

# Analysis of Algorithms

V. Adamchik

CSCI 570

Lecture 4

University of Southern California

Fall 2023

## Greedy Algorithms

Reading: chapter 4

# Heaps for Priority Queue

	Binary	Binomial	Fibonacci
findMin	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
deleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$ (ac)
insert	$\Theta(\log n)$	$\Theta(1)$ (ac)	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)
merge	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)

# Review

Another proof is that in lecture 1 we proved that the lower bound for sorting is  $\Omega(n \log n)$

1. (T/F) By using a binomial heap we can sort data of size  $n$  in  $O(n)$  time.

- ① First we will need to turn the raw data into a heap
- ② Then run `deleteMin` to sort  $\therefore$  we cannot sort in  $O(n)$

2. Prove that it is impossible construct a min-heap (not necessarily binary) in a comparison-based model with both the following properties:

- a. `deleteMin()` runs in  $O(1)$ ,  $\hookrightarrow$  in these algorithms the sorting order can be determined only based on comparisons between input elements
- b. `buildHeap()` runs in  $O(n)$ , where  $n$  is the input size.

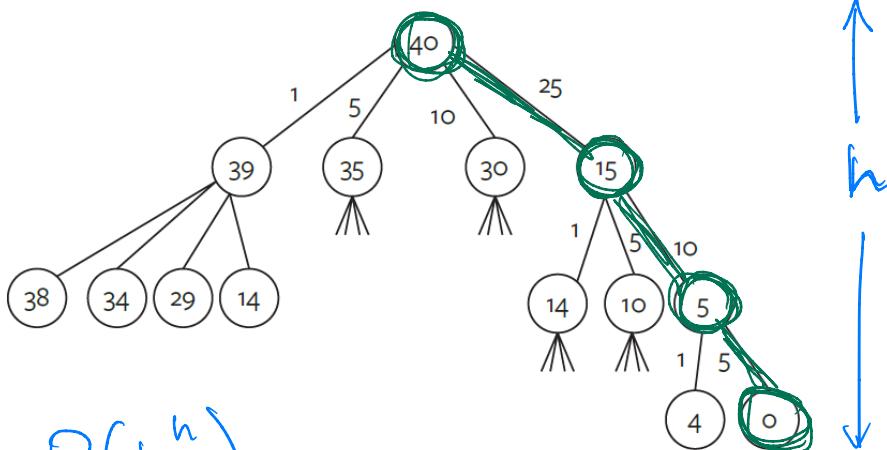
1. `'deleteMin()'`: In a comparison-based model, deleting the minimum element from a min-heap requires reorganizing the heap to maintain its properties. To achieve  $O(1)$  time complexity for `'deleteMin()'`, you would need to somehow know the minimum element in the heap without examining all the elements, which is not possible in a comparison-based model. In a comparison-based model, you must compare elements to determine their order, and finding the minimum element will always take at least  $O(n)$  time because you have to examine each element at least once.

2. `'buildHeap()'`: Building a heap with  $O(n)$  time complexity means that you can construct a heap from an arbitrary array of  $n$  elements without making more than  $O(n)$  comparisons. Again, this is not possible in a comparison-based model because each comparison involves examining two elements. To build a heap, you need to compare and potentially swap elements to satisfy the heap property, which requires  $\Omega(n \log n)$  comparisons in the worst case.

Therefore, it is impossible to have both `'deleteMin()'` in  $O(1)$  and `'buildHeap()'` in  $O(n)$  in a comparison-based model. These two properties are conflicting in such a model.

# The Money Changing Problem

We are to make a change of \$0.40 using US currency and assuming that there is an unlimited supply of coins. The goal is to compute the minimum number of coins.



penny , nickel , dime , quarter

greedy approach :

1° choose the largest coin

Runtime :  $O(h)$

$O(4^h)$ .

Since using a DFS to find the leaf nodes for the solution, the runtime will be exponential! Therefore, greedy approach

## TRADITIONAL APPROACH

During the algorithm execution, we don't consider all available choices at any given point, but use a heuristic (greedy choice) to pick just one.

# What is Greedy Algorithm?

There is no formal definition...

- It is used to solve optimization problems
- It makes a local optimal choice at each step
- Earlier decisions are never undone
- Does not always yield the optimal solution

# Elements of the greedy strategy

There is no guarantee that such a greedy algorithm exists, however a problem to be solved must obey the following two common properties:

greedy-choice property

and

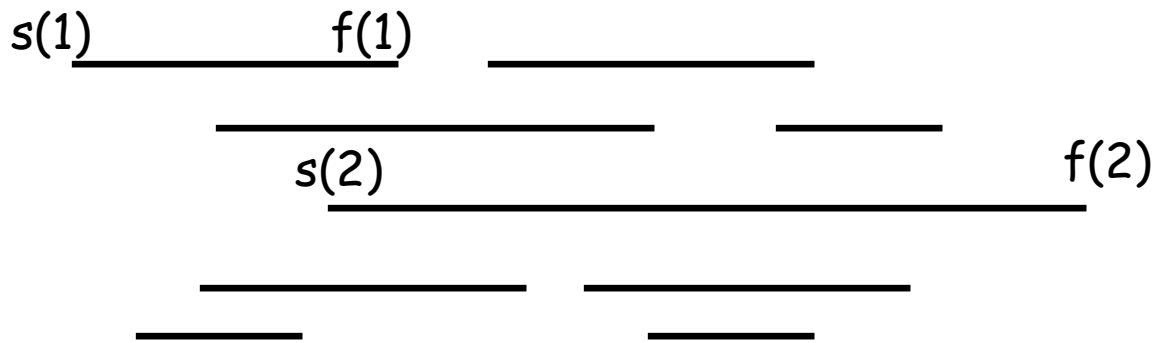
optimal substructure.

The proof of optimal substructure correctness is usually by induction.

The proof that a greedy choice for each subproblem yields a globally optimal solution is usually by contradiction.

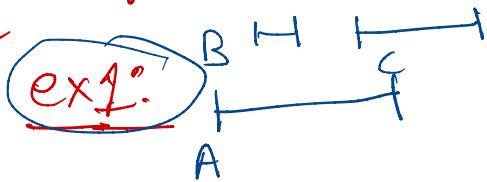
# Scheduling Problem

There is a set of  $n$  requests. Each request  $i$  has a starting time  $s(i)$  and finish time  $f(i)$ . Assume that all requests are equally important and  $s(i) \leq f(i)$ . Our goal is to develop a greedy algorithm that finds the largest compatible (non-overlapping) subset of requests.



# How do we choose requests?

① Sort by starting time  $s(i)$ , choose the earliest time



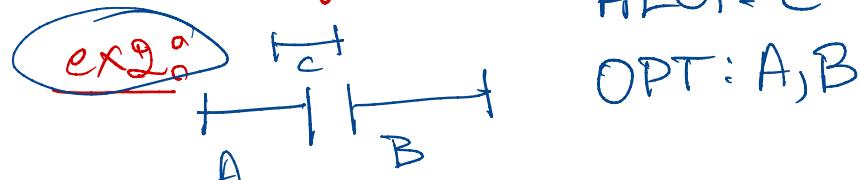
ex1:

ALG: A

OPT: B, C

In this case the algorithm can choose only 1 interval. But the optimal case is able to choose 2. ∴ one greedy approach sucks.

② Sort by  $f(i) - s(i)$



ex2:

ALG: C

OPT: A, B

Greedy approach is hard because there are many choices but there is only one correct answer. It is difficult to tell. When you come up w/ an idea think of a counter example. If you can't think of any then try to prove it.

③ Sort by  $f(i)$

ex 1: ALG: B, C

ex 2: ALG: A, B

# Proof

In this approach we sort requests with respect to  $f(i)$  in ascending order. Pick a request that has the earliest finish time. Since we don't know what the optimal is, we need to prove that we do better than or same as the worst case.

ALG:  $i_1, i_2, \dots, i_k$

OPT:  $j_1, j_2, \dots, j_m$

Prove  $f(i_r) \leq f(j_r)$   
by induction on r

[Here we are not saying that we repeated the same interval, we are saying that our finish time  $f(i)$  cannot be bigger]

Proof:

Base case:  $r=1$ , first request

$$f(i_1) \leq f(j_1)$$

Inductive Hypothesis (IH): assume  $f(i_{r-1}) \leq f(j_{r-1})$   
for  $(r-1)$  requests

Inductive step (IS): prove it for the next request

By IH we have,

$$f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$$

The next  
starting  
time

what does it mean?

Exists  $\exists$  another request  
our ALG will take it

Next step: prove  $k \geq n$   
proof by contradiction

Assume  $k < n$

$$\textcircled{1} \quad f(j_k) \leq s(j_{k+1})$$

compatible  
requests

finish time of previous request  
cannot be bigger than  
starting time of next request

$$\textcircled{2} \quad f(i_k) \leq f(j_k) \text{ by IH}$$

Combine them together

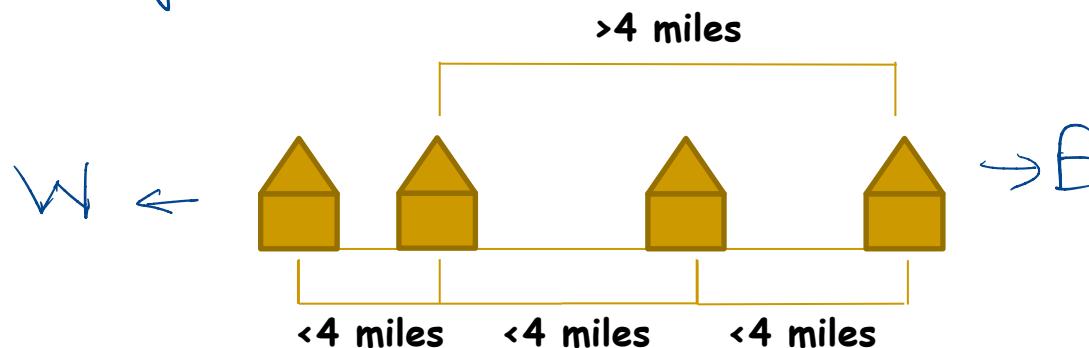
$$f(i_k) \leq f(j_k) \leq s(j_{k+1})$$

$\exists$  another request, ALG will pick it.

# Discussion Problem 1

Let's consider a long, quiet country road with  $n$  houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. You want to place cell phone base stations at certain points along the road so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal and uses as few base stations as possible.

input: array of houses,  $h_k$



## Algorithm:

- ① Sort  $h_k$ , from W to E
- ② Start with  $h_1$ , walk 4 miles, put a station



- ③ Mark all houses within 4 miles

Runtime:  $O(n \log n + n)$

Proof of correctness

ALG:  $S_1, S_2, \dots, S_k$

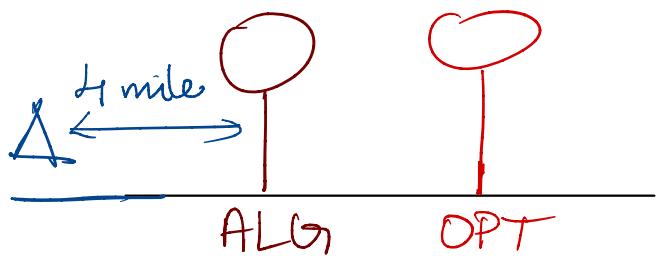
OPT:  $t_1, t_2, \dots, t_m$

Basecase: first station : walk 4 miles and put a station

Ind: assume ( $C-1$ ) stations

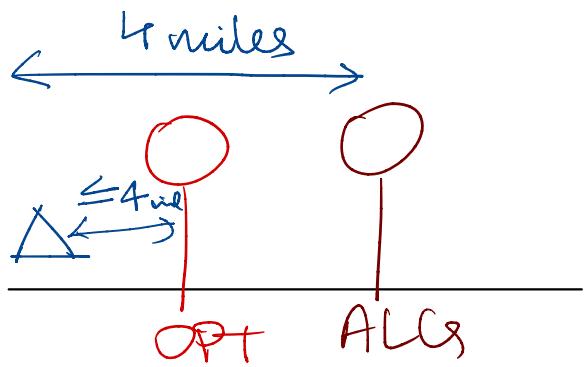
IS : prove for the next  $C$ -th station

A)



This is impossible as the optimal solution will never be that far apart.

B)



$\text{OPT \# stations} \geq \text{ALG \#}$

# The Minimum Spanning Tree



Given a weighted undirected graph. Find a spanning tree of the minimum total weight.

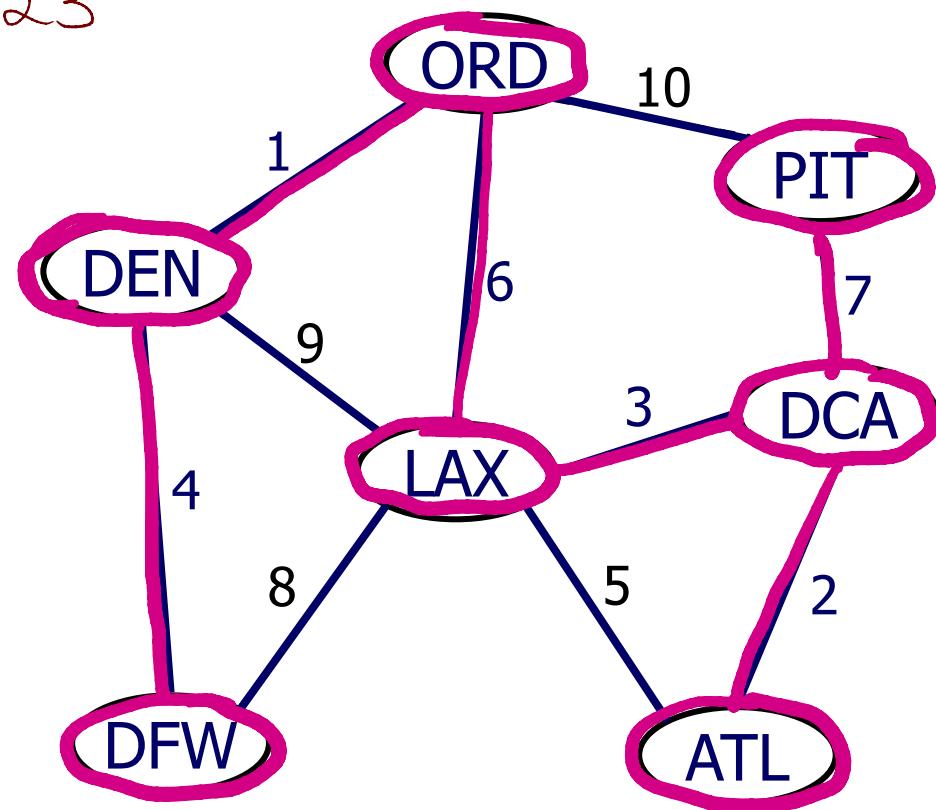
MST is fundamental problem with diverse applications.

# The Minimum Spanning Tree

→ no cycles

Find a **spanning tree** of the minimum total weight.

$$\text{cost} = 1 + 2 + 3 + 4 + 6 + 7 = 23$$



# Kruskal's Algorithm

The algorithm builds a tree one EDGE at a time:

- Sort all edges by their weights.  $O(E \log E)$
- Loop:
  - Choose the minimum weight edge and join correspondent vertices (subject to cycles). detecting a cycle =  $O(V)$
  - Go to the next edge.
  - Continue to grow the forest until all vertices are connected.

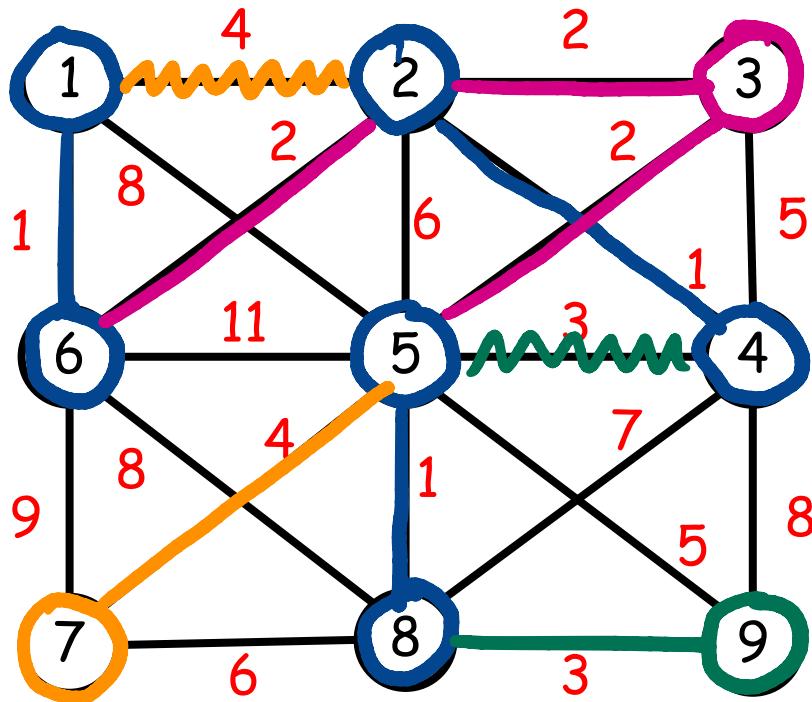
Sorting edges -  $O(E \log E)$

Cycle detection -  $O(V)$  for each edge

Total:  $O(V^*E + E^*\log E)$

Cycle detection      Sort

# Kruskal's Algorithm



Exercise: Suppose you do not sort edges.  
What is the runtime?

$$O(\underbrace{V \cdot E}_{\text{cycle detection}} + \underbrace{E \cdot E}_{\text{findMin}})$$

## Discussion Problem 2

You are given a graph  $G$  with all **distinct** edge costs. Let  $T$  be a minimum spanning tree for  $G$ . Now suppose that we replace each edge weight  $c_e$  by its square,  $c_e^2$ , thereby creating a new graph  $G_1$  with the different distinct weights. Prove or disprove whether  $T$  is still an MST for this new graph  $G_1$ .

$$c \rightarrow c^2$$

$$\text{MST}(G) = T \quad \text{Question: } T = T_1 ?$$

$$\text{MST}(G_1) = T_1$$

Case 1)  $\forall c \geq 0$ , then  $T = T_1$

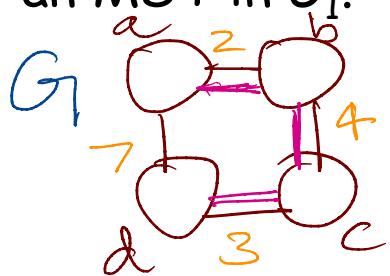
$$\begin{matrix} 1/2, & 1, & 2 \\ 1/4, & 1, & 4 \end{matrix} \xrightarrow{\text{Same}}$$

Case 2)  $\exists c < 0$ , then  $T \neq T_1$   
Sorting order

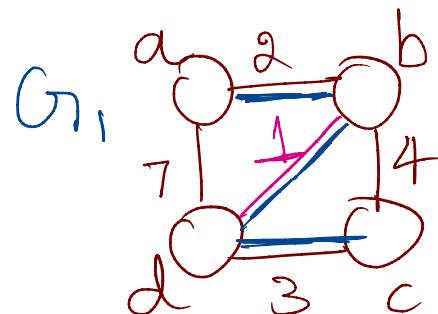
# Discussion Problem 3

You are given a minimum spanning tree  $T$  in a graph  $G = (V, E)$ .

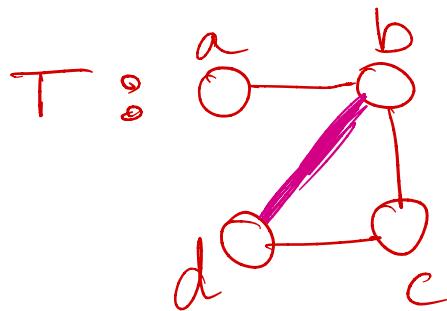
Suppose we add a new edge (without introducing any new vertices) to  $G$  creating a new graph  $G_1$ . Devise a linear time algorithm to find an MST in  $G_1$ .



$$\text{MST}(G) = T$$



$$\text{MST}(G_1) = T_1$$



Algorithm:

- ① add that new edge to  $T$
- ② Traverse the cycle and remove the largest edge.

# Prim's Algorithm

The algorithm builds a tree one VERTEX at a time:

- Start with an arbitrary vertex as a sub-tree  $C$ .
- Expand  $C$  by adding a vertex having the minimum weight edge of the graph having exactly one end point in  $C$ .
- Update distances from  $C$  to adjacent vertices.
- Continue to grow the tree until  $C$  gets all vertices.
- We make use of a Heap and for updating & deleting values we use deleteMin & decreaseKey

# Prim's Algorithm: Example

$$T = \{a\}$$

Heap: distances from  $T$  to all vertices

*root* *deleteMin*

b-4	c-2	d-1	e-infty	f-infty
-----	-----	-----	---------	---------

$$T = \{a, d\}$$

*root*

b-4	c-2	e-5	f-7
-----	-----	-----	-----

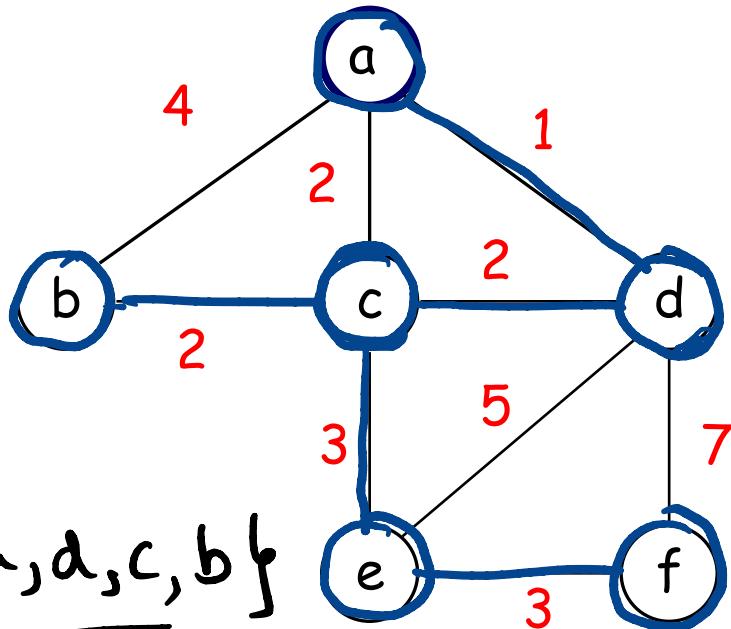
*deleteMin*

*decreaseKey*

$T = \{a, d, c\}$

b-2	e-3	f-7
-----	-----	-----

*decreaseKey*



$$T = \{a, d, c, b\}$$

e-3	f-7
-----	-----

$$T = \{a, d, c, b, e\}$$

f-3
-----

$$\underline{T = \{a, d, c, b, e, f\}}$$

# Complexity of Prim's Algorithm

Algorithm :

- ① deleteMin , run  $V$  times
- ② decreaseKey , run  $E$  times

Runtime :

- a) binary Heap  $O(V \cdot \log V + E \cdot \log V)$
- b) fibonacci Heap  $O(V \cdot \log V + E \cdot \frac{1}{\text{amortized cost}})$

# Discussion Problem 4

Assume that an **unsorted array** is used as a heap.

What would the running time of the Prim algorithm?

$$O(\underbrace{V \cdot V}_{V^2} + E \cdot 1)$$

It takes **linear** time to traverse through the array to find the minimum value for **deleteMin**.

It takes **constant** time to **decreaseKey** in an array as there are indices used to access elements.

Assume that we need to find an MST in a **dense** graph using Prim's algorithm. Which implementation (heap or array) shall we use?

In a **dense** graph  $E = O(V^2)$

∴ The runtime complexity

Binary Heap :  $O(V^2 \log V)$

Array :  $O(V^2)$

∴ use ARRAYS for DENSE GRAPHS

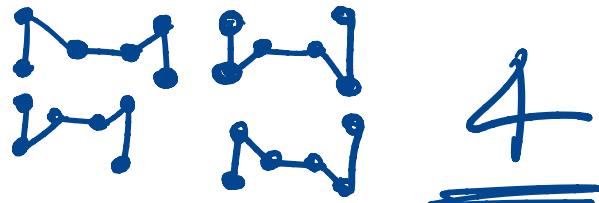
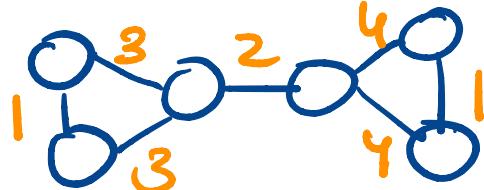
# Discussion T/F Questions

## Proof By Example

(T/F) The first edge added by Kruskal's algorithm can be the last edge added by Prim's algorithm.

## Proof By Example

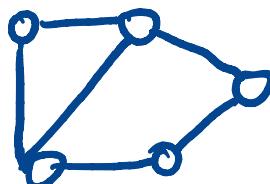
(T/F) Suppose we have a graph where each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in this graph.



(T/F) If a connected undirected graph  $G = (V, E)$  has  $V + 1$  edges, we can find the minimum spanning tree of  $G$  in  $O(V)$  runtime. Assignment 4

$$V = 5$$

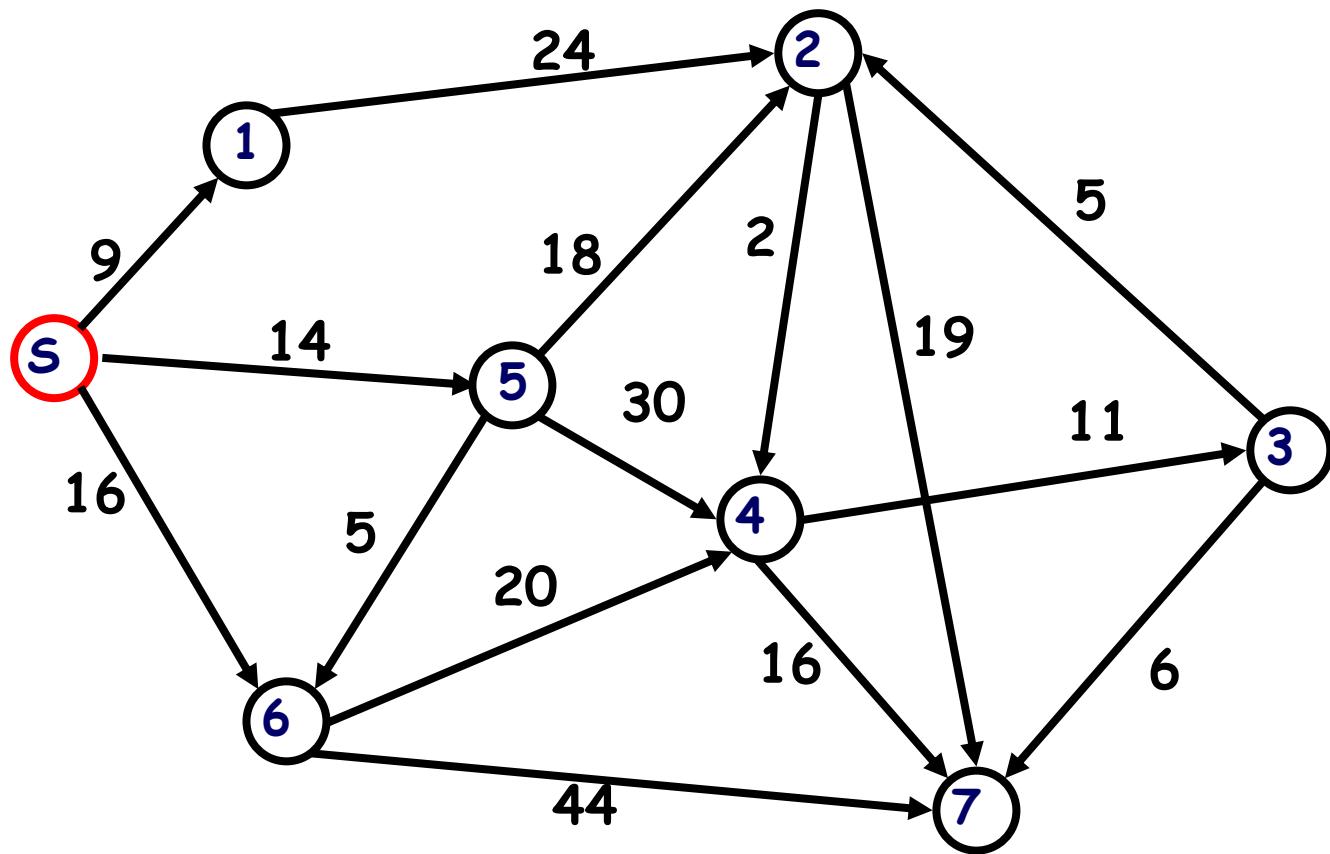
$$E = 6$$



# The Shortest Path Problem

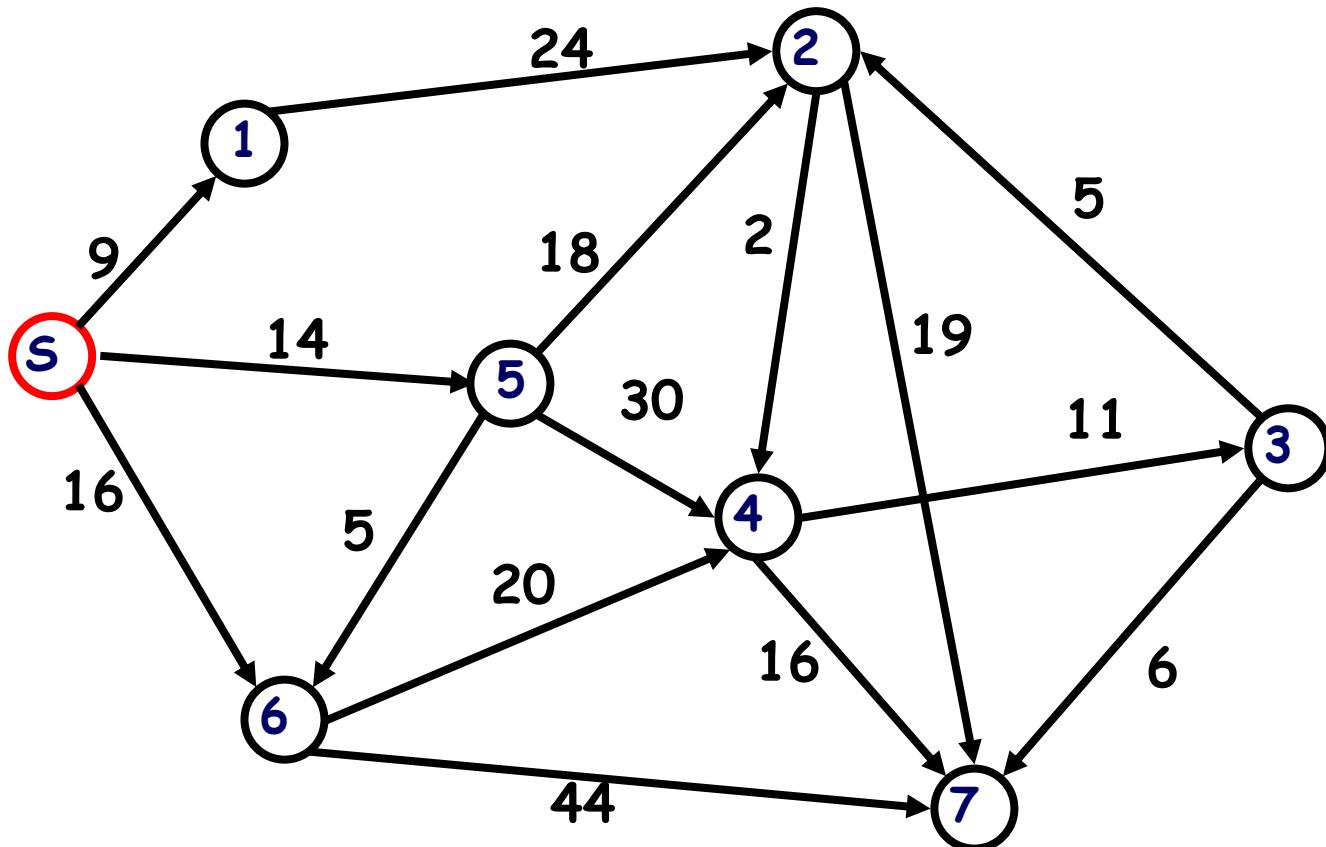
Given a positively weighted graph  $G$  with a source vertex  $s$ , find the shortest path **from  $s$  to all other vertices** in the graph.

Single Source Shortest Path



# The Shortest Path Problem

What is the shortest distance from s to 4?



# Greedy Approach

When algorithm proceeds all vertices are divided into two groups

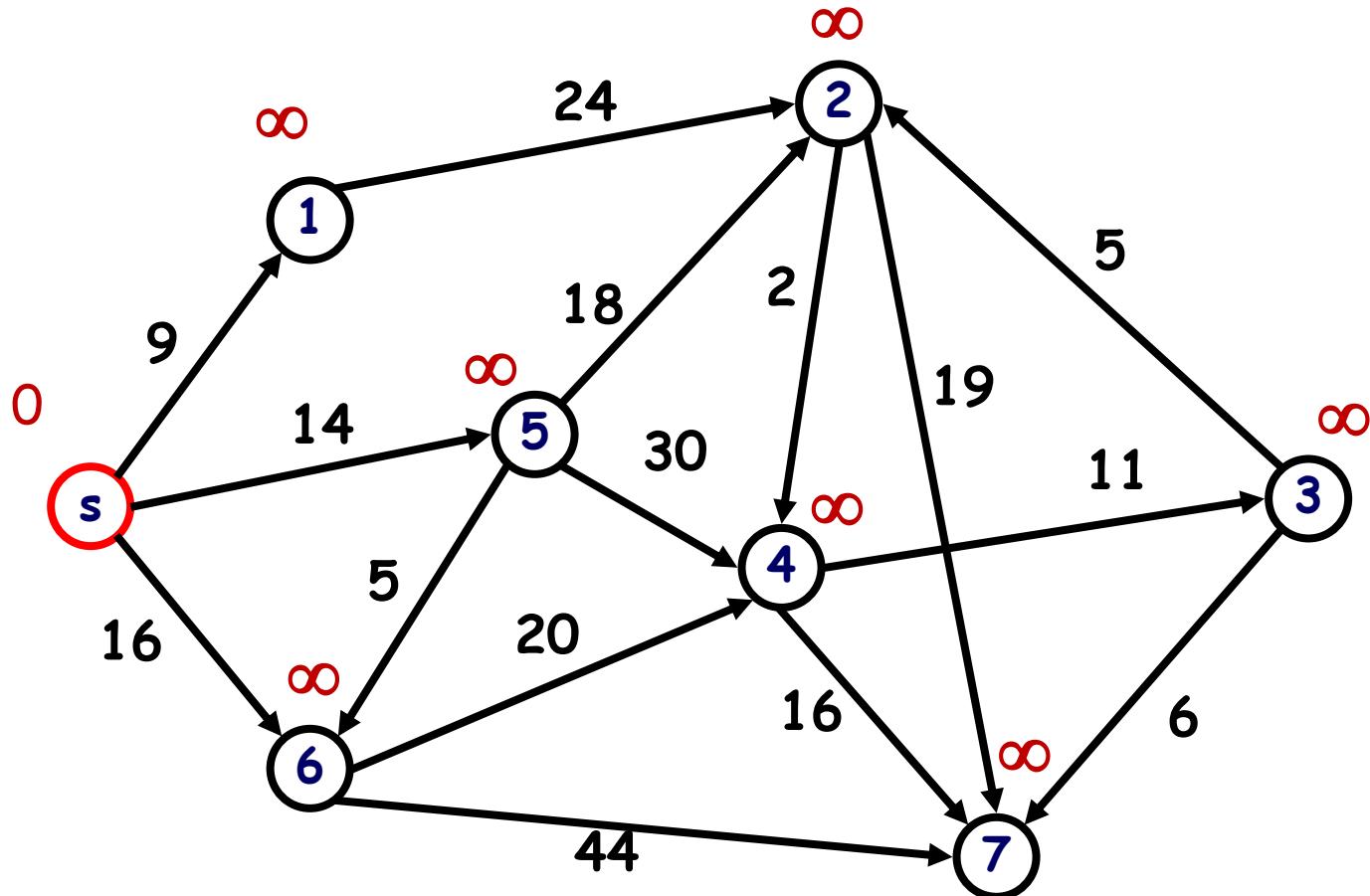
- vertices whose shortest path from the source is known
- vertices whose shortest path from the source is NOT

discovered yet.

Move vertices one at a time from the undiscovered set of vertices to the known set of the shortest distances, based on the shortest distance from the source.

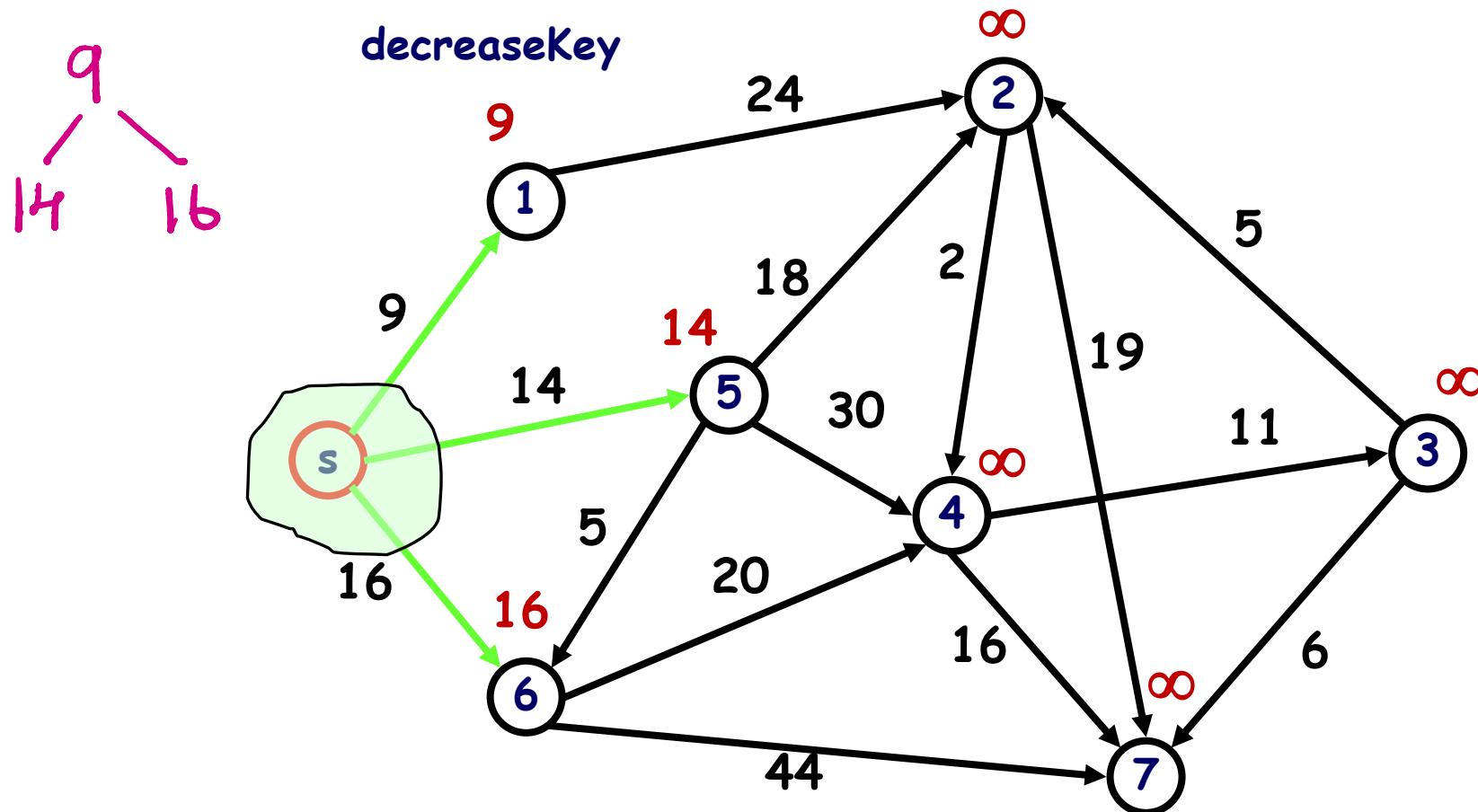
solution tree = s

heap = {1, 2, 3, 4, 5, 6, 7}



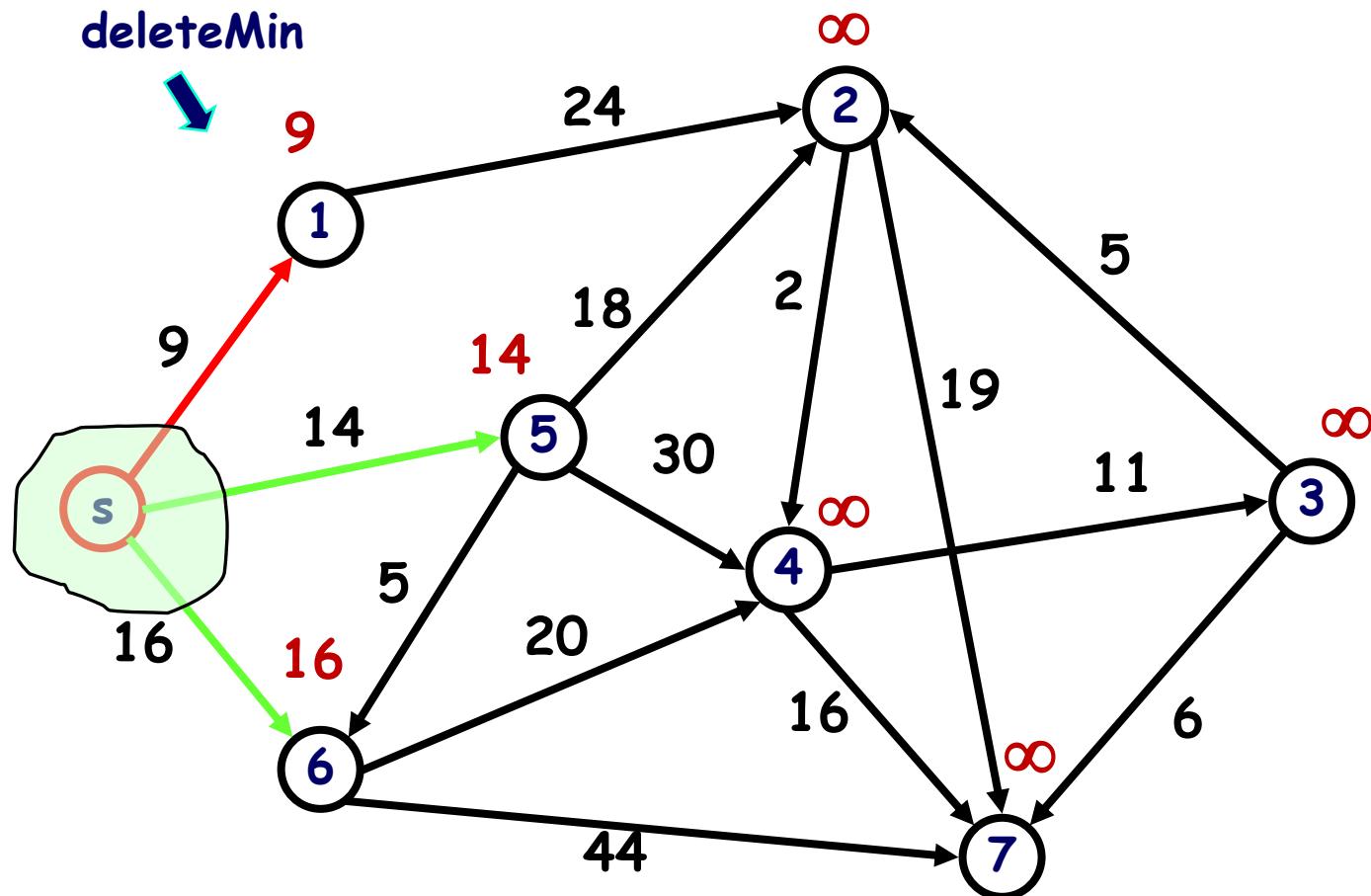
solution tree = { s }

heap = {1, 2, 3, 4, 5, 6, 7}

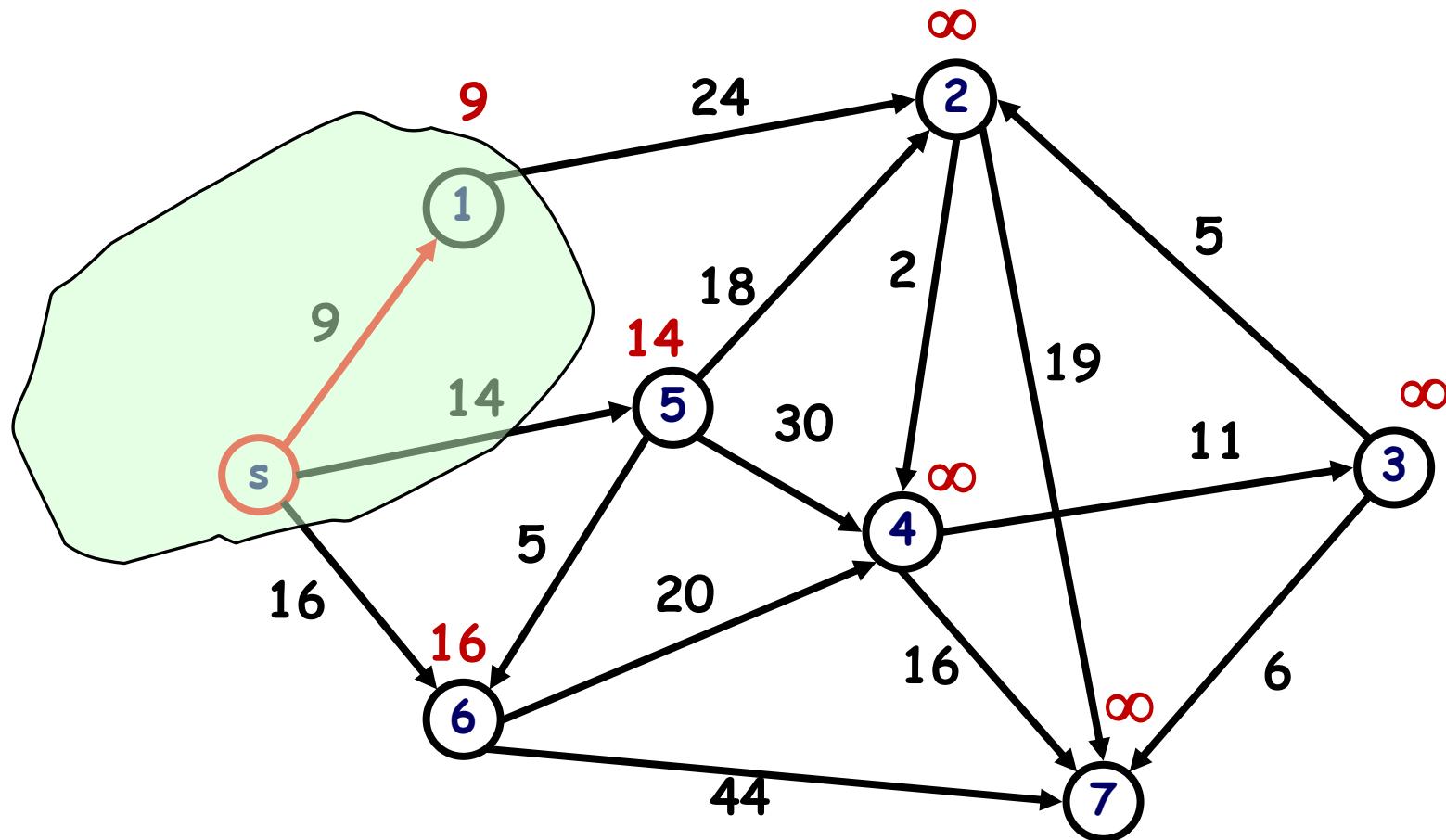


solution tree = { s }

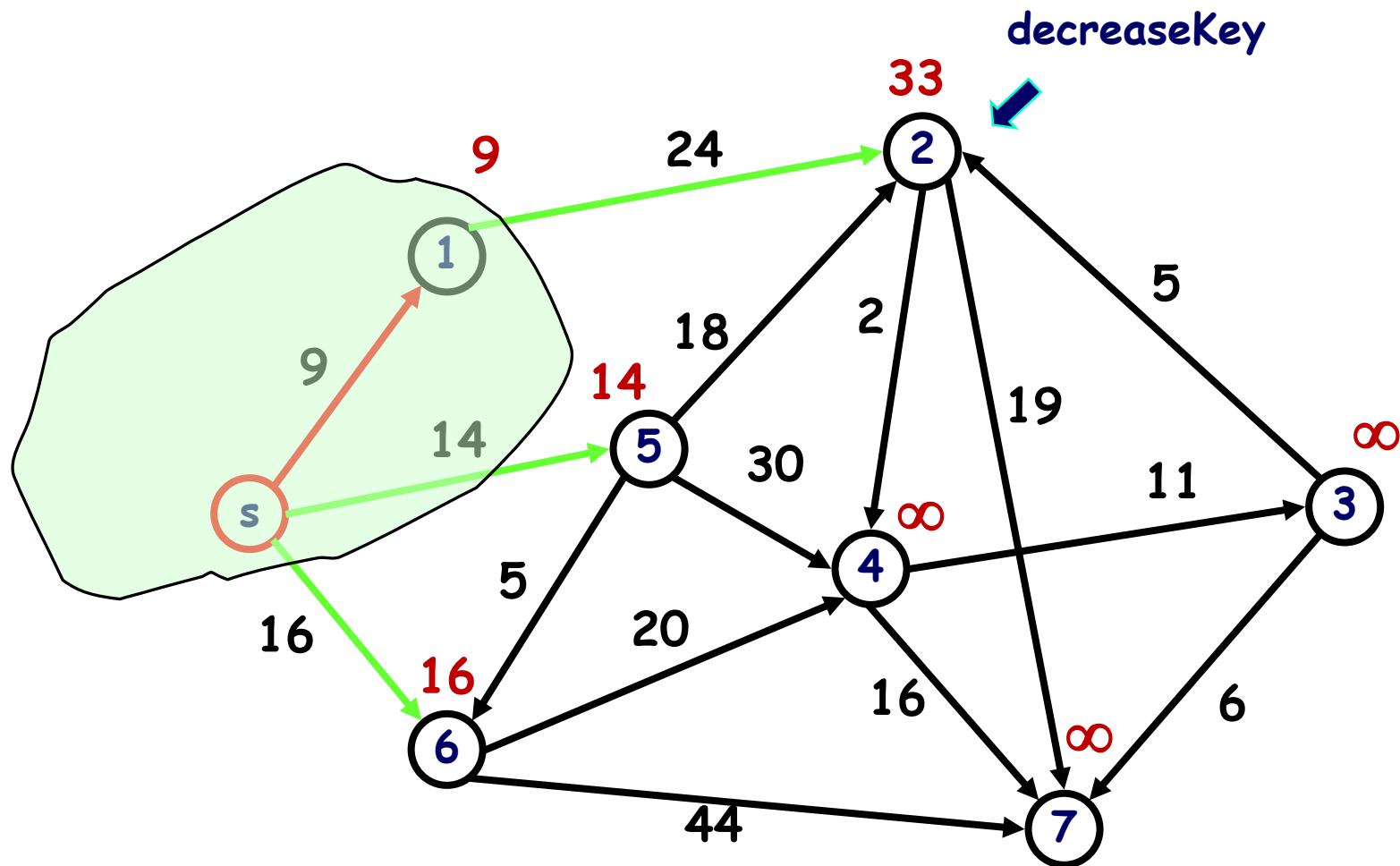
heap = {1, 2, 3, 4, 5, 6, 7}



solution tree = { s, 1 }  
heap = {2, 3, 4, 5, 6, 7}

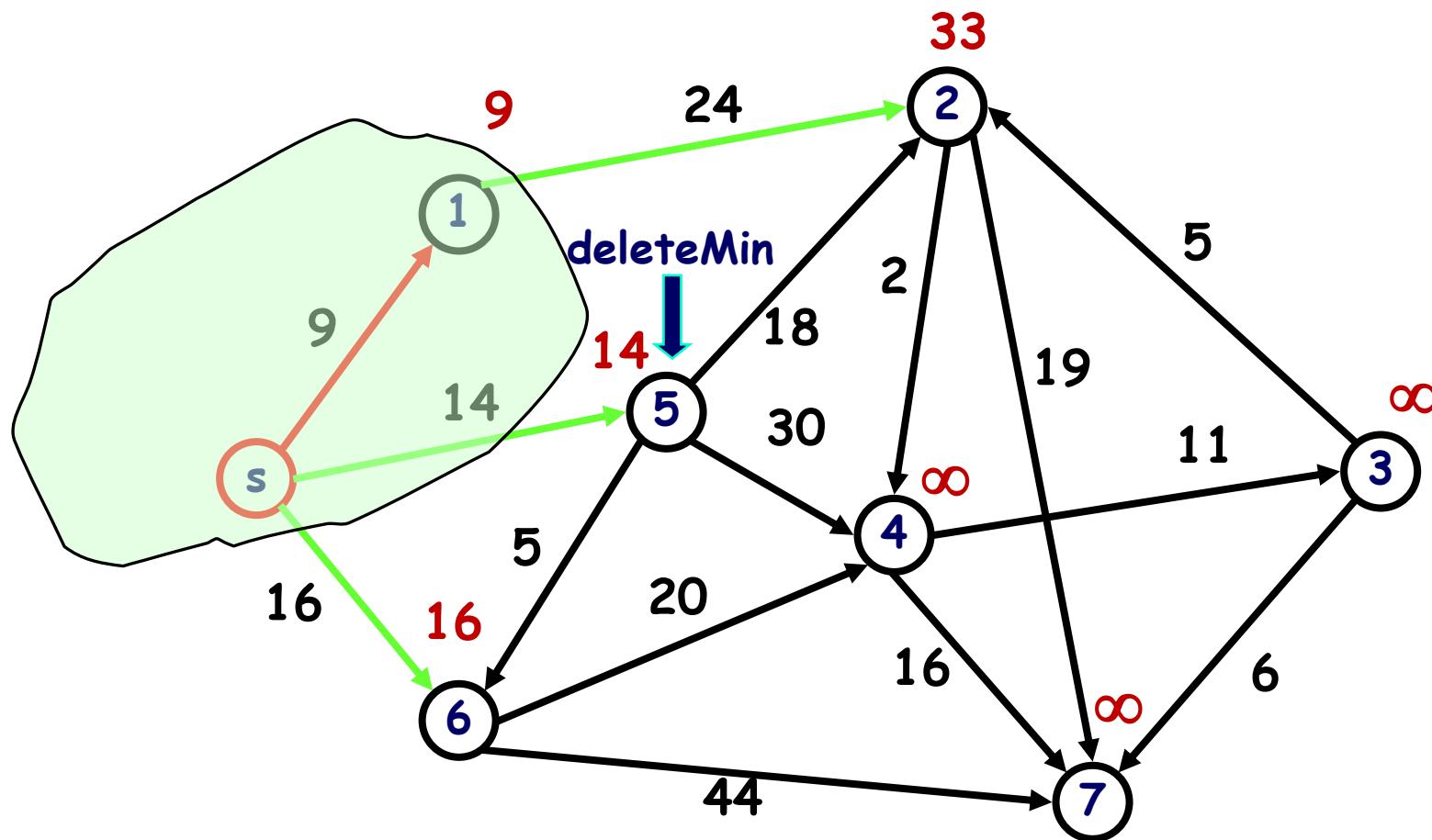


solution tree = { s, 1 }  
heap = {2, 3, 4, 5, 6, 7}



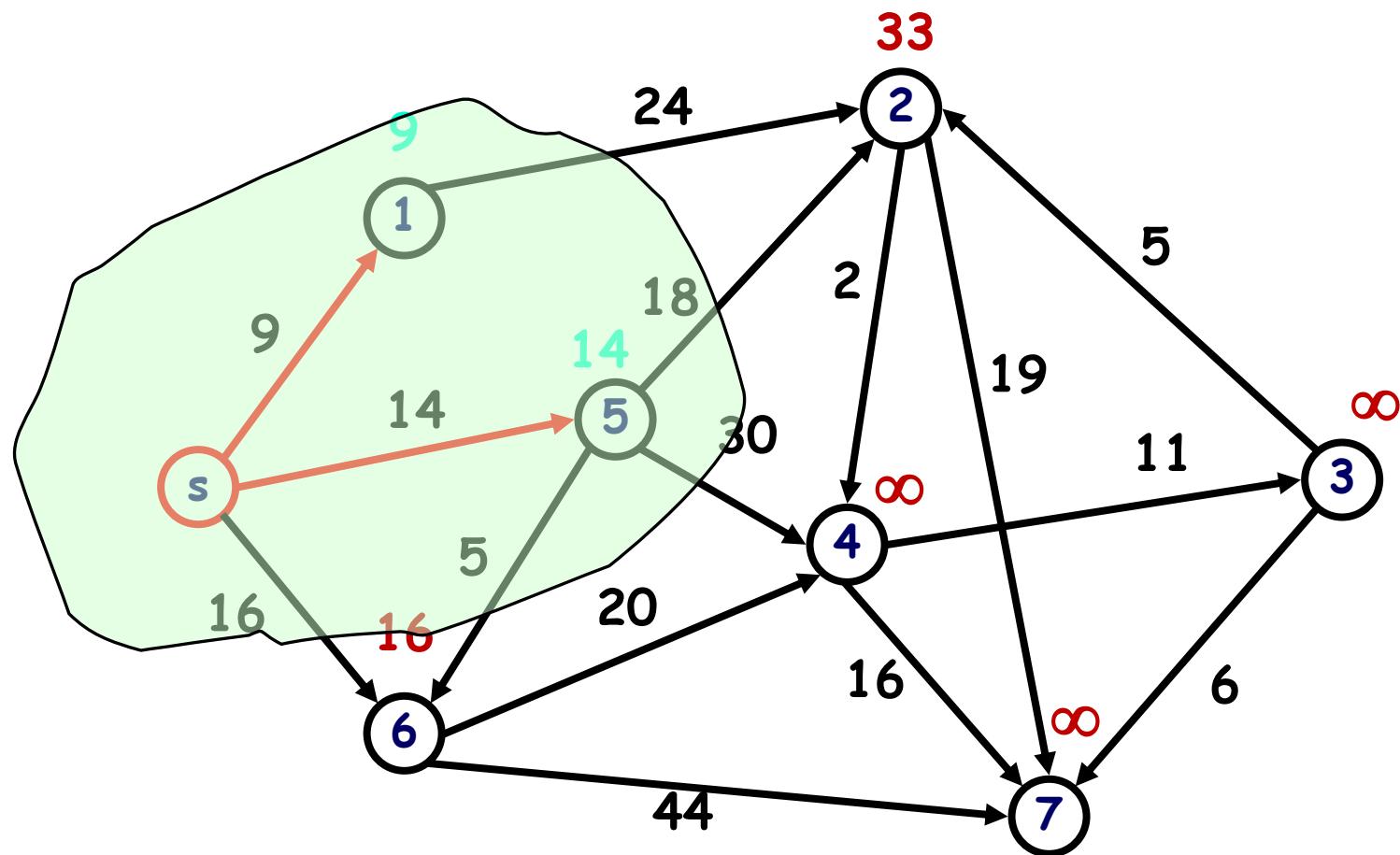
solution tree = { s, 1 }

heap = {2, 3, 4, 5, 6, 7}



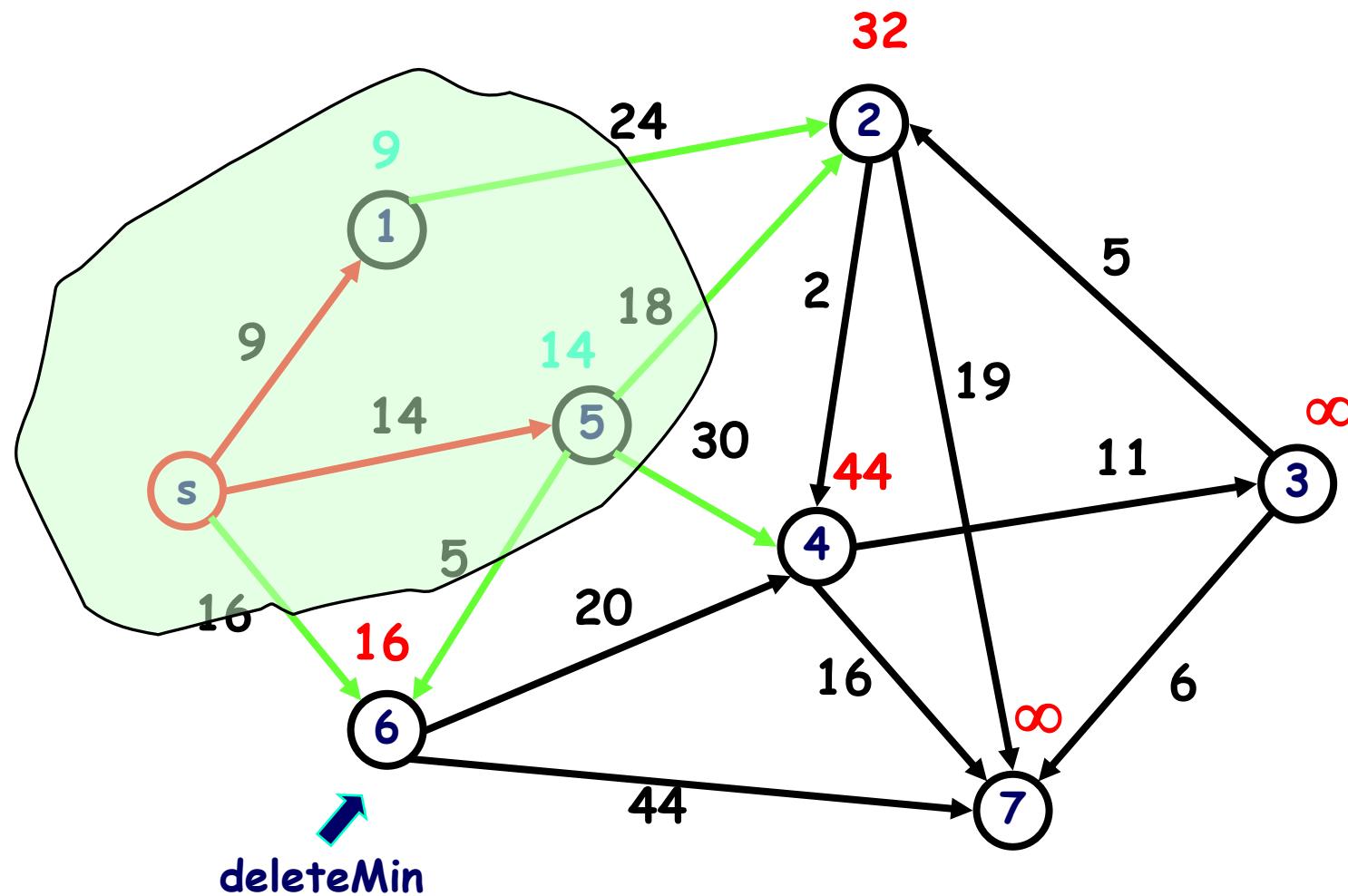
solution tree = { s, 1, 5 }

heap = {2, 3, 4, 6, 7}

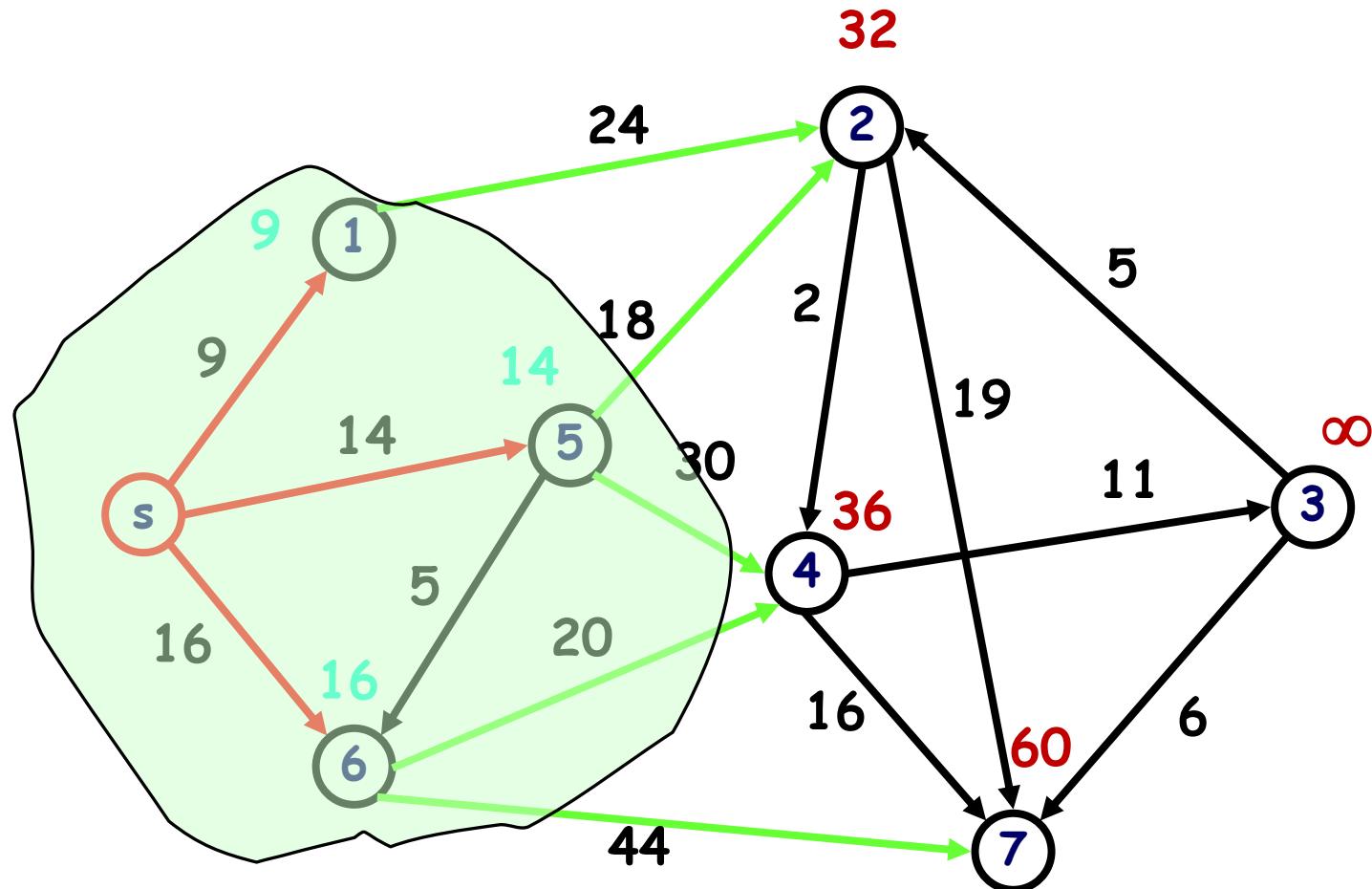


solution tree = { s, 1, 5 }

heap = {2, 3, 4, 6, 7}



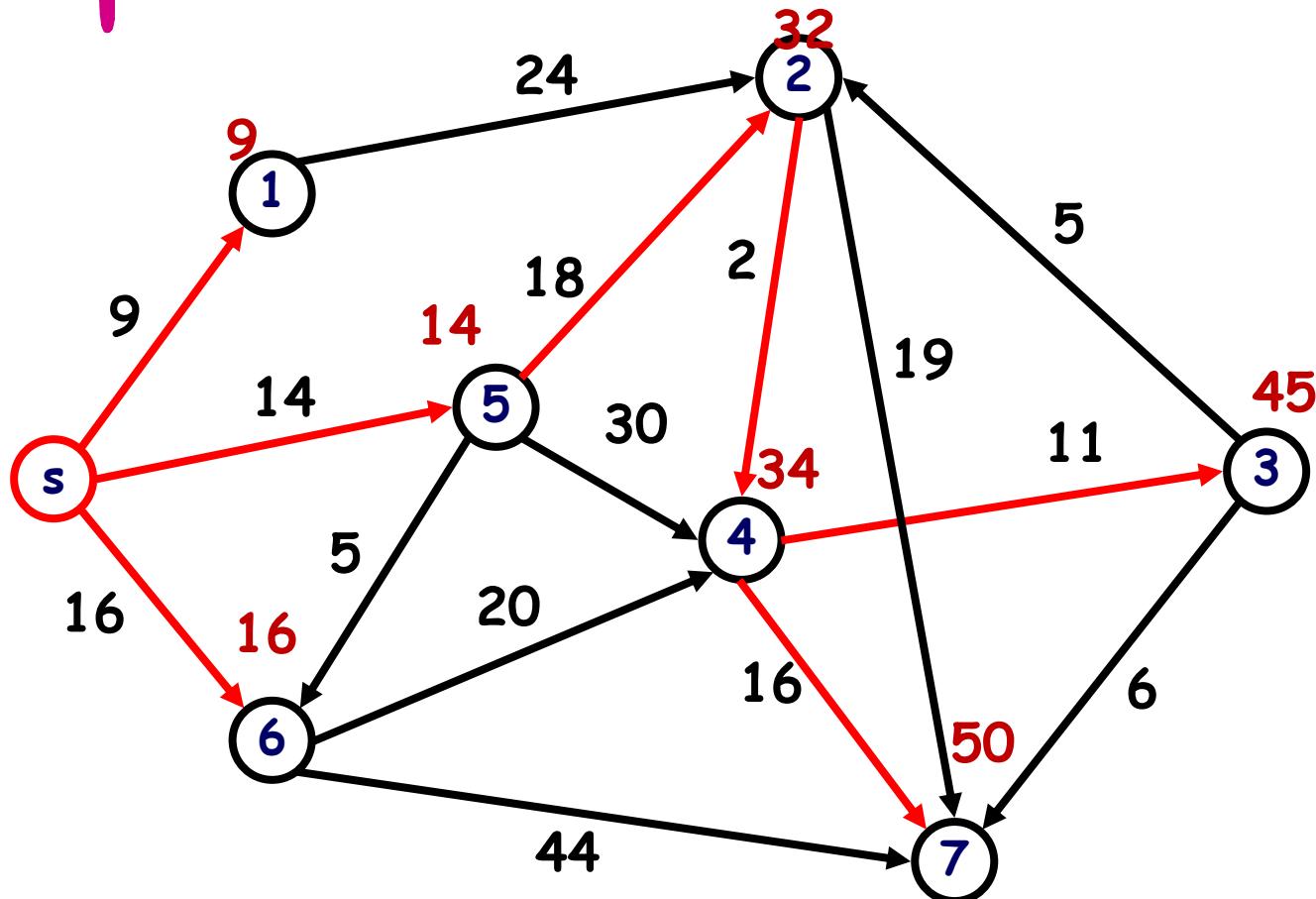
solution tree = { s, 1, 5, 6 }  
heap = {2, 3, 4, 7}



solution tree = { s, 1, 5, 6, 2, 4, 3, 7 }

heap = {}

Spanning tree  $\neq$  MST



# Runtime Complexity

Let  $D(v)$  denote a length from the source  $s$  to any vertex  $v$ .  
We store distances  $D(v)$  in a binary heap.

Loop:

- ① `deleteMin`,  $V$  times
- ② `decreaseKey` (*update distances*) ,  
 $E$  times

$$O(V \cdot \log V, E \cdot \log V)$$

## Discussion Problem 5

Assume that an **unsorted array** is used instead of a binary heap.  
What would the running time of the Dijkstra algorithm?

$$\underbrace{O(V, V + E \cdot I)}$$

SEE THE LECTURE4 ANNOTATED  
PDF. SOME SLIDES ARE MISSING HERE