Announcements

- Homework 2 online due Friday

Last Time

- Shortest Paths in Graphs with Edge Lengths
- Priority Queues

Edge Lengths

The number of edges in a path is not always the right measure of distance. Sometimes, taking several shorter steps is preferable to taking a few longer ones.

We assign each edge (u,v) a non-negative <u>length</u> $\ell(u,v)$. The length of a path is the sum of the lengths of its edges.

Problem: Shortest Paths

Problem: Given a Graph G with vertices s and t and a length function ℓ, find the shortest path from s to t.

Algorithm

```
Distances(G,s,ℓ)
  dist(s) ← 0
  While(not all distances found)
    Find minimum over (v,w) ∈ E
      with v discovered w not
      of dist(v)+ℓ(v,w)
    dist(w) ← dist(v)+ℓ(v,w)
    prev(w) ← v
```

Still too Slow

- Repeatedly ask for smallest vertex
  - Even though not much is changing from round to round, the algorithm is computing the minimum from scratch every time
- Use a data structure!
  - Data structures help answer a bunch of similar questions faster than answering each question individually
- For this kind of question, want a priority queue.

Priority Queue

A <u>Priority Queue</u> is a datastructure that stores elements sorted by a <u>key</u> value.

<u>Operations:</u>

- Insert – adds a new element to the PQ.
- DecreaseKey – Changes the key of an element of the PQ to a specified *smaller* value.
- DeleteMin – Finds the element with the smallest key and removes it from the PQ.

Today

- Priority Queues
- Dijkstra's Algorithm
- Negative Edge Weights

Attempt III

```
Distances(G,s,ℓ)

  For v ∈ V
    dist(v) ← ∞, done(v) ← false

  dist(s) ← 0
  While(not all vertices done)
    Find v not done with minimum dist(v)
    done(v) ← true
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
```

Attempt III

```
Distances(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, done(v) ← false

  dist(s) ← 0
  While(not all vertices done)
    Find v not done with minimum dist(v)
    done(v) ← true
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
```

Attempt III

```
Distances(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, done(v) ← false
    Q.Insert(v) //using dist(v) as key
  dist(s) ← 0
  While(not all vertices done)
    Find v not done with minimum dist(v)
    done(v) ← true
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
```

Attempt III

```
Distances(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, ~~done(v) ← false~~
    Q.Insert(v) //using dist(v) as key
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    ~~done(v) ← true~~
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
```

Attempt III

```
Distances(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, ~~done(v) ← false~~
    Q.Insert(v) //using dist(v) as key
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    ~~done(v) ← true~~
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
        Q.DecreaseKey(w)
```

Attempt III

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞, ~~done(v) ← false~~
    Q.Insert(v) //using dist(v) as key
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    ~~done(v) ← true~~
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        prev(w) ← v
        Q.DecreaseKey(w)
```

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

O(|V|) times

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞                     O(|V|) times
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()               O(|V|) times
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

O(|V|) times

O(|V|) times

O(|E|) times

Runtime

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

O(|V|) times

O(|V|) times

O(|E|) times

Runtime:
O(|V|) Inserts +
O(|V|) DelMins +
O(|E|) DecKeys

What is the Runtime?

So what is the actual runtime then?

What is the Runtime?

So what is the actual runtime then?

This depends on which implementation you use for your priority queue.

What is the Runtime?

So what is the actual runtime then?

This depends on which implementation you use for your priority queue.

We will discuss a few.

Unsorted List

Store n elements in an unsorted list.

___ $\times O(|V|)$

___ $\times O(|E|)$

___ $\times O(|V|)$

Unsorted List

Store n elements in an unsorted list.

Operations:

- Insert - O(1) — $\times O(|V|)$
- DecreaseKey – O(1) — $\times O(|E|)$
- DeleteMin – O(n) — $\times O(|V|)$

Unsorted List

Store n elements in an unsorted list.

Operations:

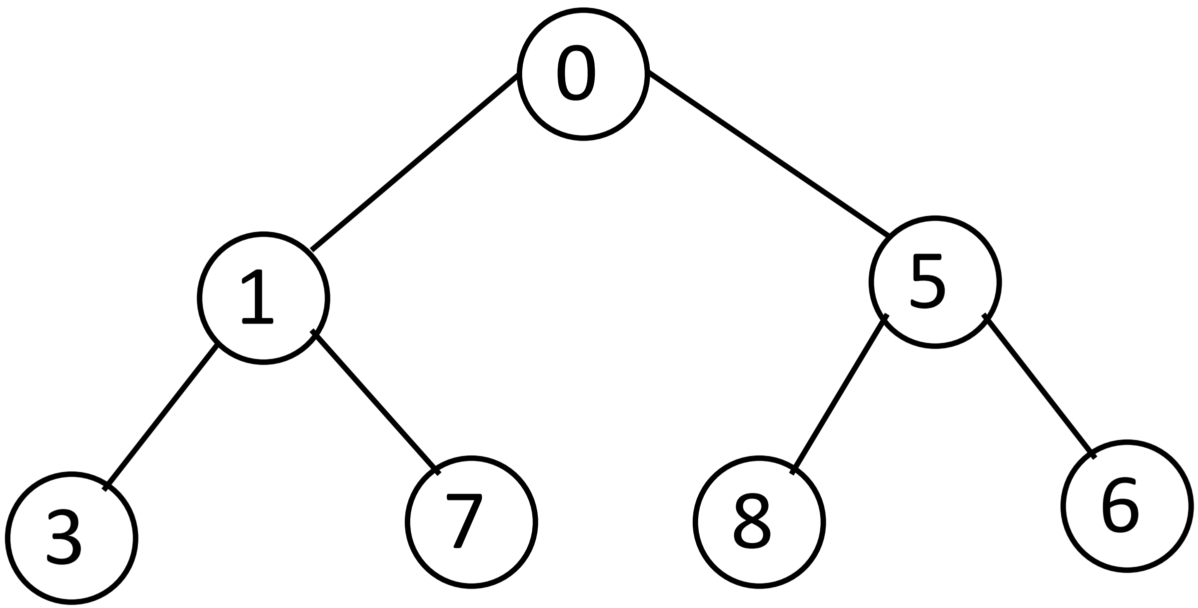
- Insert - O(1) — $\times O(|V|)$
- DecreaseKey – O(1) — $\times O(|E|)$
- DeleteMin – O(n) — $\times O(|V|)$
- Dijkstra – $O(|V|^2+|E|)$

Binary Heap

Store elements in a balanced binary tree with each element having smaller key than its children.

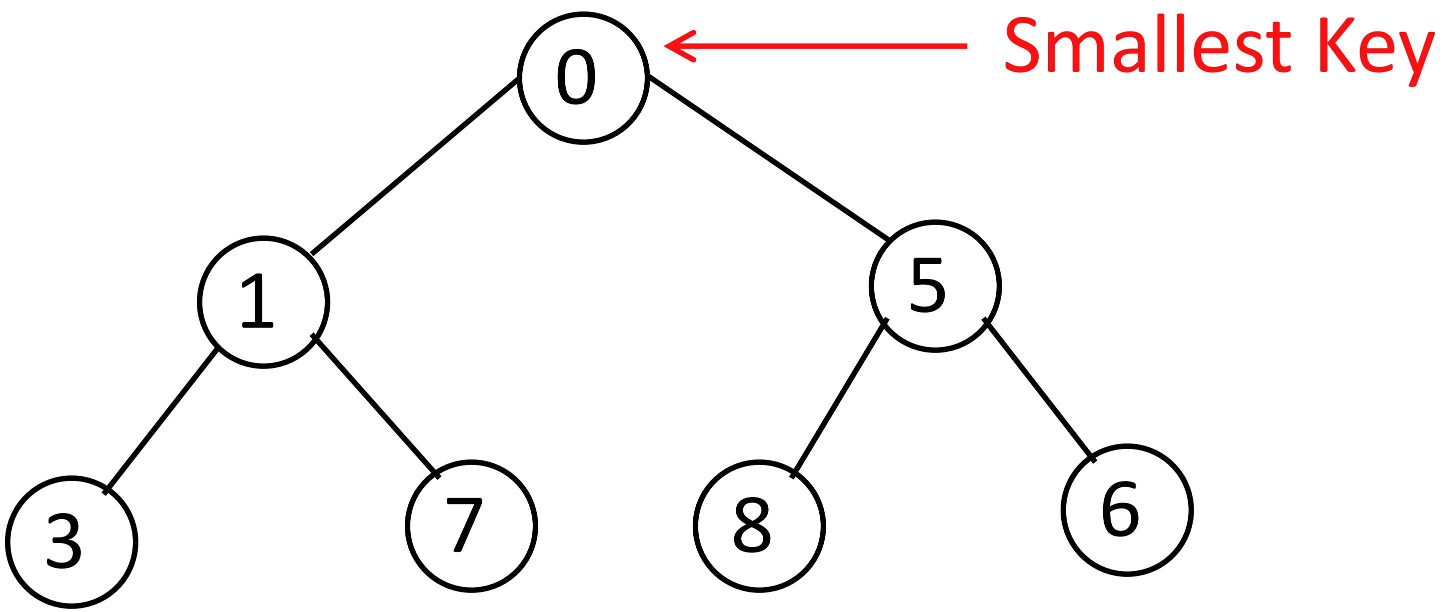
Binary Heap

Store elements in a balanced binary tree with each element having smaller key than its children.



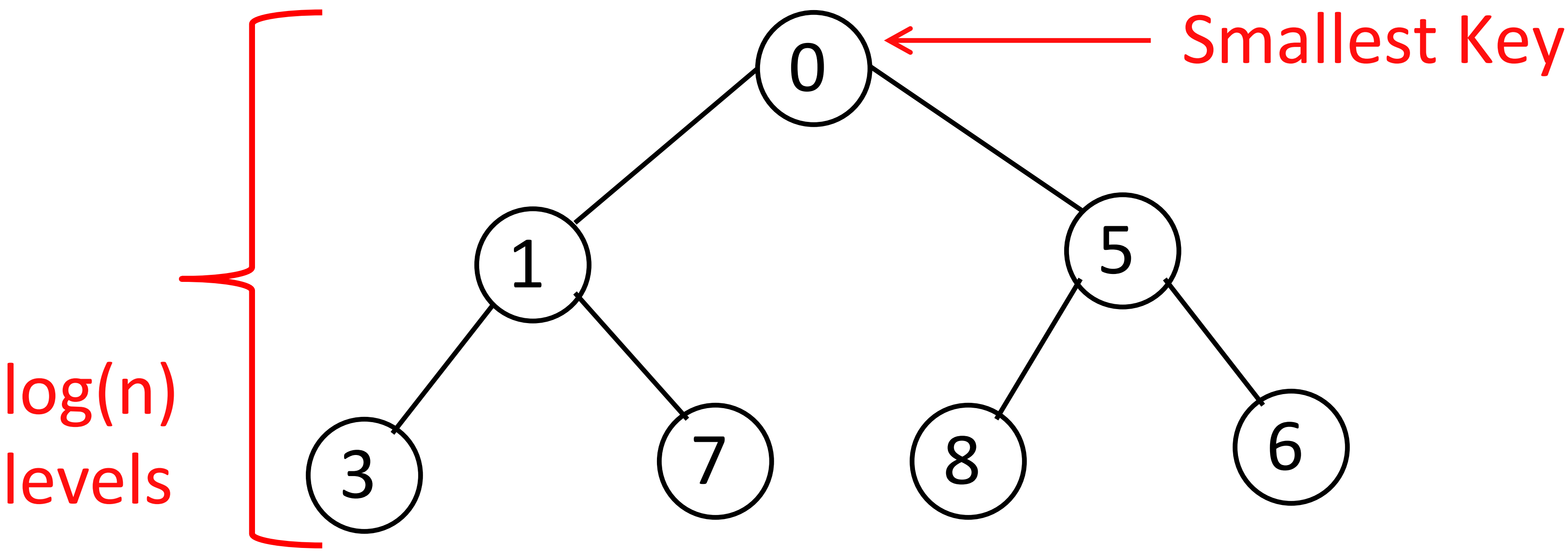
Binary Heap

Store elements in a balanced binary tree with each element having smaller key than its children.

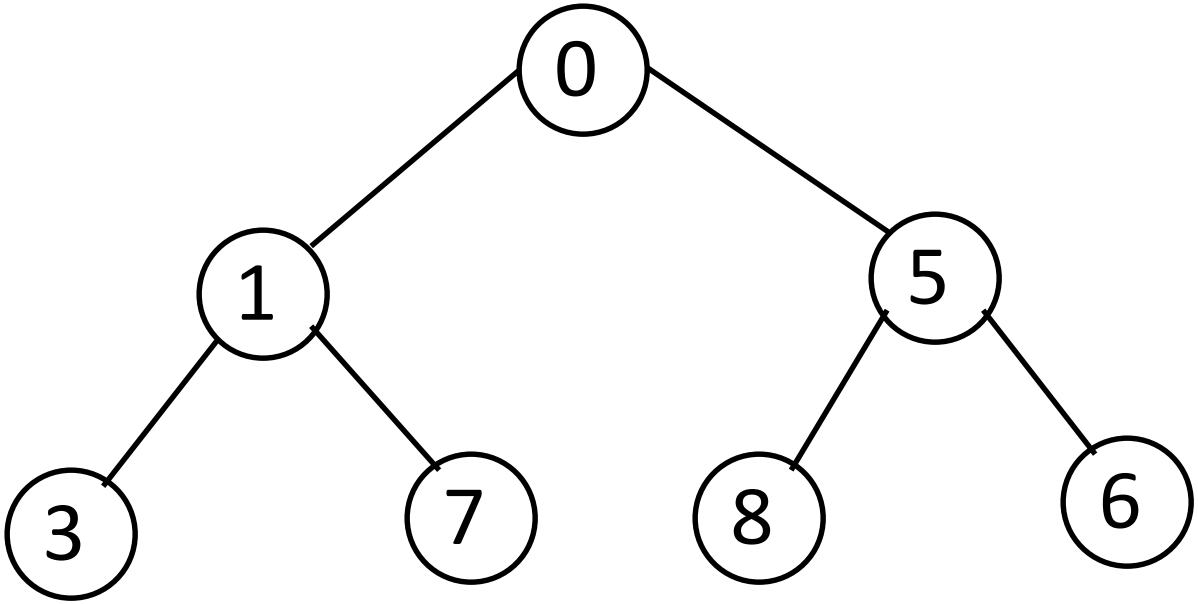


Binary Heap

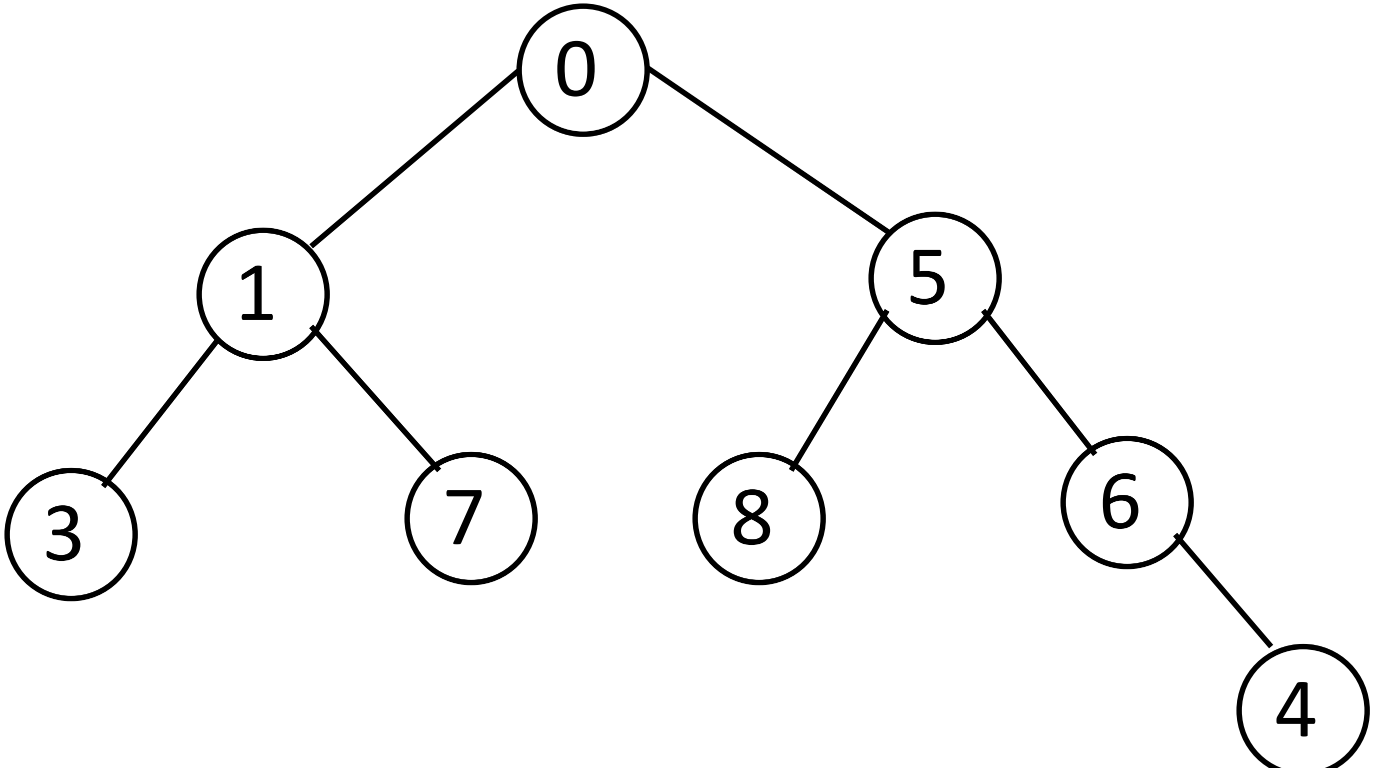
Store elements in a balanced binary tree with each element having smaller key than its children.



Insert

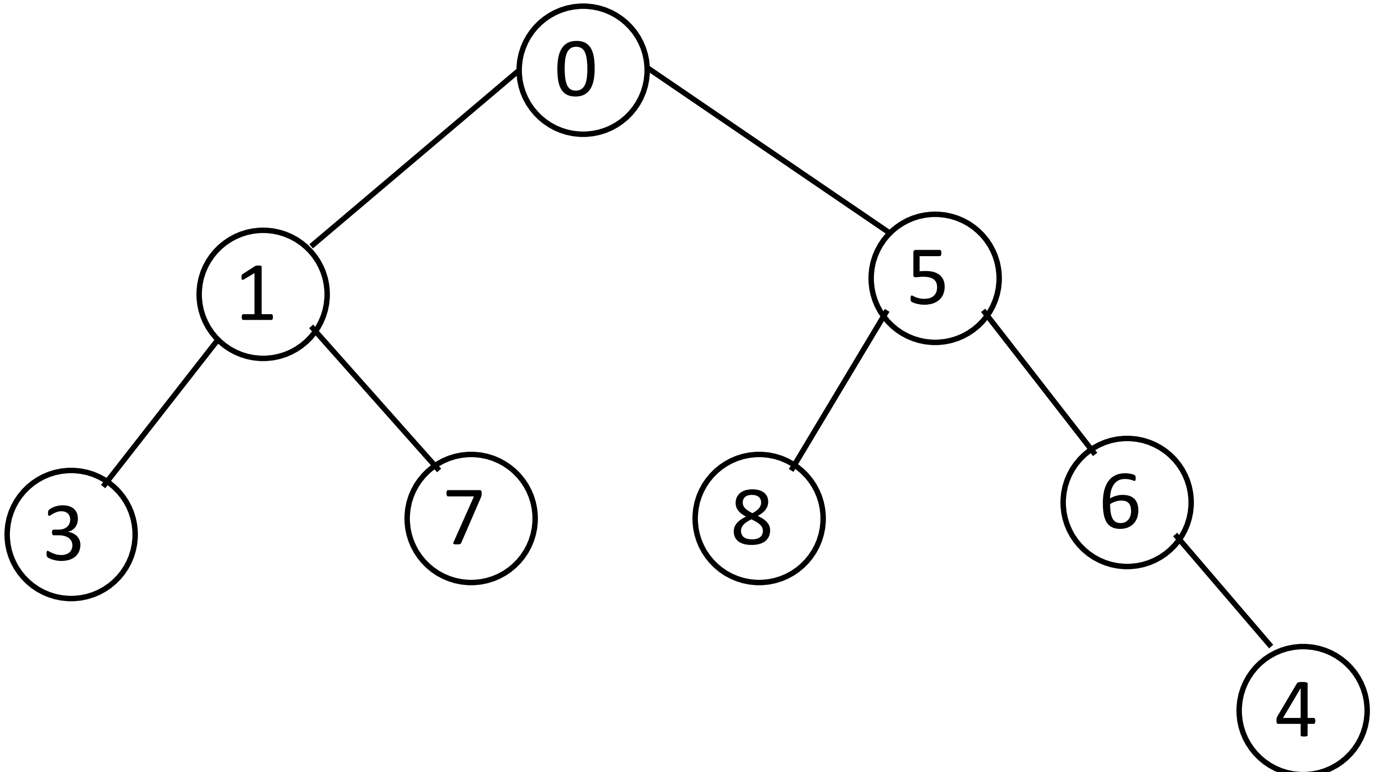


Insert



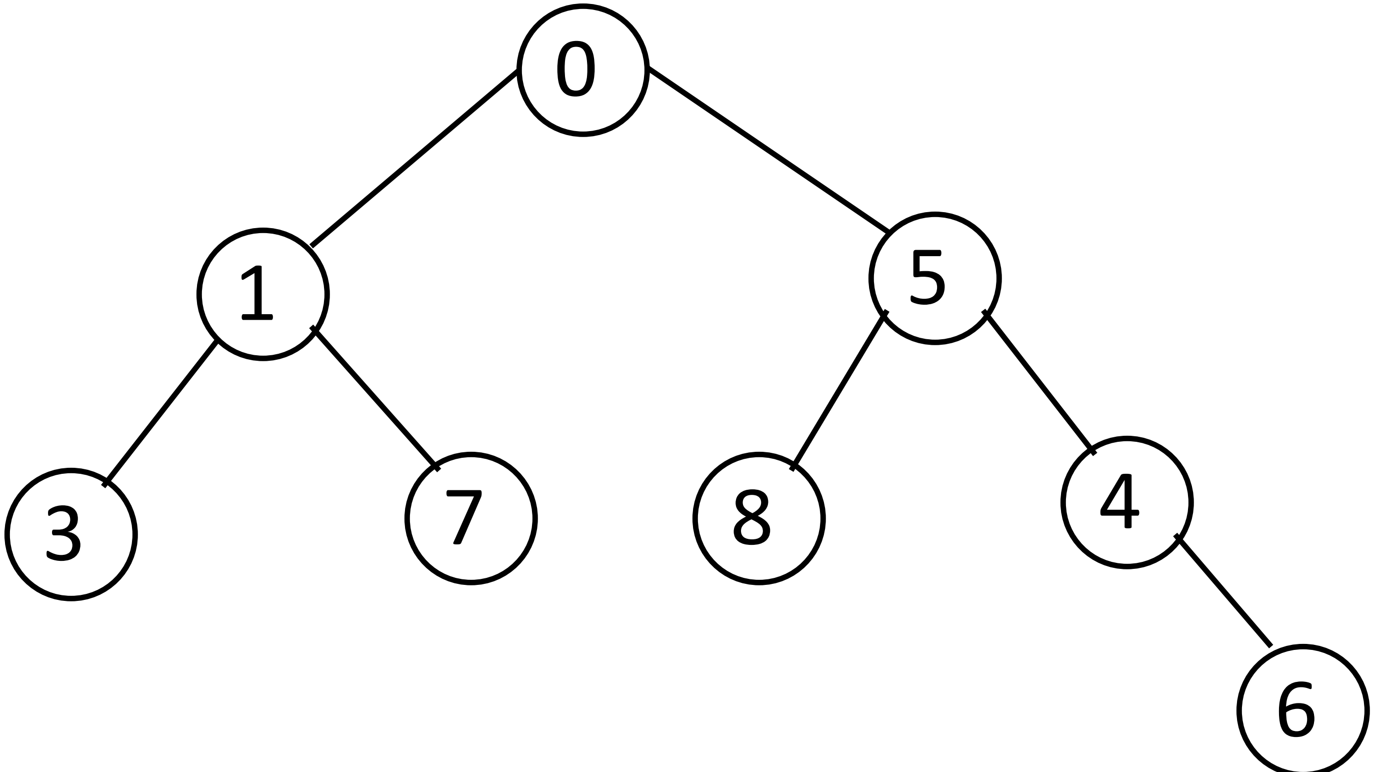
- Add key at bottom

Insert



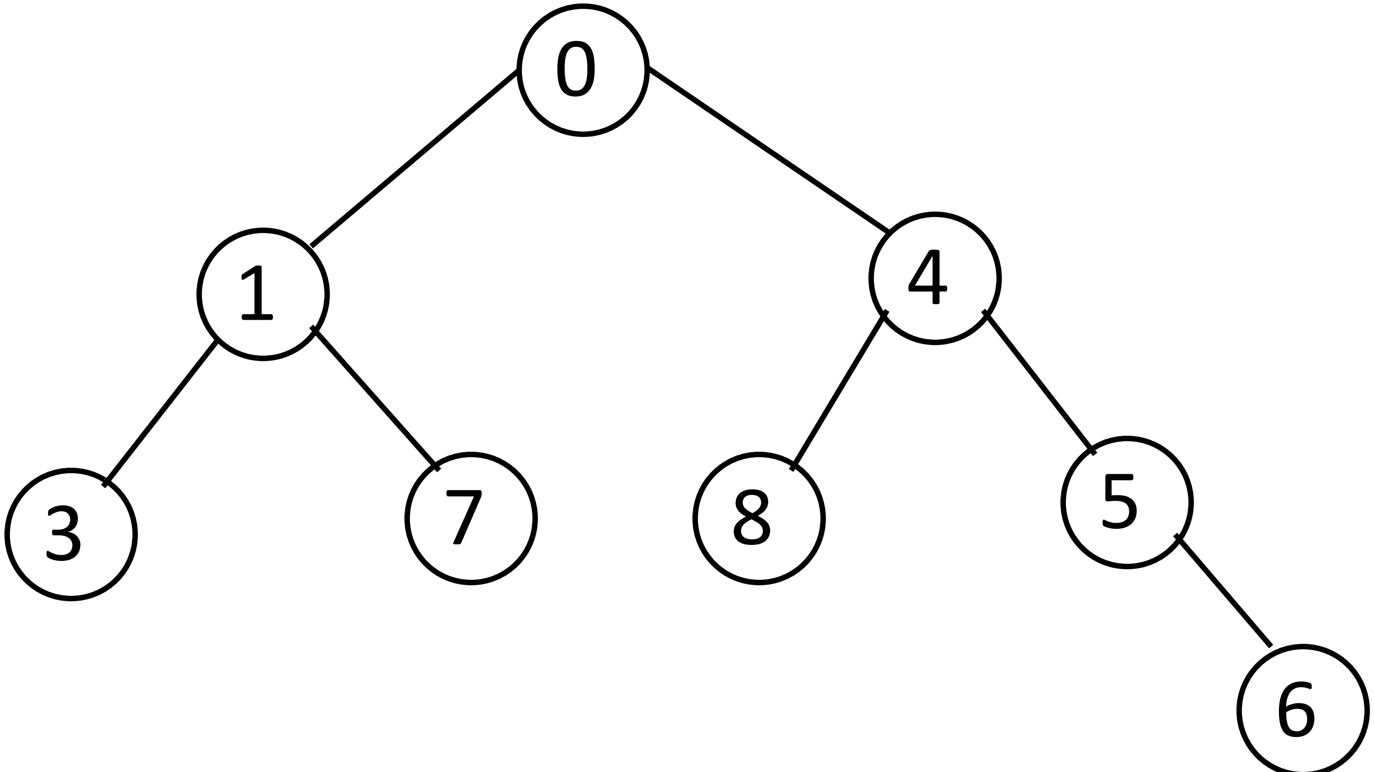
- Add key at bottom
- Bubble up

Insert



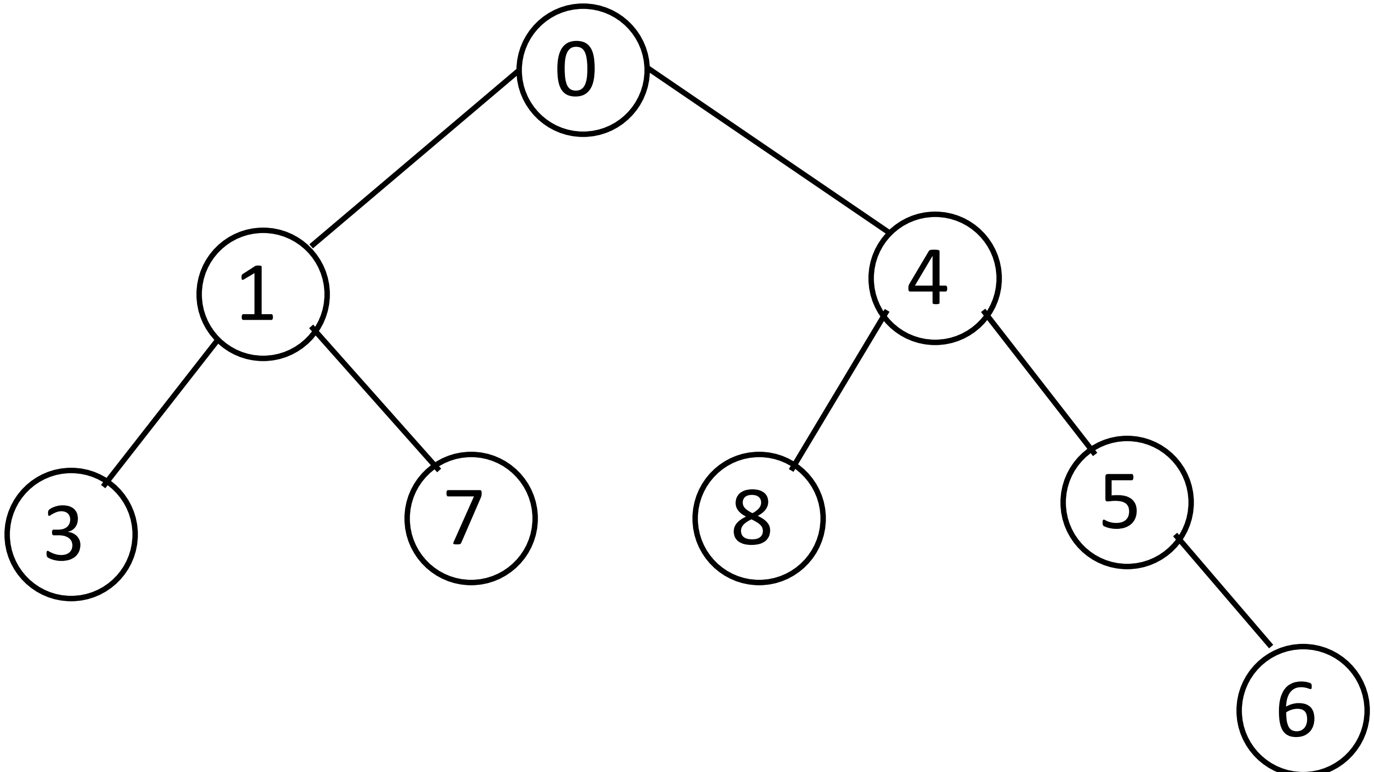
- Add key at bottom
- Bubble up

Insert



- Add key at bottom
- Bubble up

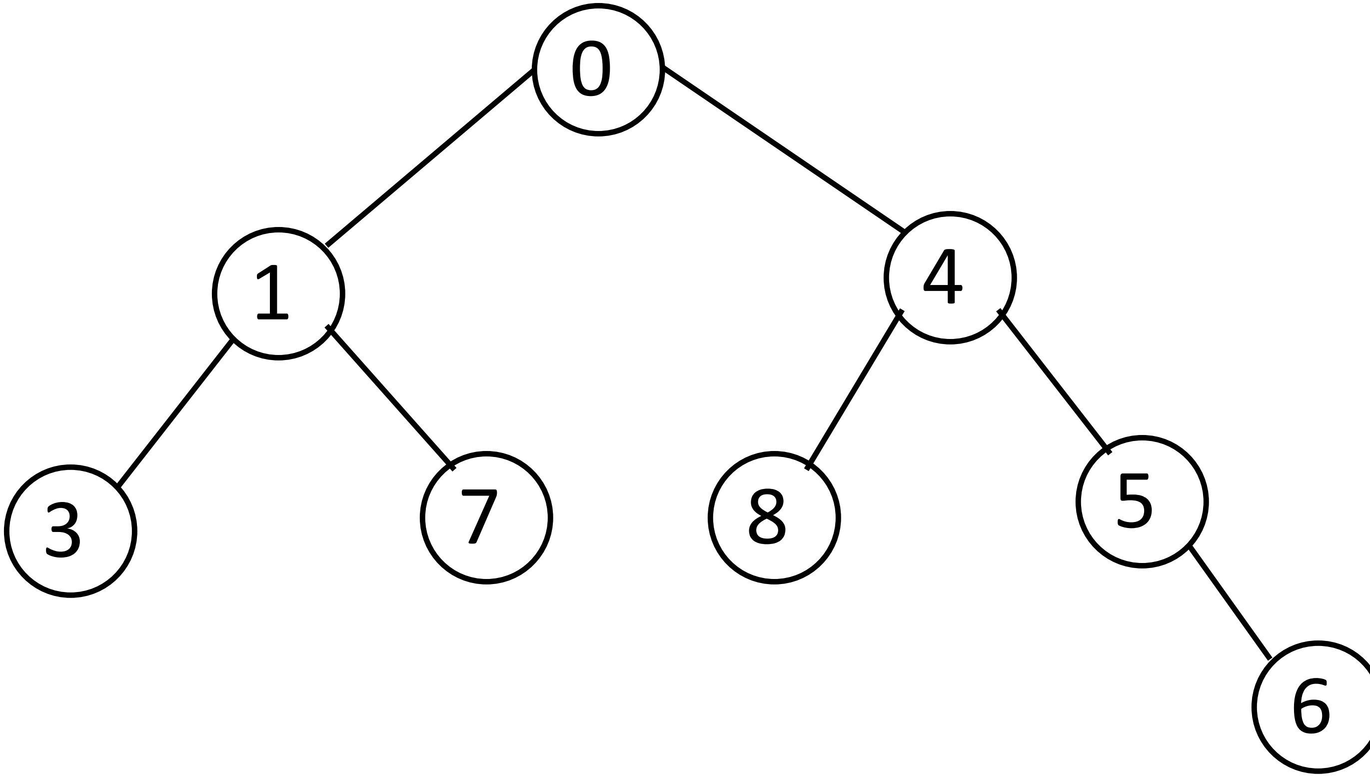
Insert



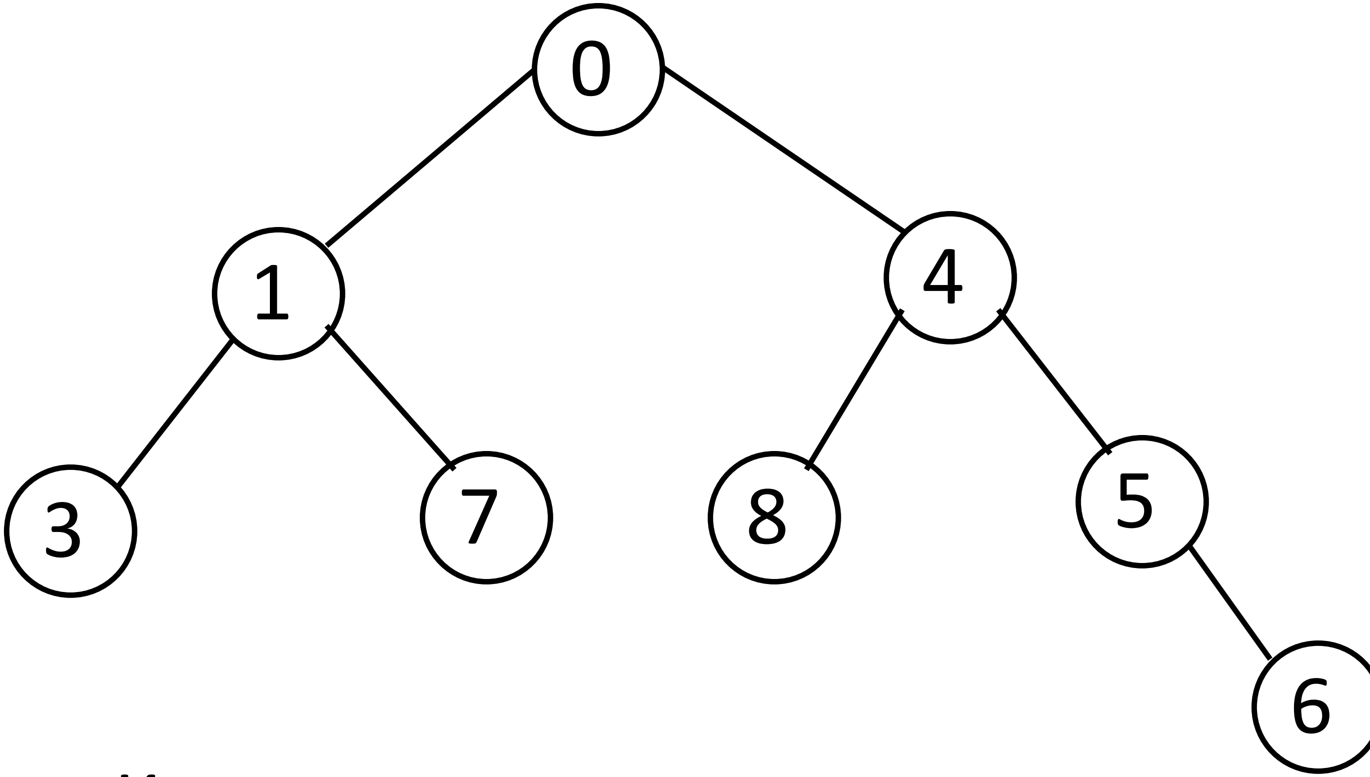
- Add key at bottom
- Bubble up

O(log(n)) time

Decrease Key

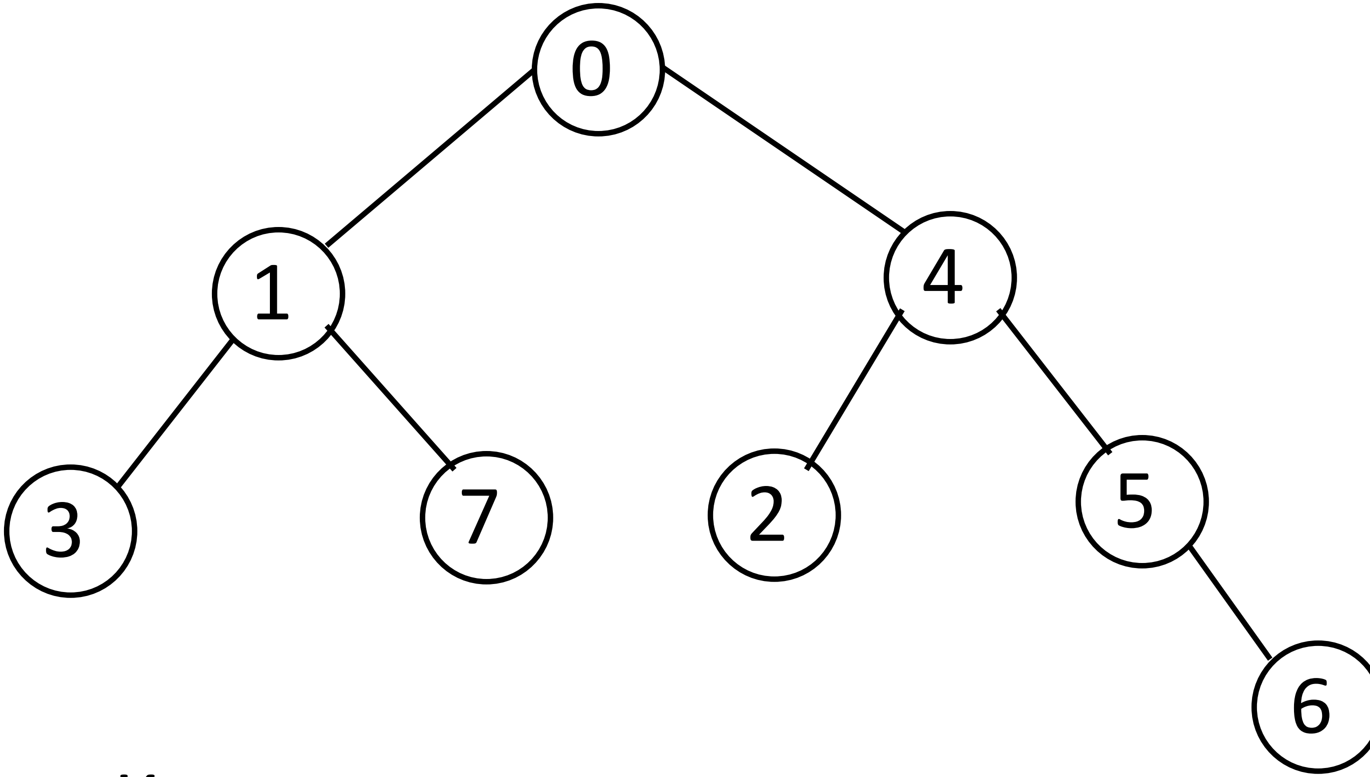


Decrease Key



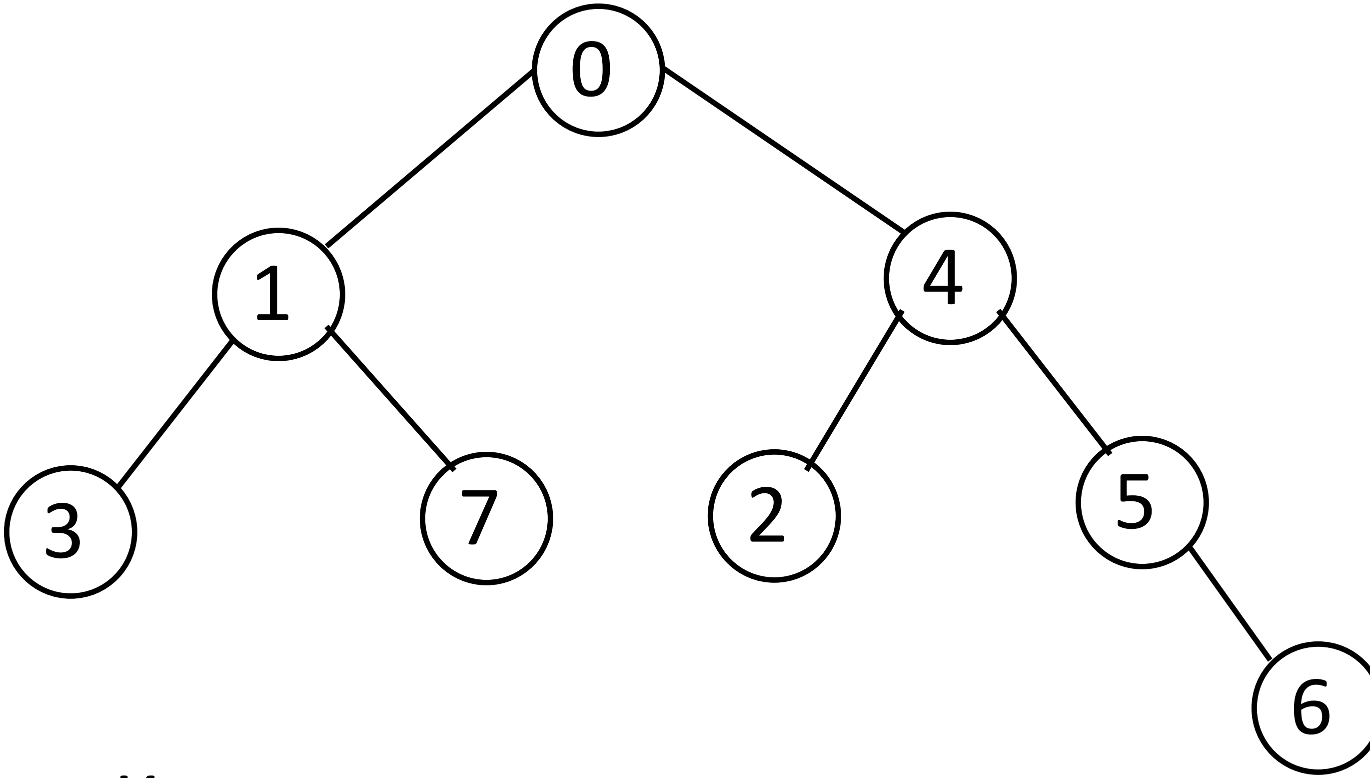
- Change Key

Decrease Key



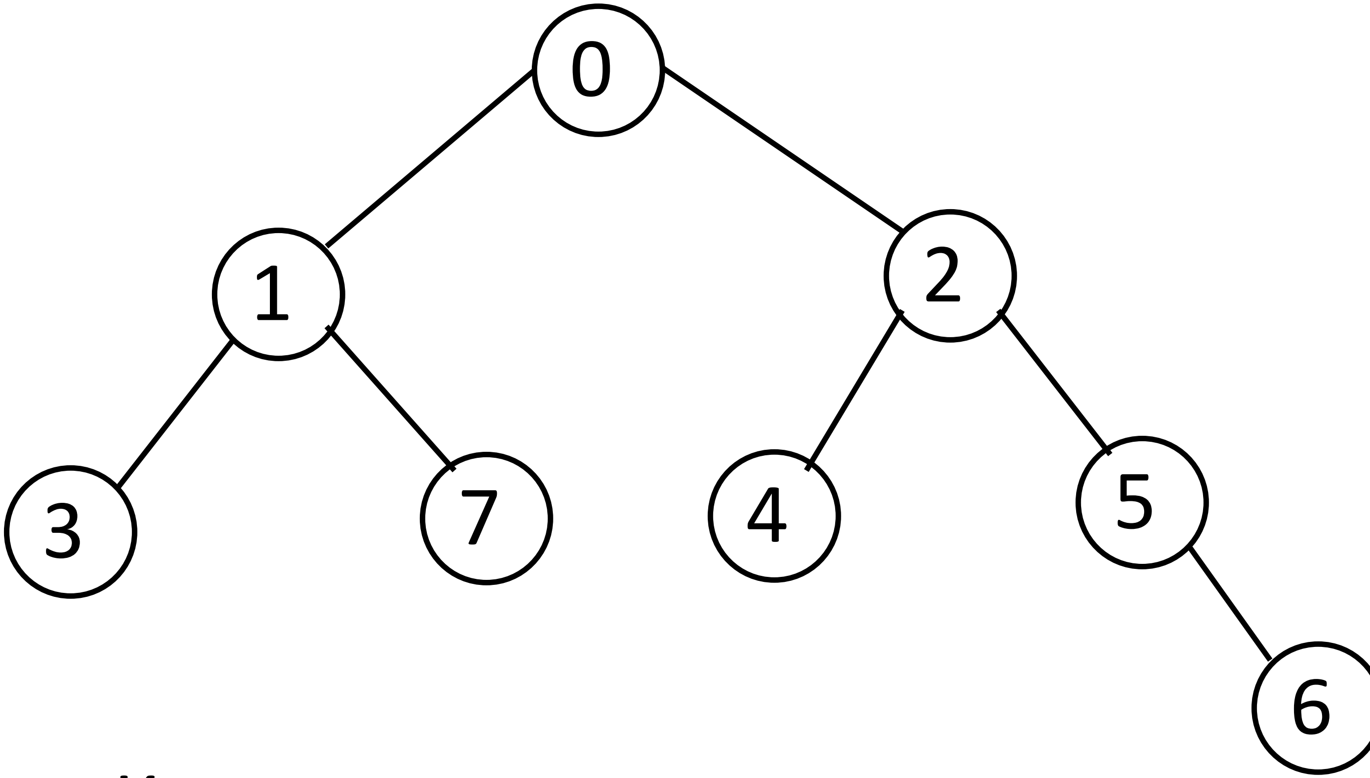
- Change Key

Decrease Key



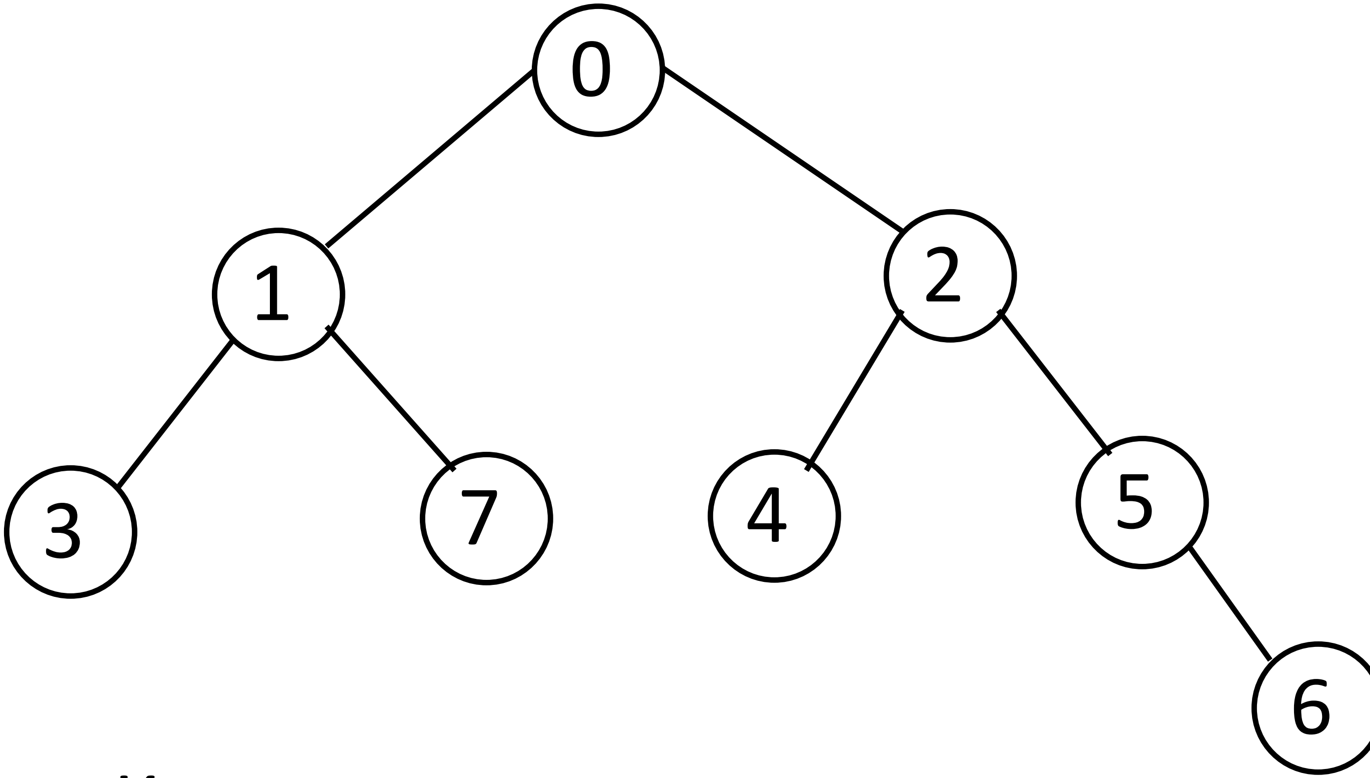
- Change Key
- Bubble up

Decrease Key



- Change Key
- Bubble up

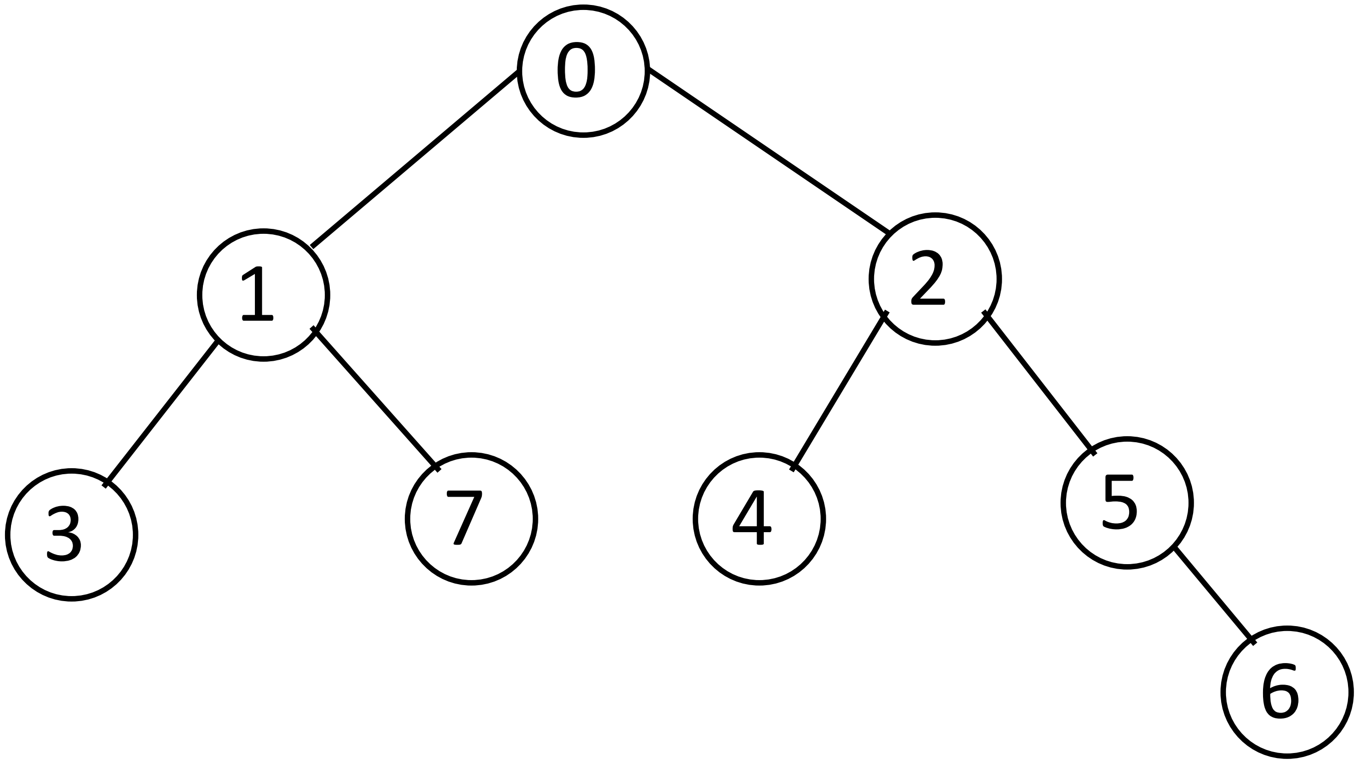
Decrease Key



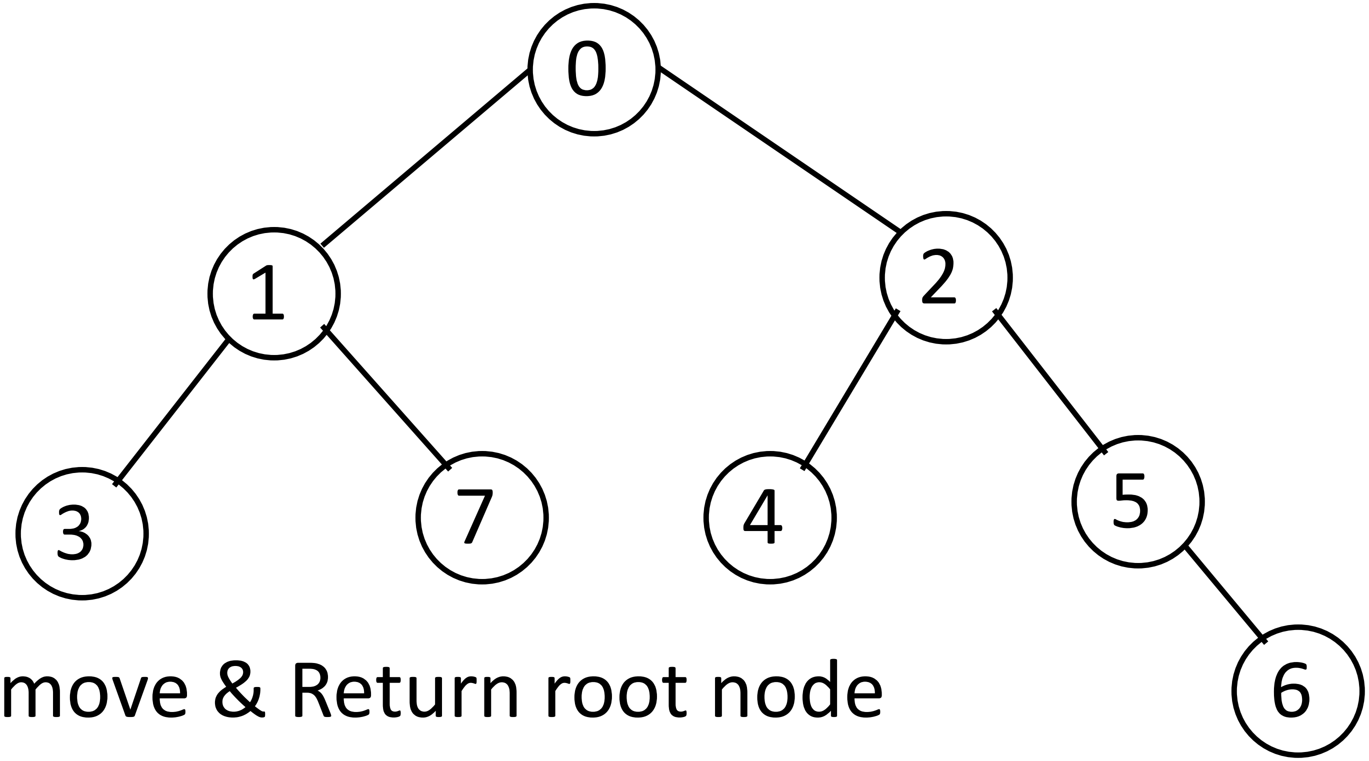
- Change Key
- Bubble up

O(log(n)) time

DeleteMin



DeleteMin

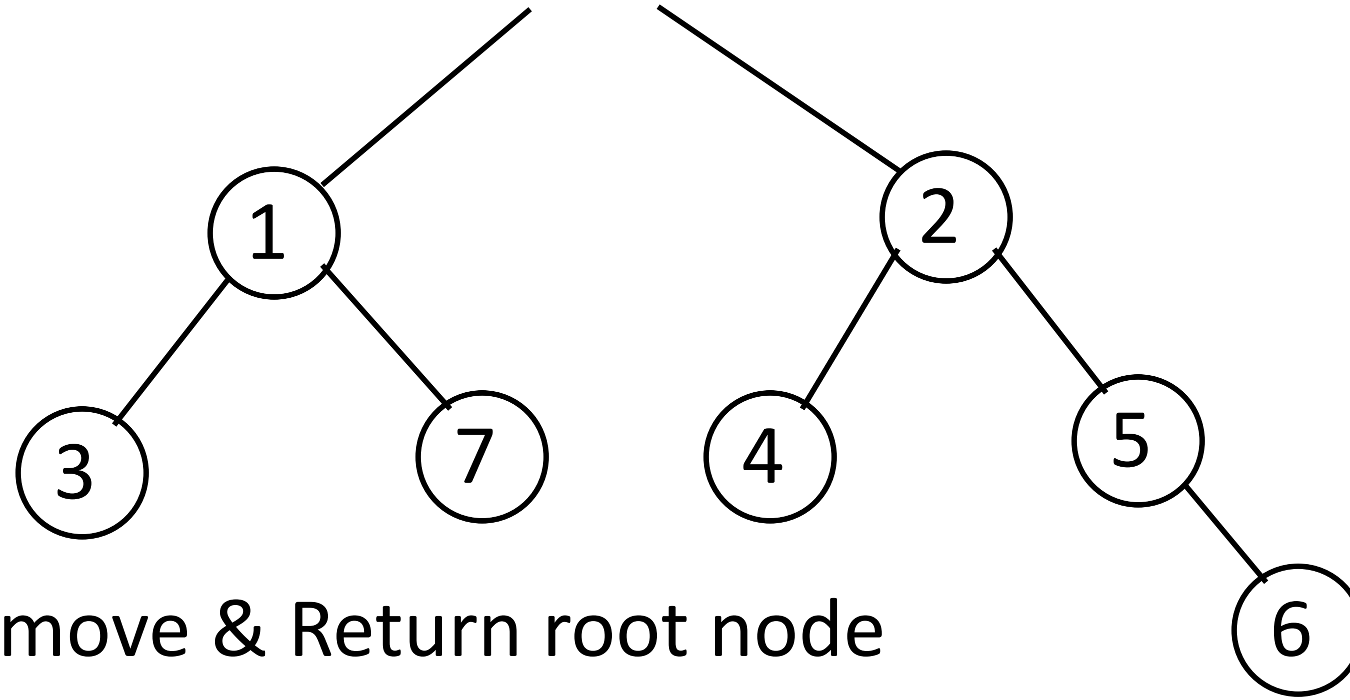


- Remove & Return root node

DeleteMin

Output:

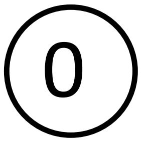
0

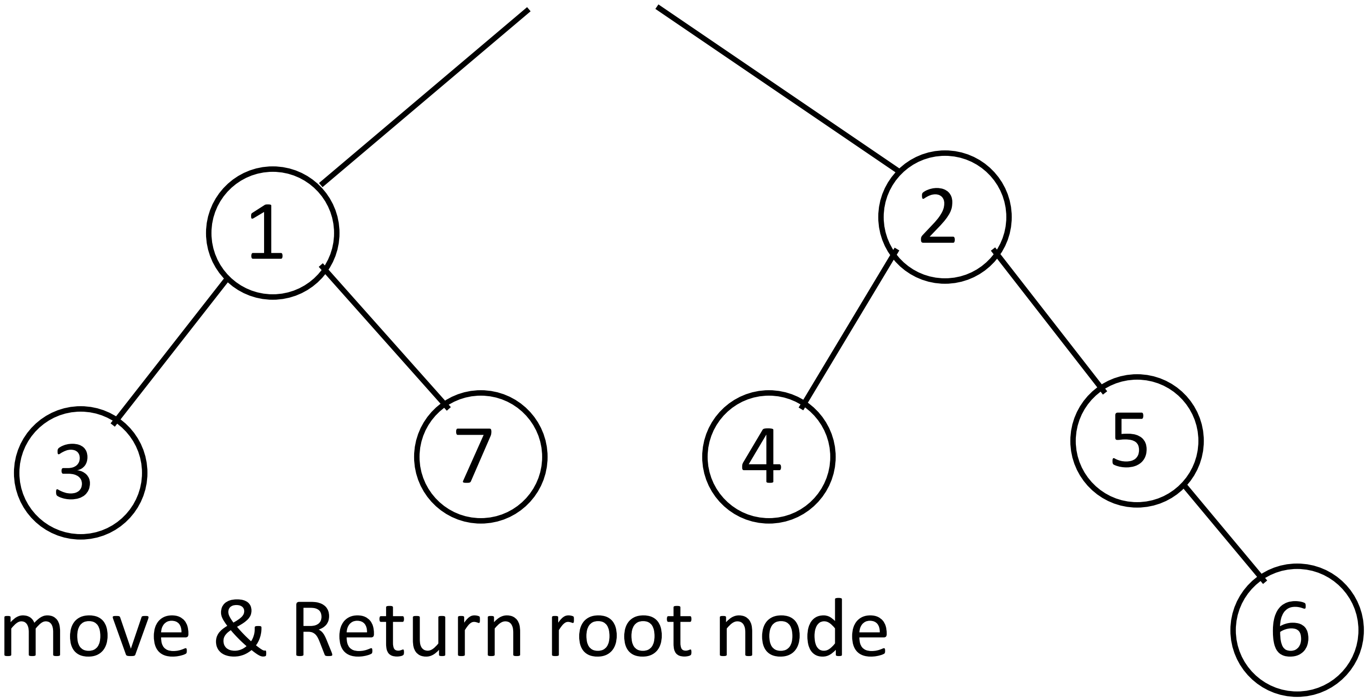


- Remove & Return root node

DeleteMin

Output:

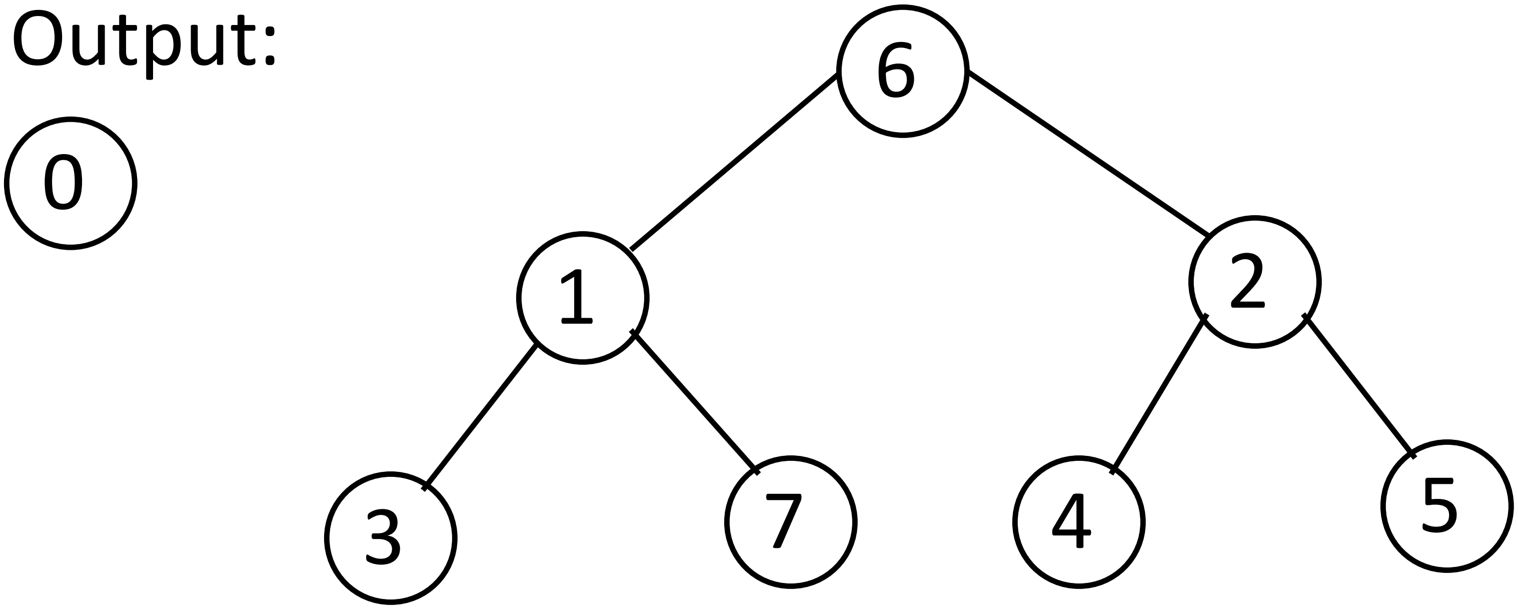




- Remove & Return root node
- Move bottom to top

DeleteMin

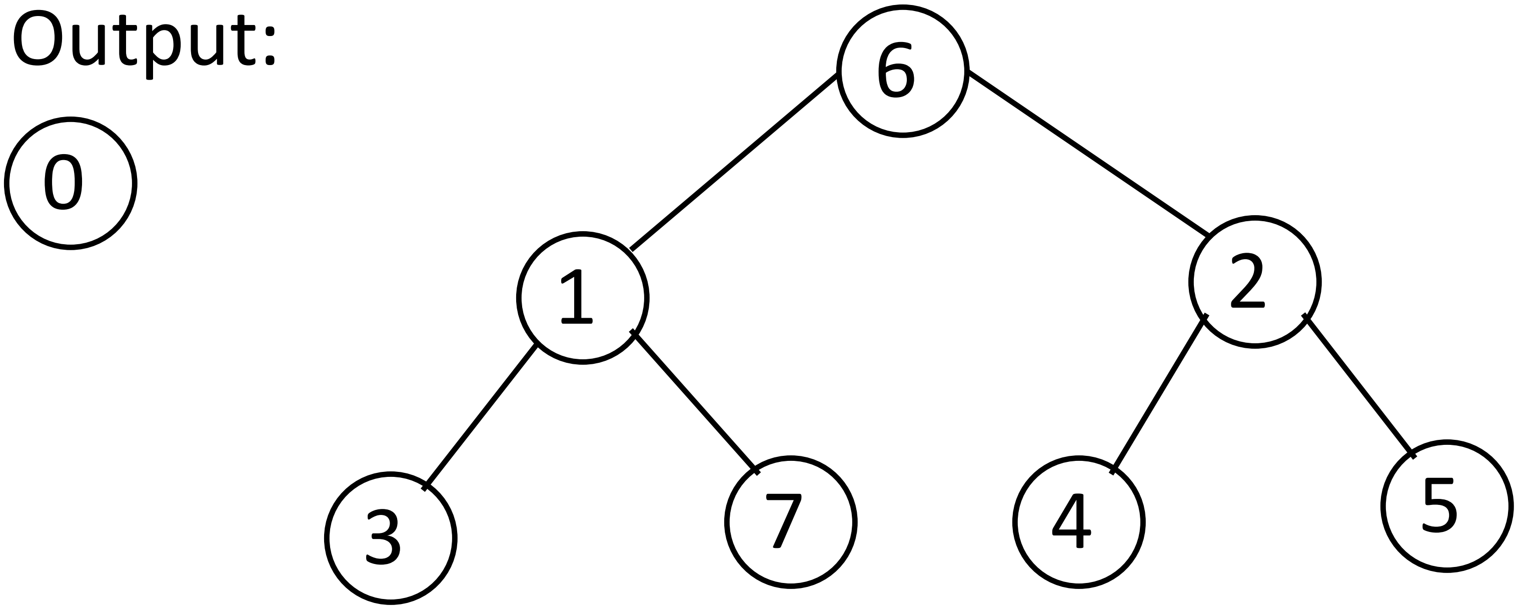
Output:



- Remove & Return root node
- Move bottom to top

DeleteMin

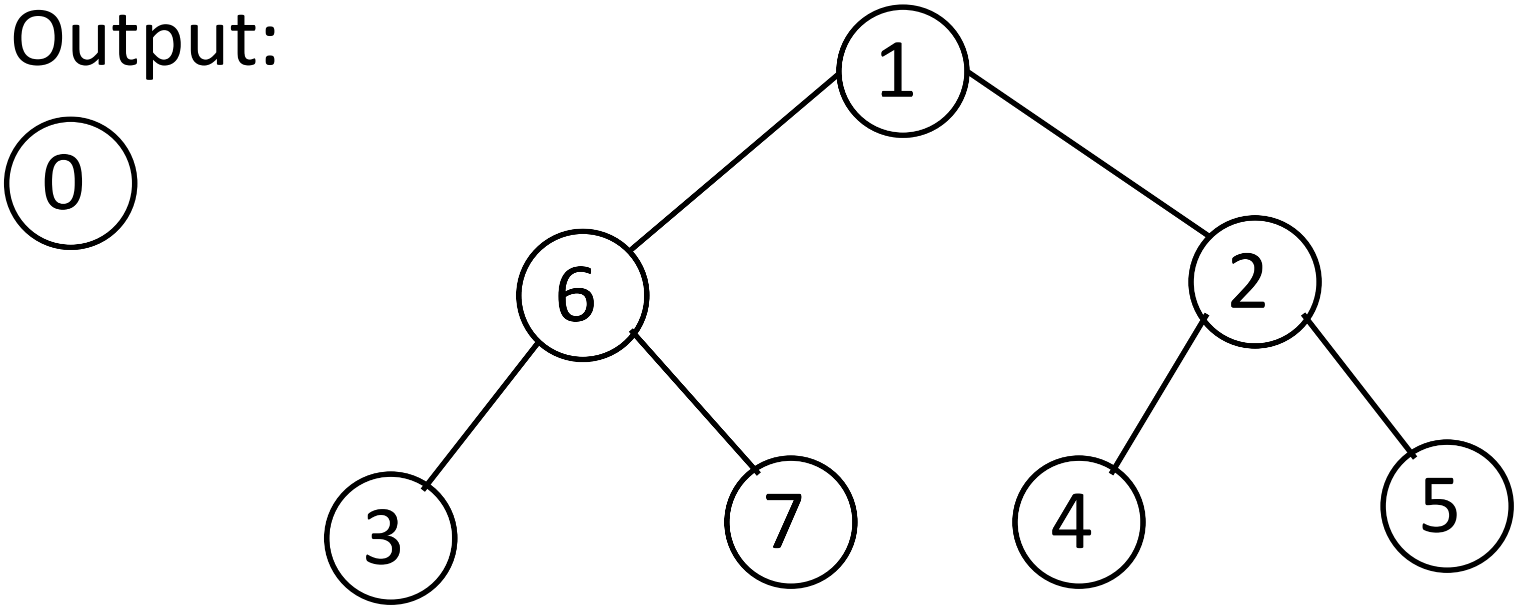
Output:



- Remove & Return root node
- Move bottom to top
- Bubble down

DeleteMin

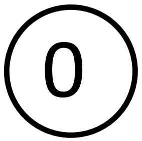
Output:

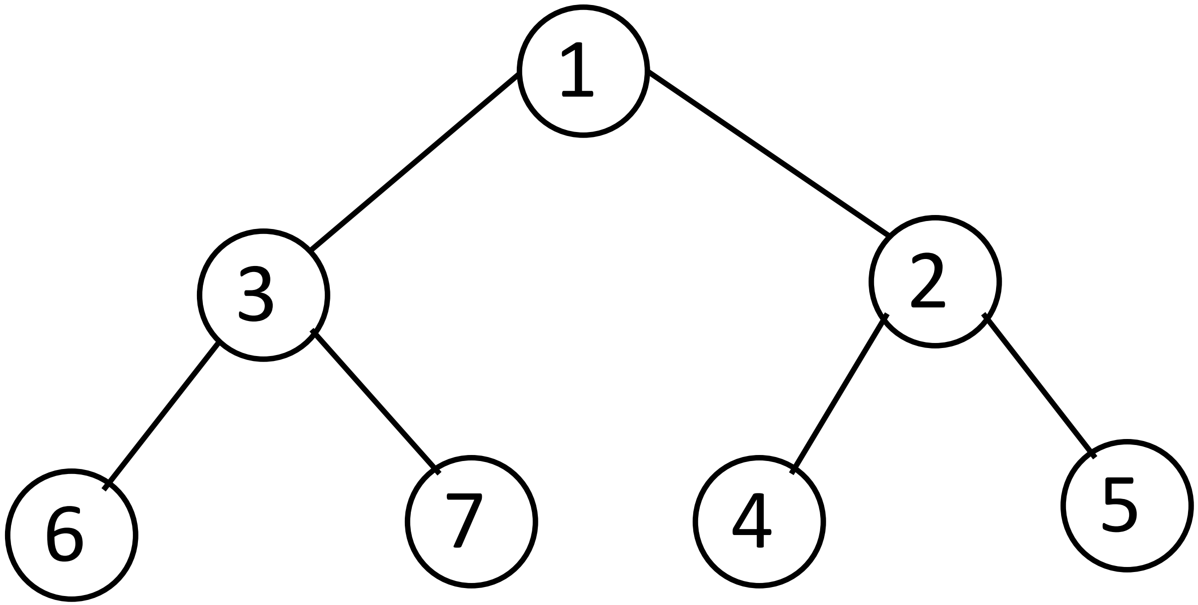


- Remove & Return root node
- Move bottom to top
- Bubble down

DeleteMin

Output:

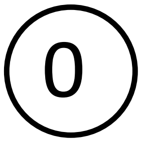


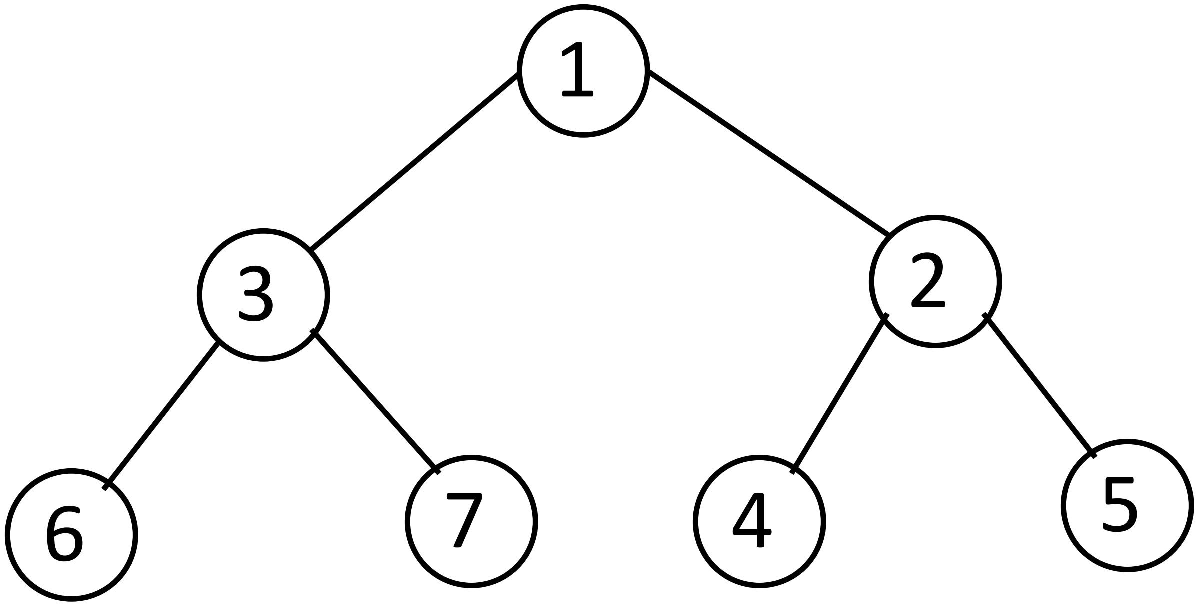


- Remove & Return root node
- Move bottom to top
- Bubble down

DeleteMin

Output:





- Remove & Return root node
- Move bottom to top
- Bubble down

Runtime O(log(n))

Summary

Runtime:

- Insert – $O(\log(n))$
- DecreaseKey – $O(\log(n))$
- DeleteMin – $O(\log(n))$

Summary

Runtime:

- Insert – O(log(n))
- DecreaseKey – O(log(n))
- DeleteMin – O(log(n))
- Dijkstra – O(log|V|(|V|+|E|))

Summary

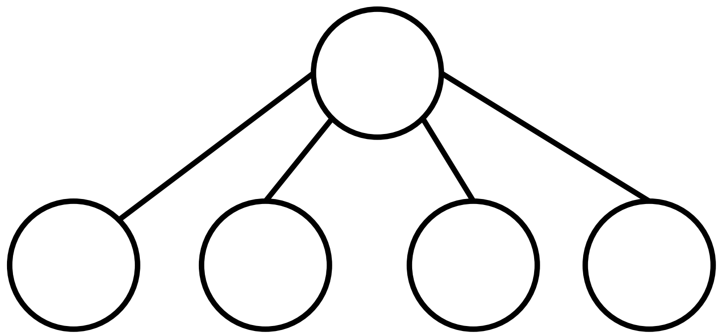
Runtime:

- Insert – $O(\log(n))$
- DecreaseKey – $O(\log(n))$
- DeleteMin – $O(\log(n))$
- Dijkstra – $O(\log|V|(|V|+|E|))$

Almost linear!

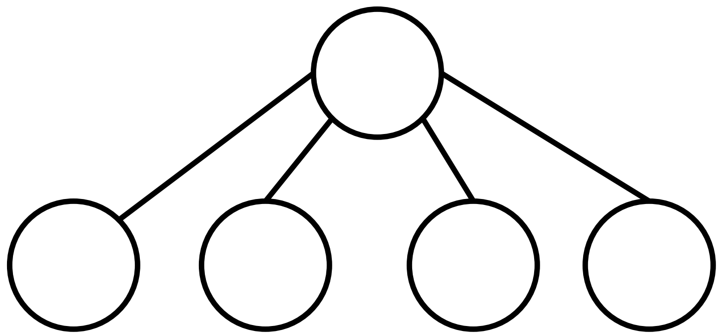
d-ary Heap

- Like binary heap, but each node has d children



d-ary Heap

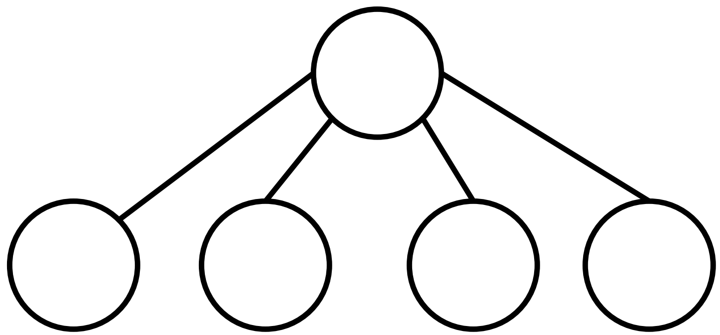
- Like binary heap, but each node has d children



- Only log(n)/log(d) levels.

d-ary Heap

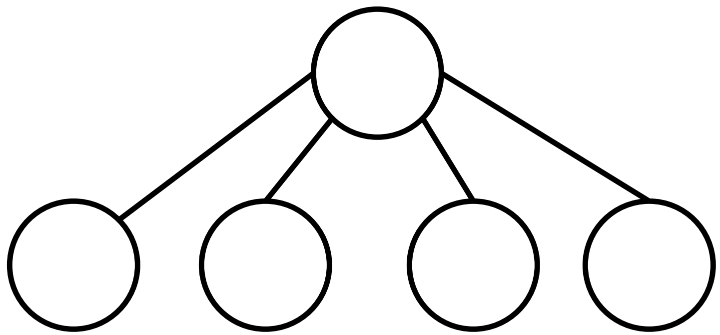
- Like binary heap, but each node has d children



- Only log(n)/log(d) levels.
- Bubble up faster!

d-ary Heap

- Like binary heap, but each node has d children



- Only log(n)/log(d) levels.
- Bubble up faster!
- Bubble down slower – need to compare to more children.

Summary

Runtime:

- Insert – O(log(n)/log(d))
- DecreaseKey – O(log(n)/log(d))
- DeleteMin – O(dlog(n)/log(d))

Summary

Runtime:

- Insert – $O(\log(n)/\log(d))$
- DecreaseKey – $O(\log(n)/\log(d))$
- DeleteMin – $O(d\log(n)/\log(d))$
- Dijkstra – $O(\log|V|(d|V|+|E|)/\log(d))$

Fibonacci Heap

- Advanced data structure.
- Uses amortization.

Fibonacci Heap

- Advanced data structure.
- Uses amortization.

Runtime:

- Insert – O(1)
- DecreaseKey – O(1)
- DeleteMin – O(log(n))

Fibonacci Heap

- Advanced data structure.
- Uses amortization.

Runtime:

- Insert – $O(1)$
- DecreaseKey – $O(1)$
- DeleteMin – $O(\log(n))$
- Dijkstra – $O(|V|\log|V|+|E|)$

Summary of Priority Queues

	Insert/DecreaseKey	DeleteMin	Dijkstra
List	$O(1)$	$O(n)$	$O(\|V\|^2 + \|E\|)$
Binary Heap	$O(\log(n))$	$O(\log(n))$	$O(\log\|V\|(\|V\| + \|E\|))$
d-ary Heap	$O\left(\frac{\log(n)}{\log(d)}\right)$	$O\left(\frac{d\log(n)}{\log(d)}\right)$	$O\left(\frac{\log\|V\|}{\log(d)}(d\|V\| + \|E\|)\right)$
Fibonacci Heap	$O(1)^*$	$O(\log(n))^*$	$O(\|V\|\log\|V\| + \|E\|)$

Negative Edge Weights

- So far we have talked about the case of non-negative edge weights.

Negative Edge Weights

- So far we have talked about the case of non-negative edge weights.
  - The usual case (distance & time usually cannot be negative).
  - However, if “lengths” represent other kinds of costs, sometimes they can be negative.

Negative Edge Weights

- So far we have talked about the case of non-negative edge weights.
  - The usual case (distance & time usually cannot be negative).
  - However, if “lengths” represent other kinds of costs, sometimes they can be negative.
- Problem statement same. Find path with smallest sum of edge weights.

Question: Dijkstra for Negative Edge Weights

Does Dijkstra’s algorithm work in graphs with negative edge weights?

A) Yes

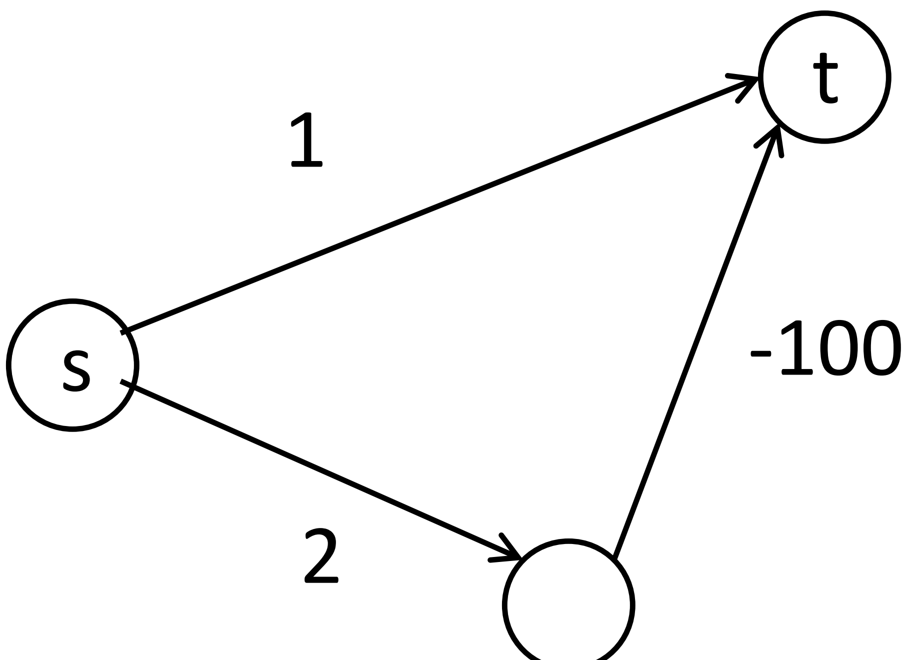
B) No

Question: Dijkstra for Negative Edge Weights

Does Dijkstra's algorithm work in graphs with negative edge weights?

A) Yes

B) No

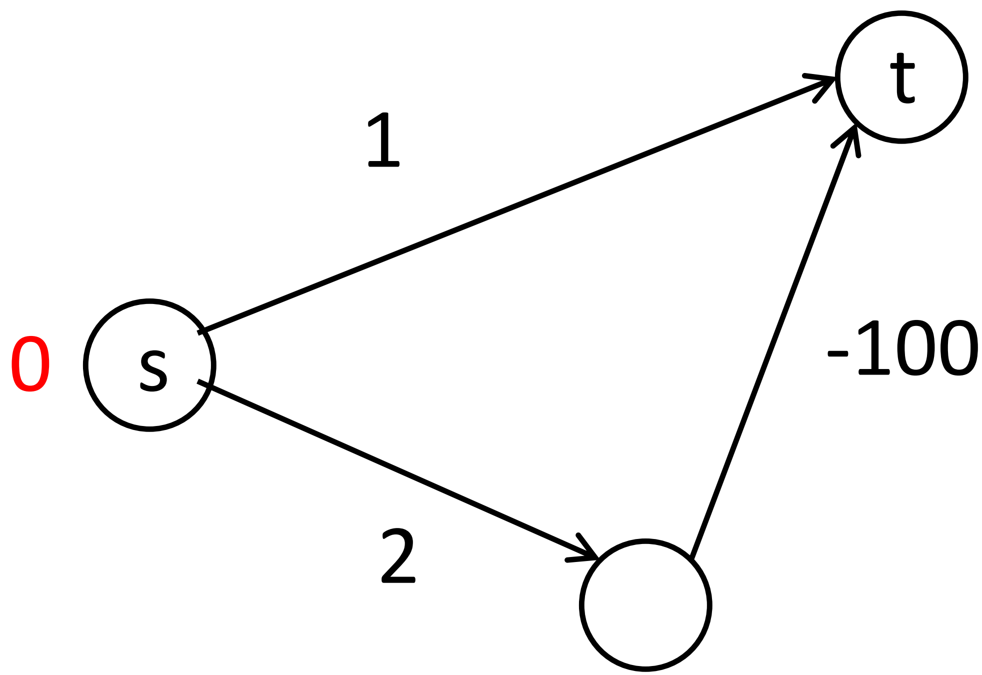


Question: Dijkstra for Negative Edge Weights

Does Dijkstra's algorithm work in graphs with negative edge weights?

A) Yes

B) No

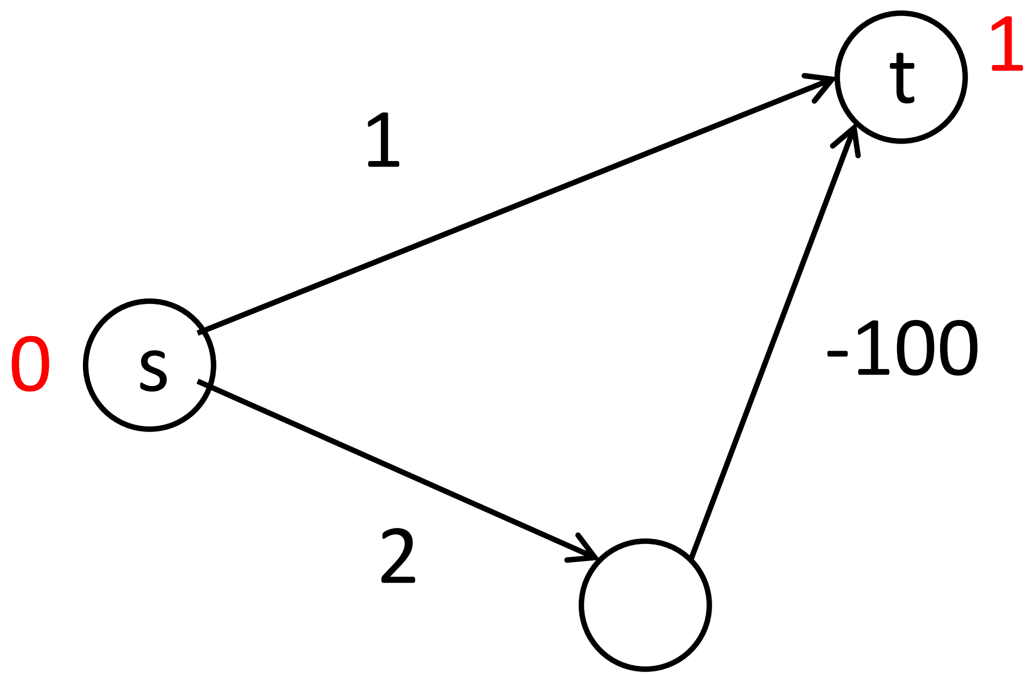


Question: Dijkstra for Negative Edge Weights

Does Dijkstra's algorithm work in graphs with negative edge weights?

A) Yes

B) No

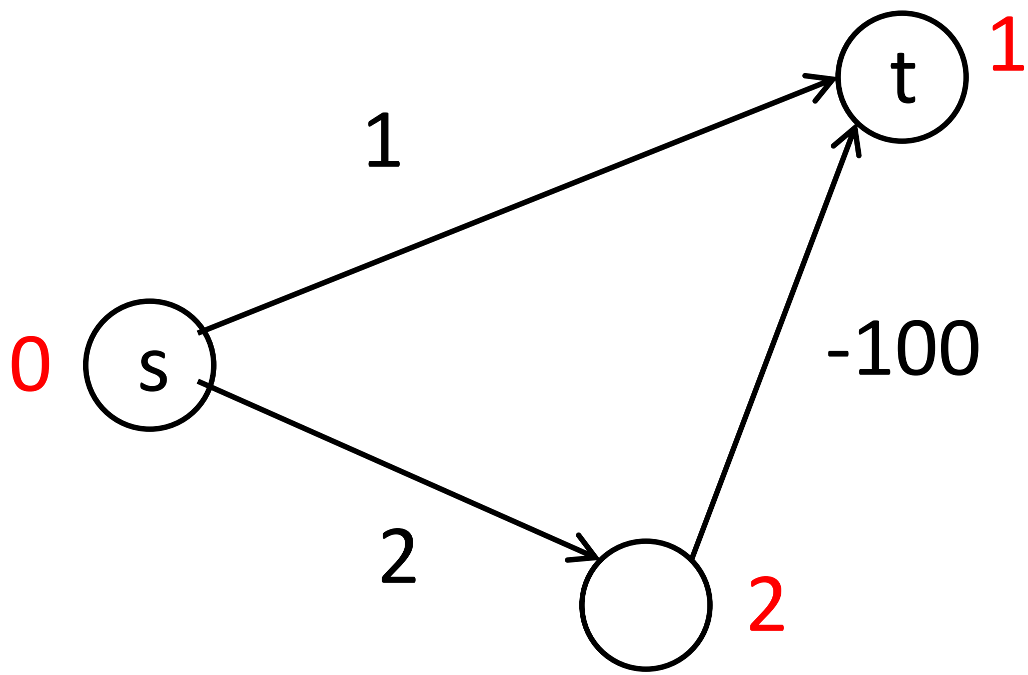


Question: Dijkstra for Negative Edge Weights

Does Dijkstra's algorithm work in graphs with negative edge weights?

A) Yes

B) No

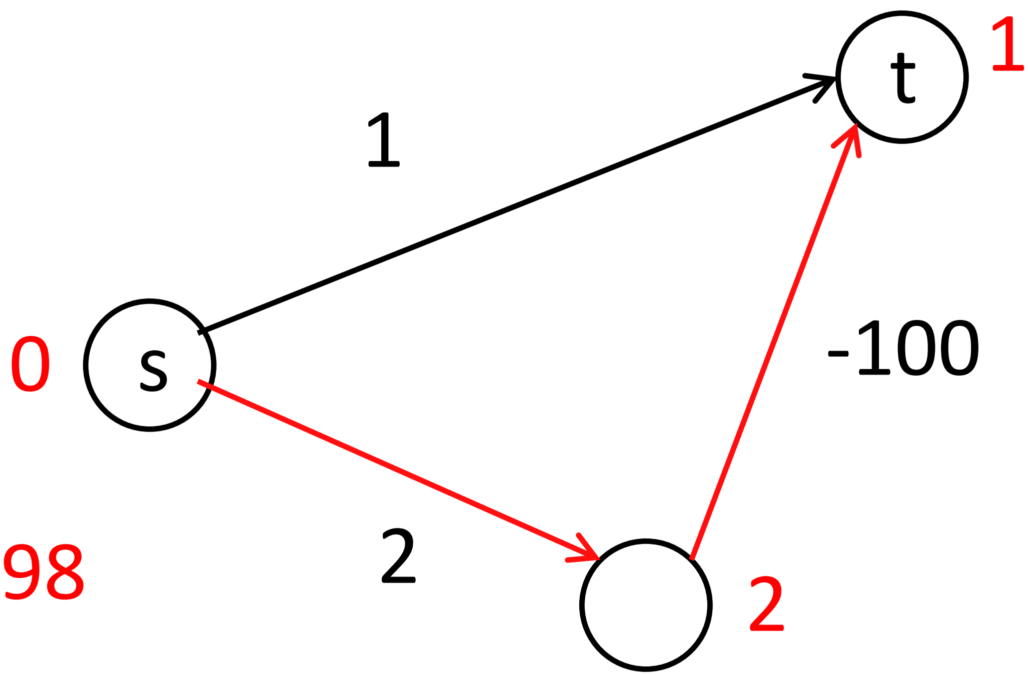


Question: Dijkstra for Negative Edge Weights

Does Dijkstra's algorithm work in graphs with negative edge weights?

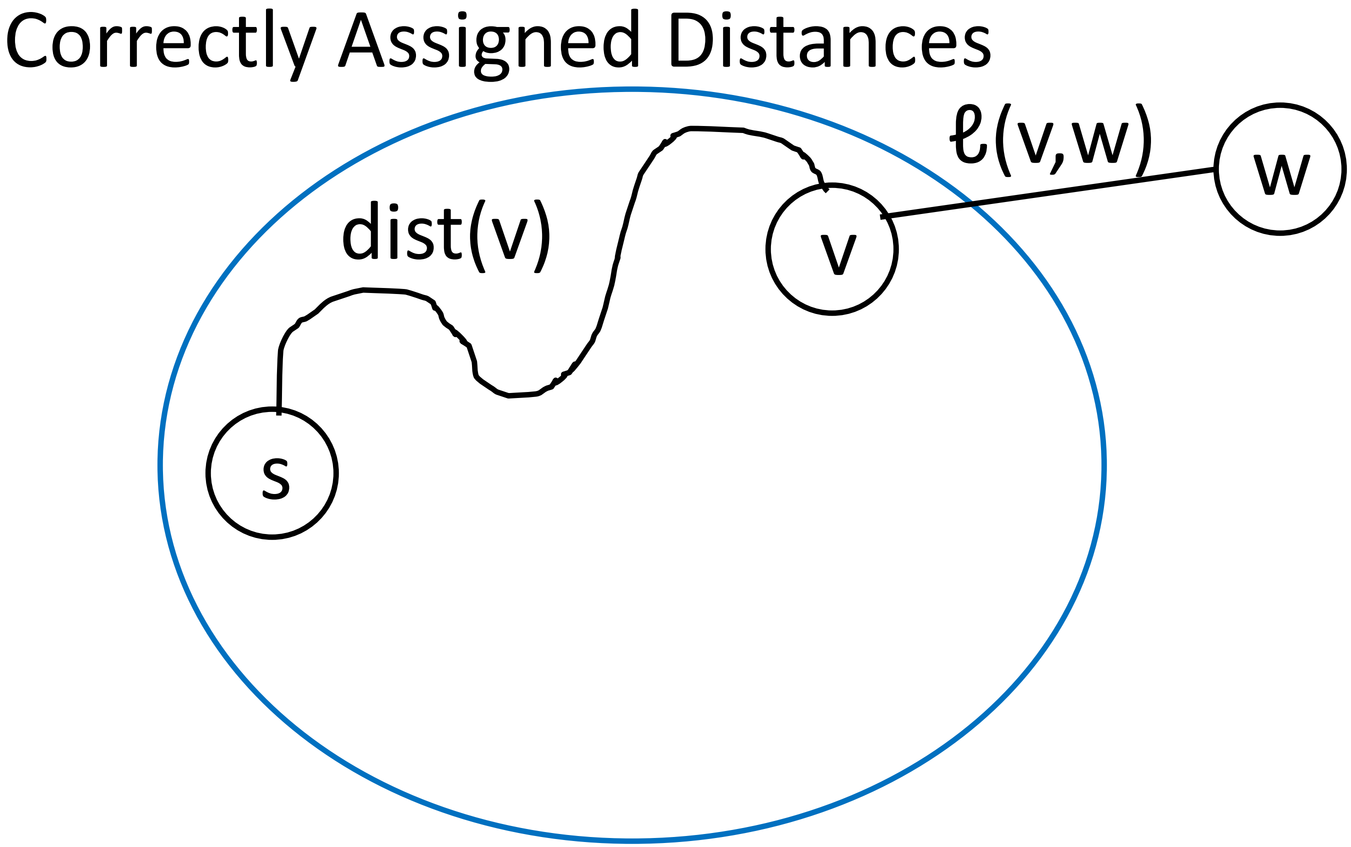
A) Yes

B) No

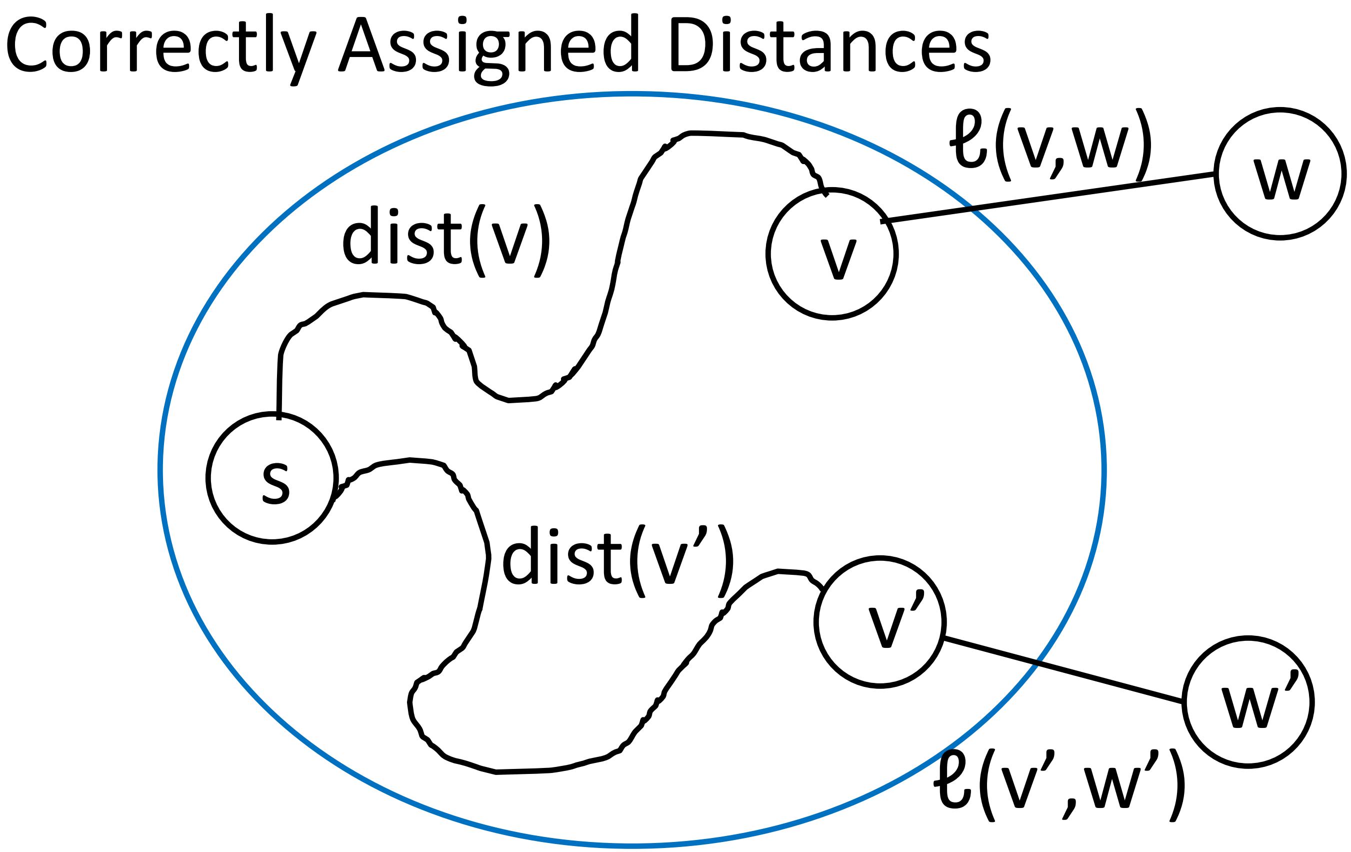


Length = -98

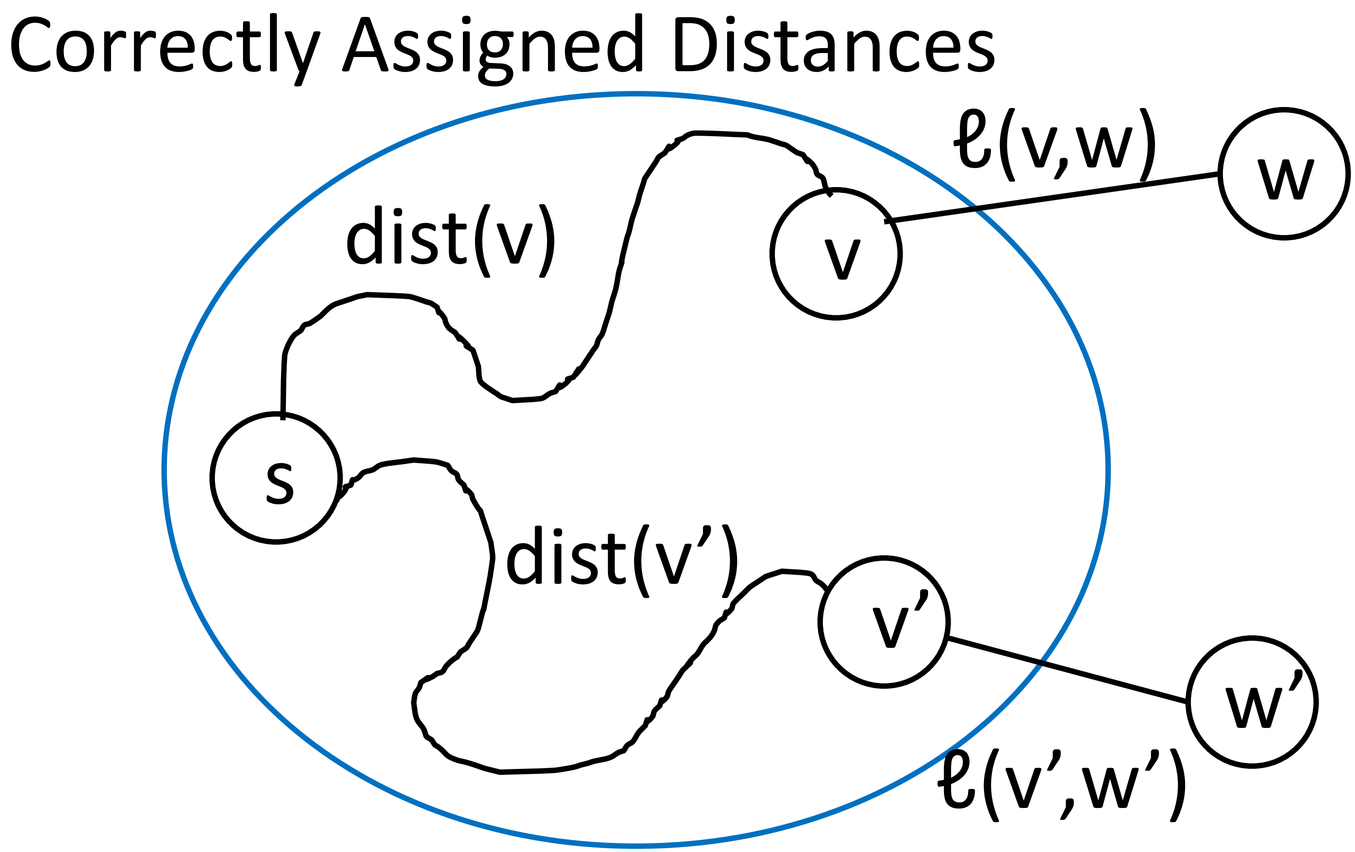
What Goes Wrong



What Goes Wrong

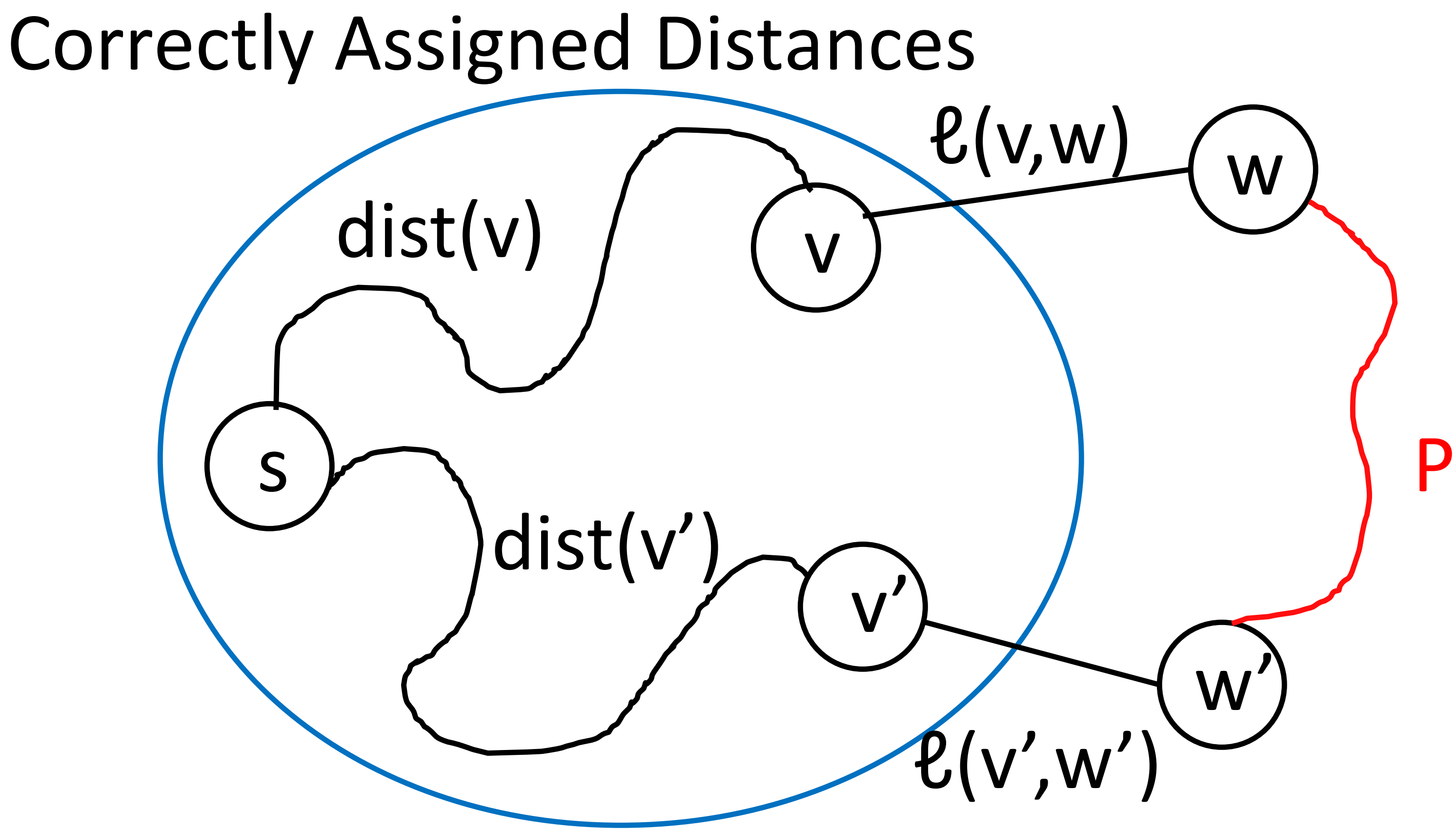


What Goes Wrong



$$\text{dist}(v) + \ell(v,w) \leq \text{dist}(v') + \ell(v',w')$$

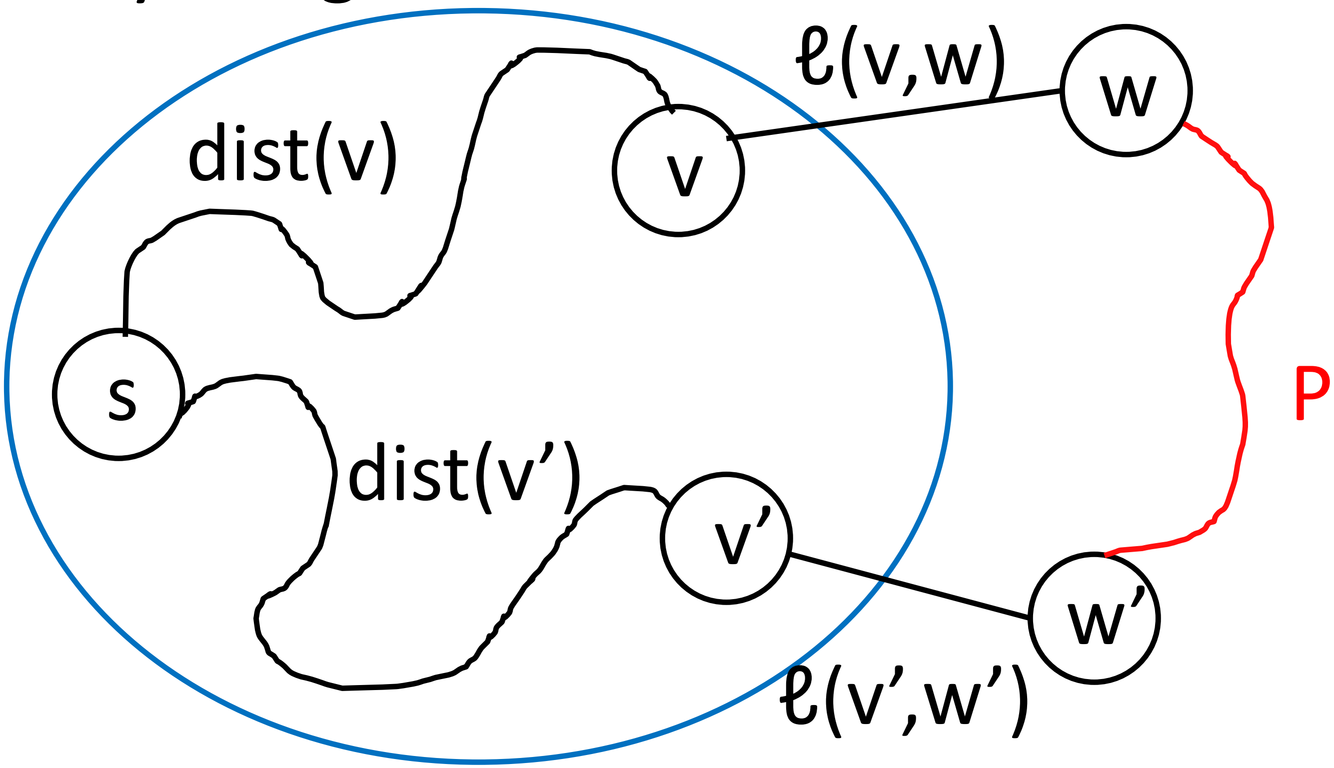
What Goes Wrong



$$\text{dist}(v) + \ell(v,w) \leq \text{dist}(v') + \ell(v',w')$$

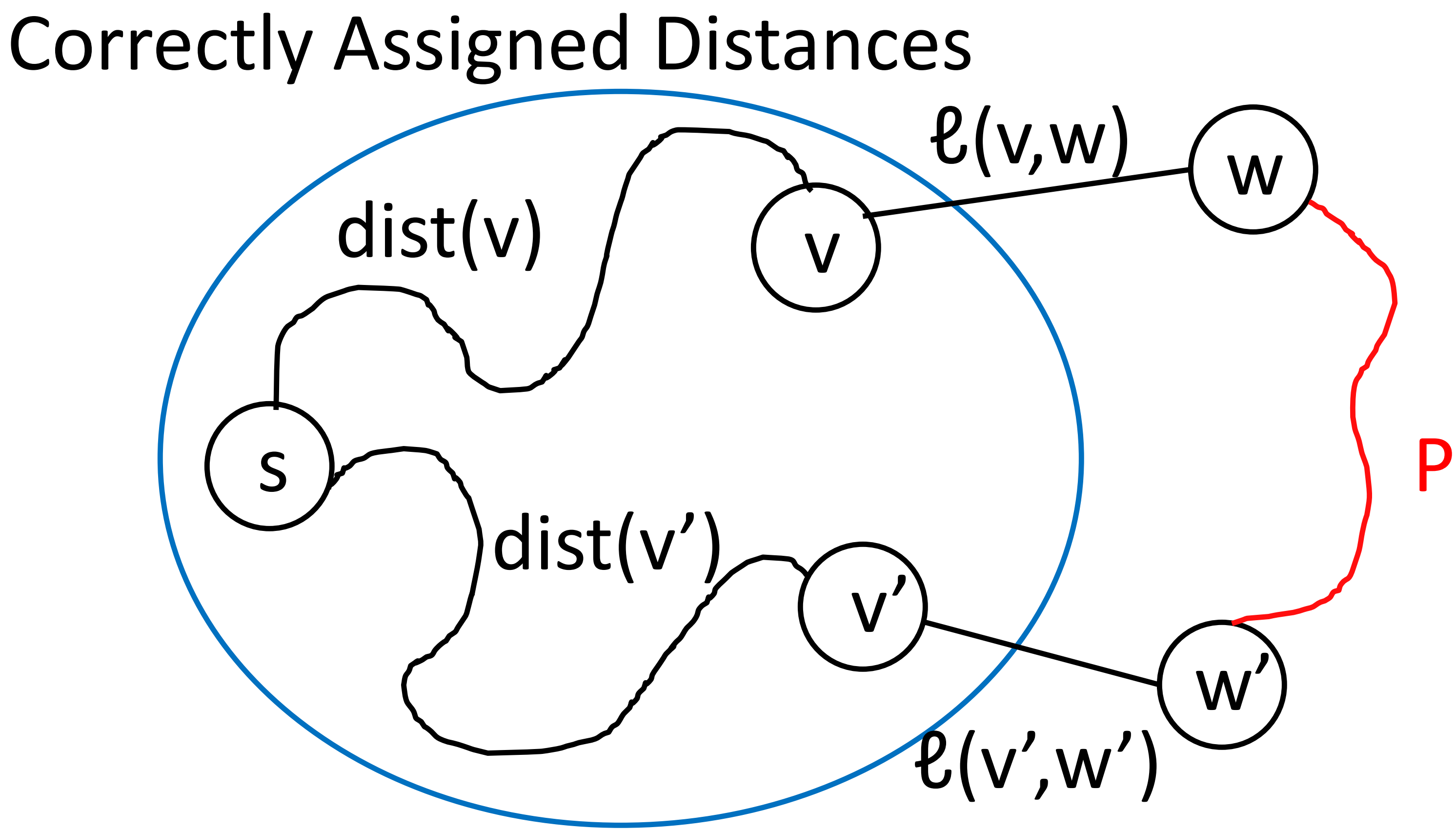
What Goes Wrong

Correctly Assigned Distances



$$\text{dist}(v) + \ell(v,w) \leq \text{dist}(v') + \ell(v',w') + \ell(P)$$

What Goes Wrong

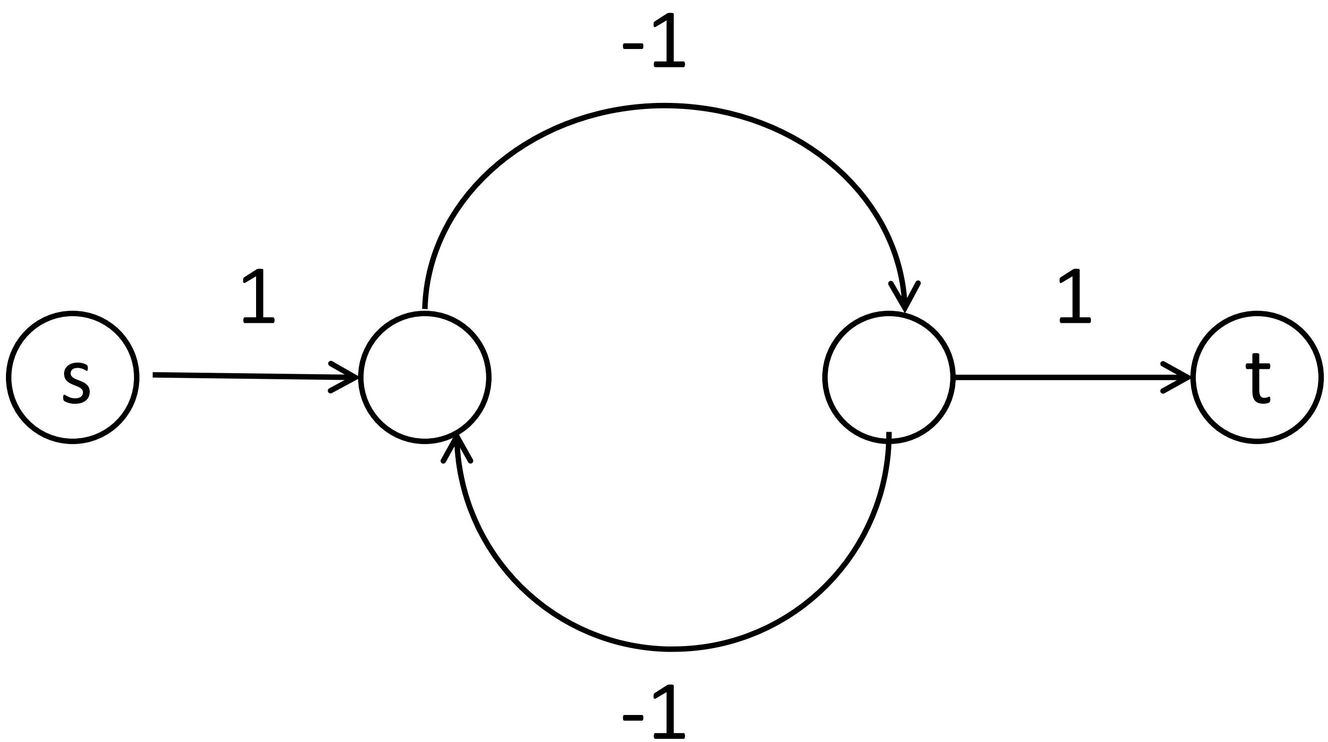


$$\text{dist}(v) + \ell(v,w) \leq \text{dist}(v') + \ell(v',w') + \ell(P)$$

Doesn’t work if $\ell(P)$ is negative!

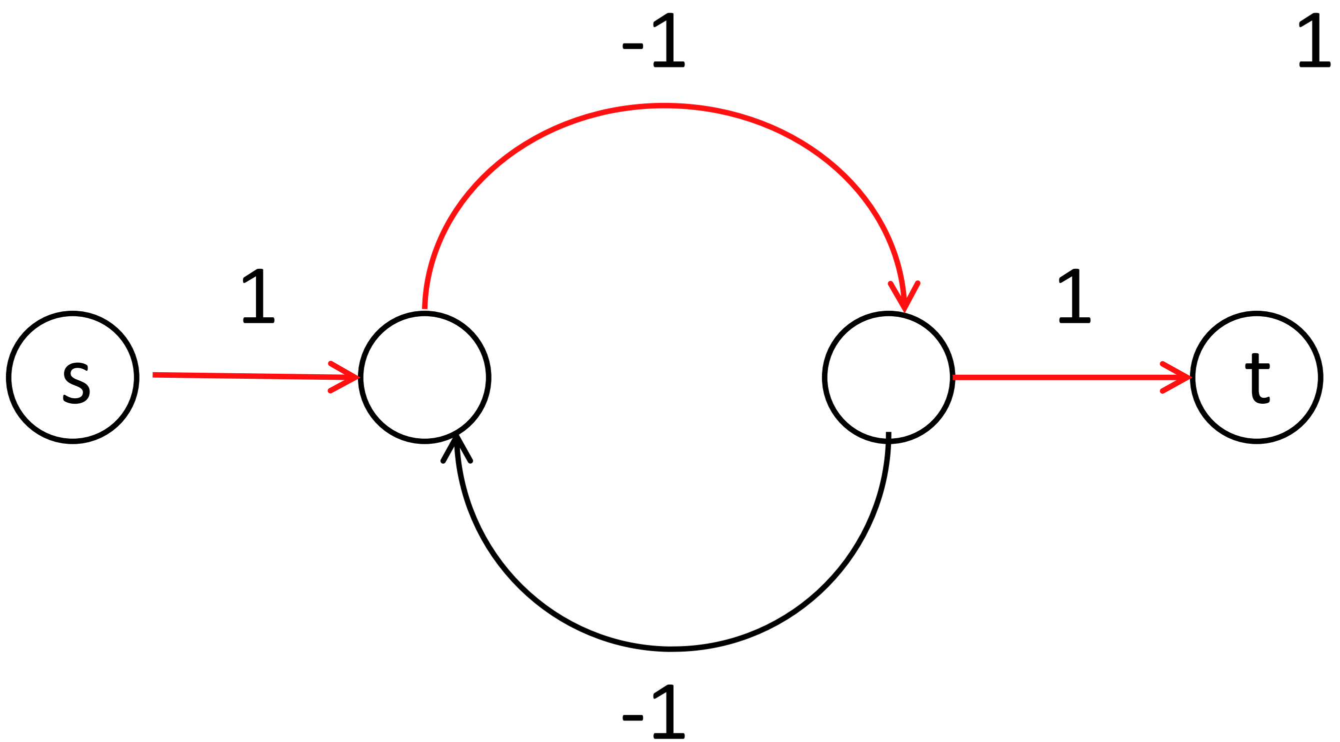
Problem

What is the shortest path length from s to t?



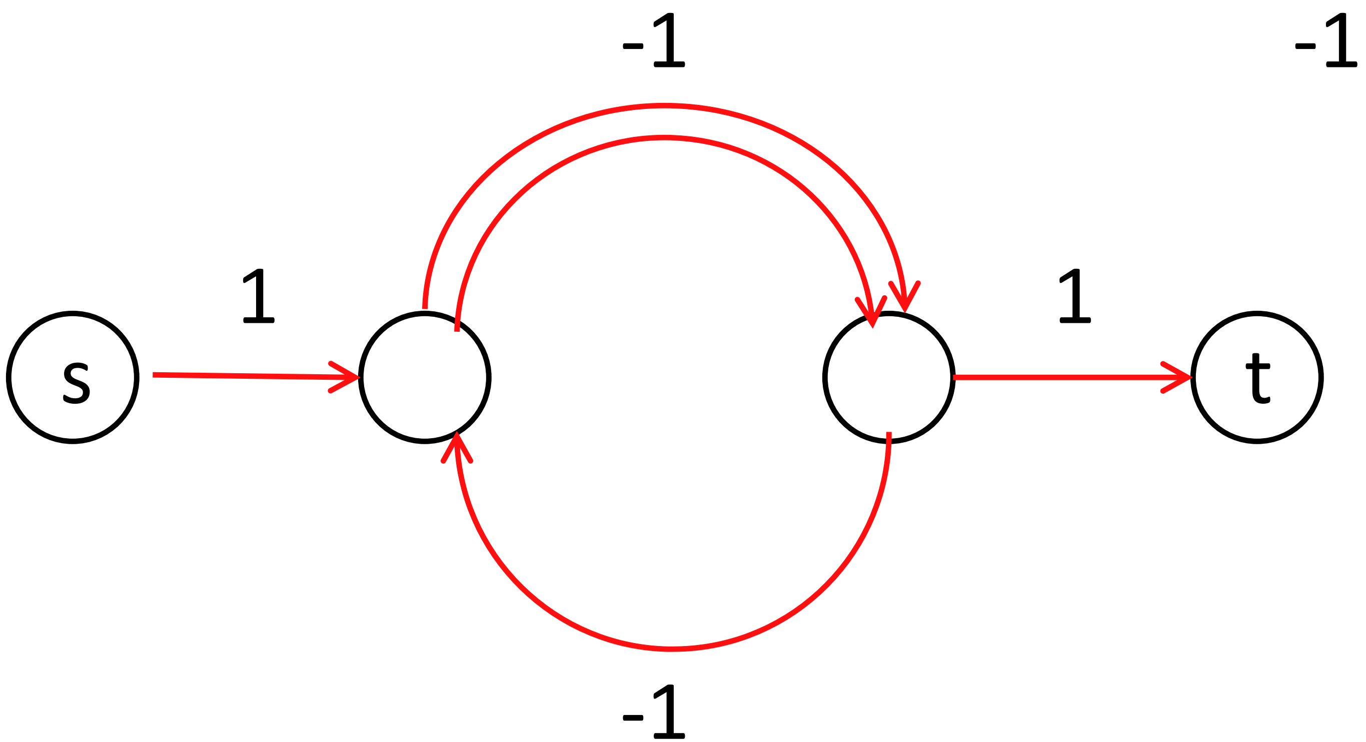
Problem

What is the shortest path length from s to t?



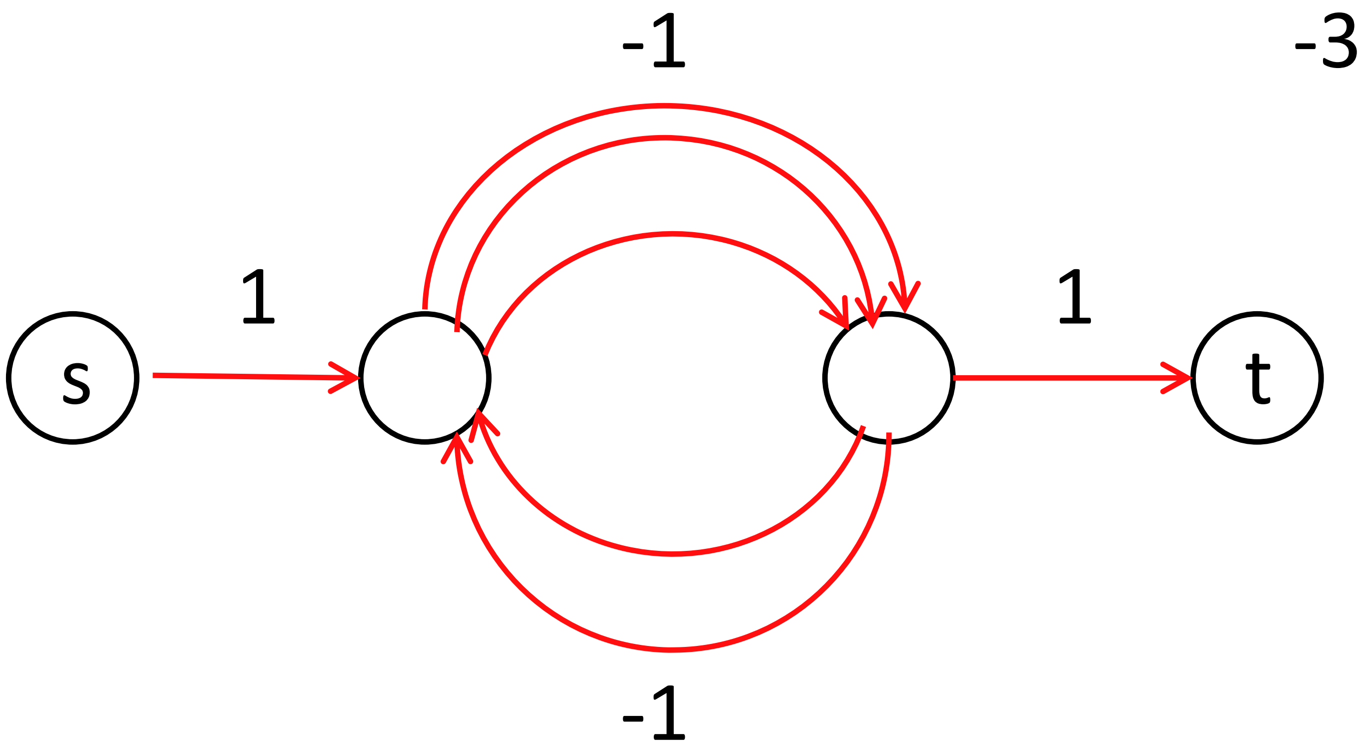
Problem

What is the shortest path length from s to t?



Problem

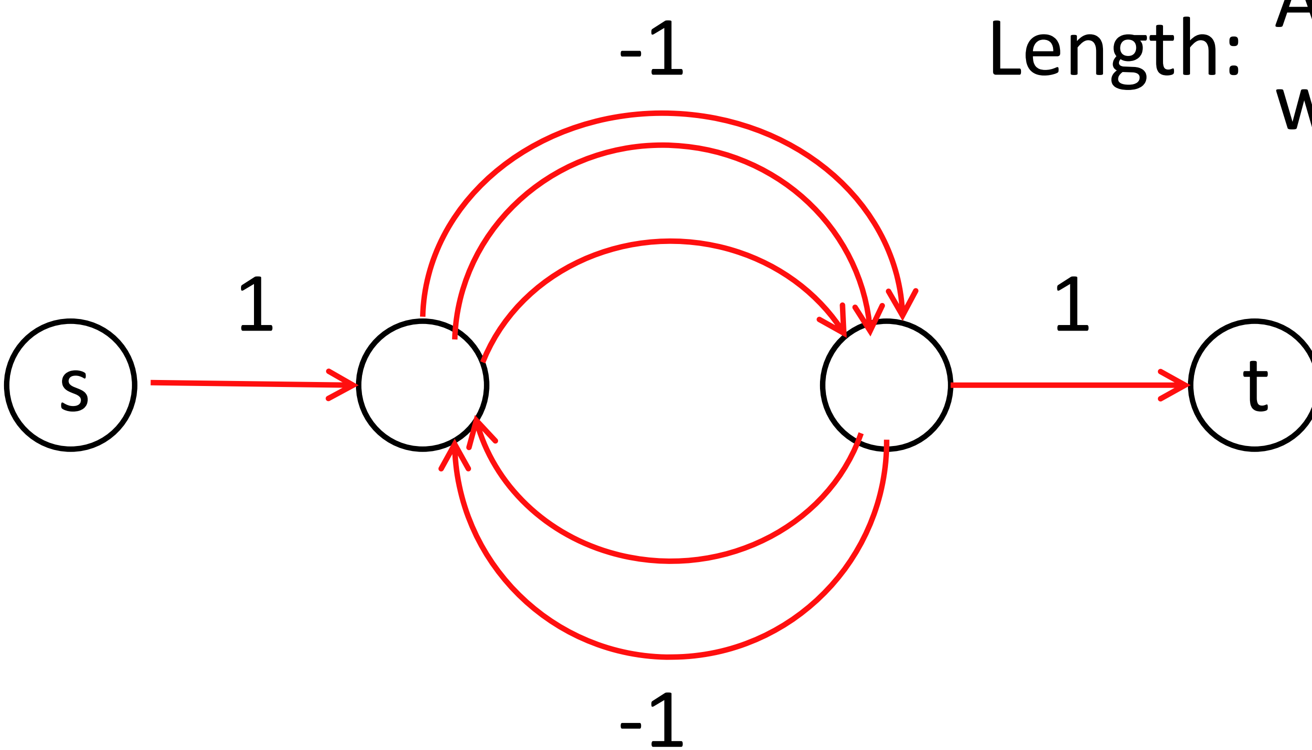
What is the shortest path length from s to t?



Problem

What is the shortest path length from s to t?

Length: As small as we like



Negative Weight Cycles

Definition: A negative weight cycle is a cycle where the total weight of edges is negative.

Negative Weight Cycles

<u>Definition:</u> A <u>negative weight cycle</u> is a cycle where the total weight of edges is negative.

- If G has a negative weight cycle, then there are probably no shortest paths.
  - Go around the cycle over and over.

Negative Weight Cycles

Definition: A negative weight cycle is a cycle where the total weight of edges is negative.

- If G has a negative weight cycle, then there are probably no shortest paths.
  - Go around the cycle over and over.
- **Note:** For undirected G, a single negative weight edge gives a negative weight cycle by going back and forth on it.

Fundamental Shortest Paths Formula

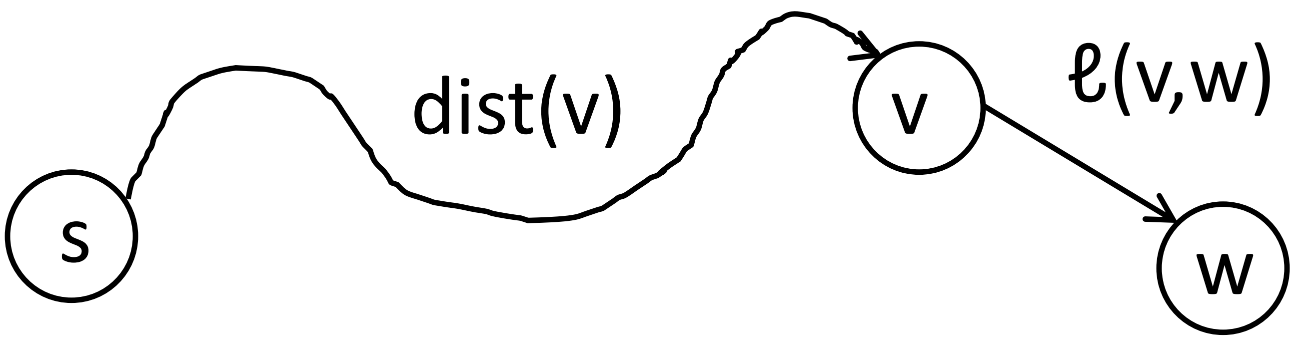
For $w \neq s$,

$$\mathrm{dist}(w) = \min_{(v,w) \in E} \mathrm{dist}(v) + \ell(v,w).$$

Fundamental Shortest Paths Formula

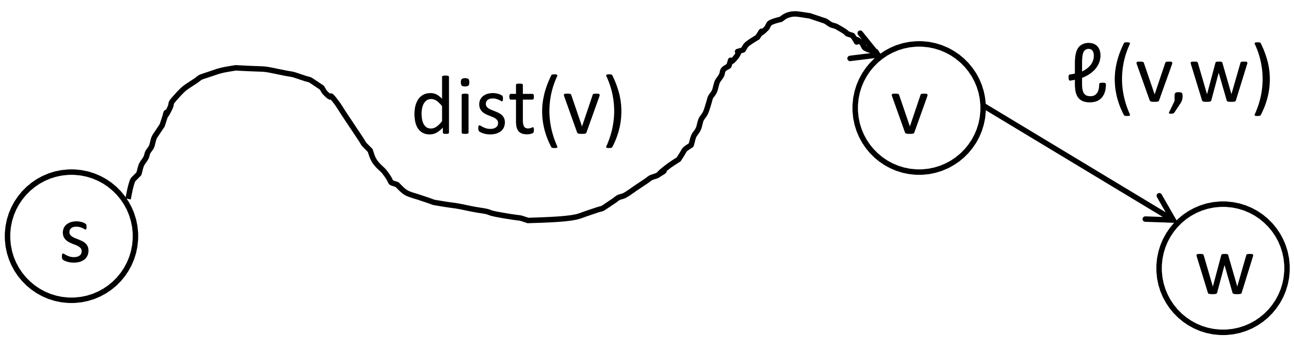
For $w \neq s$,

$$\text{dist}(w) = \min_{(v,w)\in E} \text{dist}(v) + \ell(v,w).$$



Fundamental Shortest Paths Formula

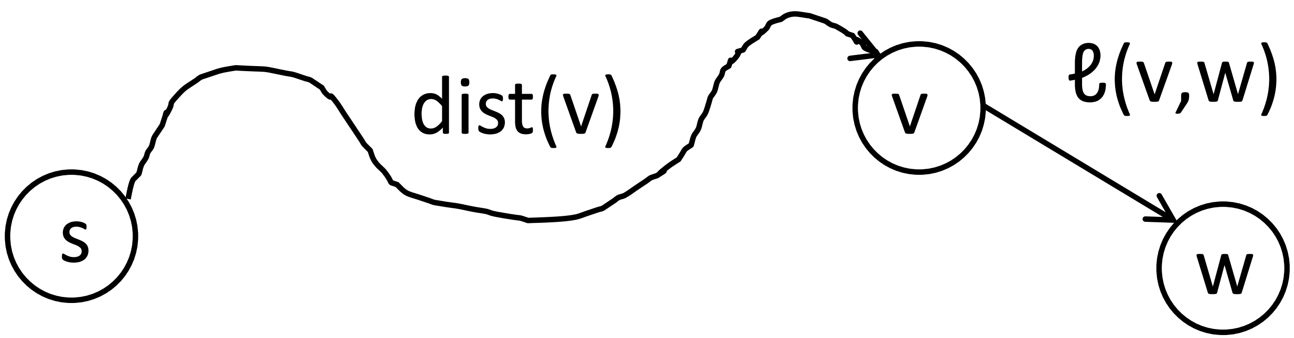
$$\text{For } w \neq s,$$
$$\mathrm{dist}(w) = \min_{(v,w)\in E} \mathrm{dist}(v) + \ell(v,w).$$



- System of equations to solve for distances.

Fundamental Shortest Paths Formula

$$\text{For } w \neq s,$$
$$\operatorname{dist}(w) = \min_{(v,w) \in E} \operatorname{dist}(v) + \ell(v,w).$$

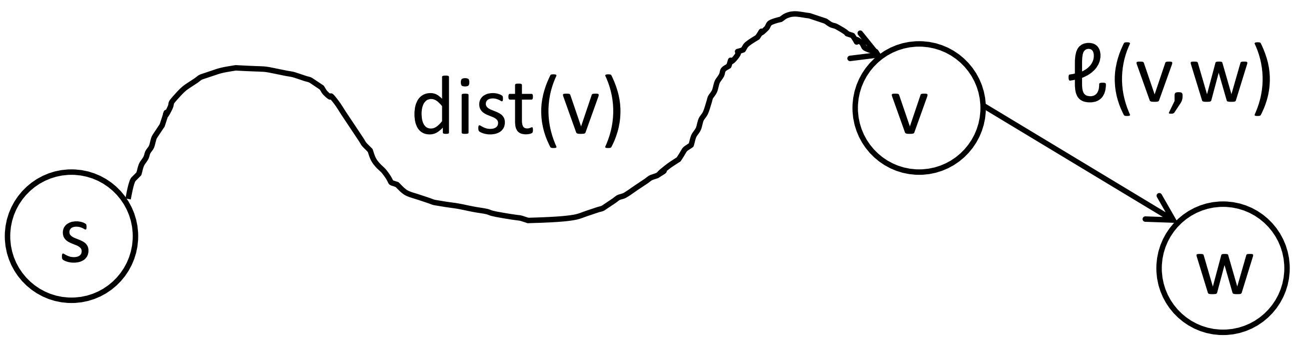


- System of equations to solve for distances.
- When $\ell \geq 0$, Dijsktra gives an order to solve in.

Fundamental Shortest Paths Formula

For $w \neq s$,

$$\operatorname{dist}(w) = \min_{(v,w) \in E} \operatorname{dist}(v) + \ell(v, w).$$



- System of equations to solve for distances.
- When $\ell \geq 0$, Dijsktra gives an order to solve in.
- With $\ell < 0$, might be no solution.

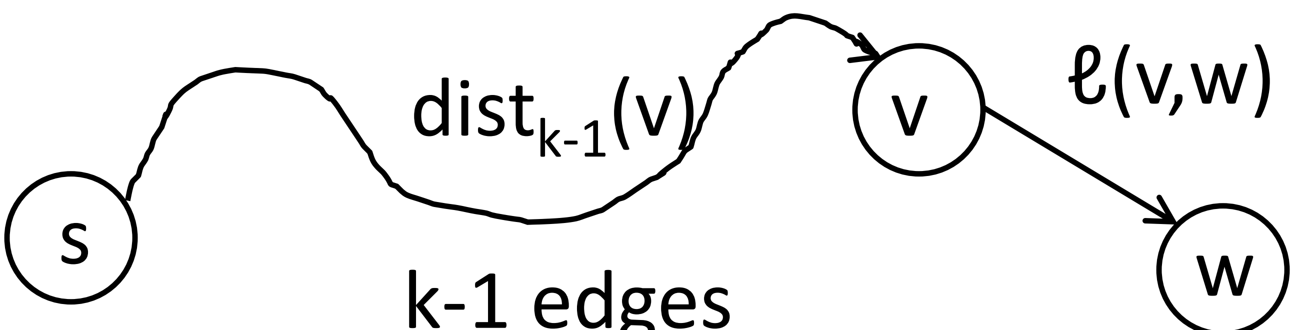
Algorithm Idea

Instead of finding shortest paths (which may not exist), find shortest paths of length at most k.

Algorithm Idea

Instead of finding shortest paths (which may not exist), find shortest paths of length at most k.

$$\text{For } w \neq s,$$
$$\text{dist}_k(w) = \min_{(v,w)\in E} \text{dist}_{k-1}(v) + \ell(v,w).$$



Algorithm

```
Bellman-Ford(G,s,ℓ)
  dist_0(v) ← ∞ for all v
    //cant reach
  dist_0(s) ← 0
  For k = 1 to n
    For w ∈ V
      dist_k(w)←min(dist_{k-1}(v)+ℓ(v,w))
    dist_k(s) ← min(dist_k(s),0)
      // s has the trivial path
```