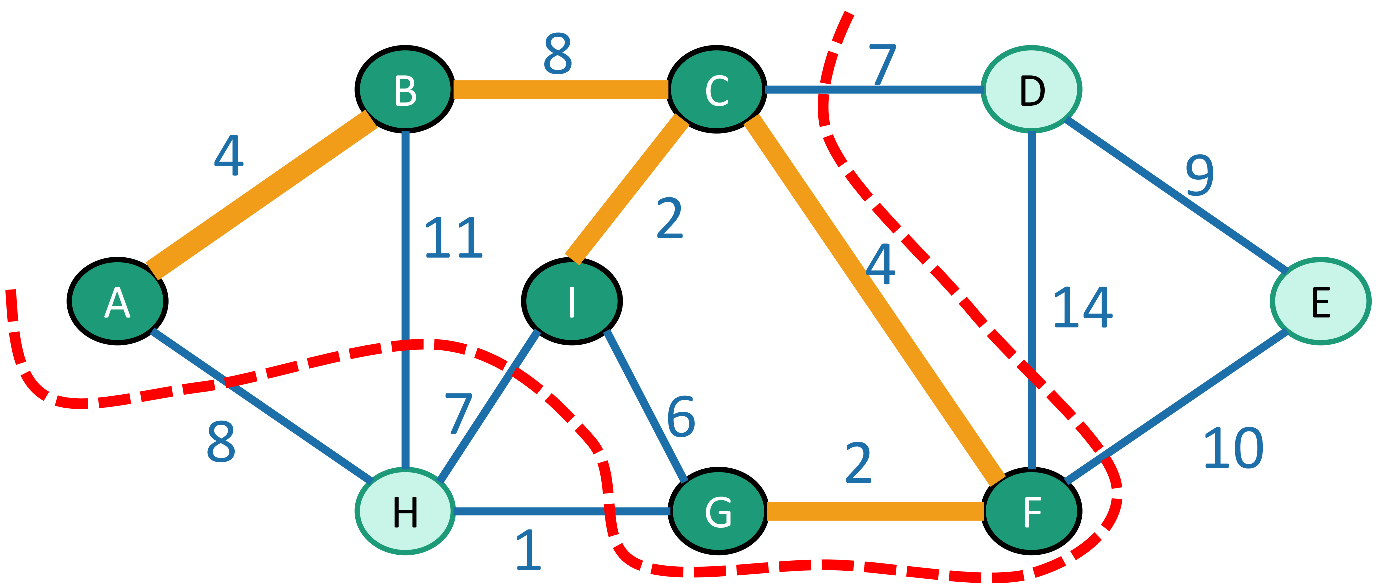
S is the set of edges selected so far.



Partway through Prim

- Assume that our choices **S** so far don't rule out success
  - There is an MST extending them
- Consider the cut {**visited**, **unvisited**}
  - This cut respects S.

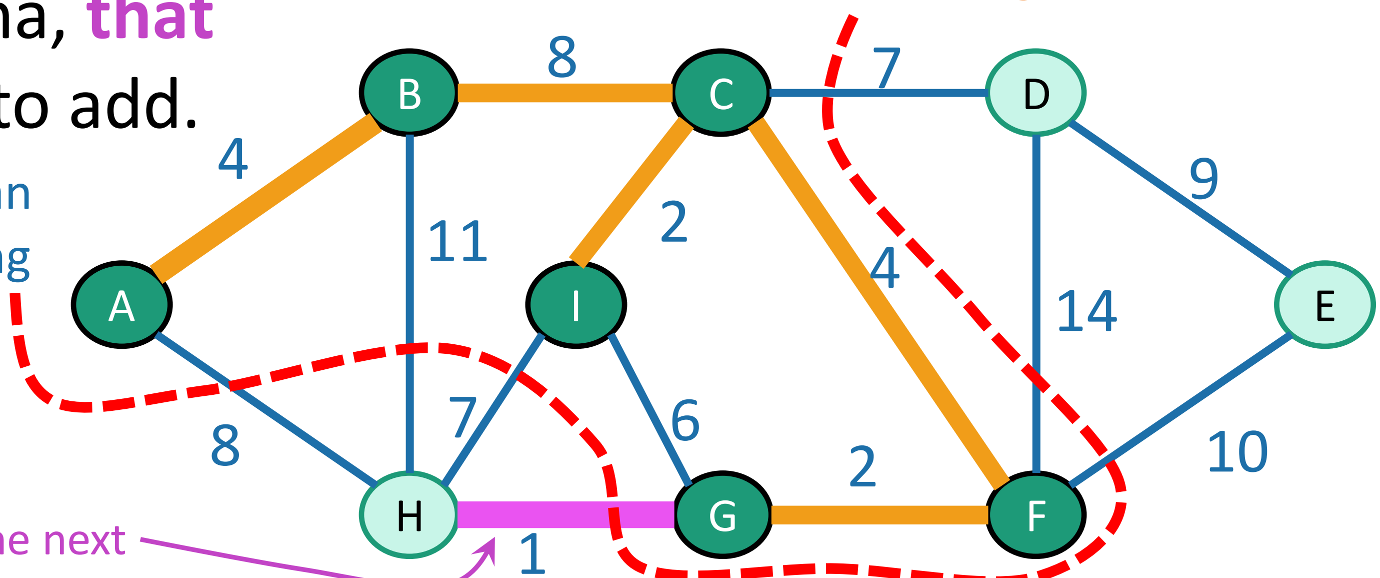
S is the set of edges selected so far.



Partway through Prim

- Assume that our choices **S** so far don't rule out success
  - There is an MST extending them
- Consider the cut {**visited**, **unvisited**}
  - This cut respects S.
- The edge we add next is a **light edge**.
  - Least weight of any edge crossing the cut.
- By the Lemma, **that edge** is safe to add.
  - There is still an MST extending the new set

S is the set of edges selected so far.



Hooray!

- Our greedy choices **don’t rule out success**.

- This is enough (along with an argument by induction) to guarantee correctness of Prim’s algorithm.

Formally(ish)

- Inductive hypothesis:
  - After adding the t'th edge, there exists an MST with the edges added so far.
- Base case:
  - After adding the 0'th edge, there exists an MST with the edges added so far. **YEP.**
- Inductive step:
  - If the inductive hypothesis holds for t (aka, the choices so far are safe), then it holds for t+1 (aka, the next edge we add is safe).
  - **That's what we just showed.**
- Conclusion:
  - After adding the n-1'st edge, there exists an MST with the edges added so far.
  - At this point we have a spanning tree, so it better be minimal.

Two questions

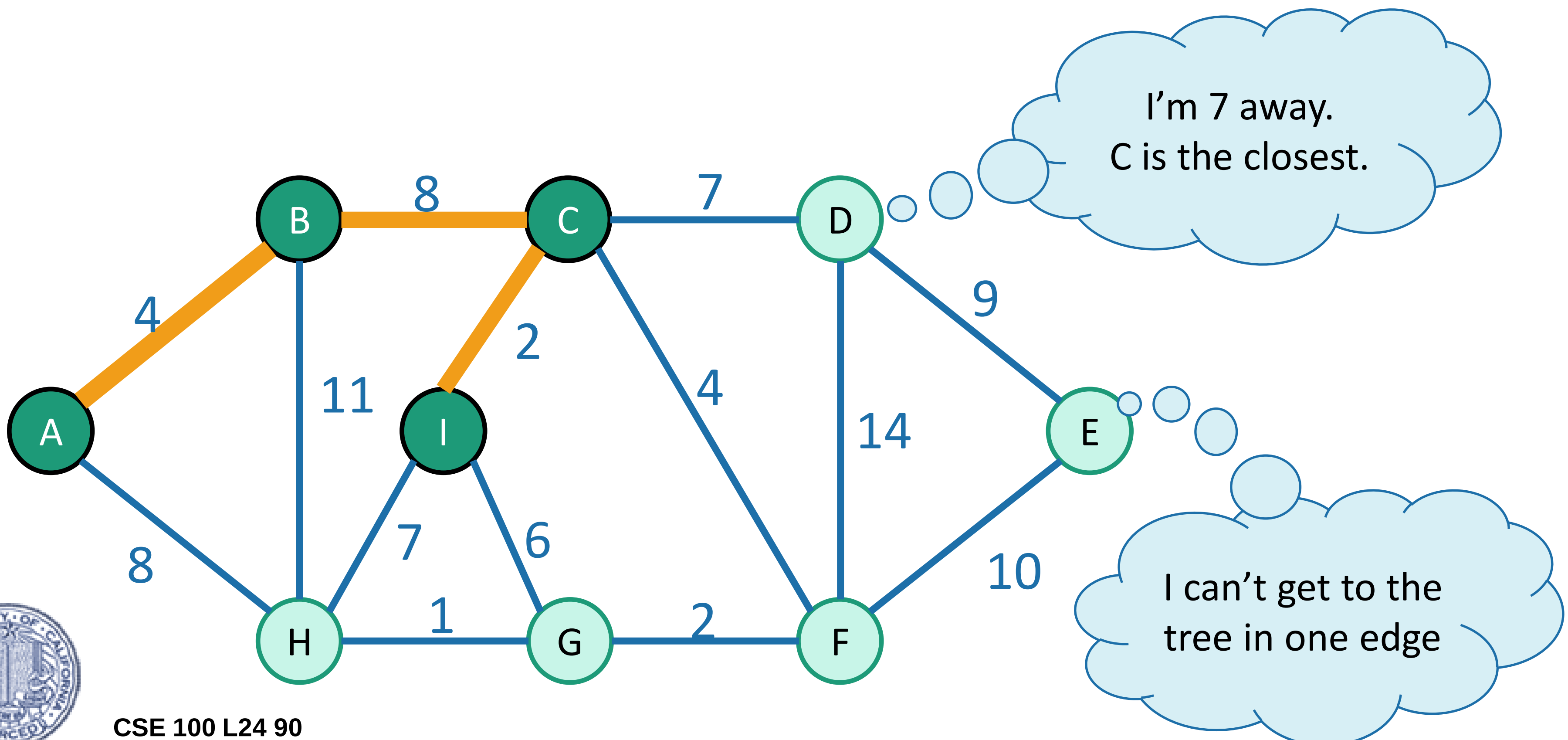
1. Does it work?
   - That is, does it actually return a MST?
     - **Yes!**

2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…

How do we actually implement this?

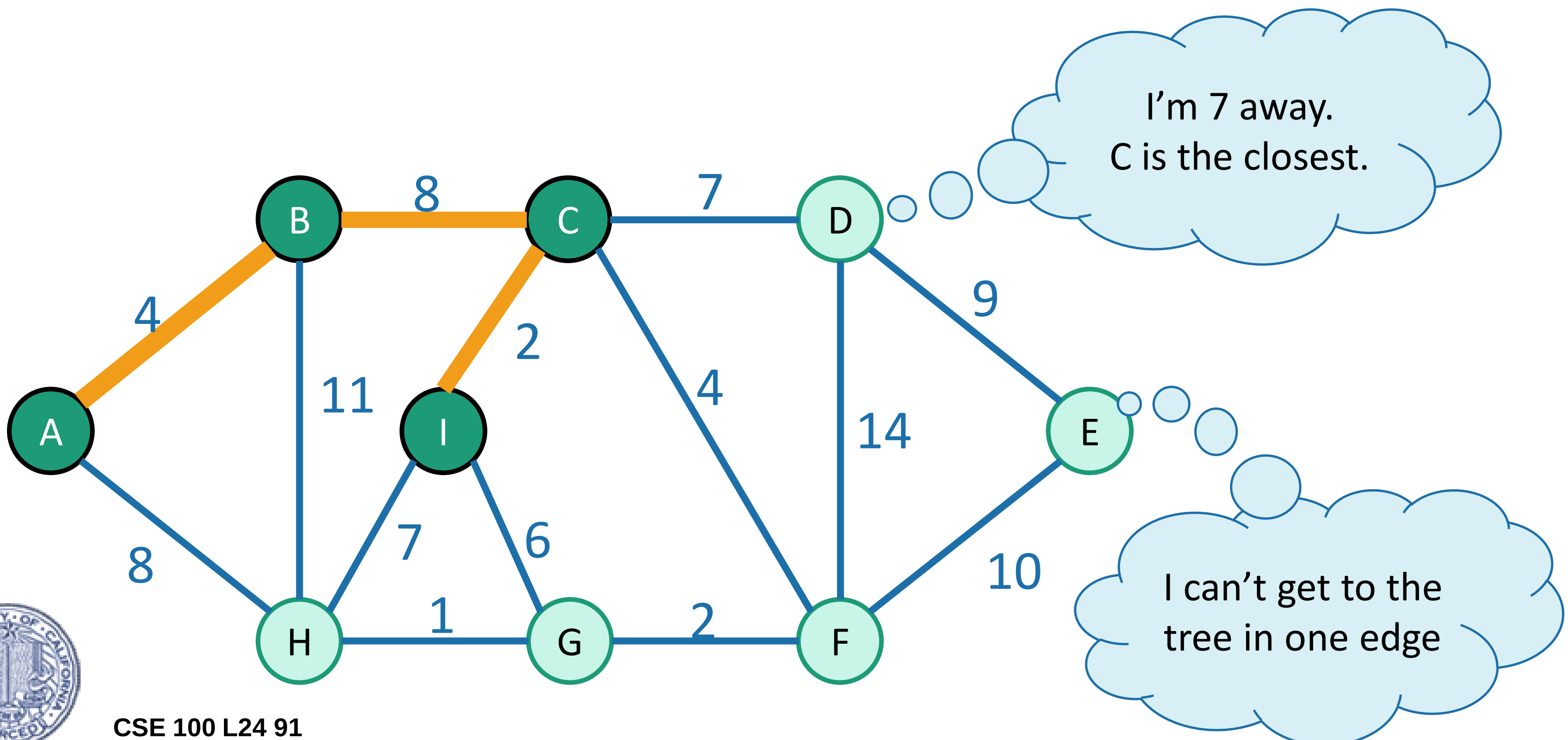
- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.

if you can get there in one edge.



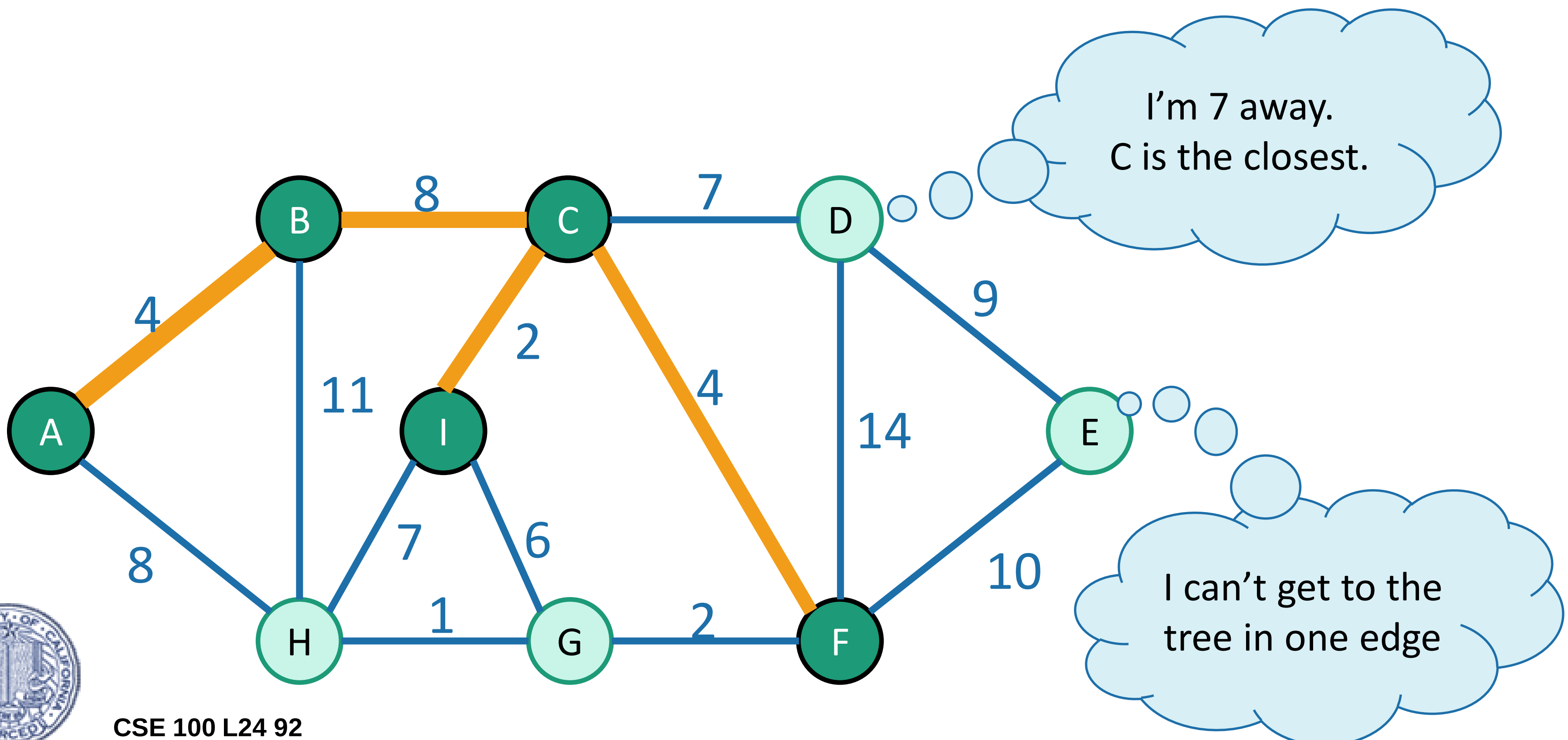
How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree** if you can get there in one edge.
  - **how to get there**.
- Choose the closest vertex, add it.



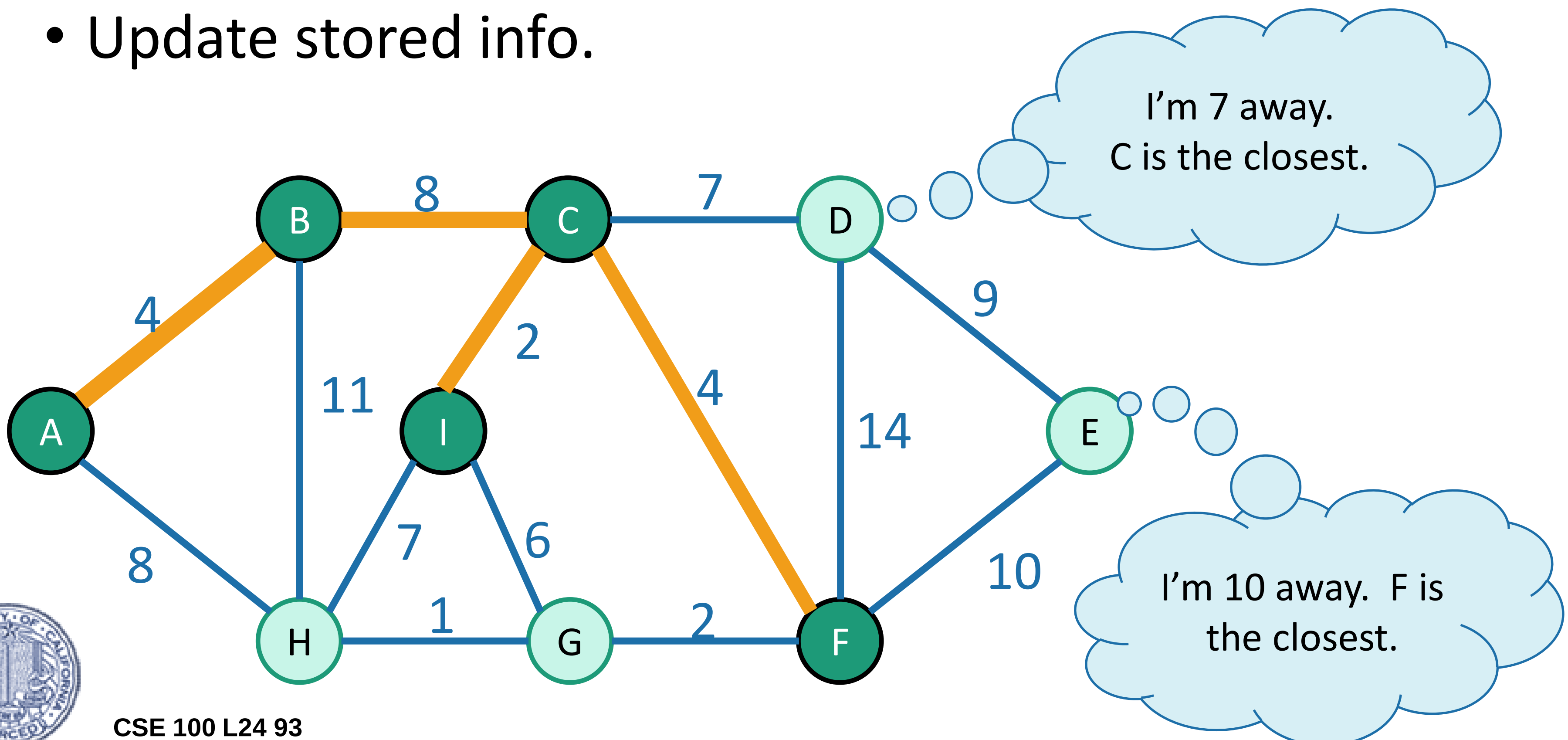
How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree** if you can get there in one edge.
  - **how to get there**.
- Choose the closest vertex, add it.



How do we actually implement this?

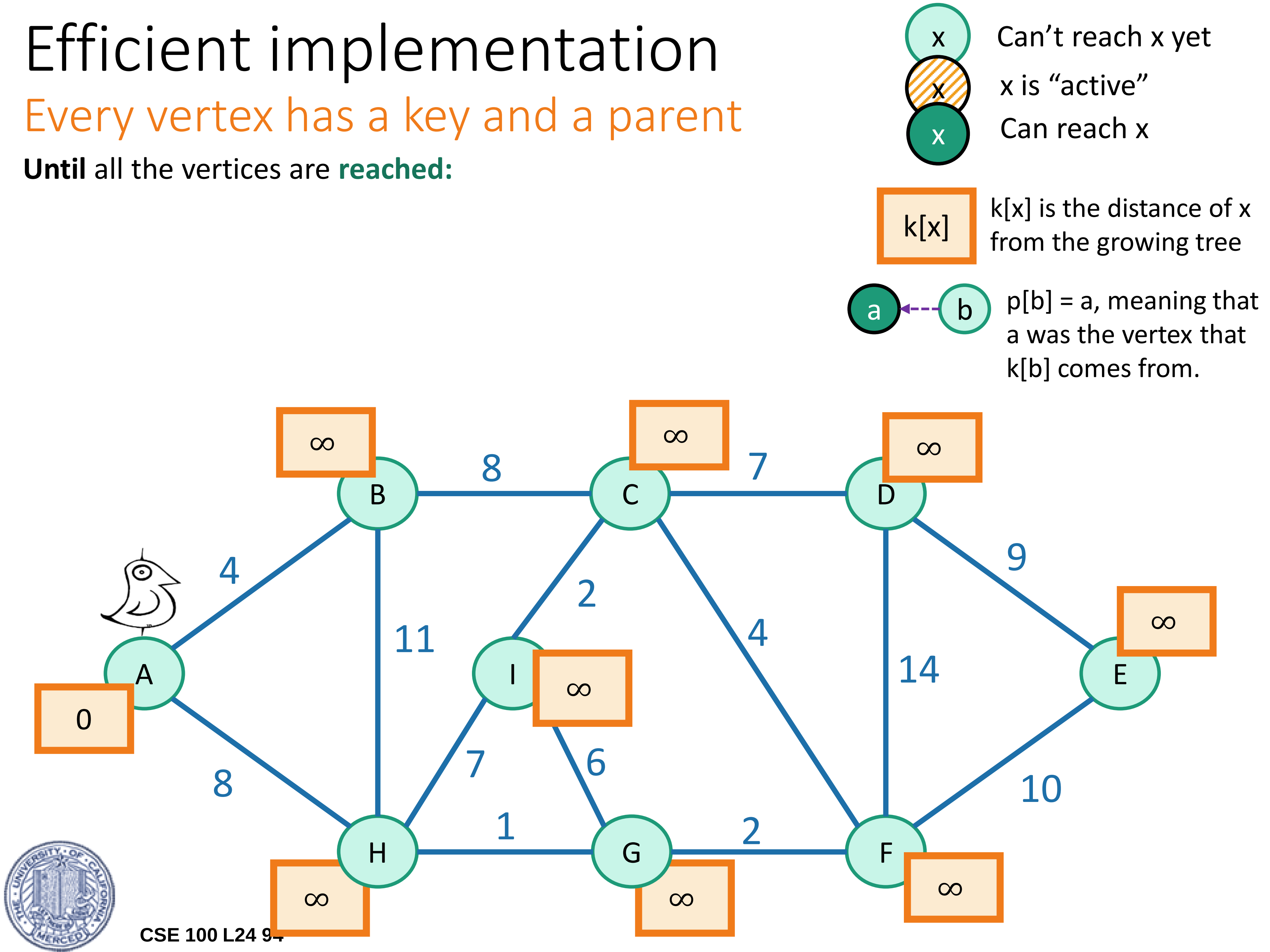
- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree** if you can get there in one edge.
  - **how to get there**.
- Choose the closest vertex, add it.
- Update stored info.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

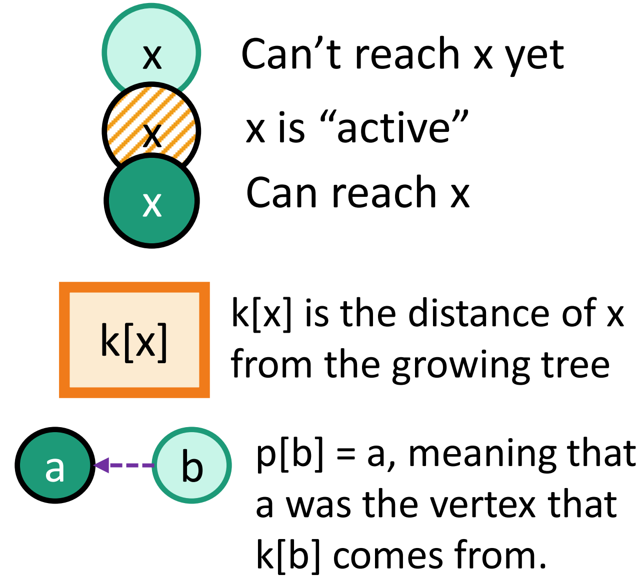


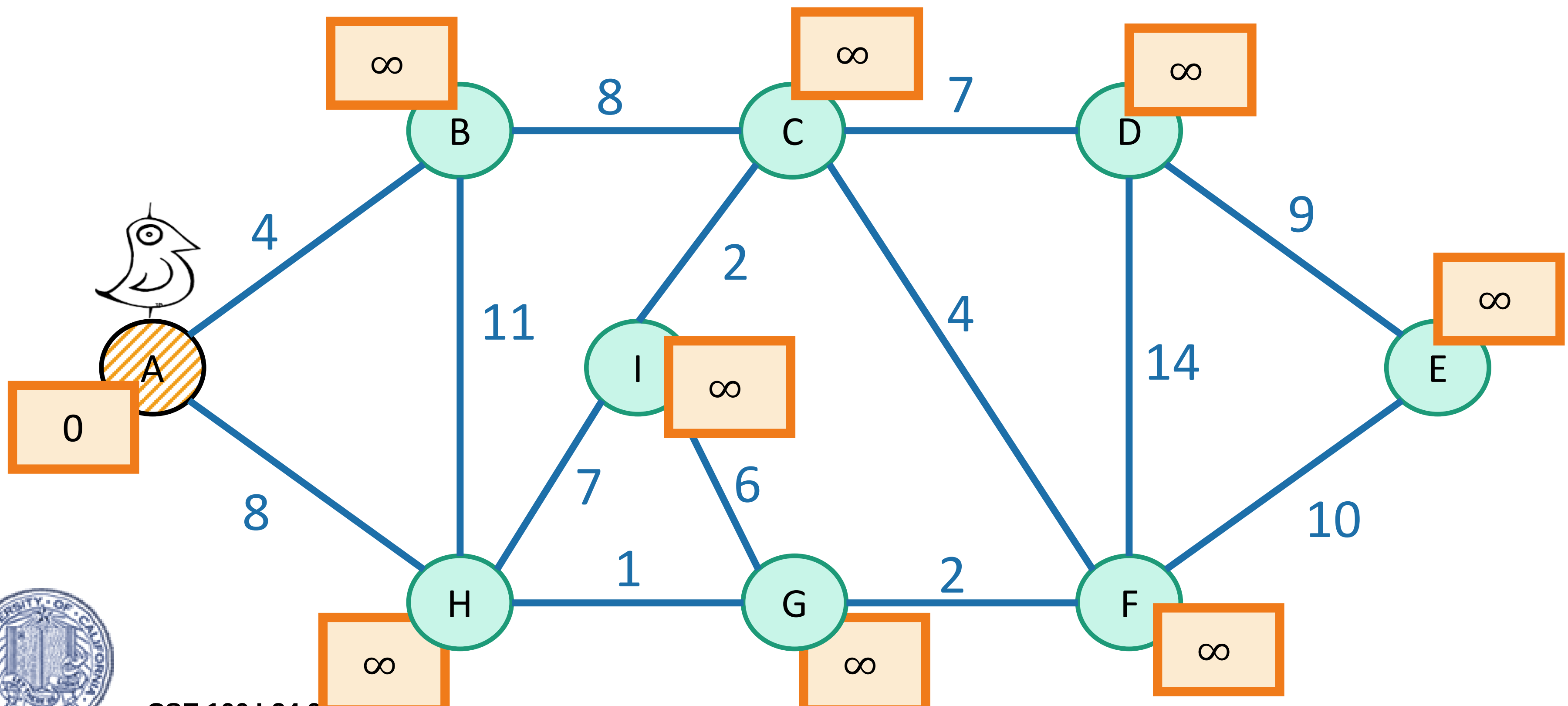
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

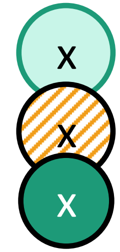
- Activate the **unreached** vertex u with the smallest key.





Efficient implementation

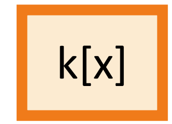
Every vertex has a key and a parent



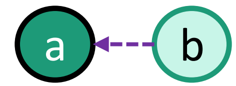
Can't reach x yet

x is "active"

Can reach x



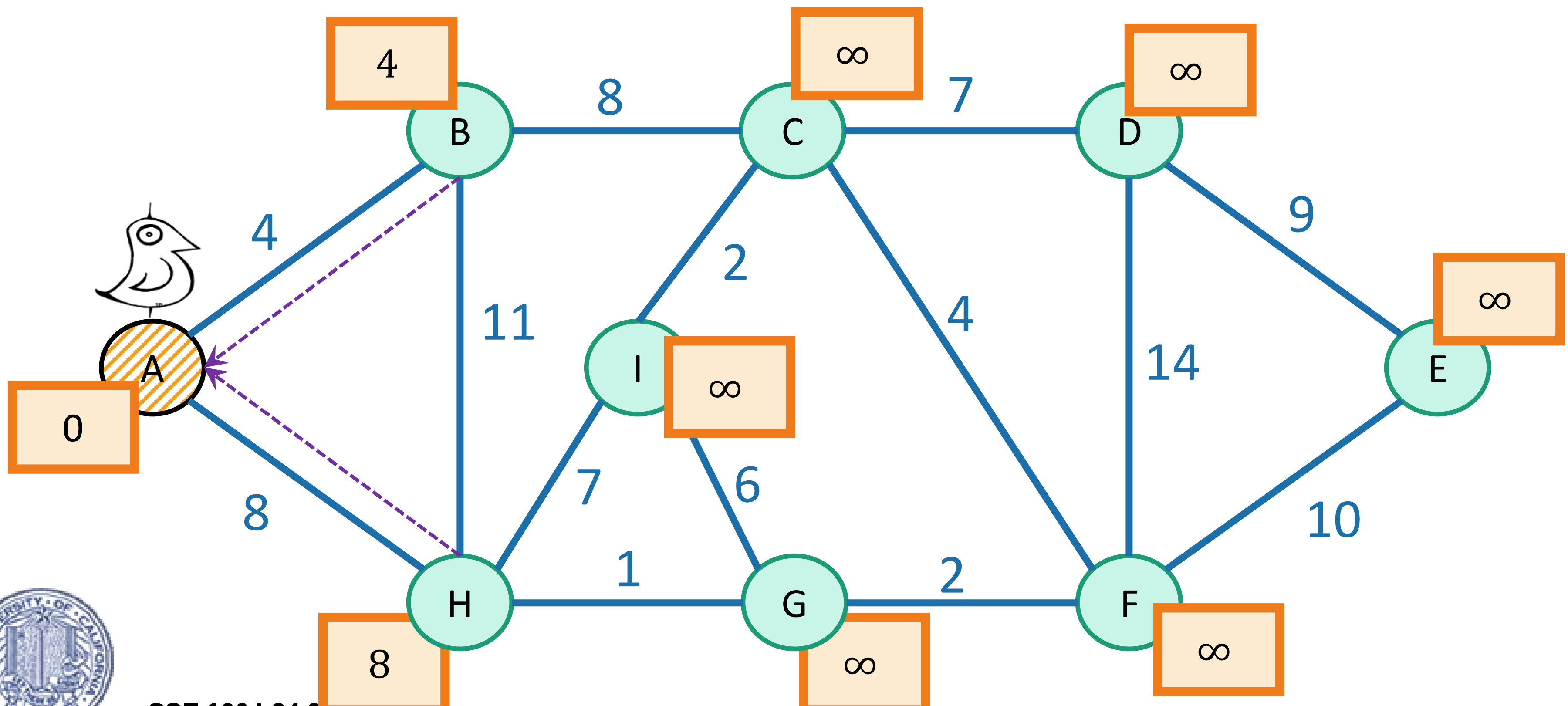
k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

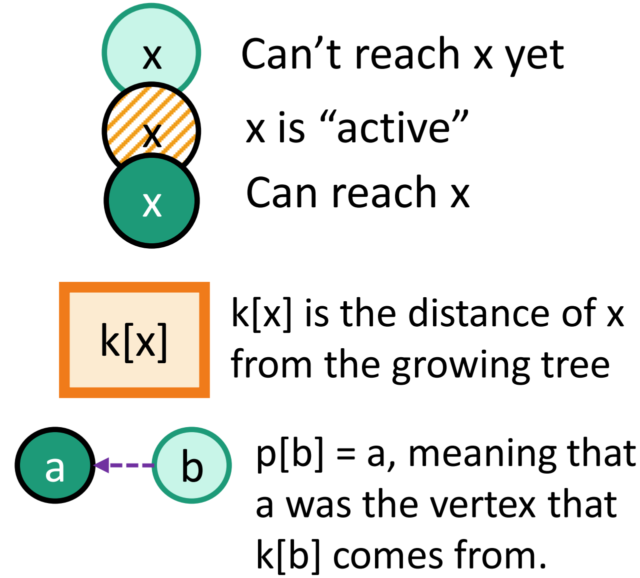
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u



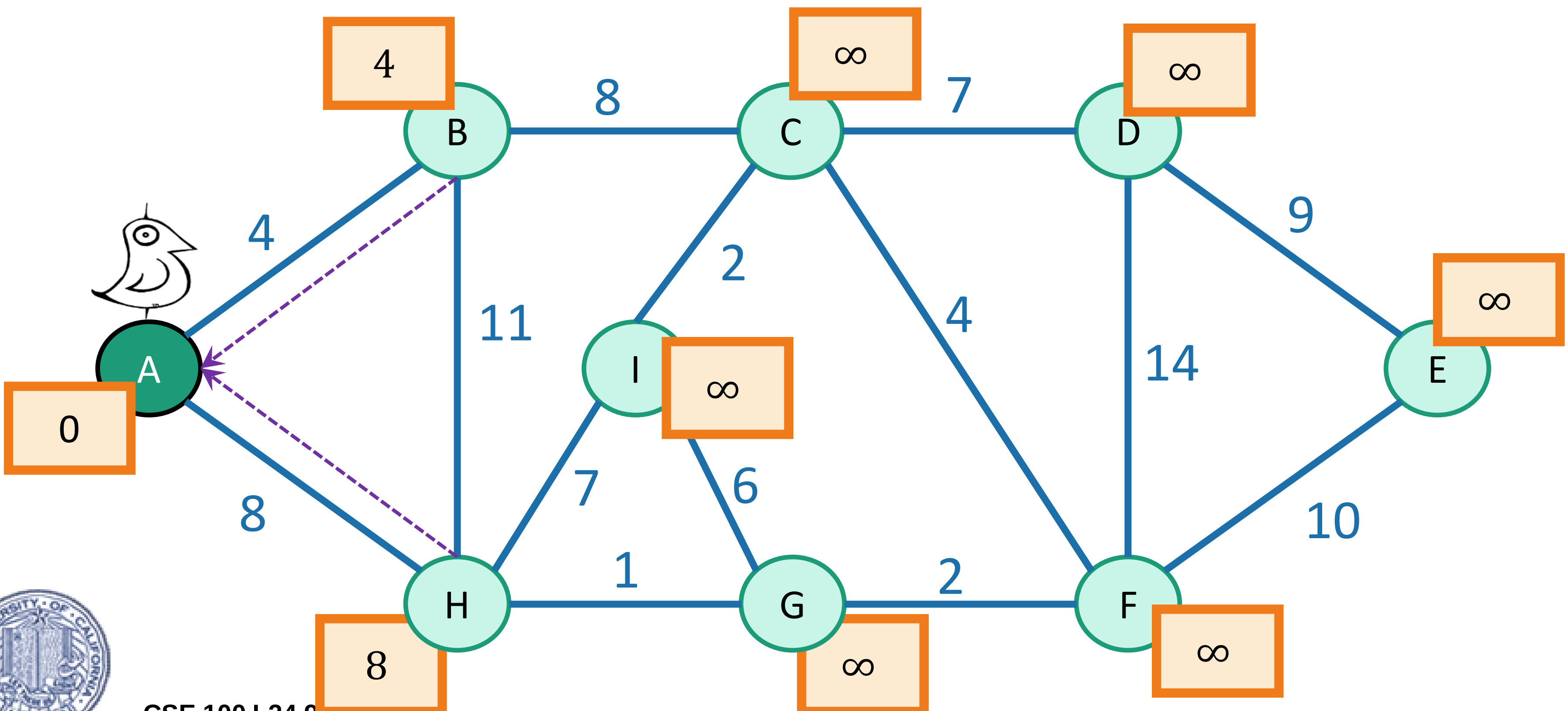
Efficient implementation

Every vertex has a key and a parent



Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

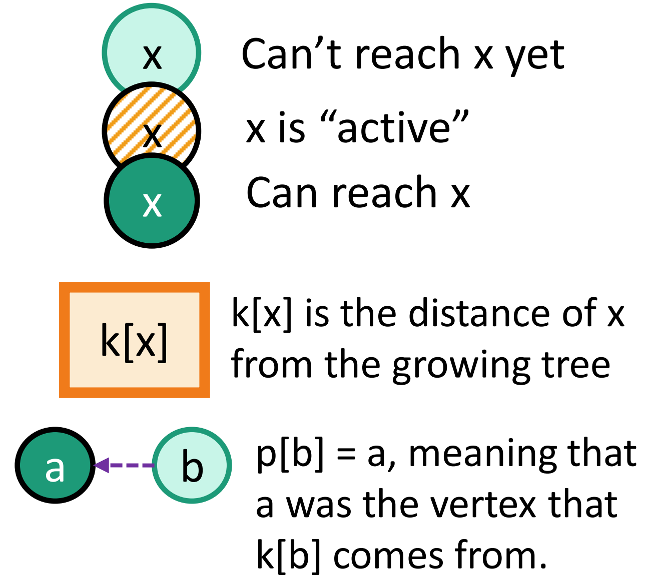


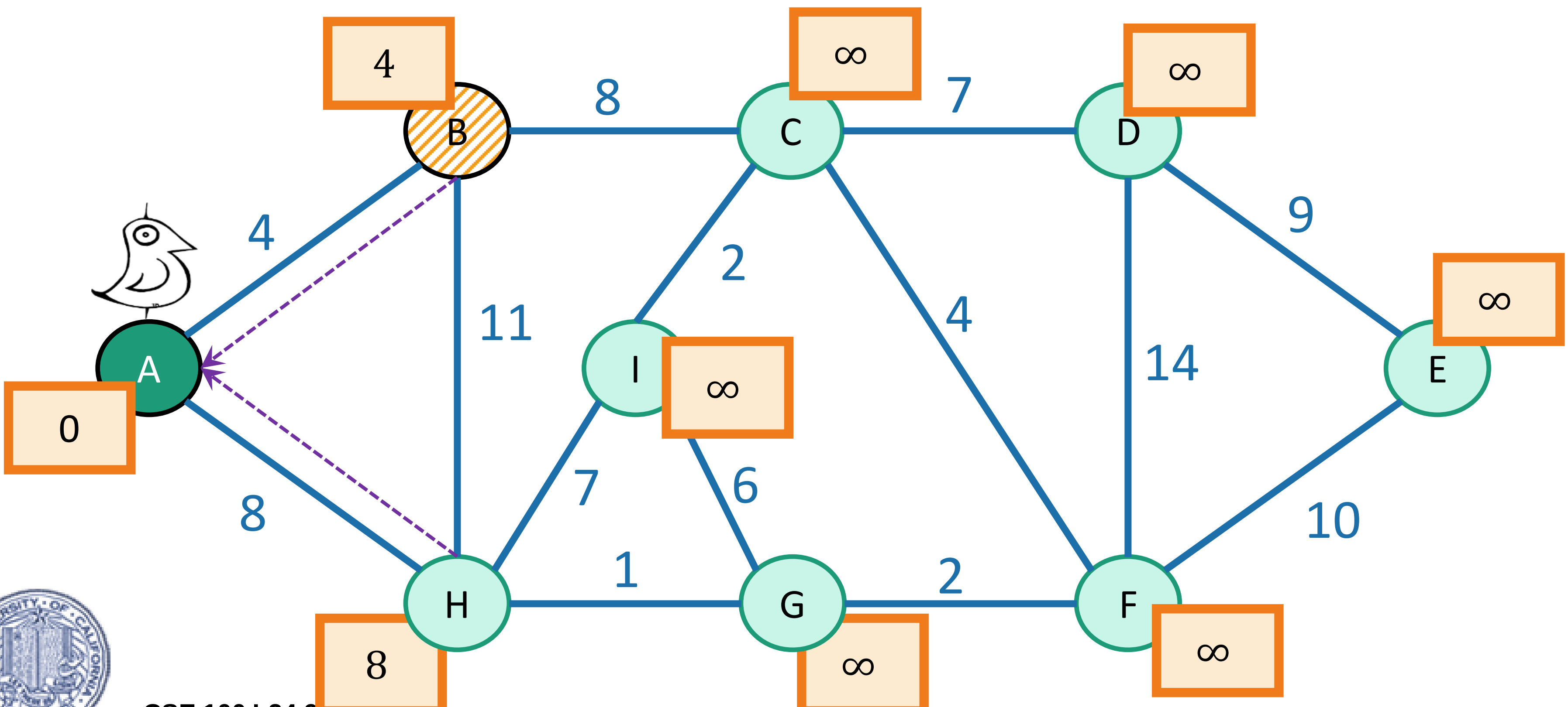
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



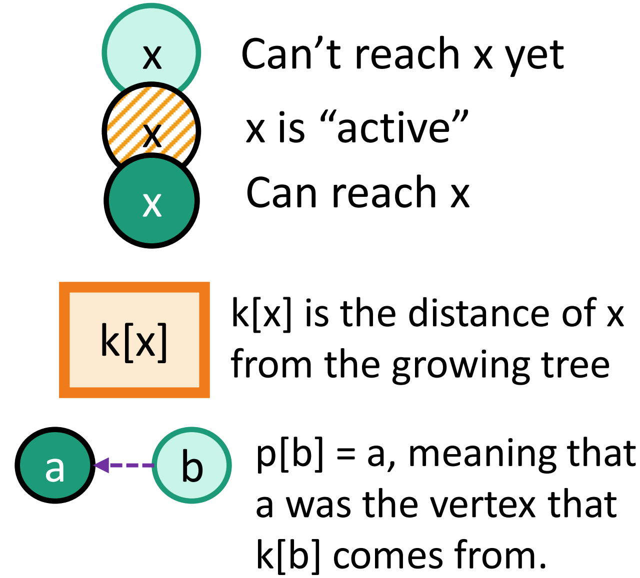


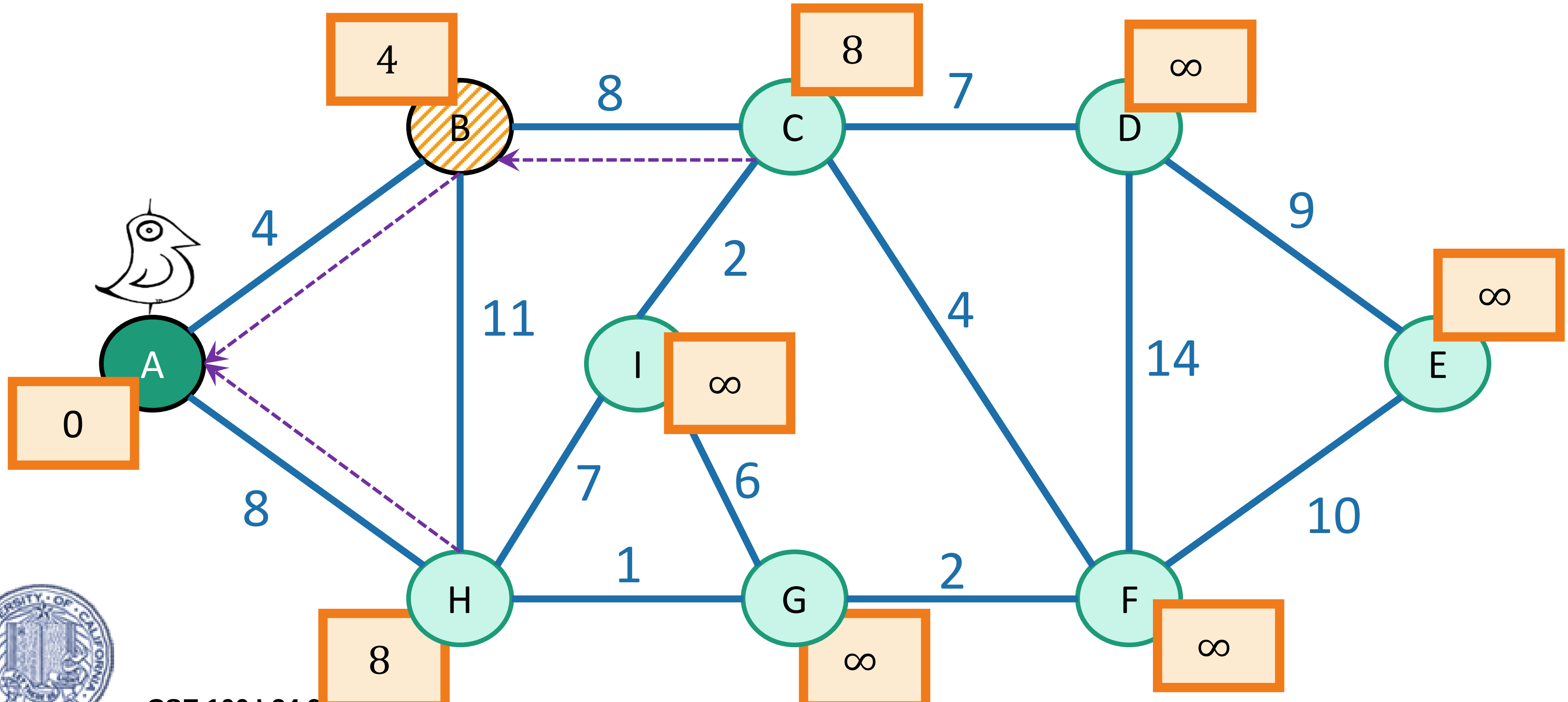
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

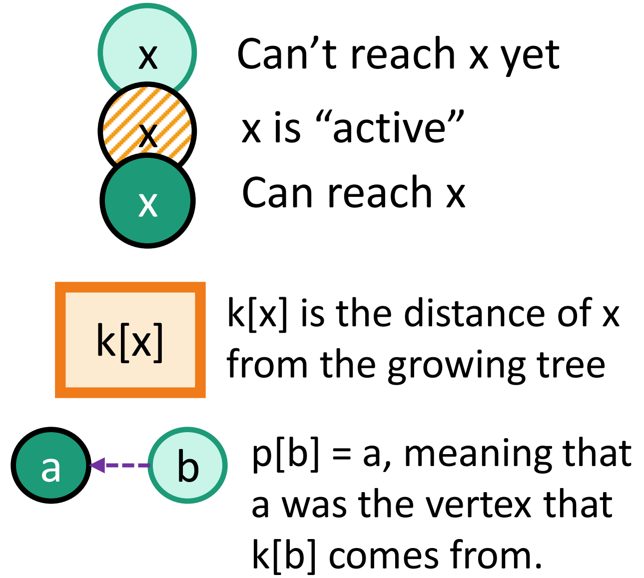
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**





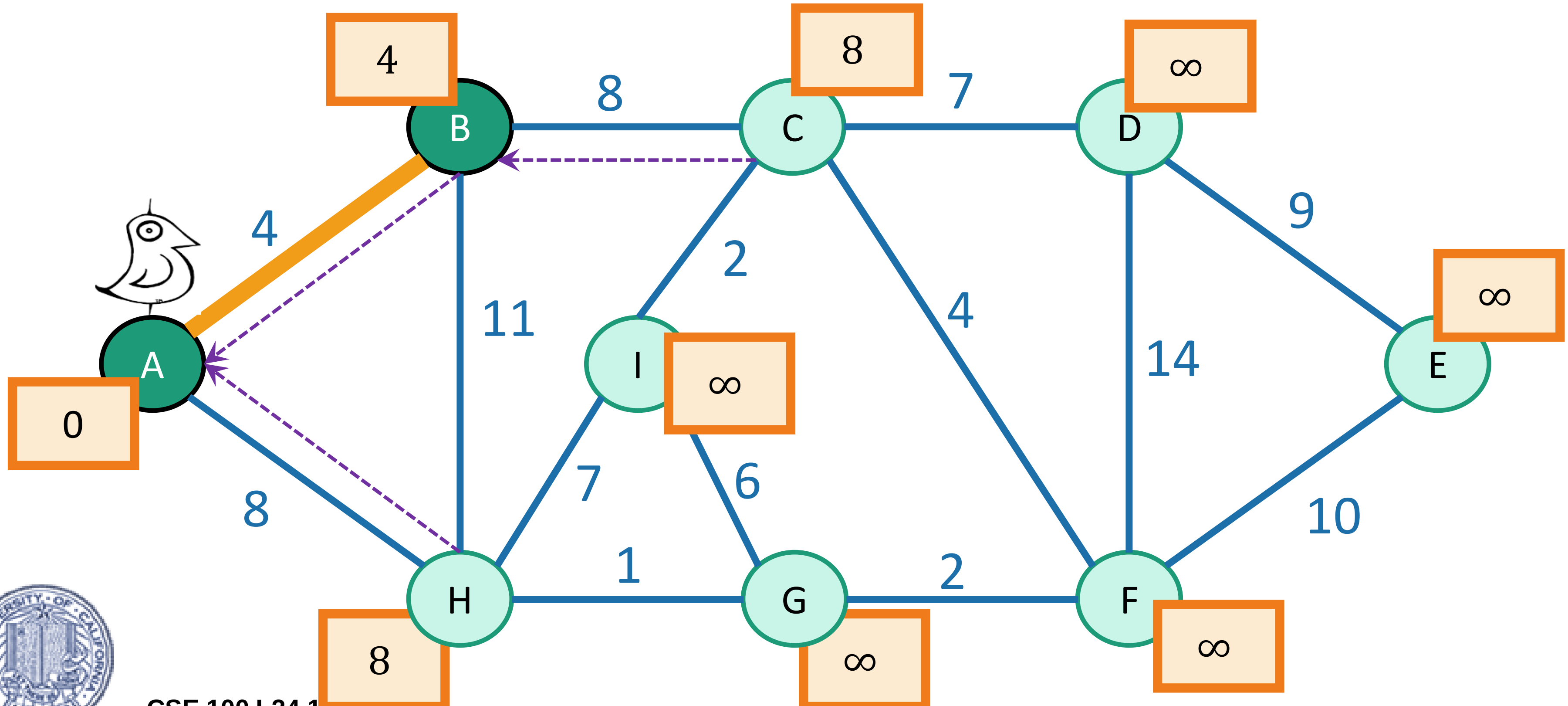
Efficient implementation

Every vertex has a key and a parent



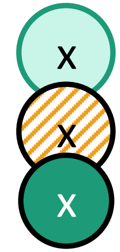
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Efficient implementation

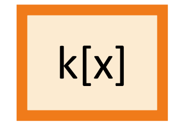
Every vertex has a key and a parent



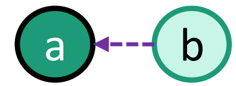
Can't reach x yet

x is "active"

Can reach x



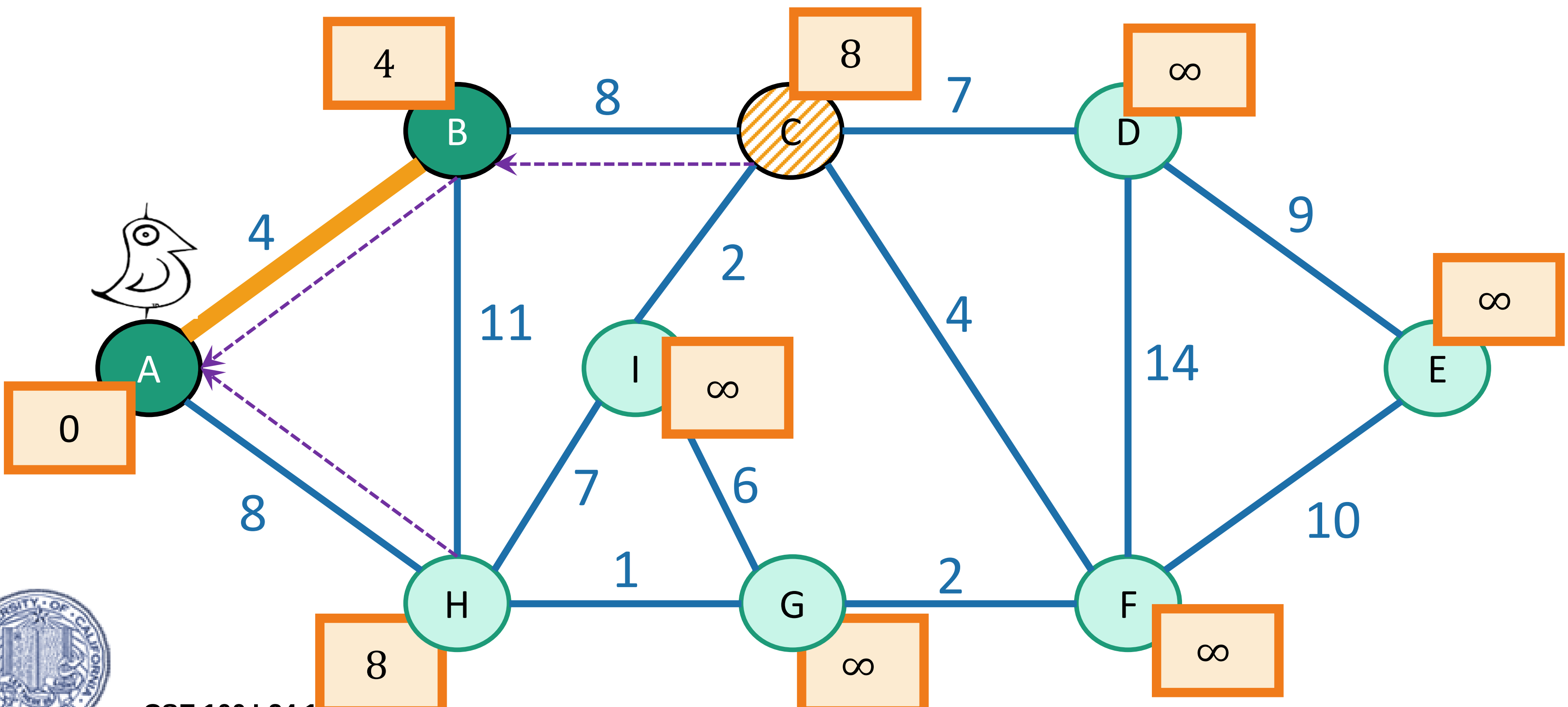
k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

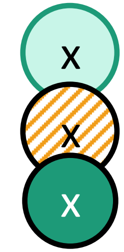
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Efficient implementation

Every vertex has a key and a parent



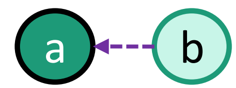
Can't reach x yet

x is "active"

Can reach x

k[x]

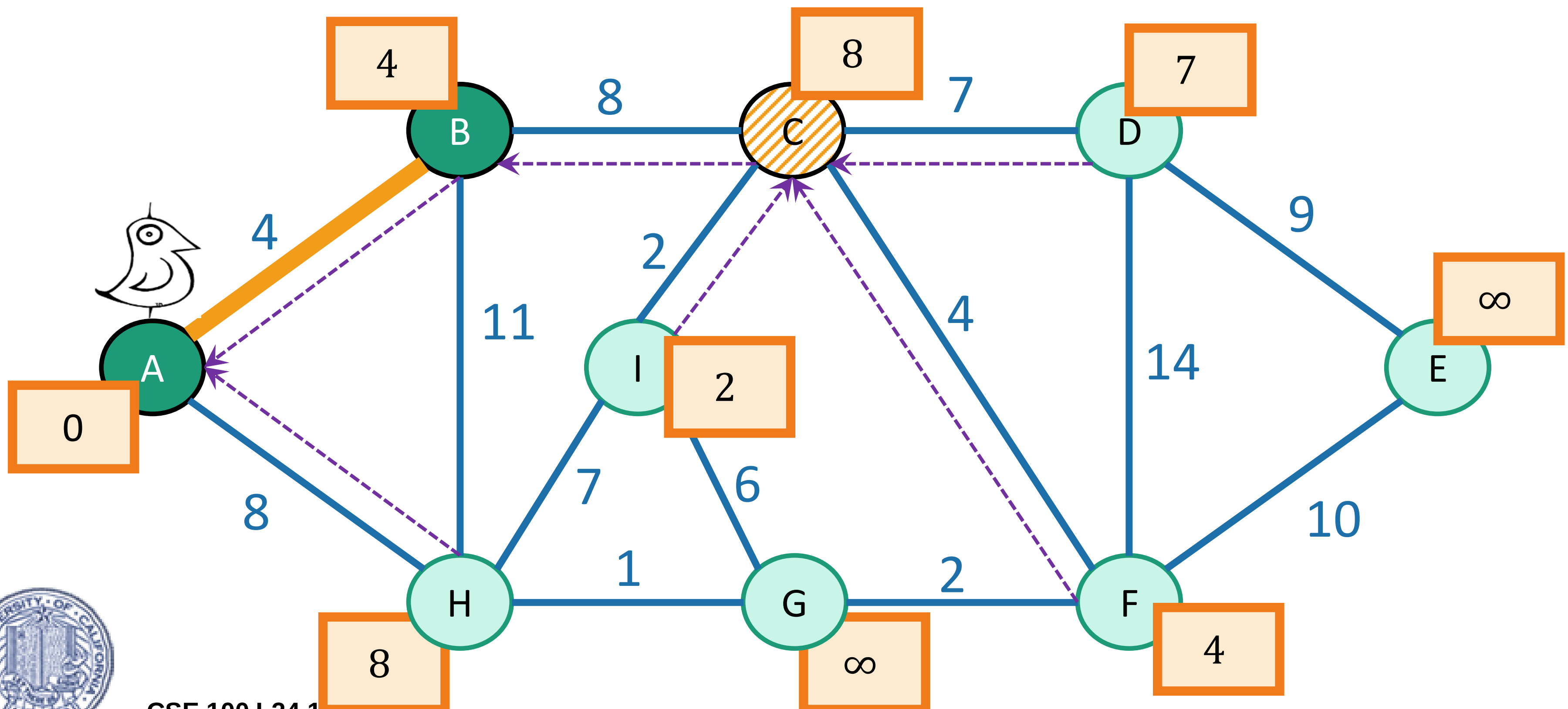
k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

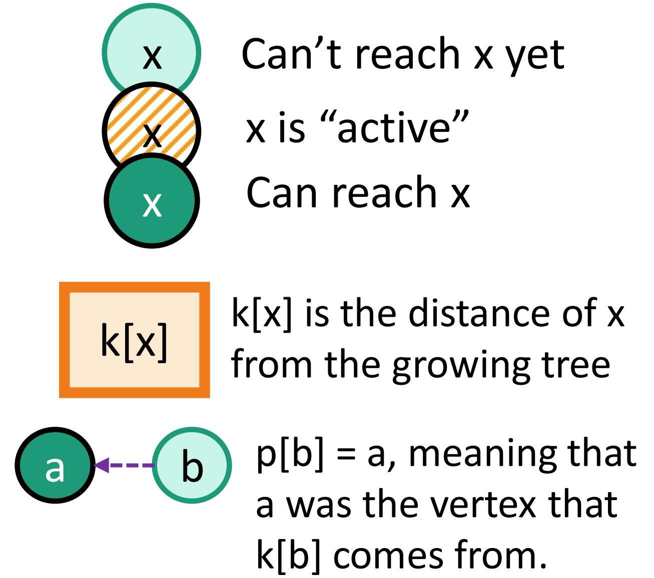


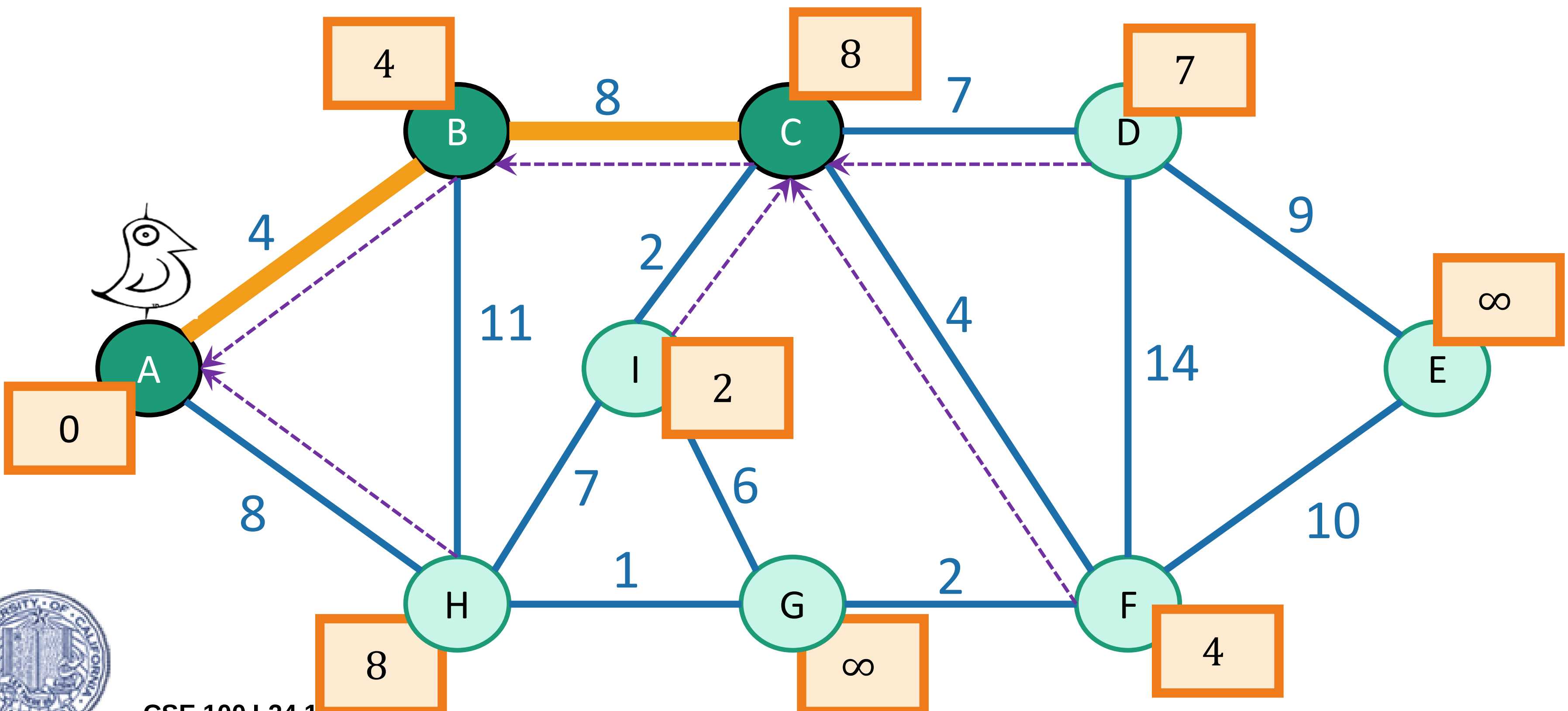
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



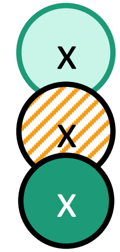


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

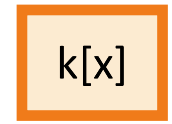
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



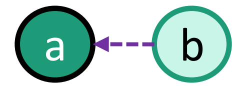
Can't reach x yet

x is "active"

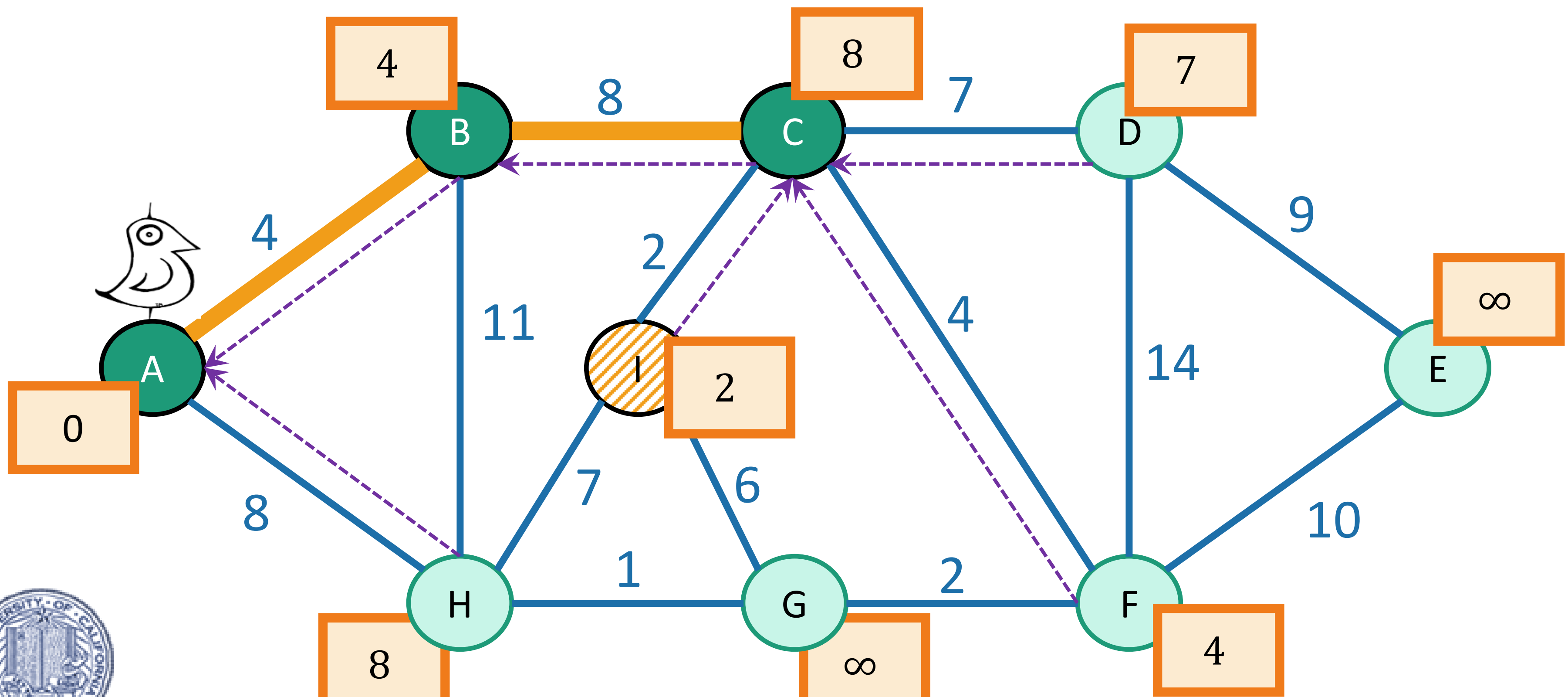
Can reach x



k[x] is the distance of x from the growing tree

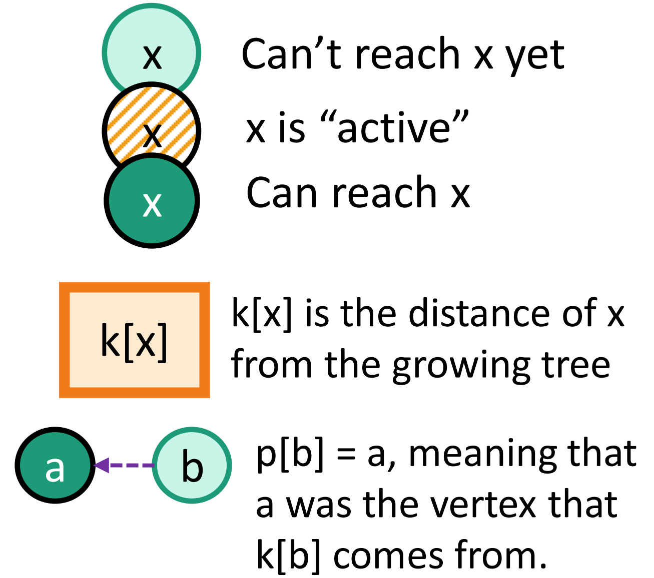


p[b] = a, meaning that a was the vertex that k[b] comes from.



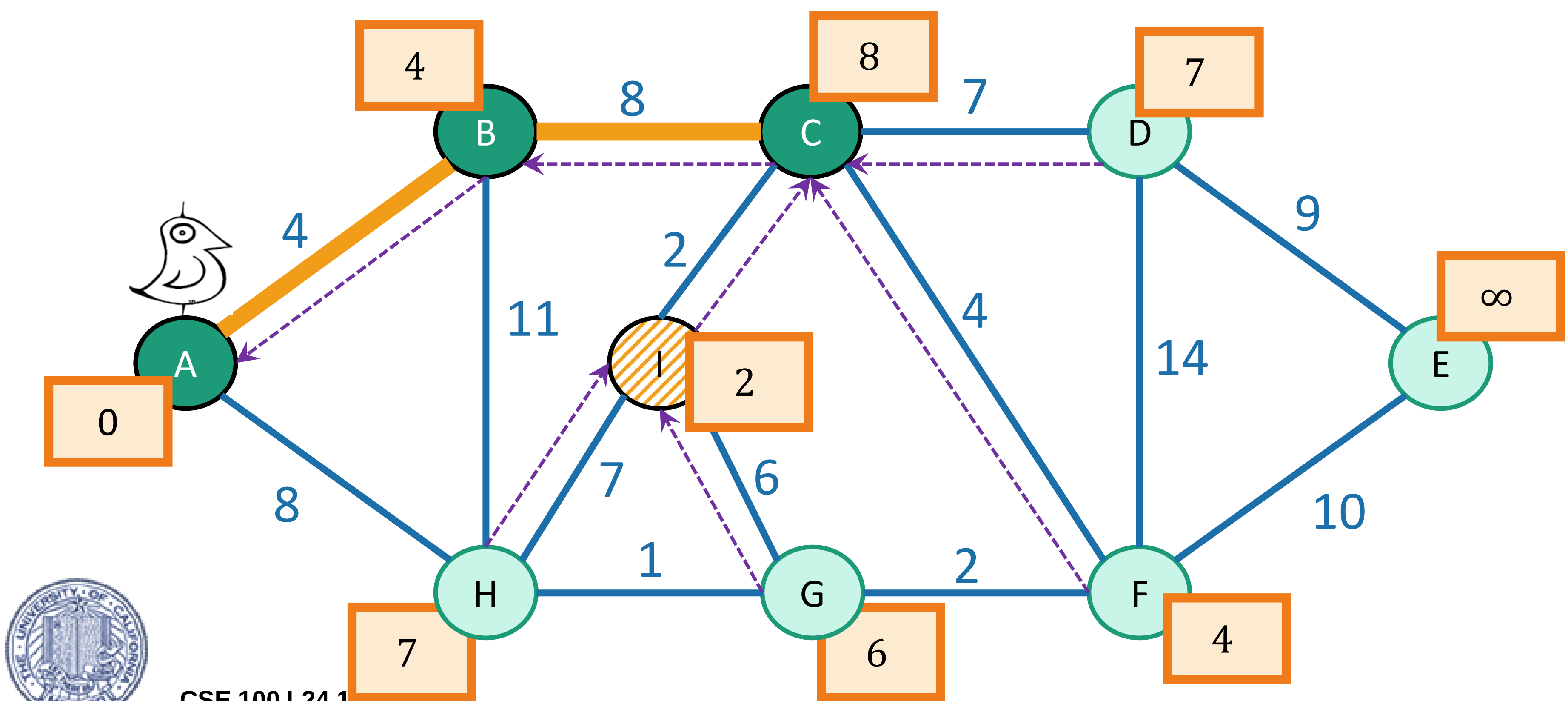
Efficient implementation

Every vertex has a key and a parent



Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

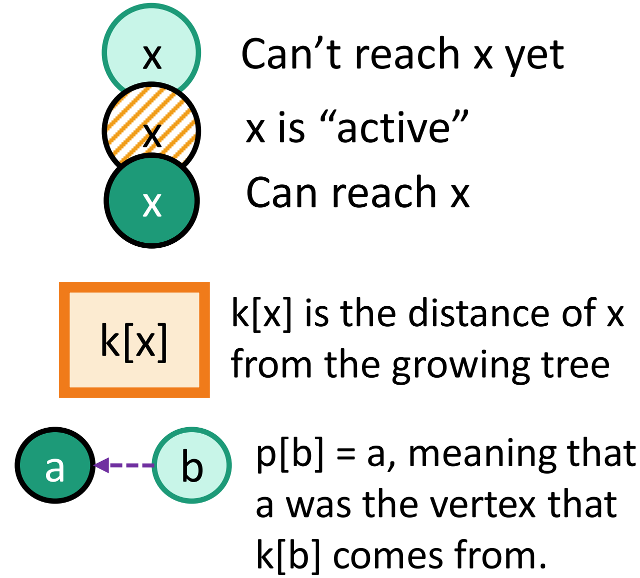


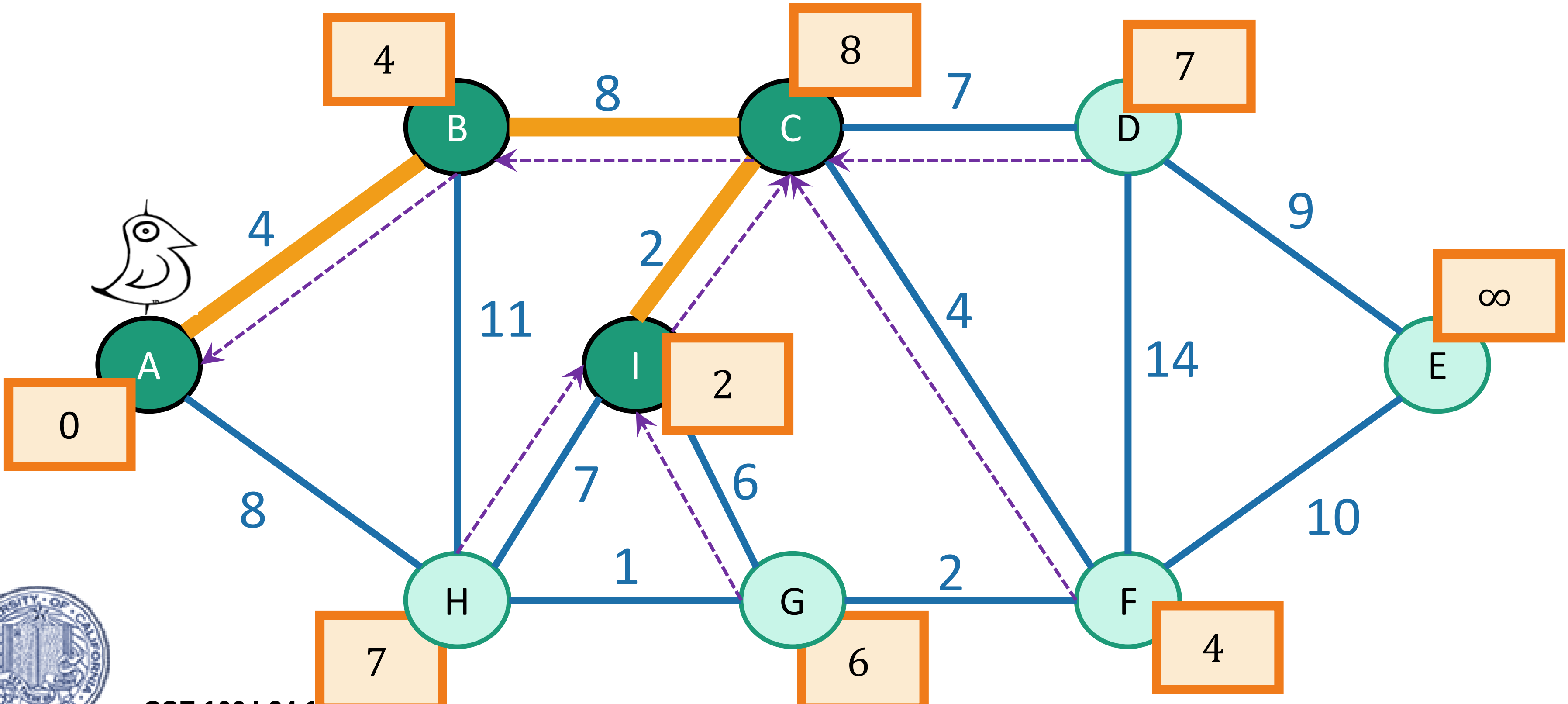
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



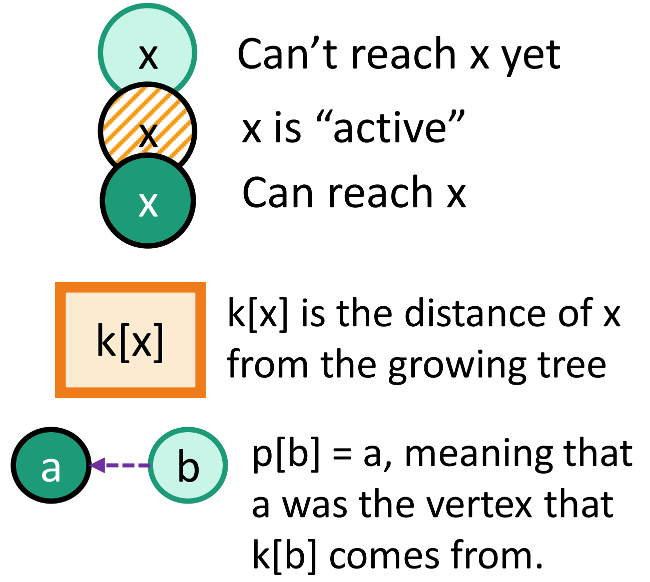


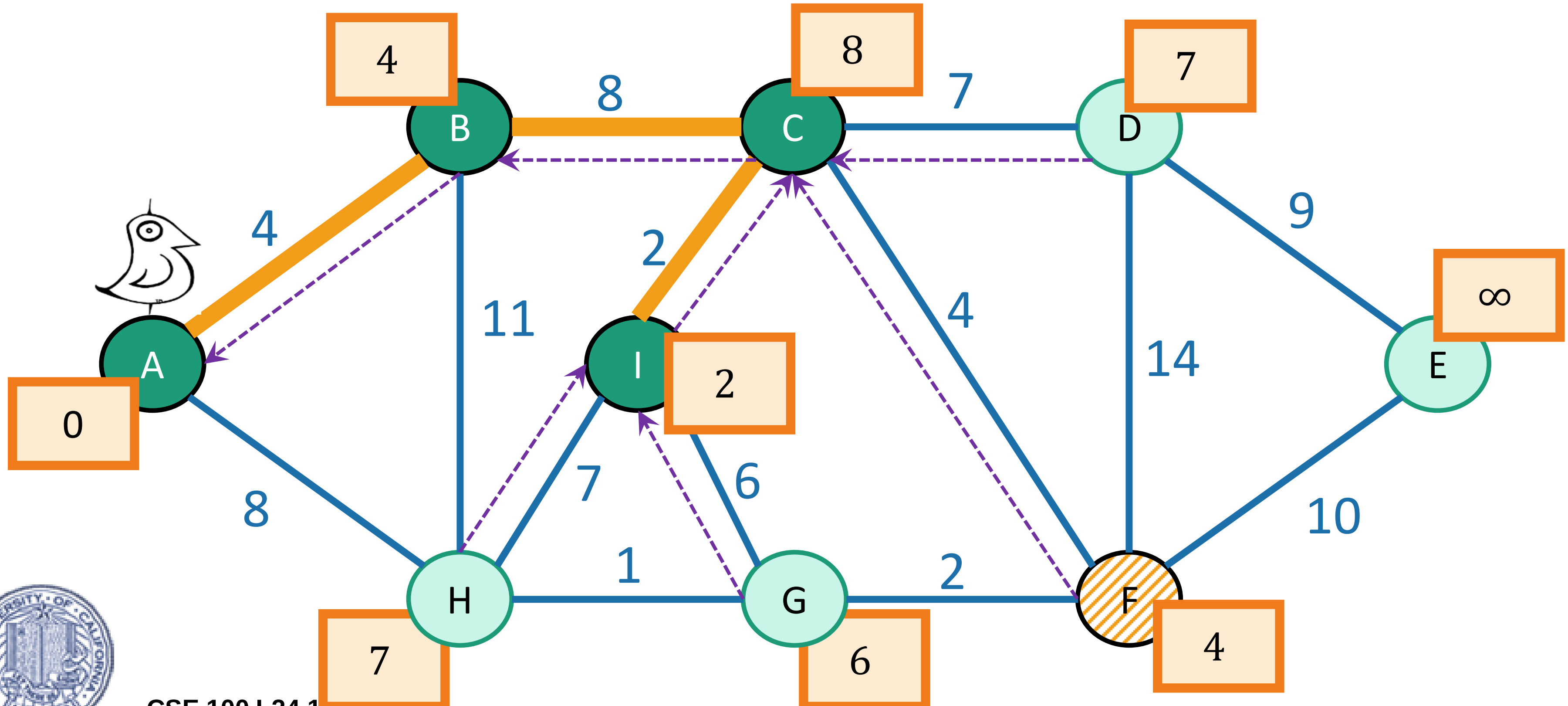
Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

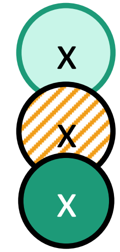
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**





Efficient implementation

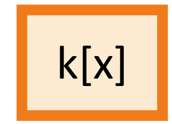
Every vertex has a key and a parent



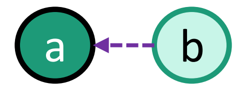
Can't reach x yet

x is "active"

Can reach x



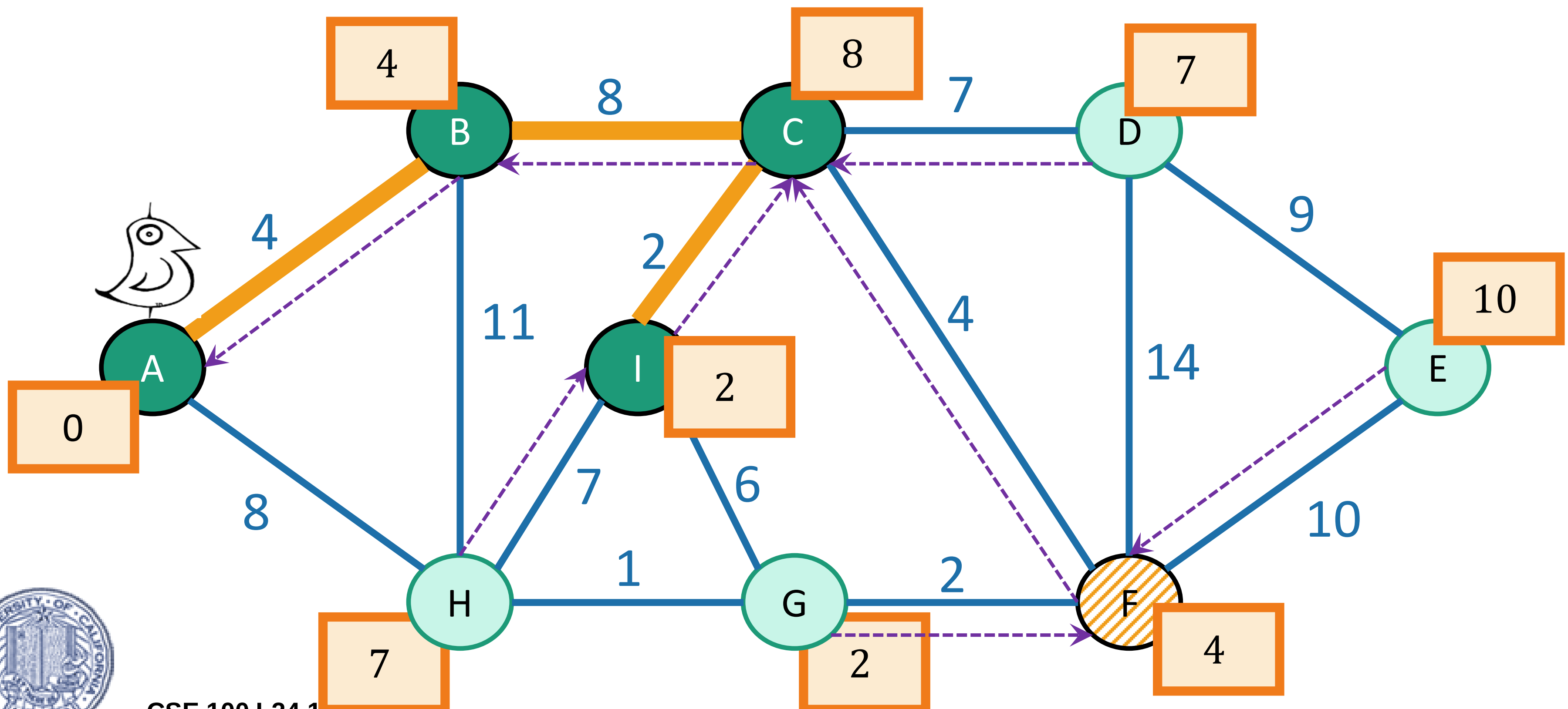
k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

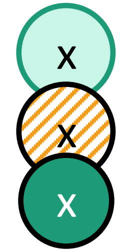


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

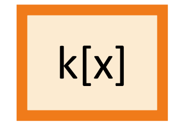
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



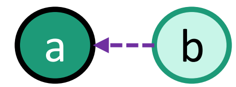
Can't reach x yet

x is "active"

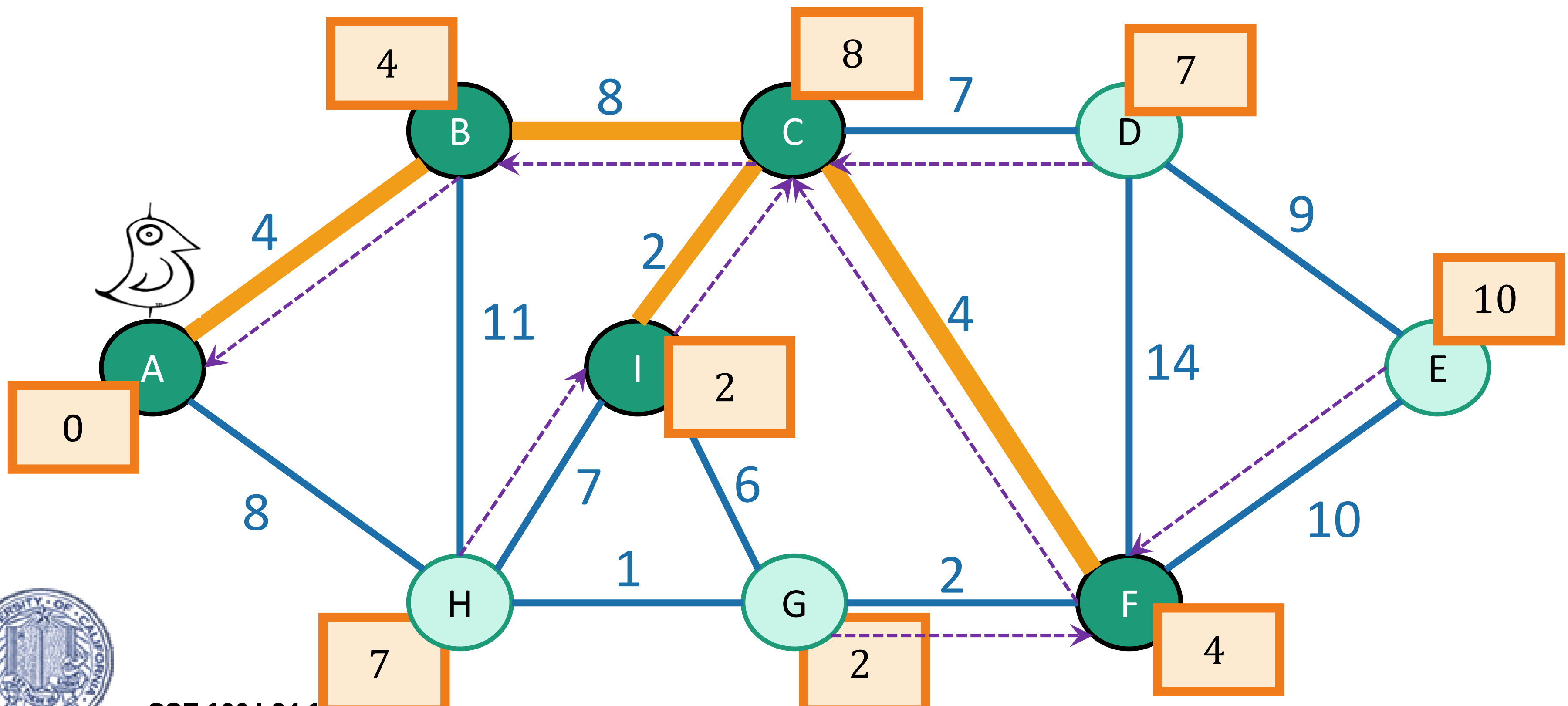
Can reach x



k[x] is the distance of x from the growing tree

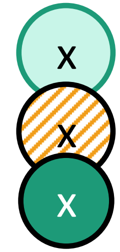


p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

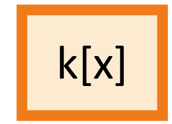
Every vertex has a key and a parent



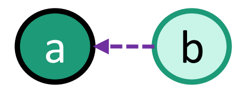
Can't reach x yet

x is "active"

Can reach x



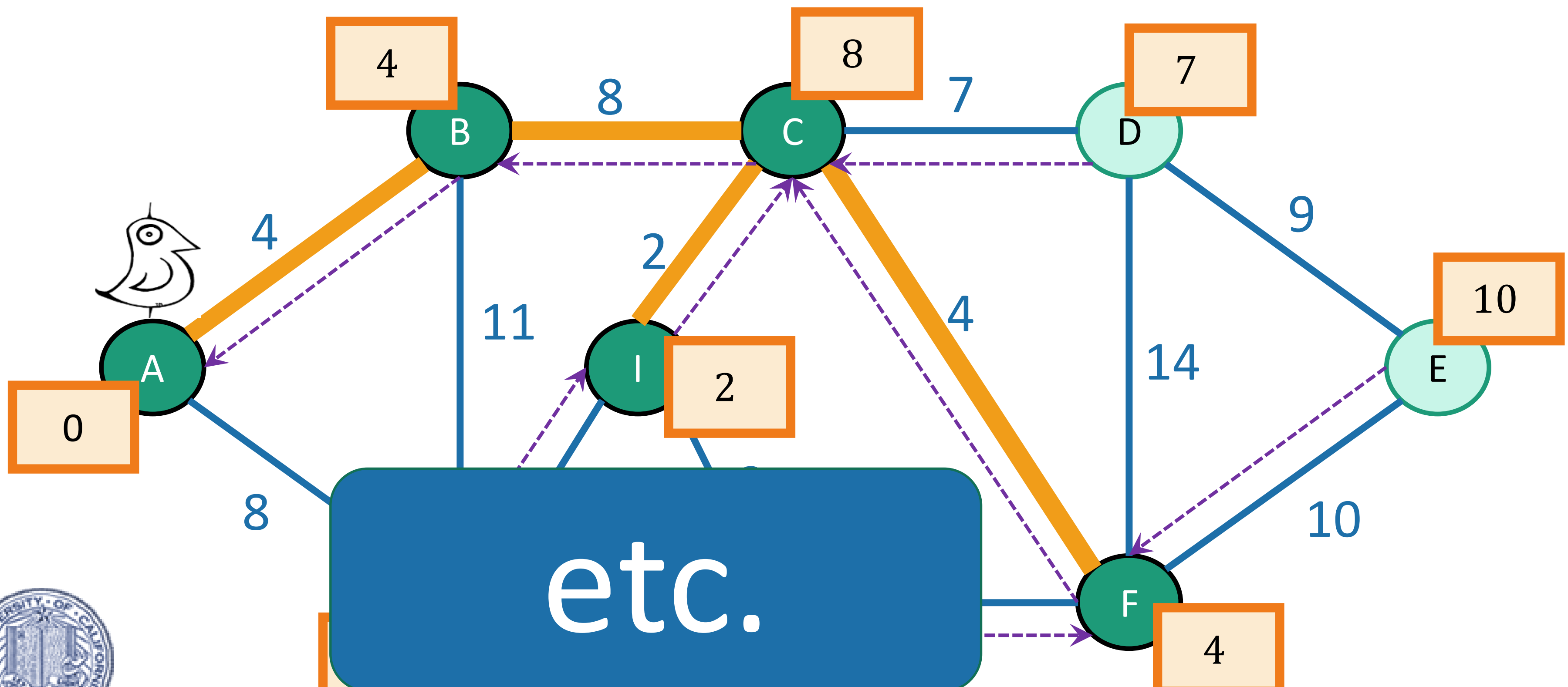
k[x] is the distance of x from the growing tree



p[b] = a, meaning that a was the vertex that k[b] comes from.

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

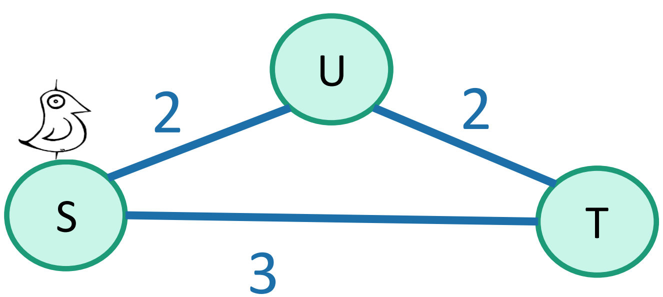


This should look pretty familiar

- Very similar to Dijkstra's algorithm!
- **Differences:**
  1. Keep track of p[v] in order to return a tree at the end
     - But Dijkstra's can do that too, that's not a big difference.
  2. Instead of d[v] which we update by
     - d[v] = min(d[v], d[u] + w(u,v))

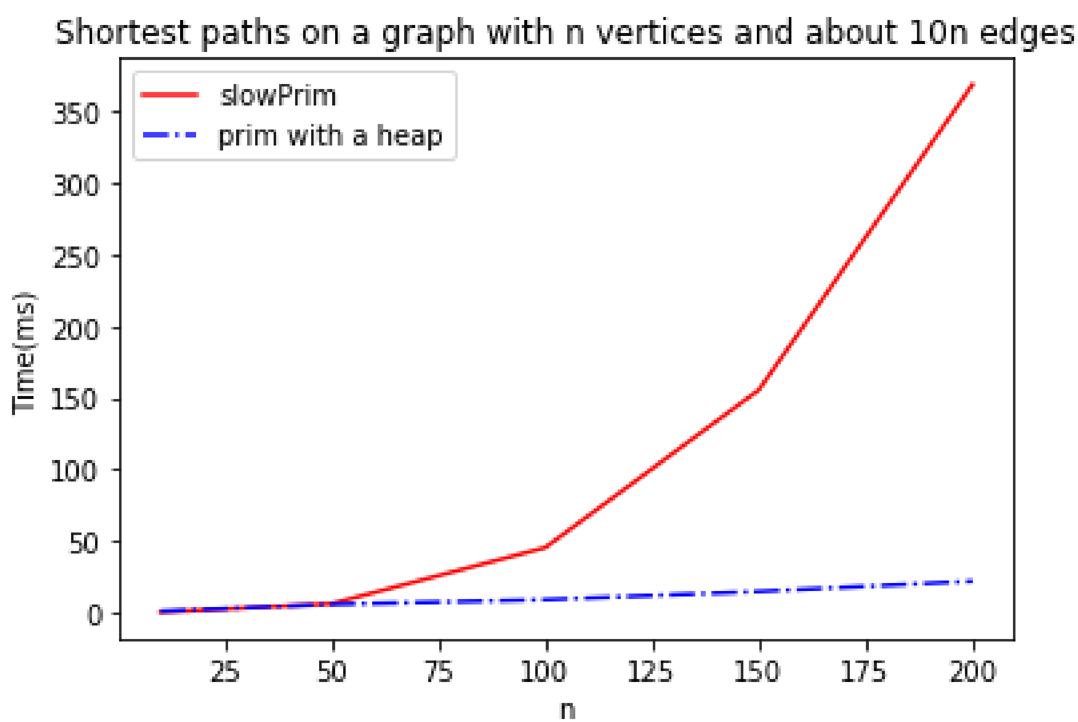
     we keep k[v] which we update by
     - k[v] = min(k[v], w(u,v))
- To see the difference, consider:

Thing 2 is the big difference.



One thing that is similar:
Running time

- Exactly the same as Dijkstra:
  - O(mlog(n)) using a Red-Black tree as a priority queue.
  - O(m + nlog(n)) amortized time if we use a Fibonacci Heap*.



*See Lecture 20, slides 10-12. Also CLRS Ch. 17 and 19.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
     - **Yes!**

2. How do we actually implement this?
   - the pseudocode above says "slowPrim"…
     - **Implement it basically the same way we'd implement Dijkstra!**

What have we learned?

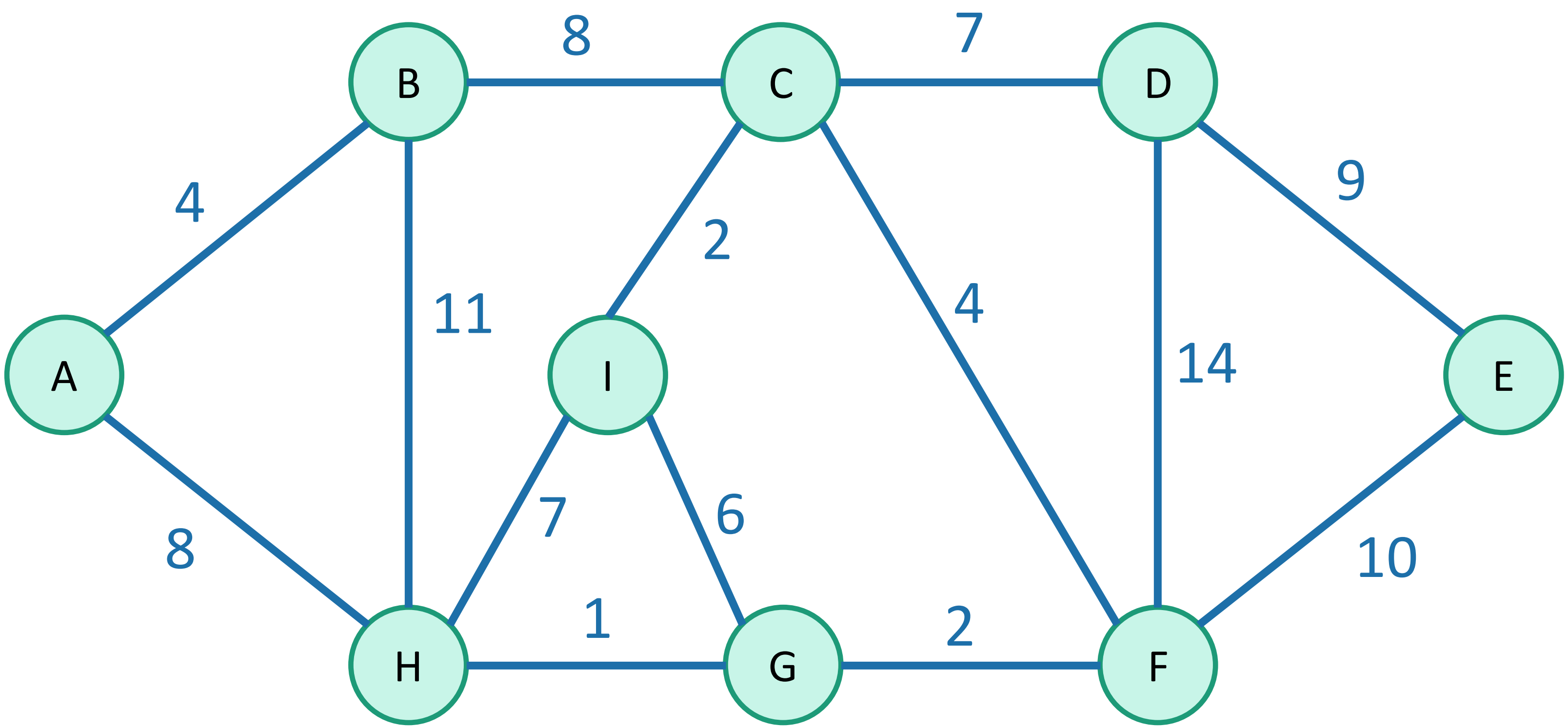
- Prim's algorithm greedily grows a tree
  - smells a lot like Dijkstra's algorithm
- It finds a Minimum Spanning Tree!
  - in time **O(mlog(n))** if we implement it with a Red-Black Tree.
  - In amortized time **O(m + nlog(n))** with a Fibonacci heap.
- To prove it worked, we followed the same recipe for greedy algorithms we saw last time.
  - Show that, at every step, we **don't rule out success.**

That’s not the only greedy algorithm for MST!

That's not the only greedy algorithm

what if we just always take the cheapest edge?

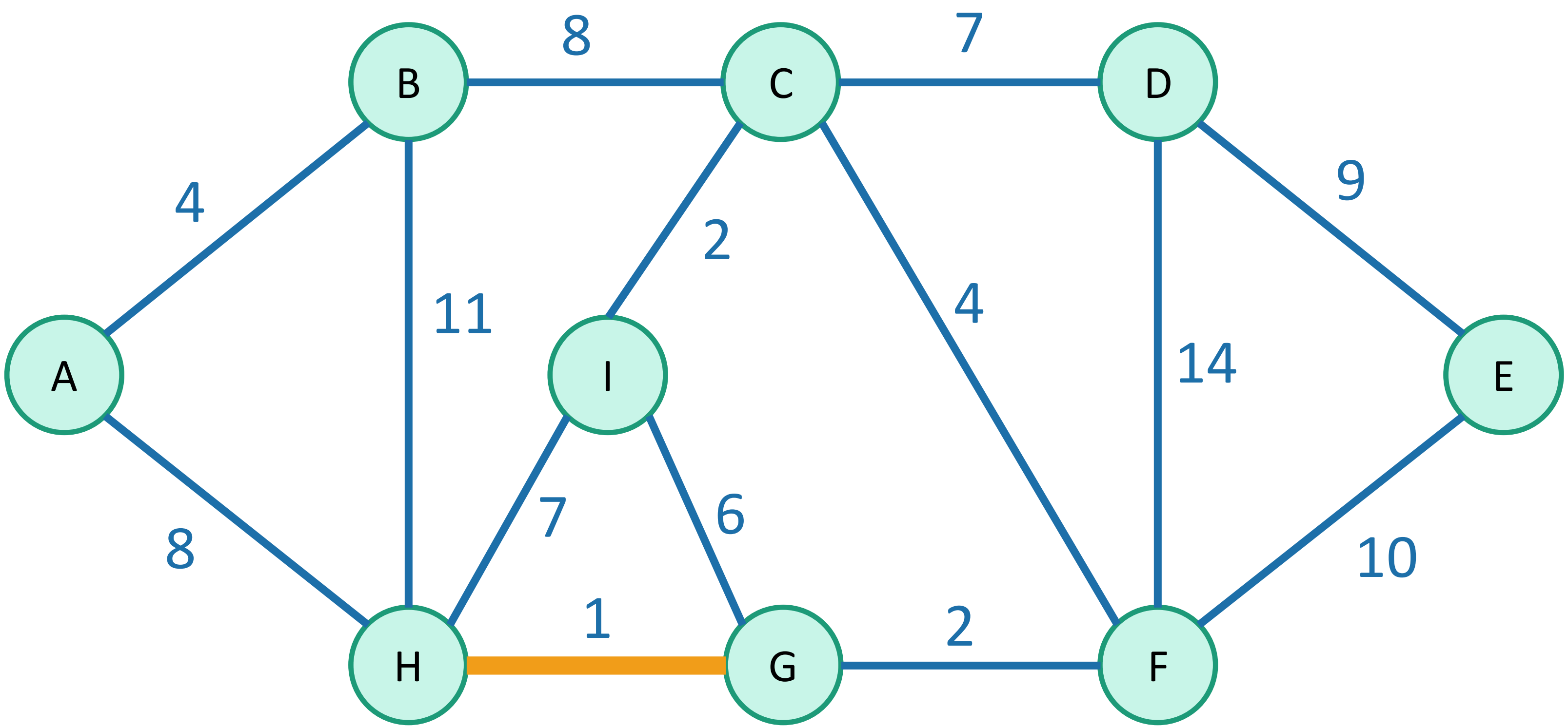
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

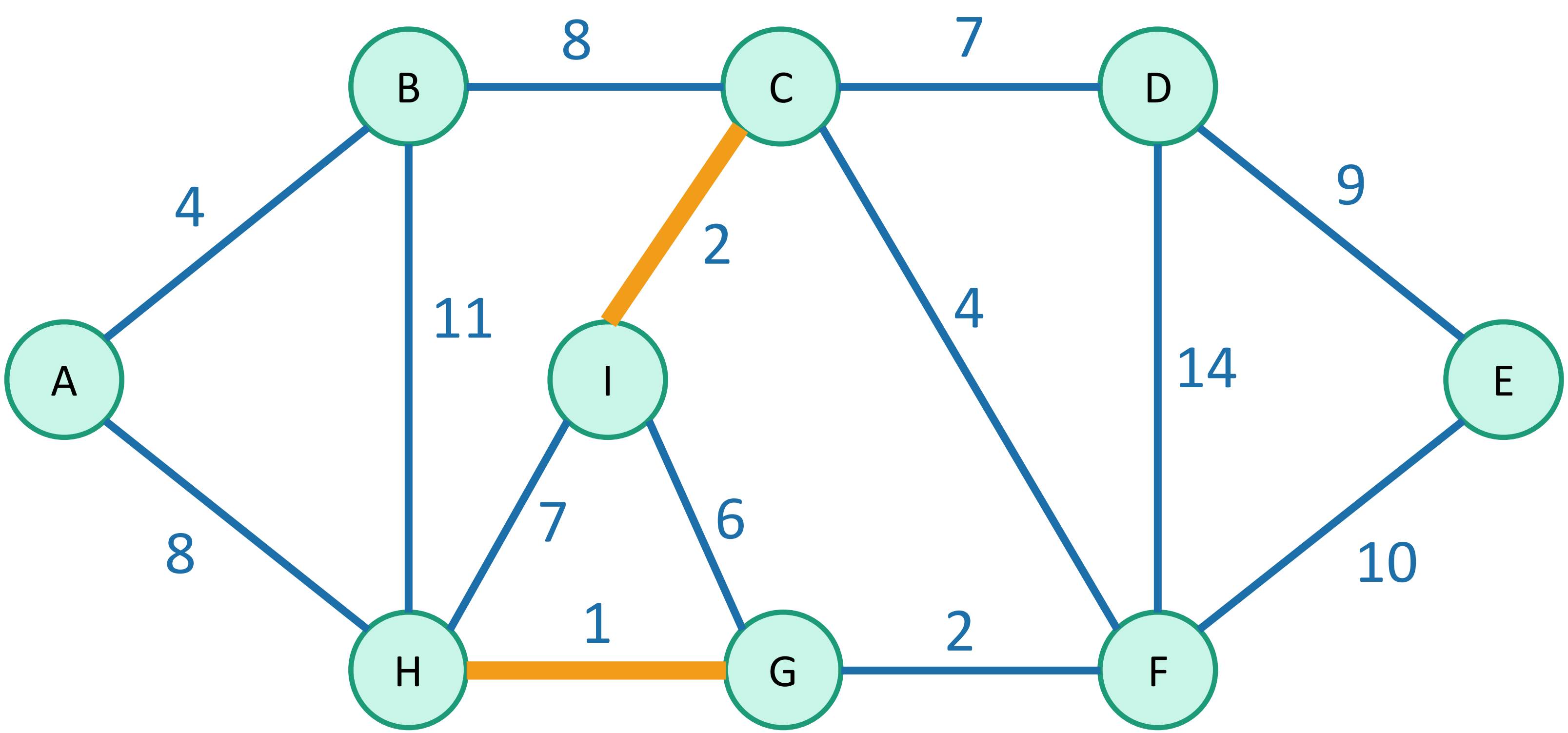
whether or not it's connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

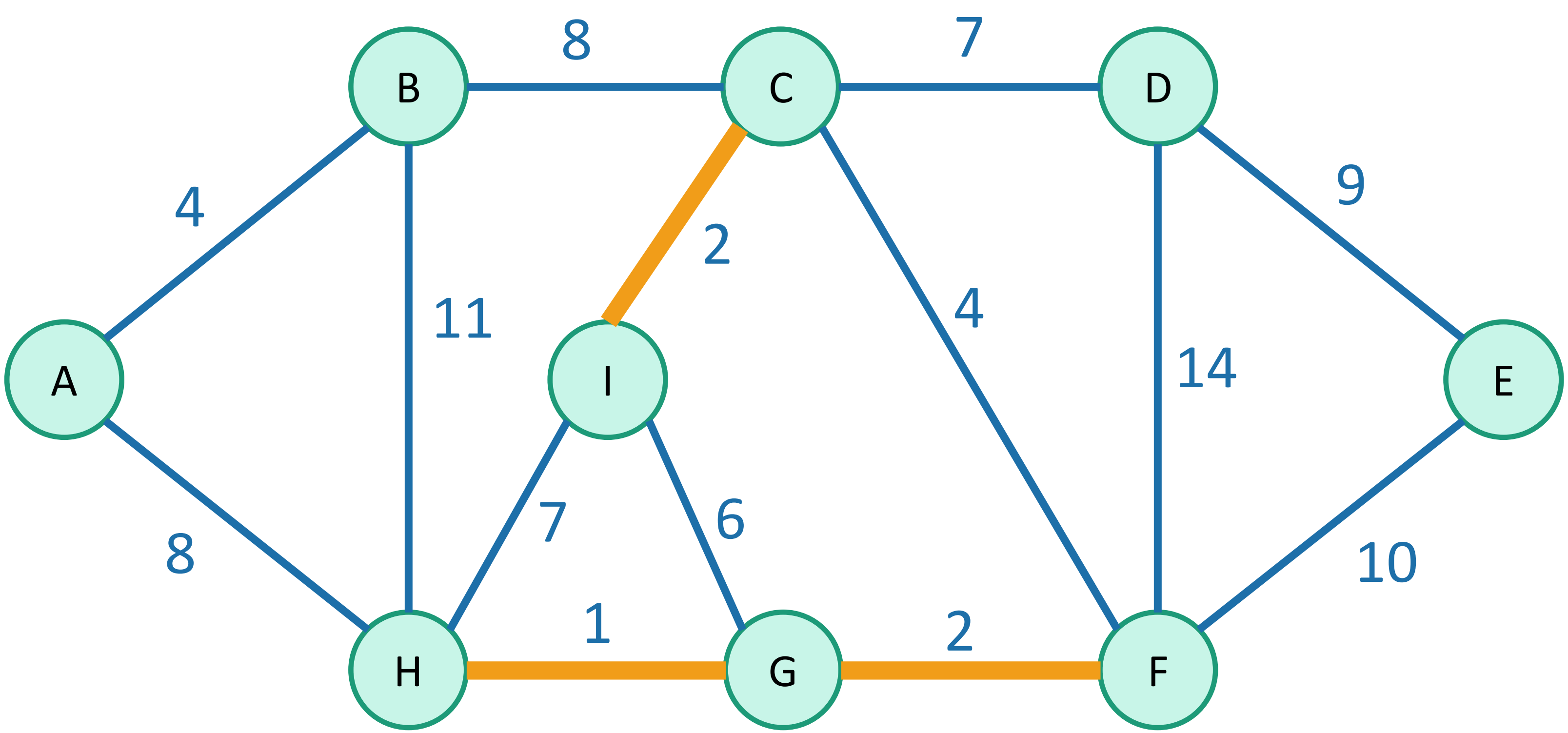
whether or not it’s connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

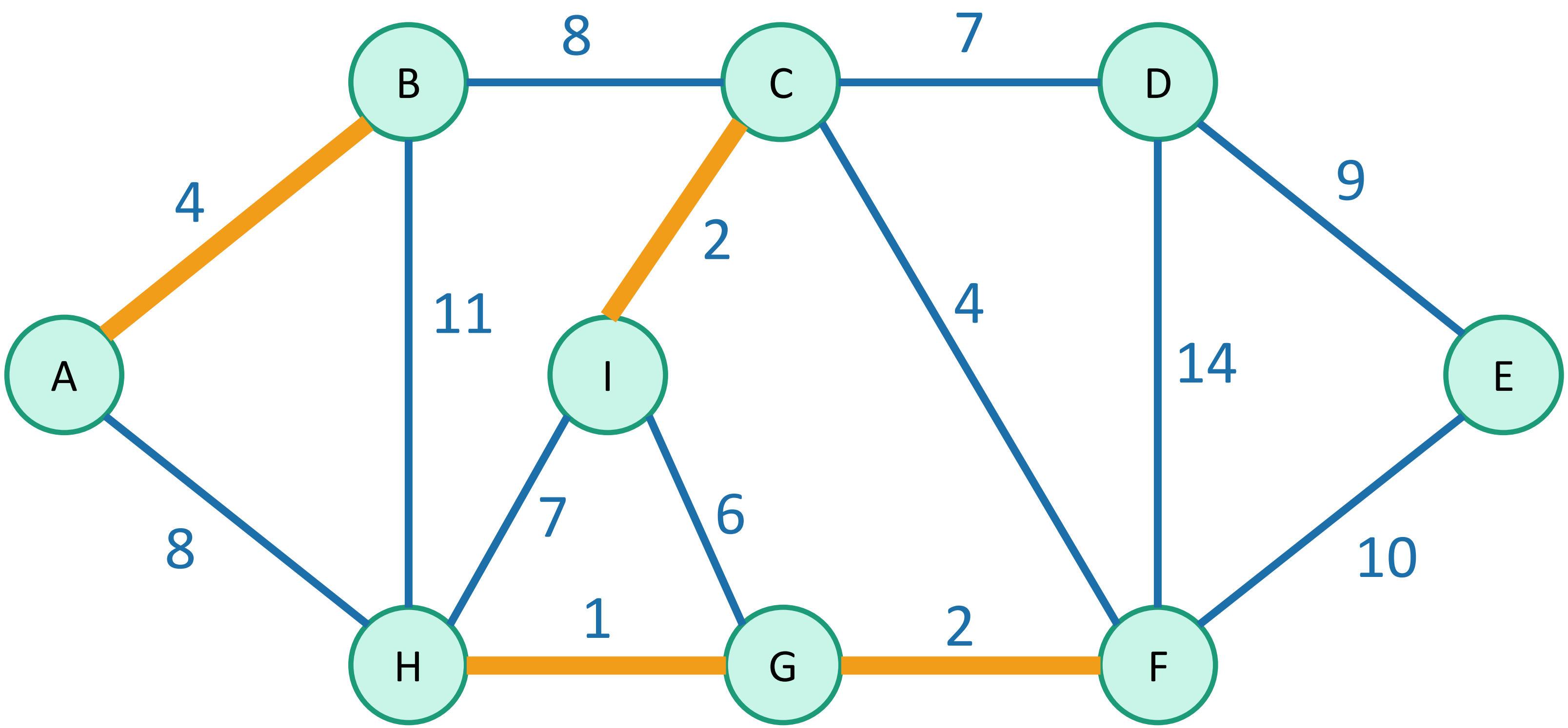
whether or not it’s connected to what we have so far?



That’s not the only greedy algorithm

what if we just always take the cheapest edge?

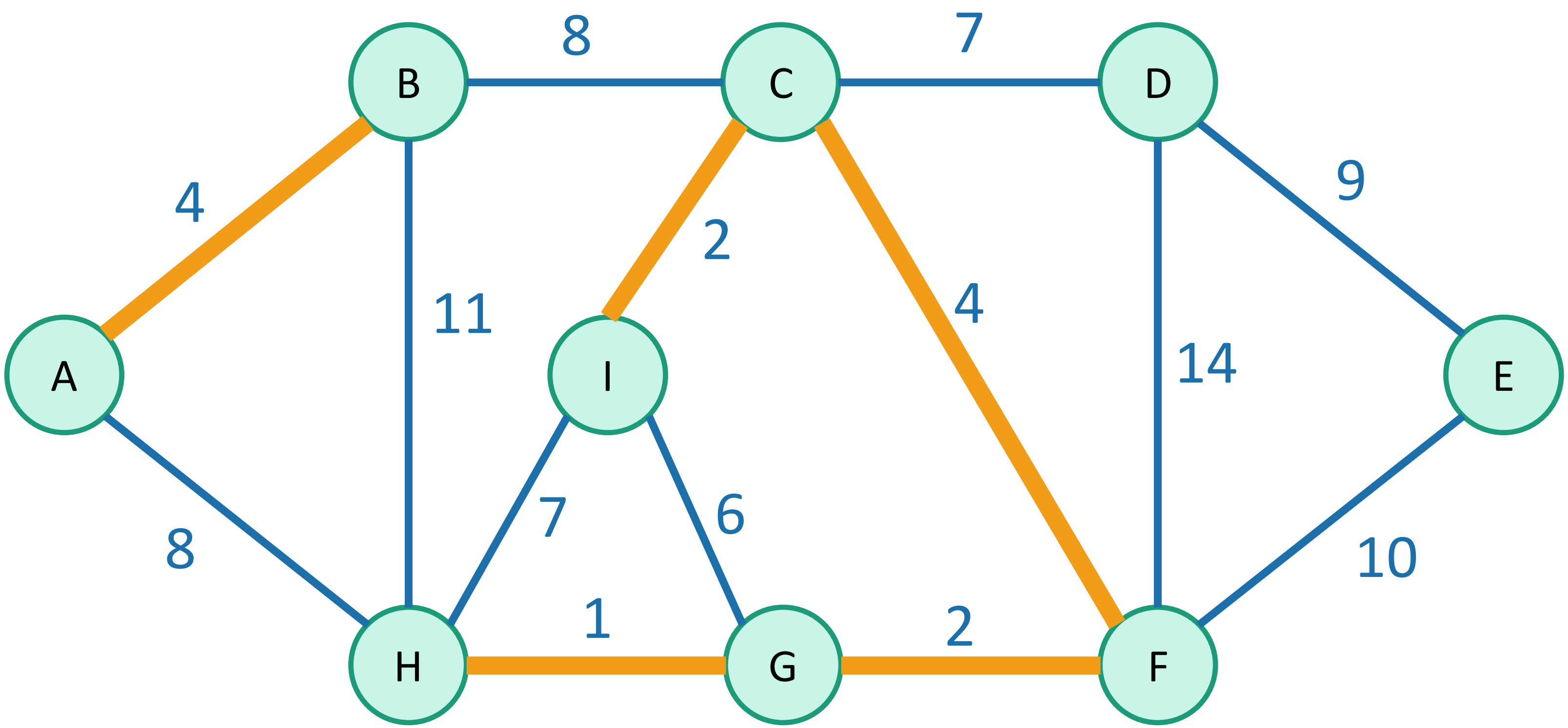
whether or not it’s connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

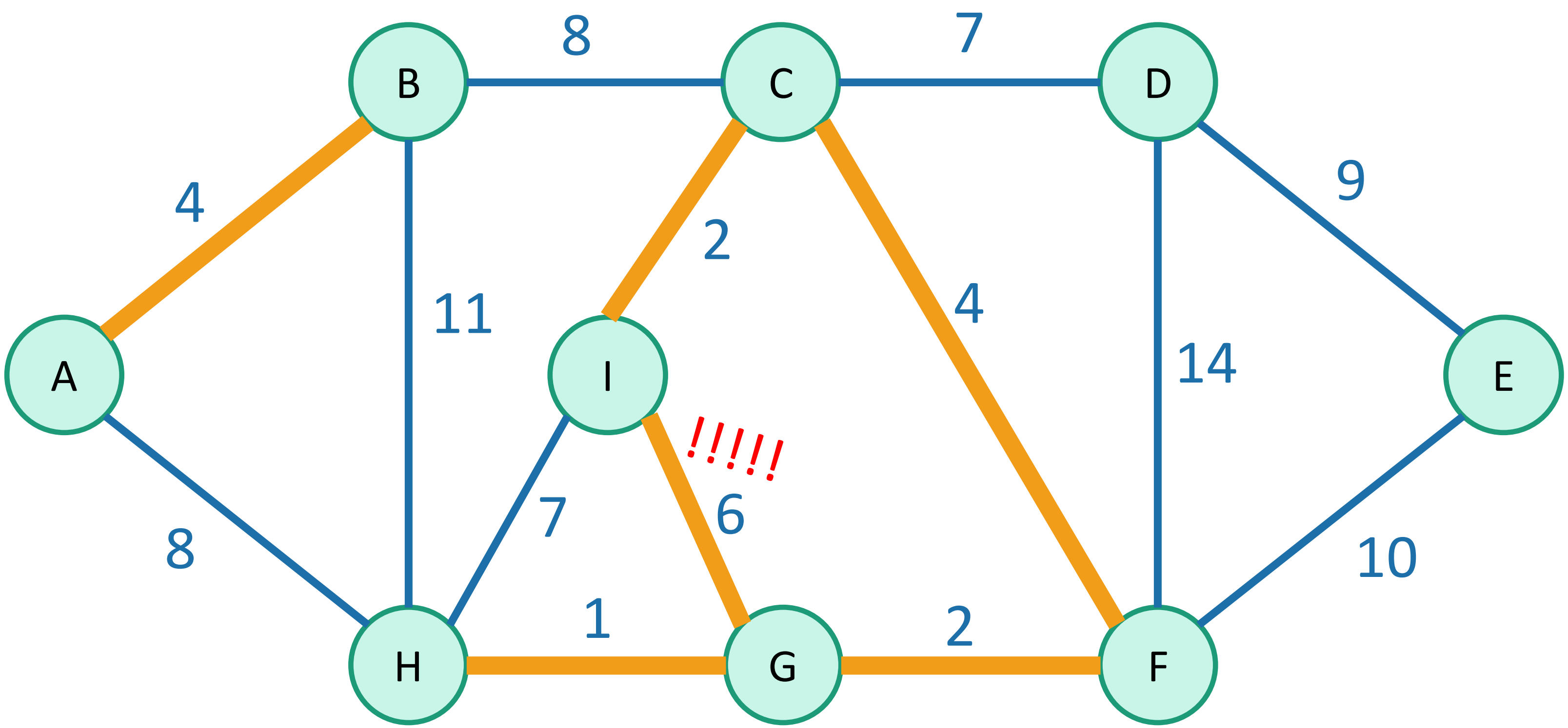


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

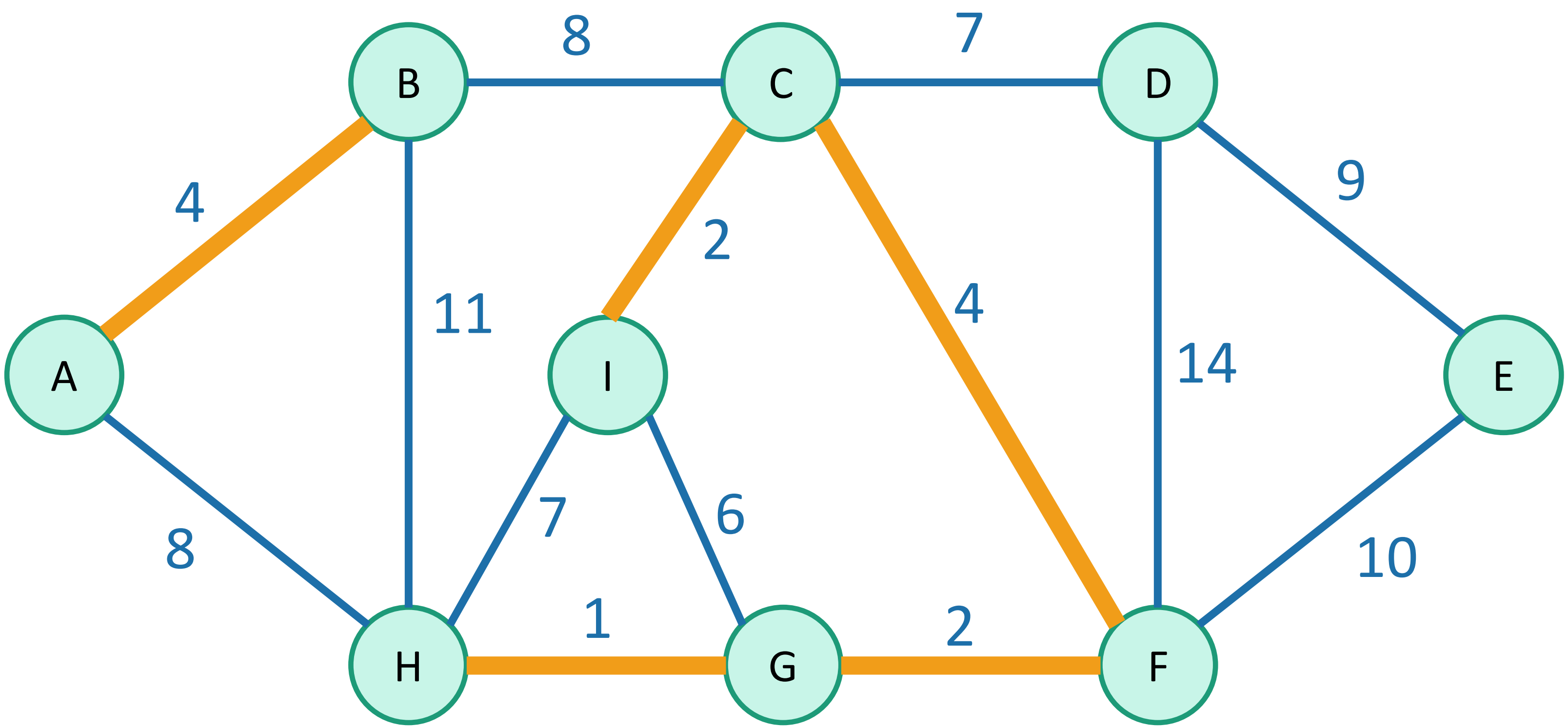
That won't cause a cycle



That’s not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it’s connected to what we have so far?

That won’t cause a cycle

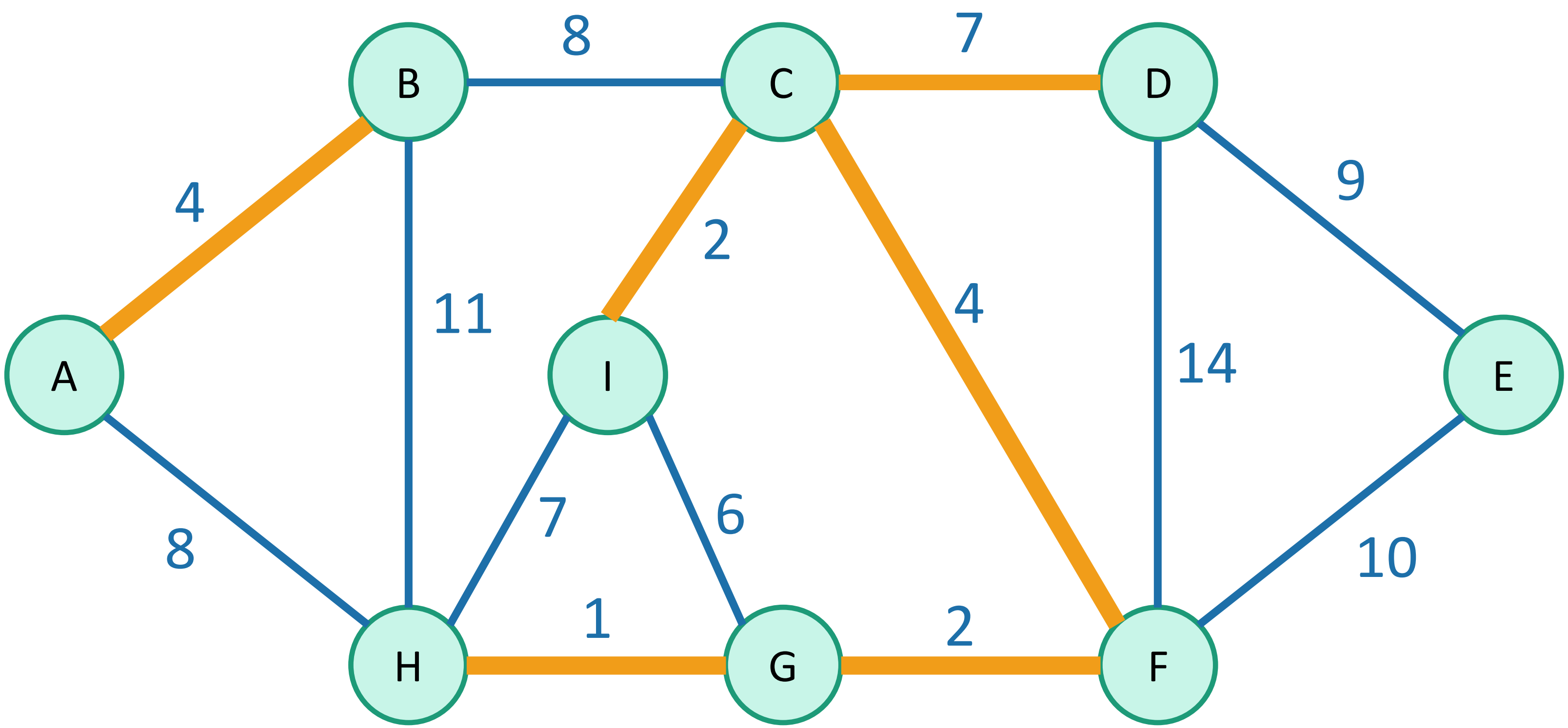


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

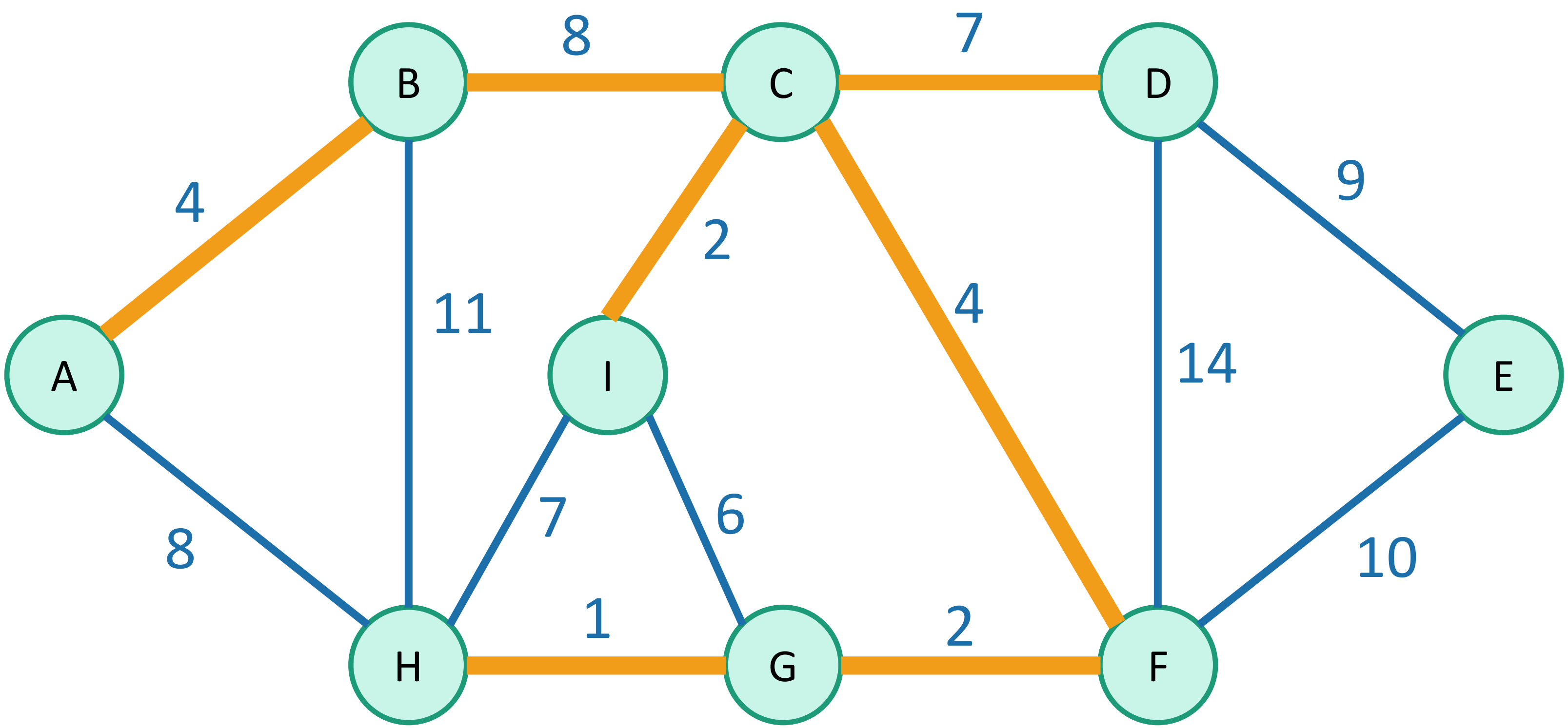


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

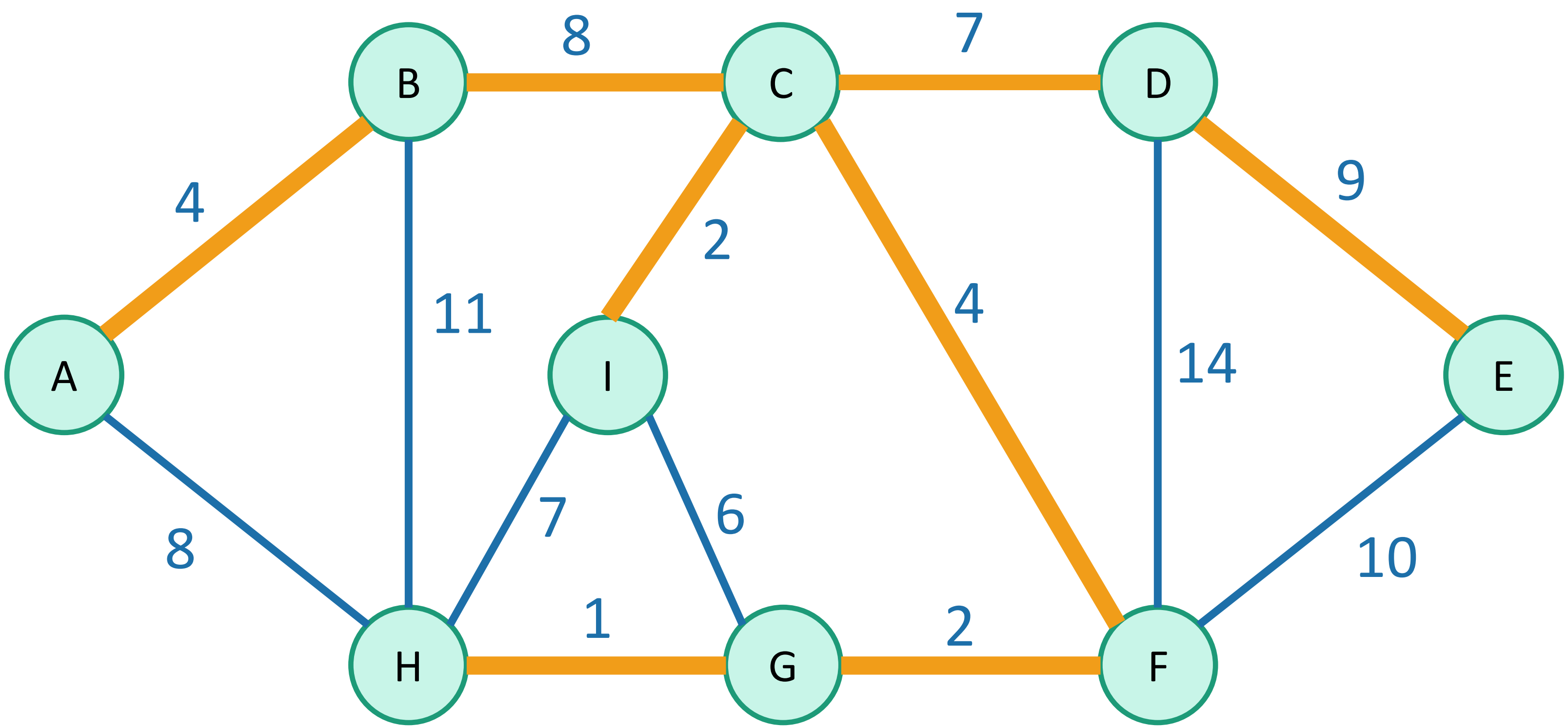
That won't cause a cycle



That’s not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it’s connected to what we have so far?

That won’t
cause a cycle

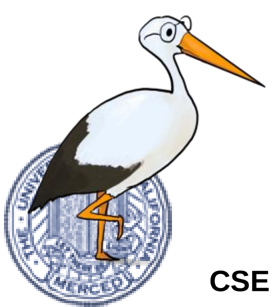


We've discovered
Kruskal's algorithm!

- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won't cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?



How **would** you figure out if added e would make a cycle in this algorithm?

<u>Naively, the running time is ???:</u>

- For each of m iterations of the for loop:
  - Check if adding e would cause a cycle…

Two questions

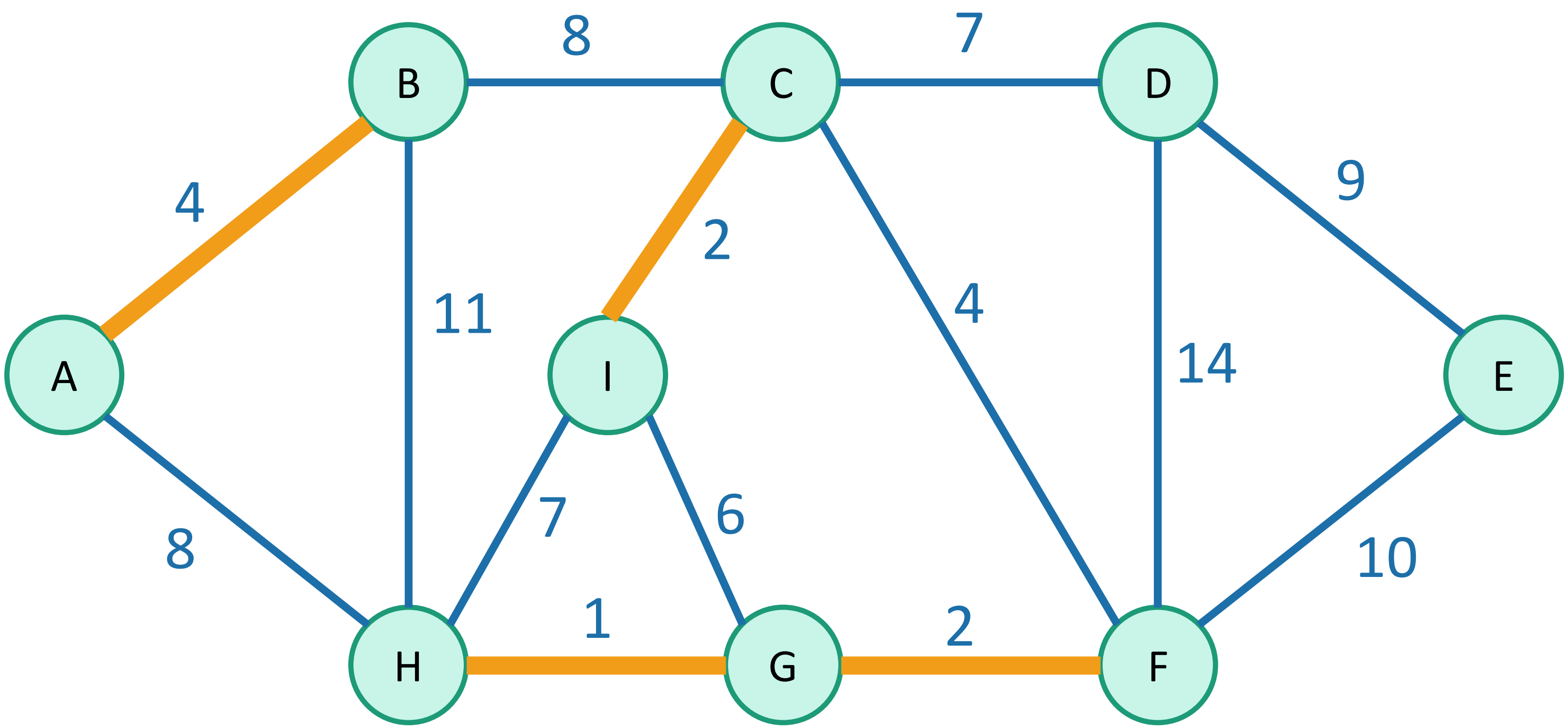
1. Does it work?
   - That is, does it actually return a MST?

2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…

Let’s do this one first

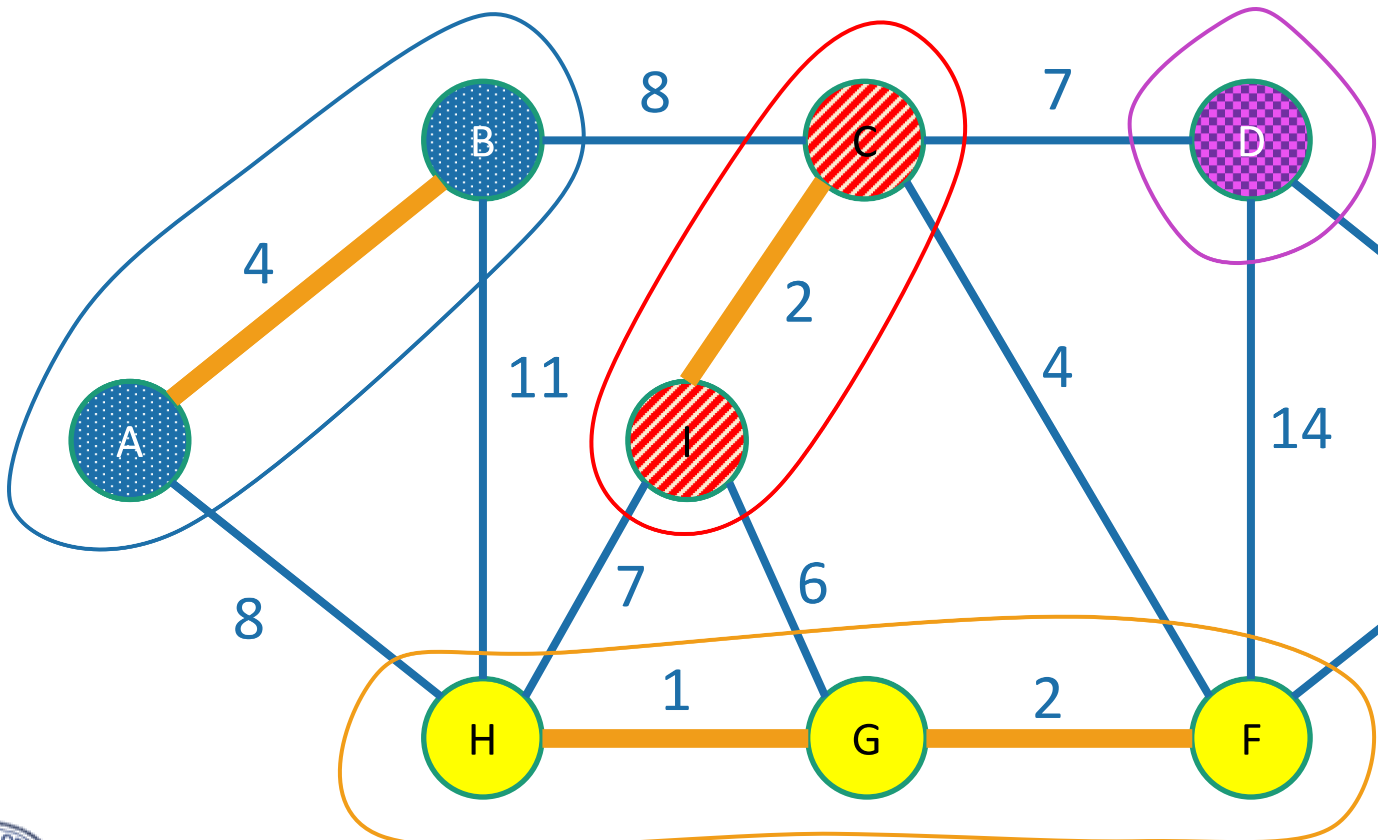
At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

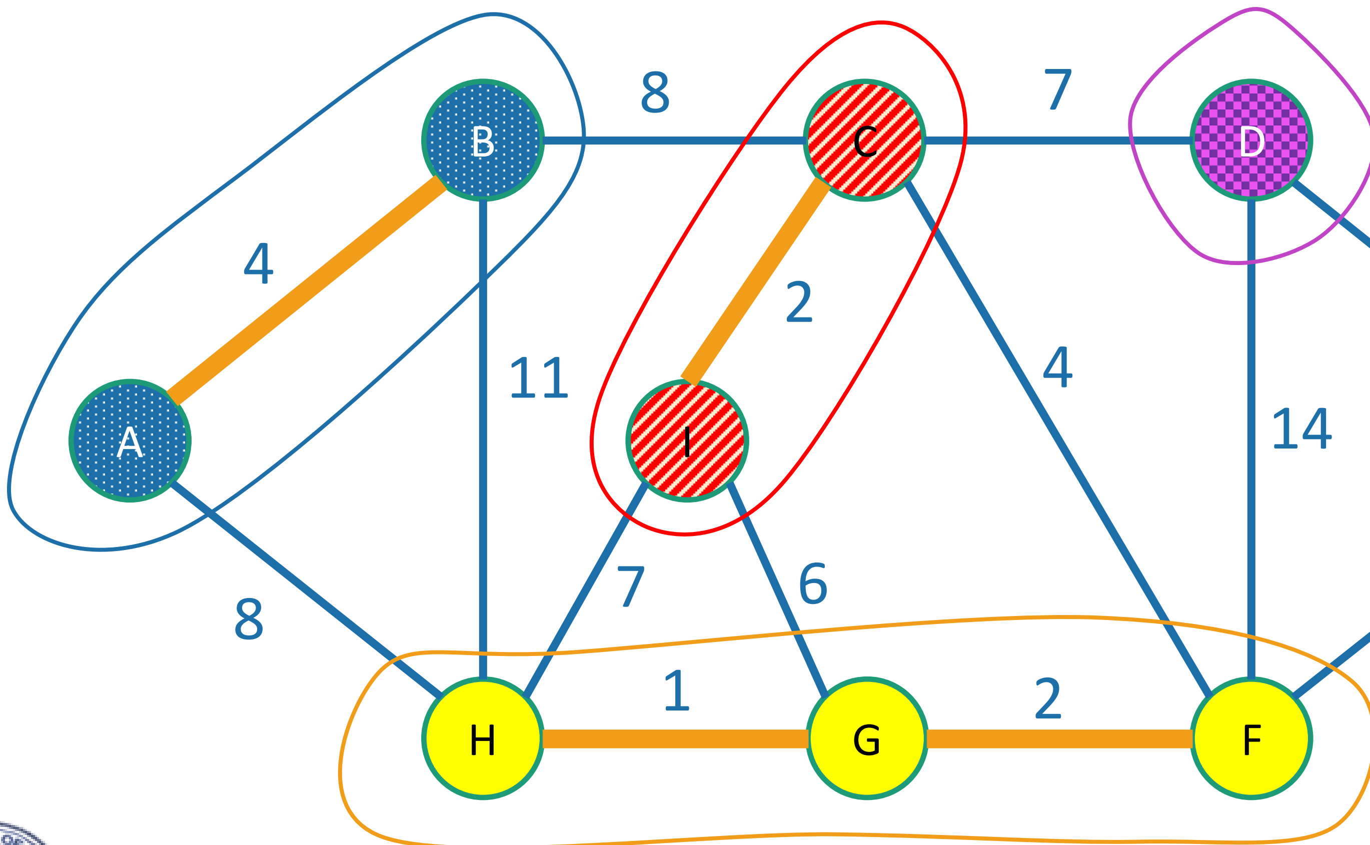
A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

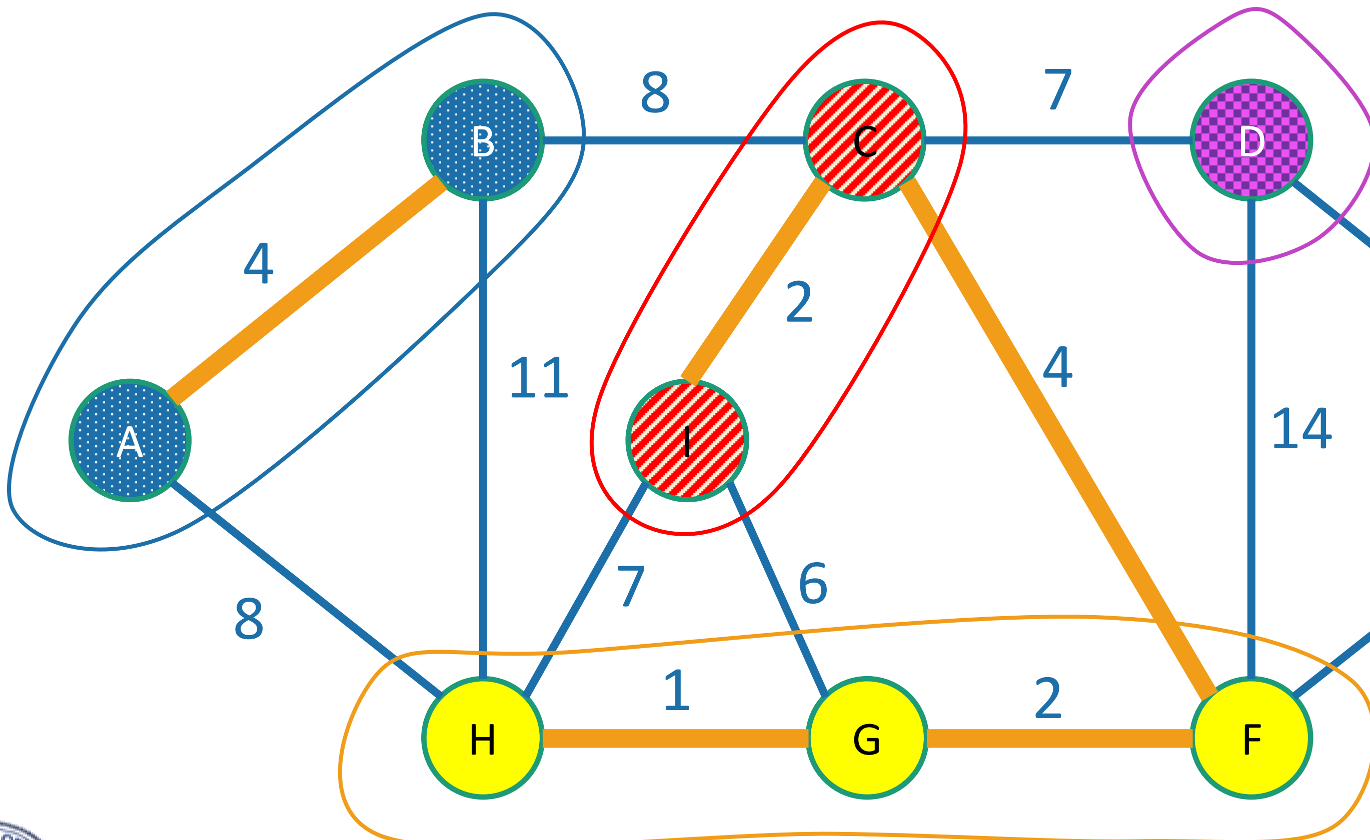
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

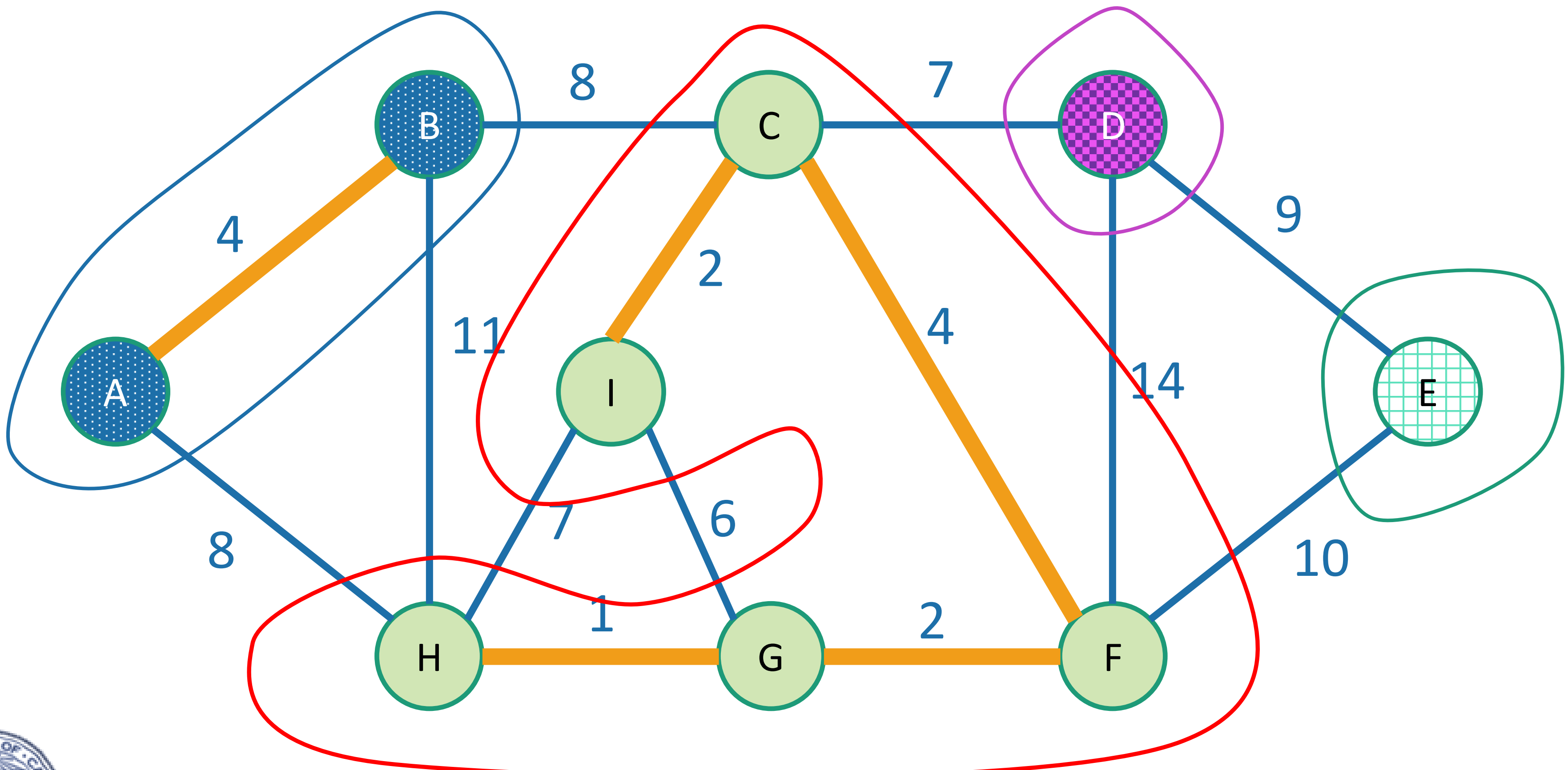
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

When we add an edge, we merge two trees:



We never add an edge within a tree since that would create a cycle.

Keep the trees in a special data structure

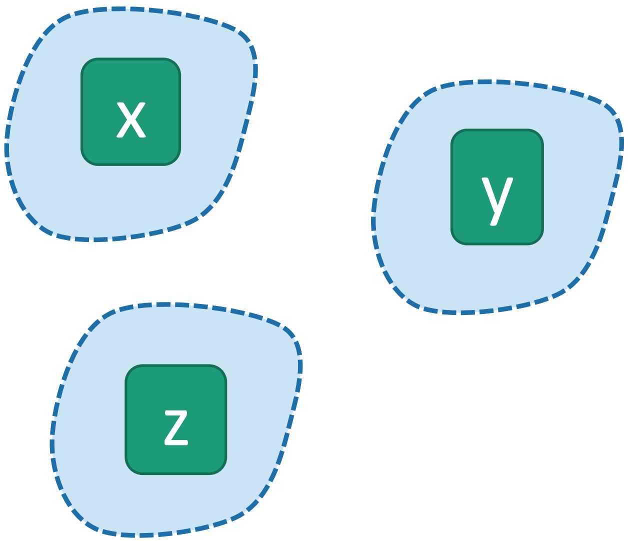
"treehouse"?

Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```

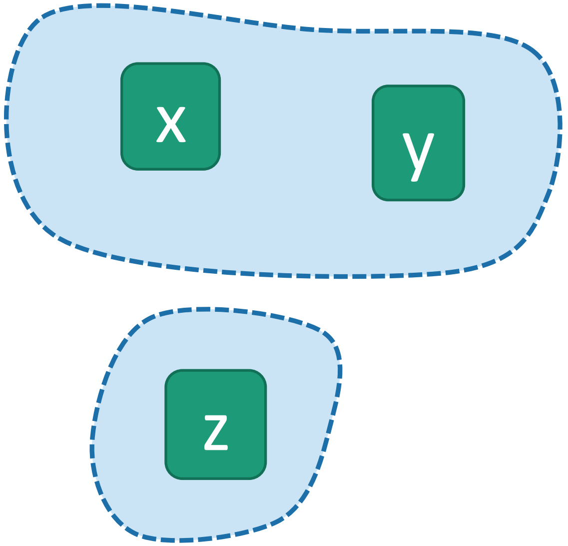


Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```



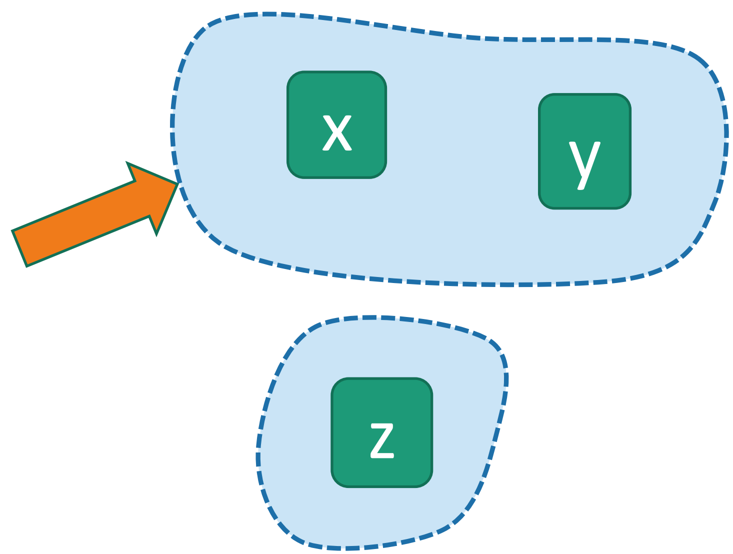
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```

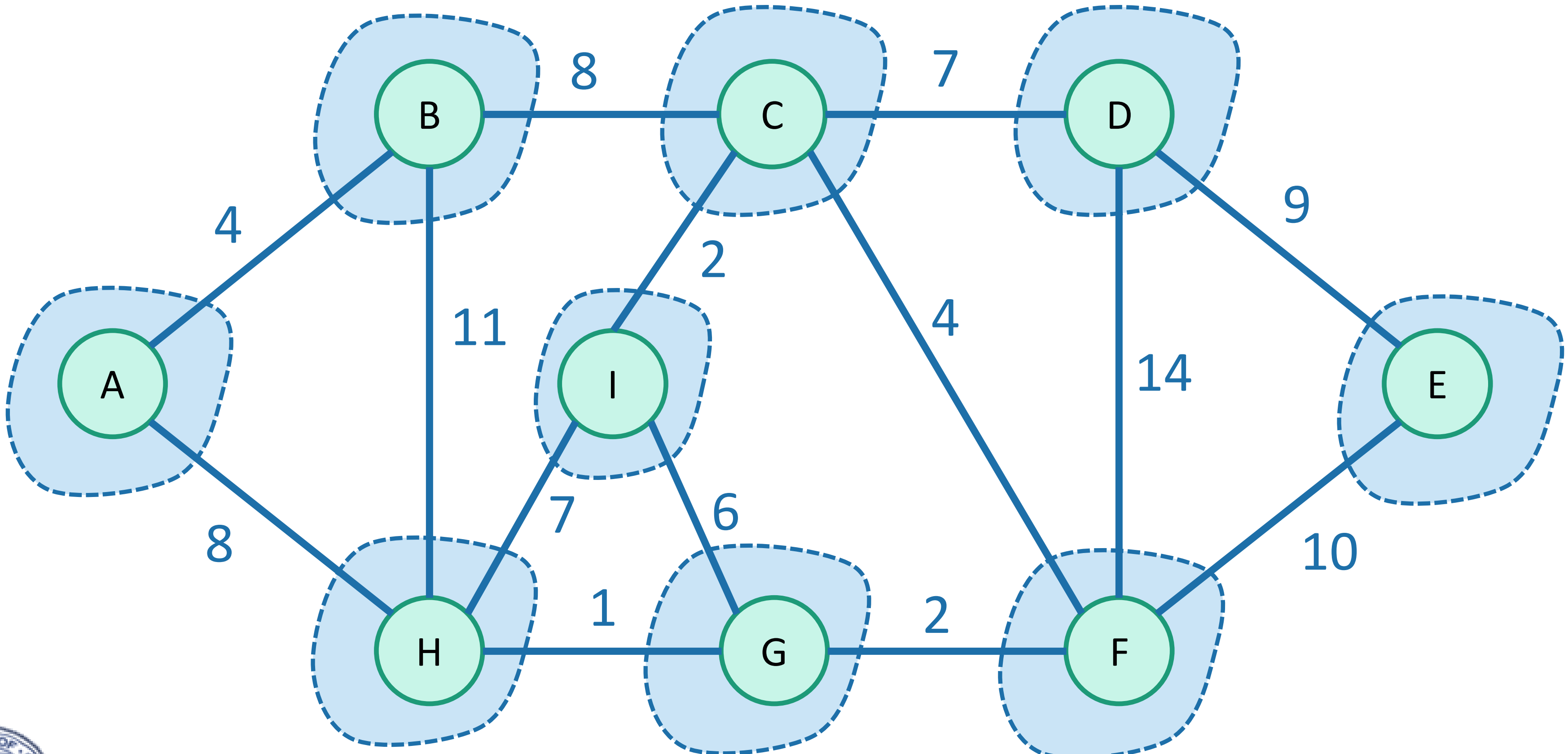


Kruskal pseudo-code

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

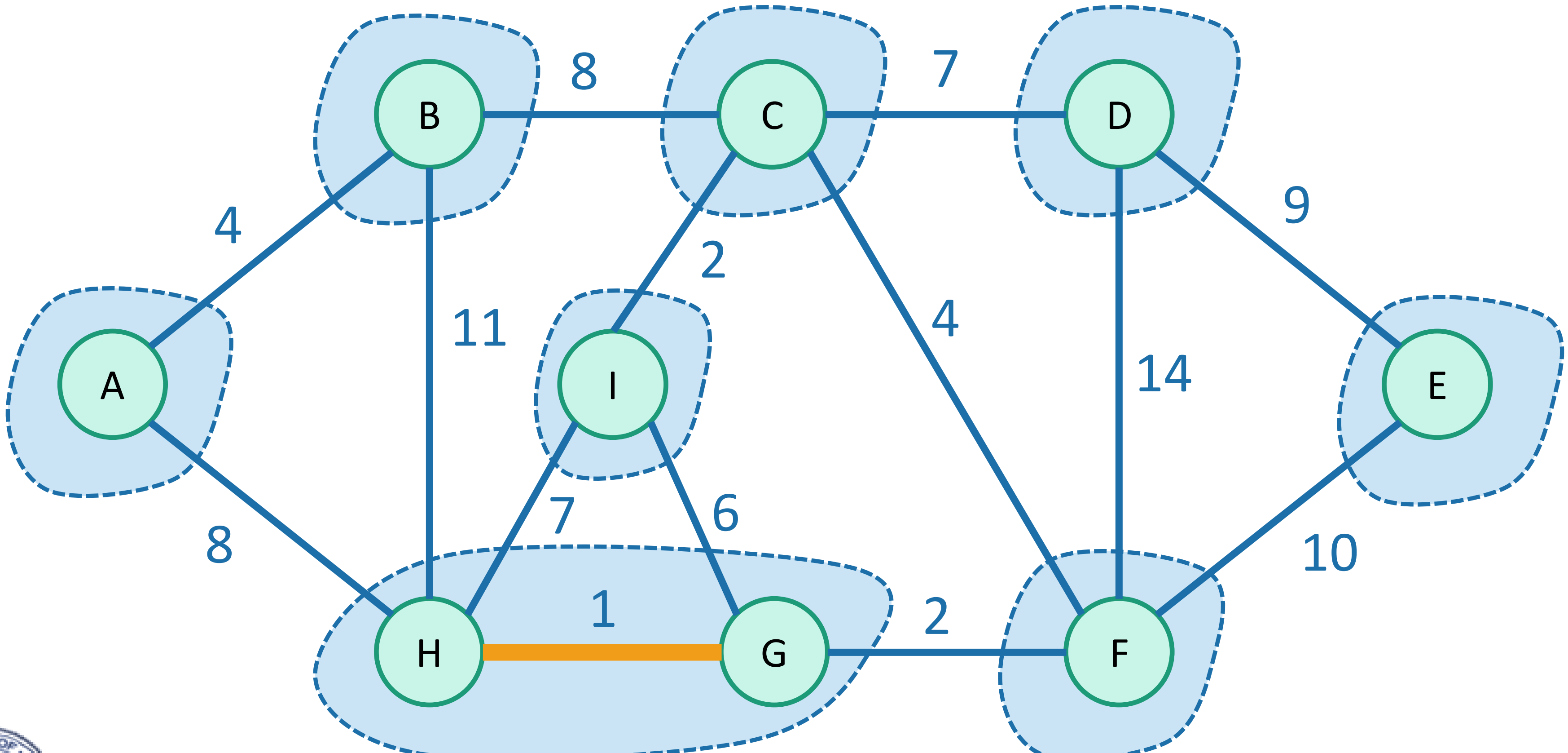
Once more…

To start, every vertex is in its own tree.



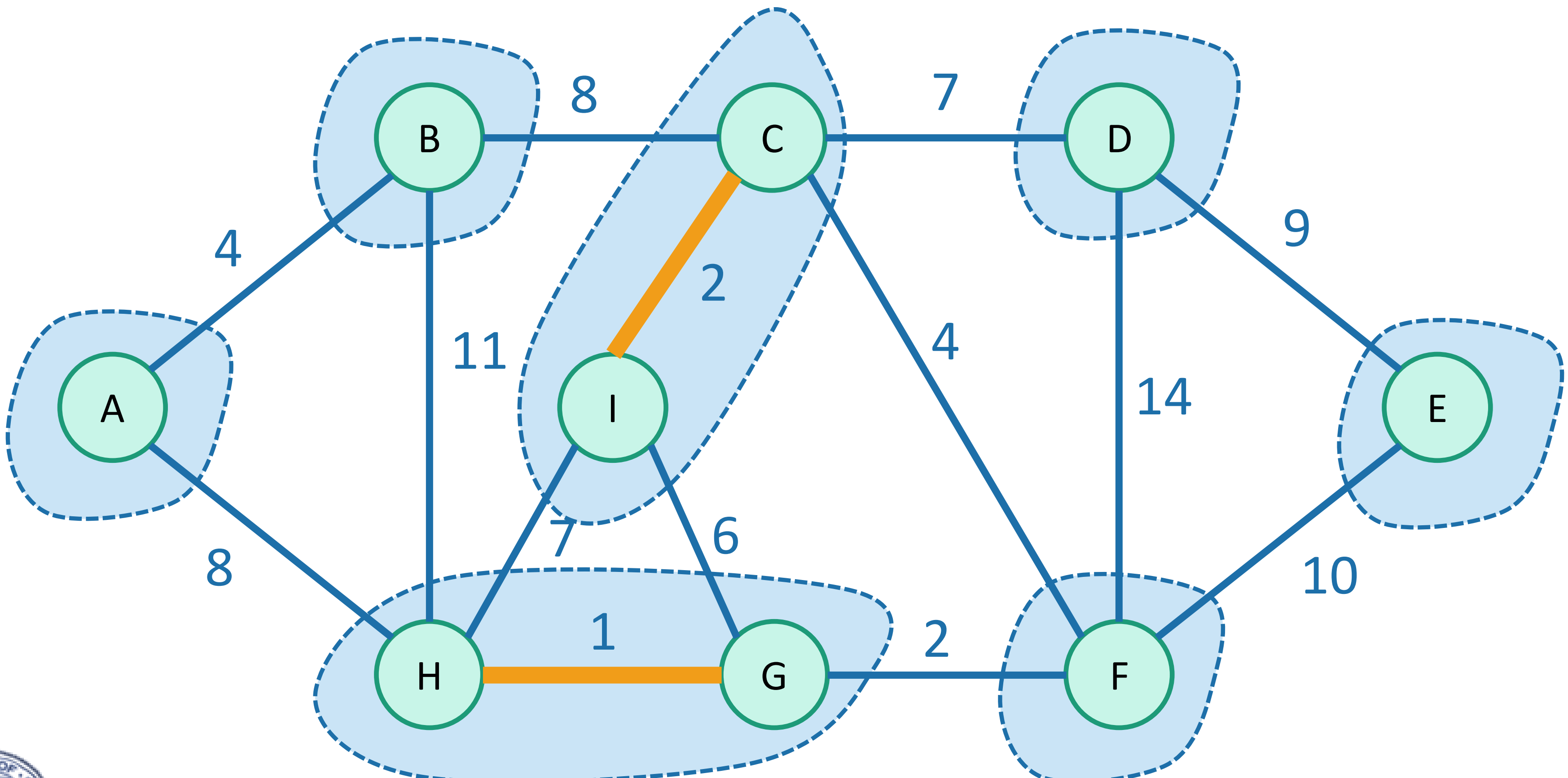
Once more…

Then start merging.



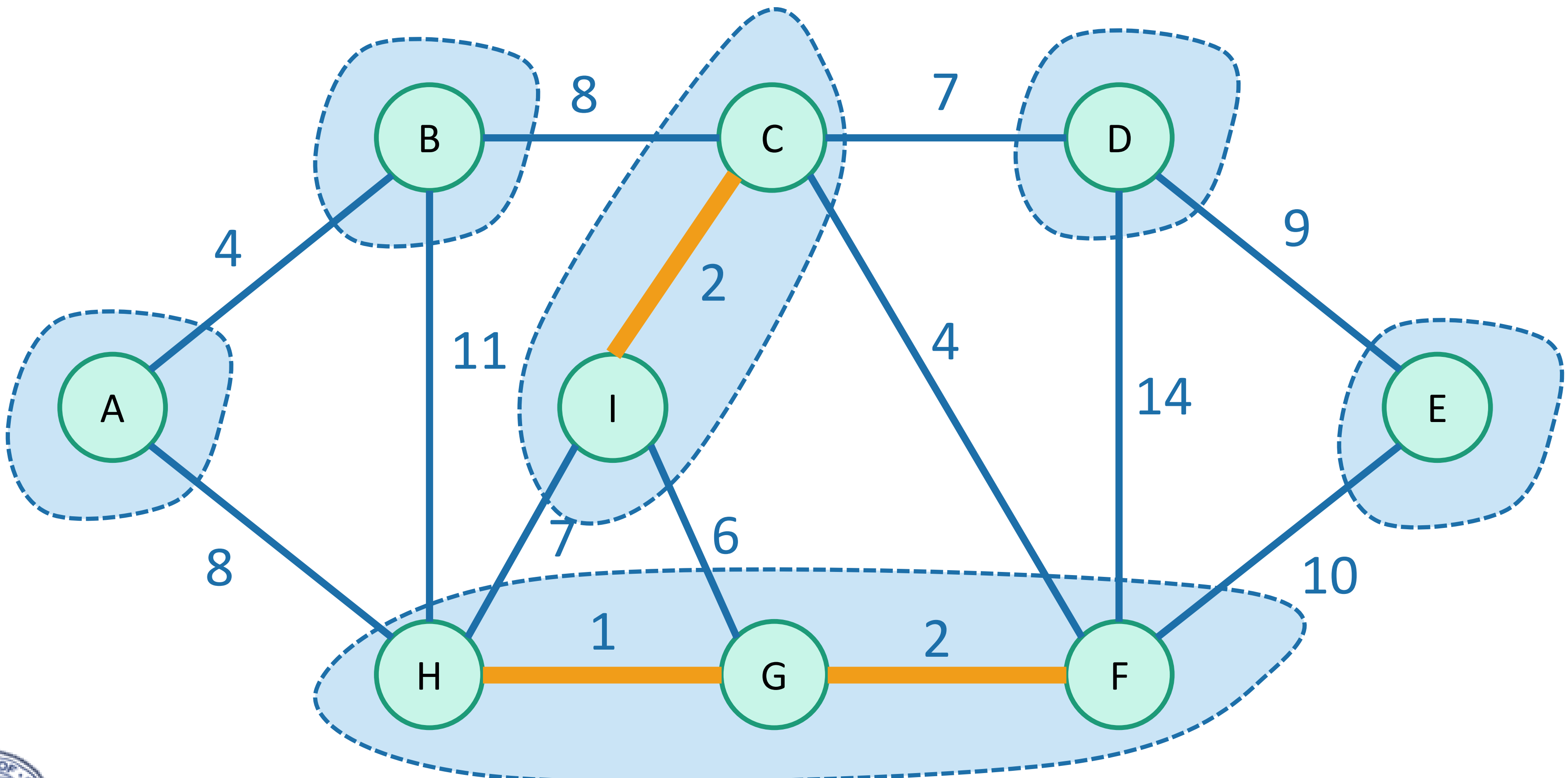
Once more…

Then start merging.



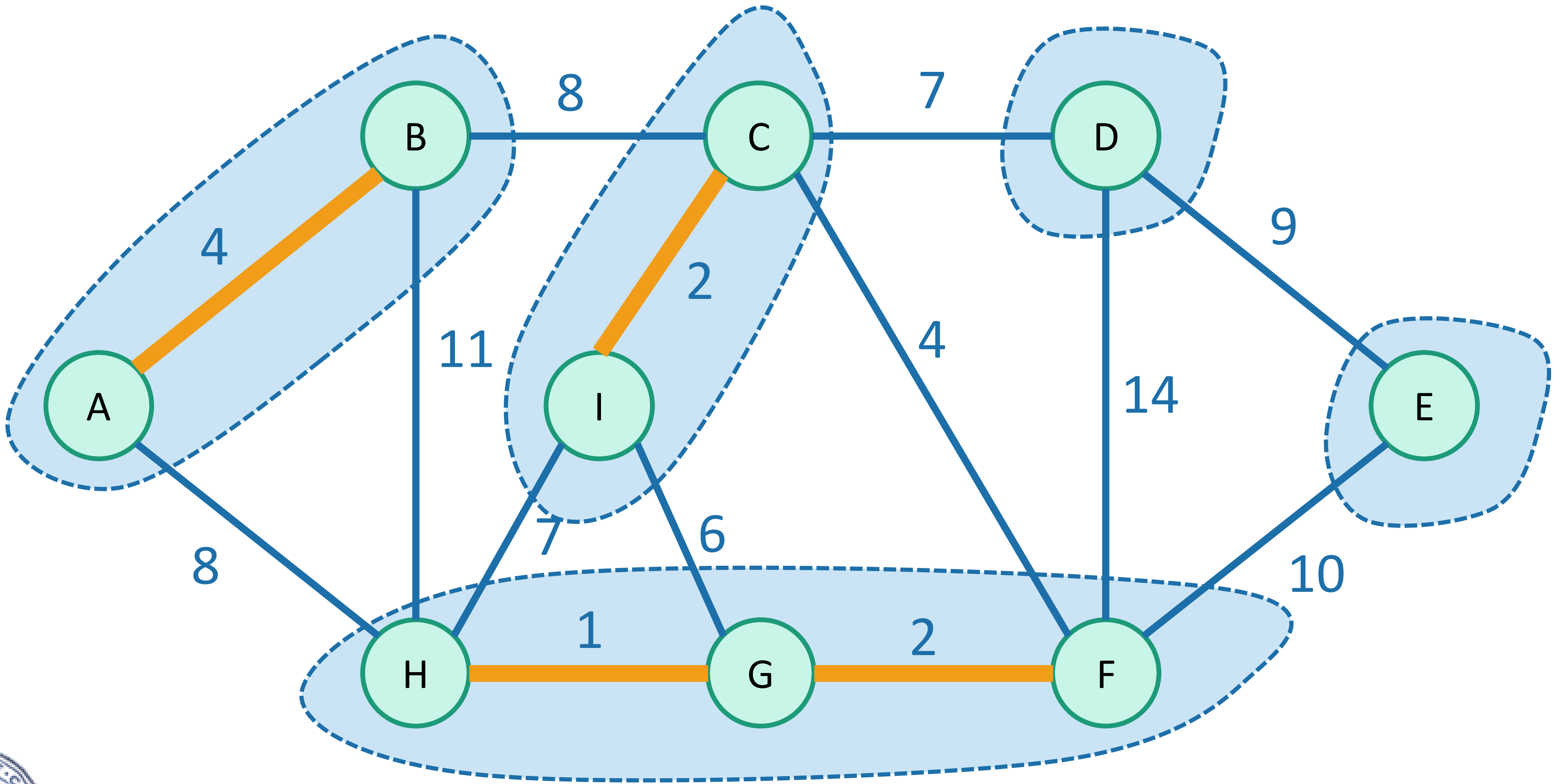
Once more…

Then start merging.



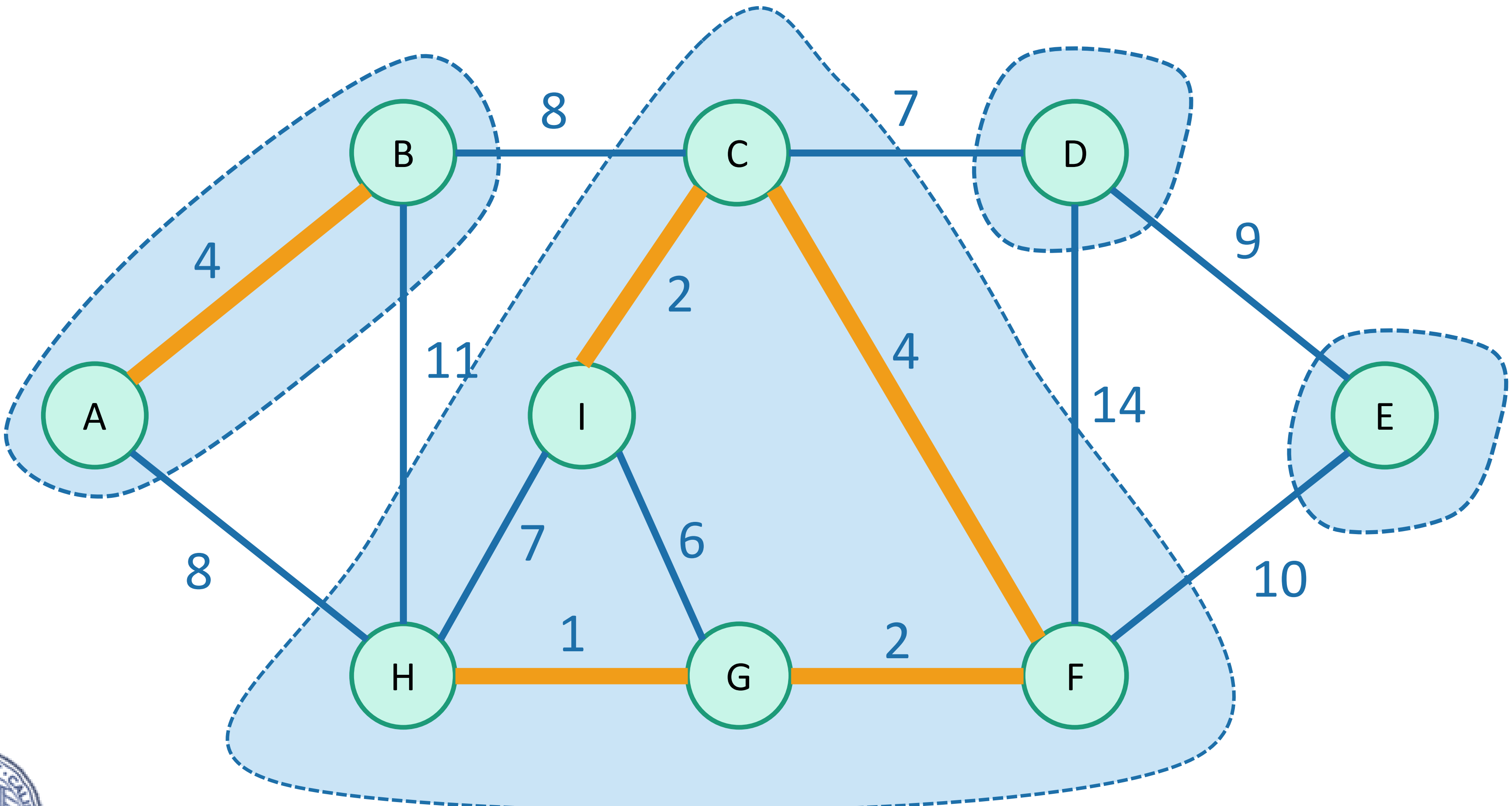
Once more…

Then start merging.



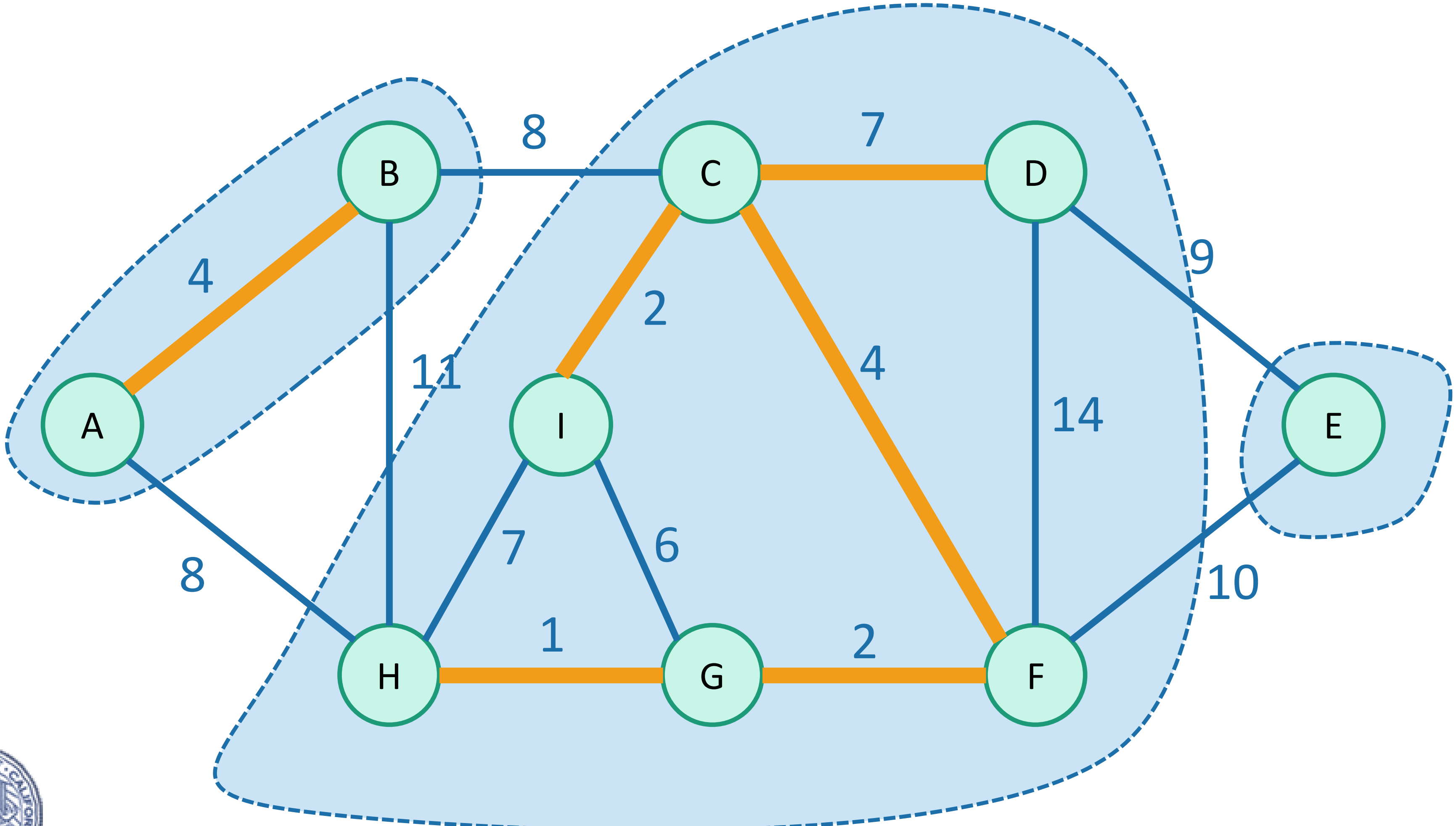
Once more…

Then start merging.



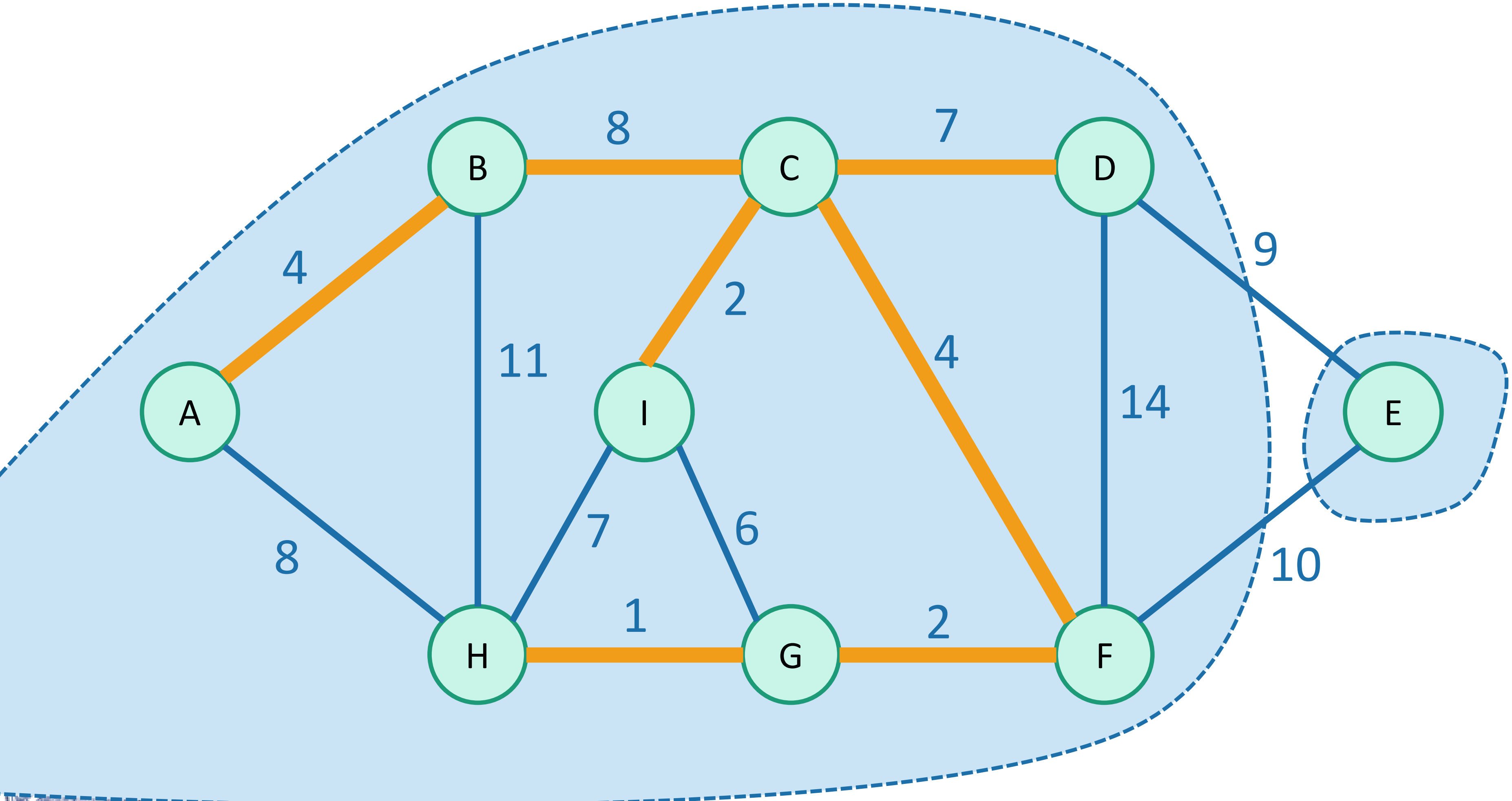
Once more…

Then start merging.



Once more…

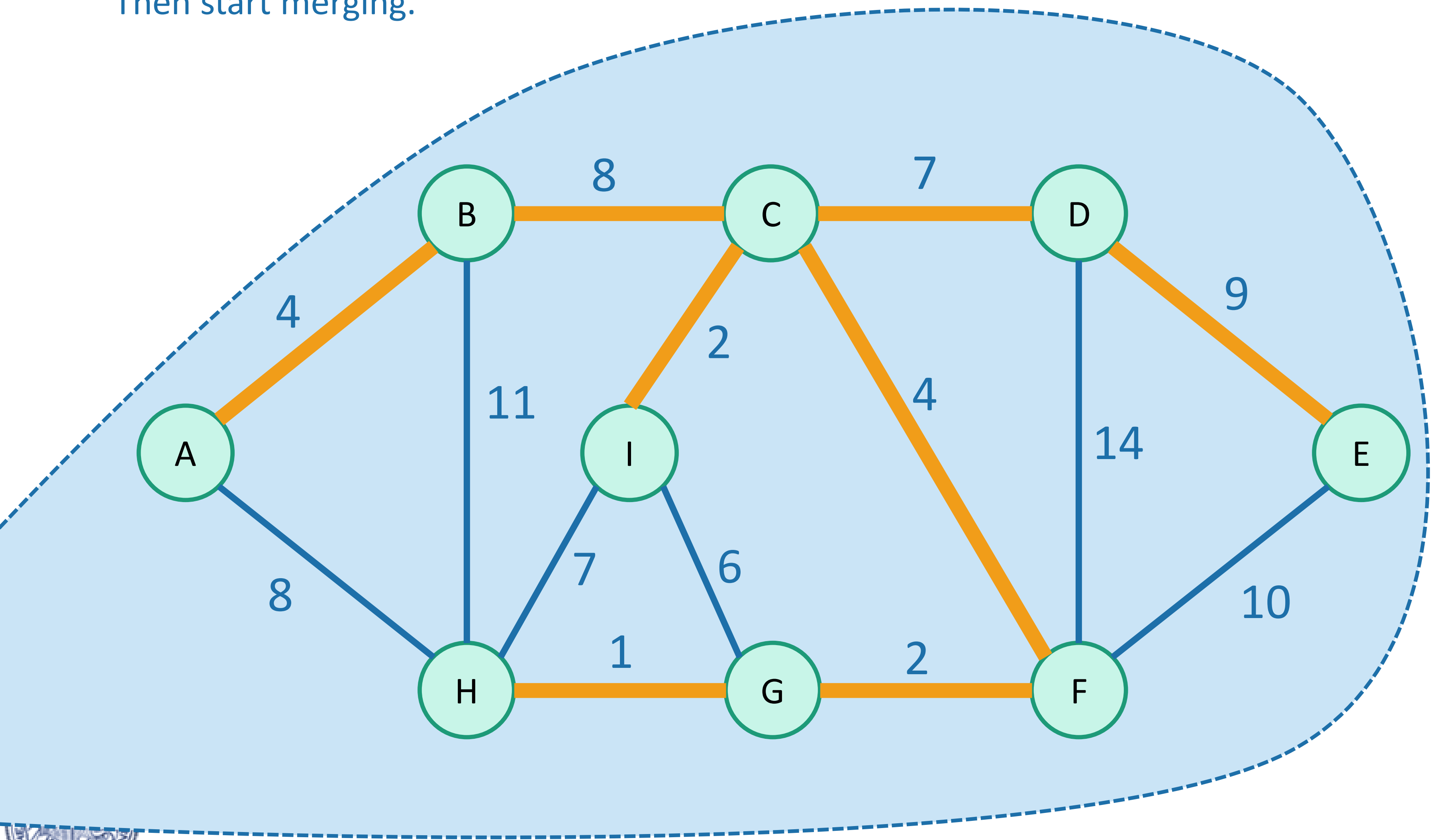
Then start merging.



Stop when we have one big tree!

Once more…

Then start merging.



Running time

- Sorting the edges takes O(m log(n))
  - In practice, if the weights are small integers we can use radixSort and take time O(m)
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - 2m calls to **find**
    - for each edge, **find** its endpoints
  - n calls to **union**
    - we will never add more than n-1 edges to the tree,
    - so we will never call **union** more than n-1 times.
- Total running time:
  - Worst-case O(mlog(n)), just like Prim with an RBtree.
  - Closer to O(m) if you can do radixSort

In practice, each of makeSet, find, and union run in constant time*

*technically, they run in *amortized time* O($\alpha(n)$), where $\alpha(n)$ is the *inverse Ackermann function*. $\alpha(n) \leq 4$ provided that n is smaller than the number of atoms in the universe.

Two questions

1. Does it work?
   - That is, does it actually return a MST?

Now that we understand this "tree-merging" view, let's do this one.

2. How do we actually implement this?
   - the pseudocode above says "slowKruskal"…
     - **Worst-case running time O(mlog(n)) using a union-find data structure.**

Does it work?

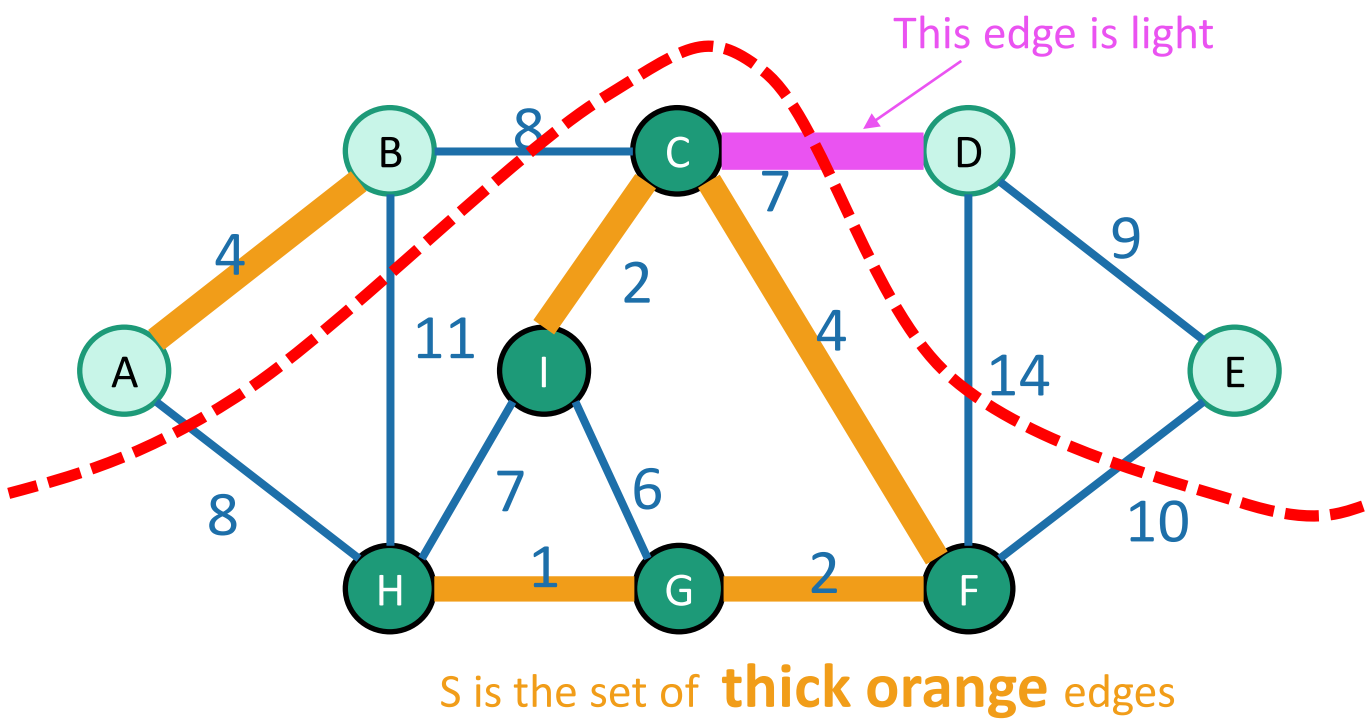
- We need to show that our greedy choices **don't rule out success.**
- That is, at every step:
  - There exists an MST that contains all of the edges we have added so far.

- Now it is time to use our lemma!

again!

Lemma

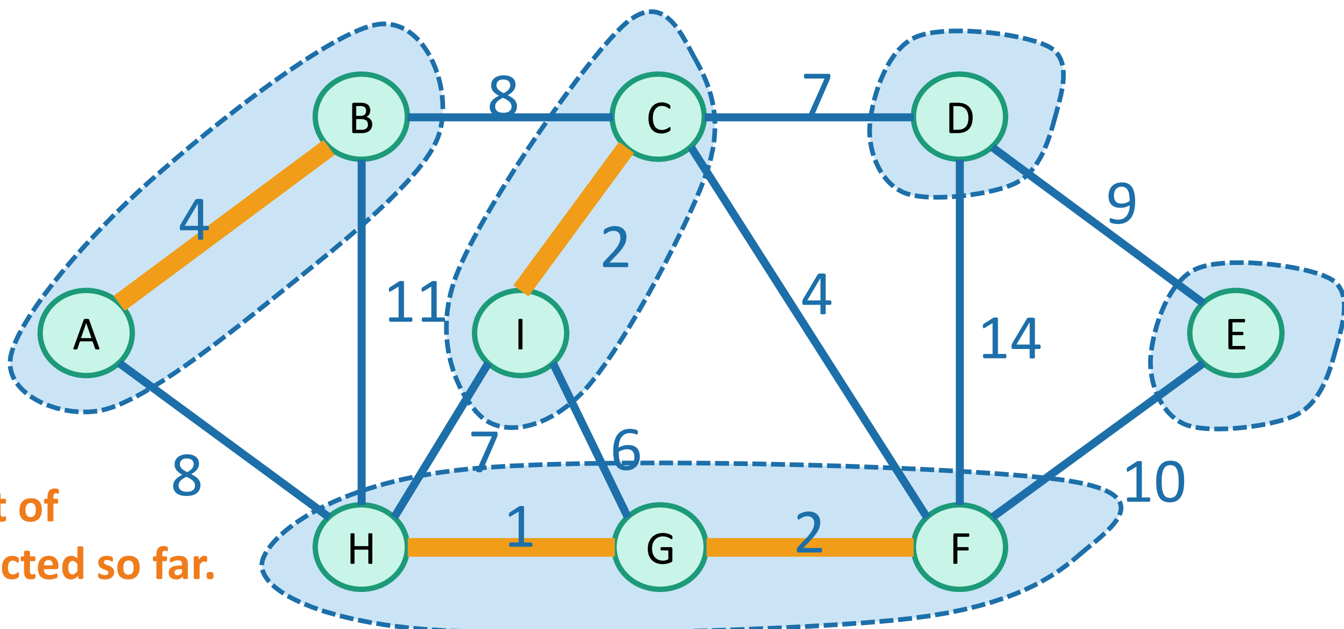
- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let {u,v} be a light edge.
- Then there is an MST containing S ∪ {{u,v}}



Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**

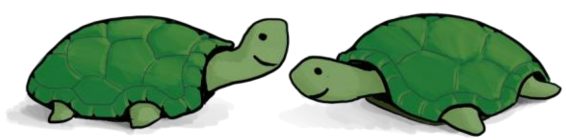
S is the set of
edges selected so far.



Partway through Kruskal

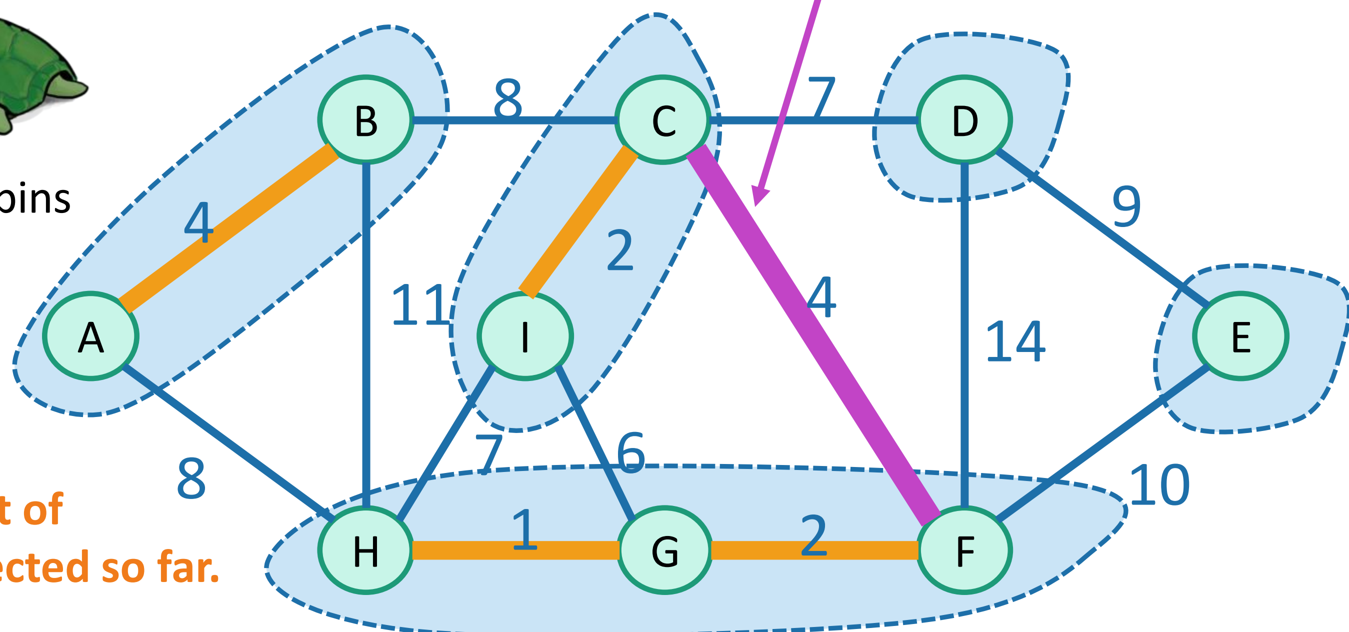
- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**

How can we use our lemma to show that our next choice also does not rule out success?



Think-Pair-Share Terrapins

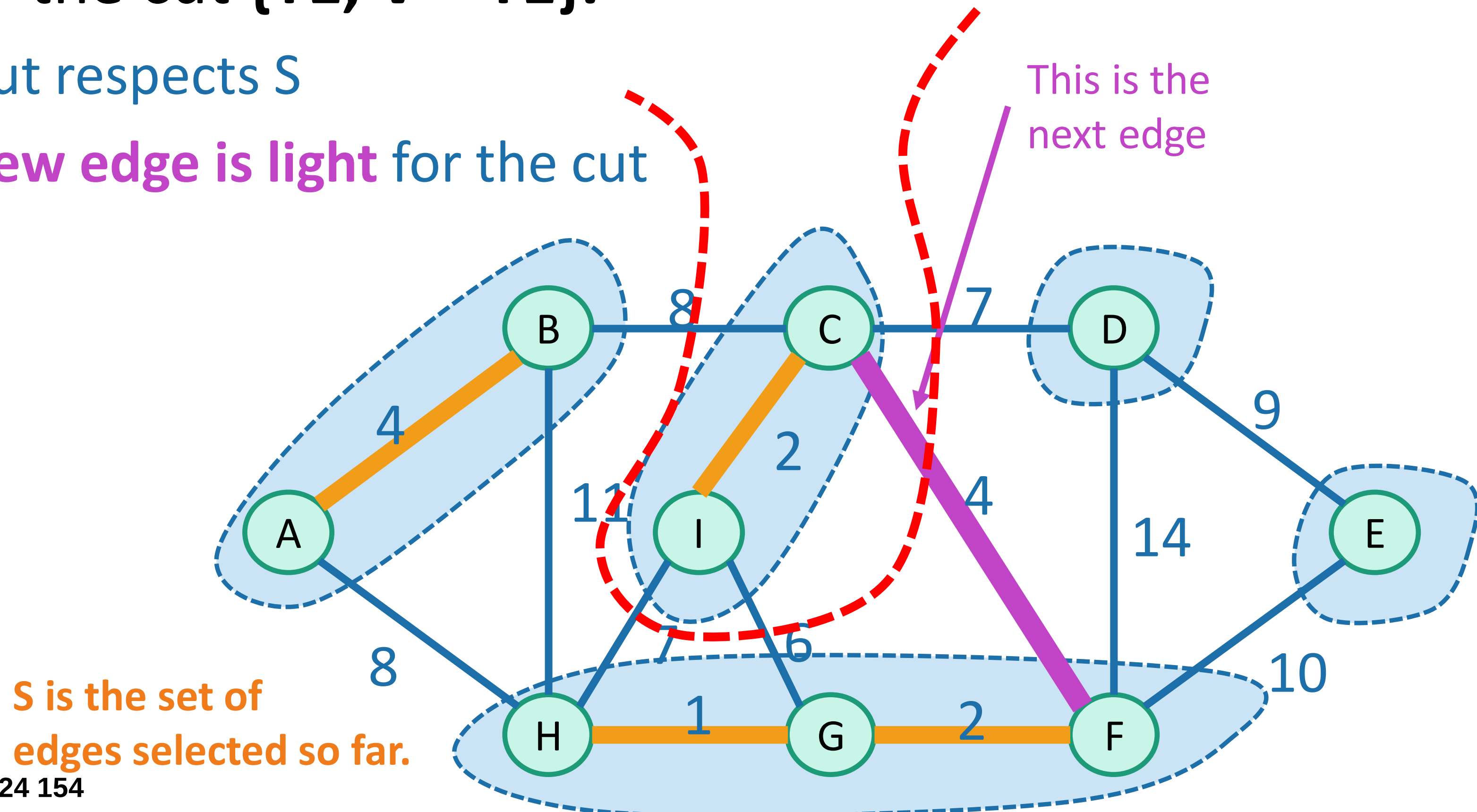
This is the next edge



S is the set of edges selected so far.

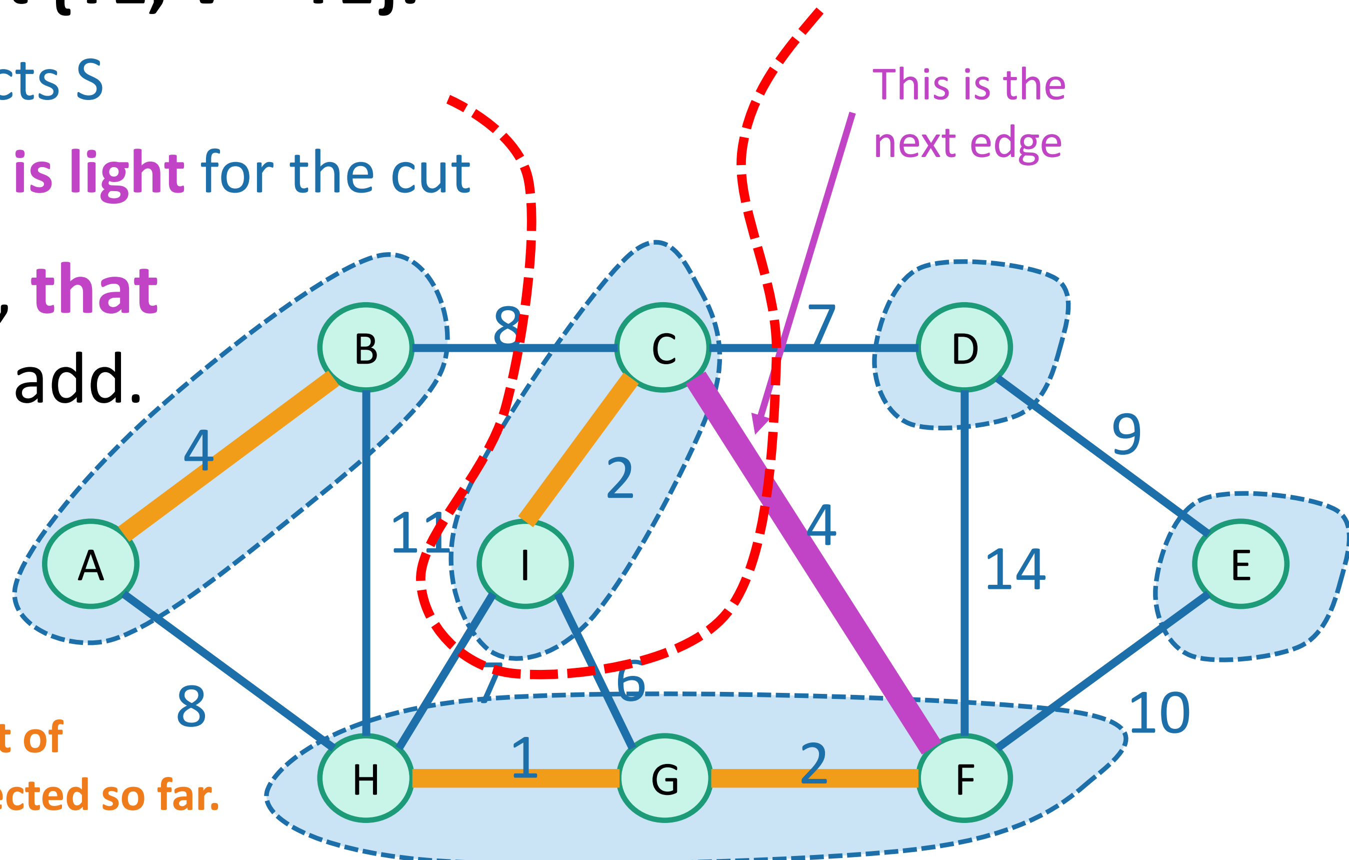
Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V – T1}.**
  - This cut respects S
  - Our **new edge is light** for the cut



Partway through Kruskal

- Assume that our choices **S** so far don't rule out success.
  - There is an MST extending them
- The **next edge** we add will merge two trees, **T1, T2**
- Consider the cut **{T1, V – T1}.**
  - This cut respects S
  - Our **new edge is light** for the cut
- By the Lemma, **that edge** is safe to add.
  - There is still an MST extending the new set



S is the set of edges selected so far.

Hooray!

- Our greedy choices **don't rule out success**.

- This is enough (along with an argument by induction) to guarantee correctness of Kruskal's algorithm.

Formally(ish)

This is exactly the same slide that we had for Prim's algorithm.

- Inductive hypothesis:
  - After adding the t'th edge, there exists an MST with the edges added so far.
- Base case:
  - After adding the 0'th edge, there exists an MST with the edges added so far. **YEP.**
- Inductive step:
  - If the inductive hypothesis holds for t (aka, the choices so far are safe), then it holds for t+1 (aka, the next edge we add is safe).
  - **That's what we just showed.**
- Conclusion:
  - After adding the n-1'st edge, there exists an MST with the edges added so far.
  - At this point we have a spanning tree, so it better be minimal.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
     - **Yes**

2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…
     - **Using a union-find data structure!**

What have we learned?

- Kruskal's algorithm greedily grows a forest
- It finds a Minimum Spanning Tree in time O(mlog(n))
  - if we implement it with a Union-Find data structure
  - if the edge weights are reasonably-sized integers and we ignore the inverse Ackerman function, basically O(m) in practice.

- To prove it worked, we followed the same recipe for greedy algorithms we saw last time.
  - Show that, at every step, we **don't rule out success.**

Compare and contrast

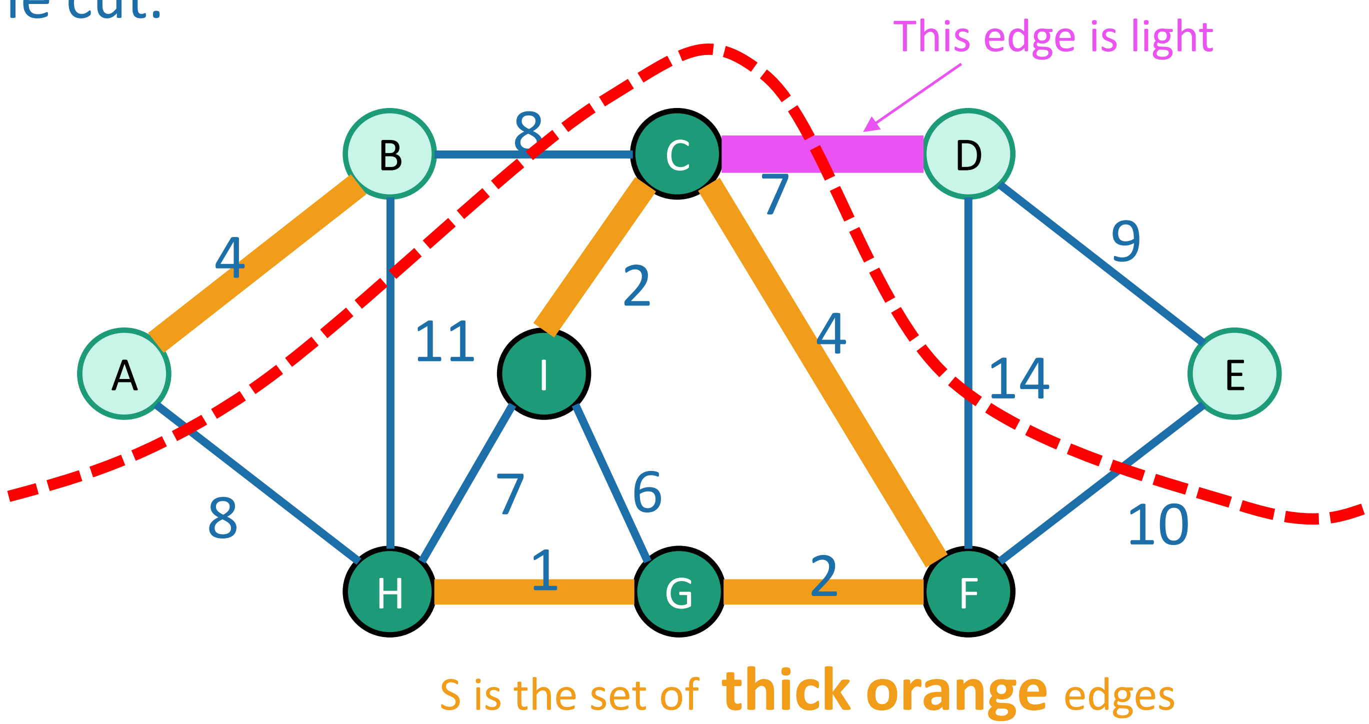
Prim might be a better idea on dense graphs if you can't radixSort edge weights

- Prim:
  - Grows a tree.
  - Time O(mlog(n)) with a red-black tree
  - Time O(m + nlog(n)) with a Fibonacci heap
- Kruskal:
  - Grows a forest.
  - Time O(mlog(n)) with a union-find data structure
  - If you can do radixSort on the edge weights, morally O(m)

Kruskal might be a better idea on sparse graphs if you can radixSort edge weights

Both Prim and Kruskal

- Greedy algorithms for MST.
- Similar reasoning:
  - Optimal substructure: subgraphs generated by cuts.
  - The way to make safe choices is to choose light edges crossing the cut.



Can we do better?

State-of-the-art MST on connected undirected graphs

- Karger-Klein-Tarjan 1995:
  - O(m) time randomized algorithm
- Chazelle 2000:
  - O(m$\cdot$ $\alpha(n)$) time deterministic algorithm
- Pettie-Ramachandran 2002:
  - O(The optimal number of comparisons N*(n,m) you need to solve the problem, whatever that is…) time deterministic algorithm

What is this number?
Do we need that silly $\alpha(n)$?
Open questions!

Recap

- Two algorithms for Minimum Spanning Tree
  - Prim's algorithm
  - Kruskal's algorithm

- Both are (more) examples of greedy algorithms!
  - Make a **series of choices**.
  - Show that at each step, your choice **does not rule out success.**
  - At the end of the day, you haven't ruled out success, so **you must be successful**.

Next time

- Back to randomized algorithms
- Minimum cuts!