CS 350 Algorithms and Complexity

Winter 2019

Lecture 14: Greedy Algorithms

(slides based on those of Mark Jones)

Andrew P. Black

Department of Computer Science

Portland State University

Greedy Algorithms

- Solves an optimization problem by breaking it into a sequence of steps, and making the best choice at each step.
- Key idea: a series of locally-optimal choices yields a globally-optimal choice.
- Not all problems can be solved by Greedy Algorithms; if the problem forms a matroid, then it can be so solved.

Example: making change

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?
  - Solve the problem using a Greedy Algorithm

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?
  - Solve the problem using a Greedy Algorithm
  - Numeric answer

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?
  - Solve the problem using a Greedy Algorithm
  - Numeric answer
- Now suppose that the US had a 20¢ coin (as does the UK, for example). Can you still solve the problem using a Greedy Algorithm?

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?
  - Solve the problem using a Greedy Algorithm
  - Numeric answer
- Now suppose that the US had a 20¢ coin (as does the UK, for example). Can you still solve the problem using a Greedy Algorithm?

A. Yes

Example: making change

- What is the smallest number of US coins (denominations 1¢, 5¢, 10¢ and 25¢) that can be used to make up 41¢?
  - Solve the problem using a Greedy Algorithm
  - Numeric answer
- Now suppose that the US had a 20¢ coin (as does the UK, for example). Can you still solve the problem using a Greedy Algorithm?

A. Yes

B. No

Example: Knapsack problem

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- What is the "greedy solution"?

A. Item 5
B. Items 3 & 5
C. Items 2 & 4
D. Items 1 & 5
E. None of the above

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

  - What is the "greedy solution"
  - Is this optimal?
    A. Yes
    B. No

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- What is the "greedy solution"
- Is this optimal?
- Will a greedy algorithm always work?
  - Suppose that $W = 5$? $W = 3$?

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- What is the "greedy solution"
- Is this optimal?
- Will a greedy algorithm always work?
  - Suppose that $W = 5$? $W = 3$?

A. Yes

B. No

Example: Knapsack problem

- This is the instance of the Knapsack problem that we solved previously:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

  - What is the "greedy solution"
  - Is this optimal?
  - Will a greedy algorithm always work?
    - Suppose that $W = 5$? $W = 3$?

Huffman Coding

The Coding Problem:

- A data file contains 100,000 "characters" each of which is either an a, b, c, d, e, or f

- Using three bits for each character takes:

  3 x 100,000 = 300,000 bits

- How could we do better?

The Coding Problem:

- A data file contains 100,000 "characters" each of which is either an a, b, c, d, e, or f

- Using three bits for each character takes:

  3 x 100,000 = 300,000 bits

- How could we do better?

Letter	Code
a	000
b	001
c	010
d	011
e	100
f	101

Using Frequency Information:

- *Variable length coding* gives shorter codes to more frequent letters.
- Encoded size:
  (45 * 1
  + (13+12+16+9) * 2
  + 5 * 3) * 1,000
  = 160,000
- A saving of of over 46%
- Is there a flaw?

Letter	Frequency	Code
a	45,000	0
b	13,000	01
c	12,000	10
d	16,000	00
e	9,000	11
f	5,000	100

Using Frequency Information:

Letter	Frequency	Code
a	45,000	0
b	13,000	01
c	12,000	10
d	16,000	00
e	9,000	11
f	5,000	100

- *Variable length coding* gives shorter codes to more frequent letters.
- Encoded size:
  (45 * 1
  + (13+12+16+9) * 2
  + 5 * 3) * 1,000
  = 160,000
- A saving of of over 46%
- Is there a flaw? A. Yes B. No

Unique Decoding:

- What string does the code 10000011010 represent?

- One reading:
  100 0 00 11 01 0
  f a d e b a

- Another reading:
  10 00 0 01 10 10
  c d a b c c

- Oh dear: we've lost too much of the information that was in the original!

Letter	Frequency	Code
a	45,000	0
b	13,000	01
c	12,000	10
d	16,000	00
e	9,000	11
f	5,000	100

Use a Prefix-free Code

- Prefix(-free) property:
  no codeword is a prefix of
  another codeword

- Encoded size:
  (45 * 1
   + (13+12+16) * 3
   + (9 + 5) * 4) * 1,000
  = 224,000

- Still reduce size by ~25%

- And this time, it can be
  decoded!

Use a Prefix-free Code

- Prefix(-free) property:
  no codeword is a prefix of another codeword

- Encoded size:
  (45 * 1
   + (13+12+16) * 3
   + (9 + 5) * 4) * 1,000
  = 224,000

- Still reduce size by ~25%

- And this time, it can be decoded!

Letter	Frequency	Code
a	45,000	0
b	13,000	101
c	12,000	100
d	16,000	111
e	9,000	1101
f	5,000	1100

Prefix Coding & Decoding:

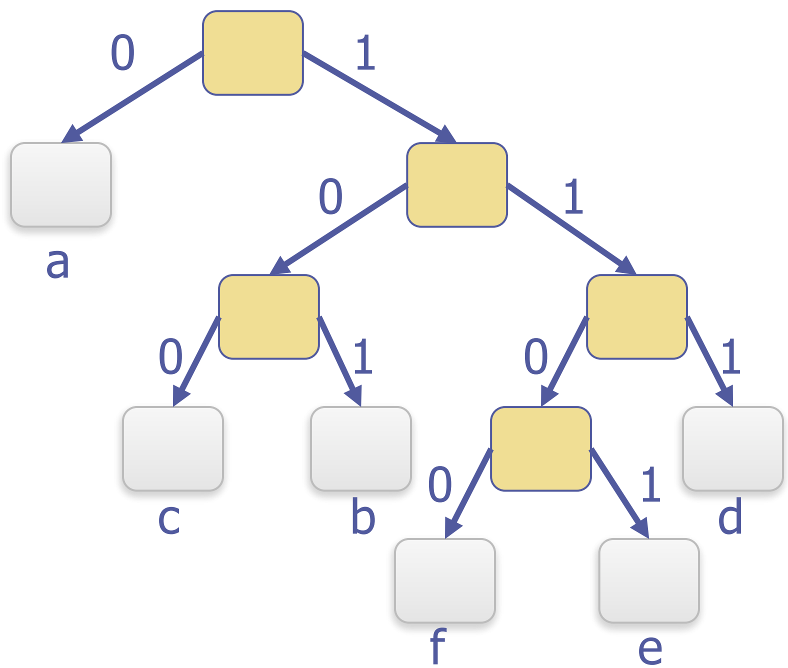
- A *prefix code* can achieve compression that is optimal among any character code

- Code can be represented by a tree:

Prefix Coding & Decoding:

- A *prefix code* can achieve compression that is optimal among any character code

- Code can be represented by a tree:

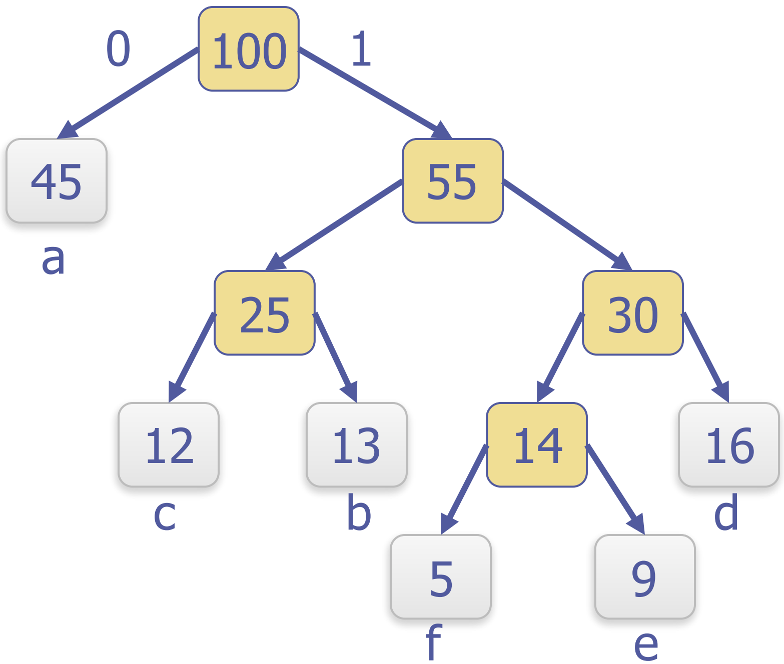


Letter	Frequency	Code
a	45,000	0
b	13,000	101
c	12,000	100
d	16,000	111
e	9,000	1101
f	5,000	1100

Frequencies & Costs:

- For any given coding tree T, the number of bits required to code a message is:

$$cost(T) = \sum_{c \in C} freq(c) \cdot depth_T(c)$$



Letter	Frequency	Code
a	45,000	0
b	13,000	101
c	12,000	100
d	16,000	111
e	9,000	1101
f	5,000	1100

Building a Huffman Coding Tree

- We can use a table to avoid doing a calculation more than once:

```
initialize a empty priority queue, Q
add a leaf node to Q for each character
while (|Q|>1) do
   l = extractMin(Q)
   r = extractMin(Q)
   t = new tree node
       with left=l, right=r, freq=l.freq+r.freq
   insert t into Q
return extractMin(Q)
```

Using frequency as key

- Complexity?
- Complexity for computing frequencies?

Building a Huffman Coding Tree

- We can use a table to avoid doing a calculation more than once:

```
initialize a empty priority queue, Q
add a leaf node to Q for each character
while (|Q|>1) do
   l = extractMin(Q)
   r = extractMin(Q)
   t = new tree node
       with left=l, right=r, freq=l.freq+r.freq
   insert t into Q
return extractMin(Q)
```

Using frequency as key

Greedy choices!

- Complexity?
- Complexity for computing frequencies?

Building a Huffman Coding Tree

- We can use a table to avoid doing a calculation more than once:

```
initialize a empty priority queue, Q
add a leaf node to Q for each character
while (|Q|>1) do
   l = extractMin(Q)
   r = extractMin(Q)
   t = new tree node
        with left=l, right=r, freq=l.freq+r.freq
   insert t into Q
return extractMin(Q)
```

Using frequency as key

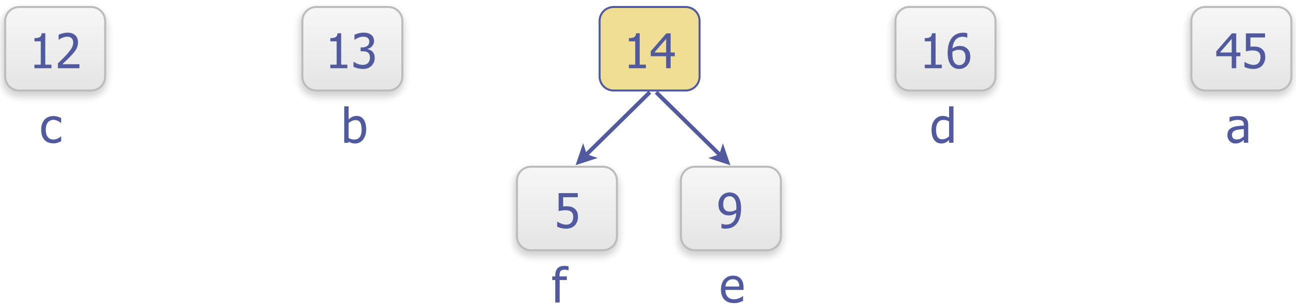
Greedy choices!

Last element in the queue

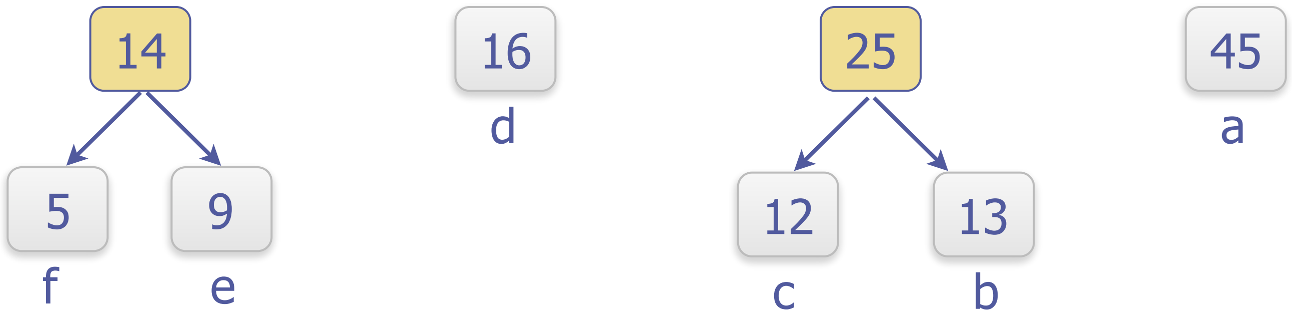
- Complexity?
- Complexity for computing frequencies?

Example:

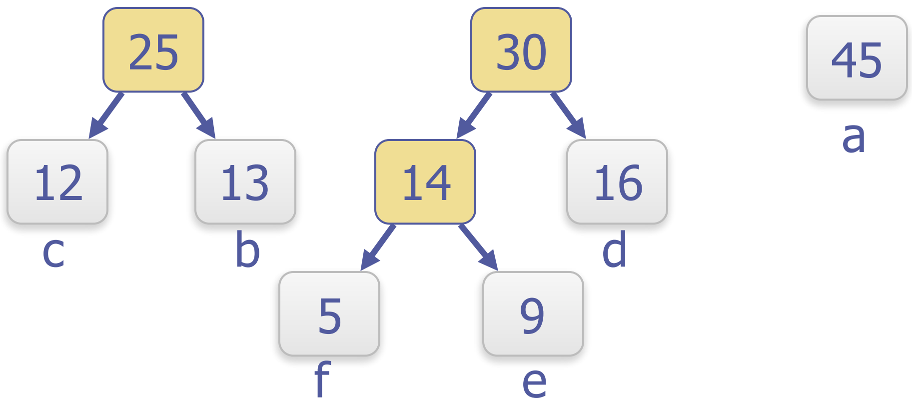
Example:



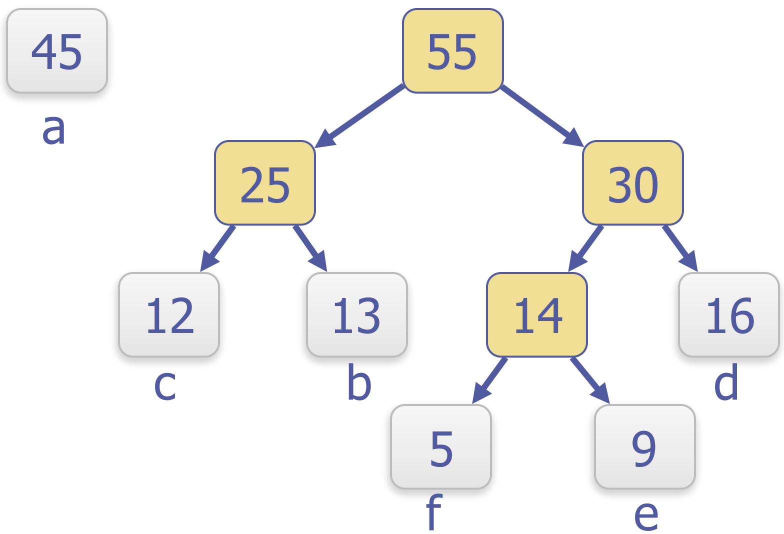
Example:



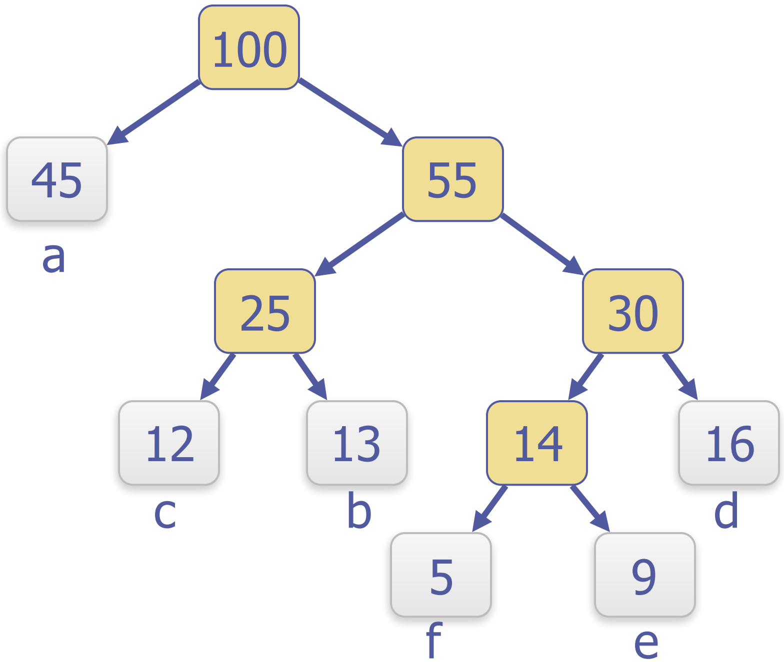
Example:



Example:



Example:



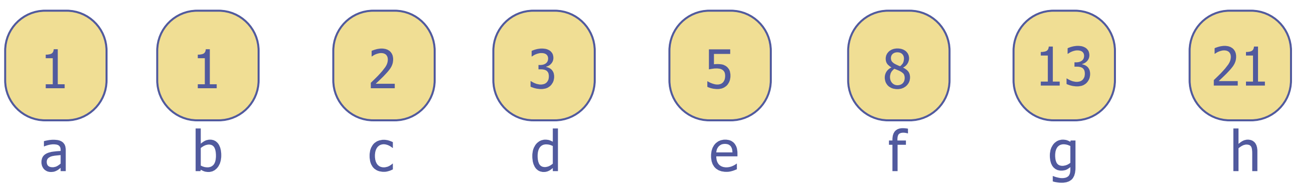
“Optimal Subproblems”

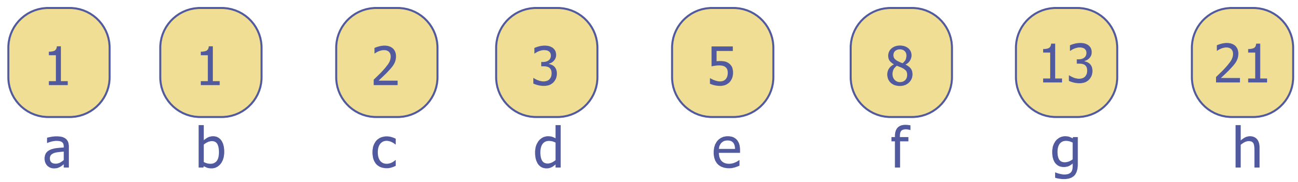
- At each iteration, our task is to find an optimal code for |Q| items
- We pick the pair of characters that have the lowest frequencies
- We reduce the original problem to the task of finding an optimal code for |Q|-1 items
- We can prove that the resulting coding scheme is indeed optimal

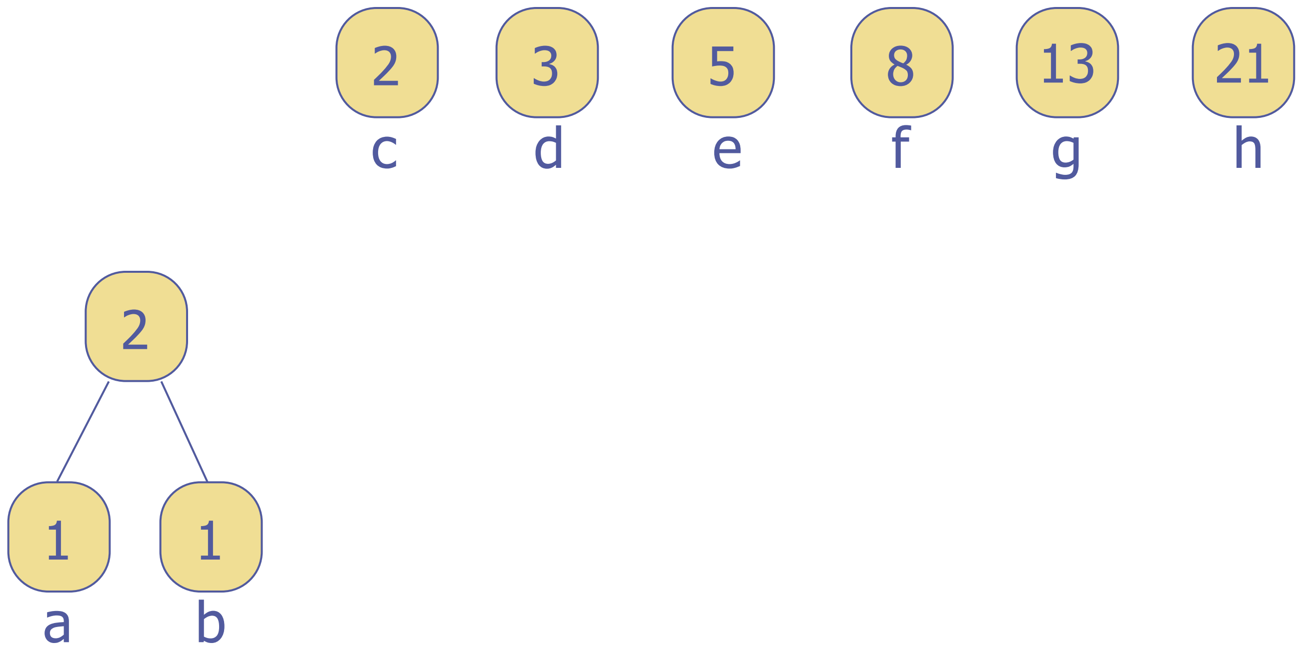
Huffman Trees (2nd Example)

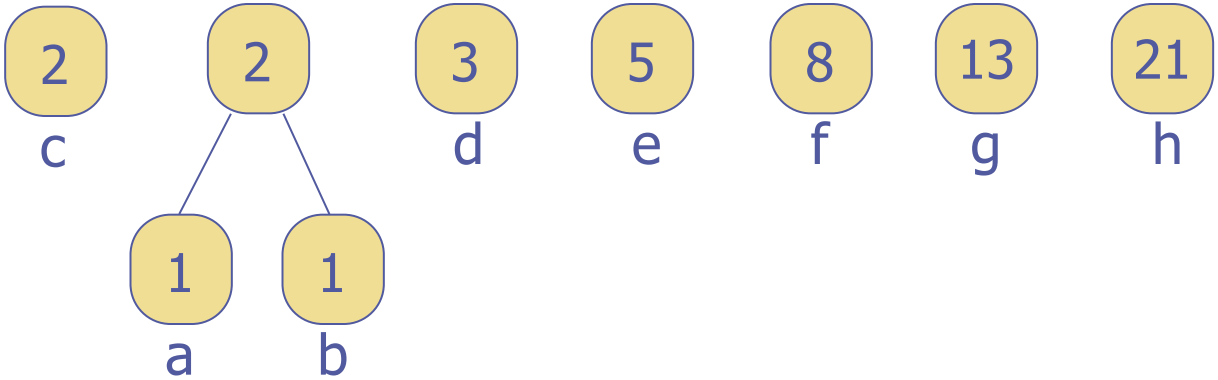
- Build the optimal Huffman code for the following set of frequencies

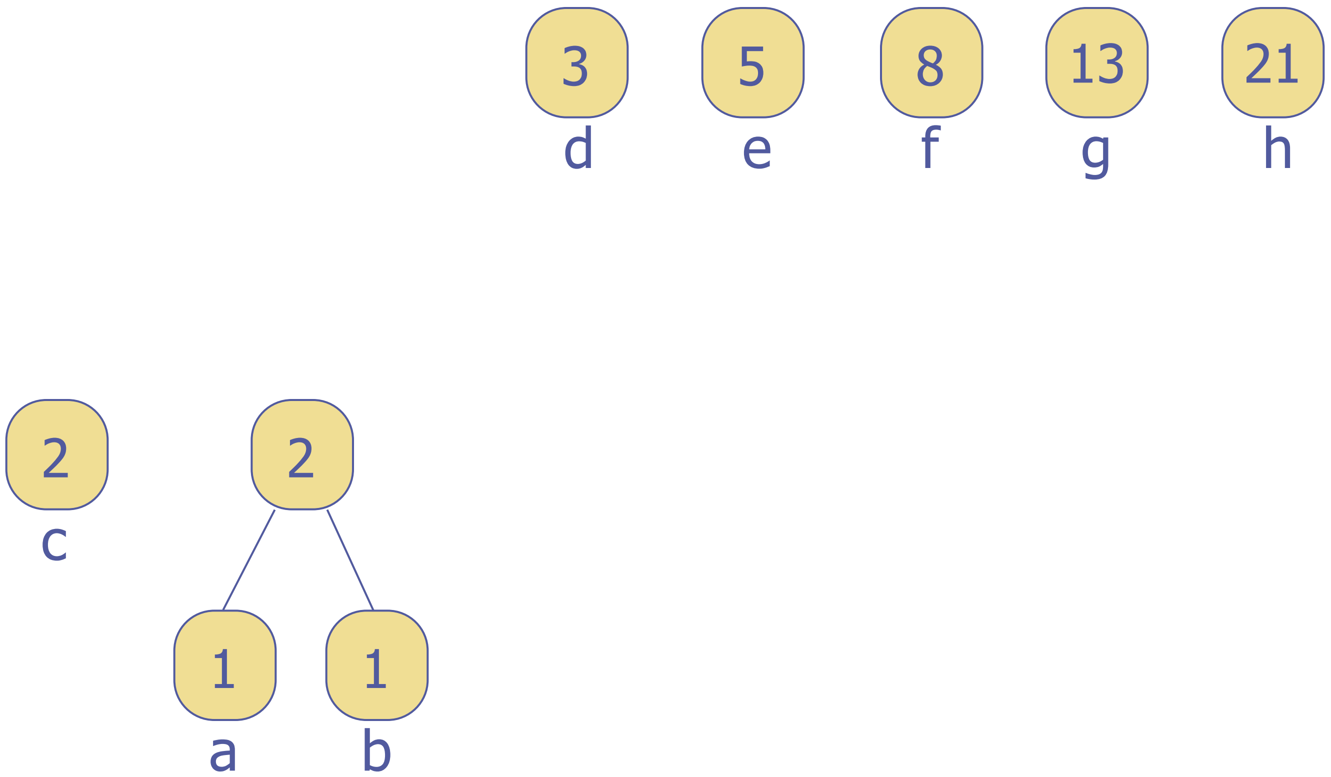
  a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

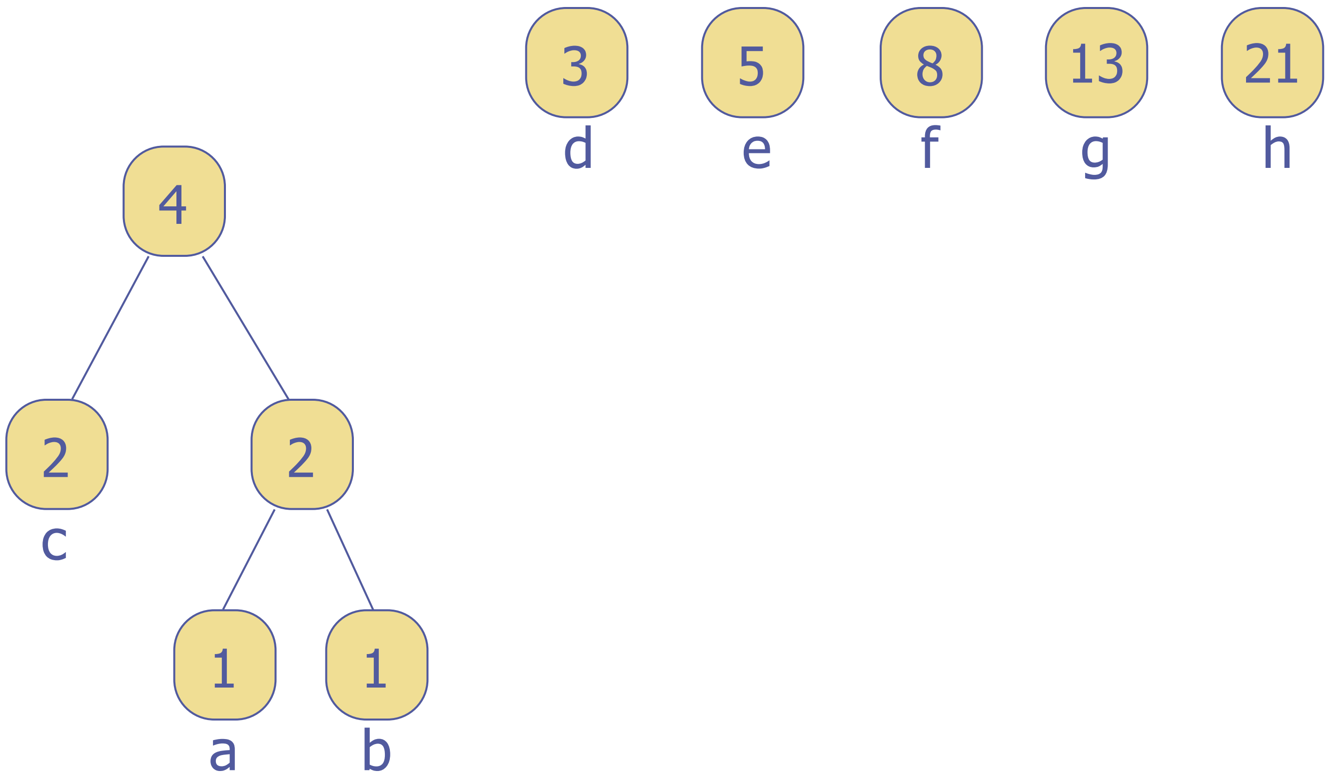


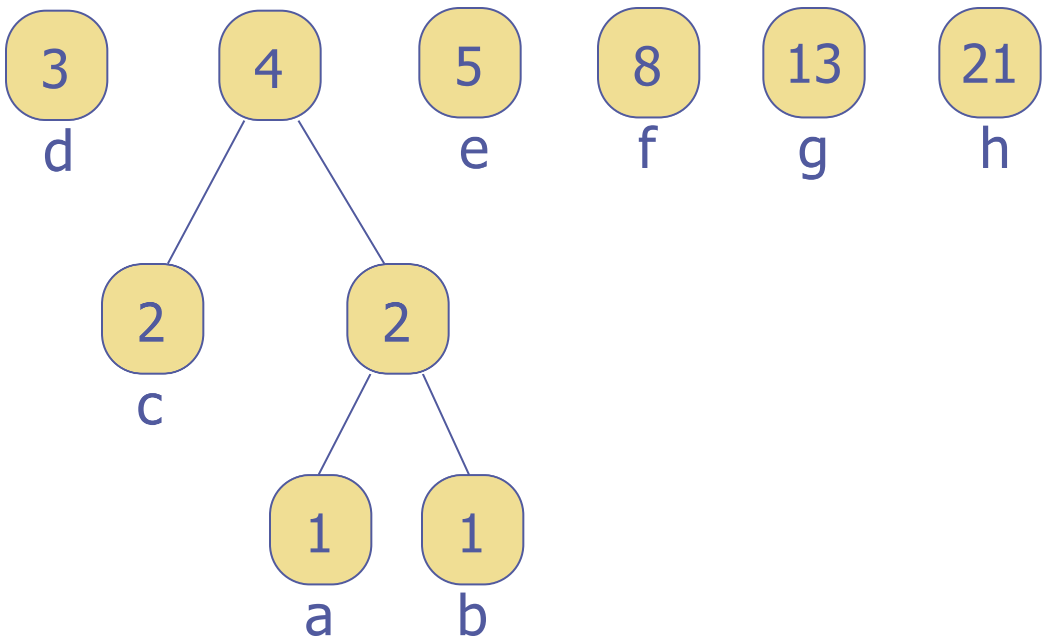

1
1
2
3
5
8
13
21
a
b
c
d
e
f
g
h


2
3
5
8
13
21
c
d
e
f
g
h
2
1
1
a
b


2
c
2
1
a
1
b
3
d
5
e
8
f
13
g
21
h


3
d
5
e
8
f
13
g
21
h
2
c
2
1
a
1
b


3
5
8
13
21
d
e
f
g
h
4
2
2
c
1
1
a
b


3
d
4
5
e
8
f
13
g
21
h
2
c
2
1
a
1
b

Correctness of Huffman Code

Proof Idea

- Step 1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
- Step 2: Show that this problem has an optimal substructure property, that is, an optimal solution to Huffman's algorithm contains optimal solutions to subproblems.
- Step 3: Conclude correctness of Huffman's algorithm using step 1 and step 2.

Lemma: Greedy Choice Property

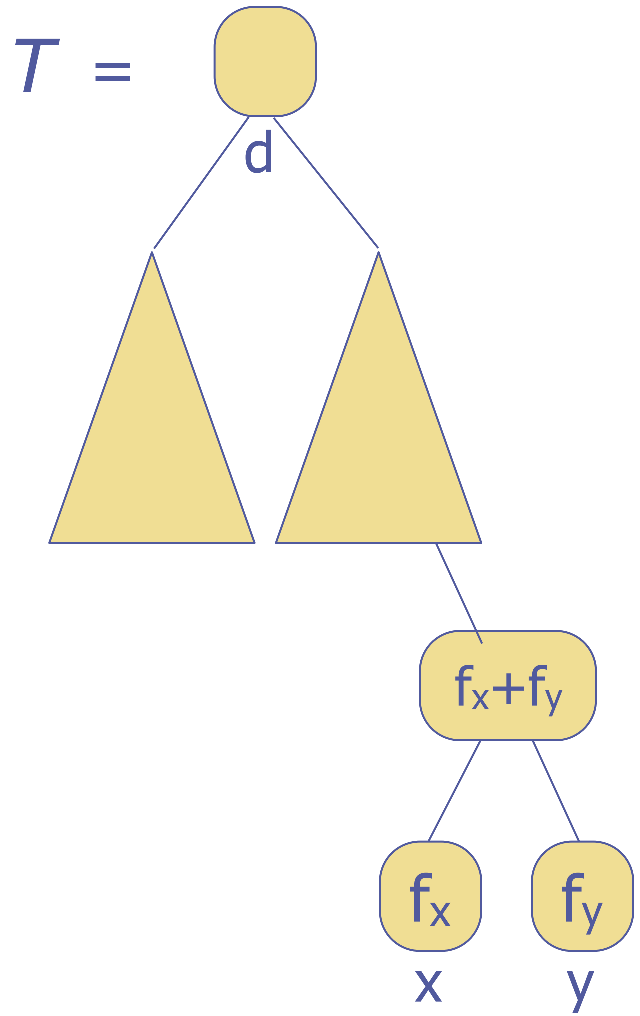
Let c be an alphabet in which each character c has frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Lemma: Optimal Substructure Property

- Let T be a full binary tree representing an optimal prefix code over an alphabet C, where each $c \in C$ has frequency f_c.
- Consider any two characters x and y that appear as sibling leaves in the tree T.
- Consider alphabet $C' = C - \{x, y\} \cup \{z\}$ with frequency $f_z = f_x + f_y$, and label with z the parent of x and y
- Then $T' = T - \{x, y\}$ represents an optimal code for alphabet C'

T represents an optimal prefix code for alphabet *C*

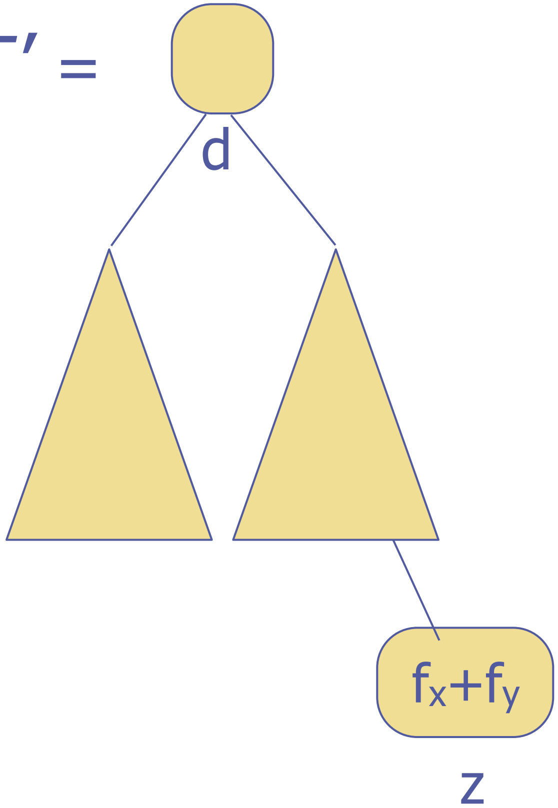
x and *y* appear as sibling leaves



T′ represents an optimal prefix code for alphabet *C′*

x and *y* replaced by *z*

$T' =$



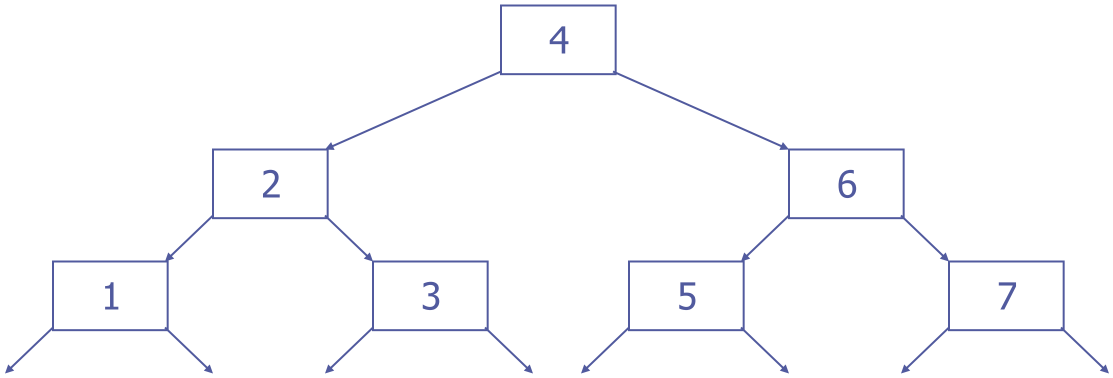
Priority Queues

Priority Queues

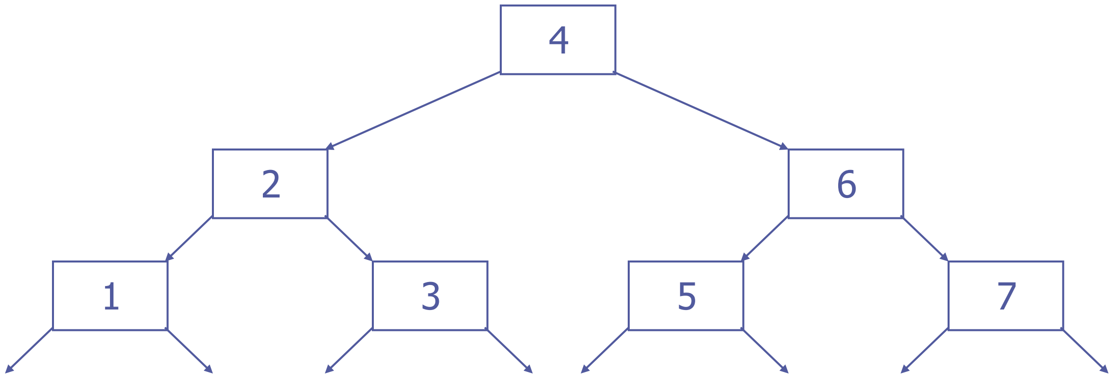
- A Priority Queue is a data structure optimized for finding and removing the element with the max (or min) key. It has operations to:
  - find the highest priority element (with max key)
  - delete the highest priority element
  - add a new item
- We want to avoid insertion sort at each step
  - Complexity of insertion would be O(n)
- We use a *Heap* (Levitin §6.4) — a particular kind of balanced tree.

The ideal:

- O(log n) complexity
- Everybody happy



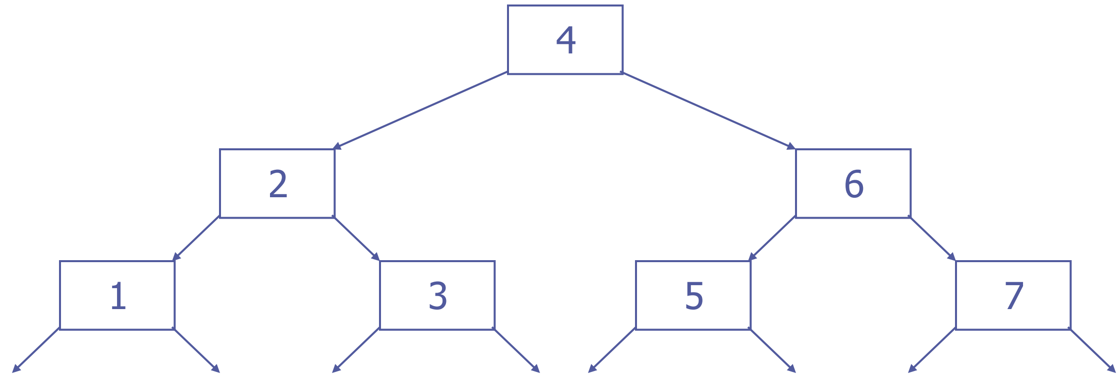
The ideal:



- O(log n) complexity
- Everybody happy

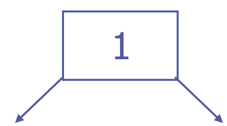
The (possible) reality:

The ideal:

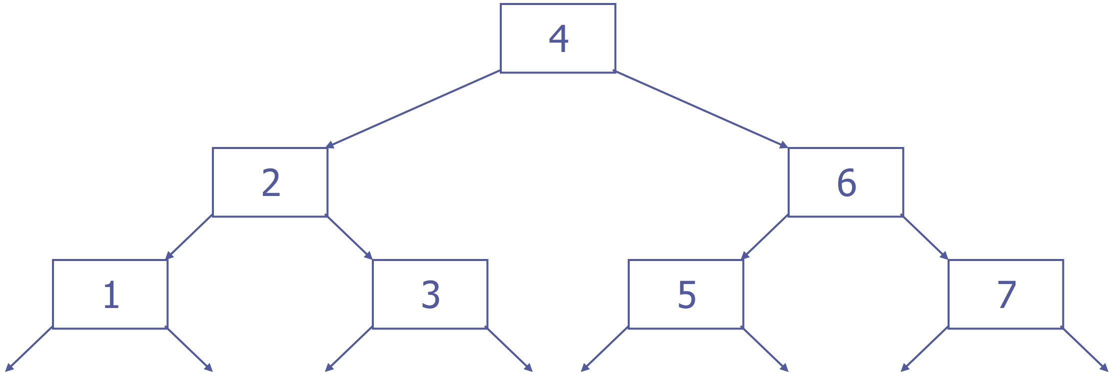


- O(log n) complexity
- Everybody happy

The (possible) reality:

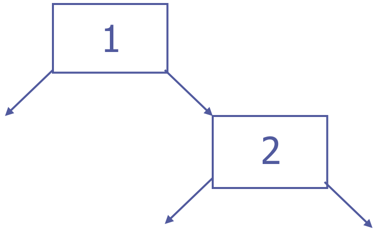


The ideal:

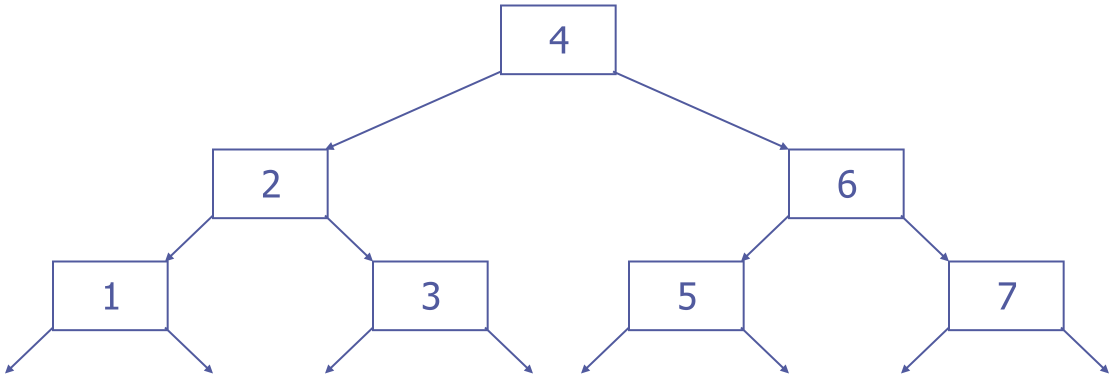


- O(log n) complexity
- Everybody happy

The (possible) reality:

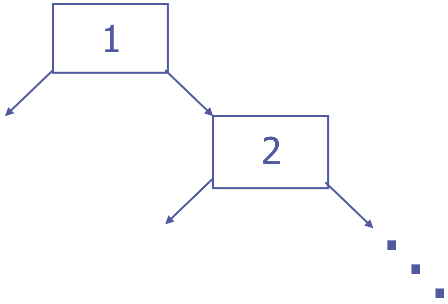


The ideal:

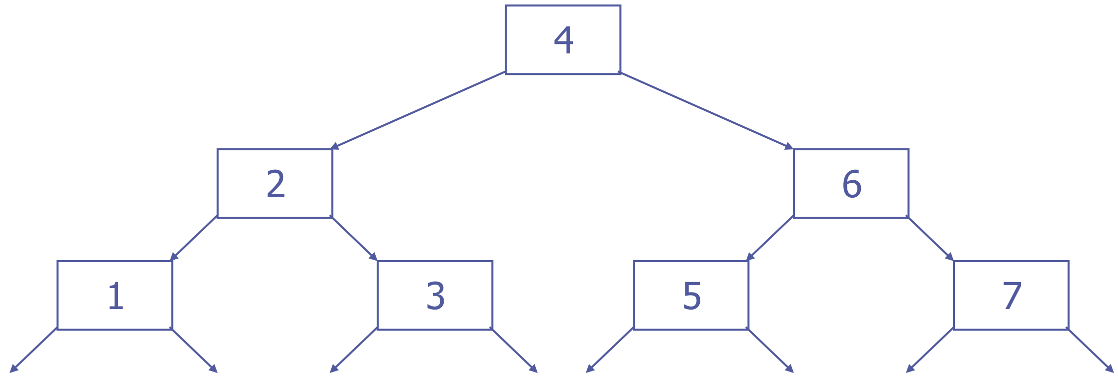


- O(log n) complexity
- Everybody happy

The (possible) reality:

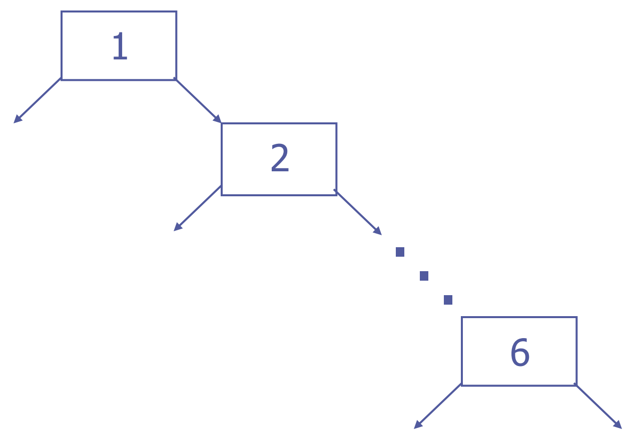


The ideal:

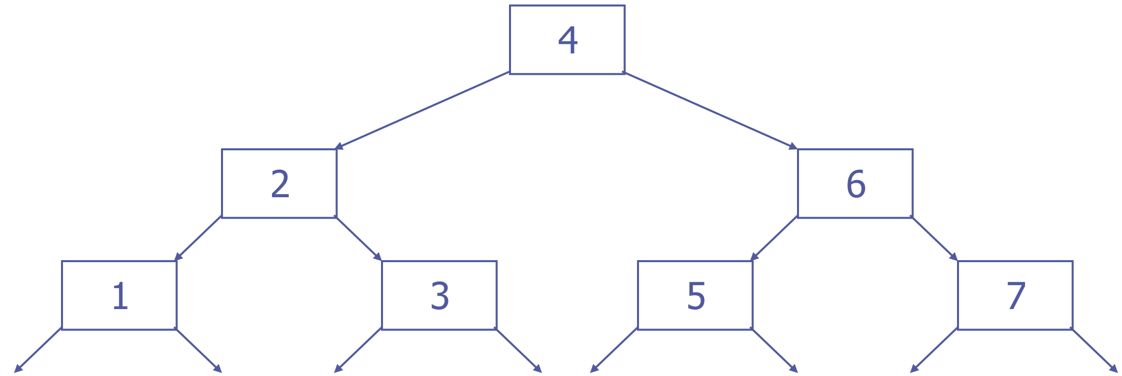


- O(log n) complexity
- Everybody happy

The (possible) reality:

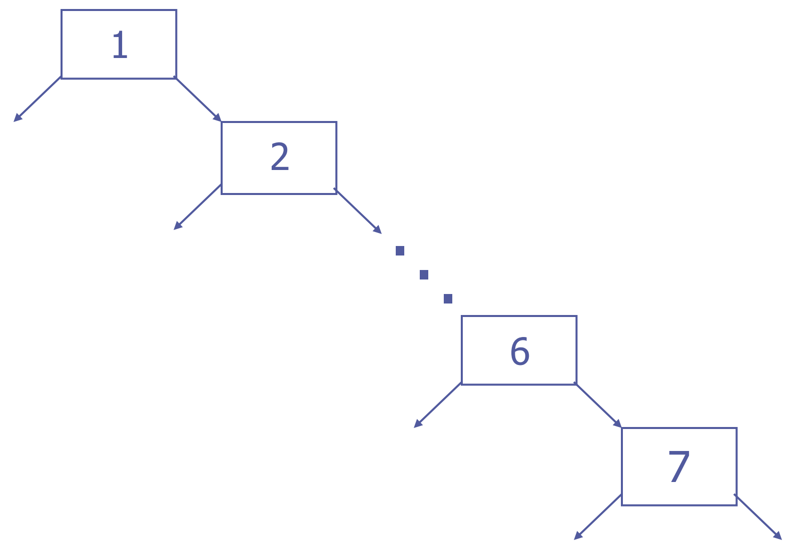


The ideal:

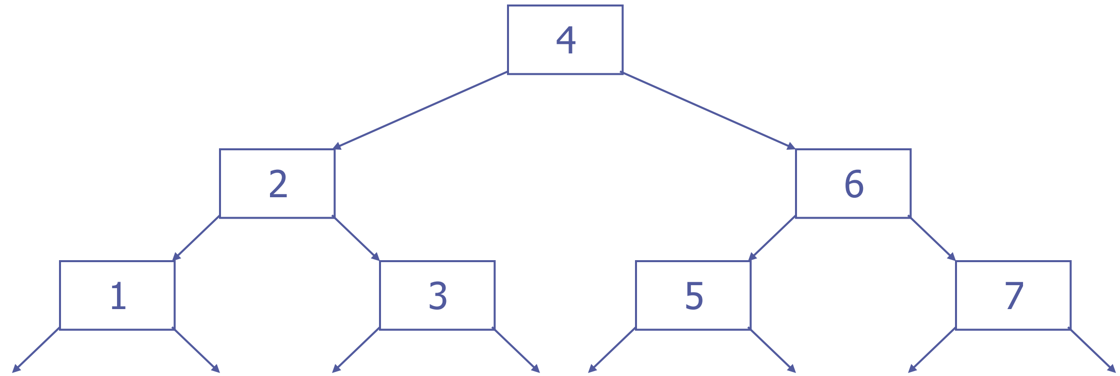


- O(log n) complexity
- Everybody happy

The (possible) reality:

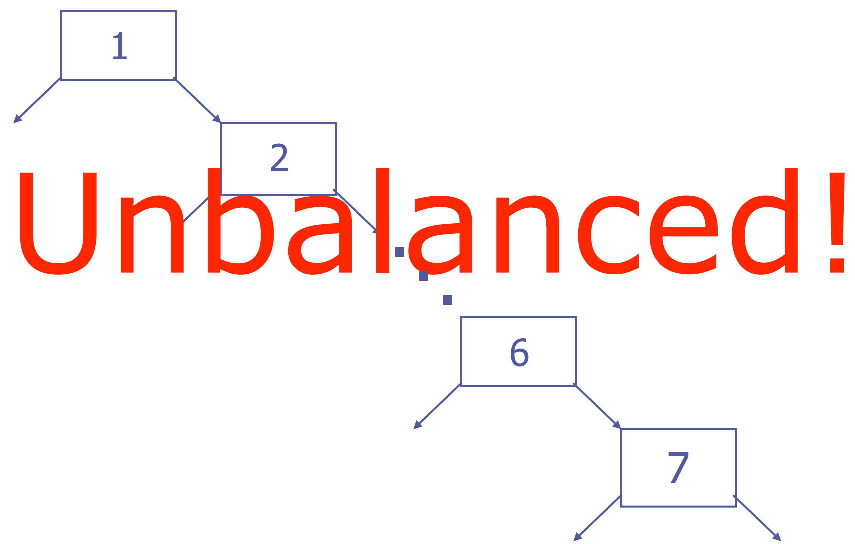


The ideal:

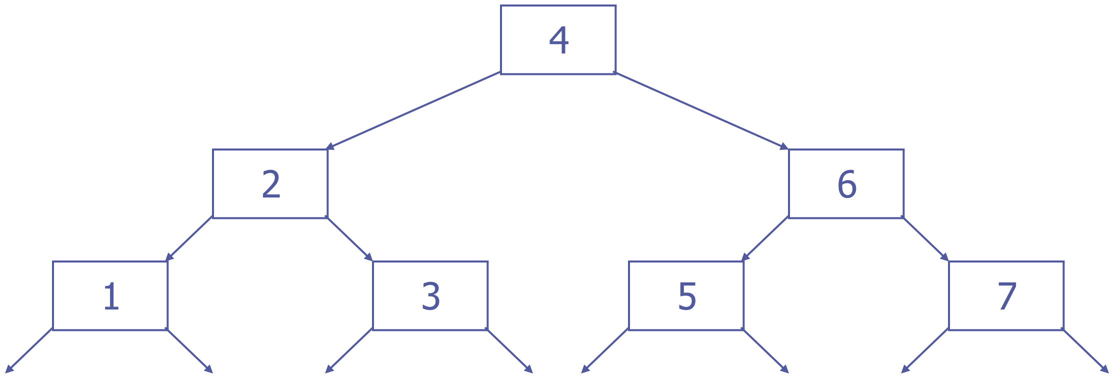


- O(log n) complexity
- Everybody happy

The (possible) reality:



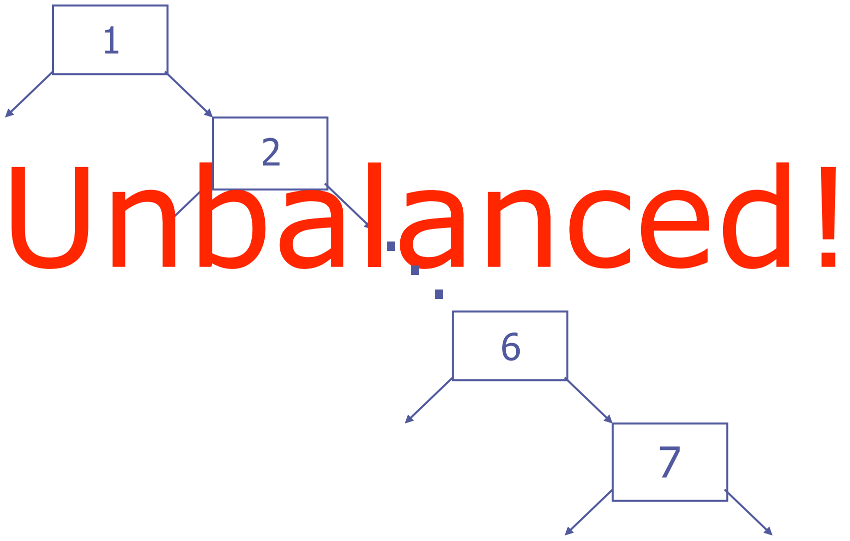
The ideal:



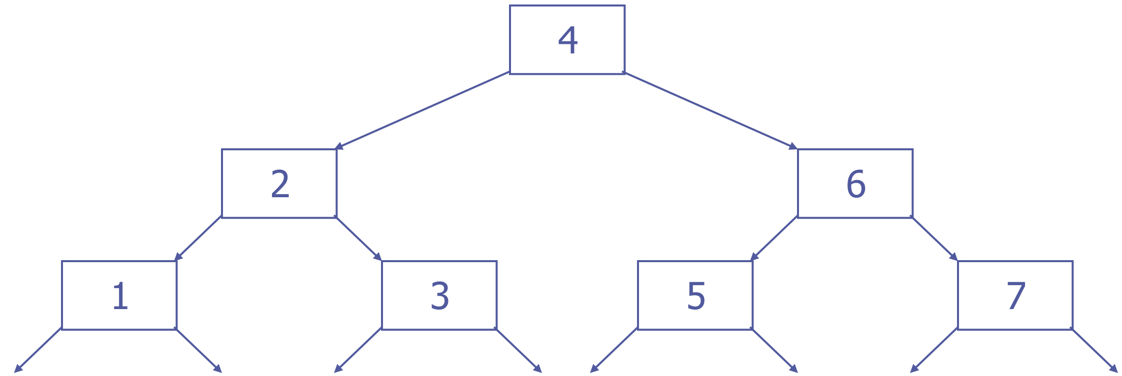
- O(log n) complexity
- Everybody happy

The (possible) reality:

- O(n) complexity
- Could have used lists!

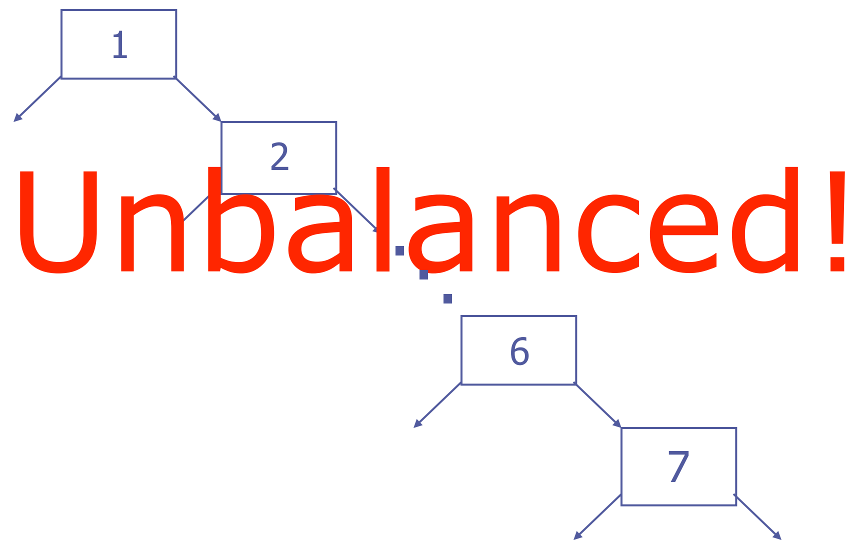


The ideal:



- O(log n) complexity
- Everybody happy

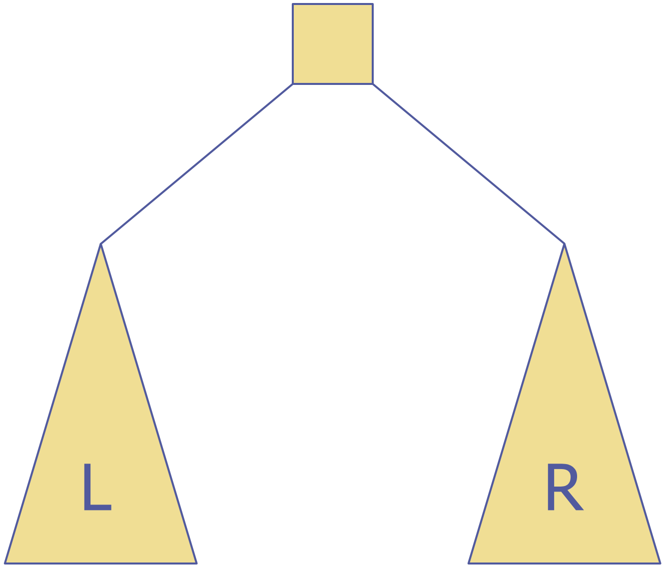
The (possible) reality:



- O(n) complexity
- Could have used lists!

What does “balanced” mean?

Perhaps:



size L = size R

?

Too constraining!

- A balanced binary tree of height h has exactly n_h elements, where:

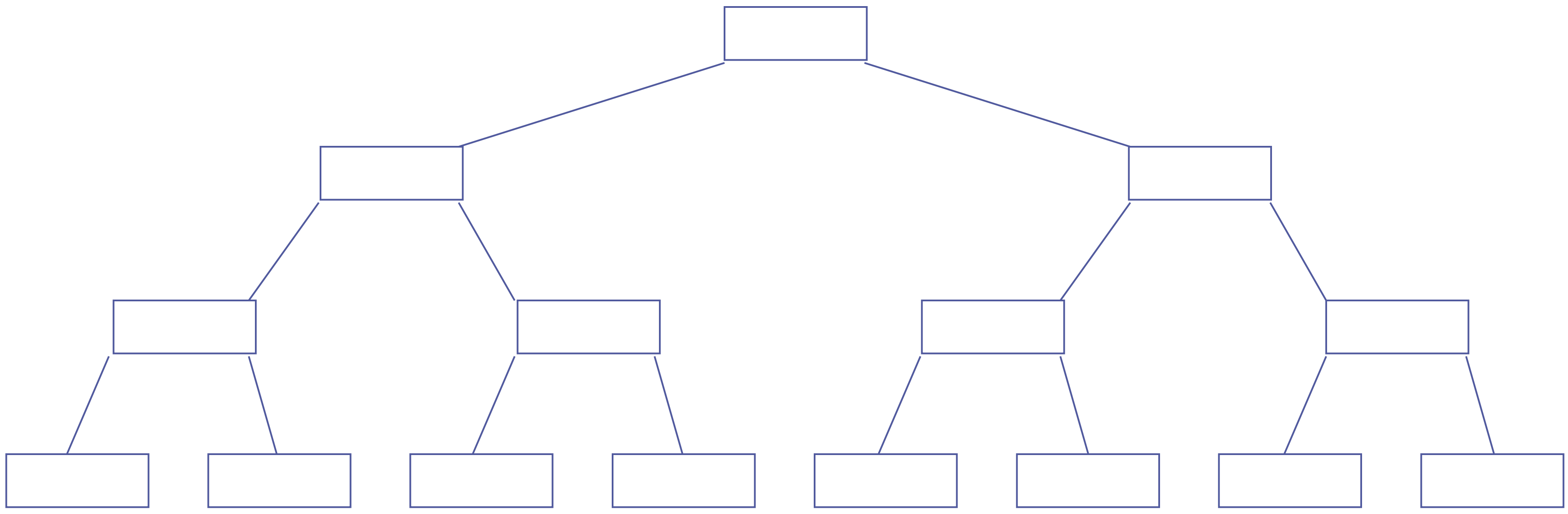
$$n_{-1} = 0 \text{ and } n_{(h+1)} = 1 + 2\, n_h;$$

- So if T is perfectly balanced, then:

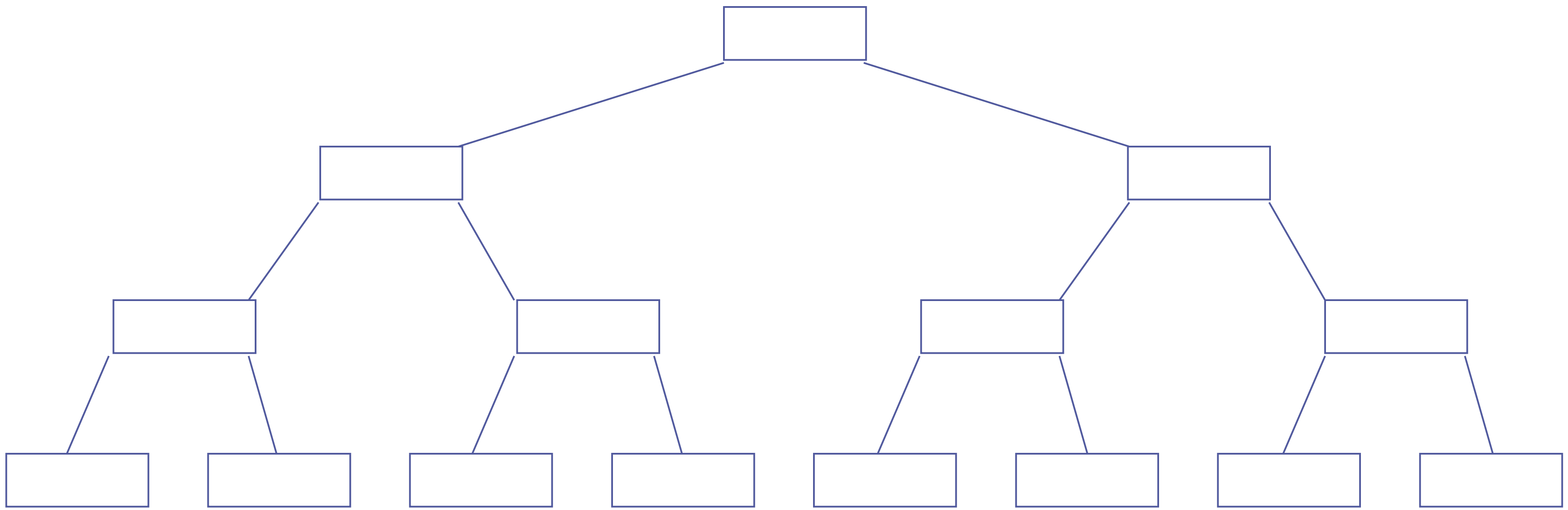
$$\text{size } T \in \{0, 1, 3, 7, 15, 31, 63, \ldots, 2^h\text{-}1, \ldots\};$$

- There is no perfectly balanced tree with any other number of elements.

A perfectly balanced tree:

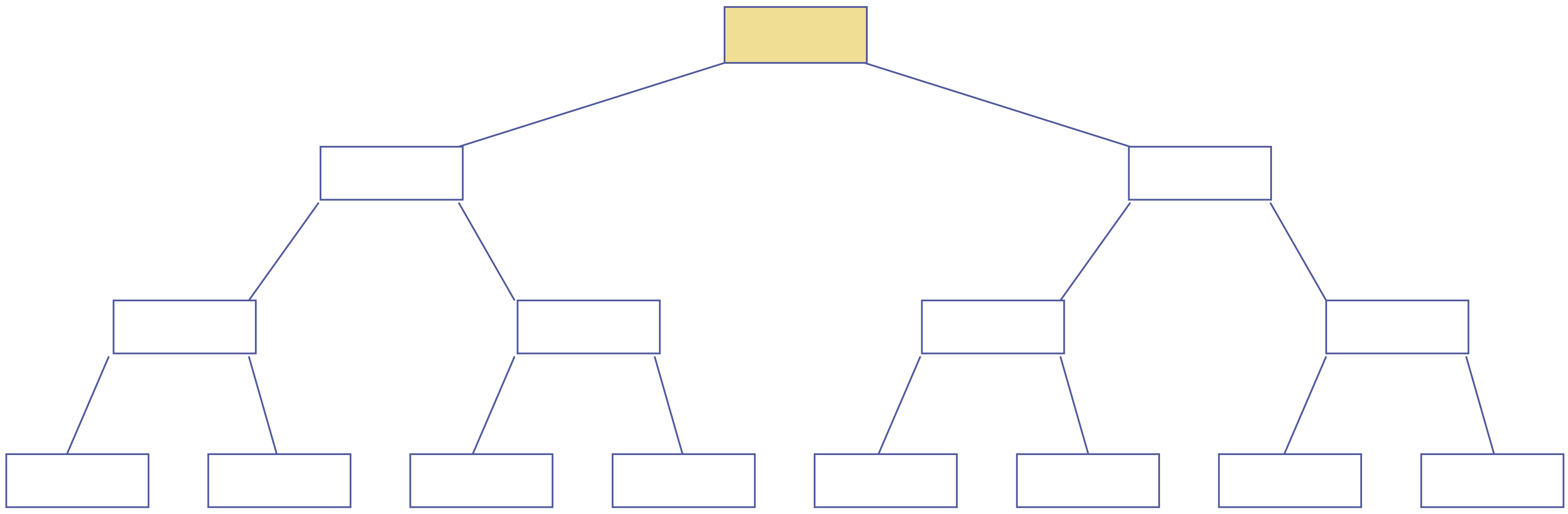


A perfectly balanced tree:



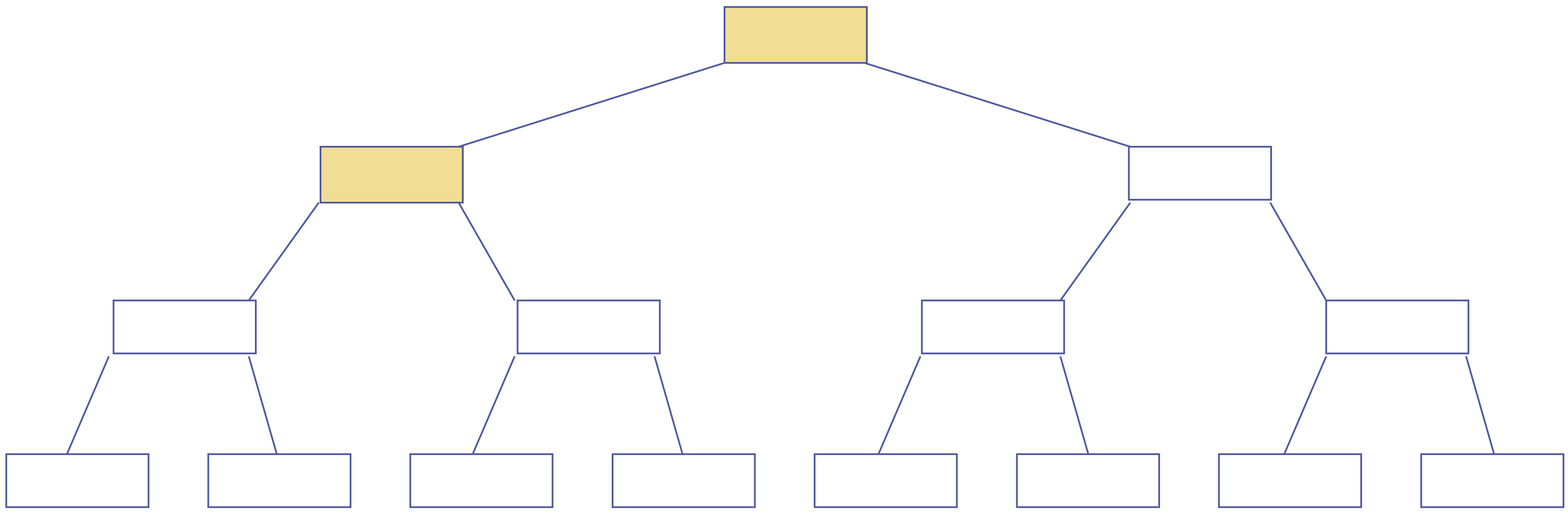
Think of this as an empty frame that
we can fill with elements ...

A perfectly balanced tree:



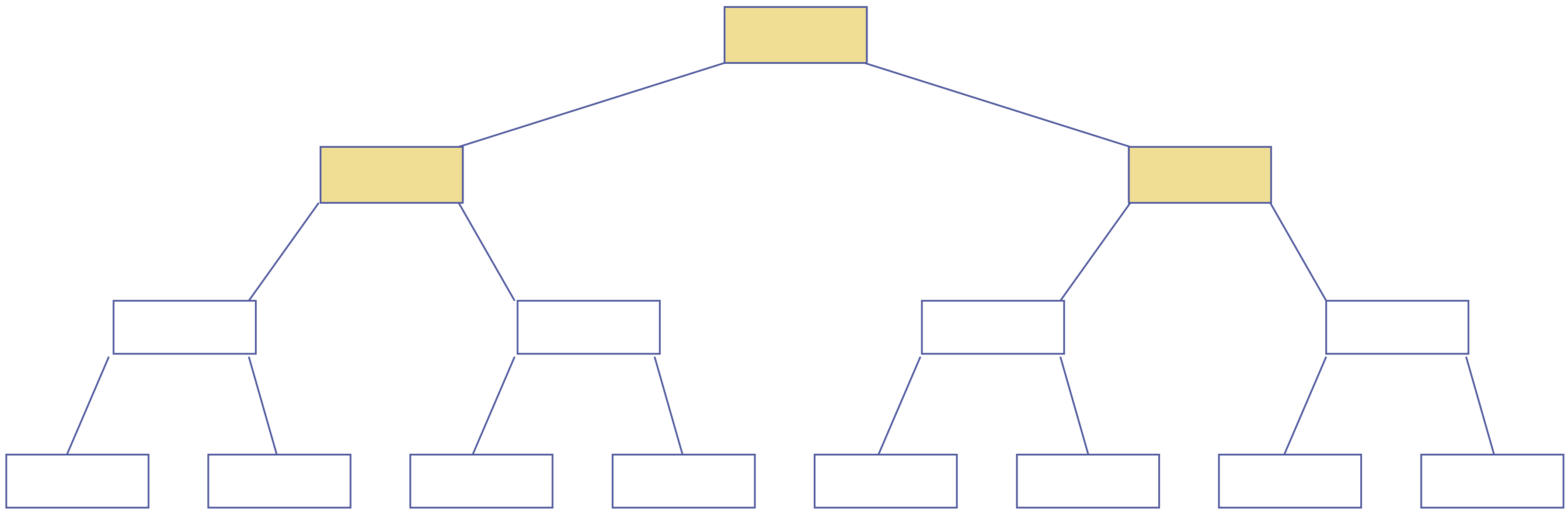
Think of this as an empty frame that we can fill with elements ...

A perfectly balanced tree:



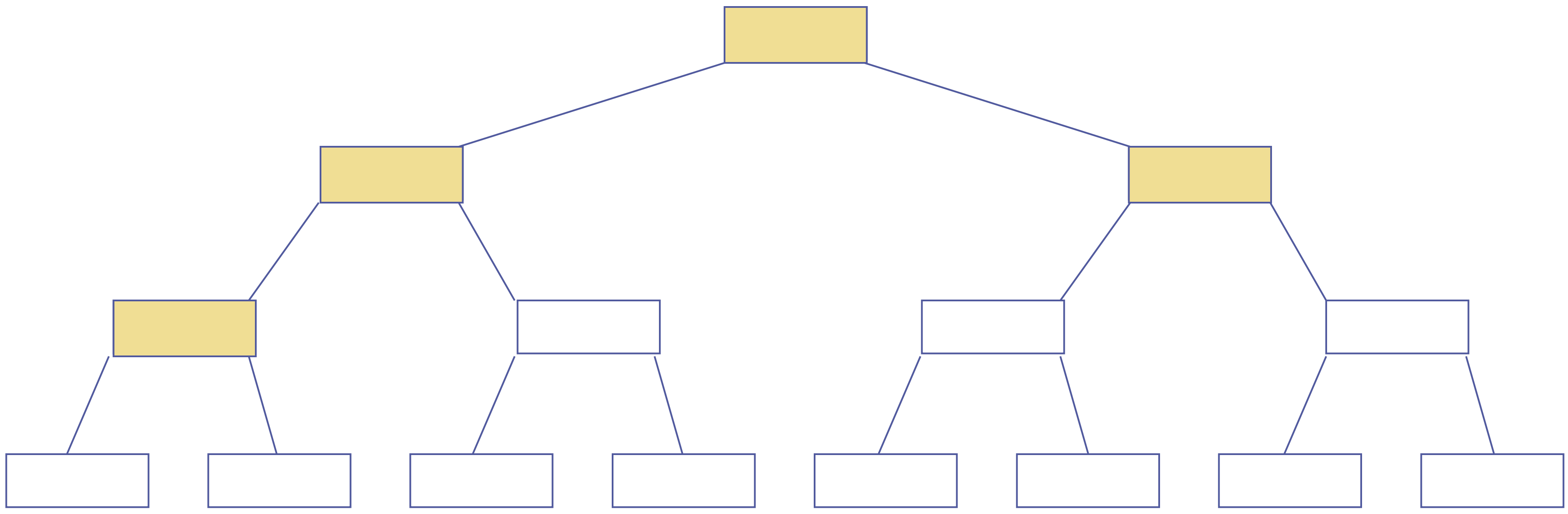
Think of this as an empty frame that
we can fill with elements ...

A perfectly balanced tree:



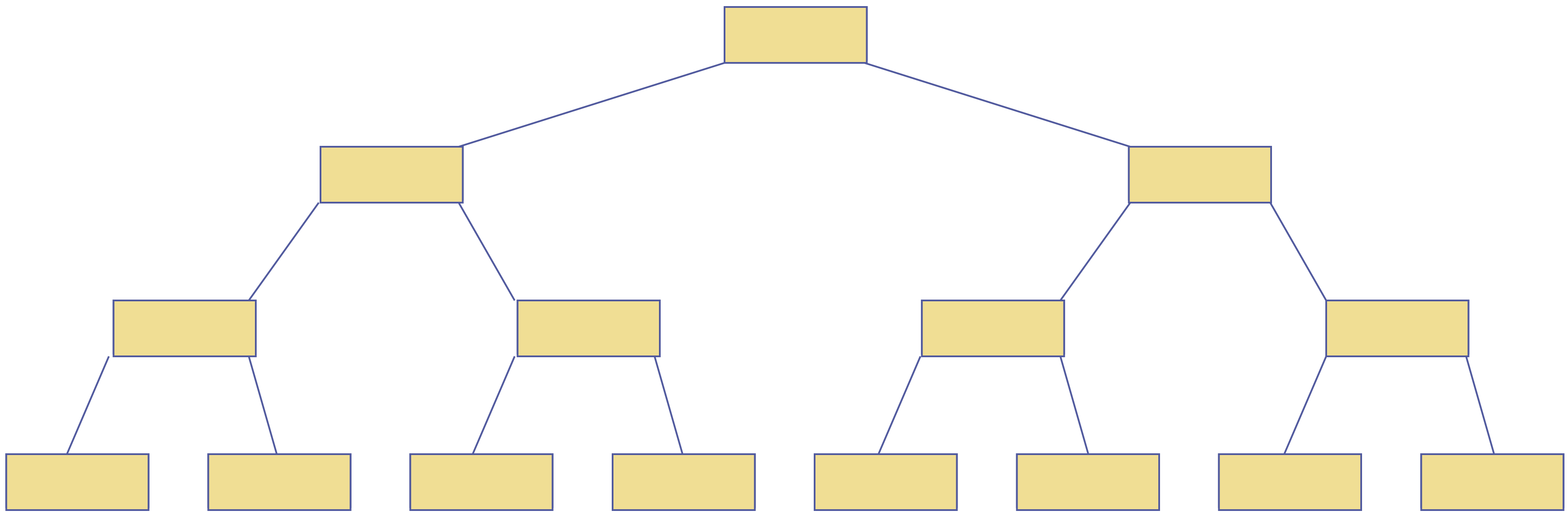
Think of this as an empty frame that we can fill with elements ...

A perfectly balanced tree:



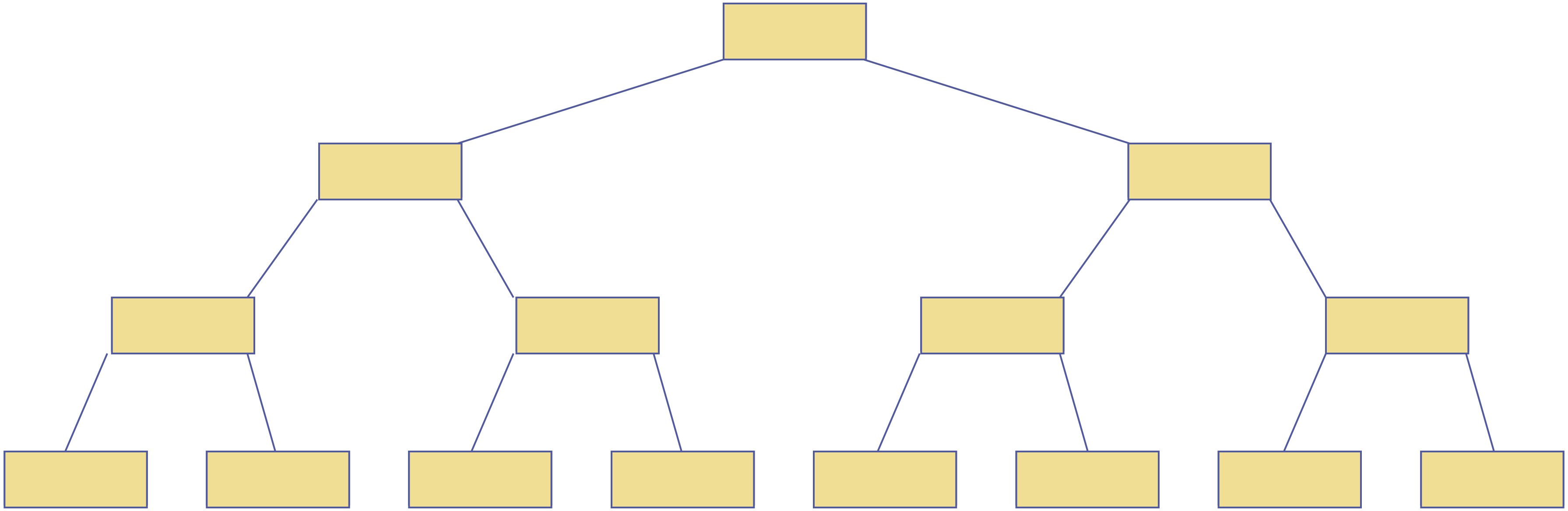
Think of this as an empty frame that
we can fill with elements ...

A perfectly balanced tree:



Think of this as an empty frame that
we can fill with elements ...

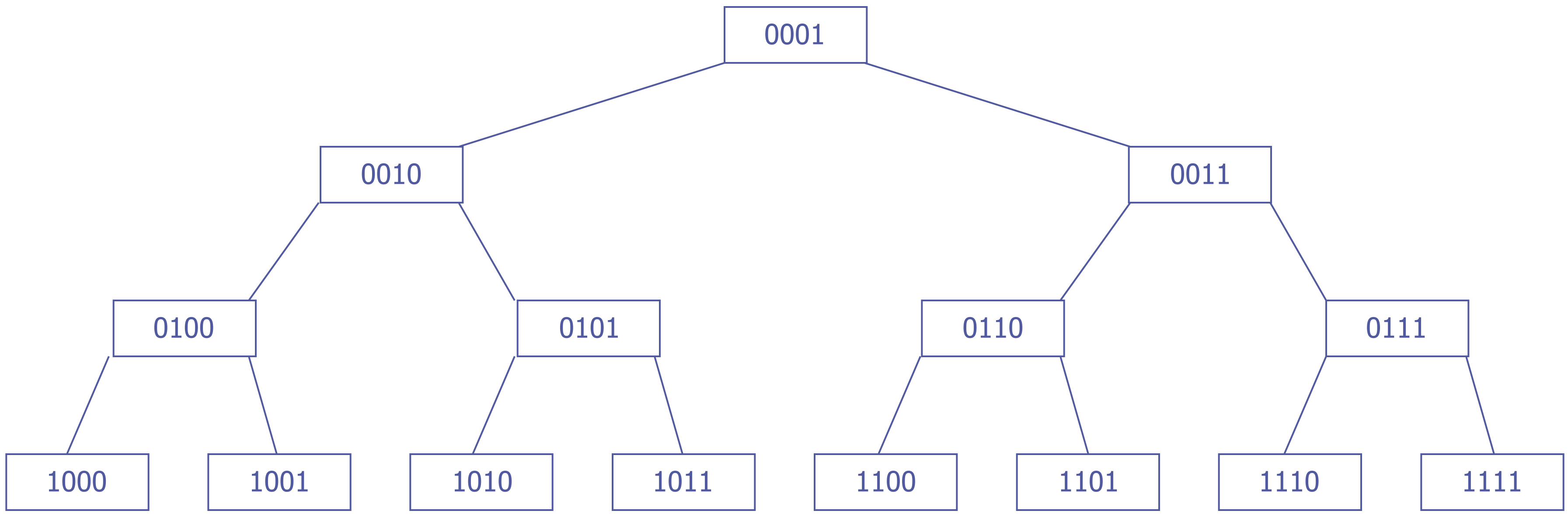
A perfectly balanced tree:



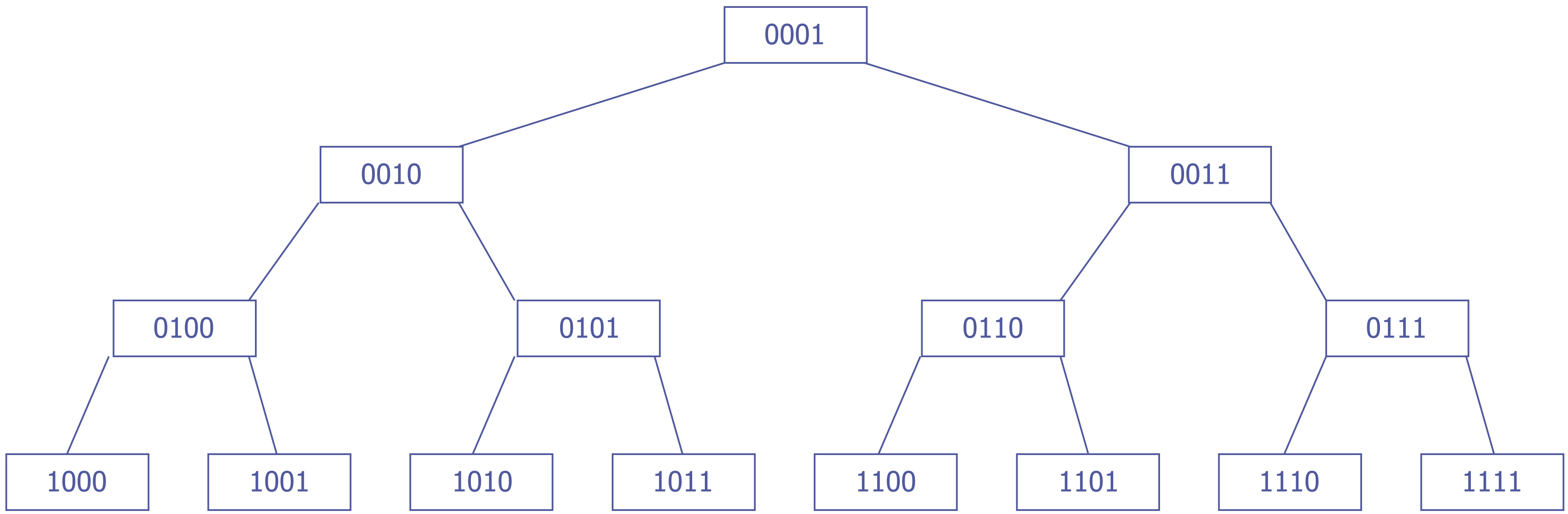
Think of this as an empty frame that we can fill with elements ...

... filling the rows up one at a time makes the tree as balanced as possible!

Number the nodes — in binary!

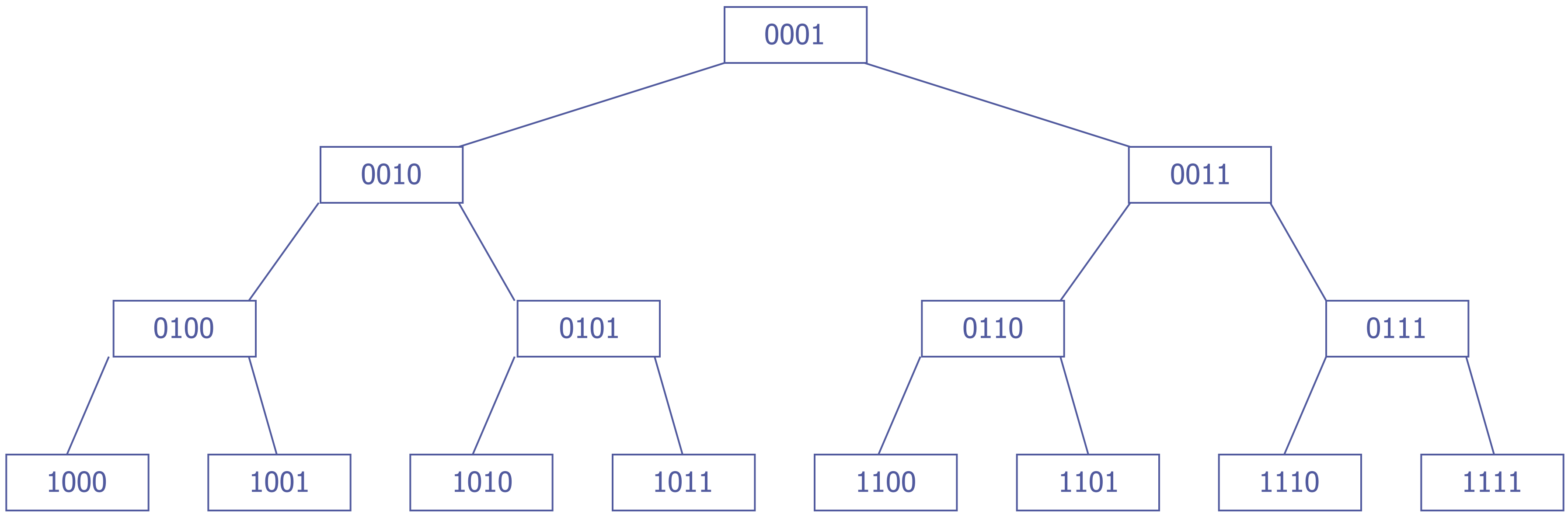


Number the nodes — in binary!

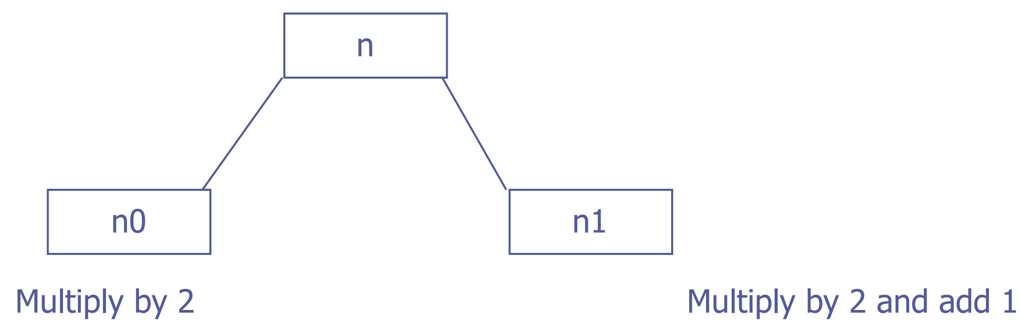


There is a common pattern at each node:

Number the nodes — in binary!



There is a common pattern at each node:



Embed a tree in an array

- A tree with $t < 2^n$ elements can be implemented using an array a and variable t:
  - elements a[1..t], (a[t +1 .. 2^n–1] are empty)
  - the root is held in position a[1]
  - left child of node a[i] is a[$2i$]
  - right child of node a[i] is a[$2i$+1]
  - parent of node a[i] is a[$\lfloor i/2 \rfloor$]

- True or False: all elements of the array with index $\geq 2^{n-1}$ represent leaf nodes

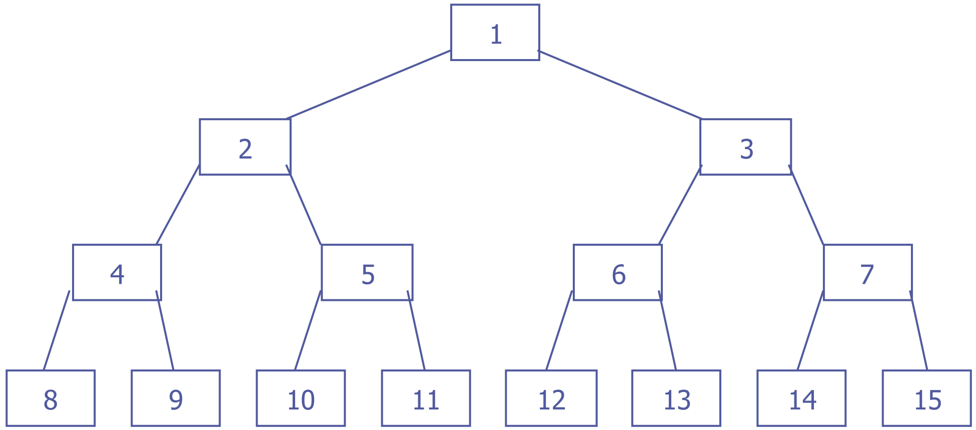
Too good to be true?

- So now we can build (almost) perfectly balanced binary trees with:
  - the smallest possible height for any number of elements stored;
  - O(1) complexity for addition.

- Where’s the flaw?

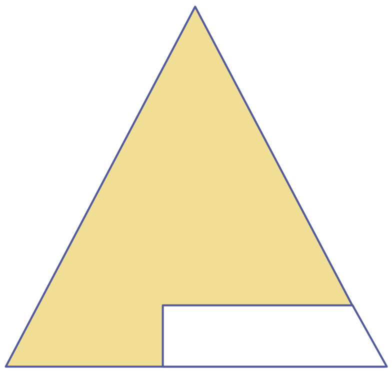
Out of order!

- Building a tree in this way does not give binary <u>search</u> trees:



- We <u>cannot</u> preserve the binary search tree invariant <u>and</u> retain O(1) time for insertion.

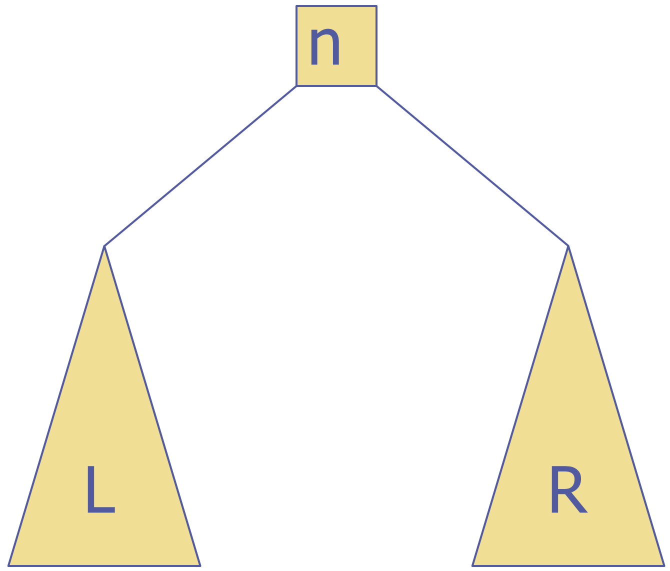
Properties of a Heap:



1. Shape Property:

   The binary tree is **essentially complete**, that is, all levels are filled except some of the rightmost leaves may be missing in the last level

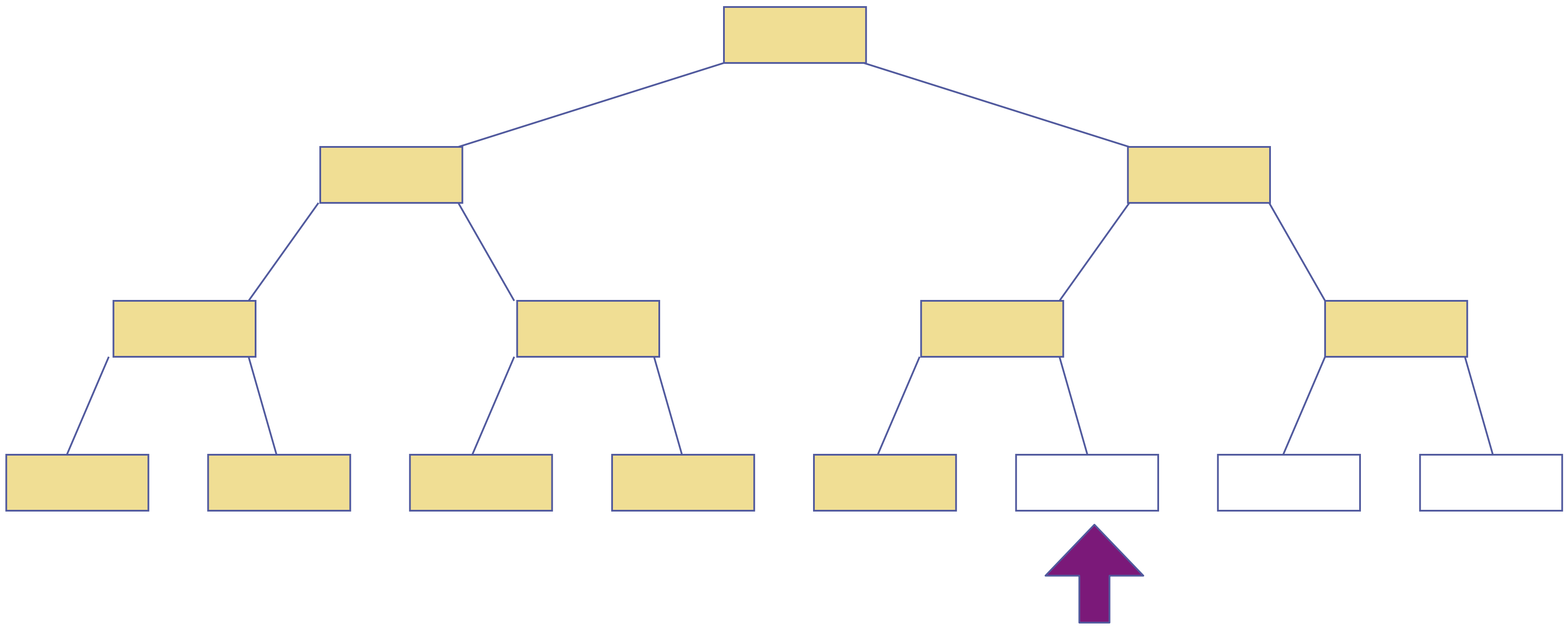
Properties of a Heap:



2. Parental dominance Property:

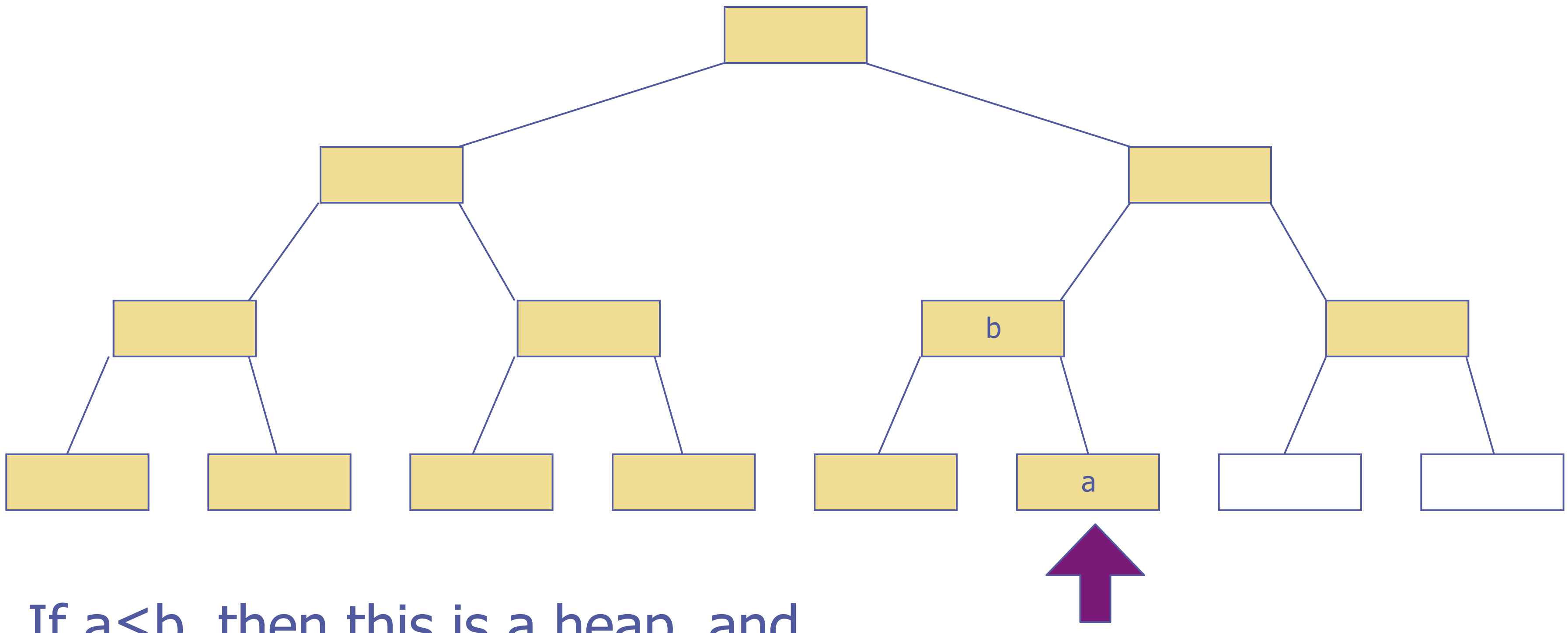
   The key in each node is greater than or equal to the keys of its children. So, all values in L are ≤n, and all values in R are <u>also</u> ≤n

Inserting an element:



The new element should be added here (takes O(1) time)

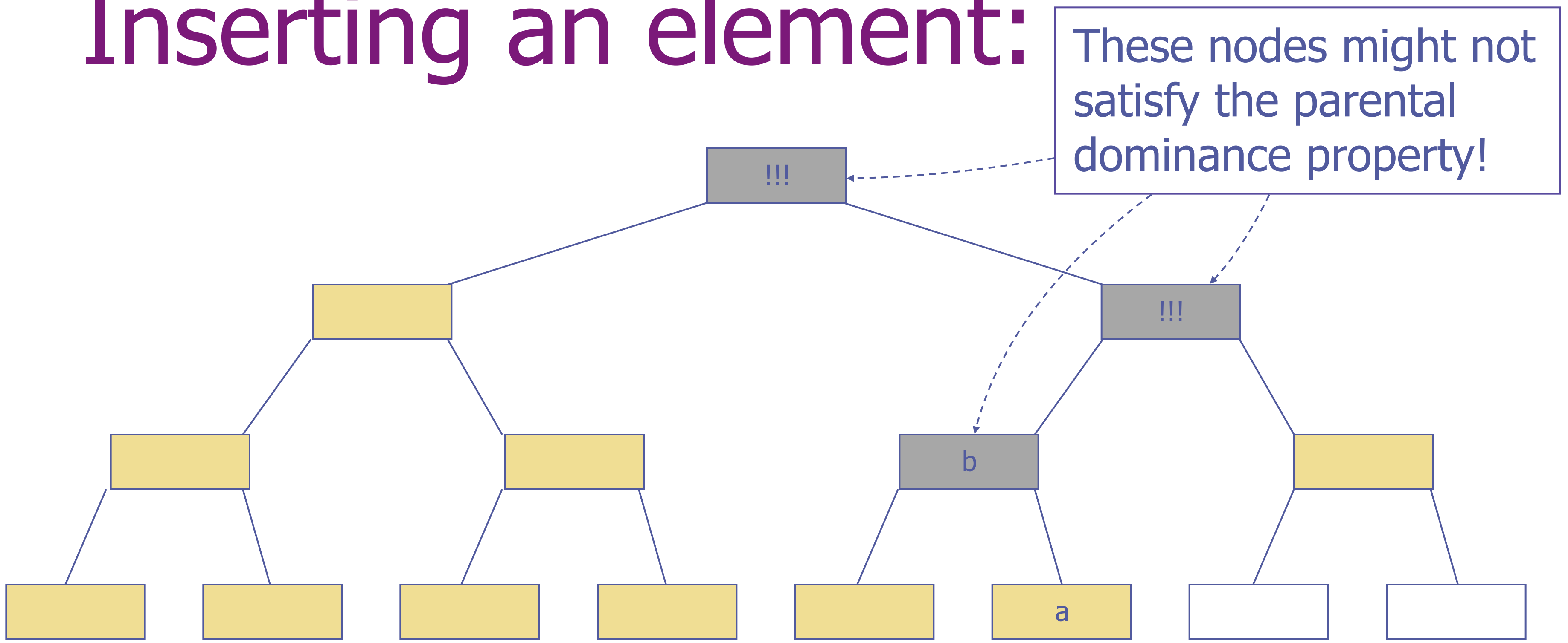
Inserting an element:



If a≤b, then this is a heap, and we are done!

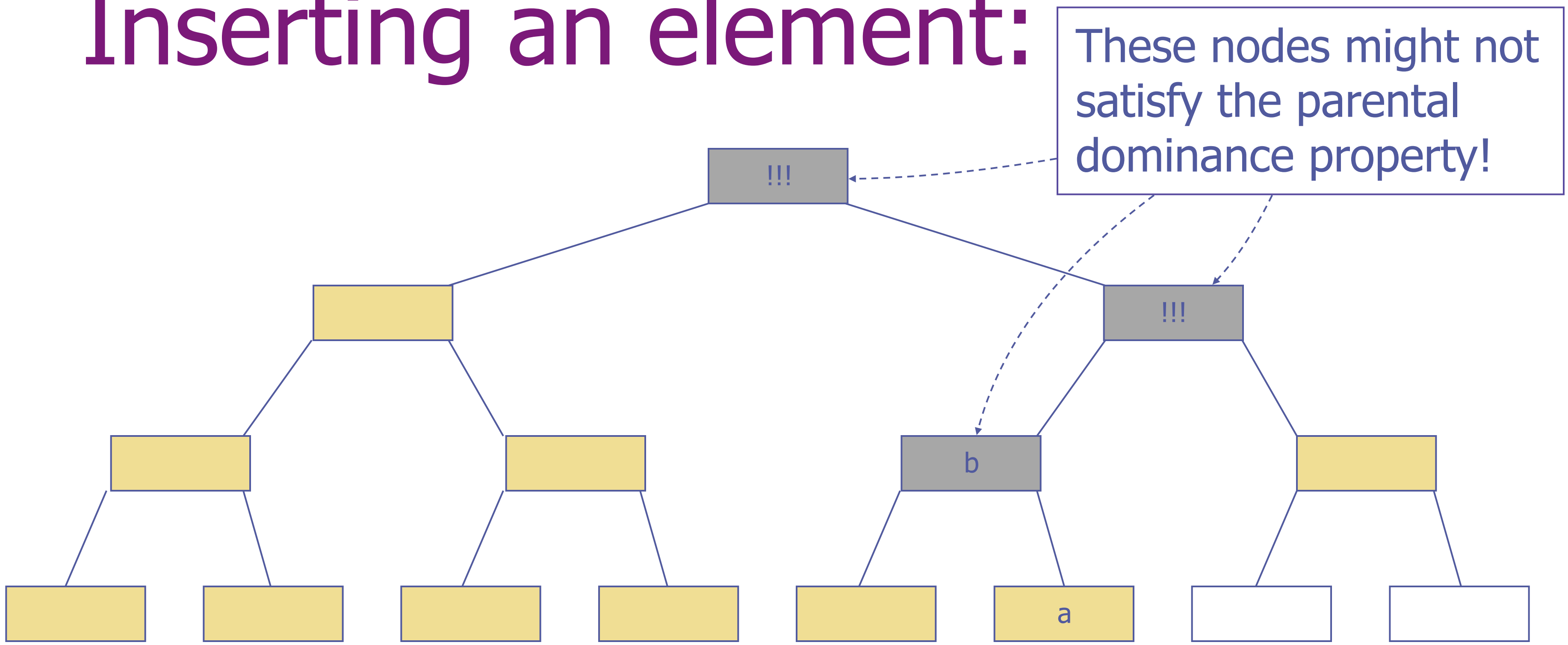
New value, a

Inserting an element:



But if a>b, then we need to do some work to restore the heap property.

Inserting an element:

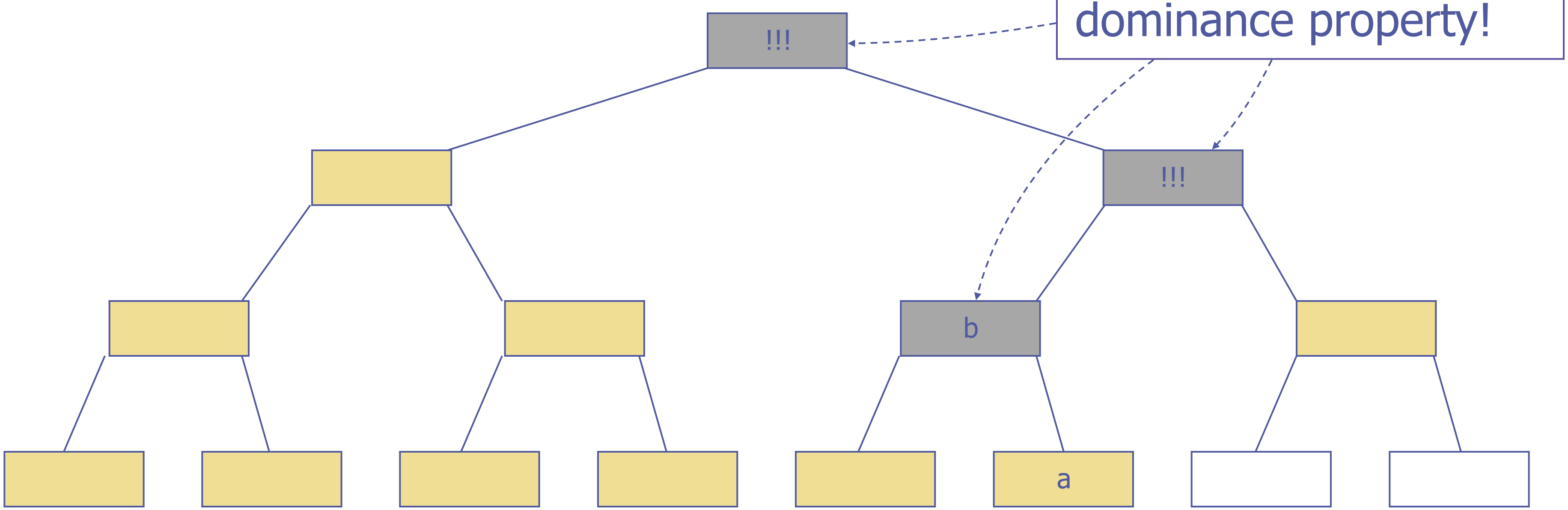


But if a>b, then we need to do some work to restore the heap property.

Start by swapping a and b ...

Inserting an element:

These nodes might not satisfy the parental dominance property!



Repeat until we're done.

Takes O(log n) time: we have to worry about the nodes on only one path in the tree.

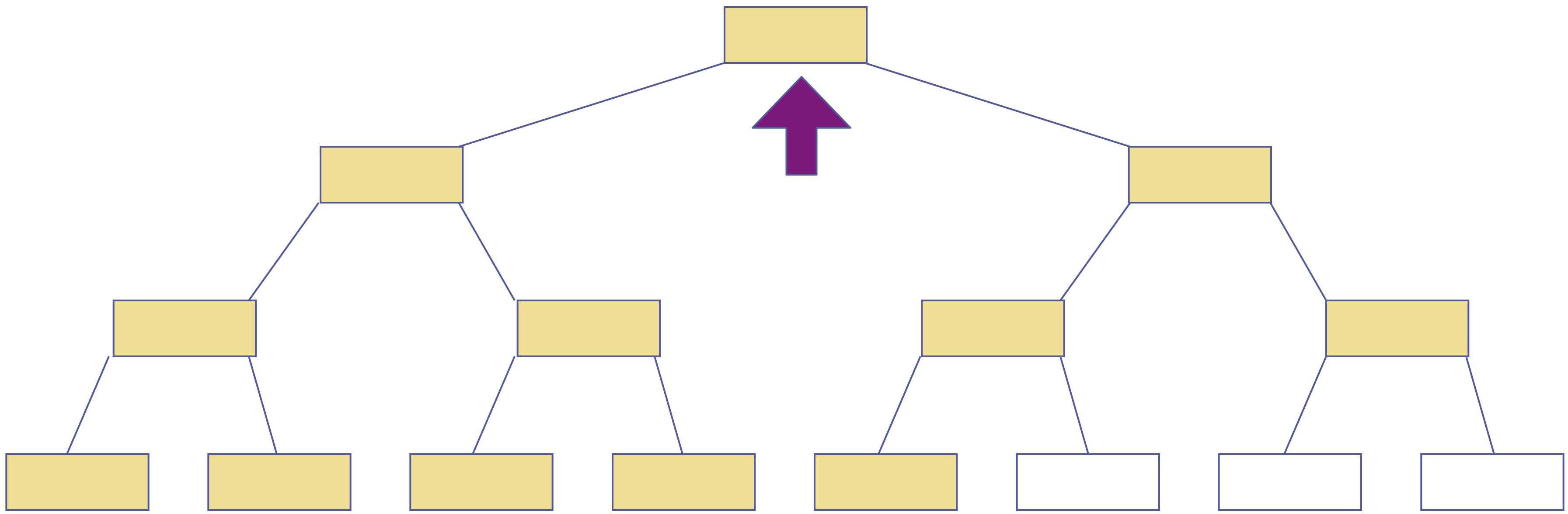
Implementation:

```
heapInsert(value) {
    size ← size + 1
    int i ← size;
    while (i>1 ∧ h[parent(i)]<value) do {
        h[i] ← h[parent(i)]
        i    ← parent(i)
    }
    h[i] ← value;
}
```

`h[]` is an array containing the heap elements;

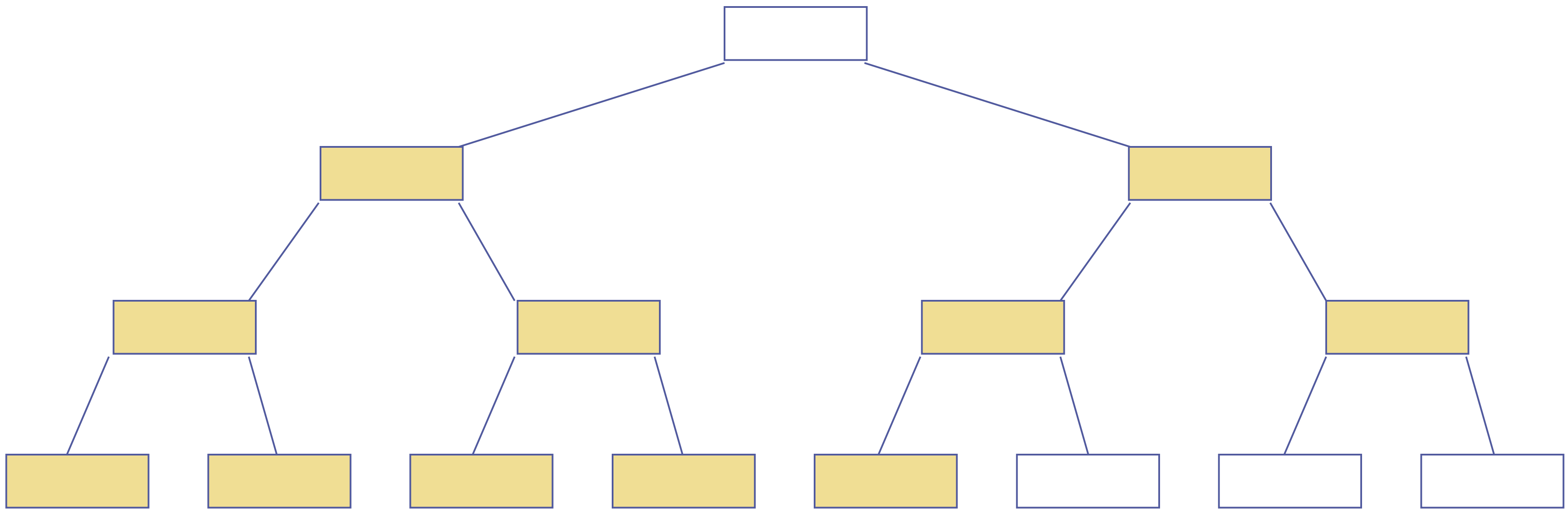
`size` is the number of entries in the heap that have been used.

Removing maximal element:



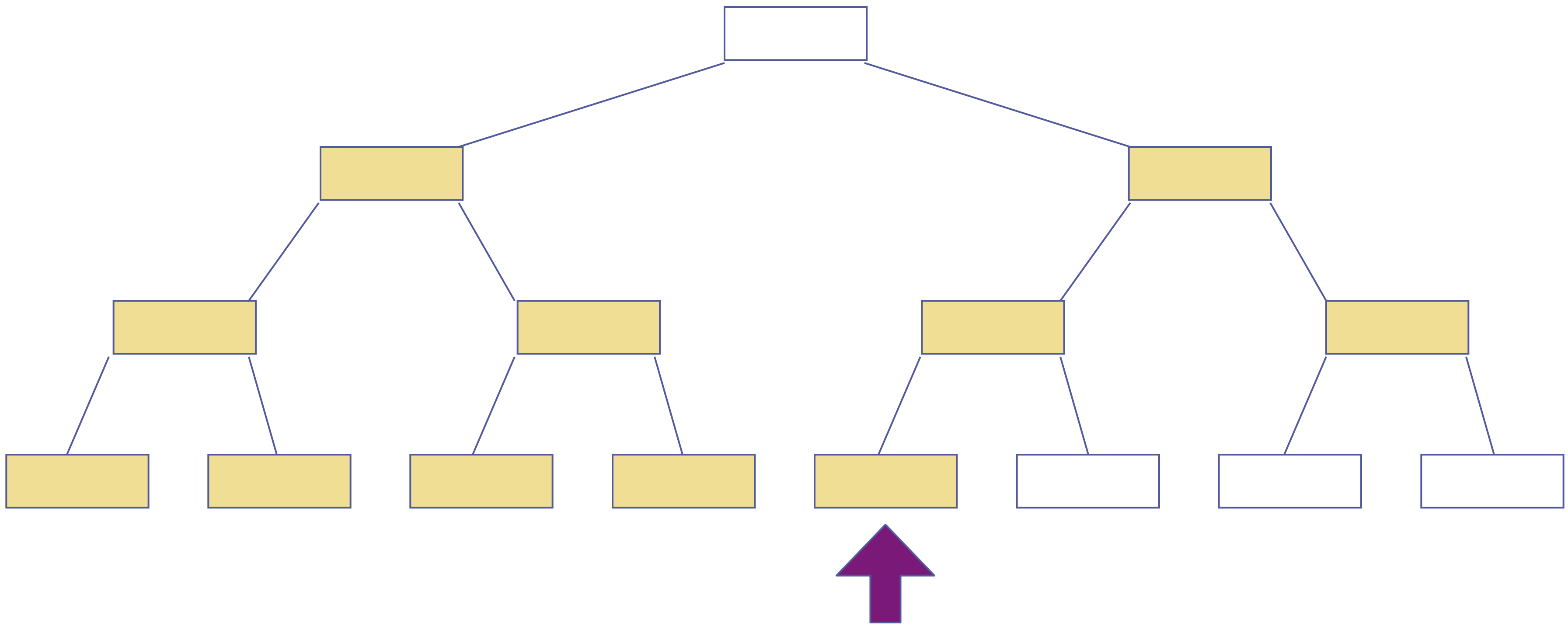
Finding the maximum element is easy! (takes O(1) time)

Removing maximal element:



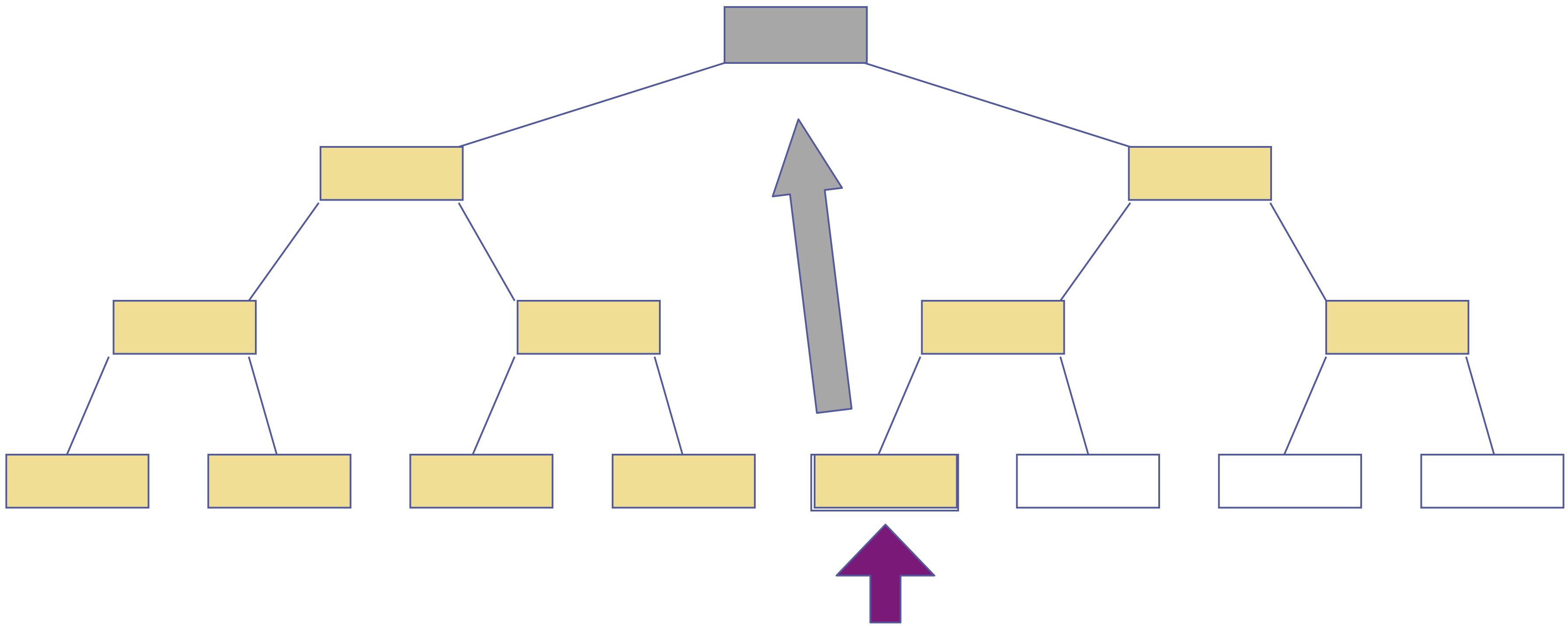
We can fill the gap with the last value in the array (takes O(1) time)

Removing maximal element:



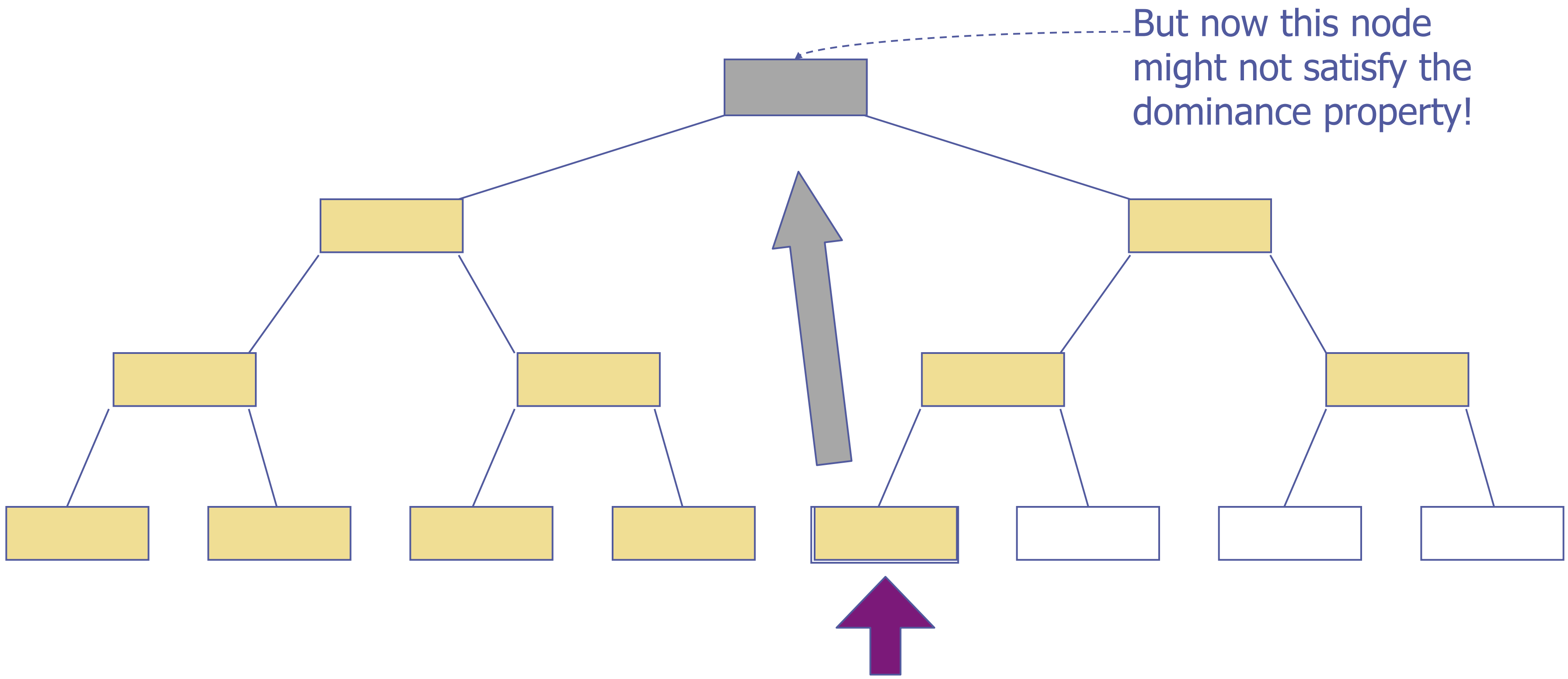
We can fill the gap with the last value in the array (takes O(1) time)

Removing maximal element:



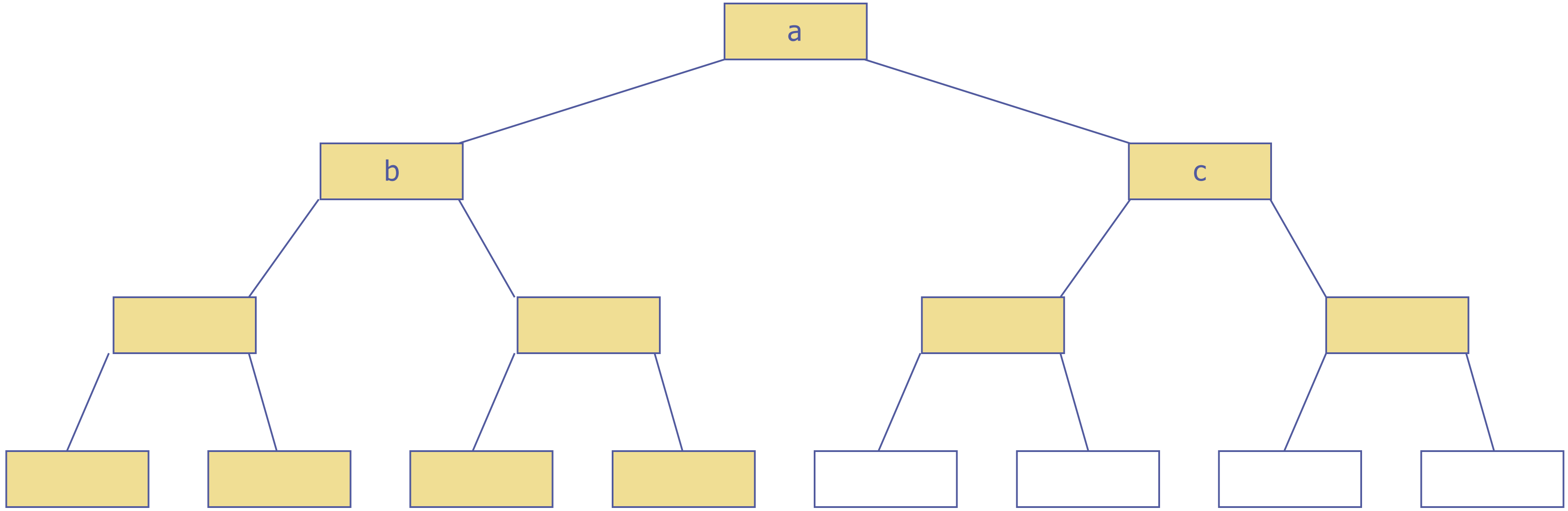
We can fill the gap with the last value in the array (takes O(1) time)

Removing maximal element:



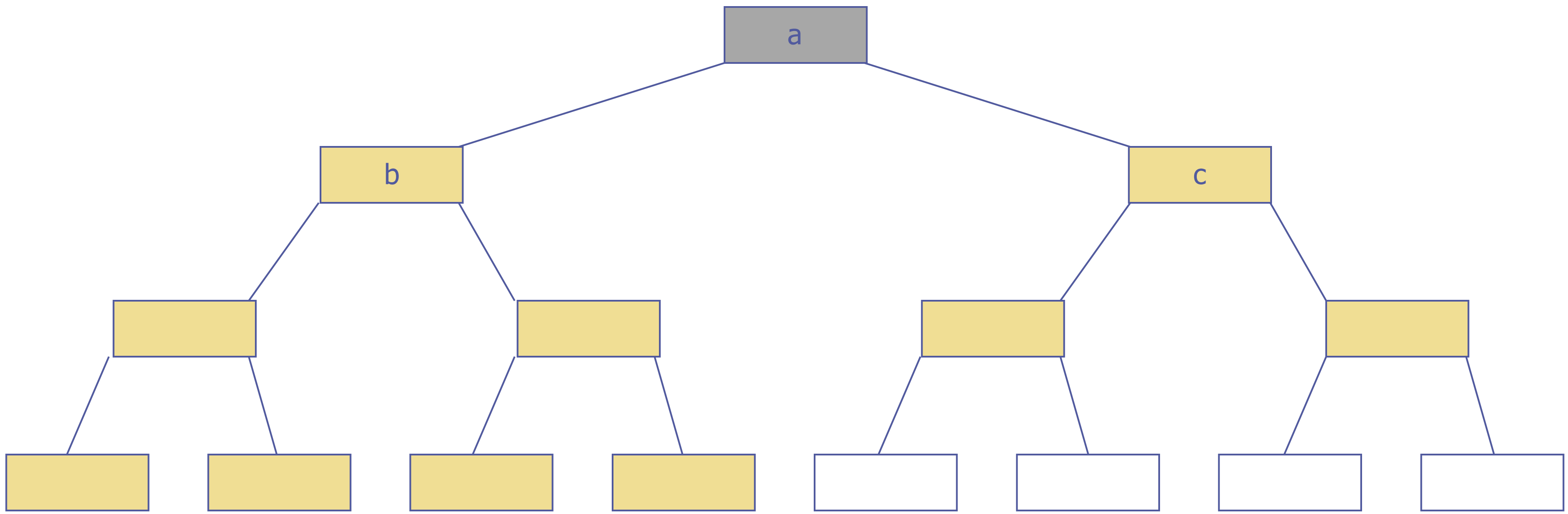
We can fill the gap with the last value in the array (takes O(1) time)

Removing maximal element:



If a>b and a>c, then this is a heap, and we are done!

Removing maximal element:

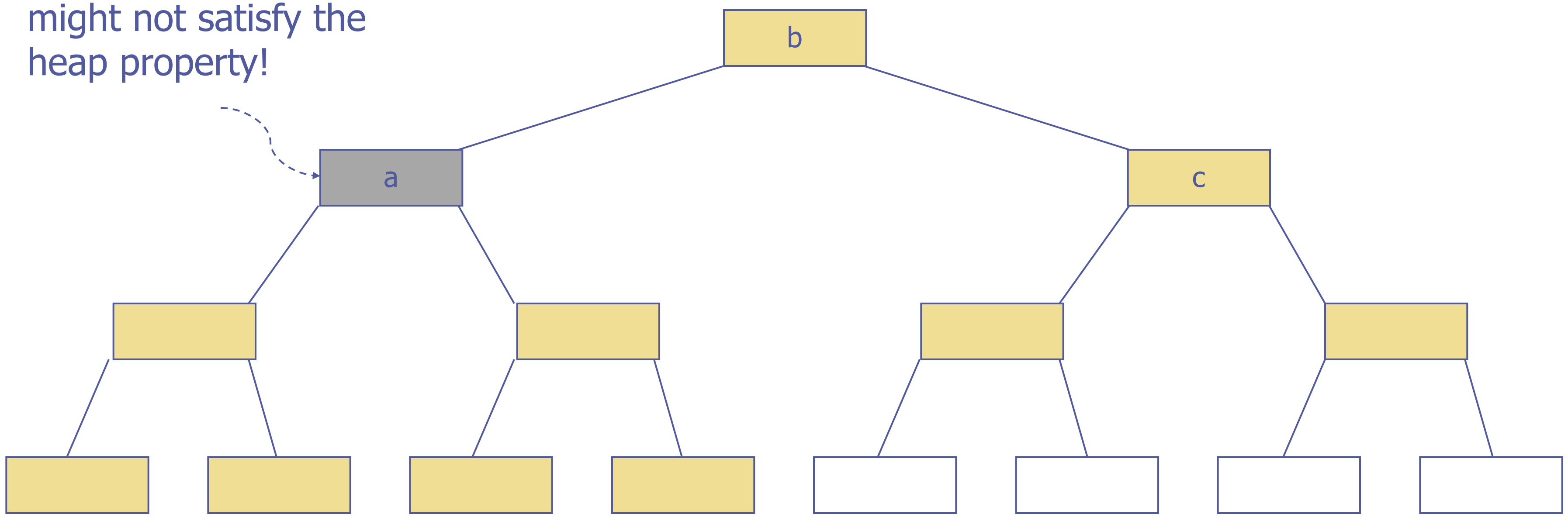


Otherwise, suppose b>a and b>c.

Then we can swap a with b …

Removing maximal element:

But now this node might not satisfy the heap property!



Repeat until we're done.

Takes $O(\log n)$ time: we have to worry about the nodes on only one path in the tree.

Implementation:

```
heapExtractMax() {
    size ← size - 1
    int max ← h[1];
    h[1]    ← h[size];
    heapify(1);
    return max;
}
```

Implementation:

```
heapify(i) {
   l ← left(i); r ← right(i);
   largest ← i;
   if (l≤size) {
      if (h[l]>h[i])
         largest ← l;
      if (r≤size ∧ h[r]>h[largest])
         largest ← r;
   }
   if (largest≠i) {
      h.swap(i, largest);
      heapify(largest);
   }
}
```

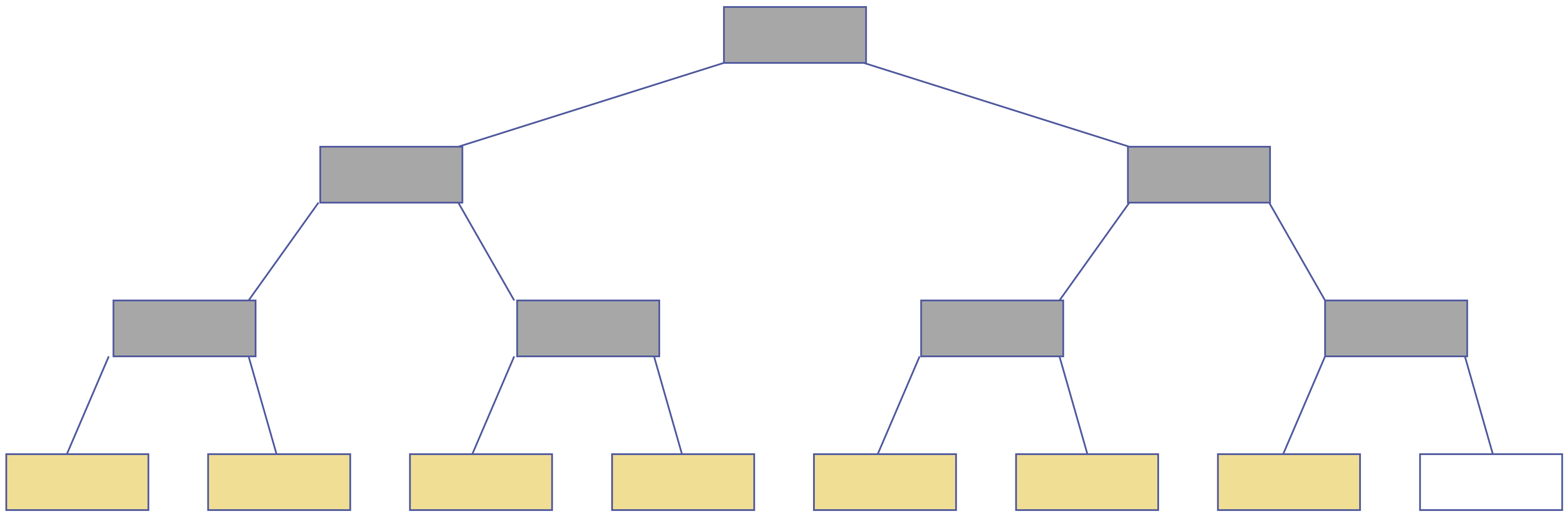
Priority queues:

- A priority queue is a variation on the queue data structure with a "highest-priority first out" policy.

- More concretely, a priority queue supports operations to:
  - Add an element, and
  - Remove highest priority element.

- Heaps can be used as an implementation of priority queues—one of the most common uses of heaps in practice.

Building a heap:



Suppose we start with an arbitrary array of values.

Run `heapify` on each of the interior nodes, starting at the bottom, and working back to the root. Now we have a heap!

Implementation:

```
buildHeap() {
    size ← h.length;
    for i from size/2 downto 1 do {
        heapify(i);
    }
}
```

Complexity:

- To a first approximation: there are $O(n)$ calls to `heapify`, and $O(\log n)$ steps for each such call, giving a total:

$$O(n \log n)$$

Complexity:

- To a first approximation: there are $\mathrm{O}(n)$ calls to `heapify`, and $\mathrm{O}(\log n)$ steps for each such call, giving a total:

$$\mathrm{O}(n \log n)$$

  - But we can do better than this!
  - Many of the calls to `heapify` involve trees with heights that are $< \log n$.

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right) = O(n)$$

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}}\right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}}\right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h}}\right) = O(n)$$

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

cost of heapify on trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

cost of heapify on trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

cost of heapify on trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

converges to 2

- The total cost of buildHeap is:

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

trees of height h

cost of heapify on trees of height h

- Simplifying:

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

converges to 2

Spanning Trees

Spanning Trees

- If e is a minimum-weight edge in a connected graph, then e must be an edge in <u>at least one</u> minimum spanning tree

  - True or False?

Spanning Trees

- If e is a minimum-weight edge in a connected graph, then e must be an edge in all minimum spanning trees of the graph
  - True or False?

Spanning Trees

- If every edge in a connected graph G has a distinct weight, then G must have exactly one minimum spanning tree
  - True or False?

Kruskal’s Algorithm

Building bridges:

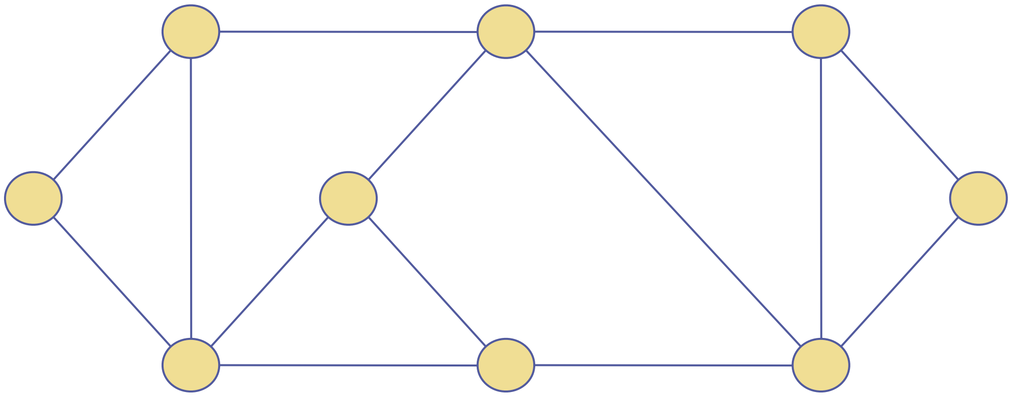
- Suppose that we want to link a group of n small islands together with bridges.
- There will be many possible ways to do this, each corresponding to a connected graph, with the islands as vertices and bridges as edges.
- What is the minimum number of bridges that we will need to build?

Spanning trees:

- A spanning tree T of a connected graph G = (V,E) is a subgraph of G that is:
  - connected;
  - acyclic;
  - includes all of V as vertices.

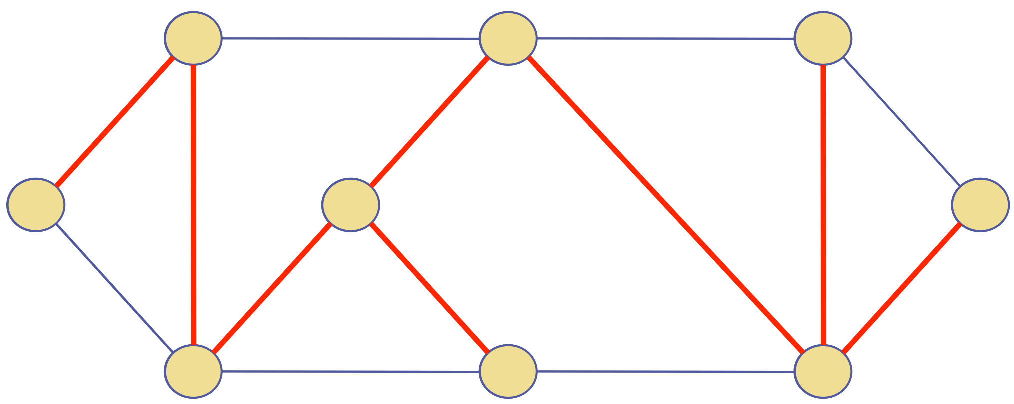
Spanning trees:

- A spanning tree T of a connected graph G = (V,E) is a subgraph of G that is:
  - connected;
  - acyclic;
  - includes all of V as vertices.



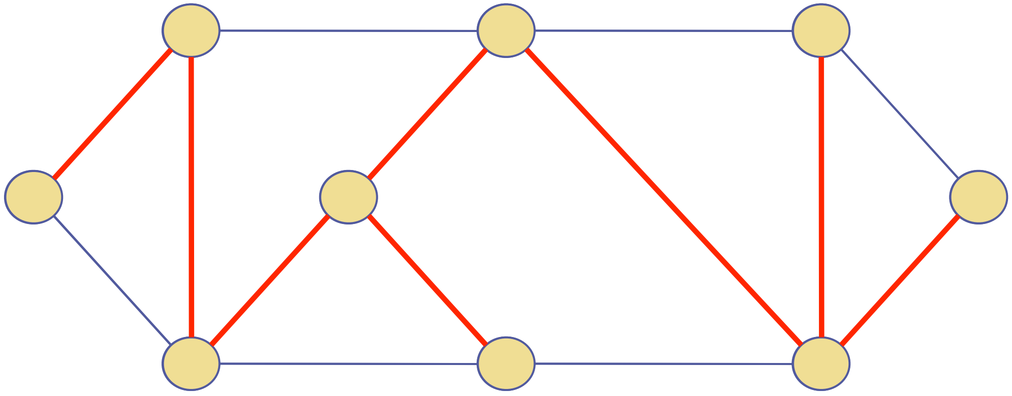
Spanning trees:

- A spanning tree T of a connected graph G = (V,E) is a subgraph of G that is:
  - connected;
  - acyclic;
  - includes all of V as vertices.



Spanning trees:

- A spanning tree T of a connected graph G = (V,E) is a subgraph of G that is:
  - connected;
  - acyclic;
  - includes all of V as vertices.



- Any spanning tree has |V|−1 edges.

Growing a forest:

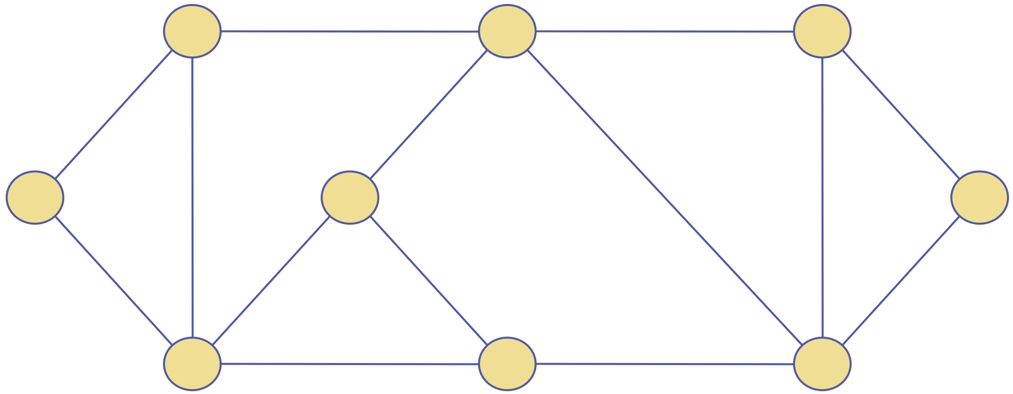
- Find a spanning tree for connected graph G=(V,E):

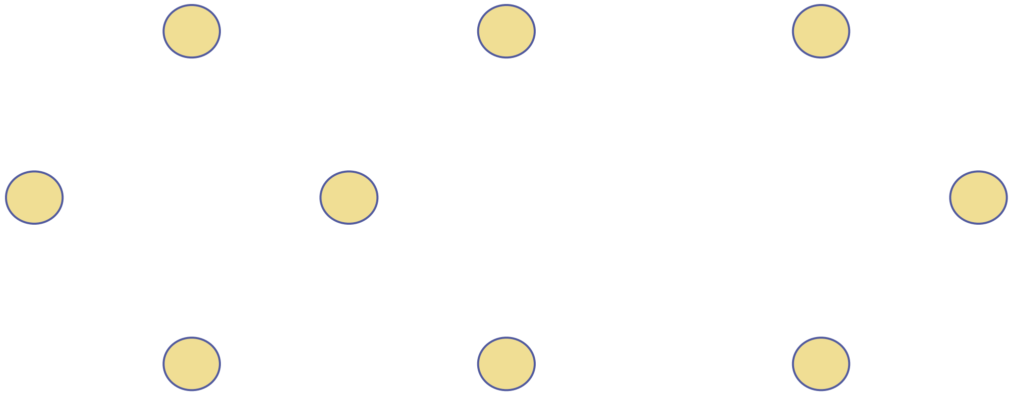
```
partition V into |V| singleton sets of the form {v}.
let E_T be an empty set of edges.
for each edge (u,v) in E:
   let S_u be the set containing u
   let S_v be the set containing v
   if S_u ≠ S_v, then
      replace S_u and S_v with S_u ∪ S_v
      add (u,v) to E_T
return (V, E_T) as the spanning tree
```

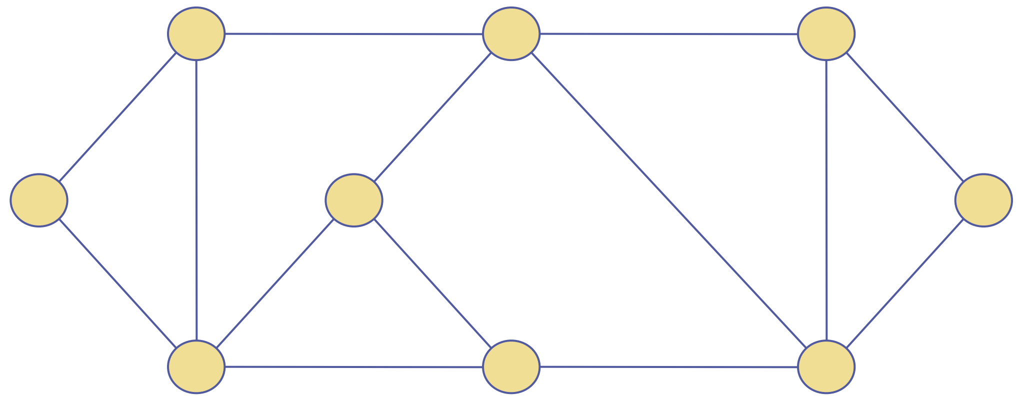
- We start with |V| sets …

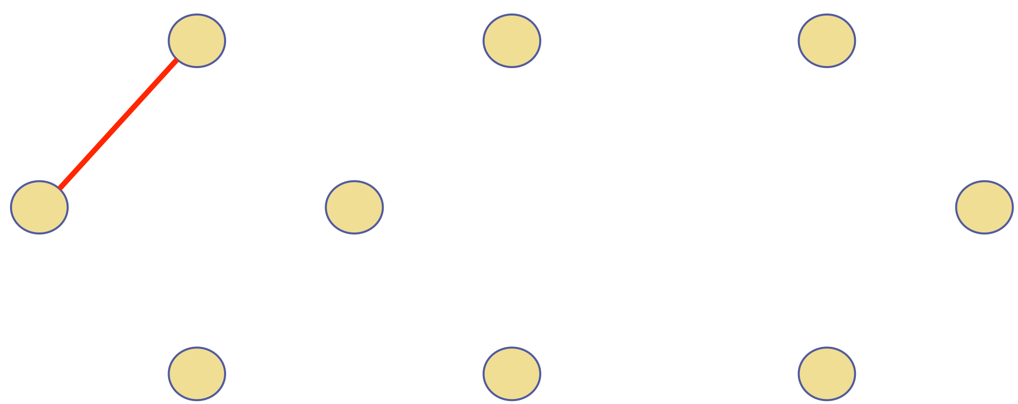
  … we end up with just 1 set.

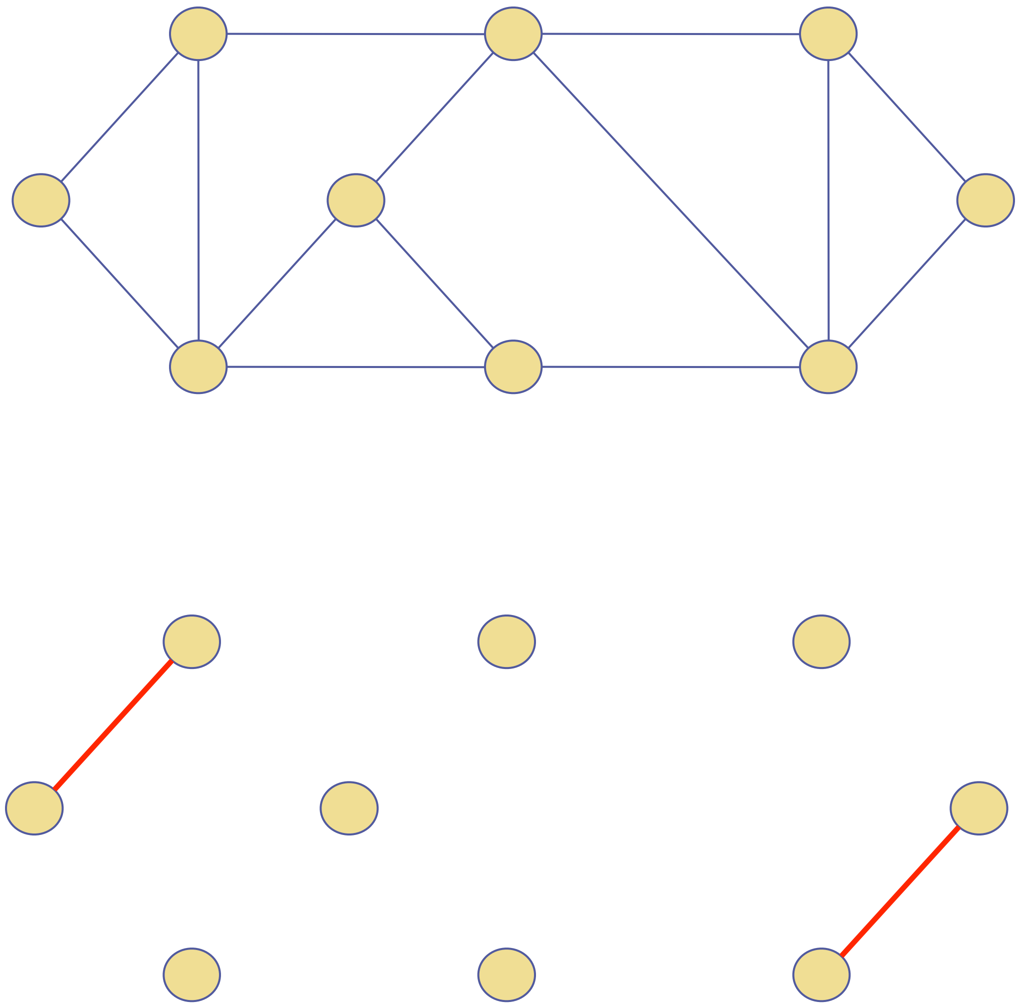
- Hence: |V|−1 unions, |V|−1 edges added to E_T.

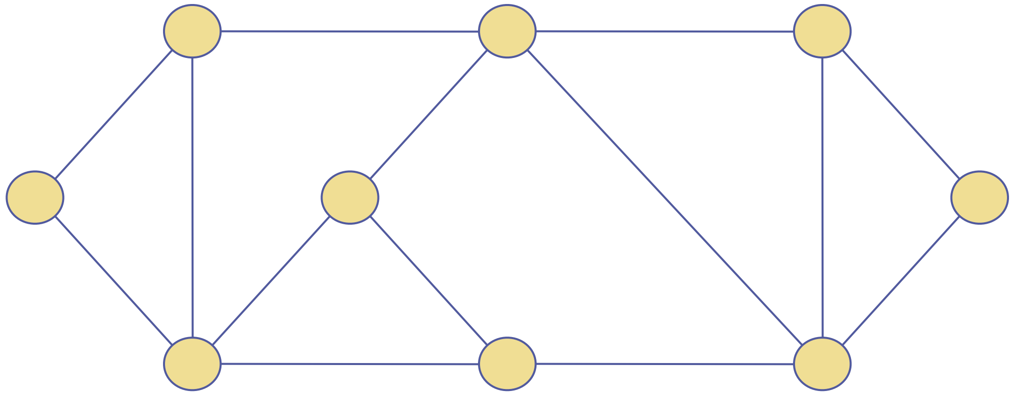


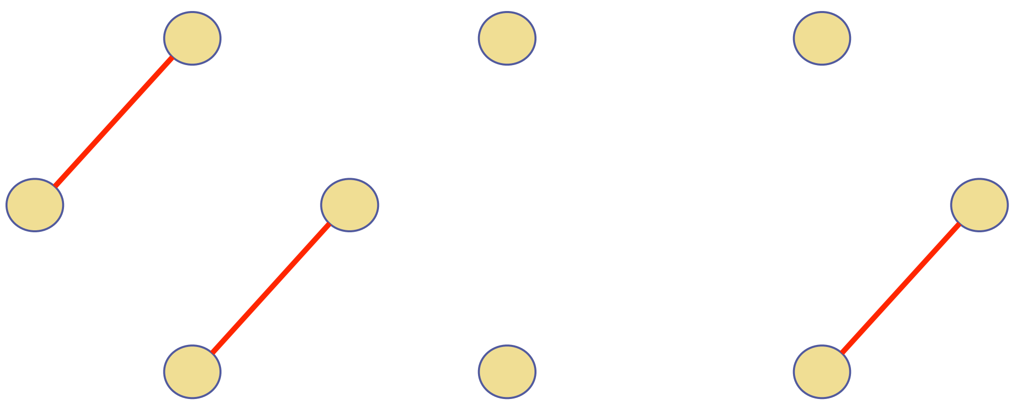


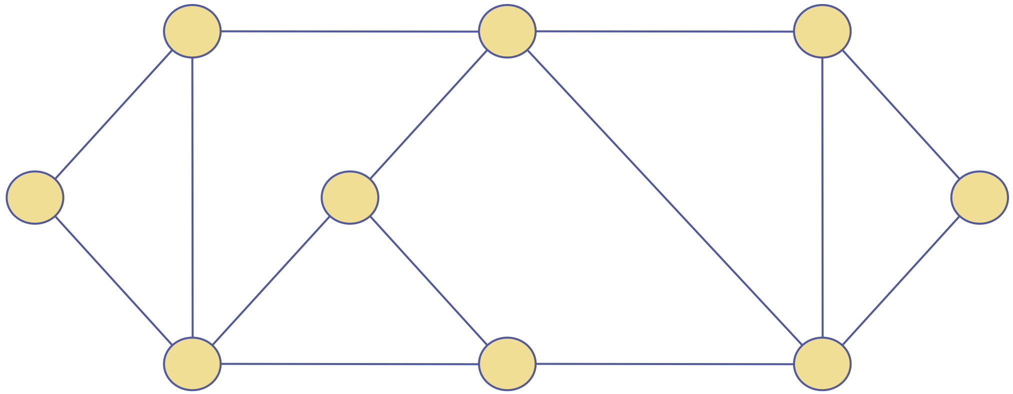


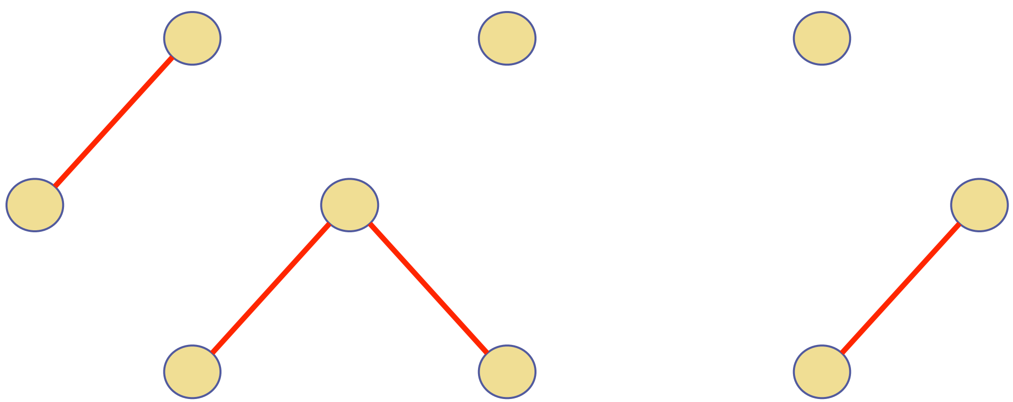


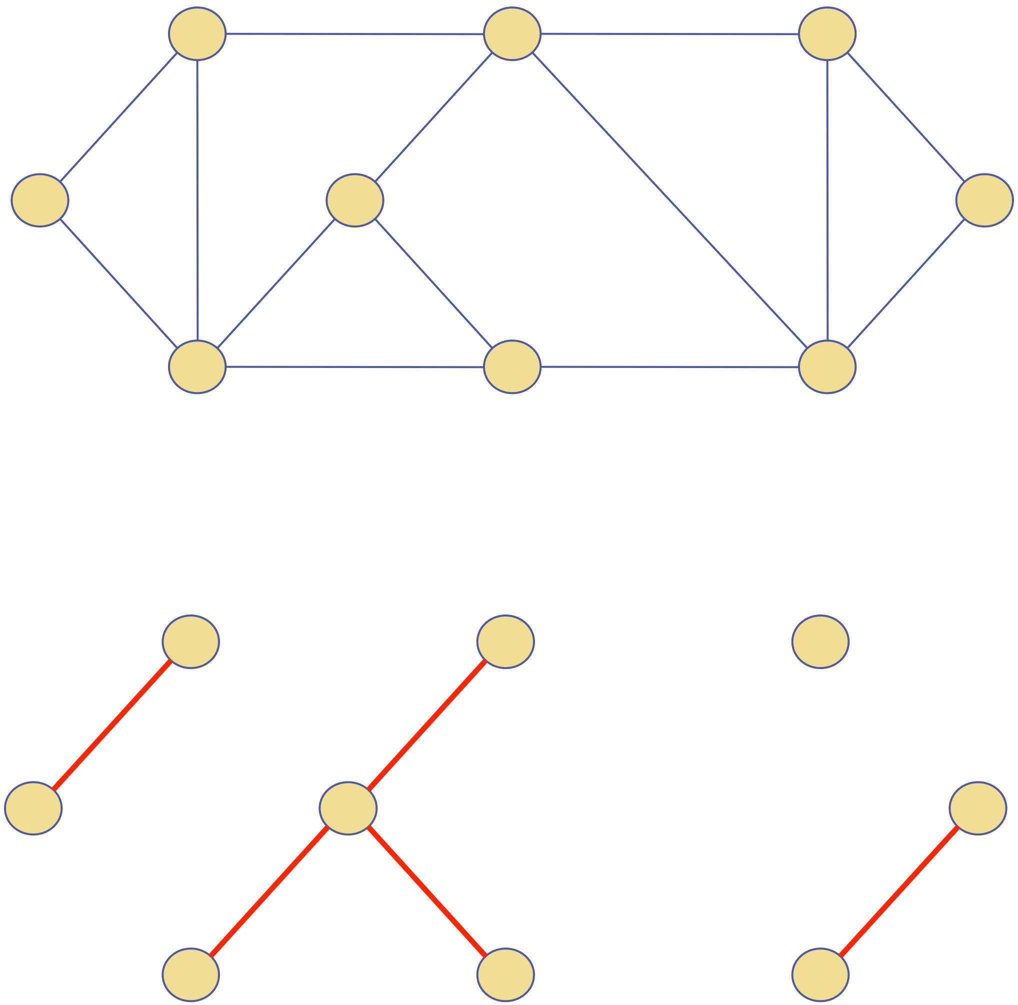


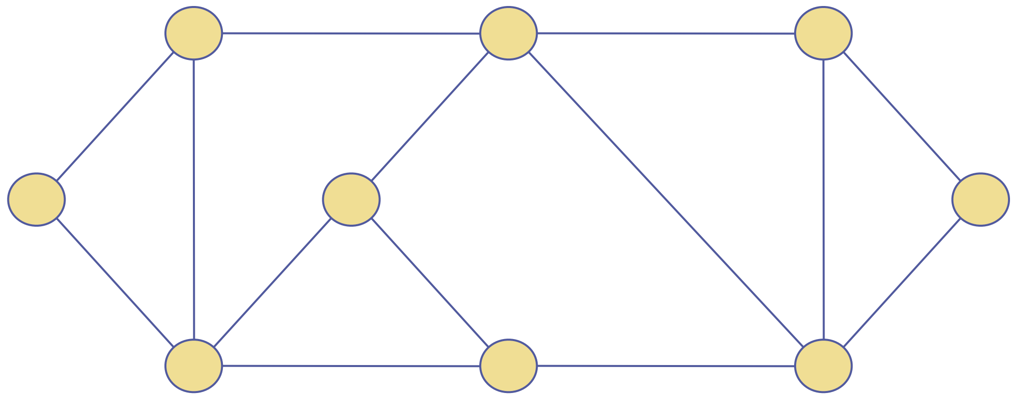


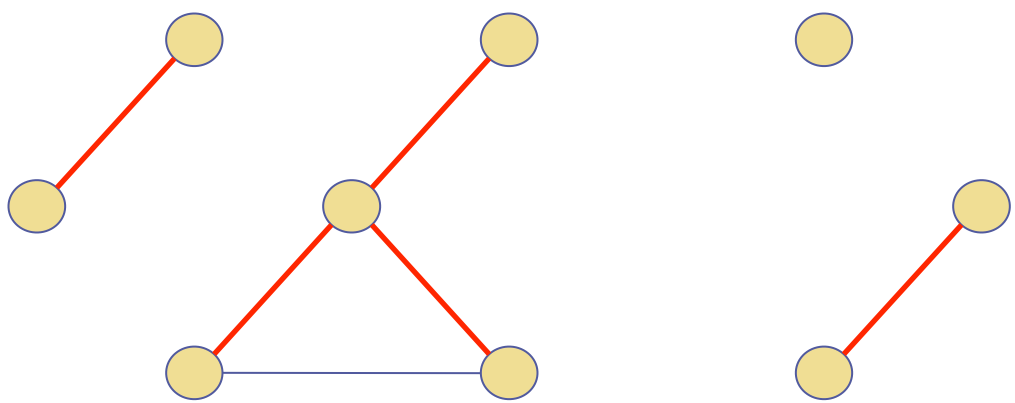


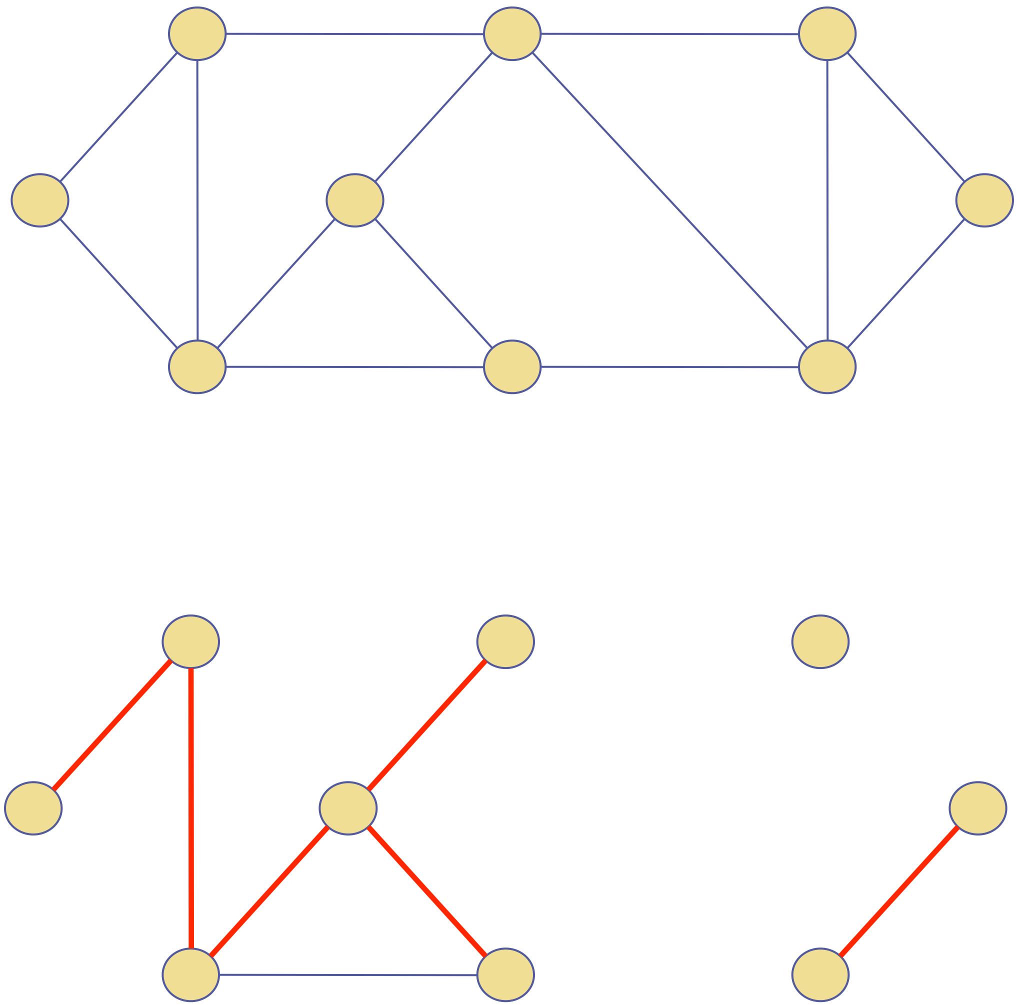


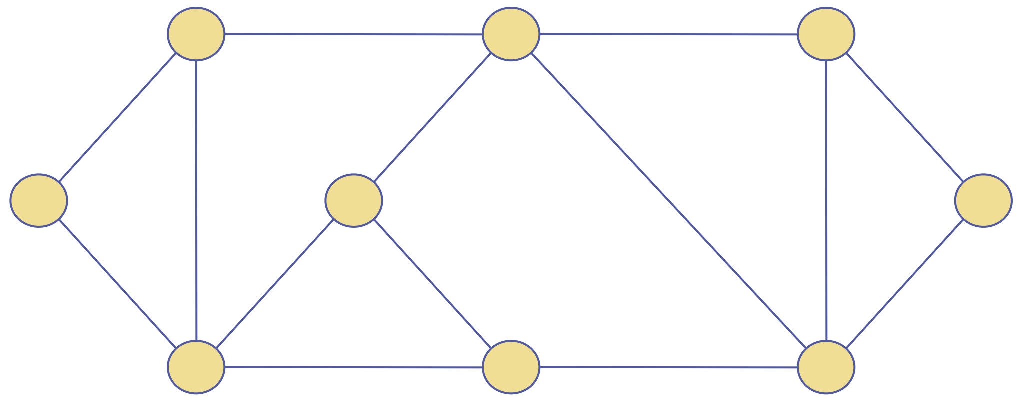


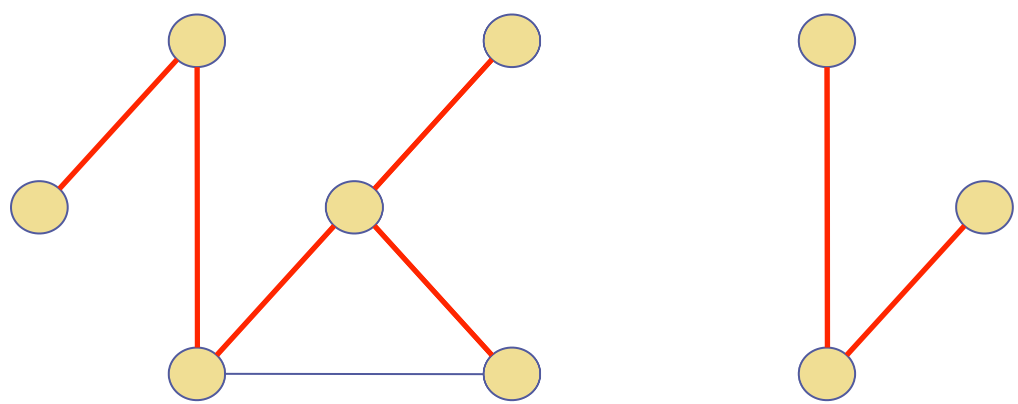


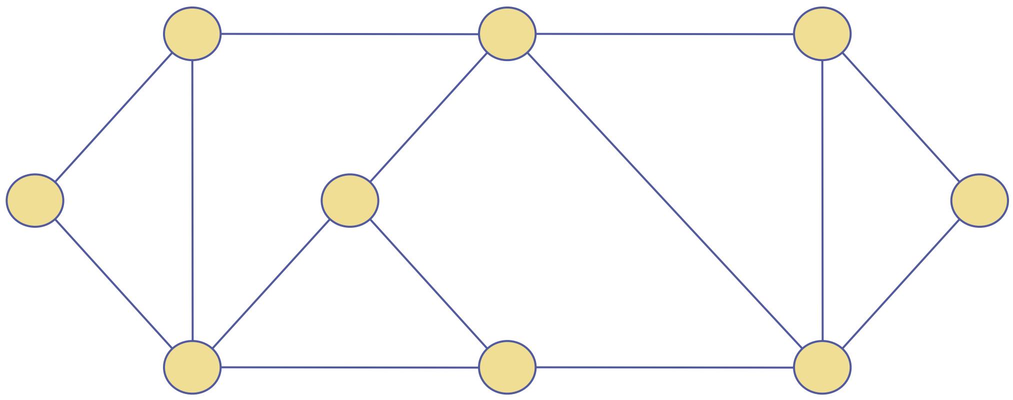


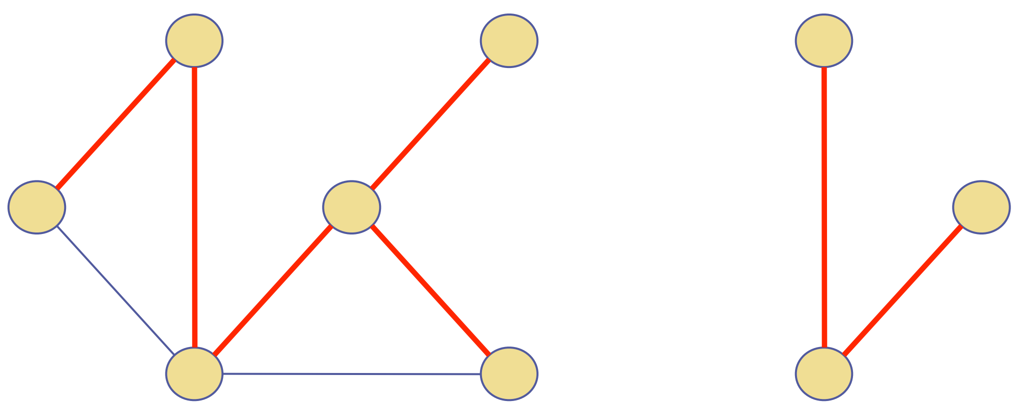


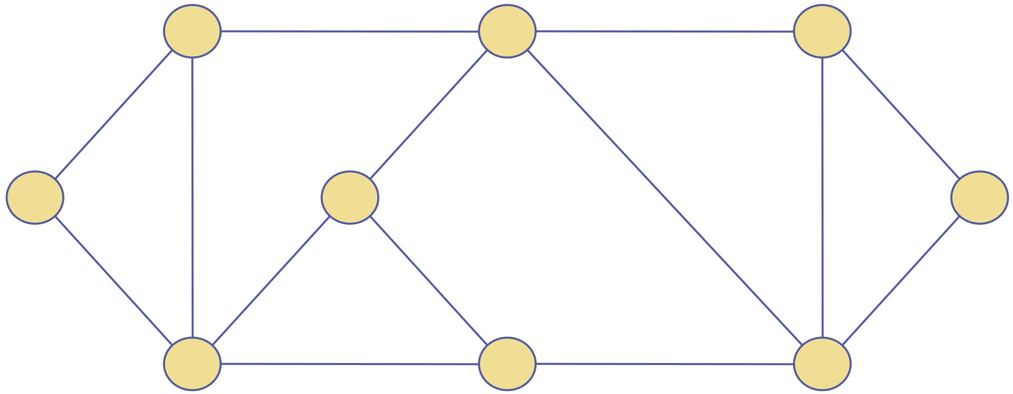


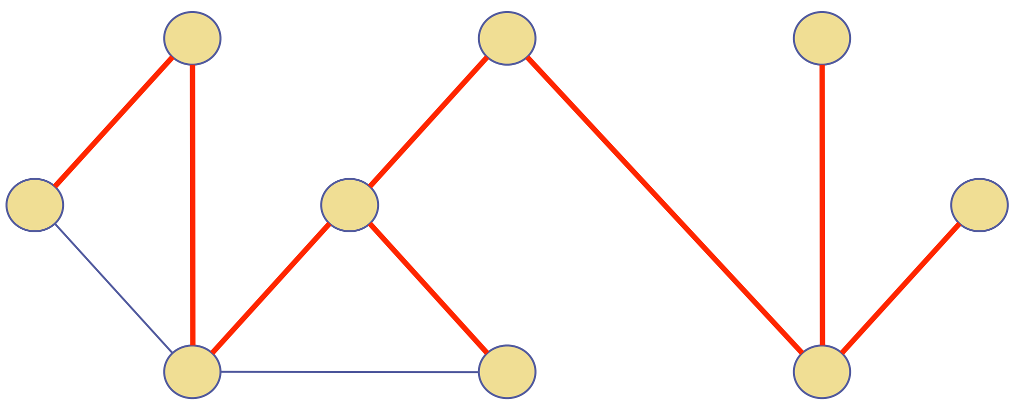


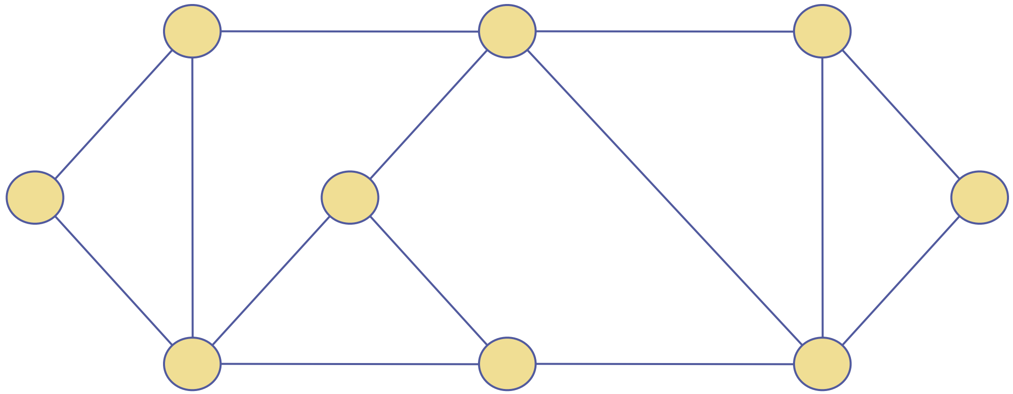


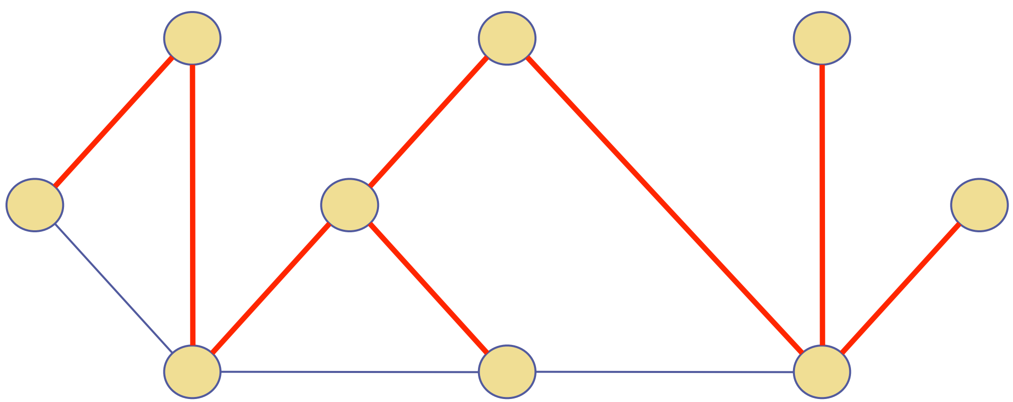


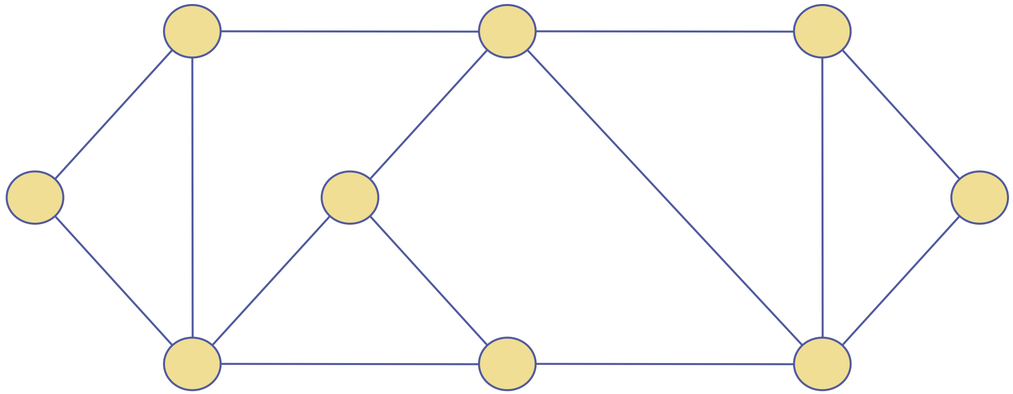


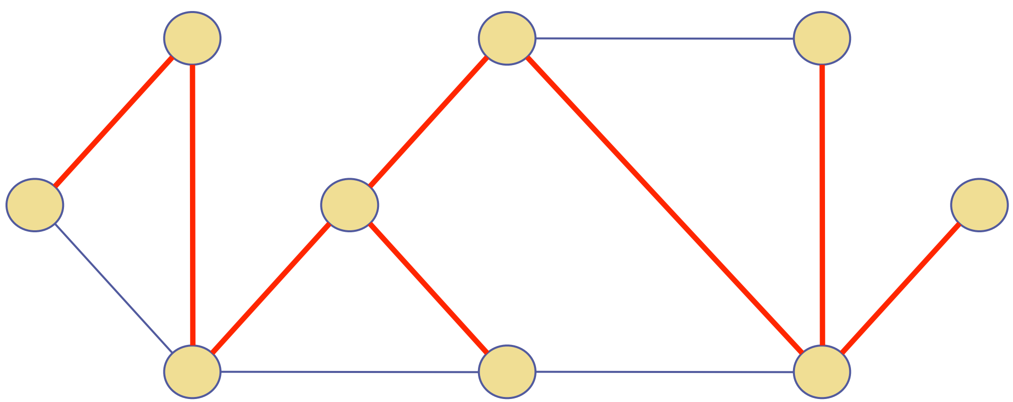


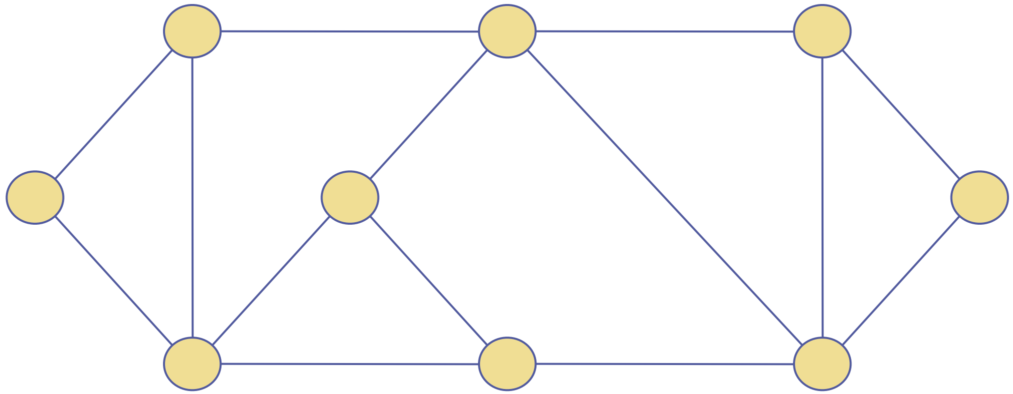


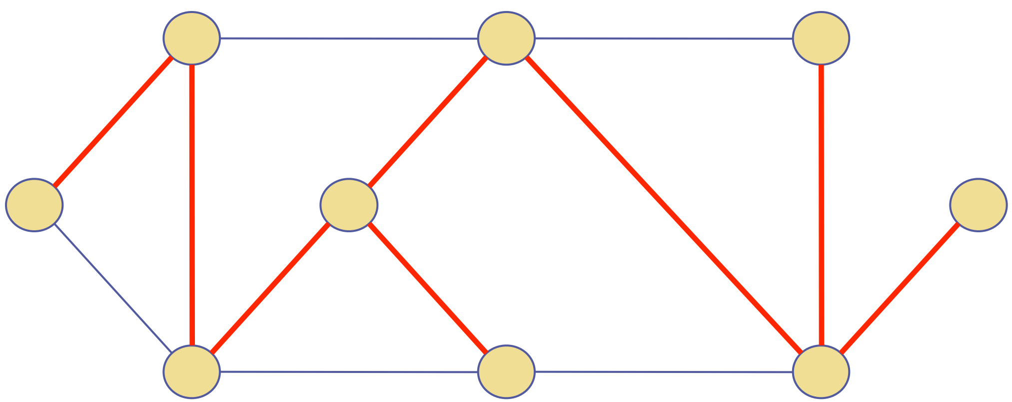


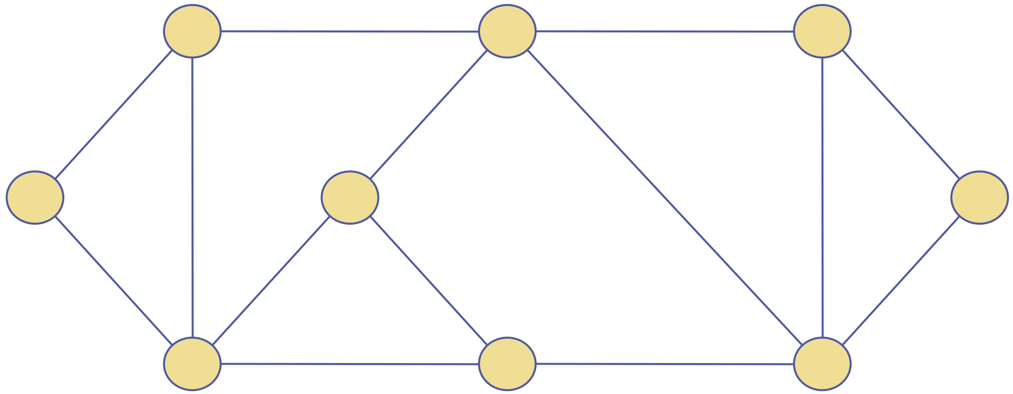


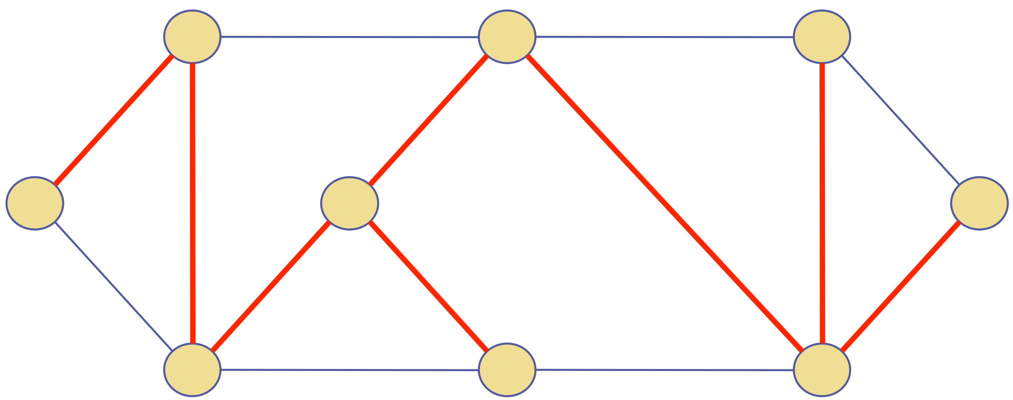












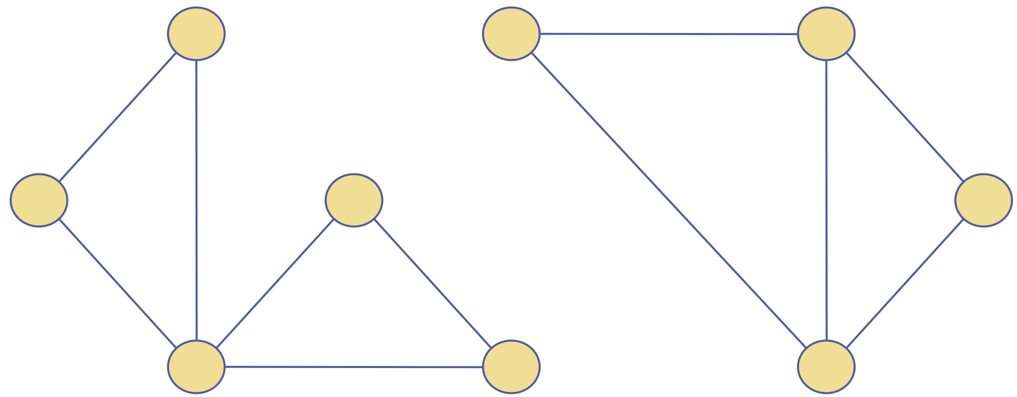
Calculating connected components:

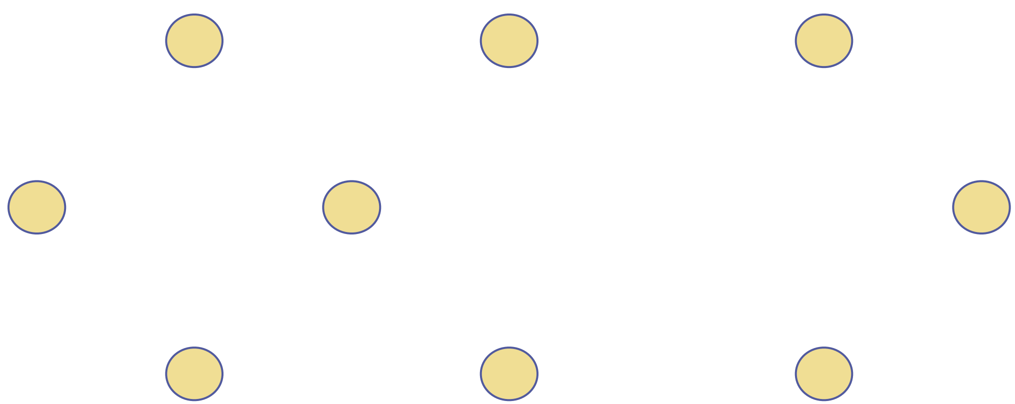
- What if G=(V,E) is not connected?

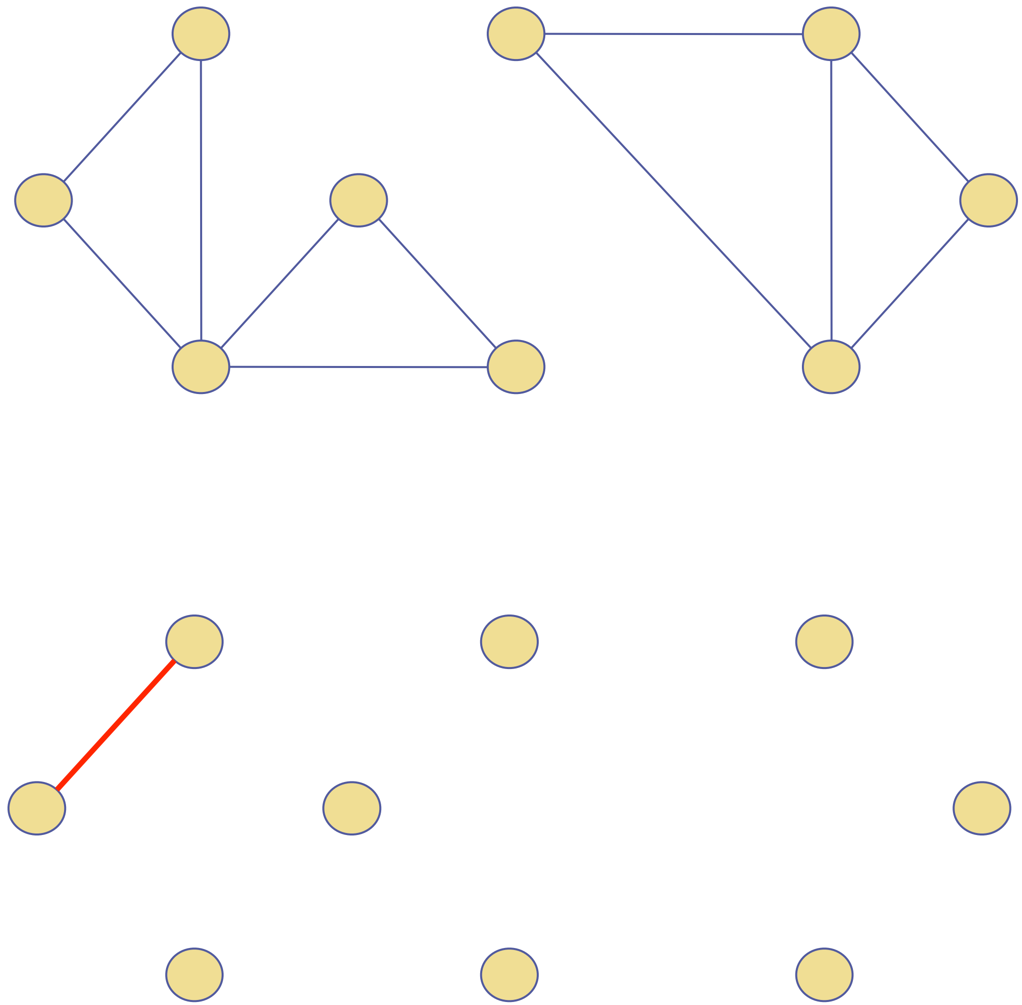
```
partition V into |V| singleton sets of the form {v}.
let E_T be an empty set of edges.
for each edge (u,v) in E:
   let S_u be the set containing u
   let S_v be the set containing v
   if S_u ≠ S_v, then
      replace S_u and S_v with S_u ∪ S_v
      add (u,v) to E_T
```

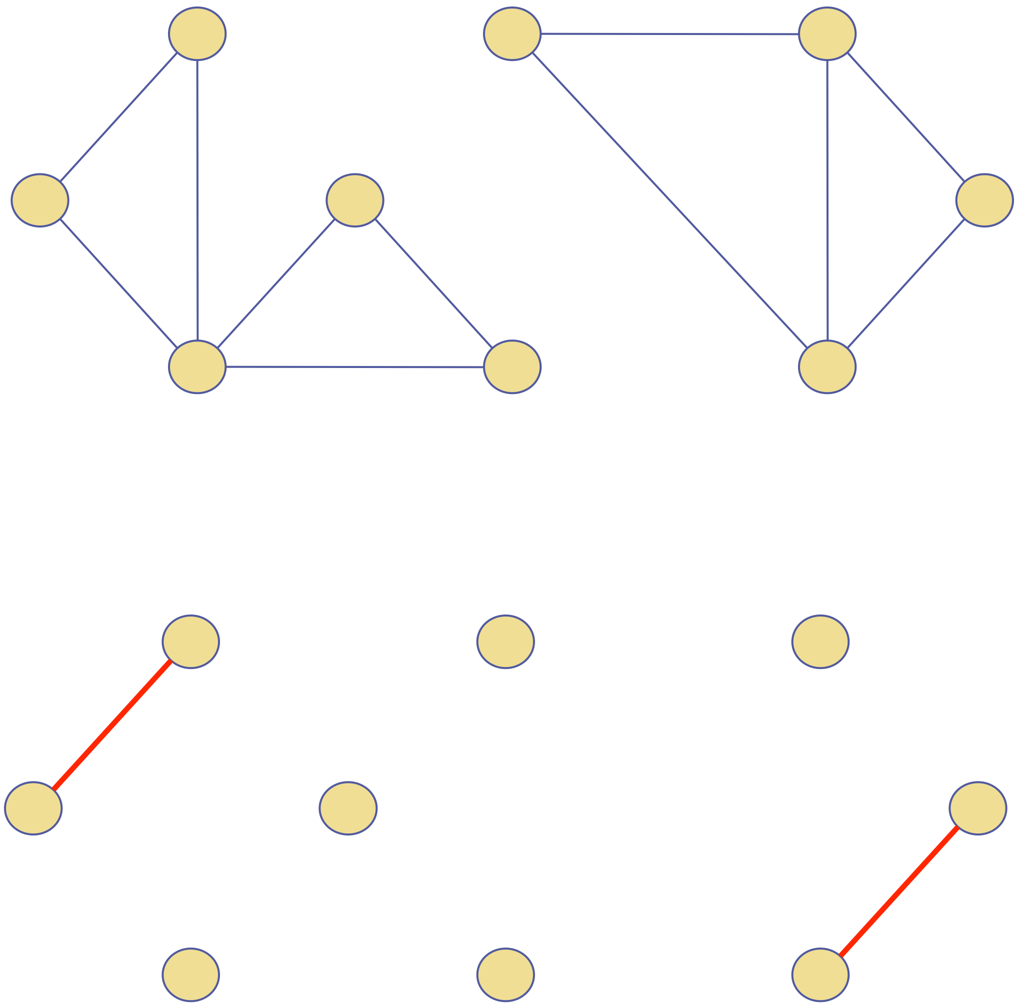
NB: exactly the same algorithm as before, but repeated for convenience!

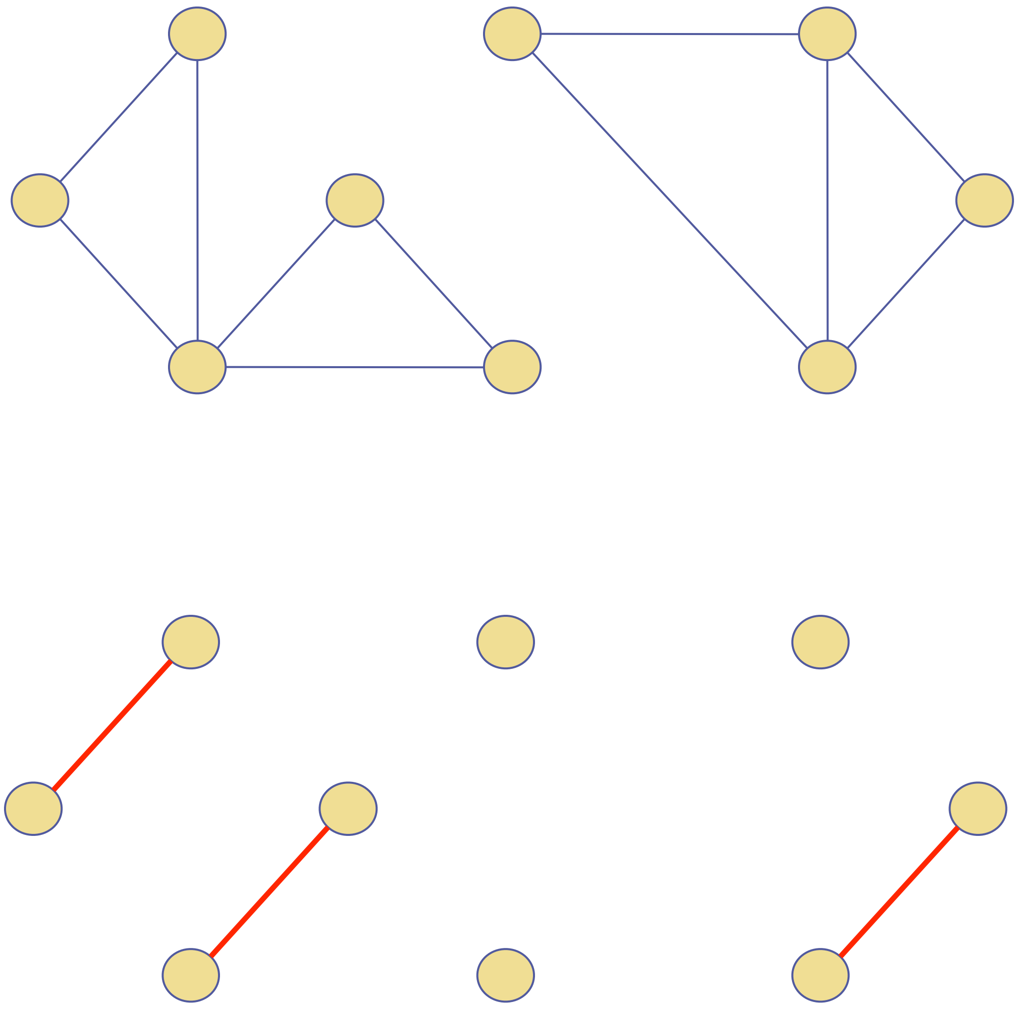
- We end up with c distinct sets S_u, where c is the number of connected components of G;
- E_T is a spanning forest for G, with $|V|-c$ edges.

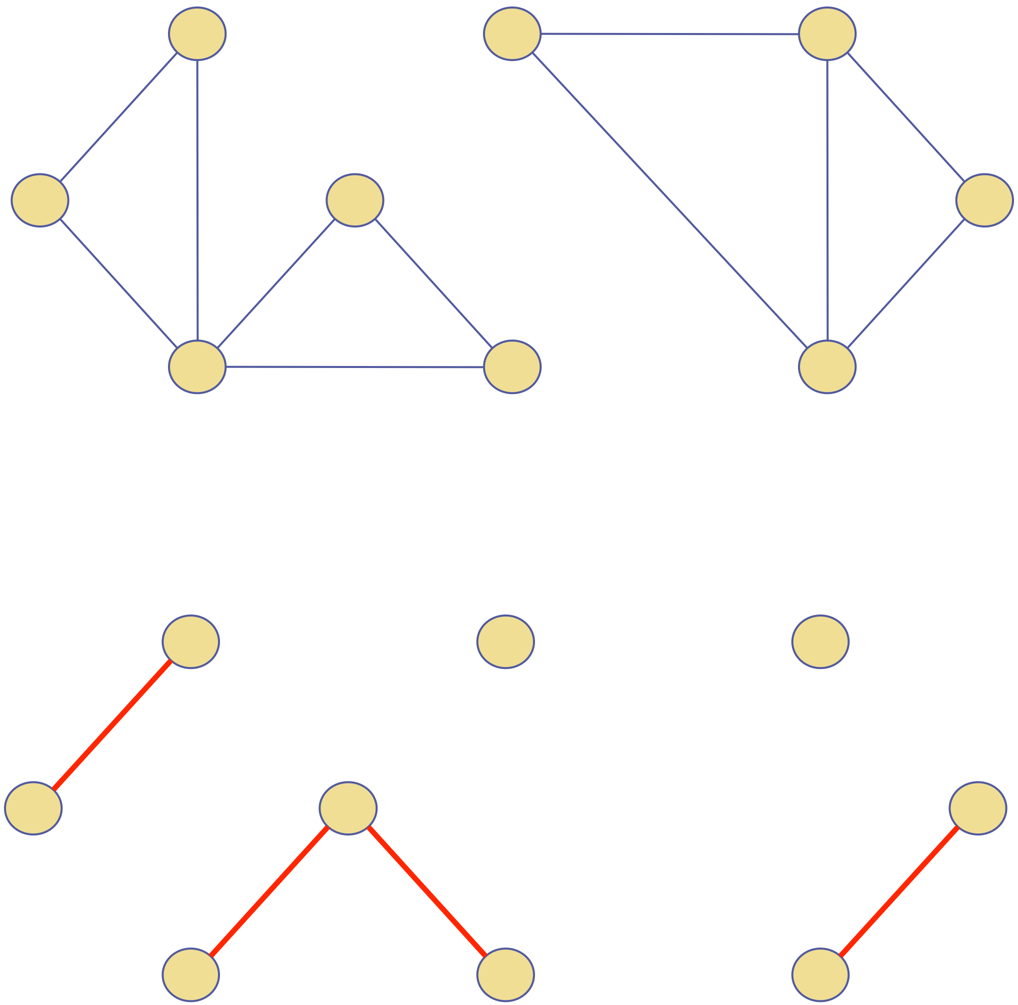


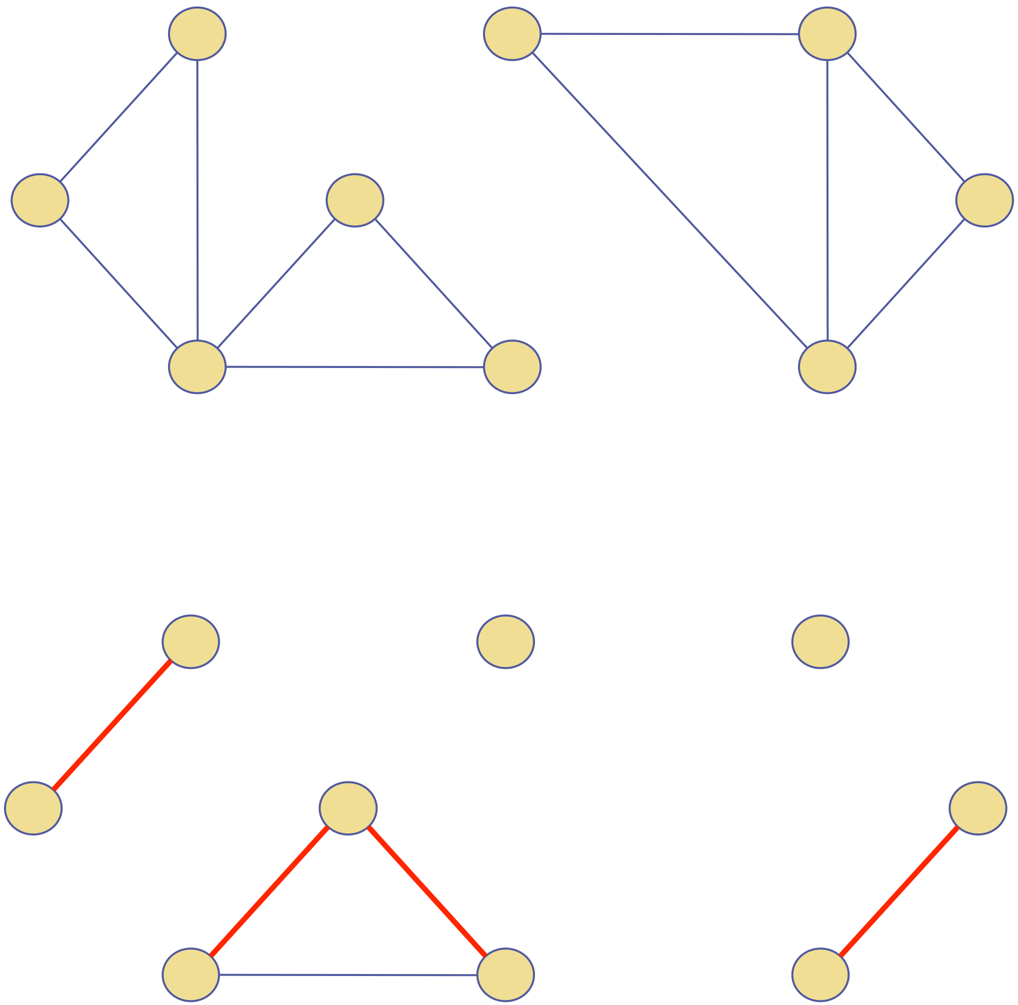


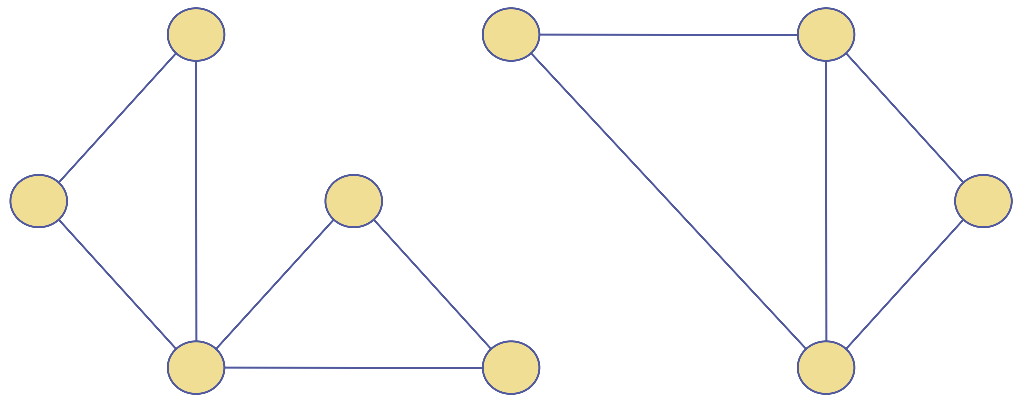


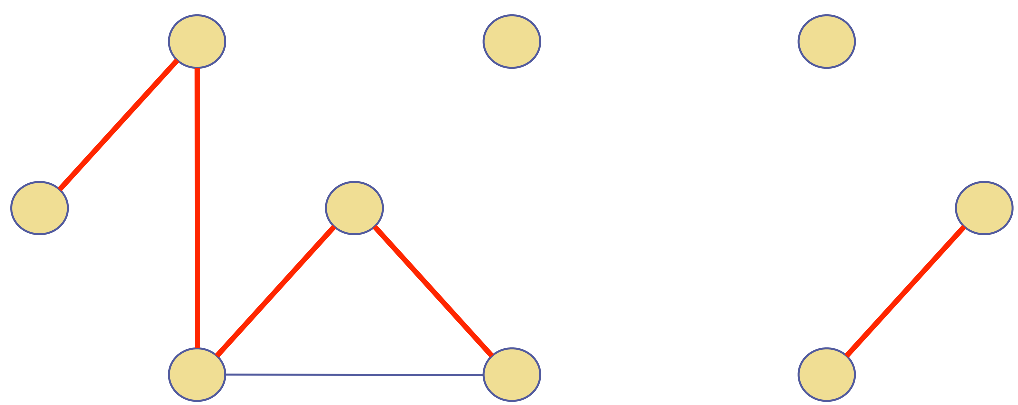


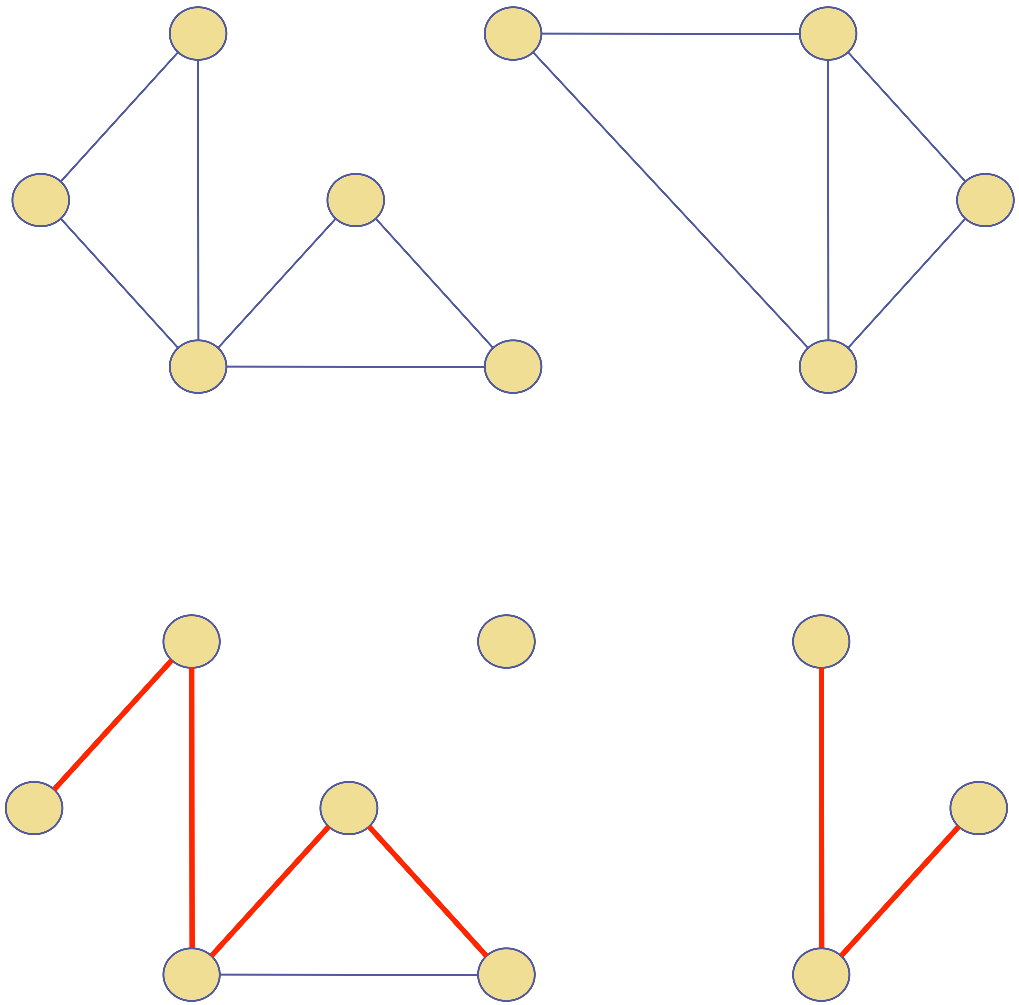


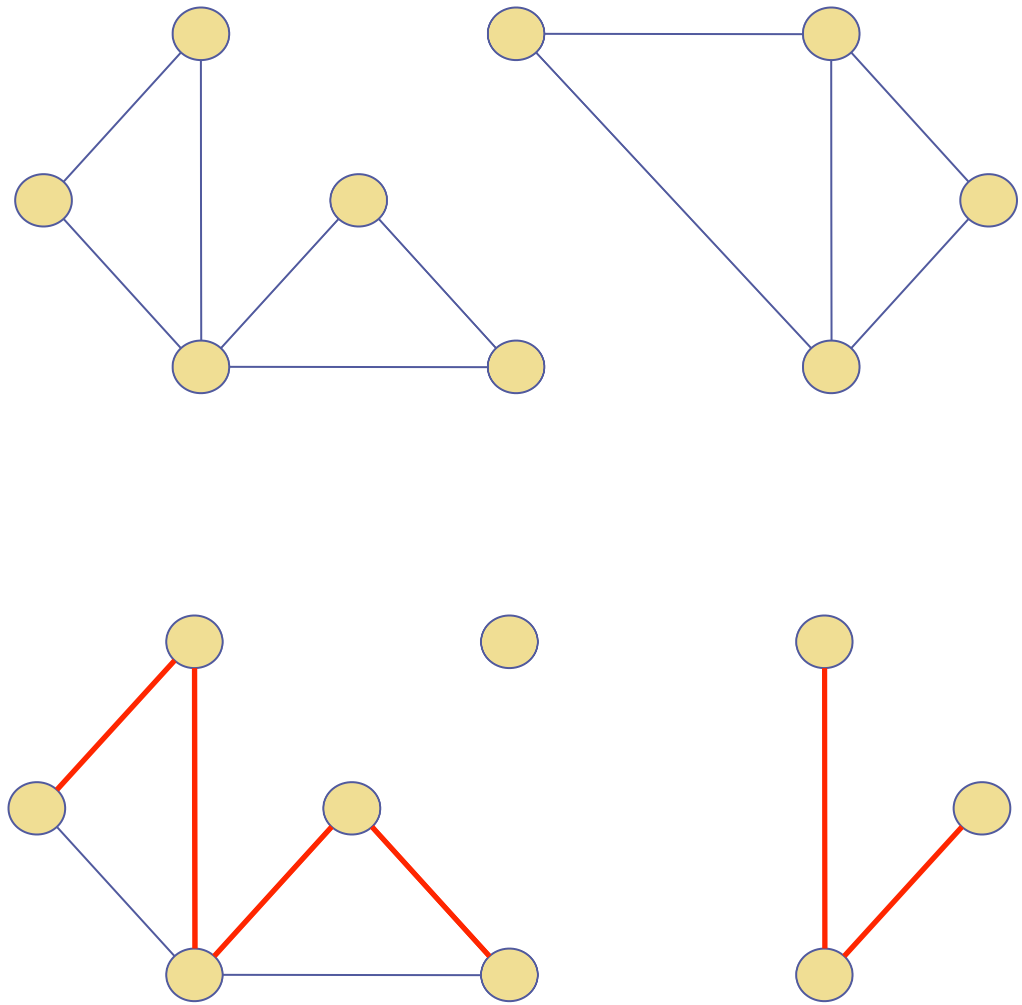


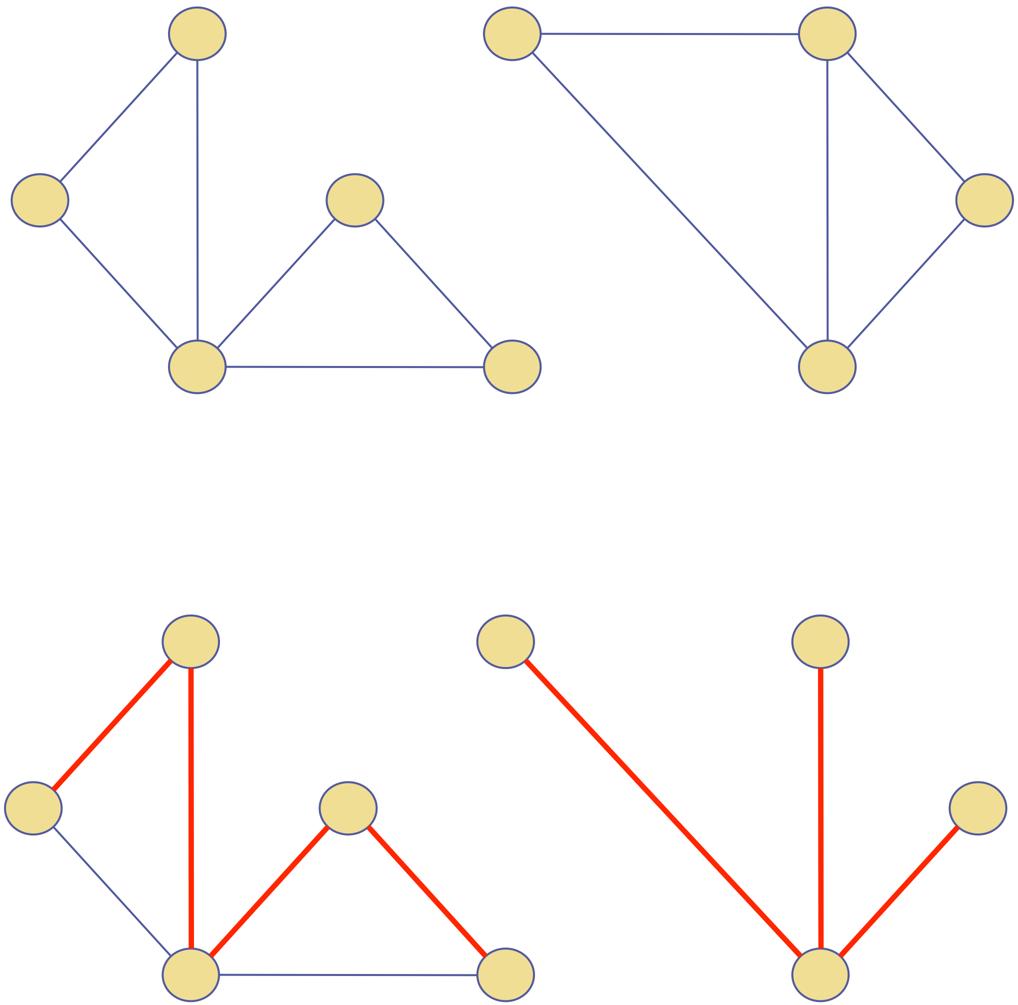


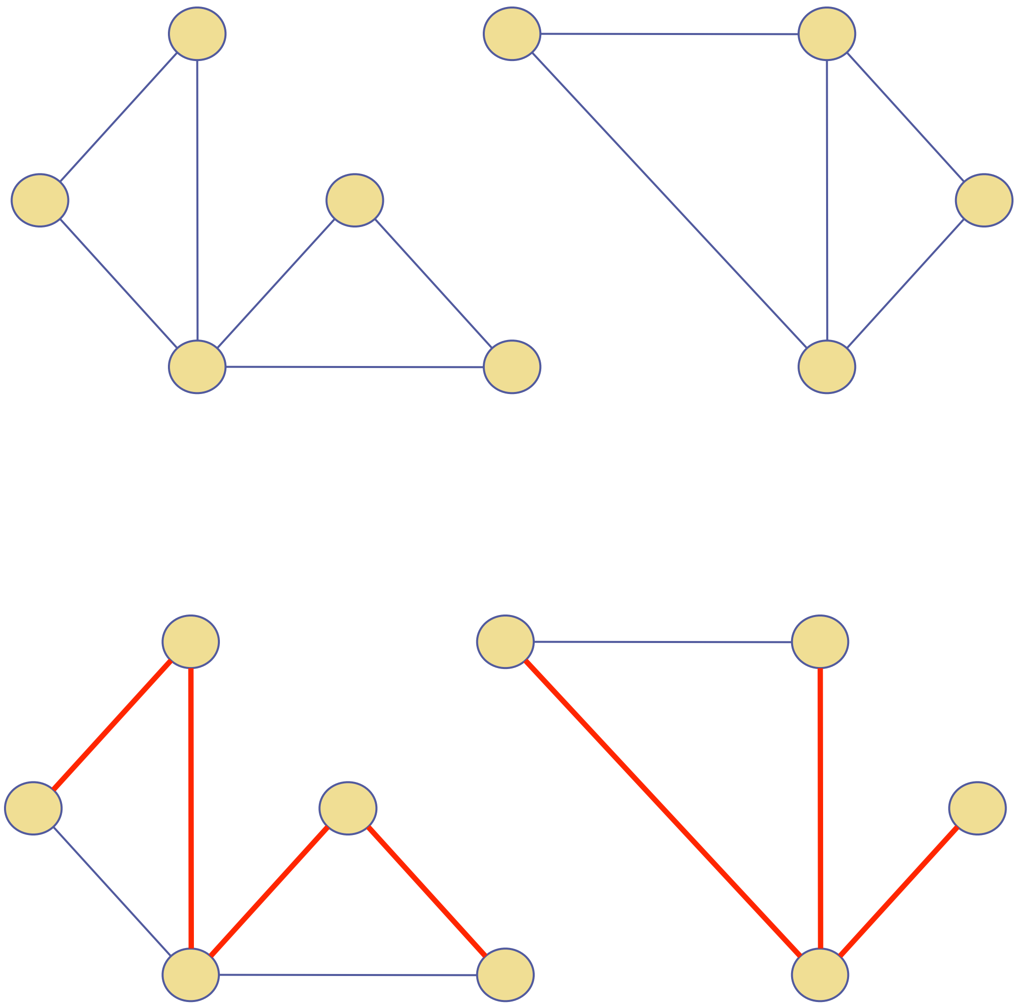


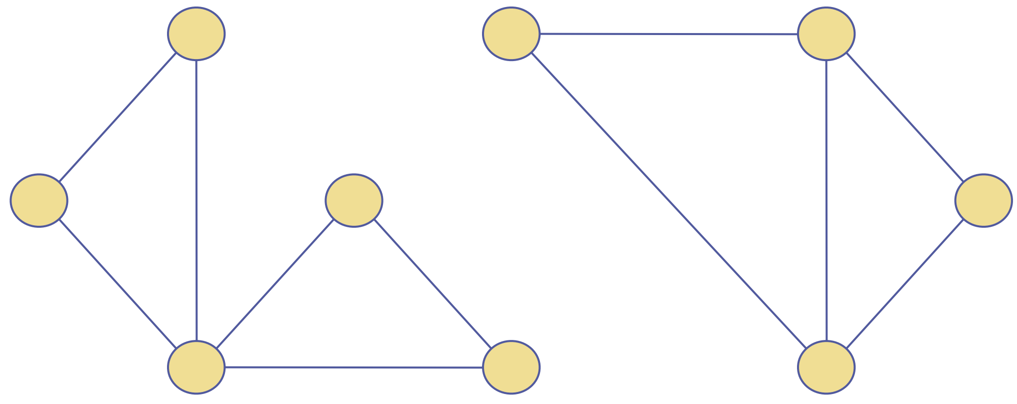


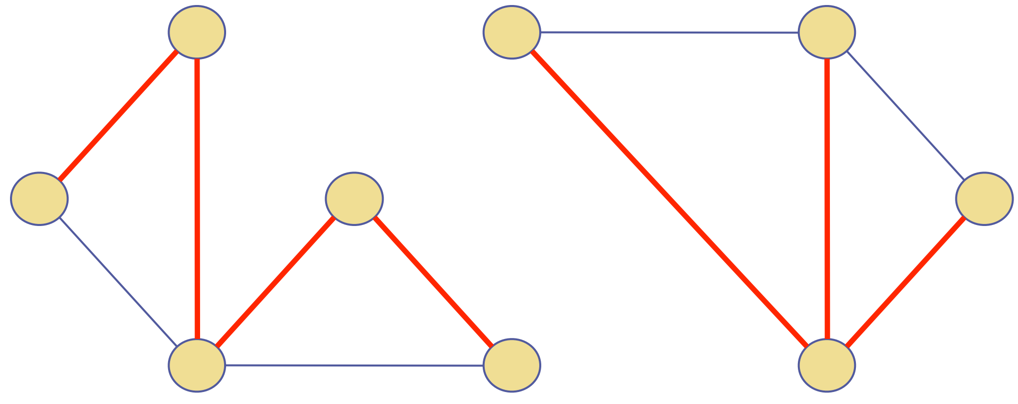






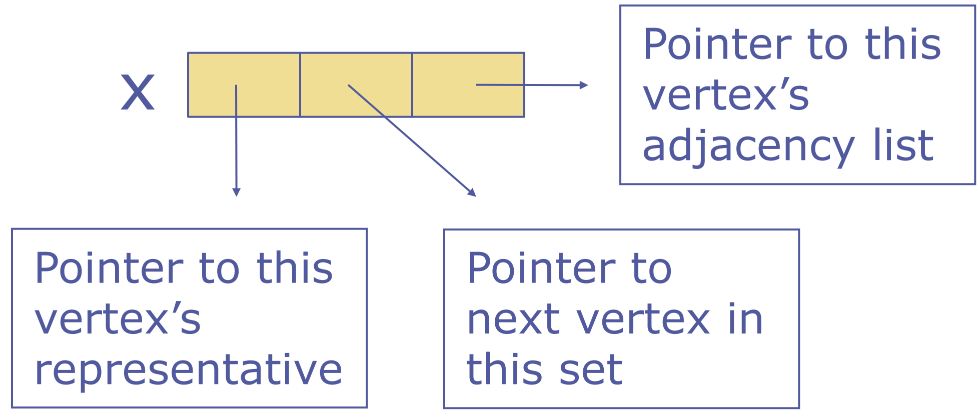




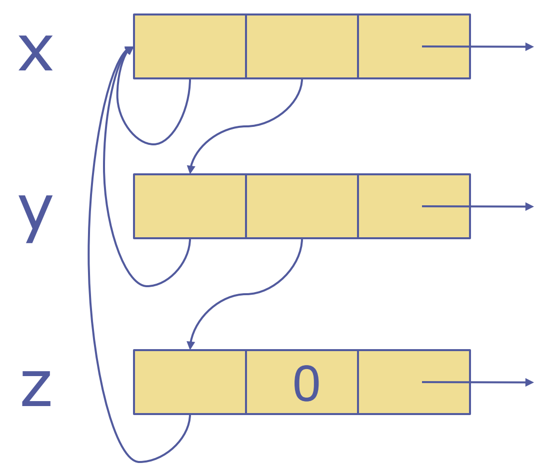


Union-find:

- The operations we need are:
  - Make a singleton set;
  - Test if two sets are equal;
  - Union two sets together.

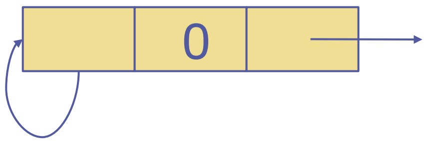


- There is a simple data structure that we can use to implement these operations.

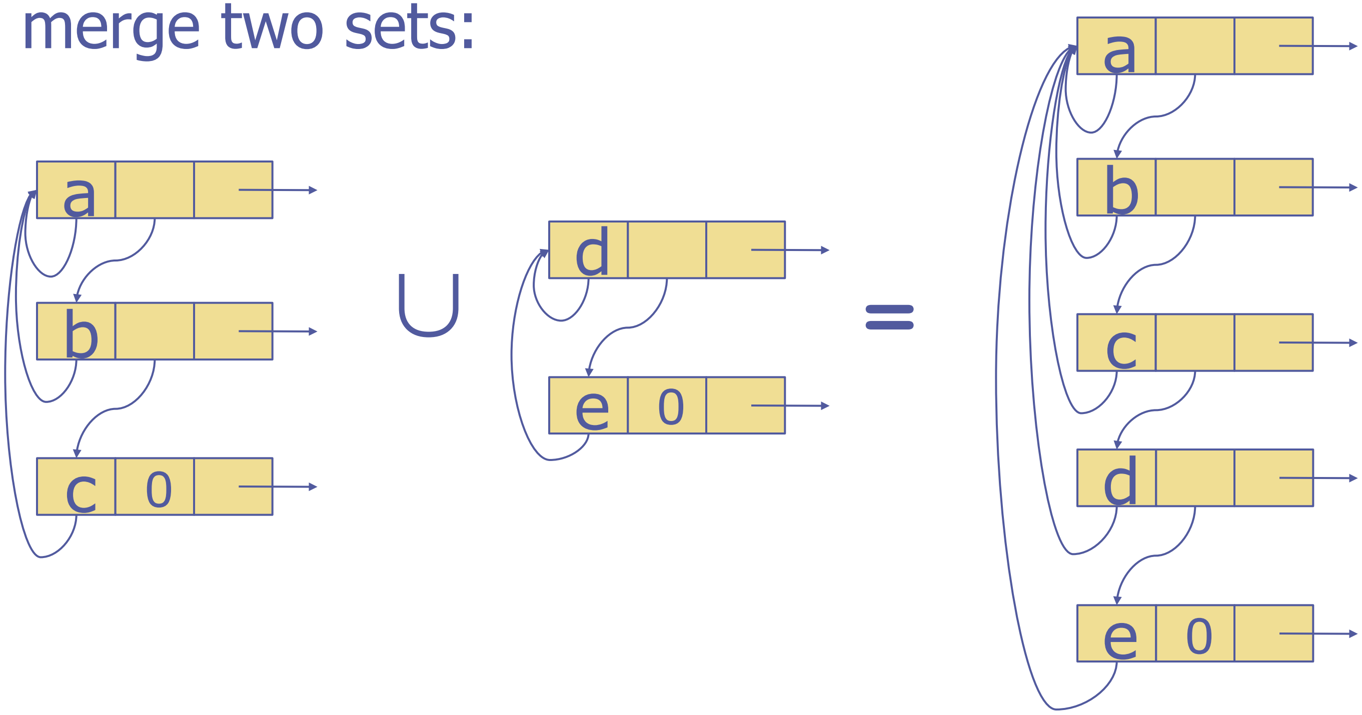


Implementation:

- To make a singleton set:

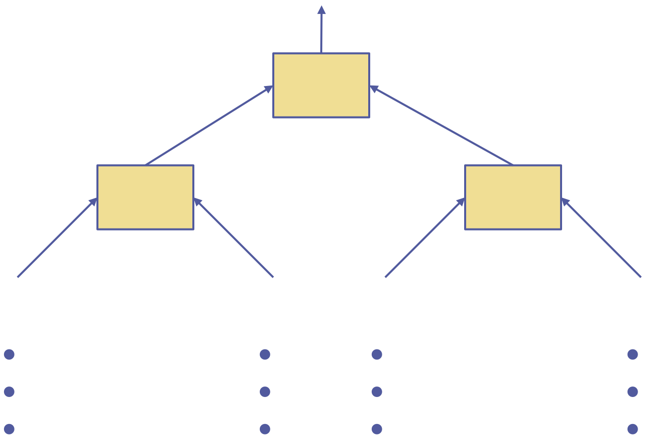


- To test if two sets are the same:
  - Test if the representatives are the same.
- To merge two sets:



Complexity:

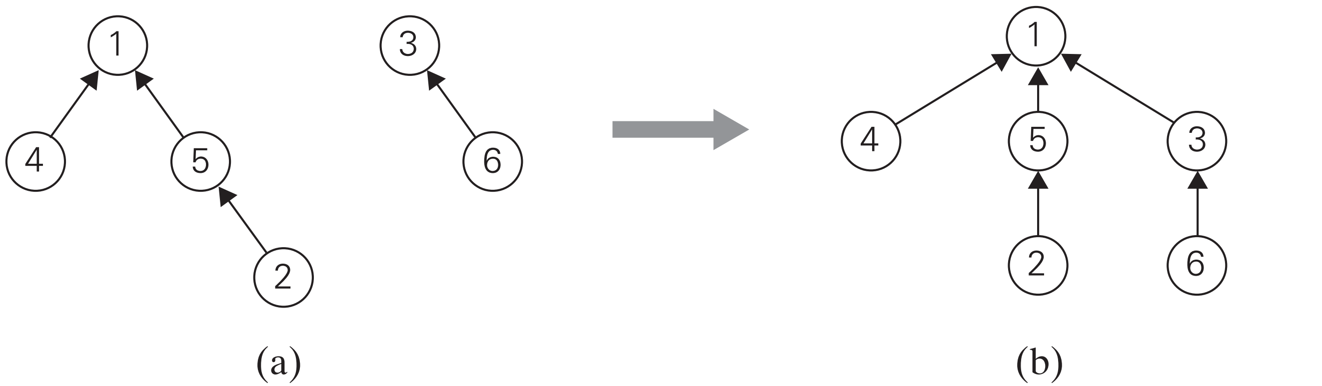
- A sequence of m operations can take $\Theta(m^2)$ time (amortized time per operation is $\Theta(m)$)
- More sophisticated variations are possible, with better complexity bounds.
- A tree based approach
  - Optimization heuristics:
    - Union by rank
    - Path compression



- See Levitin §9.2 or CLRS Chapter 21 for more details.

Quick Union:

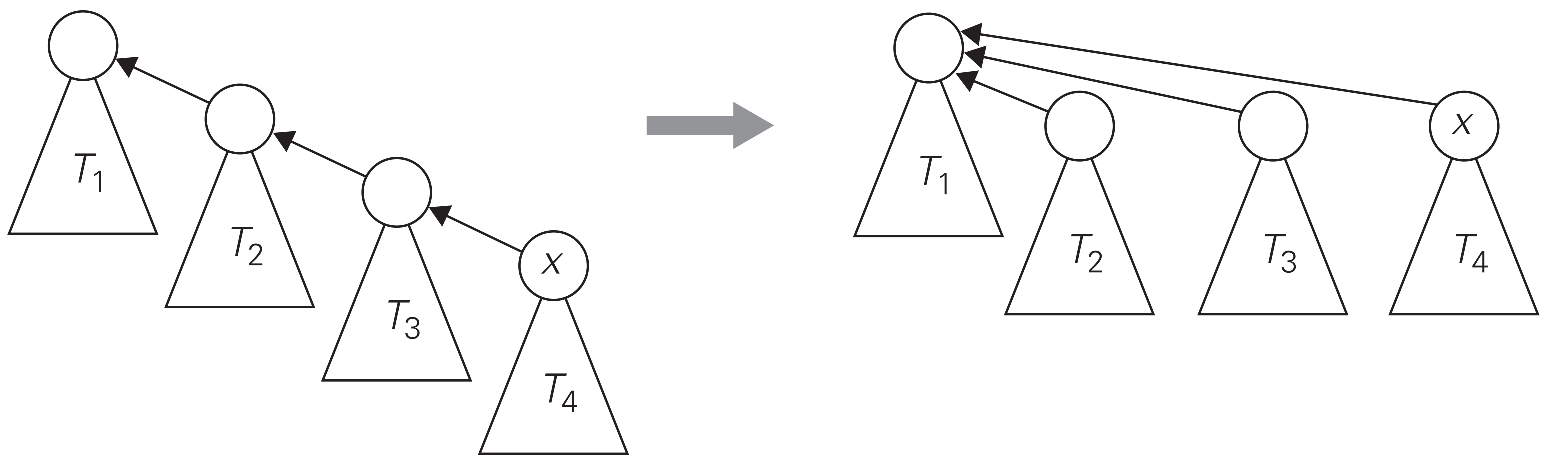
- Uses Tree-based representation of sets
  - root of tree used as representative of set



Tree representing {1, 4, 5, 2} and {3, 6}

After union(5,6)

Path Compression



- Amortized cost can be reduced by updating pointers to point directly to the root when they are queried.
- See Levitin §9.2 or CLRS Chapter 21 for more details.

Back to ...

Kruskal’s Algorithm

Growing a tree:

- Suppose that we have a connected graph G=(V, E) and pick an arbitrary vertex r ∈ V:

```
let W ← {r}, E_T ← empty set;

while (W≠V) do {
    find an edge (u,v) with u∈W and v∉W;

    W ← W ∪ {v};
    E_T ← E_T ∪ {(u,v)};
}
```

Growing a tree:

- Suppose that we have a connected graph G=(V, E) and pick an arbitrary vertex $r \in V$:

```
let W ← {r}, E_T ← empty set;

while (W≠V) do {
    find an edge (u,v) with u∈W and v∉W;

    W ← W ∪ {v};
    E_T ← E_T ∪ {(u,v)};
}
```

How many times will this loop execute?

Growing a tree:

- Suppose that we have a connected graph G=(V, E) and pick an arbitrary vertex $r \in V$:

Invariant: (W, E_T) is a connected, acyclic subgraph of G

```
let W ← {r}, ET ← empty set;

while (W≠V) do {
    find an edge (u,v) with u∈W and v∉W;

    W ← W ∪ {v};
    ET ← ET ∪ {(u,v)};
}
```

How many times will this loop execute?

Growing a tree:

- Suppose that we have a connected graph G=(V, E) and pick an arbitrary vertex $r \in V$:

Invariant: (W,E_T) is a connected, acyclic subgraph of G

```
let W ← {r}, ET ← empty set;

while (W≠V) do {
    find an edge (u,v) with u∈W and v∉W;

    W ← W ∪ {v};
    ET ← ET ∪ {(u,v)};
}
```

There must always be such an edge, otherwise G would not be connected.

How many times will this loop execute?

Growing a tree:

- Suppose that we have a connected graph G=(V, E) and pick an arbitrary vertex $r \in V$:

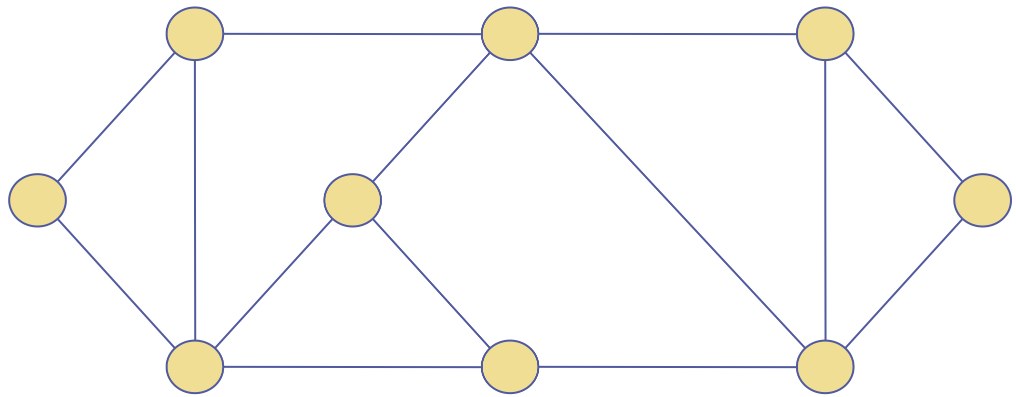
```
let W ← {r}, ET ← empty set;

while (W≠V) do {
    find an edge (u,v) with u∈W and v∉W;

    W ← W ∪ {v};
    ET ← ET ∪ {(u,v)};
}
```

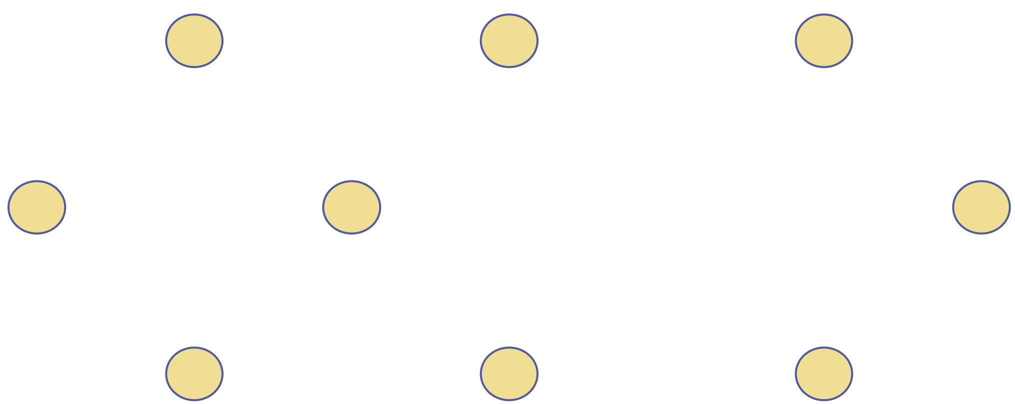
Invariant: (W,E_T) is a connected, acyclic subgraph of G

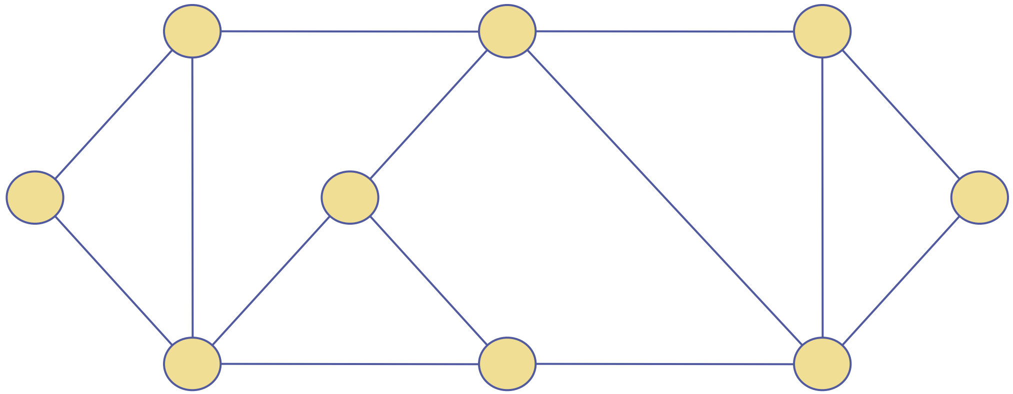
There must always be such an edge, otherwise G would not be connected.

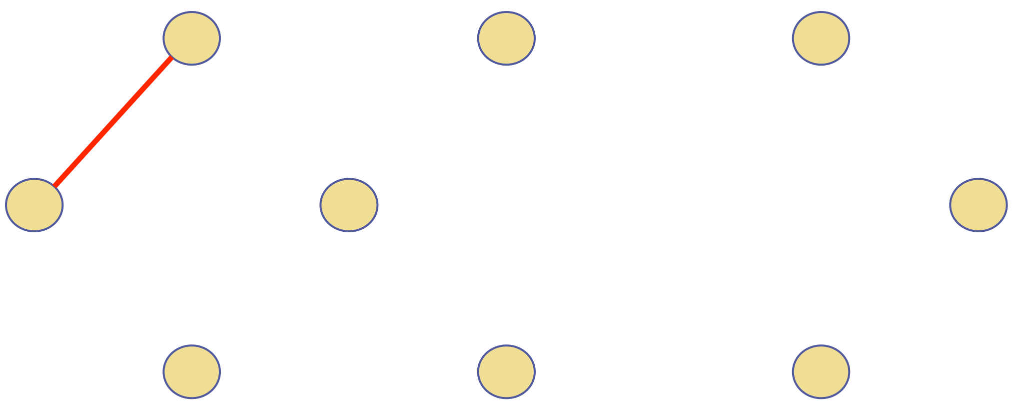
How many times will this loop execute?

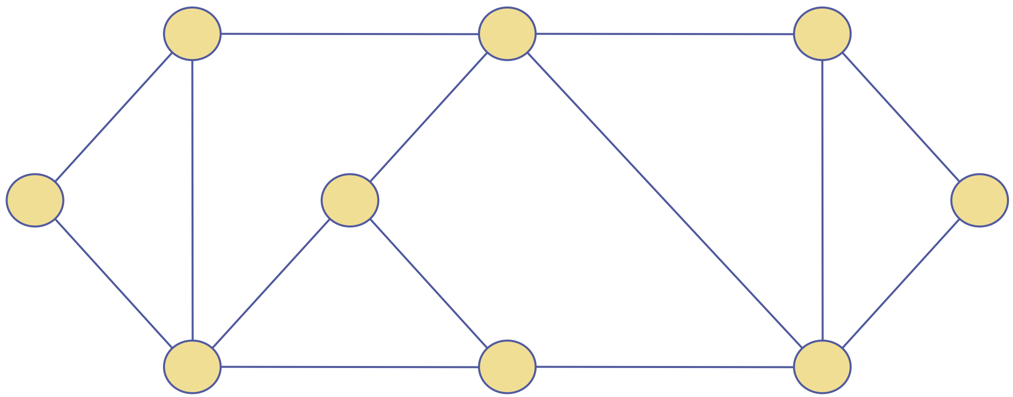
We add a total of |V|-1 edges to E_T

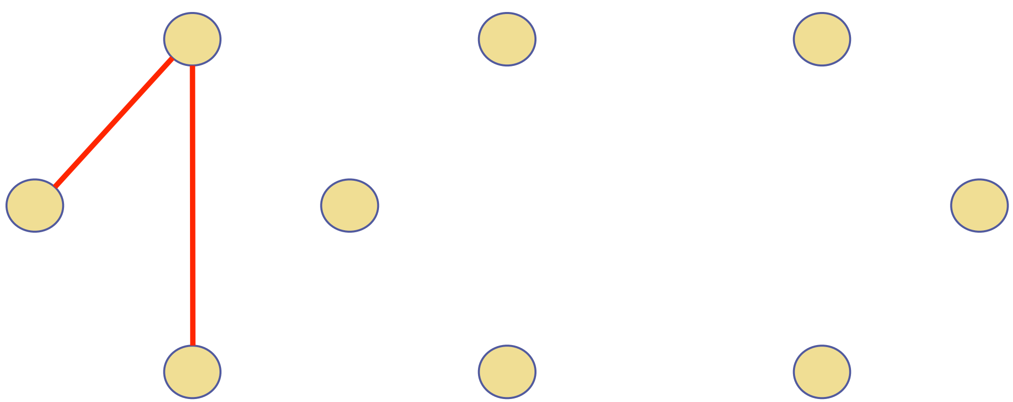


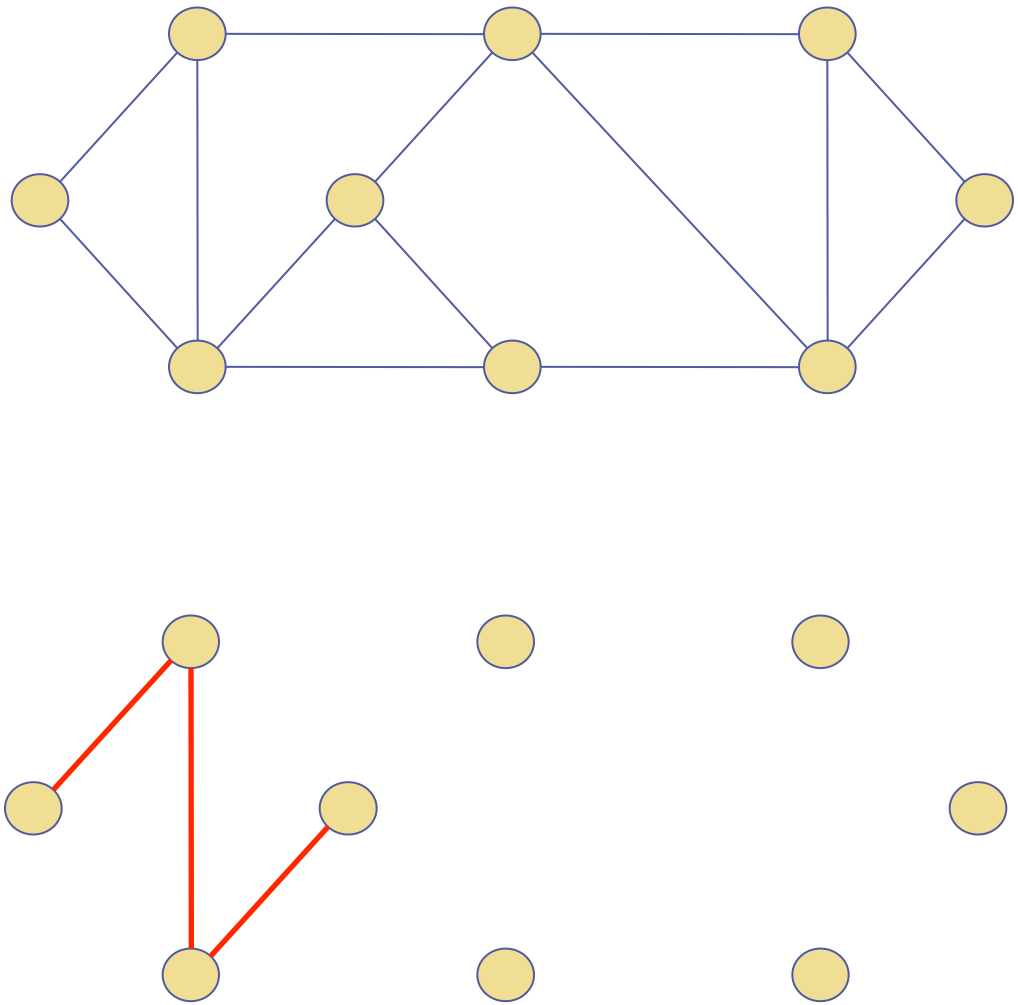


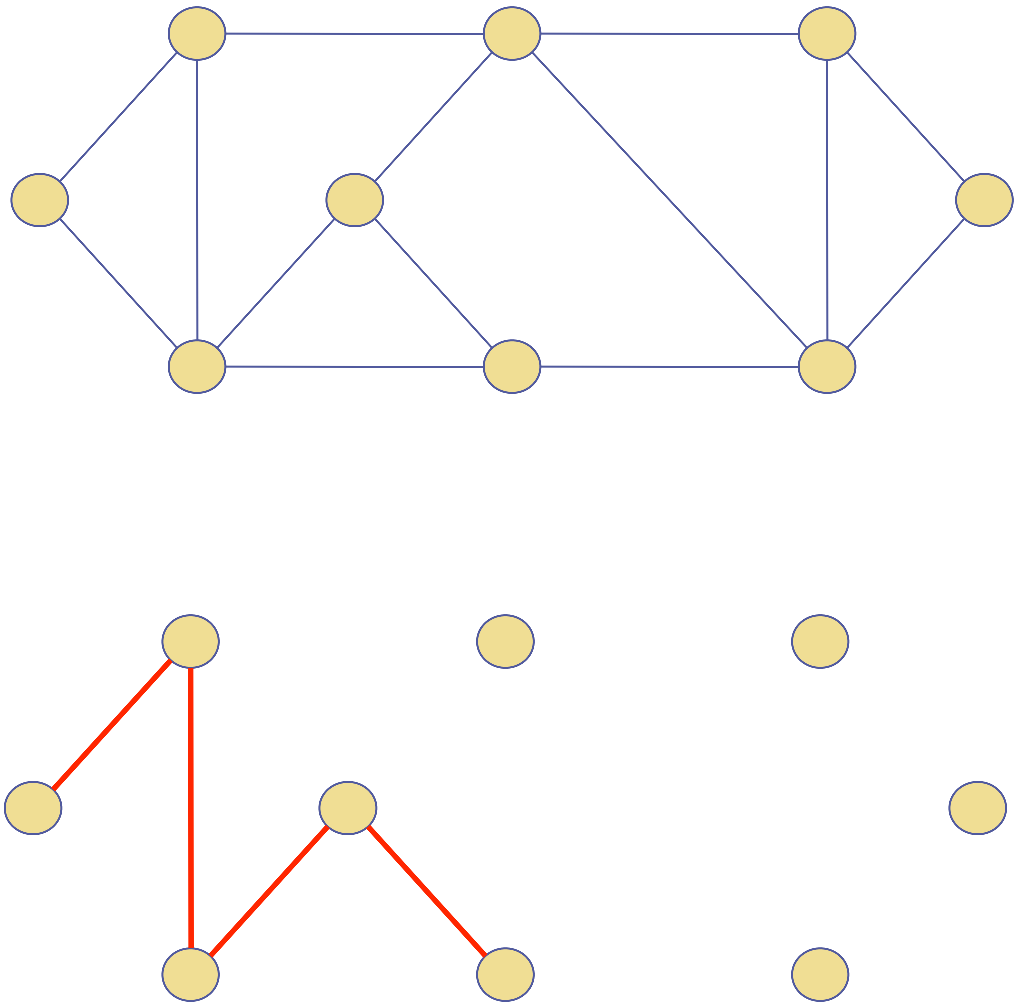


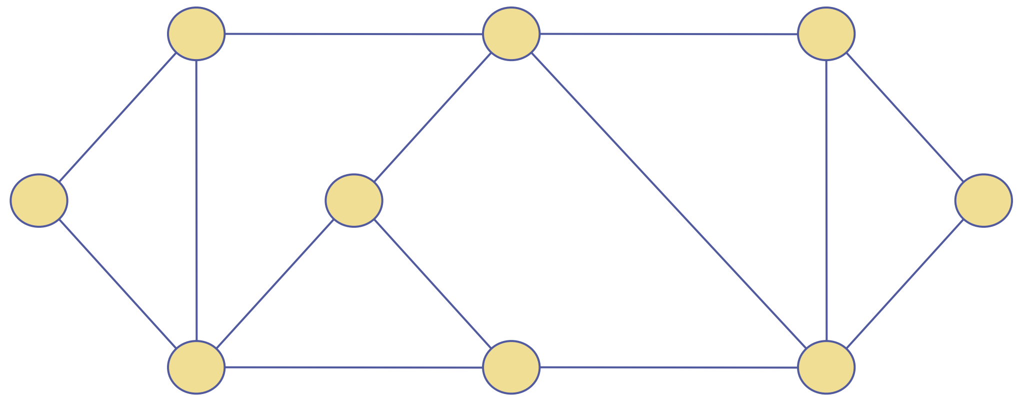


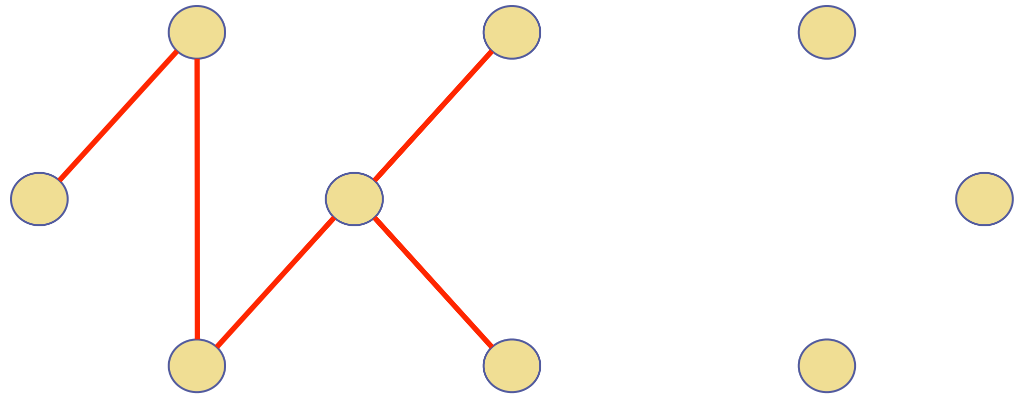


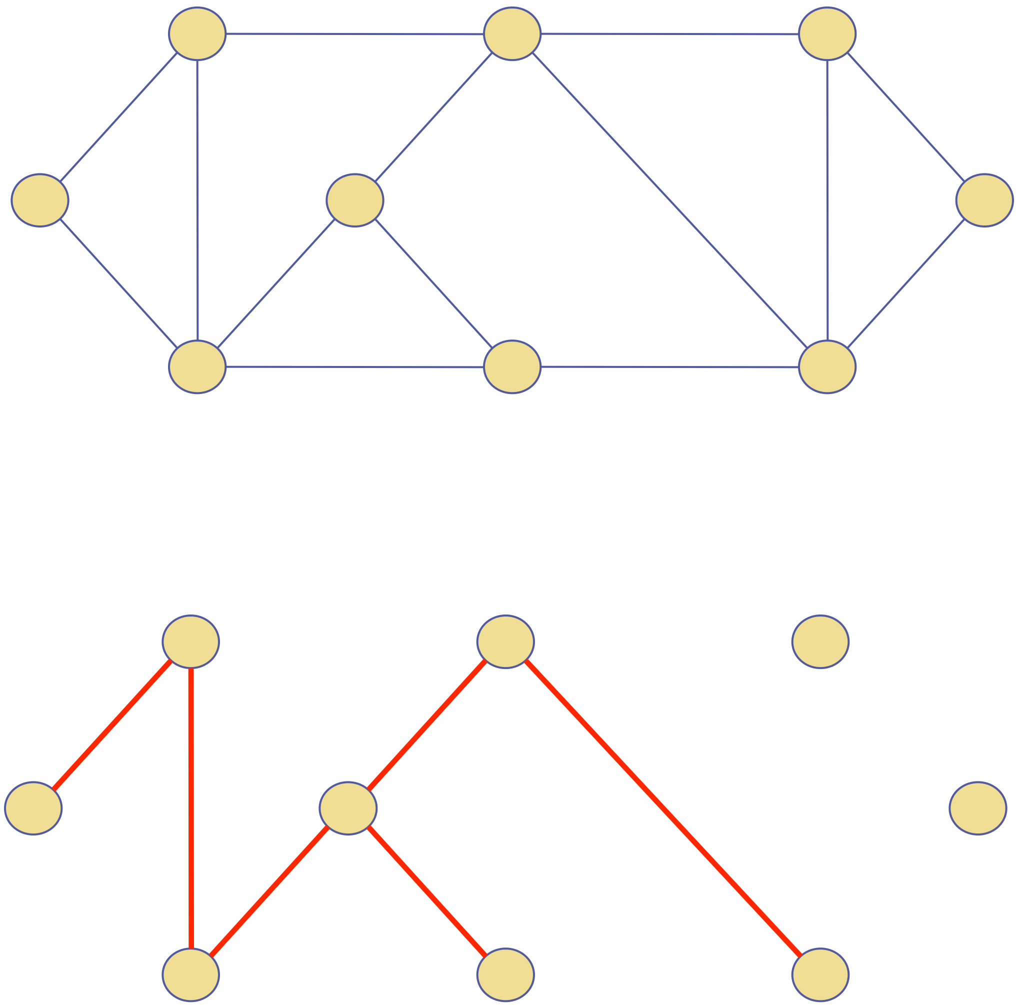


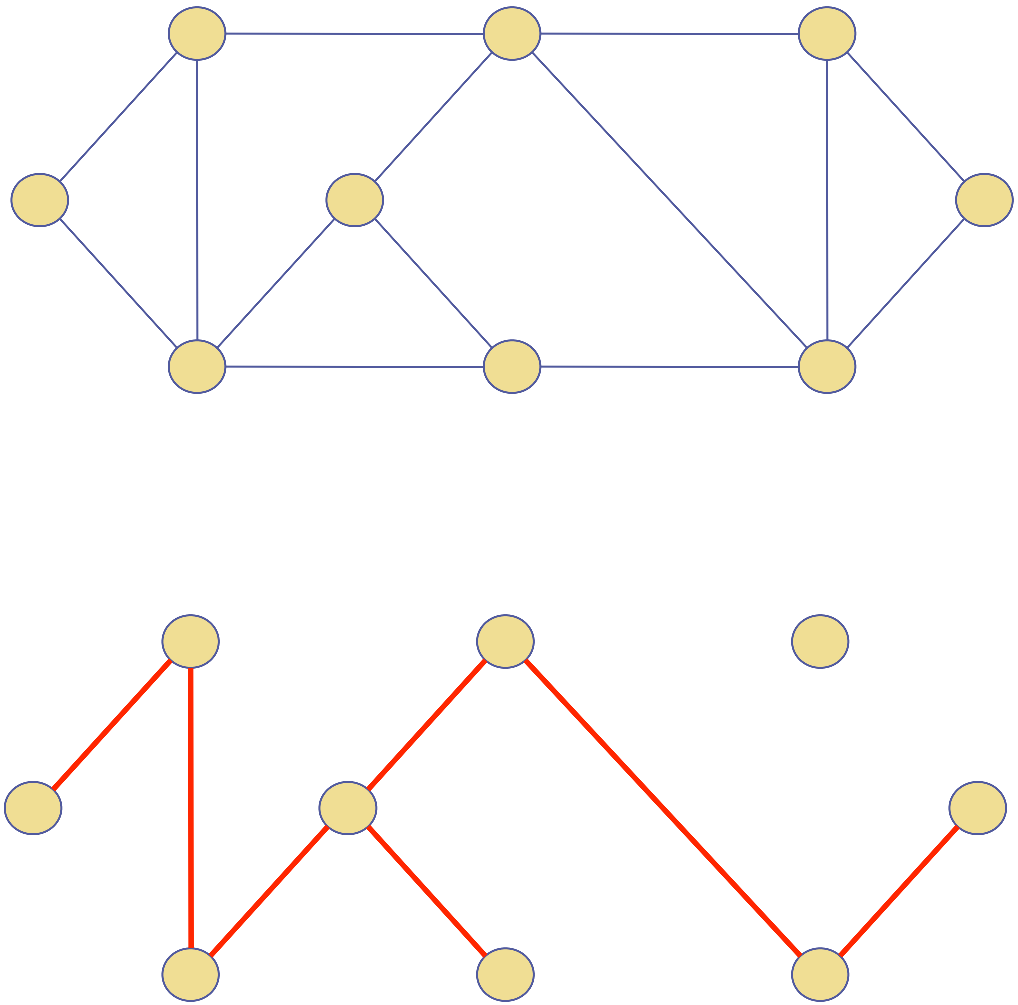


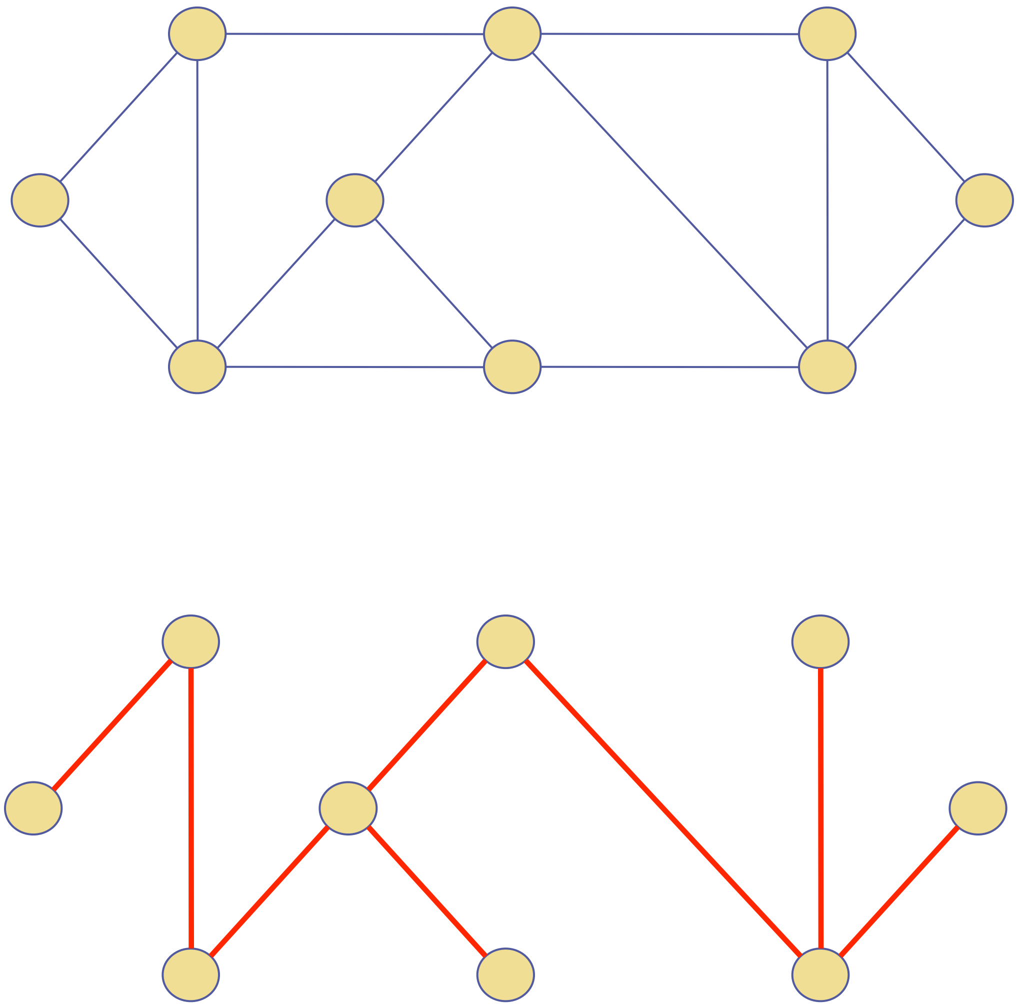












Minimum Spanning Trees

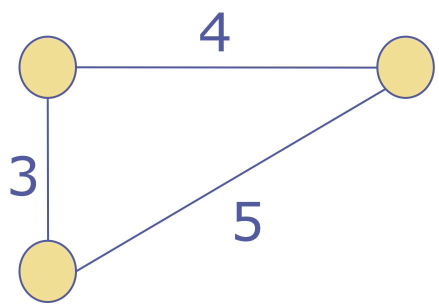
Back to bridge building …

- To link a group of n small islands together with bridges, we will need to build at least (n-1) bridges; any spanning tree will do for this.

- But now suppose that we want to minimize the total span of all the bridges as well … How should we proceed?

Minimum spanning trees:

- To take account of the distances between the islands, we need to use a labeled, or weighted graph.



- A *minimum spanning tree* (MST) is a spanning tree that minimizes the total of the weights on its edges.

- Not all spanning trees have this property.

The MST problem:

- Suppose that we have a connected, undirected graph G=(V,E), with a numerical weighting w(u,v) for each edge (u,v).

  Problem: Find an acyclic subset $T \subseteq E$ that connects all of the vertices in V, and minimizes:

  $$\Sigma \{w(u,v) \mid (u,v) \in T\}$$

  Solution: We will look for an algorithm of the form:

```
E_T ← empty set of edges
while (E_T is not a spanning tree)
    add an edge to E_T
```

- At each stage we will ensure that E_T is a subset of a MST.
- Obviously true when we start … the trick is to ensure that the invariant is preserved when we add an element ...

Greedy Choice

- Whenever we add an edge, let’s make the Greedy choice:
  - add the edge with the lowest weight that does not form a cycle
  - Edges that do form a cycle are not needed in the spanning tree
- Does making the Greedy choice ever add an edge that we don't need?

A key result:

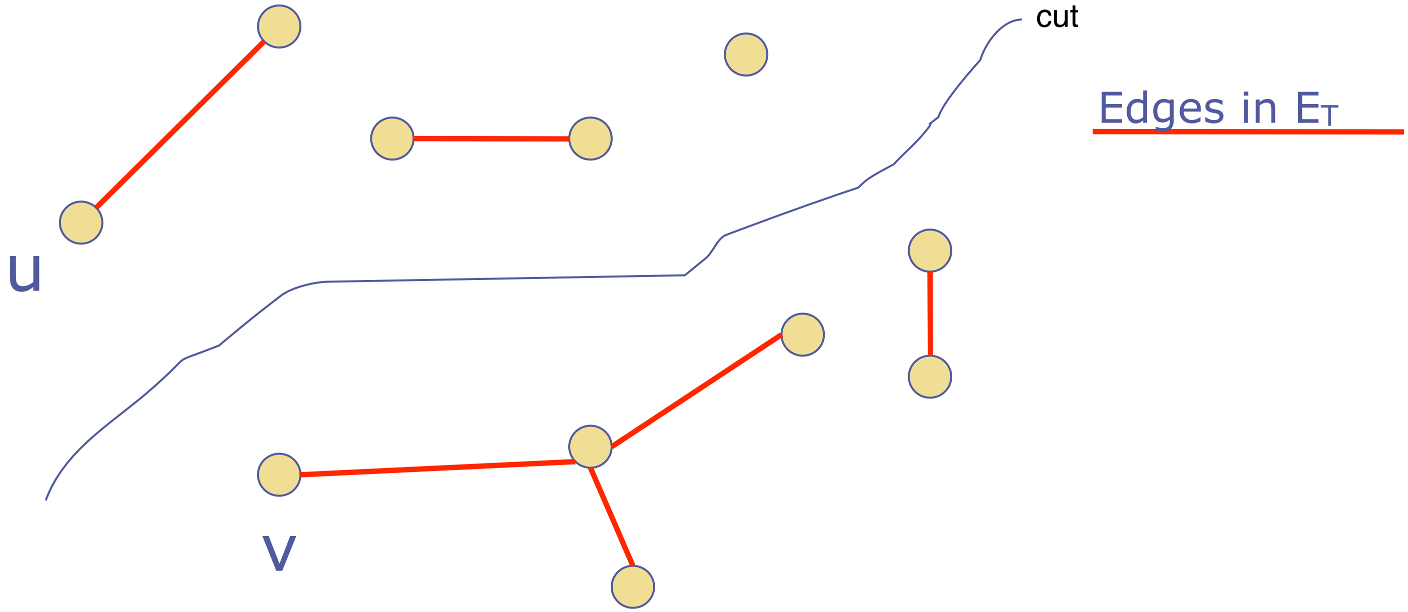
Suppose that we partition V into two sets (a "*cut*"), and that none of the edges in E_T crosses between the two sets (the cut "*respects*" E_T).

Suppose also that (u,v) is an edge that crosses between the two halves, and that no other edge that crosses has lower weight — (u,v) is a "*light edge*".

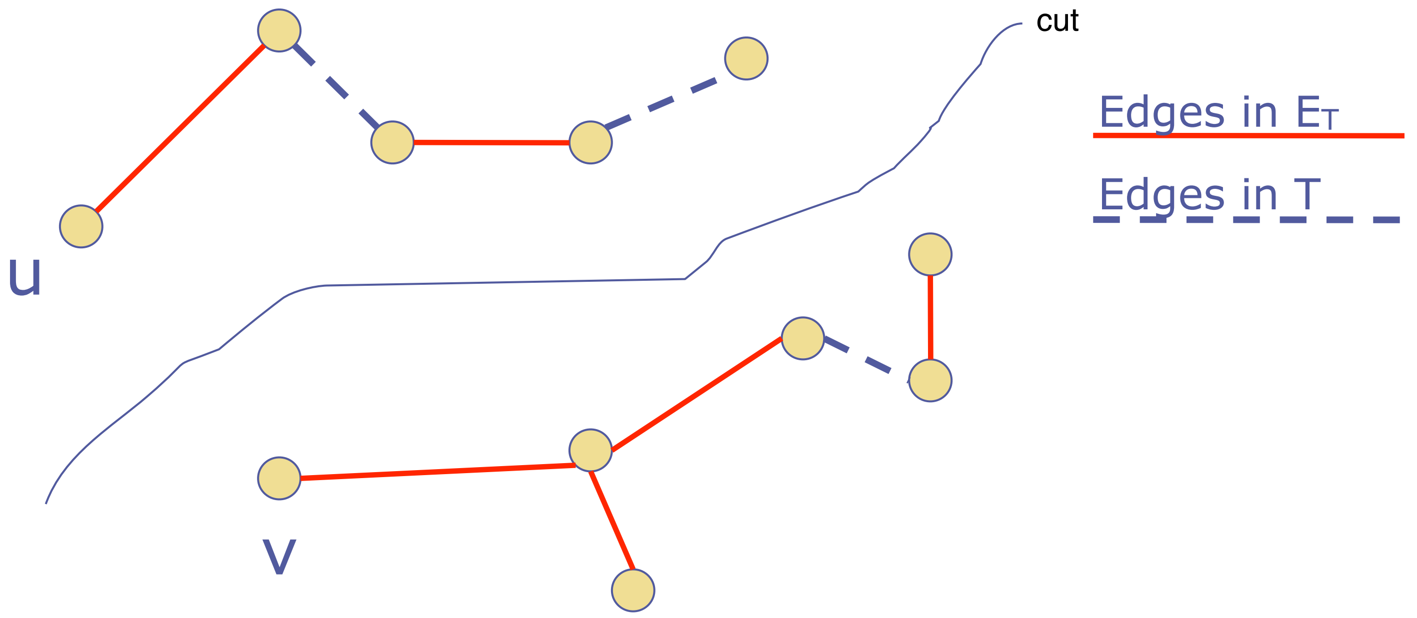
Claim: $E_T \cup \{(u,v)\}$ is a subset of a minimum spanning tree: (u,v) is "*safe*" for E_T.

Proof:

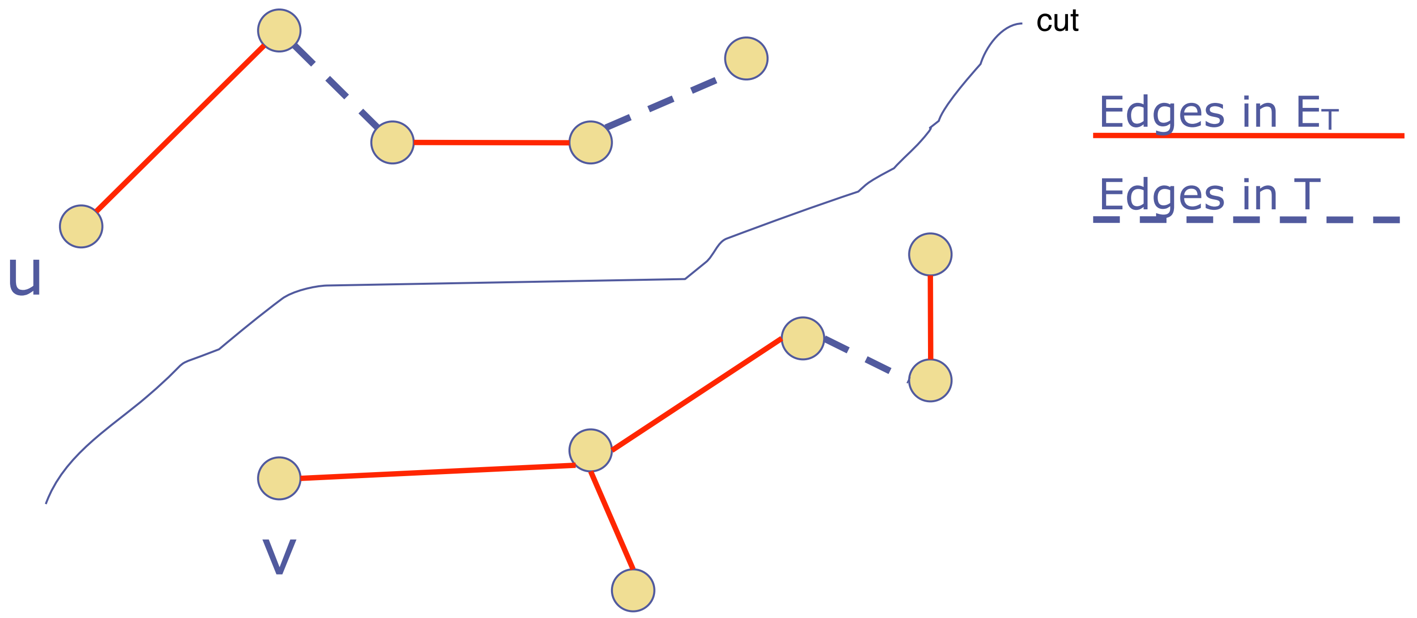
Proof:



Proof:

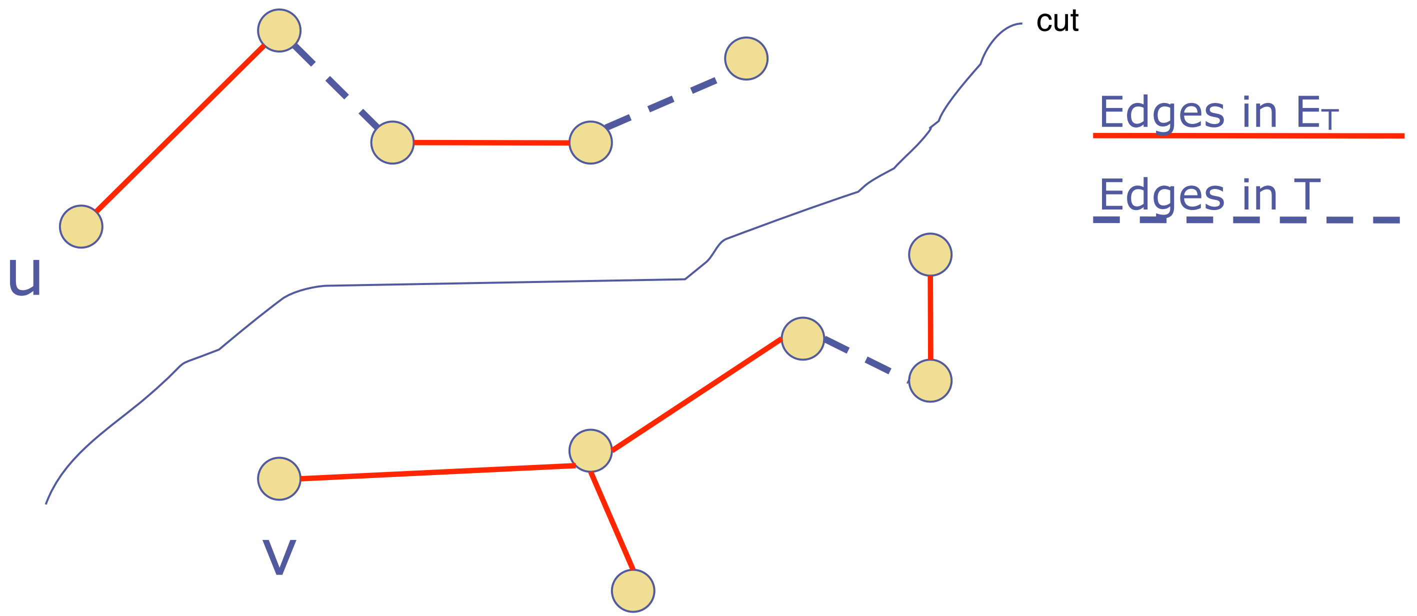


Proof:



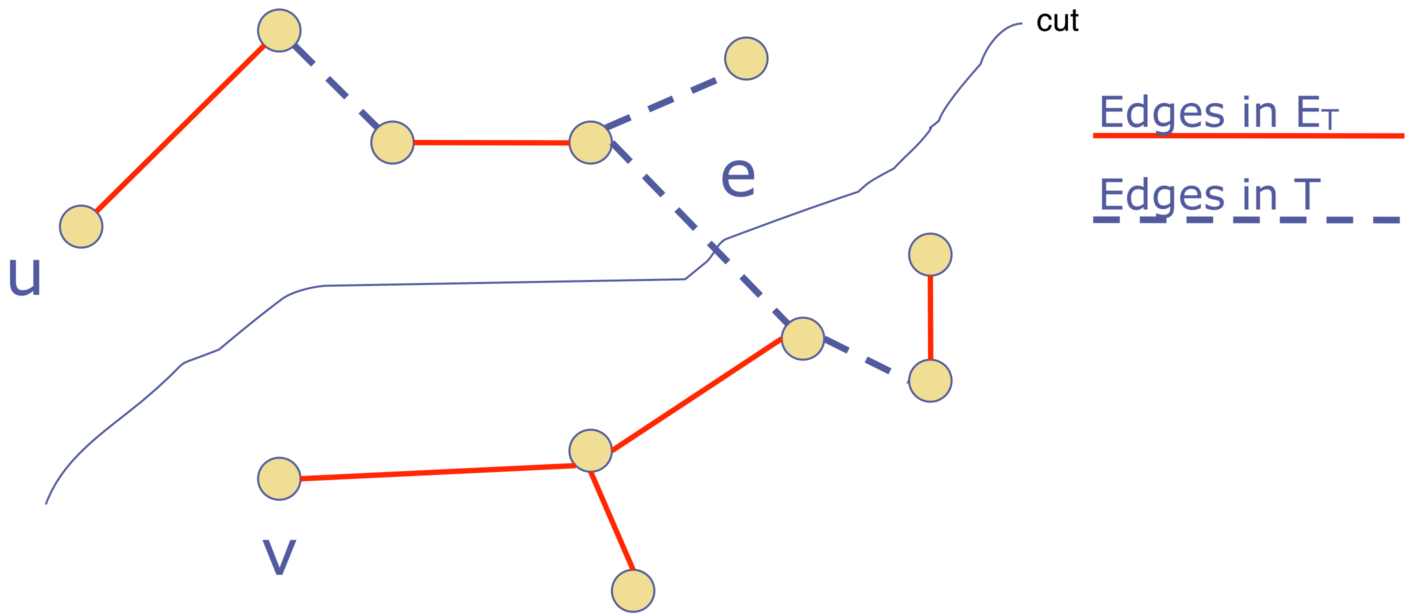
- E_T is a subset of some minimum spanning tree T.

Proof:



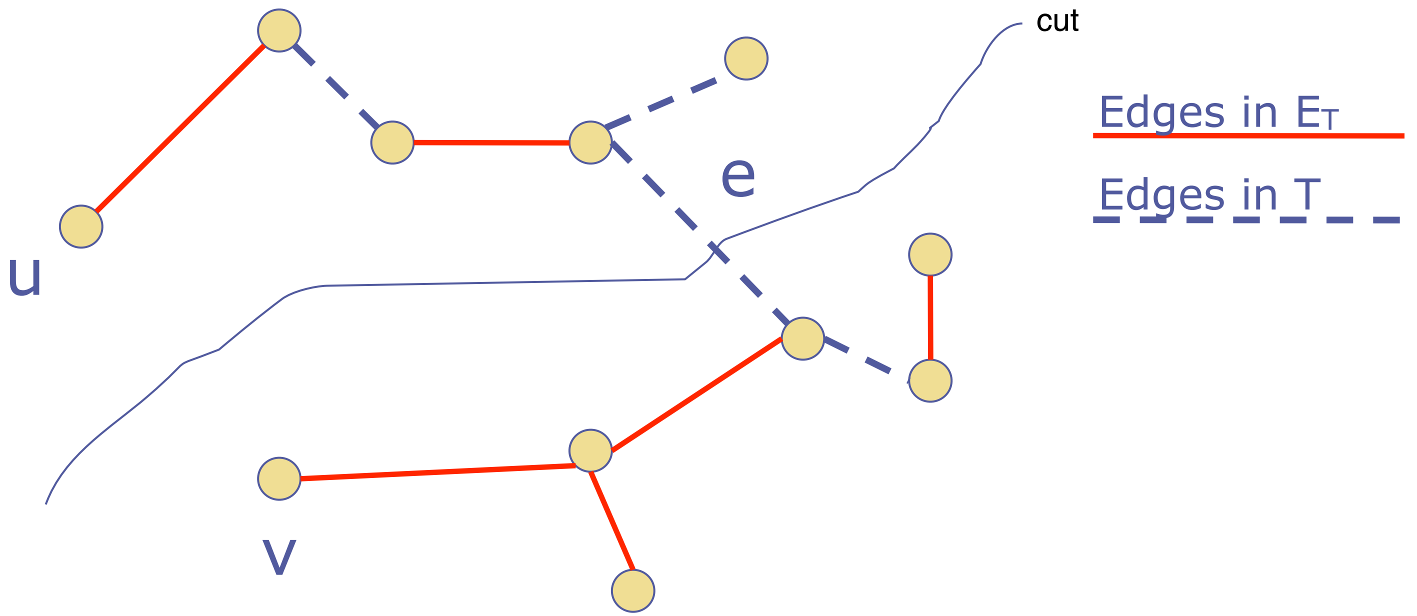
- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.

Proof:



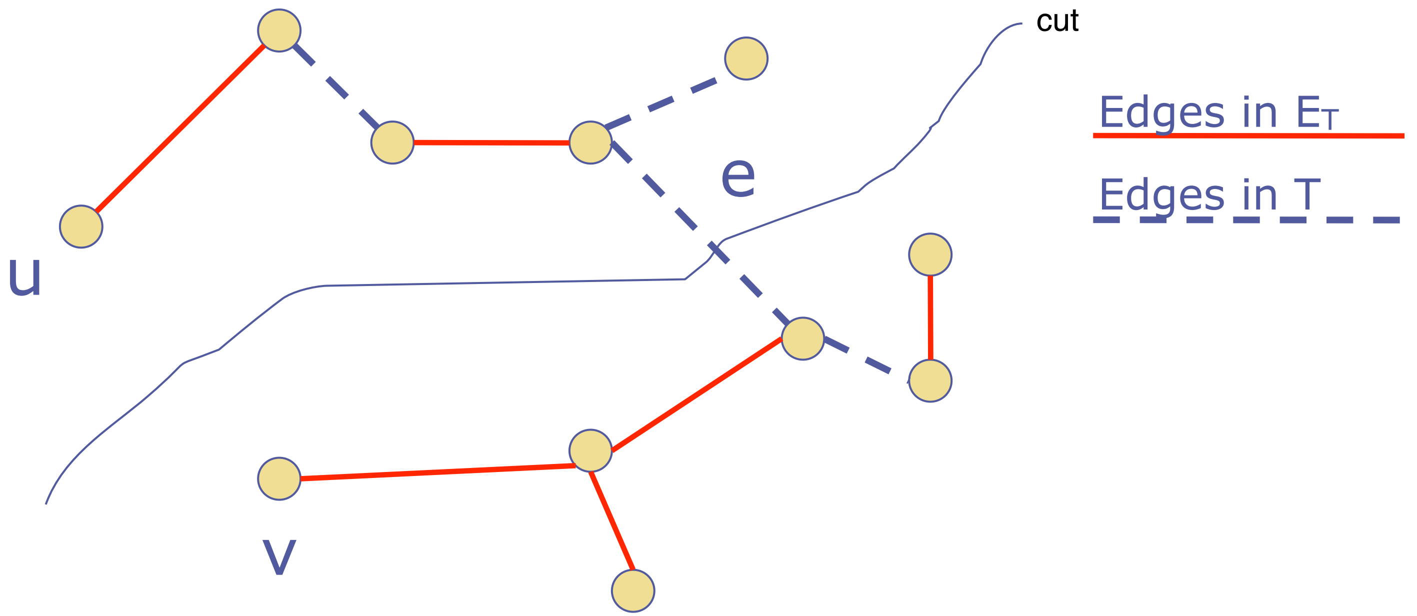
- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.

Proof:



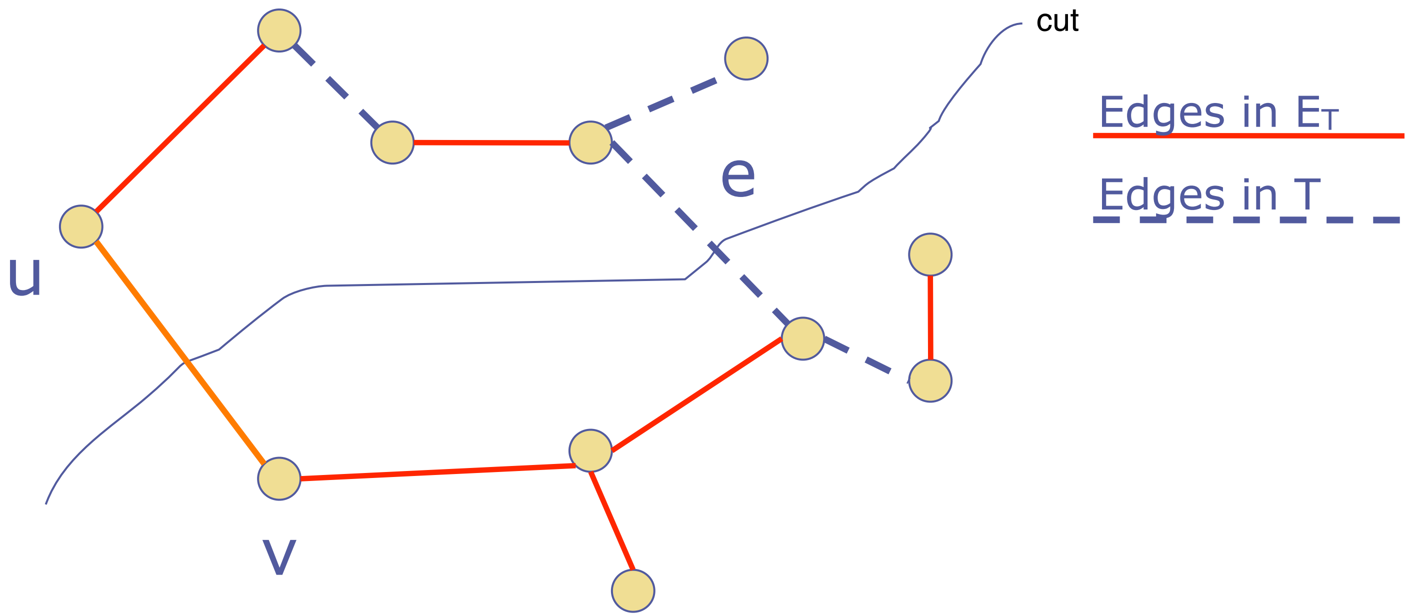
- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.
- By assumption weight of (u,v) ≤ the weight of e.

Proof:



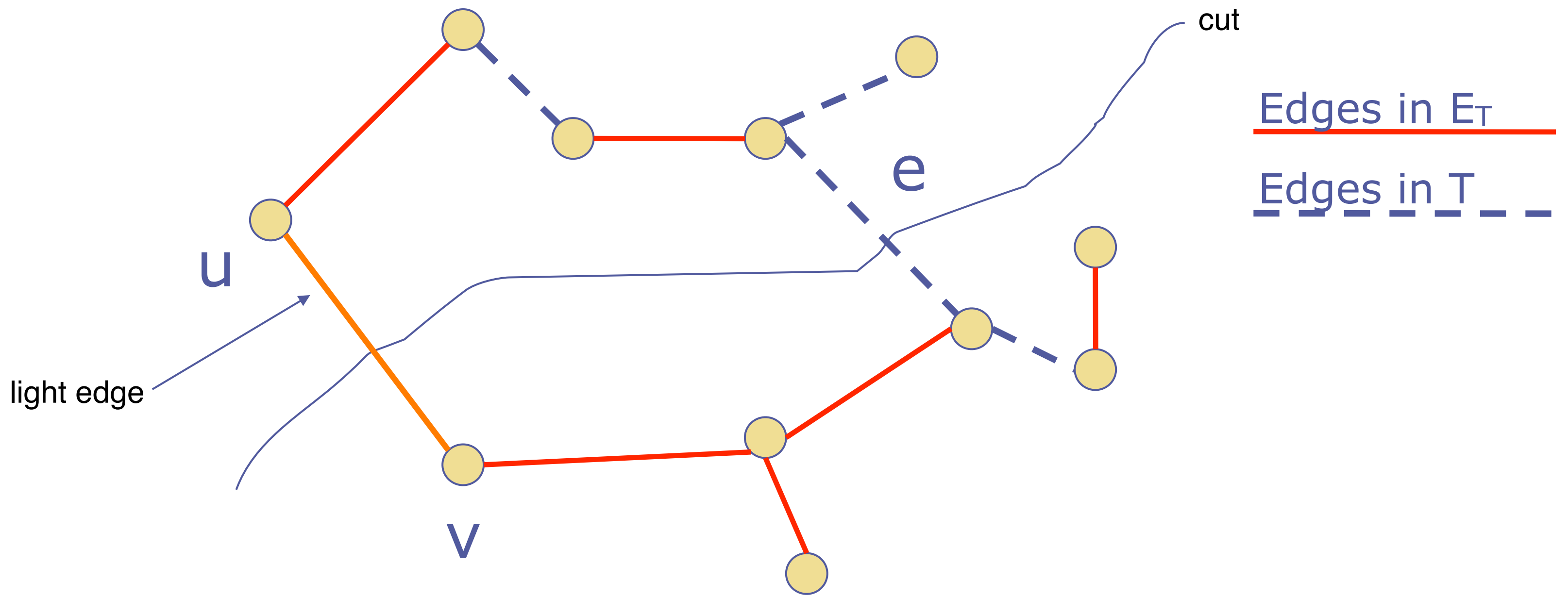
- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.
- By assumption weight of (u,v) ≤ the weight of e.
- So if we replace e with (u,v), we get a minimum spanning tree … which contains $E_T \cup \{(u,v)\}$.

Proof:



- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.
- By assumption weight of (u,v) ≤ the weight of e.
- So if we replace e with (u,v), we get a minimum spanning tree … which contains $E_T \cup \{(u,v)\}$.

Proof:



- E_T is a subset of some minimum spanning tree T.
- Because u and v are on opposite sides, there is an edge e in T that crosses the cut.
- By assumption weight of (u,v) ≤ the weight of e.
- So if we replace e with (u,v), we get a minimum spanning tree … which contains $E_T \cup \{(u,v)\}$.

Corollary:

- Suppose that:
  - C is a connected component in the forest (V, E_T);
  - (u,v) is a *light edge* connecting C to some other component in G.
- Then (u,v) is safe for E_T.

- Follows directly by using a cut to separate the vertices in C from the vertices outside.
- Requiring C to be a connected component of (V, E_T) ensures that no edge in E_T crosses the cut.

Kruskal's algorithm:

- Given a connected graph G=(V, E):

```
E_T ← empty set of edges
for each v in V
   make a singleton set {v}

sort the edges of E by nondecreasing weight

for each edge (u,v) in E
   if S_u ≠ S_v, then
      replace S_u and S_v with S_u ∪ S_v
      add (u,v) to E_T
```

- Complexity is O(|E| log |E|).
  - (With our simple union-find, more like $O(|E|^2)$)

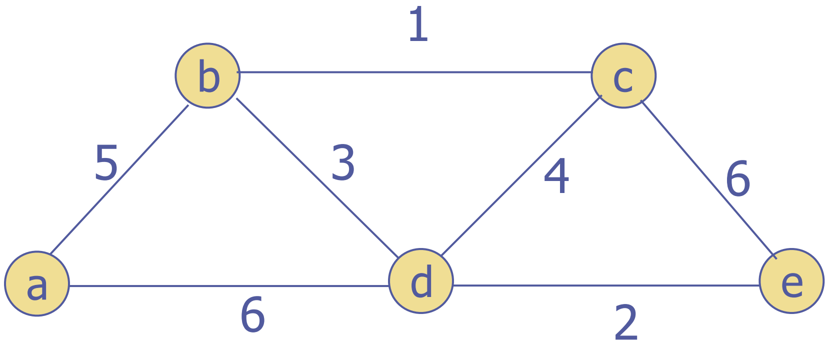
How does this work?

- Suppose that C and D are the two connected components in the forest (V,E_T) that are connected by an edge (u,v).

- Then (u,v) must have the least weight of any edge between C and D (otherwise C and D would have already been connected).

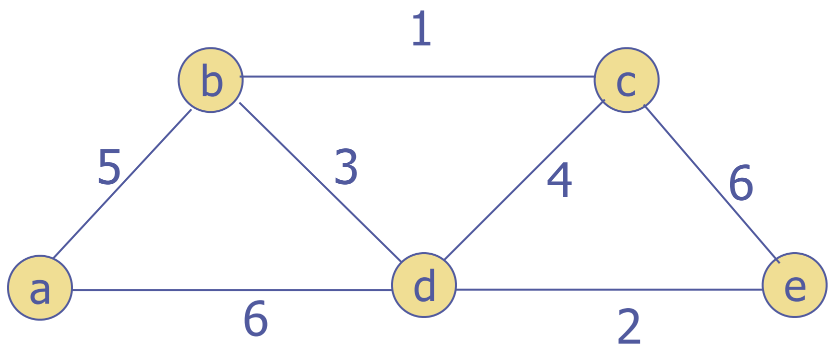
Your turn!

- Apply Kruskal's algorithm to this graph:



Your turn!

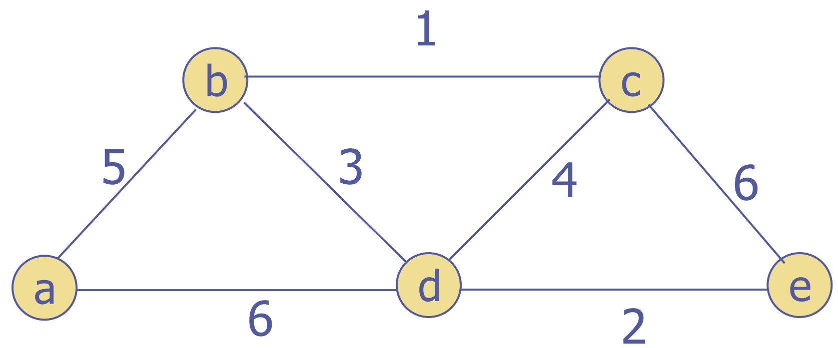
- Apply Kruskal's algorithm to this graph:



Tree edges	List of edges (sorted by weight)
	bc_1 de_2 bd_3 cd_4 ab_5 ad_6 ce_6

Your turn!

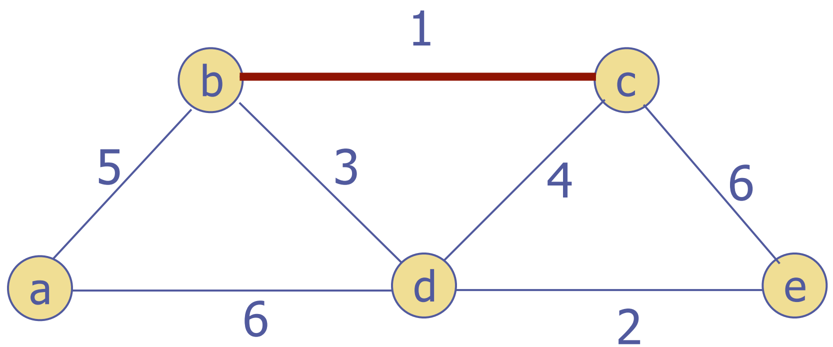
- Apply Kruskal's algorithm to this graph:



Tree edges	List of edges (sorted by weight)
bc_1	de_2 bd_3 cd_4 ab_5 ad_6 ce_6

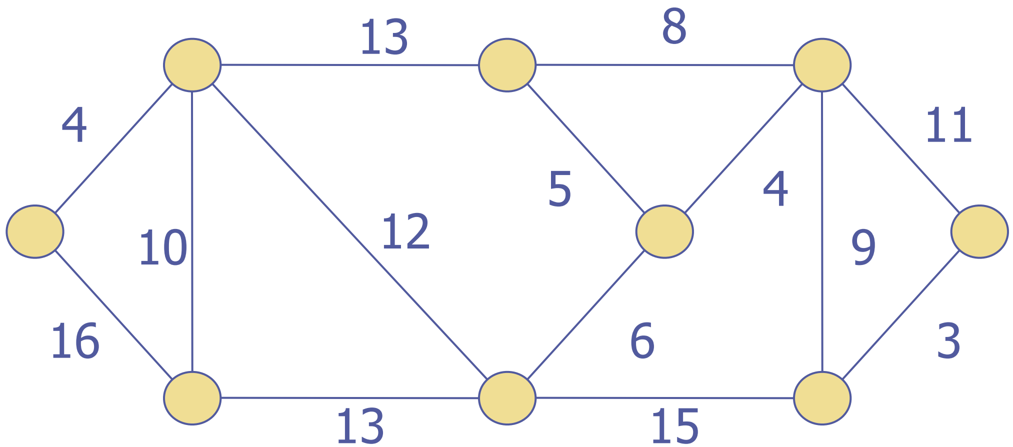
Your turn!

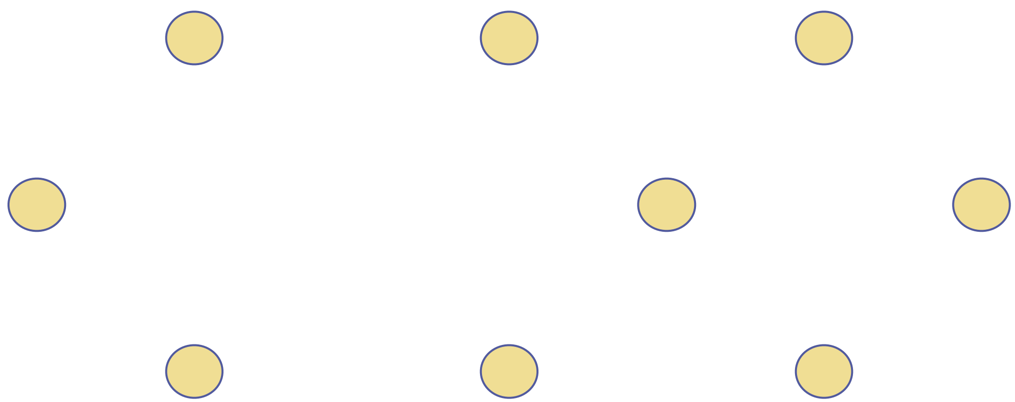
- Apply Kruskal's algorithm to this graph:



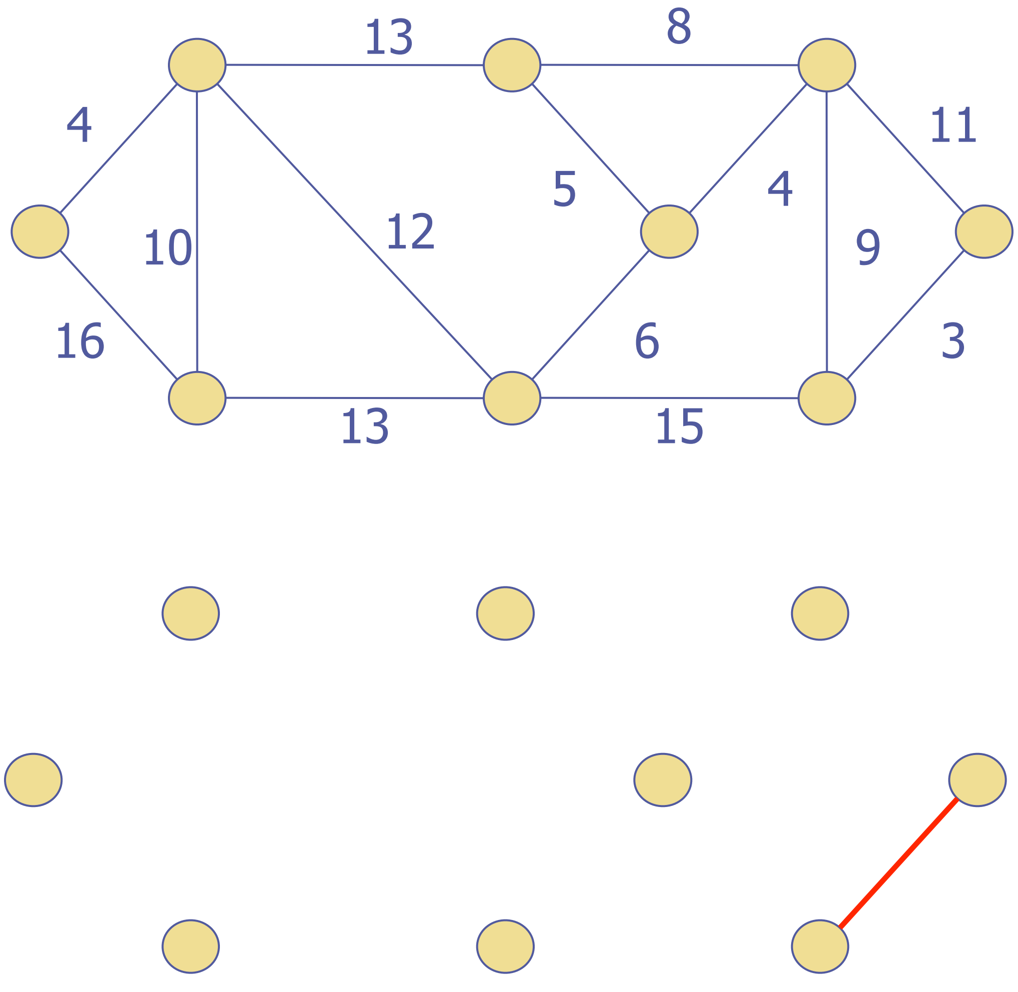
Tree edges	List of edges (sorted by weight)
bc_1	de_2 bd_3 cd_4 ab_5 ad_6 ce_6

Try it out!

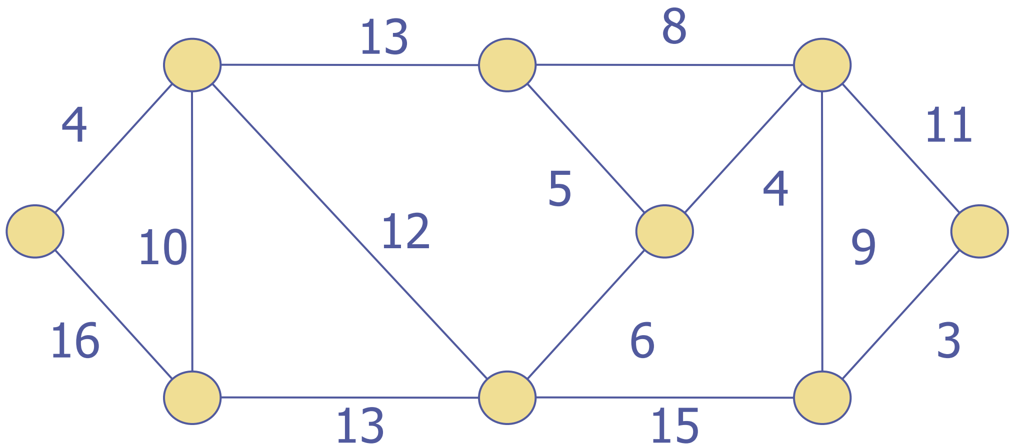


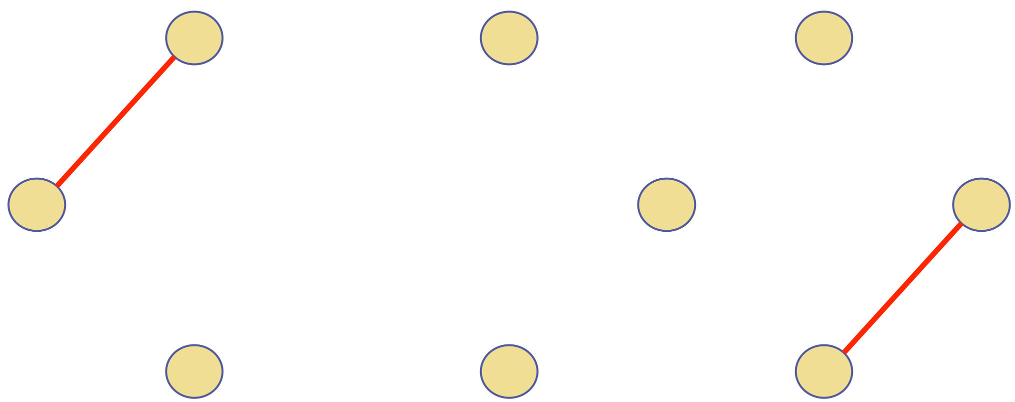


Try it out!

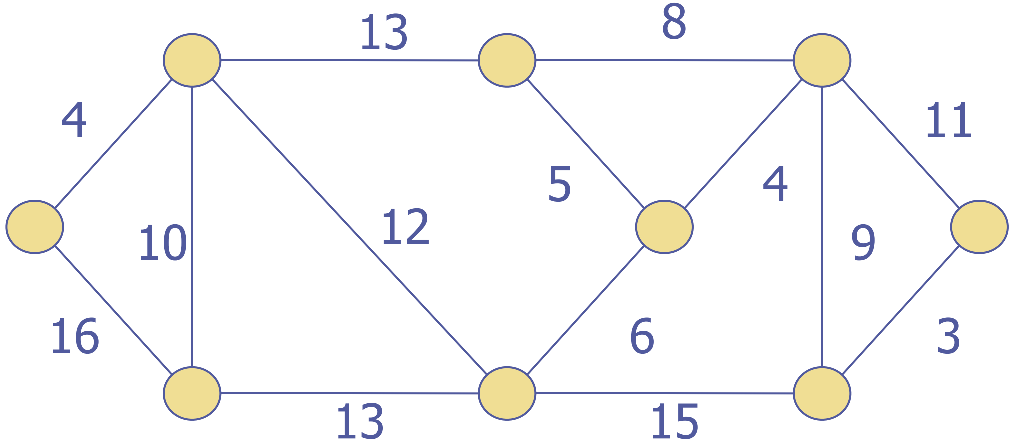


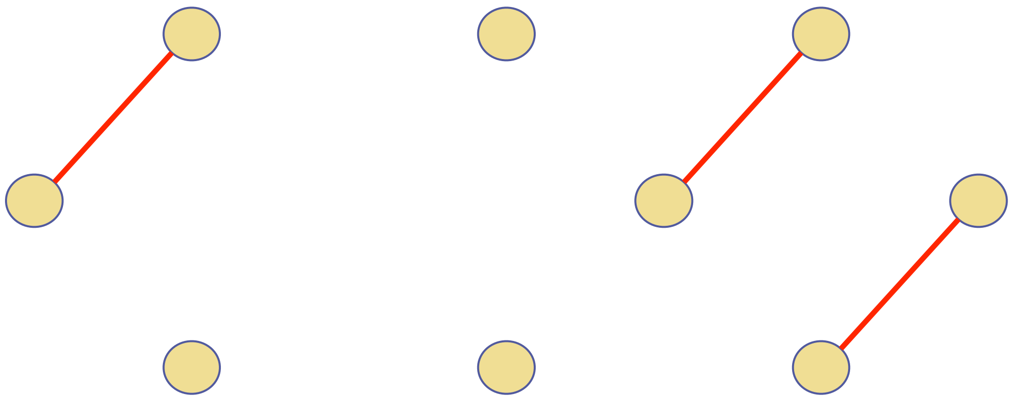
Try it out!



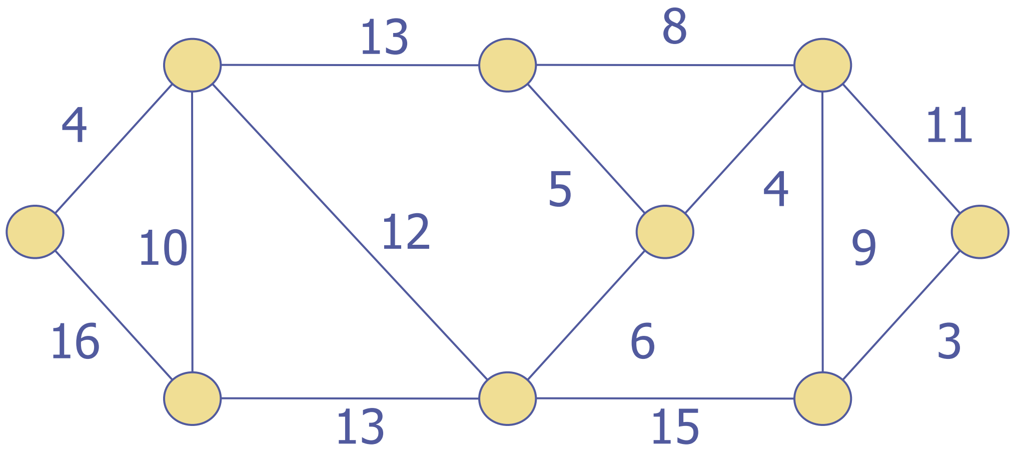


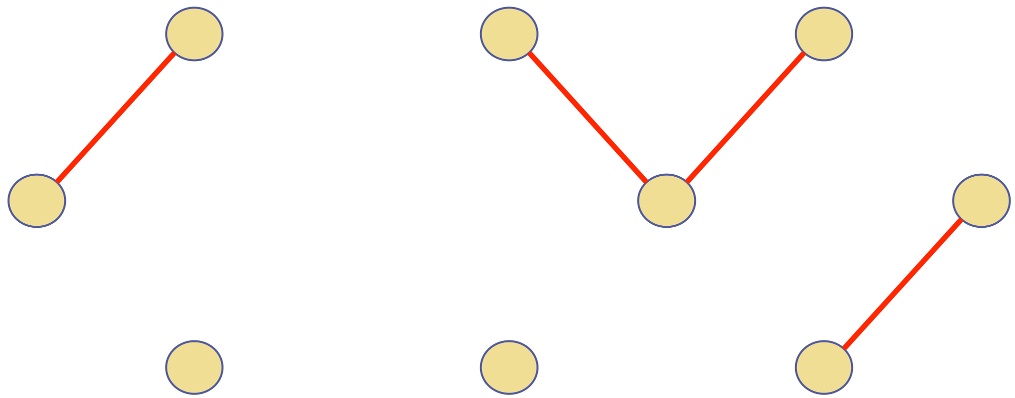
Try it out!



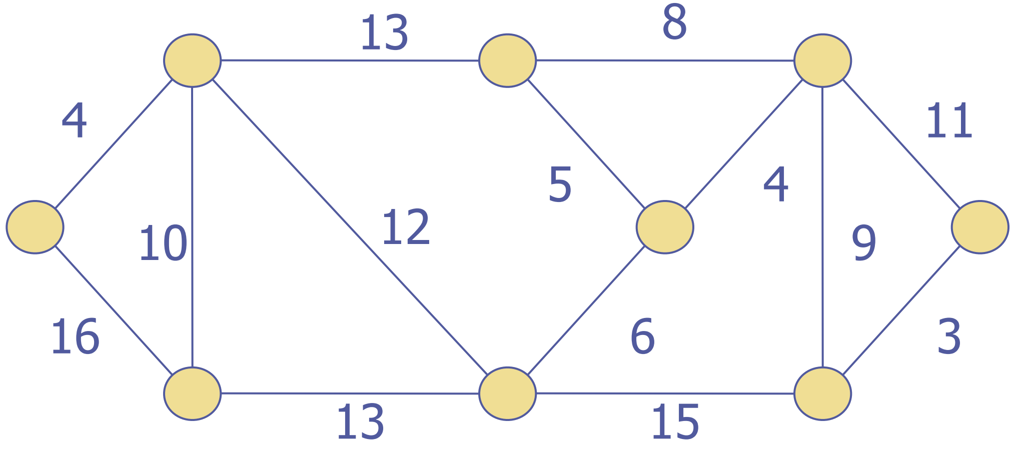


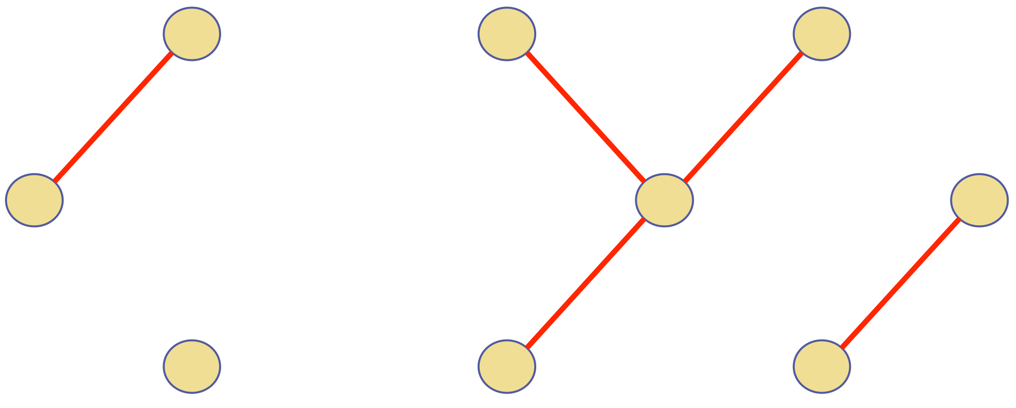
Try it out!



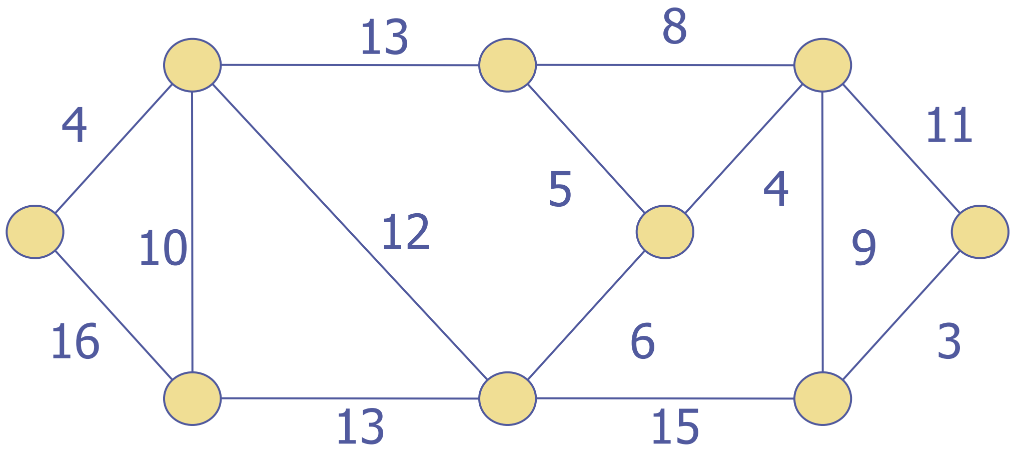


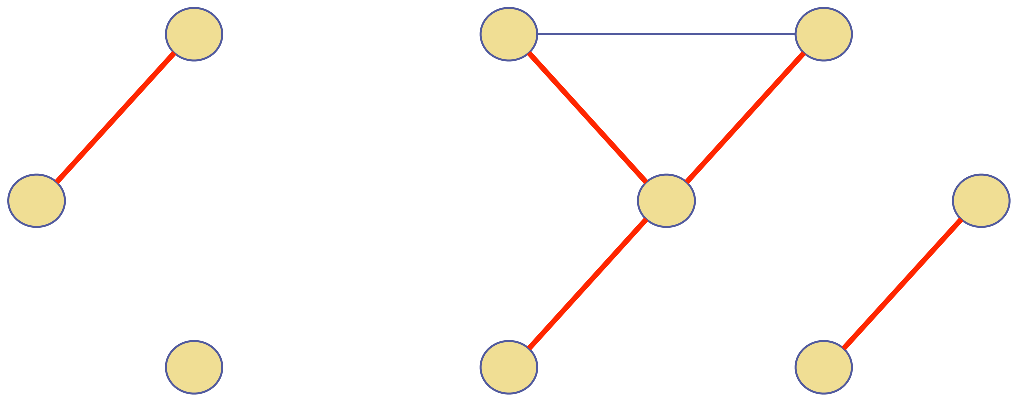
Try it out!



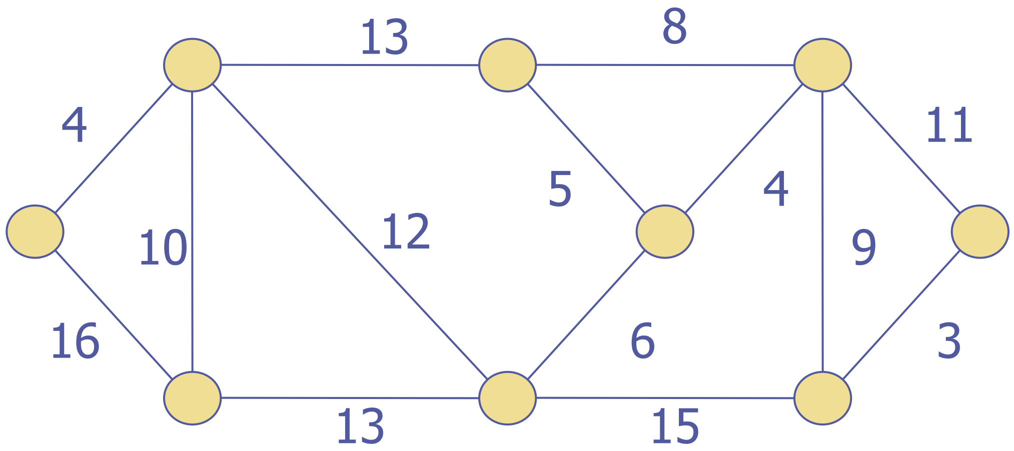


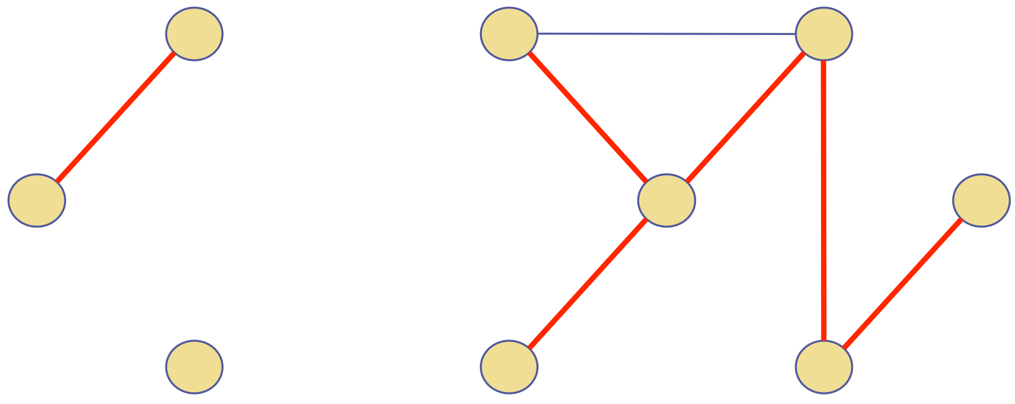
Try it out!



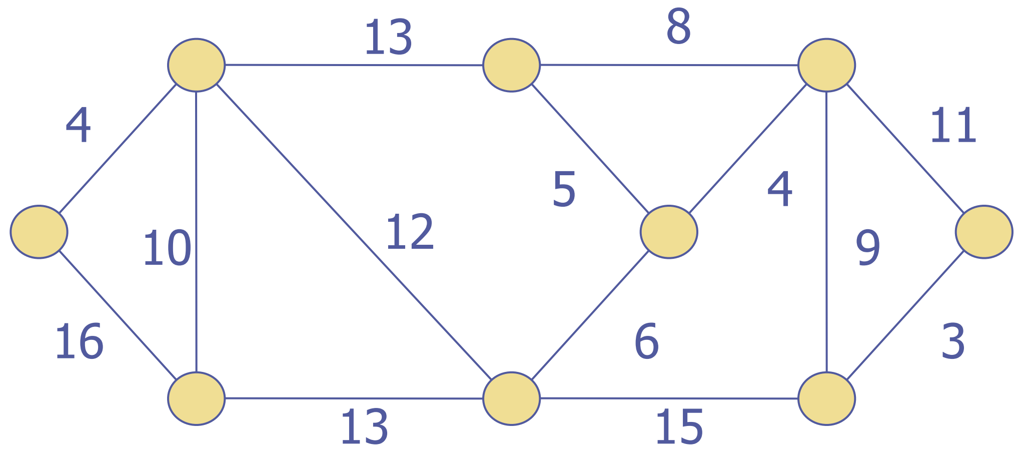


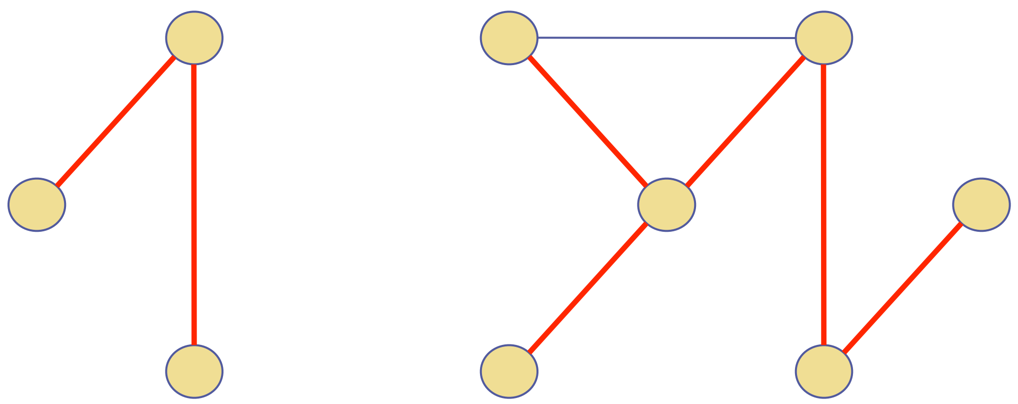
Try it out!



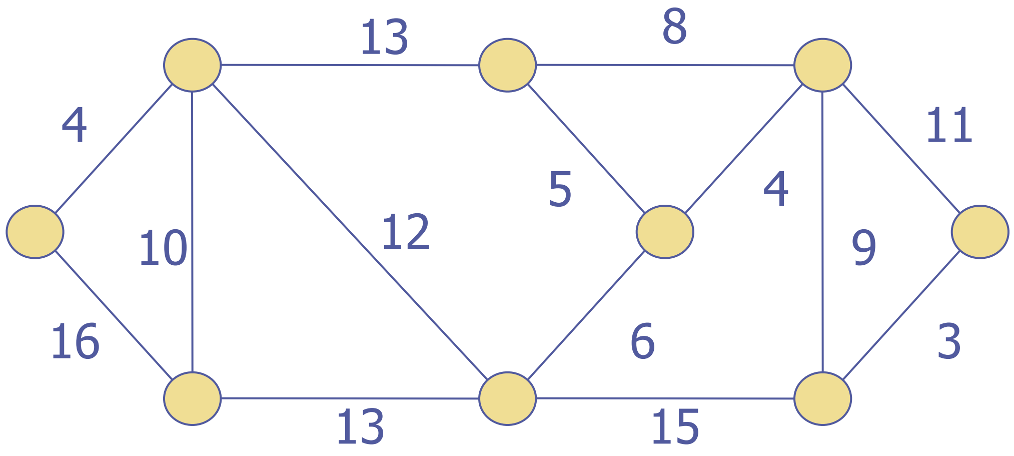


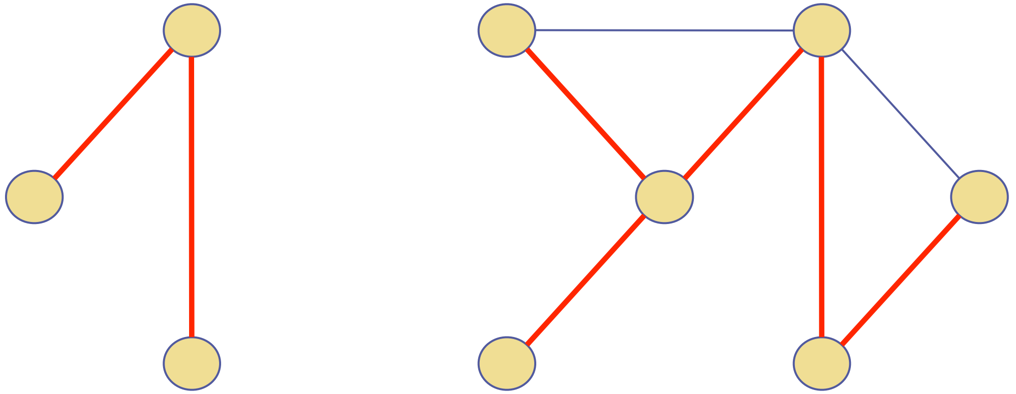
Try it out!



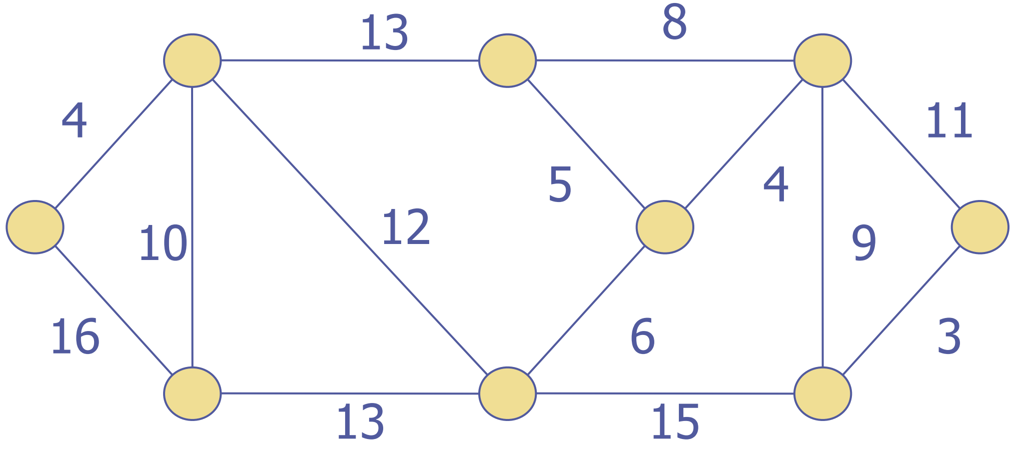


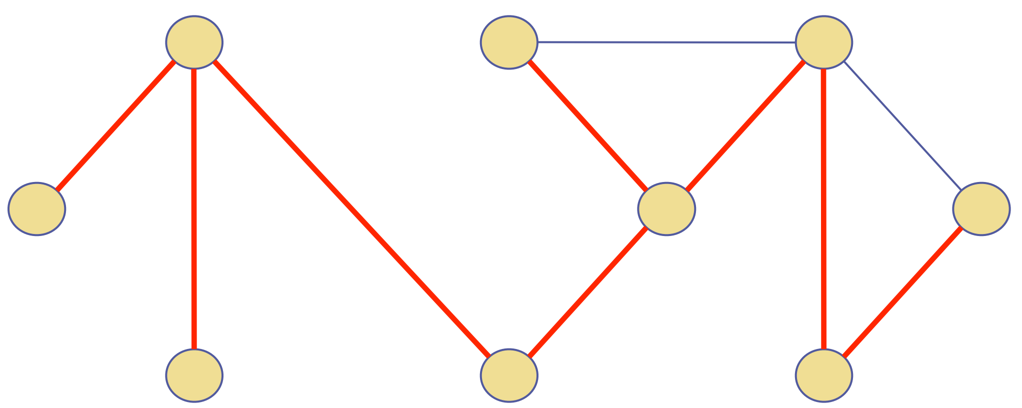
Try it out!



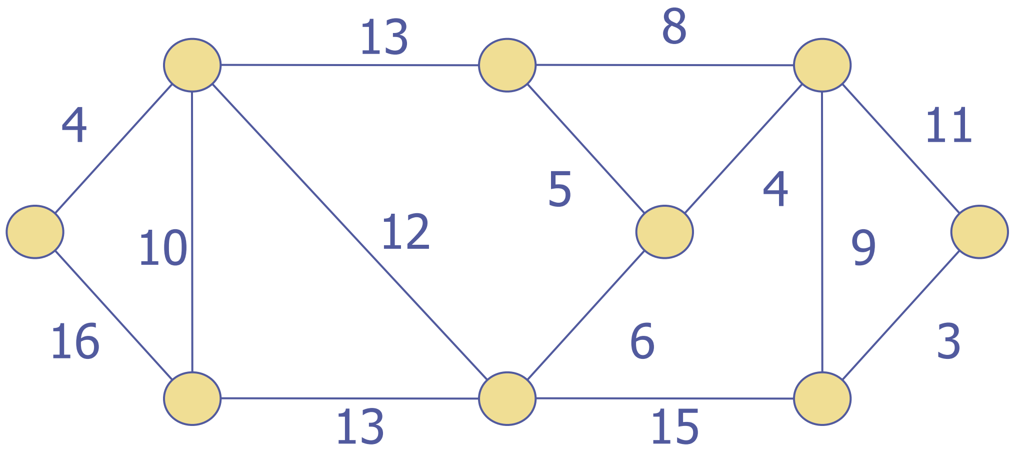


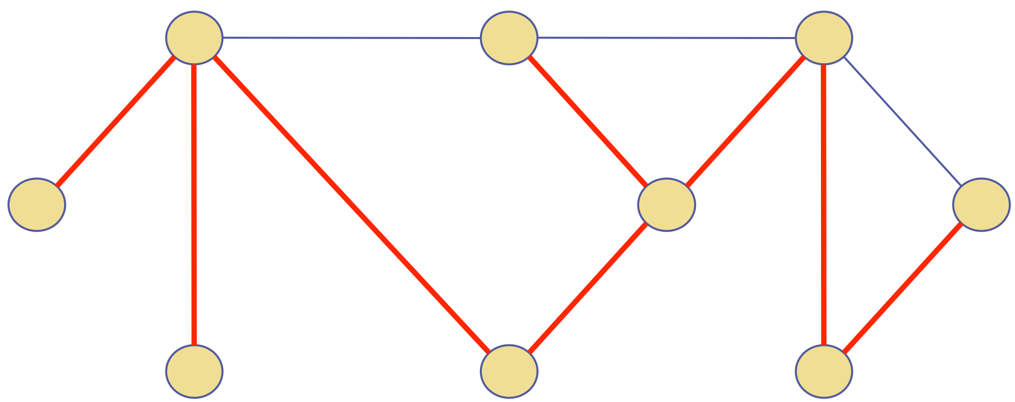
Try it out!



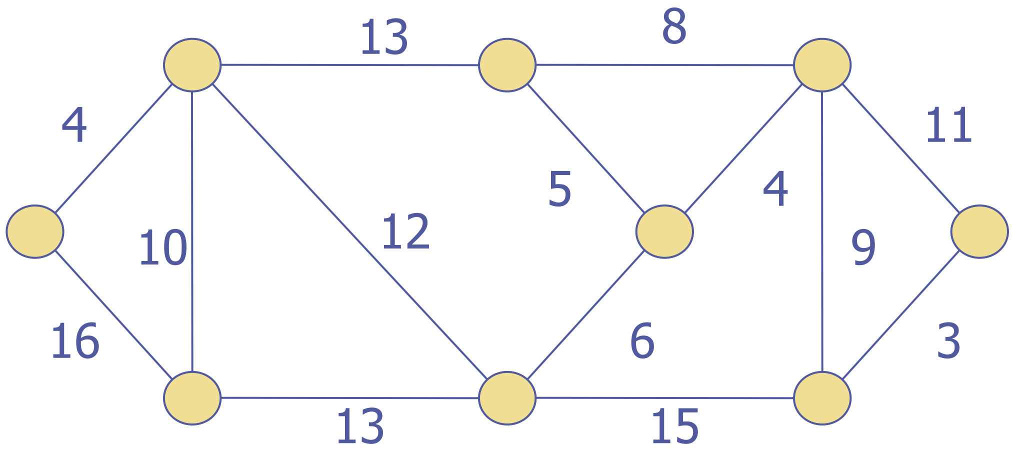


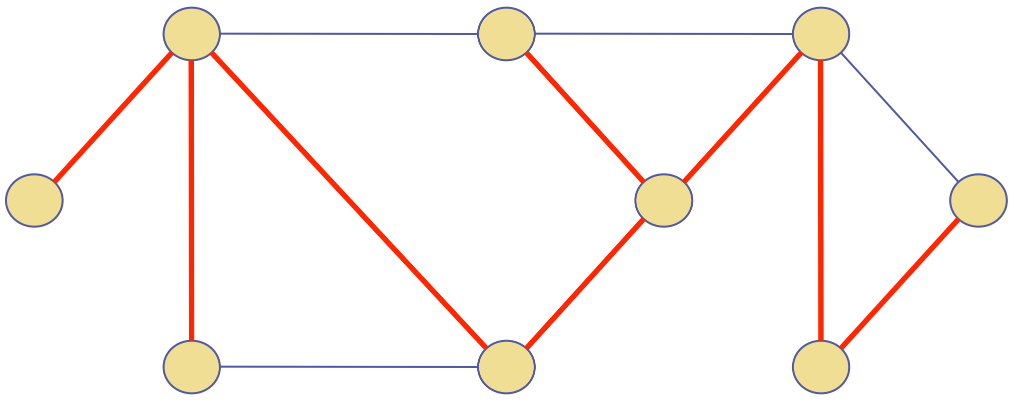
Try it out!



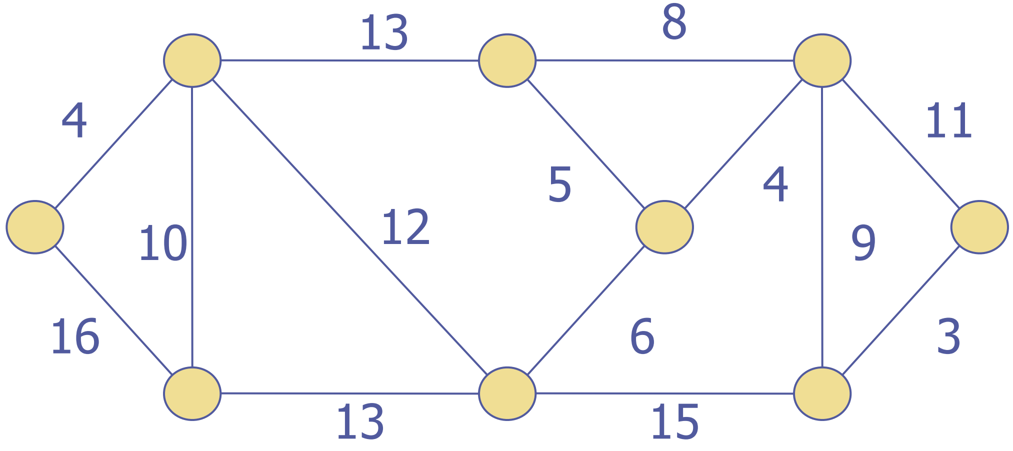


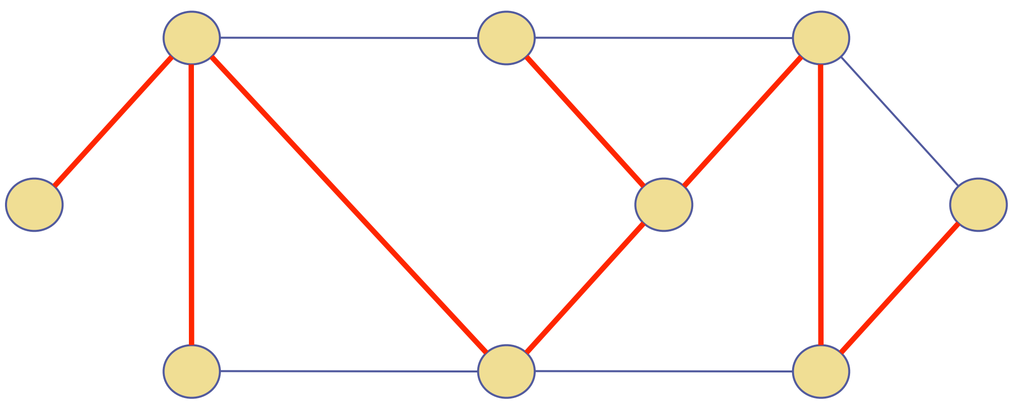
Try it out!



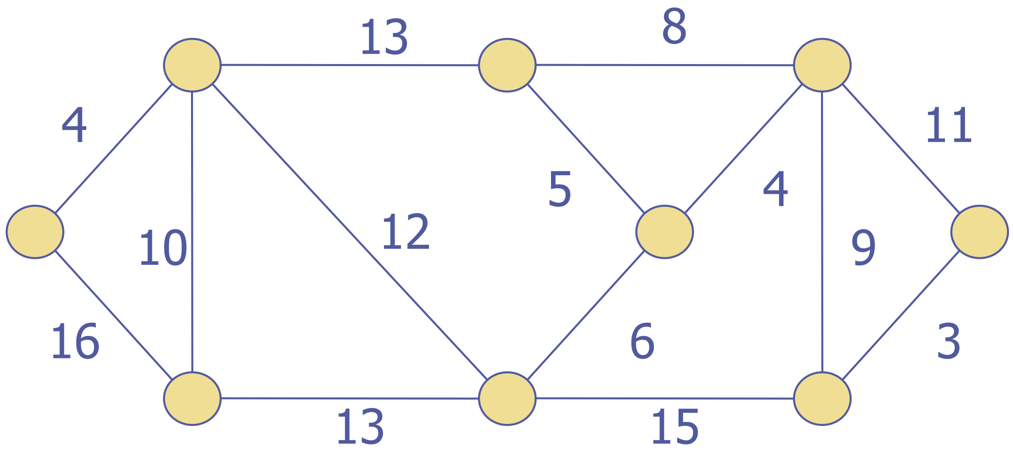


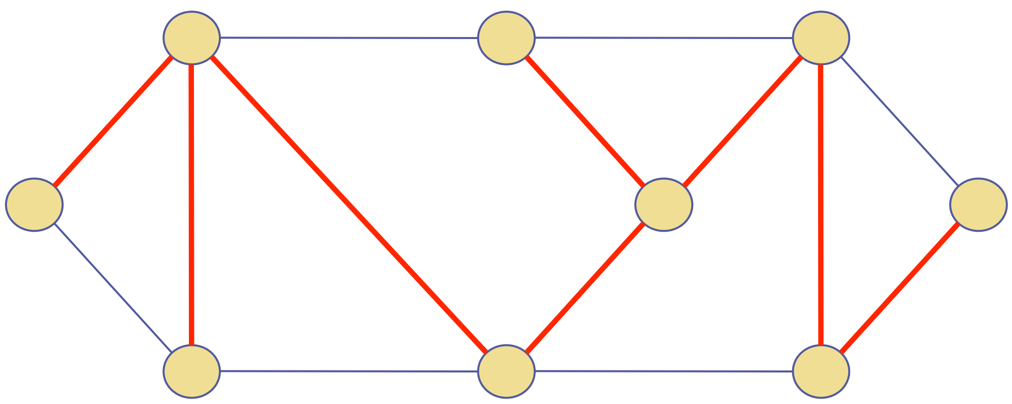
Try it out!



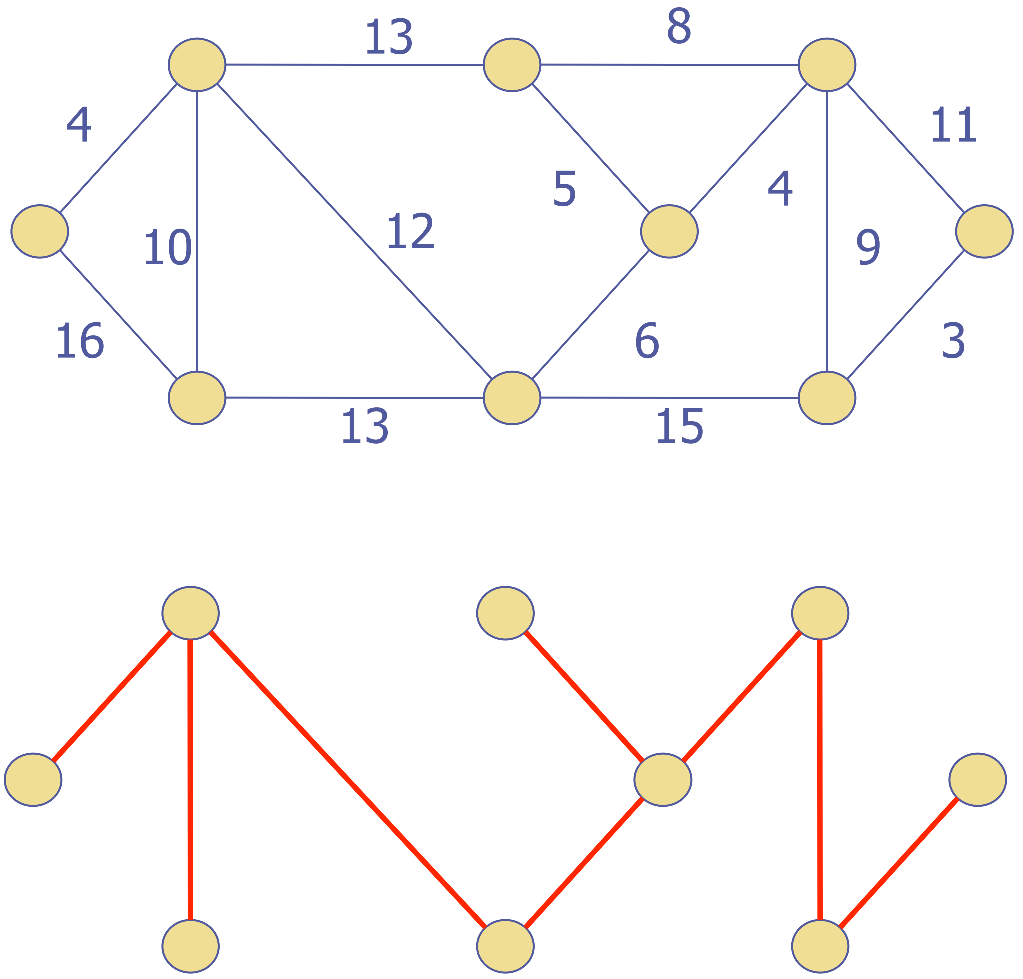


Try it out!

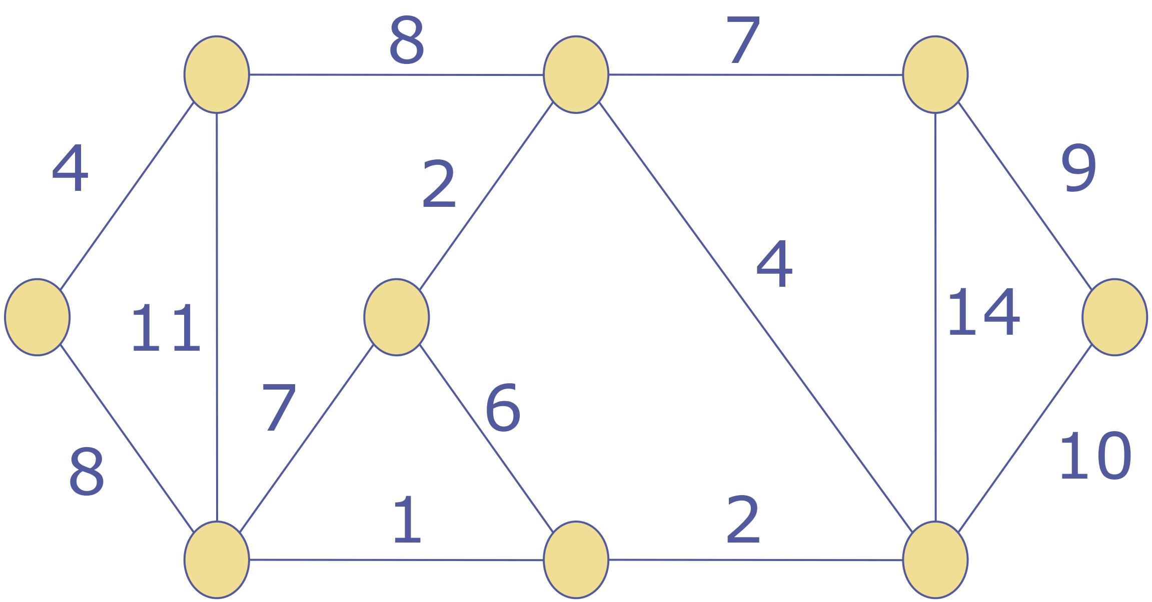




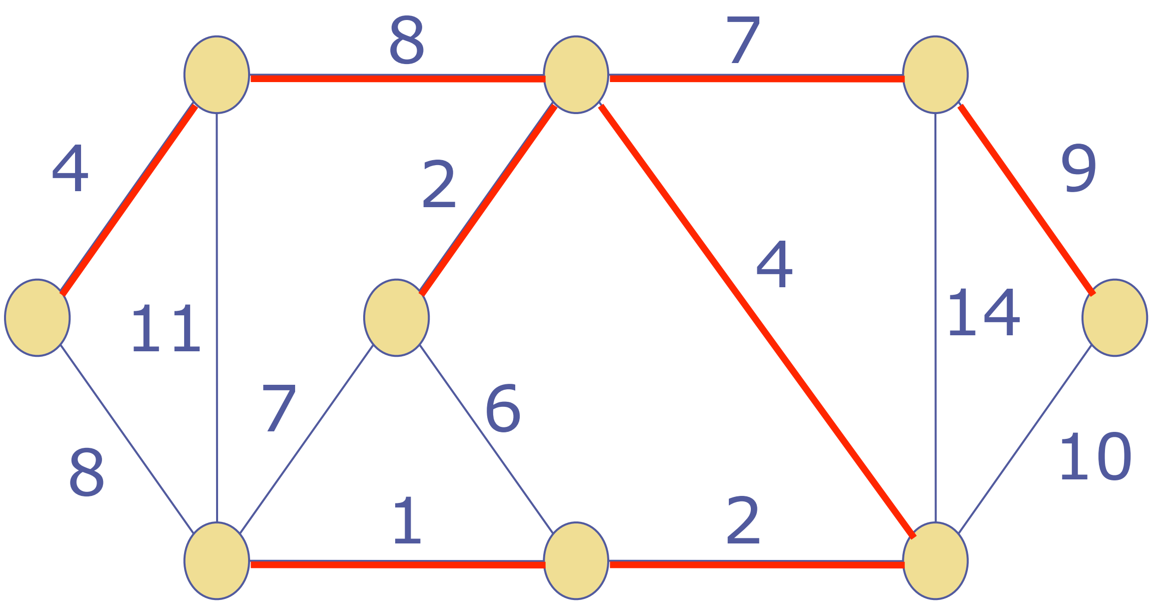
Try it out!



Try it out!



Try it out!



Prim’s Algorithm

Prim's algorithm:

- In Kruskal's algorithm, E_T is a forest whose components are combined as the algorithm runs until just one component remains.
- Suppose instead that we start with an arbitrary vertex r, and then add edges while ensuring that E_T is just a single tree at each stage.
- This is the essence of Prim's algorithm:

```
let VT ← {r}, ET ← empty set
while (VT ≠ V)
    find a light edge (u,v) for some u∈VT and v∉VT
    add v to VT; add (u,v) to ET
(V, ET) is the required MST
```

Why does this work?

- V_T cuts V into two pieces: V_T and $(V - V_T)$;
- The edges that we add to E_T are light edges across the cut;
- Hence, they are safe to add.

Choosing the edges:

- Store all vertices that are <u>not</u> in (V_T, E_T) in a priority queue Q with an extractMin operation.

- If u is a vertex in Q, what's key[u] (the value that determines u's position in Q)?
  - key[u] = minimum weight of edge from u into V_T
  - if no such edge exists, key[u] = ∞.

- We maintain information about the parent (in (V_T, E_T)) of each vertex v in an array parent[].
  - E_T is kept implicitly as $\{(v, parent[v]) \mid v \in V - Q - \{r\}\}$.

- The input is the graph $G=(V, E)$, and a root $r \in V$.

```
for each v in V
   key[v]    ← ∞;
   parent[v] ← null;
key[r] ← 0;
add all vertices in V to the queue Q.

while (Q is nonempty) {
   u ← extractMin(Q);
   for each vertex v that is adjacent to u {
      if v ∈ Q and weight(u,v) < key[v] {
         parent[v] ← u;
         key[v]    ← weight(u,v);
      }
   }
}
```

- The input is the graph $G=(V, E)$, and a root $r \in V$.

```
for each v in V
   key[v]    ← ∞;
   parent[v] ← null;
key[r] ← 0;
add all vertices in V to the queue Q.

while (Q is nonempty) {
   u ← extractMin(Q);
   for each vertex v that is adjacent to u {
      if v ∈ Q and weight(u,v) < key[v] {
         parent[v] ← u;
         key[v]    ← weight(u,v);
      }
   }
}
```

How can this test be implemented in O(1)?

- The input is the graph $G=(V, E)$, and a root $r \in V$.

```
for each v in V
    key[v]     ← ∞;
    parent[v] ← null;
key[r] ← 0;
add all vertices in V to the queue Q.

while (Q is nonempty) {
    u ← extractMin(Q);
    for each vertex v that is adjacent to u {
        if v ∈ Q and weight(u,v) < key[v] {
            parent[v] ← u;
            key[v]     ← weight(u,v);
        }
    }
}
```

How can this test be implemented in O(1)?

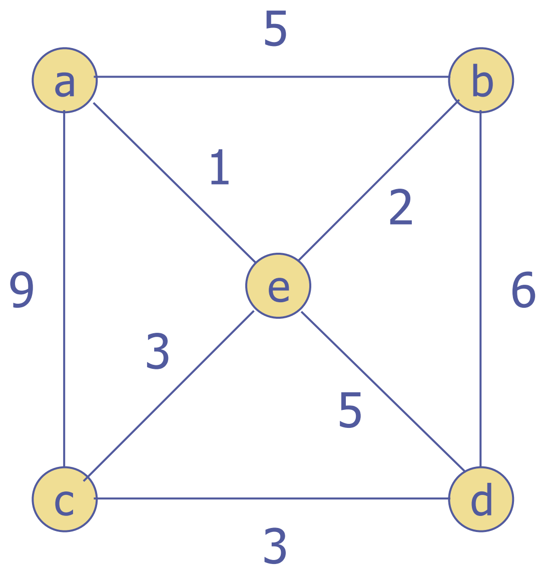
When we change a key, we will also need to readjust the priority queue

Complexity:

- Assuming a binary heap …
  - Initialization takes $O(|V|)$ time.
  - Main loop is executed $|V|$ times, and each extractMin takes $O(\log |V|)$.
  - The body of the inner loop is executed a total of $O(|E|)$ times; each adjustment of the queue takes $O(\log |V|)$ time.
- Overall complexity: $O((|V|+|E|) \log |V|) = O(|E| \log |V|)$.

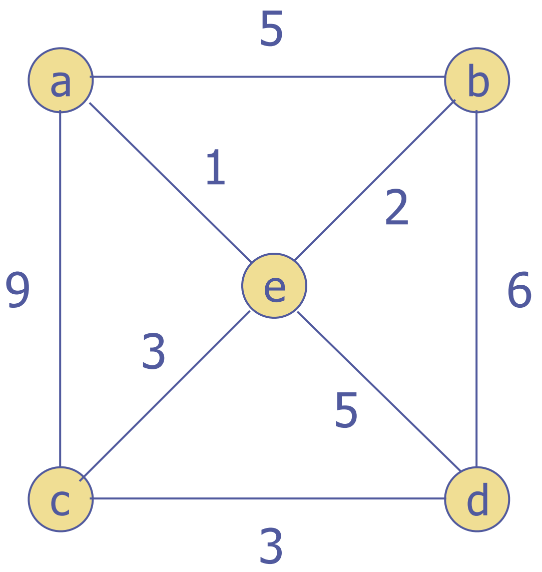
Your turn!

- Apply Prim's algorithm to this graph:



Your turn!

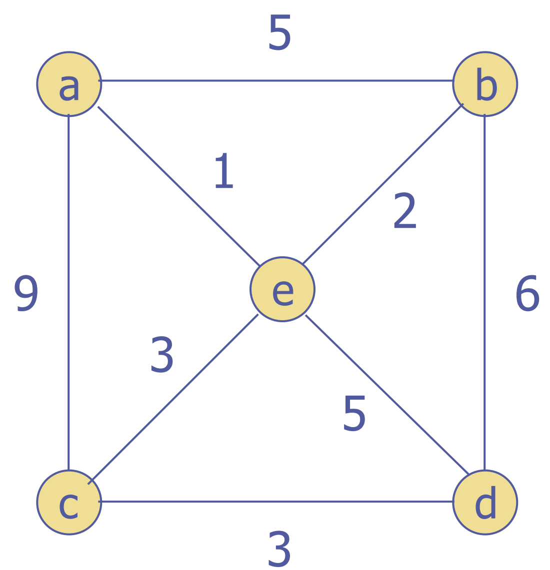
- Apply Prim's algorithm to this graph:



Tree vertices	Priority Queue of remaining vertices

Your turn!

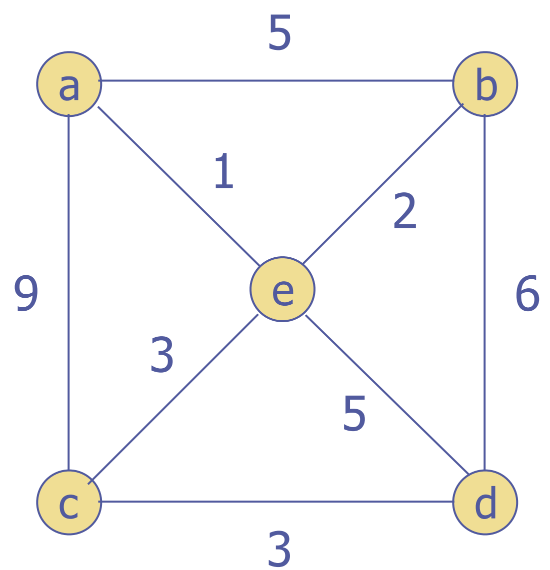
- Apply Prim's algorithm to this graph:



Tree vertices	Priority Queue of remaining vertices
a(−, −)	

Your turn!

- Apply Prim's algorithm to this graph:

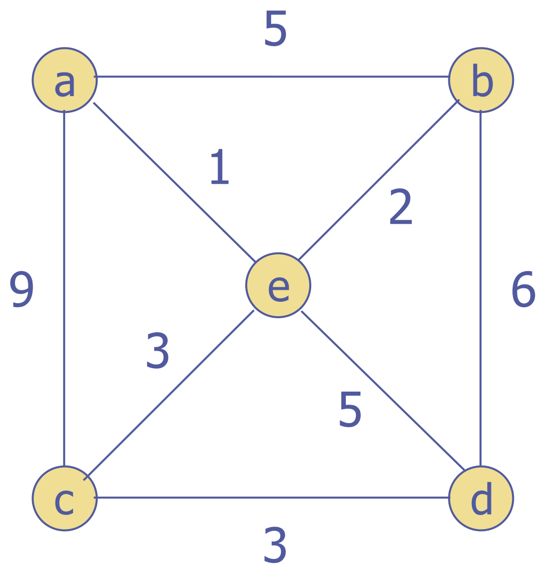


parent

Tree vertices	Priority Queue of remaining vertices
a(−, −)	

Your turn!

- Apply Prim's algorithm to this graph:

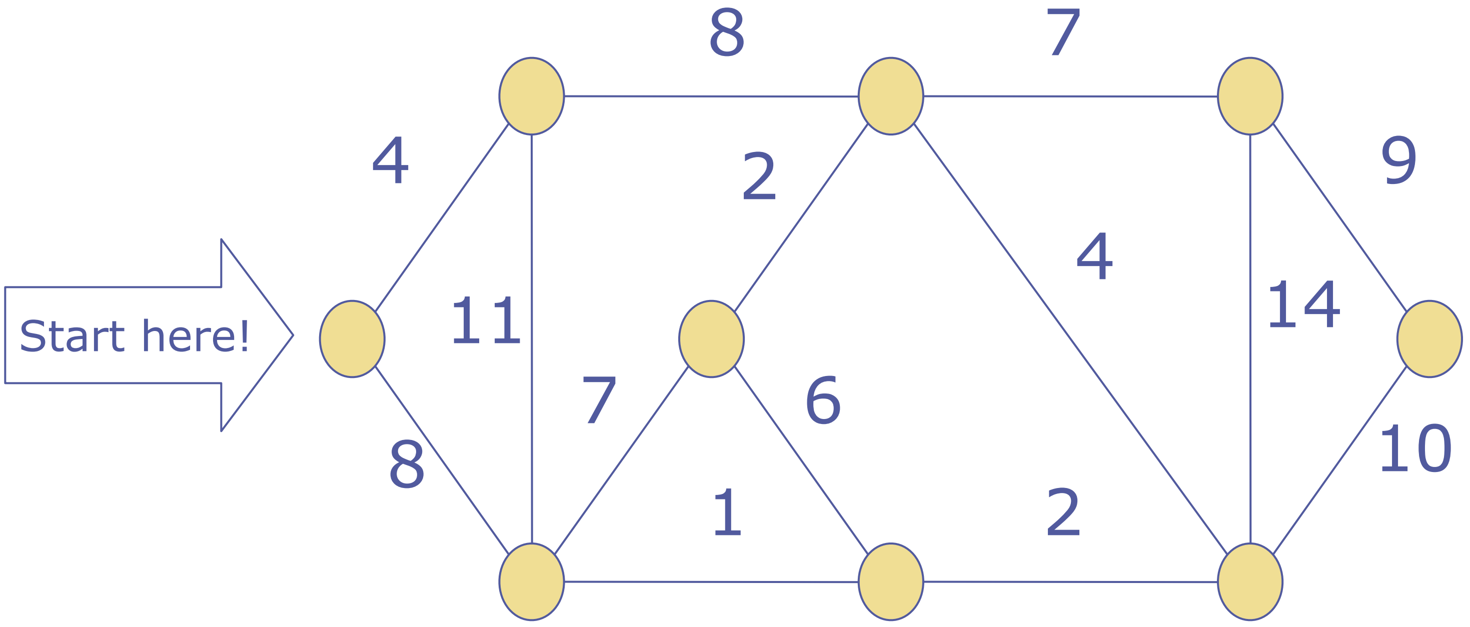


weight of edge

parent

Tree vertices	Priority Queue of remaining vertices
a(−, −)	

Apply Prim's Algorithm



Try it out!